AD-A259 293

AFIT/GCE/ENG/92D-04

Solution to a Multicriteria Aircraft Routing Problem
Utilizing Parallel Search Techniques

THESIS

James Joseph Grimm II
Captain, USAF

AFIT/GCE/ENG/92D-04

DTIC
ELECTE
JAN 07 1993
S E D

93-00110

Approved for public release; distribution unlimited

93 1 04 122

AFIT/GCE/ENG/92D-04

Solution to a Multicriteria Aircraft Routing Problem Utilizing Parallel Search Techniques

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Engineering

Accesion For	
NTIS CRA&I	X
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and / or Special
A-1	

James Joseph Grimm II, B.S.E.E.

Captain, USAF

December, 1992

Approved for public release; distribution unlimited

Acknowledgements

This research was motivated by the difficulties encountered by automated mission routing software used during Operation Desert Storm. The selection of routes is a multicriteria problem which is a complex problem. I began this research having little knowledge or understanding of the problem, but through the effort of my instructors, committee members, and others I have come to appreciate the difficulty and complexity of this problem.

I would like to acknowledge the members of my thesis committee, Dr. Andrew Terzouli, Dr. Yupo Chan, and Maj Gregg Gunsch for their help and guidance. They were willing to talk with me and answer any questions I had. I am especially indebted to my thesis advisor Dr. Gary B. Lamont, without his direction, suggestions, and sharing of his knowledge of search algorithms and parallel computers this thesis effort would not have been possible. It was under his watchful eye that this research was conducted and thesis written.

I am indebted to the other students in the Parallel Algorithms and Applications Group (PAAG), Capt Chuck Wright for his help in debugging parts of my code, and Capt Larry Merkle for the many hours we spent together in the lab. He took time out of his research to answer my many and sometimes "stupid" questions which he patiently tried to answer. Also I must thank Capt David Griffin, a fellow classmate, for his expertise with the C language and his willingness to share some of that knowledge with me. I must also thank Mr. Rick Norris, the iPSC/2 system manager, for his help and answers to my questions.

Most of all I owe great thanks to my family who has suffered with me throughout this experience at AFIT. My wife Angela and my children, Whitney, CJ, Ryan and Jason have put up with the many late hours, stopping at AFIT while out running errands for "just a few minutes" to collect data and start the next job, not having enough time for each of them, and enduring the stress I was under. I thank my wife for being willing to proof-read parts of my thesis even though

she didn't understand the content. Without my family's support, comfort, and love I could not have completed my studies or this research effort. I will be forever in their debt.

James Joseph Grimm II

Table of Contents

	Page
Acknowledgements	ii
Table of Contents	iv
List of Figures	x
List of Tables	xi
Abstract	xii
I. Problem Description	1-1
1.1 Background	1-1
1.2 Problem	1-2
1.3 Assumptions	1-4
1.4 Scope	1-5
1.5 Approach and Methodology	1-6
1.6 Materials and Equipment	1-8
1.7 Overview of the Thesis	1-9
II. Literature Review	2-1
2.1 Introduction	2-1
2.2 Parallel Processing	2-1
2.2.1 Parallel Architecture	2-1
2.2.2 Parallel Software	2-3
2.3 Search Strategies	2-7
2.3.1 General Heuristic Search Strategy	2-9
2.3.2 Parallelized Heuristic Search Strategy	2-12
2.4 Mission Routing	2-14

Page

2.4.1 Representation of the Environment 2-15

2.4.2 Calculation of Enemy Radar Field of View 2-16

2.4.3 Planning 2-17

2.5 Summary 2-19

III. Requirements and High-Level Design 3-1

3.1 Introduction 3-1

3.2 Understanding the Mission Routing Problem 3-1

3.2.1 Detailed Description 3-1

3.2.2 Requirements 3-11

3.3 Design Methodology 3-12

3.3.1 Software Engineering Principles 3-12

3.3.2 UNITY 3-14

3.4 High-Level Design 3-16

3.4.1 English Description 3-16

3.4.2 UNITY Description 3-17

3.4.3 Formal Proof of UNITY Description Correctness 3-18

3.4.4 Fixed Point Exists 3-18

3.4.5 Fixed Point Reachable 3-19

3.4.6 Mapping Schema 3-20

3.4.7 Pseudo-Code 3-20

3.5 Tasks 3-23

3.6 Summary 3-23

IV. Low-Level Design and Implementation 4-1

4.1 Introduction 4-1

4.2 Low-Level Design 4-1

4.2.1 English Description 4-1

Page

4.2.2 UNITY Description 4-1

4.2.3 Mapping Schema 4-2

4.2.4 Pseudo-Code 4-3

4.3 Data Structures 4-10

4.3.1 OPEN List 4-10

4.3.2 Route Information 4-10

4.3.3 Environment Representation 4-12

4.4 Heuristics 4-12

4.4.1 Weighting of Criteria 4-13

4.4.2 Straight-Line Distance 4-14

4.4.3 Recursive Search with Straight-Line Distance Calculation . . . 4-14

4.5 Implementation 4-16

4.5.1 Host Program 4-16

4.5.2 Control Program 4-17

4.5.3 Worker Program 4-20

4.5.4 Sequential Version 4-24

4.5.5 Problems Encountered 4-24

4.6 Summary 4-28

V. Experimental Testing, Results, and Analysis 5-1

5.1 Introduction 5-1

5.2 Metrics 5-1

5.2.1 Nodes Expanded 5-1

5.2.2 Execution Time 5-2

5.2.3 Program Efficiency 5-4

5.2.4 Speed-up 5-5

5.3 Input Data 5-6

5.3.1 Terrain 5-6

Page

5.3.2 Radar 5-6

5.3.3 Air Tasking Order 5-8

5.4 Experiments 5-8

5.4.1 Test Plan 5-9

5.4.2 Reduction of the Search Space 5-10

5.4.3 Parallel Architecture 5-11

5.5 Test Results 5-12

5.5.1 Reduction of the Search Space 5-12

5.5.2 Parallel Architecture 5-13

5.5.3 Number of Processors Used (Granularity) 5-20

5.6 Summary 5-22

VI. Conclusions and Recommendations 6-1

6.1 Introduction 6-1

6.2 Interpretations of Results 6-1

6.2.1 Reduction of the Search Space 6-1

6.2.2 Metrics 6-3

6.2.3 Parallel Architecture 6-3

6.2.4 Granularity 6-4

6.3 Further Improvements to the Software 6-5

6.3.1 Error Checking 6-6

6.3.2 Location Representation 6-6

6.3.3 Load Actual Terrain and Radar Data 6-7

6.3.4 Load Aircraft Information 6-7

6.3.5 Route Representation 6-7

6.3.6 Recursion Efficiency 6-8

6.3.7 Initial Route 6-9

6.3.8 Accuracy of Calculations 6-9

Page

6.3.9 Reporting of Results. 6-9

6.4 Recommendations for Further Research 6-10

6.4.1 Local and Global Bests. 6-10

6.4.2 Centralized versus Distributed Open List. 6-11

6.4.3 Parallel Search Strategy. 6-11

6.4.4 Monitoring of the Search. 6-13

6.4.5 Reporting of Results. 6-13

6.4.6 Architectures. 6-13

6.4.7 Expert Systems. 6-14

6.5 Summary 6-15

Appendix A. Source Code Listings A-1

A.1 Parallel Version A-1

A.1.1 Host Program A-1

A.1.2 Controller Program A-4

A.1.3 Worker Program A-14

A.2 Sequential Version A-30

A.3 Support Files A-34

A.3.1 Header File. A-34

A.3.2 Make File. A-35

A.3.3 Test Angle Calculation. A-36

Appendix B. Input Data B-1

B.1 Terrain Data B-1

B.2 Radar Data B-2

B.3 Air Tasking Order Data B-7

B.3.1 AFIT-0A. B-7

B.3.2 AFIT-1A. B-7

B.3.3 AFIT-GOA. B-7

Page

Appendix C. Raw Data C-1

C.1 iPSC/2 (Mission AFIT-1A) C-1

C.1.1 Bounded (Depth = 2). C-1

C.1.2 Bounded (Depth = 3). C-4

C.1.3 Bounded (Depth = 4). C-8

C.2 iPSC/2 (Mission AFIT-GOA) C-11

C.2.1 Recursion Only (Depth = 3). C-11

C.2.2 Bounded (Depth = 2). C-16

C.2.3 Bounded (Depth = 3). C-18

C.2.4 Bounded (Depth = 4). C-25

C.3 iPSC/860 (Mission AFIT-1A) C-30

C.3.1 Bounded (Depth = 3). C-30

C.3.2 Bounded (Depth = 4). C-34

C.4 iPSC/860 (Mission AFIT-GOA). C-37

C.4.1 Bounded With Angle = 60.0 (Depth = 3). C-37

C.4.2 Bounded With Angle = 60.0 (Depth = 4). C-44

C.4.3 Bounded With Angle = 59.0 (Depth = 3). C-51

C.4.4 Bounded With Angle = 59.0 (Depth = 4). C-54

Appendix D. Angles Between Directional Vectors D-1

D.1 Problem Analysis D-1

D.2 Execution Results D-2

D.2.1 Insertions Into the Open List. D-2

D.2.2 Accuracy of Angle Calculations. D-6

Bibliography BIB-1

Vita VITA-1

List of Figures

Figure		Page
1.1.	Overview of the Mission Routing Software	1-7
2.1.	Hypercube Interconnections	2-4
2.2.	Exhaustive Search	2-9
2.3.	Heuristic Search	2-10
2.4.	Example of a Mission	2-15
3.1.	Digitized Bias	3-5
3.2.	Terrain Representation Using Polygons	3-6
3.3.	Example of a Depth First Search Strategy	3-8
3.4.	Example of a Breadth First Search Strategy	3-9
3.5.	Example of a Depth First, Branch and Bound, Search Strategy	3-10
3.6.	Overview of the Entire Mission Routing Package	3-16
4.1.	Structure Chart for the Host Program	4-4
4.2.	Structure Chart for the Control Program	4-5
4.3.	Structure Chart for the Worker Program	4-7
5.1.	The Terrain With Radar Coverage Used	5-7
5.2.	Timing Effects of Implementing a Bounding Strategy	5-14
5.3.	Efficiency Effects of Implementing a Bounding Strategy	5-14
5.4.	Timing Comparison of the iPSC/2 and iPSC/860	5-16
5.5.	Effect of Changing the Maximum Angle Allowed	5-20
5.6.	Speed-up Comparison of the iPSC/2 and iPSC/860	5-21
5.7.	Timing Effects of Changing the Recursion Depth	5-21
5.8.	Efficiency Effects of Changing the Recursion Depth	5-23
D.1.	Examples of Directional Vectors	D-2

List of Tables

Table		Page
3.1.	Prioritization of Research Tasks	3-23
4.1.	Memory Requirements For Data Types	4-11
5.1.	Characteristics of the iPSC/2 and iPSC/860	5-11
5.2.	Bounded Algorithm with a Depth of 3 on the iPSC/2	5-12
5.3.	Recursion Only Algorithm with a Depth of 3 on the iPSC/2	5-13
5.4.	Work Performed by the Bounded and Un-bounded Algorithms	5-13
5.5.	Bounded Algorithm with a Depth of 4 on the iPSC/2	5-15
5.6.	Bounded Algorithm with a Depth of 3 on the iPSC/860	5-15
5.7.	Bounded Algorithm with a Depth of 4 on the iPSC/860	5-15
5.8.	Expansion on the iPSC/2 and iPSC/860	5-16
5.9.	Possible Angles	5-17
5.10.	Angle = 59.0 & Depth = 3 on the iPSC/860	5-18
5.11.	Angle = 59.0 & Depth = 4 on the iPSC/860	5-19
5.12.	Effect of Changing the Maximum Angle Allowed	5-19
6.1.	Parallel Computer Characteristics	6-14
D.1.	All Possible Angles	D-8

AFIT/GCE/ENG/92D-04

Abstract

There are a number of factors a pilot must weigh when selecting routes, such as threats, fuel, time on target, target locations, distance flown, and refueling points. This multicriteria auto-routing problem approach is a time consuming task. This thesis presents the software engineering synthesis of an automated software tool, based on a parallelized search algorithm, to determine mission routes. This computational investigation studied various areas of the mission routing problem and their impact on the execution time. In conjunction with execution time, the efficient usage of the supercomputer system was also examined.

A centralized open list is used with one processor running the open list management program while the rest of the supercomputer's processors run the program which performs the expansion of part al routes. This decomposition results in a dynamically load balanced system. It is important to match the granularity of the programs to the parallel architecture to ensure maximum utilization of the supercomputer and to minimize execution time. This phenomenon is examined in this research.

A number of search parameters are changed to study their impact on the overall execution time. The use of a branch and bound technique to reduce the search space and its impact on the execution time was studied. Other parameters examined were the size of the supercomputer and granularity of the algorithm. Each of these areas are discussed in detail as well as the applicability to real-time processing using parallel supercomputers. Tests were run on both an iPSC/2 and an iPSC/860 to determine the effects of the architecture upon the execution time.

Solution to a Multicriteria Aircraft Routing Problem
Utilizing Parallel Search Techniques

I. Problem Description

1.1 Background

Computers are entering into nearly every part of our lives, and as they do our reliance on them is increasing. Within the Department of Defense, as well as society, computers are used for a broad range of tasks from simple word processing to controlling sophisticated systems. The money being spent for these systems is "growing rapidly, as more Federal agencies use computers to meet the responsibilities given them by Congress" (49:1). Computer users expect them to provide timely and accurate responses.

As the state-of-the-art in computer technology expands, the ability to store large amounts of information within computers is also increasing. Computers are expected to store and manipulate increasingly more complex information, thus users are placing greater emphasis and reliance on the computer's capability to solve their problems. Users are straining the computer's ability to provide timely and accurate responses. Research is being conducted into not only increasing the computational speed of the computer, but also into techniques used for storing, retrieving, and manipulating information. "The search for increased capability is leading to machines with multiple processors and software capable of managing simultaneous computation in thousands of processors" (49:28).

"Until recently pilots still planned their missions in much the same way as during World War II — using pencils and rulers to plot courses, way points, fuel burns, and threat-evasion tactics on paper maps" (11:35). Research has been conducted into the use of computers as aides to pilots

performing mission planning (3)(9)(48). Some preliminary systems have been used by operational squadrons. These systems allow pilots to plot their mission on a computer screen and have the computer perform necessary calculations automatically, reducing the time needed for pilots to plan a mission. The first operational test, in a combat situation, for such systems occurred during Operation Desert Storm and they were "one of the keys to the crushing allied victory over Iraq" (11:35). With the viability of such systems demonstrated, the Department of Defense is paying closer attention to the development of more sophisticated mission planning systems.

The majority of mission planning systems are interactive programs where a pilot, using the computer, selects mission routes. A few systems are autonomous in that the computer performs the planning with no pilot interaction. The use of these autonomous mission routing systems, during Operation Desert Storm, did not meet the time requirements of the operational units. Planning of aircraft routes "involves an elaborate search through numerous possibilities" (17:126) which can severely task the resources of the system being used to select the routes. The operational systems could take up to 30 hours to arrive at a solution. This is not acceptable because a pilot only has a few hours from the time the mission assignment is received to the time the mission must be flown (3).

1.2 Problem

Mission planning consists of the selection of routes to (ingress) and from (egress) targets and is based primarily on the ability of the pilot to complete the mission with the least probability of detection by enemy forces. Timely identification of ingress and egress paths not only affects the accuracy of safe routes, but also the lives of friendly forces which may be threatened by the presence of the targets. If routes are not selected in a timely manner and the mission carried out soon afterwards, then the information upon which the routes were based may have changed. Thus,

what may have been a safe route at planning time may have become an extremely dangerous route to the aircraft and its crew. This research investigates a means of timely mission planning.

Mission planning, as defined for this research, is a multicriteria routing problem based on distanced traveled and the probability of detection by both active and passive radar. The Artificial Intelligence and Operations Research communities, as well as others, have conducted research into search strategies and techniques to reduce the amount of searching performed. Heuristic search is a class of search strategies which reduces the search space by using information about the domain problem. Two strategies within this class are the A* and dynamic programming search algorithms. The goal for each search strategy is to reduce the time necessary to find a solution by reducing the search space.

The A* algorithm is a specialized best-first strategy where the path which appears to lead to the best solution is explored. This is a divide-and-conquer, top-down, approach in that the A* approach divides the search space into smaller and smaller instances as the algorithm progresses (10:142). The overhead associated with the A* algorithm is management of a data structure used to store all the paths being explored.

Dynamic programming, on the other hand, is a bottom-up technique. It is based on the principle of optimality which states that an optimal solution is composed of optimal subsolutions. This principle is illustrated in the following shortest path example. " if k is a node on the shortest path from i to j, then that part of the path from i to k, and that from k to j, must also be optimal" (10:150). "Dynamic programming efficiently solves every possible subinstance in order to figure out which are in fact relevant, and only then are these combined into an optimal solution to the original instance" (10:144). A dynamic programming approach finds the optimal path between a start node and any other node once the goal is found then the path leading to that goal is constructed. The overhead associated with the dynamic programming approach is the data structure used to record the connections between any two nodes in the search space.

Research efforts are also focusing on the decomposition of search algorithms to achieve computational speed-up promised by parallel processing computer systems (22). Even though heuristic search and parallel programming have separately proven their ability to reduce the time needed to find a solution, the times are still not within acceptable limits. This research investigates the use of a parallel processing environment to select mission routes within an acceptable time. The A* search algorithm, versus dynamic programming, was selected because of its ease of understanding and implementation. Also Garmon (22) developed a parallelized A* search for the traveling salesmen problem in support of his research, so the foundation had been laid for a parallel A* search though the algorithms he used had to be modified to meet the requirements of the mission routing problem.

The automated mission routers fielded have been designed for a specific aircraft, thus each aircraft uses its own system. This research designed a general purpose mission planner using a modular approach which would allow a user to easily change aircraft characteristics and allow ease of maintenance of the software.

1.3 Assumptions

The process of selecting routes is a complicated process which must take into account many variables. Due to the time constraints imposed on the thesis research and the author's limited knowledge of operational mission planning procedures this thesis simplified the model used in the design of a mission routing system.

There are various types of missions including intercept, air-to-ground, search and rescue, and reconnaissance. No matter what type of mission is being flown there is a general location to which the aircraft needs to fly. Some missions may include more than one location to which the aircraft needs to fly, such as multiple targets and in-flight refueling points. This research was based on a route between two points, therefore it was assumed that a mission consisted of a staging base and

a single target. The maximum distance between the staging base and the target was less than the combat radius of the aircraft, therefore in-flight refueling was not a factor.

Selecting a mission route is simply finding the optimal path between the staging base and the target. What makes a route optimal is not a clear principle, though it is the route which results in the maximum likelihood of success of the mission. When planning a mission, routes are not selected based on a single criterion, but rather on multiple criteria. For this research, the criteria used was based on the total distance flown and the probability of detection by enemy radar. Aircraft configuration, fuel consumption, and aircraft speed are some variables pilots must take into account when planning routes but was neglected in this study because of time constraints. The probability of detection by enemy radar is based on the location of the radar, terrain, an aircraft's radar cross section (RCS), and employed electronic counter measures (ECM). In order for this research to remain unclassified both RCS and ECM were neglected.

The design of the mission routing system was towards a general purpose tool independent of aircraft type. Using a modular design approach, based on software engineering techniques, the design of the system separated the characteristics of the aircraft from the search algorithm. The capabilities of the aircraft, such as combat radius, maximum rates of climb and dive, and minimum turn radius, are only needed when checking the validity of moving from one location to the next, therefore only the module used to validate aircraft movement would need to be changed to reflect each aircraft's capabilities.

1.4 Scope

This research examines the feasibility of using a parallel processing environment in selecting mission routes. As with any problem there are infinitely many areas which can be investigated, but there is not enough time or resources to investigate all areas extensively. This problem is no exception, so areas were identified in which to concentrate. Other areas are left for future research

(see section 6.4). This research focuses on four main areas of the mission routing problem. The areas of concentration are:

1. representation of the threat environment,
2. decomposition of the A* search algorithm for use on a parallel processing system,
3. application of heuristics to reduce the search space and the time to find a solution, and
4. analysis of the effects of parallel computer architecture on the computational time needed to select a route.

The goal of this research is not to design a real-time, on-board the aircraft system operating in a dynamic environment, but to show the feasibility of a static mission routing system running on a parallel processing system. These types of systems are being researched; however, they are beyond the scope of this research, though the ultimate goal is such a system.

1.5 Approach and Methodology

The result of this research is the design, implementation, and evaluation of an automated mission routing system utilizing a parallel processing computer.

The design centered on a parallelized A* search algorithm to find the best route. A model of the world is first created through which the system searchs for the best route. Two elements of the real-world are modelled: the earth's terrain, and the radar detection capabilities of an enemy. This model provides the mission routing system with a scenario much like that faced by a pilot planning a mission. Figure 1.1 shows an overview of the information needed by the mission routing system.

Most current mission planning systems model the aircraft's movement in two dimensions. The systems assume the aircraft remains at a constant altitude with respect to the ground. Even though the aircraft travels in three dimensions the systems do not model the world in true three dimensions, thus rendering a model inconsistent with the real world. The mission router, for this

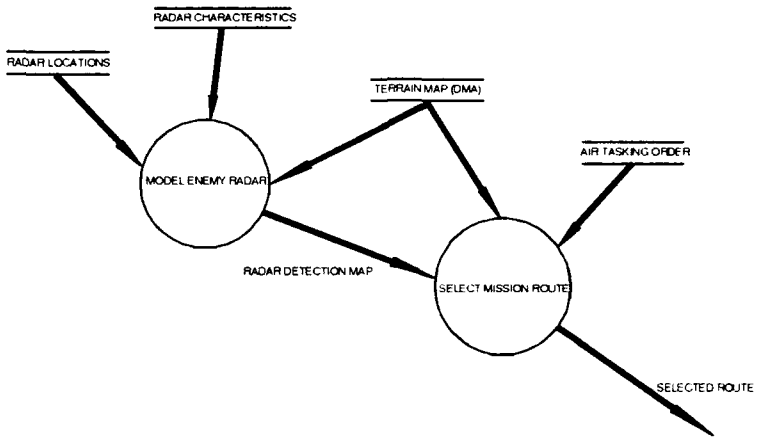

Figure 1.1. Overview of the Mission Routing Software

research, models the world in three dimensions. This is done by not only modelling the earth's terrain, but by also constructing a model of the space above that terrain.

The low-level design and implementation of the parallel A* algorithm is divided into two parts. One part manages the list containing all the partial routes being explored, the open list, and ascertains when the best route has been found. The second part finds all possible locations adjacent to a given location. As each location is examined, the cost to arrive at the location and the projected cost of going to the target, based on a heuristic, are calculated. A set of locations, beginning at the staging base and ending at the target, constitute a route. The application of software engineering principles aids in the design and implementation of the software. Designing and implementing a complicated algorithm on a sequential computer can be difficult for even an experienced person. The first step in solving the problem is to have a thorough understanding of the problem, which is presented in Chapter II. The design is a two phase process consisting of a high-level and a low-level design. A design specification language and structure charts are software engineering tools also employed to support the design of a mission routing software package. Other

software engineering techniques such as top-down design approach, software reuse, and modularity are also used.

Evaluation of the mission routing system is based on overall execution time and on the amount of search space actually examined. The programs, on each processor, record timing information along with information on the number of partial paths examined. This information is used when making comparisons. The effect of changing each of the following parameters is investigated:

- the heuristic used,
- the type of parallel computer the software was run on, and
- the number of processors used by the mission routing software.

1.6 Materials and Equipment

The Intel Corporation series of parallel supercomputers, known as the iPSC family, was the key resource for this research. Access to both an iPSC/2 and iPSC/860 computer, each with a minimum of 8 nodes, was obtained. These computer systems were available through the Air Force Institute of Technology's (AFIT) parallel processing research facility and from Wright Laboratory's Avionics Directorate's parallel computing center. A mission routing system must know the terrain in which the mission will be flown, thus the system must be provided with a model of that terrain. The Defense Mapping Agency (DMA) has produced digitized maps of most areas on the earth and some of this data was available through AFIT's computer graphics research group. Routines to access the DMA data were also available, but the time needed to incorporate these routines into the software was greater than the time allotted for this research effort. Therefore an ASCII file containing dummy terrain data was developed. A software package was needed to model the radar capabilities of an enemy. The Improved Many-on-Many (IMOM) system, developed at the Air Force Electronic Warfare Center (AFEWC), provided such modelling. Contact with the AFEWC was made and they were able to support this research by giving access to the IMOM source code. The

point of contact (POC) for the IMOM system agreed to provide the IMOM system, but because of time constraints the system was not made available in time to have it incorporated for this research effort.

1.7 Overview of the Thesis

This chapter provided a description of the mission routing problem. The scope of this research and the approach to solving the problem were also presented.

The remainder of the thesis is composed of five chapters. Chapter II is a review of the literature of not only the mission routing problem, but also parallel processing architectures and search techniques. Chapter III provides a high-level design while Chapter IV provides the low-level designs. Chapter V discusses the results and Chapter VI presents conclusions and recommendations for future work.

II. Literature Review

2.1 Introduction

A basic understanding of parallel processing systems, search strategies, and mission routing requirements is necessary to conduct this research effort. This chapter examines some of the research which has been conducted in each of these fields. An overview of parallel processing, including hardware and software, is discussed in section 2.2. Section 2.3 examines general search techniques including a parallelized heuristic search strategy. Research into mission routing is discussed in section 2.4 along with applicable software packages developed to aid in mission routing.

2.2 Parallel Processing

There are a number of hardware techniques in use to increase the computational speed of computers; such as, reduced instruction set computers (RISC), instruction pipelines, vectorization of instructions, functional units, and parallel processors.

The basic principle of parallel processing is to decompose a problem into parts and to concurrently execute as many of the parts as possible, thus decreasing the time needed to solve the whole problem (23:1829) (40:24)(39:35). Designing and implementing software for a parallel processing system is more complicated than simply taking an algorithm decomposing it into parts and running each of the parts on a separate processor. Many considerations must be taken into account, such as the manner in which the problem is decomposed, the interaction of each of the parts, scheduling the execution of each part, the communication necessary between each of the processors, and the type of architecture available.

2.2.1 Parallel Architecture. There are a number of different parallel processing environments available. The main factors which distinguish each environment are memory management, the instruction/data network, and the type of connections between each processor.

The two major approaches to parallel processing memory management are known as shared and distributed memory. A shared memory approach is one in which the system has a single large memory unit. Each processor has access to this memory and information is stored or passed through the memory. This type of approach is subjected to contention for memory access, which can result in performance loss (23:1829). Also, there must be a method to synchronize the execution of each process (24:574) and resolve any memory contention problems. A distributed memory system is one in which each processor has its own local memory, which only it can access (16:18). Information is passed between processors using messages. This approach eliminates contention for memory, though performance can still be degraded by message traffic (23:1829).

In 1966 Flynn created a simple model, based on the instruction/data network, which categorized computers into one of four categories (20:1902):

1. Single instruction stream, single data stream (SISD)
2. Single instruction stream, multiple data streams (SIMD)
3. Multiple instruction streams, single data stream (MISD)
4. Multiple instruction streams, multiple data streams (MIMD)

These categories have become the standard by which computer systems are described. The two most common types, in the parallel processing environment, are SIMD and MIMD. Each is briefly described as follows (16:63-65):

- The SIMD computer is a multiple processor system where each processor executes the same instruction, on different data, concurrently. Each of the processors must be synchronized for the instructions to execute concurrently.
- The MIMD computer is also a multiple processor system, but it can execute different instructions asynchronously. Each processor can be run independently or as a group.

Another category which is beginning to be used is single program, multiple data streams (SPMD) (35:211). This is a cross of the SIMD and MIMD models. The SPMD model, as the name implies, requires each processor to execute the same program, but the individual instructions need not be synchronized.

Another attribute used to categorize parallel computers is the type of connections between each of the processors, also referred to as nodes. Hayes and Mudge mention four types of connection networks: (1) mesh, (2) pyramid, (3) multistage network, and (4) hypercube (23:1829). Duncan also lists two additional types of connection networks, ring and tree (18:10). Many of the parallel computers have been constructed using the hypercube structure (23:1829). Hayes and Mudge describe the hypercube structure as "a generalization of the 3-dimensional cube graph to arbitrary numbers of dimension" (23:1830). Figure 2.1 shows the structure of the hypercube architecture. The n not only refers to the number of processors in the hypercube, but also the number of other processors which have a direct communication connection to any processor.

The hypercube architecture consists of a host processor, known as a system resource manager (SRM), and n processors, called nodes. The SRM acts as the supervisor of the hypercube. It provides operating system functions, editing, compilation, and cube management. The host provides the user interface to the hypercube. Each node "is a self-contained computer with a CPU, local memory ..., and an input/output (I/O) subsystem." Also each node has a set of bi-directional I/O channels, bit-serial links with direct memory access (DMA) to the local memory, connected to a nodes immediate neighbors. The nodes can communicate with other nodes, which aren't immediate neighbors through intermediate nodes which relay the messages to the destination node. (23:1831-1832)

2.2.2 Parallel Software. As stated previously the principle behind parallel processing is to decompose a problem into parts and to execute as many of the parts as possible concurrently, thus decreasing the time needed to solve the whole problem. Designing and implementing software for

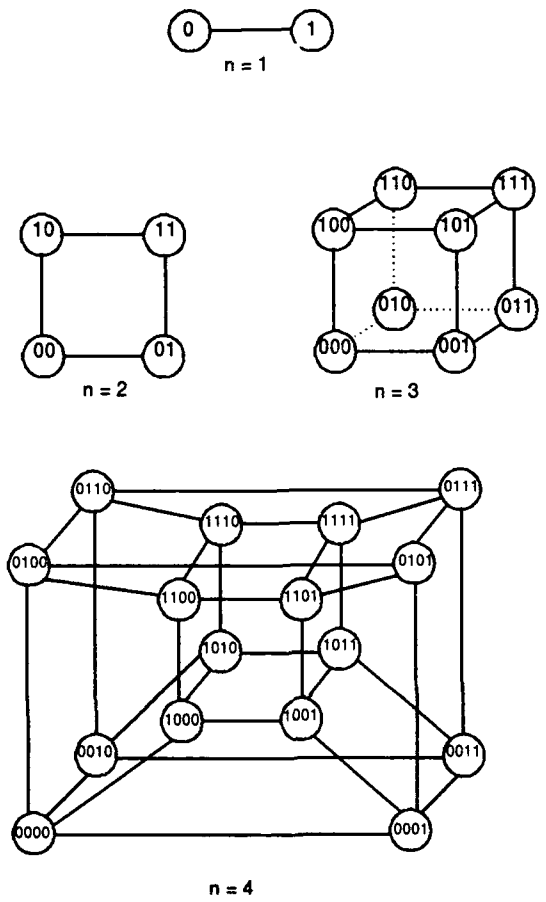

Figure 2.1. Hypercube Interconnections

a parallel computer is more complicated than simply taking an algorithm breaking it into parts and running each of the parts on a separate processor. Many considerations must be taken into account, such as the manner in which the problem is decomposed into parts, the communication between each part, the type of communication network used by the architecture, the usage of the parallel system, and the type of architecture (i.e. SIMD, MIMD, etc.) used. Each of the areas is discussed further in the following sections.

2.2.2.1 Problem Decomposition. There are two methods, or paradigms, used when distributing a problem across a parallel computing environment. Most problems are composed of two parts, the data structure and the control structure. The data structure is simply the data associated with the problem, while the control structure is the procedures which act upon the data. The two paradigms are domain decomposition and functional decomposition.

Domain decomposition deals with the data structure. In this approach, data is partitioned and then each partition is distributed to each processor (35:138). Since the control structure is unchanged, this paradigm allows a single program to be written which can be executed on any and all nodes of the parallel computer. The disadvantage of this paradigm is there may be a need for global communication. Preliminary results from one processor may need to be broadcast to the other processors so they can make decisions on the usefulness of continuing processing along their present path. Since the control structure is the same for any processor, this type of decomposition allows the use of any type of architecture. It is still important to consider the communication network and its impact, if any, on any communication.

Functional decomposition, also referred to as control decomposition, deals with the control structure. As the name suggests the control structure is partitioned into functions, or tasks, and these functions are distributed among the systems processors (35:138). This results in an approach which is analogous to an assembly line. The data is sent to the first function which operates on the data. The data proceeds along the assembly line stopping at each function (8:30). By its nature, functionally decomposed problems can not be run on SIMD or SPMD systems. By definition the SIMD and SPMD architectures require the same programs to be executed on each of the processors. The difference between the SIMD and SPMD architectures is the SIMD requires synchronous execution of the same instructions while the SPMD allows asynchronous execution without necessarily the same instructions being executed on each of the processors. Thus problems decomposed using functional decomposition can not be run on either a SIMD or a SPMD architecture.

The domain and functional decomposition paradigms are not mutually exclusive; in other words, the use of one decomposition method does not preclude the use of the other in some manner. "In general, it is not always obvious which decomposition technique is best" (35:139). The software engineer must carefully examine the problem along with the architecture to be used determine the trade-offs and select the decomposition to be employed.

2.2.2.2 Communication. Another factor which must be taken into account when designing software for parallel systems is the communication between processors. This not only refers to the communication hardware, but also to the communication of information between processors. Regardless of the decomposition paradigm used most implementations required some type of communication whether it is passing data or for controlling the execution of processors. A designer must be aware of the difference between communication time and processing time (35:136) and the limitations, or communication bandwidth, of the parallel computer. The objective of parallel computers is to reduce the overall execution time of programs. If a software system spends more time passing information around than processing the data then little actual work towards finding a solution will be accomplished and the system will run very slowly. This is analogous to the operating system concept of thrashing where the system spends more time satisfying a process' requests for memory than executing. In both cases the end result is the same, a system which runs much slower.

The term granularity is defined as the ratio of time spent by a node communicating to the time spent by the same node performing computations (39:37). Granularity gives a measure by which a designer can evaluate the software running on the parallel computer. The designer can a priori find the communication and execution times. This is not as simple as it may seem since different types of instructions have different execution times. Also there is more than one communication time. Communication time is relative to the path between sending and receiving node and the size of the message being sent. The path between nodes could be memory as in shared memory

architectures or communication channels as in distributed memory architectures. Not only does the type of architecture influence the communication times, but so does the hardware which handles the trafficking of messages.

Thus, there is much to consider when designing a system to run on a parallel computing machine. Not only does the designer need to keep the problem in mind, but also the limitations imposed by the hardware. As Lewis wrote "it becomes very clear that one of the goals of a parallel design is to develop a communication strategy that maximizes the time a processor spends computing and minimizes the time it spends communicating" (35:136).

2.2.2.3 Load Balancing. The efficient use of any system is always a problem. The usage of a computer is said to be inefficient when it is idle. This is especially true of parallel computing systems. The maximum speed-up of a parallel computer can not be realized if some processors are idle while others are busy processing data. This principle, for parallel processing, is known as load balancing (8:32).

"The goal of *load balancing* is to keep processor nodes busy and have them finish roughly at the same time" (35:137). If the designer knows the work load of each processor ahead of execution then the system can be statically balanced. However, if the work load is not known then the work must be distributed dynamically to achieve a balanced system. Static load balancing is implemented by the programmer, while dynamic load balancing can be either implemented by the operating system or the designer. In either case the software engineer developing software for parallel processing environments needs to ensure the system is properly balanced to not only ensure the software executes correctly, but also that the system is being fully utilized. (35:137)

2.3 Search Strategies

"One of the most widely used problem solving techniques is exhaustive search, which searches all possible answers and selects the best solution" (22:1-1). As the number of possible answers, or

the number of possible choices, increases so too the time it takes to find a solution increases. Barr and Feigenbaum write

> The critical problem of search is the amount of time and space necessary to find a solution. ... Examining all sequences of n moves, for example, would require operating in a search space in which the number of nodes grows exponentially with n. Such a phenomenon is called a *combinatorial explosion*. (4:27)

In this context, nodes does not indicate the processors of a parallel computing system, but decision points in the search space. Many problems exhibit this combinatorial explosion characteristic. Some of the problems which fall into this category include: game playing (i.e. chess) (4:99), theorem proving (4:155)(5:313)(14:78), transformational grammar parsing (4:260), synthesis of organic compounds (5:134), some speech recognition algorithms (4:339), planning (14:519), and the traveling salesmen problem (TSP) (15:960). This is just a small sampling of problems exhibiting combinatoric explosion, but it does the wide variety of problems with this characteristic. With a large number of choices at each decision point and a large number of decision points, it is possible that the computer can not find a solution within our life time, no matter how much the computational speed of the computer is increased. When problem spaces exhibit this phenomenon, it is important to limit the search space and not pursue paths which do not lead to a solution (46:188-189). Figure 2.2 is a representation of an exhaustive search. The triangle represents the search space and the shaded area is the space actually searched.

There are a number of approaches which attempt to reduce the number of nodes needing to be examined during the search process (4:27). The results of this type of approach are reflected in Figure 2.3, where the triangle is the search space and the shaded area the space actually searched. The best strategy is one in which the only locations of the search space examined are those on the solution path.

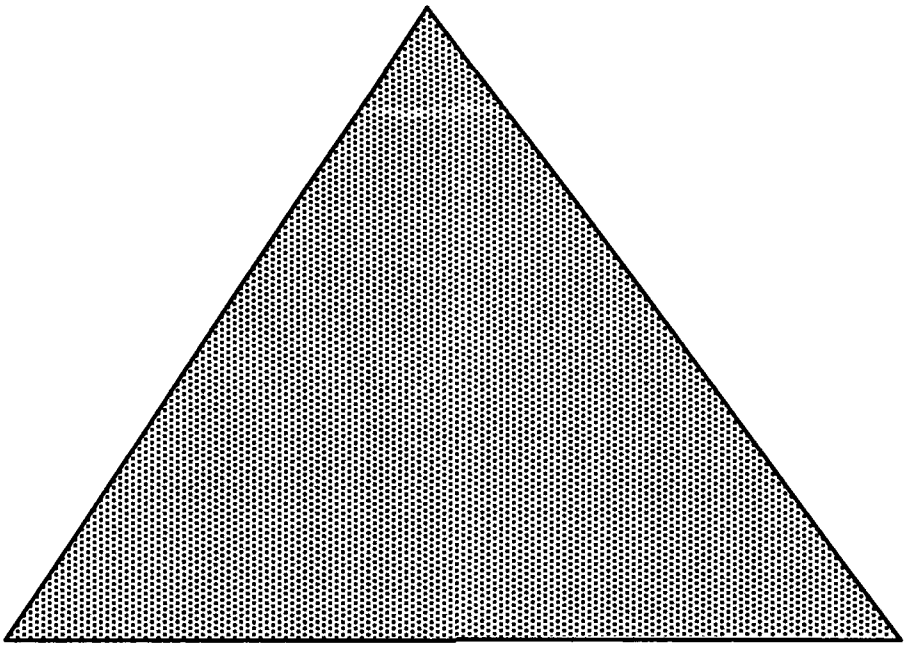

Figure 2.2. Exhaustive Search

2.3.1 General Heuristic Search Strategy. Barr and Feigenbaum describe a method used to reduce the search space

> Several graph- and tree-searching methods have been developed, and these play an important role in the control of problem-solving processes. Of special interest are those graph-searching methods that use *heuristic knowledge* from the problem domain to help focus the search. In some types of problems, these *heuristic search* techniques can prevent a combinatorial explosion of possible solutions. ...Various theorems have been proved about the properties of search techniques, both those that do and those that do not use heuristic information. Briefly, it has been shown that certain types of search methods are guaranteed to find optimal solutions (when such exist). (4:28)

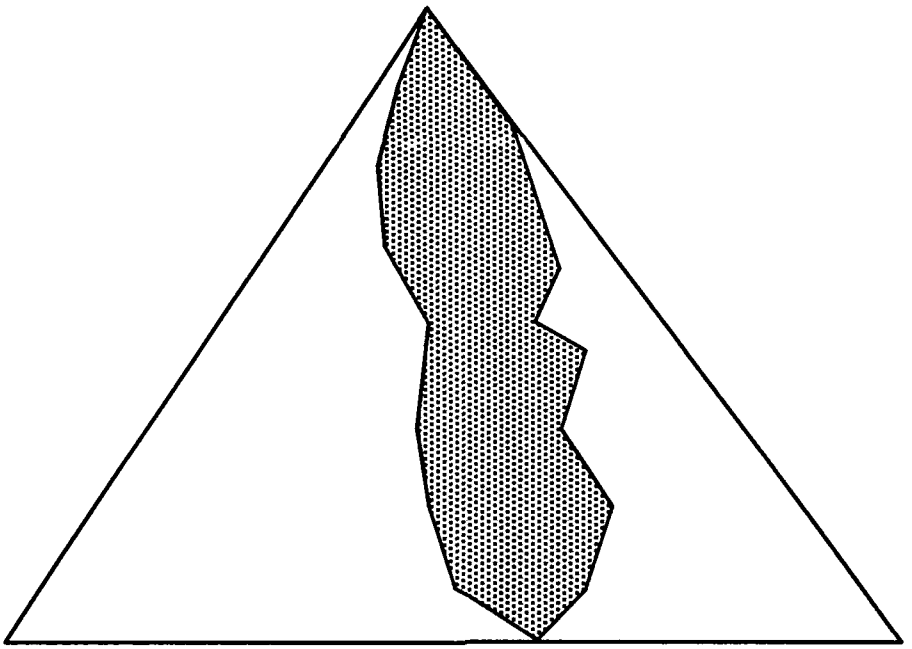

Figure 2.3. Heuristic Search

One of the most popular heuristic search strategies is the A* algorithm, a variation of the best first search strategy. The best-first search explores those paths which seem to be the most promising (4:59). The A in A* indicates that the heuristic employed is an additive function which is defined as

$$f' = g + h' \ . \tag{2.1}$$

Rich and Knight describe g and h' as "The function g is a measure of getting from the initial state to the current node. ...The function h' is an estimate of the additional cost of getting from the current node to a goal state " (46:75). The * in A* indicates the solution returned by the algorithm

is the optimal solution, based on the criteria used to calculate g and h'. This is guaranteed to be true if the heuristic is admissible. Pearl defines a heuristic function as being admissible if

$$h(n) \leq h^*(n) \quad \forall n \tag{2.2}$$

where $h^*(n)$ is the actual cost of going from the present node to the goal node (43:77). Pearl's notation is different than that used in this thesis. Throughout this thesis the heuristic value is denoted by h' and the actual cost is h. To limit the search space and guarantee an answer which is considered optimal it is necessary that h' be as close to h as possible without exceeding h. Also when creating the h' function, the designer must consider the trade-off between the time saved by reducing the search space and the time needed to calculate h'. If not careful it is possible that an h' is calculated which greatly reduces the search space, even goes straight to the optimal solution, but which takes a great amount of time to calculate. In this case a simpler h' could actually cause the search time to be reduced.

The outline for the basic A* algorithm is (46:76):

1. Start with the OPEN list containing only the initial node
2. If the OPEN list is empty, then exit algorithm and report failure
3. Remove path with the lowest f' from the OPEN list and place it on the Closed list
4. If this path is a solution, then exit and report the path
5. Generate all successor nodes
6. Find f', g, and h'
7. If successor node is on OPEN or CLOSED lists place the node in the path which yields the lowest g

8. If successor node is not on either the OPEN or CLOSED lists, then put it on the OPEN list and add it to the path

9. Return to step 2

Rich and Knight (46:74-75) describe the OPEN and CLOSED lists as:

- The OPEN list contains nodes which have been generated and f' found, but which have not been examined yet.
- The CLOSED list contains nodes which have already been examined.

2.3.2 Parallelized Heuristic Search Strategy. Within the framework of parallel processing, there are other techniques which can be applied to the A* search algorithm to further reduce the search space. Garmon incorporated a branch and bound strategy within his implementation of the A* algorithm to further reduce the search space.

The logical decomposition method for a search strategy is data decomposition. Thus each processor executes the same program, but operates on different data. As discussed previously, the nature of parallel processing requires communication between processors. Thus, the basic serial A* algorithm must be modified to incorporate communication. For this type of problem and decomposition, at a minimum the path with the best cost "to date" (f'), including the best cost of the solution needs to be communicated between processors. This information can be used by each processor to bound its search space. A search along a path whose cost exceeds the best cost can be terminated. Thus, the parallel A* algorithm must constantly check to see if new solutions have been found by other processors.

An implementation question which arises about the A* algorithm, in a parallel processing environment, is the open list. Garmon and others have looked at two methods to solve this problem, having either a centralized or distributed open list (1, 7, 19, 22, 47). In the centralized list implementation a single processor, or master, has the open list. The master assigned jobs

to "worker" processors by sending messages which contain the information needed to perform the search. Whenever a processor needs more work it simply communicates with the manager processor. The disadvantage of this method is the manager processor can become a communication bottleneck (1:1496)(7:105). Since all the processors in the system are communicating to a single processor, the manager processor could get inundated handling all the requests, thus slowing down the system. This method could solve the problem of load balancing, as long as the manager processor could handle all the requests for work in a timely manner.

The distributed list approach assigns each processor part of the problem, including the open list. In order to keep all processors busy (balanced loads), if a processor completes its processing then it simply broadcasts a request for more work. To limit the communication traffic this request would be sent to the processor's neighbors. If a neighbor had extra work it would share some of the work with the requesting processor. The advantage of this method is the elimination of a communication bottleneck at a single node and the distribution of memory and resource usage (1:1496). Each node is communicating with its neighbors for work instead of a single processor. The question that arises is "When does a processor share work?" It is possible that the system will begin to spend more time sharing work than processing the data, this is analogous to the operating system's thrashing principle. This may occur if processors have little work left and they share work. In this situation, the processor finishes its processing and then requests work from a processor to which it just sent work. Quickly, each processor can spend more time trying to share work than processing. This results in an inefficient system. (22)

Garmon found that for systems using small number of processors the centralized list implementation performed better than the distributed list implementation. As the number of processors increased the distributed list implementation began to perform better. This is logical since the centralized list implementation suffers from a potential communication bottleneck and as the number of processors increases the probability of a communication bottleneck occurring increases. (22:6-22)

2.4 *Mission Routing*

A sortie consists not only of flying to and from a target, but much more. Some of the other activities which take place are targets are selected and prioritized, weapon systems are selected to engage the targets, and plans for engagement are developed.

A selection of ingress and egress routes is based primarily on the ability of the pilot to perform the mission with the least probability of detection by enemy forces (i.e., maximum probability of accomplishing the mission); or in other words, the susceptibility, vulnerability, and survivability of the aircraft within the threat environment are the prime factors when evaluating mission routes (29:12). Figure 2.4 is an example of what a pilot is faced with when having to select routes. Timely identification of ingress and egress paths not only effects the accuracy of the routes, but also the lives of friendly forces which may be threatened by the presence of the targets. If routes are not selected in a timely manner and the mission carried out soon afterward, then the information upon which the routes were based may have changed. Thus, what may have been a safe route at planning time may now have become an extremely dangerous route to the aircraft and its crew.

Selecting a route can be viewed as simply finding the optimal, or shortest, path between the starting point and the target. The problem of finding the optimal ingress and egress paths can be reduced to the problem of performing a search within the domain of the threat environment. This holds true for the restriction that a mission will be flown to a single target. If a mission is to have multiple targets and other required locations, such as refueling points, then the problem is similar to a traveling salesman problem (TSP). In this case the optimal route between locations can be found then the optimal combination (i.e., overall route) would need to be determined using a TSP approach.

What makes a route, or even a path, optimal has not been precisely defined. Optimality is based on the maximum likelihood of success of the mission. When planning a mission, routes are not selected based on a single criterion, but rather on multiple criteria.

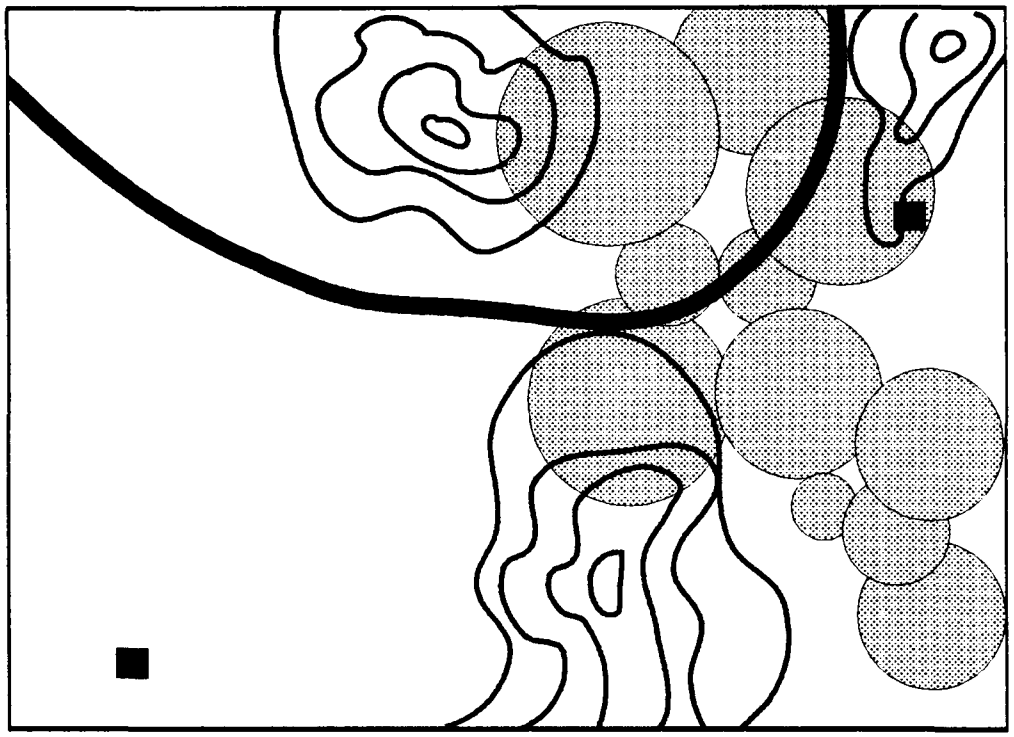

Figure 2.4. Example of a Mission

2.4.1 Representation of the Environment. One of the first problems encountered when designing any software which attempts to model the actual world is how to represent the world. As Mitchell writes, "The algorithm one uses to find shortest paths among obstacles depends on the representation used on the map" (37:174).

The "grid model", as the name implies, is based on a 2-dimensional matrix. Each location in the matrix corresponds to an actual location on the earth's surface. The value entered into the matrix is the elevation at the specified location. This is referred to as digitizing the terrain. The Defense Mapping Agency (DMA) uses this method to generate maps stored on computers.

Resolution refers to the surface area corresponding to a matrix entry. The greater the resolution (higher or finer resolution) the smaller the surface area represented by a matrix entry. The higher the resolution the greater the detail reflected in the digitized map, but also the greater the amount of space needed for the data. A side effect of resolution is the accuracy of the terrain eleva-

tion. As the resolution is decreased, the single elevation value must represent a greater area. The calculation of the elevation value introduces errors into the value used. There is a trade-off between the amount of space used and the amount of detail in the data. It is obvious that increasing the resolution results in a combinatoric explosion of the search space. Another problem with this model is known as digitized bias. Because the world has been digitized so too are the directions of traveled. Movement, using a digitized map, is limited to a finite set of directions. There are 4 or 8 directions for a 2-dimensional representation and 12 or 24 directions for a 3-dimensional representation. As the resolution of the terrain map is increased the digitized biasing effects decrease. (37:176,183)

"Representing terrain in the form of a regular grid of pixels is natural and simple" (37:182). There are many algorithms which can easily be adapted to search such a representation such as A* or dynamic programming.

2.4.2 Calculation of Enemy Radar Field of View. Since the main purpose of mission planning is the avoidance of detection by the enemy, it is necessary to model the coverage of the enemy's defenses. The Air Force Electronic Warfare Center developed a Electronic Combat (EC) software package known as the Improved Many-On-Many (IMOM).

The purpose of IMOM, as stated in the Users Manual, is to

- "Provide the battle management staff with integrated EC deployment information for better planning in the support and protection of combat crews and aircraft" (2:1-2).
- Assist in the planning of combat missions by calculating the EC environment through which an aircraft may fly (2:1-3).
- "Enhance situation awareness of the EC threat environment" (2:1-3).
- Provide a tool which allows a user to perform "what if" analysis (2:1-3).

IMOM uses digitized maps provided by the Defense Mapping Agency (DMA) (29:3). The user then enters, either manually or in a file, the locations and types of enemy radar. IMOM uses such

parameters as radar detection limits, antenna beam limit analysis to characterize the capabilities of a radar site. The software calculates the areas of radar coverage based on "terrain-limited, line-of-sight analyses", the capabilities of the radar site, and the radar cross section (RCS) and jamming capabilities of the aircraft (2:1-2)(48:2-5). The limitation of the IMOM system is that is only calculates the field of coverage for a single altitude, either above ground level (AGL) or mean sea level (MSL) (2:5-25). Thus to represent the 3-dimensional field of coverage the IMOM system must be run for each altitude desired.

Isensee modelled the radar coverage as threat cells with a probability of detection assigned to each cell (29:36-37). He used a matrix, having the same resolution as the DMA terrain data, to represent the field of coverage for the enemy radar network.

2.4.3 Planning. "Until recently pilots still planned their missions in much the same way as during World War II – using pencils and rulers to plot courses, way points, fuel burns, and threat-evasion tactics on paper maps" (11:35). Each of the armed services is directing research into automated mission planning systems. The details of each system are not available because of either security or proprietary reasons. Research has been conducted at the Air Force Institute of Technology (AFIT) into mission planning aides and these efforts are discussed in the following sections.

2.4.3.1 Tactical Mission Planner (TMP). Maj Bahnij, an Air Force fighter pilot, developed a tool to aid pilots in the formation of mission routes. He built his system using the LISP language and the Knowledge Engineering Environment (KEE) software development system. His prototype system, developed for an F-16, moved the paper and pencil onto a computer. The pilot could identify the configuration of the aircraft; this information was used to calculate fuel usage. A topographical map was displayed and using a mouse the pilot could identify legs and turn-points for the mission route. The system would automatically calculate location of the turn-point (longitude/latitude), distance traveled, and fuel consumption. This not only reduced the time needed to

formulate the mission route, but also eliminated much of the "drudgery" work performed by pilots during this phase of mission planning. This allowed the pilot to concentrate more on the mission itself, providing better situational awareness. (3)

A follow-on effort to the work of Maj Bahnij was conducted by Lt Bradshaw. He incorporated the Intelligence Analysis Expert System (IAES) thus providing a radar coverage overlay on the topographic map. This allowed the pilot to plan a route more intelligently. He also modified the user interfaces to make the TMP more user friendly. One such change was giving the pilot to change a leg any time during the planning, thus allowing a pilot to explore different options. The system also prepared the flight card for the pilot. (9)

2.4.3.2 TMP Automated Route Selection. Capt Spear performed further research on mission planning aides using the TMP. His work entailed interfacing the TMP with the IMOM system and adding an automated route selection search. He did not directly interface the TMP with IMOM. The TMP was written in LISP and resided on a Symbolics 3600 LISP computer while the IMOM system was written in FORTRAN and resided on a DEC VAX. Spear developed a procedure to capture the IMOM results, convert the results usable by the TMP, and transfer the data to the Symbolics machine. Spear's main effort was the addition of a search routine to find the "optimal" mission route. Optimality was based on total distance flown and the total probability of detection by enemy radar. The user could specify a weighting factor for each value, where the summation of the weighting factors equaled one. An A* algorithm was used to perform the search. Spear used a simple, but admissible h', where h' is calculated as the straight line distance between the present location and the goal location plus the probability of radar detection at the next location along the straight line. He not only calculated h' but also checked on the feasibility of a path being examined. As a node was expanded the total distance travelled plus the straight line distance between the present node and the goal location was compared to the combat radius

of the aircraft. If the distance was greater than the combat radius then the search along the path in question was discontinued. This was a heuristic used to bound the search space. (48)

2.4.3.3 Multicriteria Network Routing. In 1991, CPT Isensee performed work similar to that of Spear though not using the TMP system. Unlike the work of Bahnij, Bradshaw, and Spear, Isensee looked at finding the "optimal" route when engaging more than one target. His research effort was based on the principle of optimality, which states that an optimal solution is a combination of optimal sub-solutions (10:143-144). Thus he found the optimal routes between the starting point and each target, and between each of the targets. Then he had to find the combination of sub-routes which resulted in the overall best route which is simply a traveling salesman problem. He used IMOM to calculate the enemy radar coverage, then using a variation of Dijkstra's algorithm he search for each of the optimal sub-routes. Route selection was based on three factors: (1) distance traveled, (2) probability of detection by active radar, and (3) probability of detection by passive radar (29:30). Isensee entered the sub-routes into a software package known as ADBASE which calculated the overall best route. Using the linear programming approach of the ADBASE system, he looked at not only finding the optimal route, but also the combination of the criteria for selecting a route.

2.5 Summary

The concept of parallel computers has been around since the 1960s, but it is only recently that it has become a major research effort. It is not a trivial task to either modify software designed for a serial computer or to design new systems for parallel processing. A designer has much to consider when developing software for parallel computer systems. Some of the things which need to be considered are, how to decompose the problem, the communication between each part, and the type of architecture on which the software will run.

Search is not only an integral part of the AI and Operations Research communities, but also other disciplines. We as humans perform search all the time as we try to remember things by retrieving information from our brains. Research has been conducted, and still is, into efficient search techniques though there are standard strategies such as depth-first search (DFS), breadth-first search (BFS), and A*.

Pilots have become saturated with the information and responsibilities of planning missions. Research is being conducted into the development of automated tools. These tools will help to reduce the work load of pilots and allow the pilot to concentrate more on the mission itself, thus providing better situational awareness.

The design, both high-level and low-level, of a parallelized mission routing software package is presented in the next two chapters. The design addresses the concerns identified in this chapter and builds upon the results of the research described in this chapter.

III. Requirements and High-Level Design

3.1 Introduction

This chapter presents the methodology used in this research and a high-level design of the parallelized A* algorithm used to solve the mission routing problem. Before the design of the software can begin a thorough understanding of the problem is necessary, thus section 3.2 is a detailed description of the mission routing problem and its requirements. The methodology used in designing the parallelized A* search is discussed in section 3.3 and the high-level design is discussed in section 3.4. Lastly, the tasks identified for this effort are prioritized in section 3.5.

3.2 Understanding the Mission Routing Problem

3.2.1 Detailed Description. Chapter I discussed the mission routing problem in general terms and Chapter II discussed the current state of research into solving the mission routing problem. This section discusses, in detail, the mission routing problem as set forth for this research effort and the requirements established.

As stated previously, mission routing consists of the selection of a route to a target and is based primarily on the ability of the pilot to complete the mission with the greatest probability of success. The process of selecting routes is a complicated process which must take into account many variables, thus a selection using multiple criteria. The configuration of the aircraft, fuel consumption, time to reach the target, time on target, aircraft speed, areas of enemy threats, detection by the enemy, and weather are many of the variables pilots must take into account when planning routes. The question which arises is which variables are most important and how are each of the variables weighed against the others. What mental process does a pilot go through when selecting mission routes. Is that the best method and how can that process, or the best one, be modelled on a computer? These are the type of questions a knowledge engineer or a software

engineer poses and attempts to answer when designing any type of system which models human thought process.

3.2.1.1 Mission Parameters. The Air Tasking Order (ATO) is generated by higher headquarters and sent to to the wing level flying units. This starts the mission planning process. The ATO assigns all the units, under command of the higher headquarters, specific missions for that day. A Fragmentary Order (FRAG) is a subset of the ATO and is the mission assigned to a single unit. The ATO contains such information as:

- target identification and location,
- when the target needs to be attacked (time on target),
- type and number of aircraft to attack the target,
- aircraft weapons configuration, and
- support aircraft (i.e. AWACS, tankers, escorts, electronic jammers, etc).

This information on Air Tasking Orders was taken from the thesis written by Bahnij (3:II-4).

3.2.1.2 Representation of the World. More information is needed before the pilot can begin selecting mission routes. The pilot must know the terrain over which the aircraft will be flown. This is not only to avoid flying into obstacles, but also to select landmarks to aid in the navigation of the aircraft during flight. Pilots use some type of map when planning missions, but when terrain information is stored in a computer questions ensue as to how to represent the earth's terrain and the format used to store this information. The two main methods of representing terrain information are using a grid and using polygons. The grid (31, 30, 38, 37, 42) and the polygon (36, 41, 37) terrain representations have each been used in researching path planning systems.

"Representing terrain in the form of a regular grid of pixels is natural and simple" (37:182). A two dimensional matrix is used to represent the terrain. Information about the terrain at each

grid point (pixel) is collected and stored. This is the form used by the Defense Mapping Agency (DMA) for its digital terrain database (31:578)(38:172)(37:182-183). The information stored for each grid location consists of (38:173):

- Elevation
- Surface Material (i.e. water, soil, trees, etc.)
- Mobility Factor
- Structural Features (i.e. roads, bridges, dams, etc.)

Not all of this information is needed by a pilot though it could be helpful in identifying landmarks both during planning and flight.

A major advantage of the grid method is that it "is compatible with advanced navigation aids such as LORAN and GPS, which can give the position of a vehicle in longitude and latitude" (30:135). Thus the information from the aircraft sensors could be used in conjunction with the terrain data and selected route to navigate the aircraft. Another advantage is the ease of imposing threats onto the map (30:135).

As the resolution of the terrain data increases so does the number of points in the route. The increase in resolution causes the search space to increase resulting in a combinatoric explosion. The number of possible routes of a given length is defined as

$$\text{Number of possible routes}_l = n^l \tag{3.1}$$

where l is the length of a path and n is the maximum number of possible children locations from any given parent location. If the resolution were increased by 100% then the dimensions of the matrix doubled. This would mean that the length of a route between the same locations would now be $2l$ resulting in

$$\text{Number of possible routes}_l = n^{2l} \tag{3.2}$$

It is evident that increasing the resolution of the terrain data does not result in a linear increase in the search space, but rather an exponential increase in the search space. This is one of the reasons this research is being conducted, to reduce the impact combinatoric explosion has on the time necessary to find a route.

Another area of concern with a grid representation is that of digitized bias. Because the world has been digitized so too are the directions of traveled. Movement, using a digitized map, is limited to a finite set of directions. There are 4 or 8 directions for a 2-dimensional representation and 12 or 24 directions for a 3-dimensional representation. As the resolution of the terrain map is increased the digitized biasing effects decrease (37:176,183). Another phenomenon of digitized bias is that of path and distance between two points. Because of digitization a straight line may not be able to be drawn between two points. This is depicted in Figure 3.1 taken from Mitchell (38:174). As can be seen in the figure, the path between points A and B is not a straight line. The cost of traveling from point A to point B is 7.6569 units while the straight line distance is 7.2111 units. Another side-effect of digitization is that there is more than one path with the same cost.

As mentioned the other method of representing terrain information is through the use of polygons. Obstacles are described are polygons in terms of their boundary representations. "Obstacles are given as a list of k simple polygons, each represented by a doubly linked list of vertices (each vertex just being a pair of coordinates, either integer or real)" (37:173). Figure 3.2 is an example of a polygonal representation of terrain data. This type of representation is easier for a human to visualize obstacles and possible routes. An advantage of this type of encoding is that it can save on storage costs since not all points need to be specified, just the "corner points" of the polygons. Also the problem of digital bias has been eliminated, since straight lines between points can be specified. The drawback is that the accuracy of the terrain representation is reduced. Since polygons are used to approximate the terrain the actual terrain features along a polygonal edge are not exact, which could be a problem.

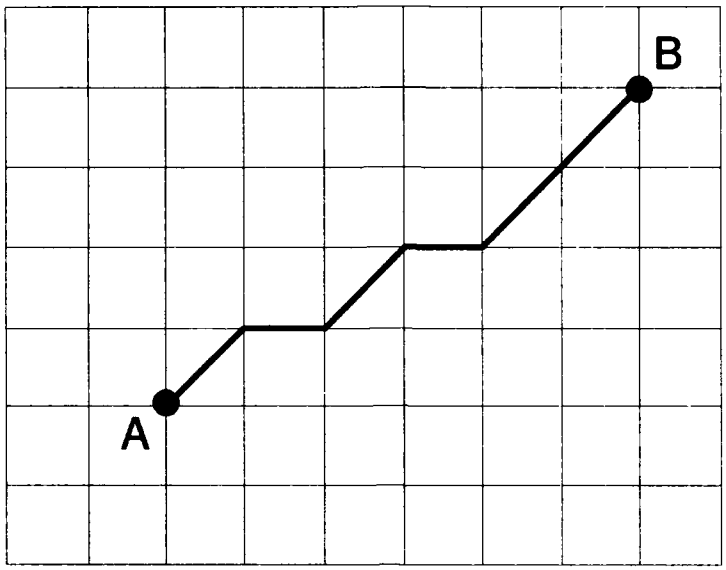

Figure 3.1. Digitized Bias

Both techniques have advantages and disadvantages. Each characteristic must be weighed when selecting which method will be used to model the terrain.

The final bit of information needed before the selection of mission routes can begin is the intelligence data. In order to avoid detection by the enemy the pilot must know where the enemy's radars and other means of detection are located. Not only is a knowledge of their locations important but so is a knowledge of their ability to detect and their field-of-coverage. Pilots receive briefings on information gathered for intelligence sources and an analysis of all known threats. This information is used by the pilot when selecting mission routes. Software packages have been developed to model radar coverage which aid in the determination of threat analysis and aircraft detection. The Improved Many-On-Many (IMOM) system developed for the Air Force Electronic Warfare Center provides such a capability (2). The IMOM system can take into account such parameters as type of radar system, terrain effects on the radar's signal, electronic countermeasures (ECM), radar cross section (RCS). The IMOM system uses the DMA digital terrain database, a

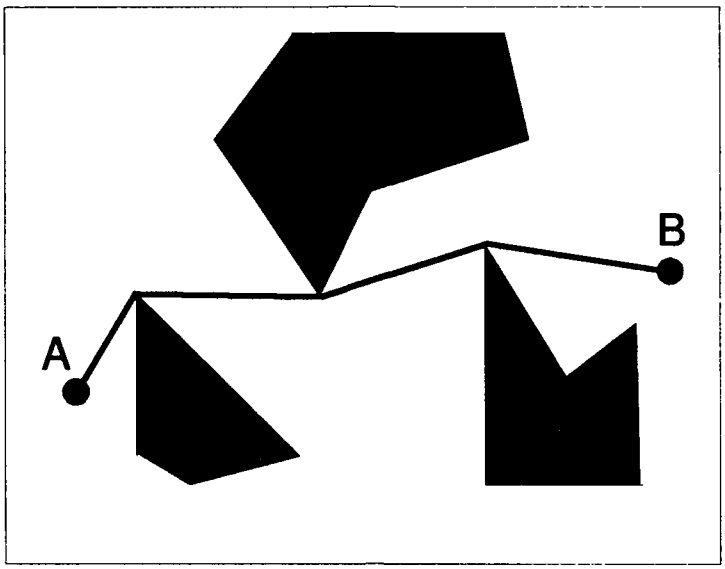

Figure 3.2. Terrain Representation Using Polygons

database containing parameters of known enemy radar, and a database of known ECM and RCS information. Thus the IMOM system also models the real-world using a grid representation.

3.2.1.3 Search. The heart of the mission routing problem is the selection of the best route which entails a search. There are a number of different search strategies available. Some of these strategies are discussed in the following paragraphs.

The uninformed, or "brute force," search method simply explores all routes until a solution is found. The two main variations of this type of search strategy are the "depth-first search" (DFS) and the "breadth-first search" (BFS).

The depth-first search, as its name implies, explores a single route at a time from beginning to end before exploring another route or in other words the algorithm gives priority "to nodes at deeper levels of the search graph. (43:36)" The algorithm begins at the starting location and selects an edge leading to a next location. This new location becomes the present location and the process

of selecting edges and moving to the next location associated with the edge continues until the goal location is reached or until a location has no edges, known as a leaf node, is encountered (15:477). With "backtracking," once a solution is found or a leaf node is reached the algorithm returns to the previous location and continues searching along another edge (10:171). This search strategy is depicted in Figure 3.3, where the number at each node in the search graph represents the order in which the node is explored. The depth-first search strategy can be implemented using a stack (43:37).

The breadth-first search algorithm "is one of the simplest algorithms for searching a graph and the archetype for many important graph algorithms " (15:469). Dijkstra's shortest path algorithm and Prim's minimum spanning tree algorithm are based on a breadth-first search strategy (15:469). The breadth-first search strategy looks at all paths at the same time. While the DFS approach gives higher priority to nodes at lower levels in the search graph, the BFS approach "assigns a higher priority to nodes at the shallower levels of the search graph, progressively exploring sections of that graph in layers of equal depth" (43:42). Thus it explores all nodes which are at the same depth of the search graph before moving to the next depth. Beginning at the starting location all the next locations are found. Then, at each of these locations, each of the next locations is found. This continues until the goal is found or there are no more nodes to be explored. The breadth-first search strategy can be implemented using a first-in-first-out (FIFO) queue (43:42)(10:182-183). Figure 3.4 depicts a breadth-first search for the same search space as that in Figure 3.3. Again the numbers at each node indicate the order in which the node was examined. Each algorithm simply finds a solution not necessarily the best solution. To find the best solution an exhaustive search would need to be conducted. These unguided search strategies can be inefficient in these cases; therefore, "several techniques have been developed to guide the search and improve its average efficiency " (1:1492). One of these techniques is known as the branch and bound strategy. To keep from performing an exhaustive search the search space can be bounded by storing the "cost" of the found solution. Then as each new node in the search graph is explored a determination

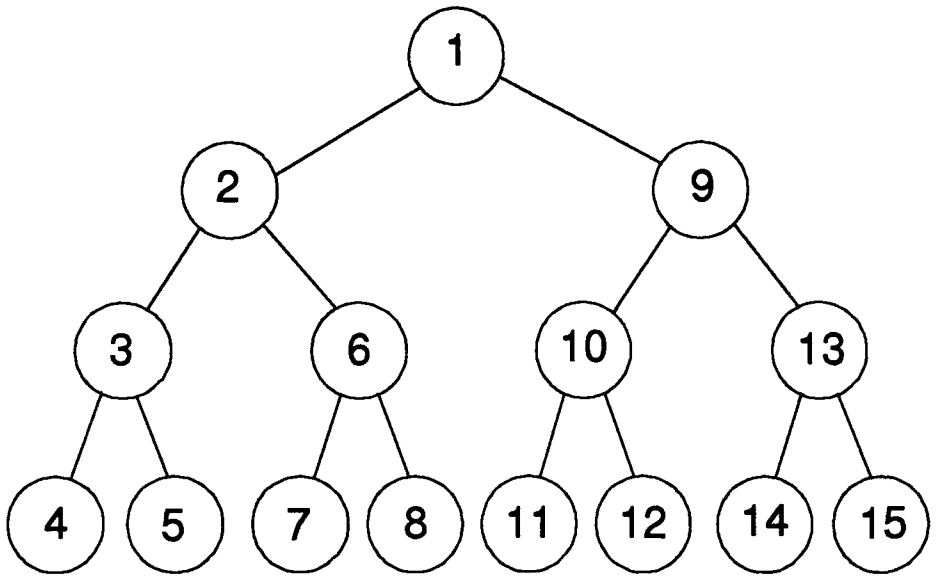

Figure 3.3. Example of a Depth First Search Strategy

is made whether continuing searching from that node will result in a solution of greater cost. If the cost at the node is equal to or greater than a known solution then exploration at that node is discontinued. The purpose of the branch and bound strategy is "to prune certain branches of a tree or to close certain paths in a graph" (10:199). Whenever the cost of a route being explored exceeds that of the best solution found thus far the search of that route is discontinued and the algorithm backtracks and continues searching along another route. This is seen in Figure 3.5 where searching is discontinued along some paths when the cost of reaching a node has exceeded the cost of a known best solution.

Another type of class of search strategies is that of heuristic search. Just as the branch and bound approach tries to reduce the search space so to does a heuristic search. The A* search algorithm is a member of the class of heuristic search. It uses a projected cost, the heuristic, added

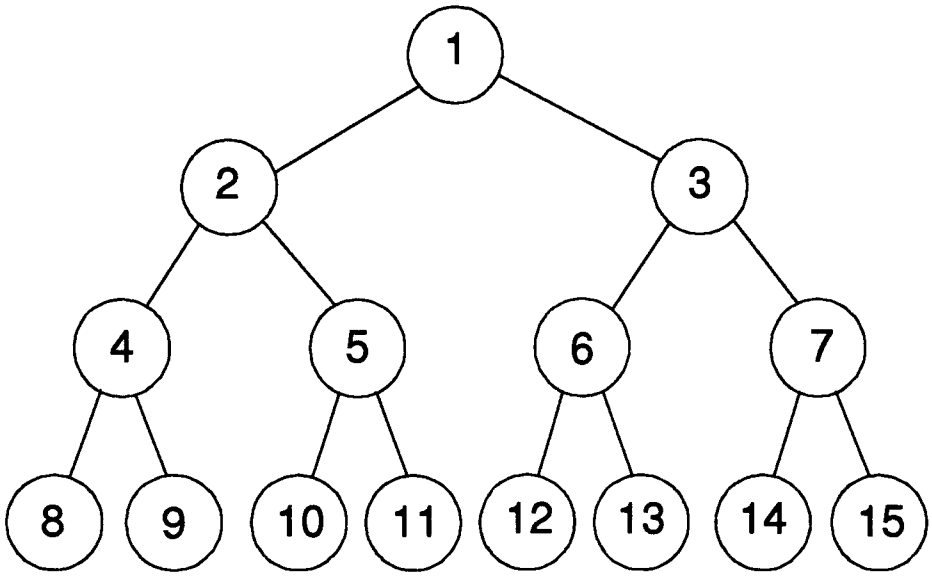

Figure 3.4. Example of a Breadth First Search Strategy

to the cost of reaching a node in the search graph. This additive function is given by

$$f' = g + h' \tag{3.3}$$

where g is the cost to reach the node and h' is the projected cost of arriving at the goal state. The A* algorithm is a specialized best-first strategy where the path which appears to lead to the best solution is explored. This is a divide-and-conquer, top-down, approach in that the A* approach divides the search space into smaller and smaller instances as the algorithm progresses (10:142). As discussed in the previous chapter the algorithm is guaranteed to give the optimal solution (the meaning of the *) as long as h' is admissible, where admissible is defined as

$$h' \leq h_{actual} \ . \tag{3.4}$$

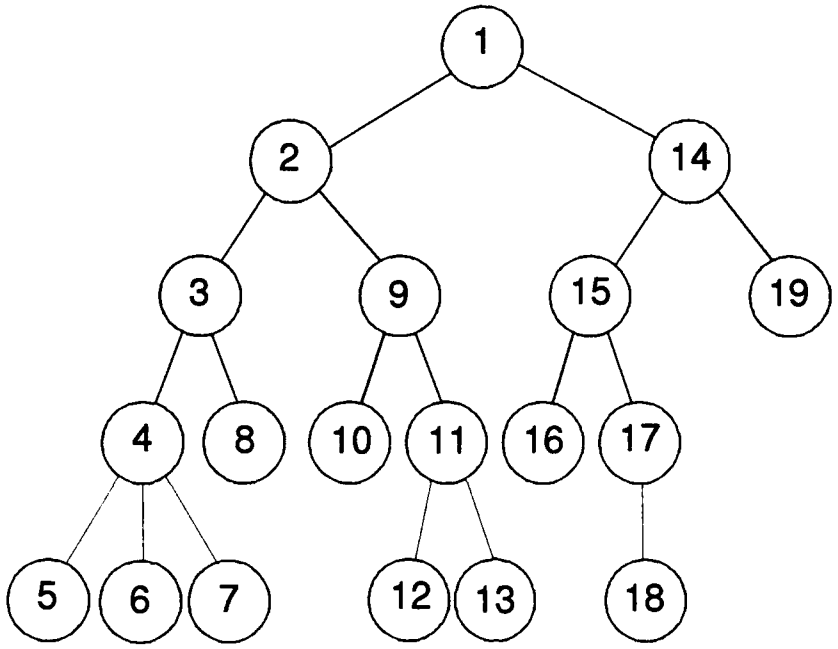

Figure 3.5. Example of a Depth First, Branch and Bound, Search Strategy

If h' is not admissible then the solution may or may not be the optimal. The overhead associated with the A* algorithm is management of a data structure used to store all the paths being explored. Dynamic programming, on the other hand, is a bottom-up technique. It is based on the principle of optimality which states that an optimal solution is composed of partial solutions which are themselves optimal. This principle is illustrated in the following shortest path example, "if k is a node on the shortest path from i to j, then that part of the path from i to k, and that from k to j, must also be optimal" (10:150). "Dynamic programming efficiently solves every possible subinstance in order to figure out which are in fact relevant, and only then are these combined into an optimal solution to the original instance" (10:144). A dynamic programming approach finds the optimal path between a start node and any other node once the goal is found then the path leading to that goal is constructed. This is true as long as the principle of optimality holds true; otherwise, as with the A* strategy, there is no guarantee that the solution will be the optimal solution. The

overhead associated with the dynamic programming approach is the data structure used to record the connections between any two nodes in the search space.

3.2.1.4 Reporting of Solution. The whole goal of the mission routing process is to select the route which maximizes the success of the assigned mission. Any system, manual or automated, must be able to present the selected route to a pilot in such a manner such that the pilot is able to navigate the aircraft to the target. A simple list of latitudes/longitudes is one means, but this reduces the pilot's situational awareness. The pilot needs to be aware of landmarks, obstacles, and threats in order to properly navigate and plan for contingencies. A graphical display including the selected route and calculated radar coverage overlaid on the terrain map helps to increase the pilot's situational awareness, but also the pilot's confidence in the system selecting the route.

3.2.2 Requirements. From the discussion in the previous section, the requirements for a system which solves the mission routing problem can be extracted. The primary requirements found in the detailed description of the problem are contained in the following list.

- Model the terrain over which the aircraft flies
- Model the enemy's detection capabilities
- Select the best route for the mission
- Develop a user interface for the entering of data and the display of the selected route

Each of the primary requirements can be further delineated into secondary, or supporting, requirements. Some of the secondary requirements identified from the primary requirements are contained in the following lists. Modelling of the enemy's detection capabilities is further refined to include:

- Model radar characteristics

- Model terrain effects on the radar signal
- Model electronic jamming
- Model radar cross section (RCS)
- Model other methods of detection (i.e. troops on the ground)

One of the primary requirements is the selection of the best route. As mentioned this is the heart of any software system used to solve the mission routing problem. There are a number of considerations associated with this requirement and some of these are contained in the following list.

- Criteria used in the decision of the best route
- Weighing of each of the criterion in the decision process
- Method used to reduce the search space
- Type of search algorithm used
- Selection of both the ingress and egress routes

3.3 Design Methodology

The use of software engineering principles is an essential element in the design of any software algorithm. Designing and implementing a complicated algorithm on a sequential computer can be difficult for even an experienced person. As discussed in Chapter II the intricacies and complexities of parallel processing makes this task even more difficult. The concept behind software engineering is to provide a designer with tools and procedures to reduce the time necessary to develop, implement, and maintain a software system. (44:xix-xx)

3.3.1 Software Engineering Principles. For any task the first step in solving the problem is to have a thorough understanding of the problem, which was presented in Chapter II and the

previous section. This is the basis for using any of the other software engineering tools available. Only after the problem is understood and the requirements defined can the development process continue. Until the groundwork has been laid any other endeavors could be a waste of time and effort. This is the basis for the "waterfall" software development model. The assumption is made that each stage of the development cycle is fully understood and completed before moving on to the next stage. This is not always a realistic assumption. Other models, such as "rapid prototyping" and "spiral", assume that during the development process changes will occur or new insights will be obtained. These models provide a means for returning to any previous stage with the new information. Thus the basic premise of these alternative models is that change will occur and that this change is not necessarily bad. A form of rapid prototyping is employed during this research effort because of the initial lack of problem understanding. The rapid prototyping model provides means to adjust the requirements and scope of the research as a clearer understanding of the problem and its associated complexity is developed and as problems are encountered during the implementation and testing of the software.

The development of a software package for this research utilizes a top down design approach. This is evident in the structure of not only this chapter, but the overall thesis with a chapter dedicated to the high-level design of the research effort and another chapter to the low-level design. This approach, as already mentioned, helps to reduce the time necessary to conduct the low-level design and implementation of the algorithm used to solve the mission routing problem.

A design specification language, structure charts, and pseudo-code are software engineering tools also employed to support the design of a mission routing software package. The design specification language being used is UNITY, which is discussed in the next section. It helps to define the high-level search requirements. Structure charts help in the refinement of the high-level design and the implementation of the algorithm. They also give a pictorial representation of the implemented software aiding in the understanding and maintenance of the software. Pseudo-code is

used to given a structured english description of the algorithm. Another software engineering tool used is that of modularity. "The concept of modularity in computer software has been espoused for almost four decades. ... software is divided into separately named and addressable elements, called modules, that are integrated to satisfy problem requirements" (44:222). Large computer programs which do not utilize modularity cannot be easily understood by others. This is because of the number of control paths, variables, and overall complexity (44:222). Thus, modularity increases the ability of others to understand a program, but this is not all. As part of the top down design approach it allows the designer to specify needs functions and procedures without having to specify how that function or procedure is to be implemented. This leads to another advantage of modularity which is the easy of testing and debugging software. Modularity permits the isolation of code which can reduce the time needed to verify the correctness of the code. This is not only important for software designed for a serial computer architecture, but also extremely important for software designed for parallel architectures.

Existing software is used as much as possible to help reduce development and testing time, though the existing software must be integrated into this research endeavor. The purpose of this research is not to model radar characteristics, but to use that information in the selection of routes. Software packages, such as the IMOM system, have already been designed, implemented, and testing which perform that function. Routines already exist which read in data from the DMA digitized terrain database and which perform the management functions of the open list queue. There is no reason why the wheel needs to be re-invented when one has knowledge and access to the wheel. This is the purpose for libraries of software routines. Also the focus of present research efforts in software engineering is in the development of reusable software packages.

3.3.2 UNITY. Chandy and Misra introduced "a computational model and proof system" known as UNITY (Unbounded Nondeterministic Iterative Transformations) (12). UNITY is not a programming language though it is a way to view to operation of a program. UNITY uses the

Backus-Naur format (BNF) notation which is based on first-order predicate logic (12:22). UNITY programs

> consist of a declaration of variables, a specification of their initial values, and a set of multiple-assignment statements. A program execution starts from any state satisfying the initial condition and goes on forever; in each step of execution some assignment statement is selected non-deterministically and executed. Non-deterministic selection is constrained by the following "fairness" rule: Every statement is selected infinitely often. (12:9)

Thus a UNITY program lists all the variables with their initial conditions. Then an assignment statement is selected and executed. The fairness rule ensures that all assignment statements will get the opportunity to be executed. Chandy and Misra continue the description of the UNITY notation.

> A UNITY program describes *what* should be done in the sense that is specifies the initial state and the state transformations (i.e., the assignments). A UNITY program does not specify precisely *when* an assignment should be executed – the only restriction is a rather weak fairness constraint: Every assignment is executed infinitely often. Neither does a UNITY program specify *where* (i.e., on which processor in a multiprocessor system) an assignment is to be executed, nor .... Also, a UNITY program does not specify *how* assignments are to be executed or *how* an implementation may halt a program execution.
>
> UNITY separates concerns between *what* on the one hand, and *when, where,* and *how* on the other. The *what* is specified in a program, whereas the *when, where,* and *how* are specified in a mapping. (12:9)

UNITY provides a means to describe an algorithm in an abstract manner and at the same time exploit the algorithm's parallelism. It also provides a means of formally proving the correctness of a design. This is an important concept which helps to identify and correct errors before implementation and testing (13:47). Identifying and correcting errors early saves both time and resources (25:368). The operation and termination of a UNITY design is proven through the use of predicate logic. It "provides temporal logic constructs to prove stability and fixed-point behavioral properties of computer programs" (33:3). This method of algorithm description and verification

helps to reduce the time necessary to design and implement an algorithm on a parallel processing architecture.

3.4 High-Level Design

3.4.1 English Description. The problem encounter when selecting routes for missions is finding the route which possesses the highest probability of success. There are a number of parameters which figure into the determination of which route is the best. The focus of this research is that of the search strategy, and supporting parallel architecture, used to select the best route. Other components of the software package are needed to provide the information through which the search shall be conducted. Figure 3.6 shows the overview of the software which will be used to find the best route for a mission.

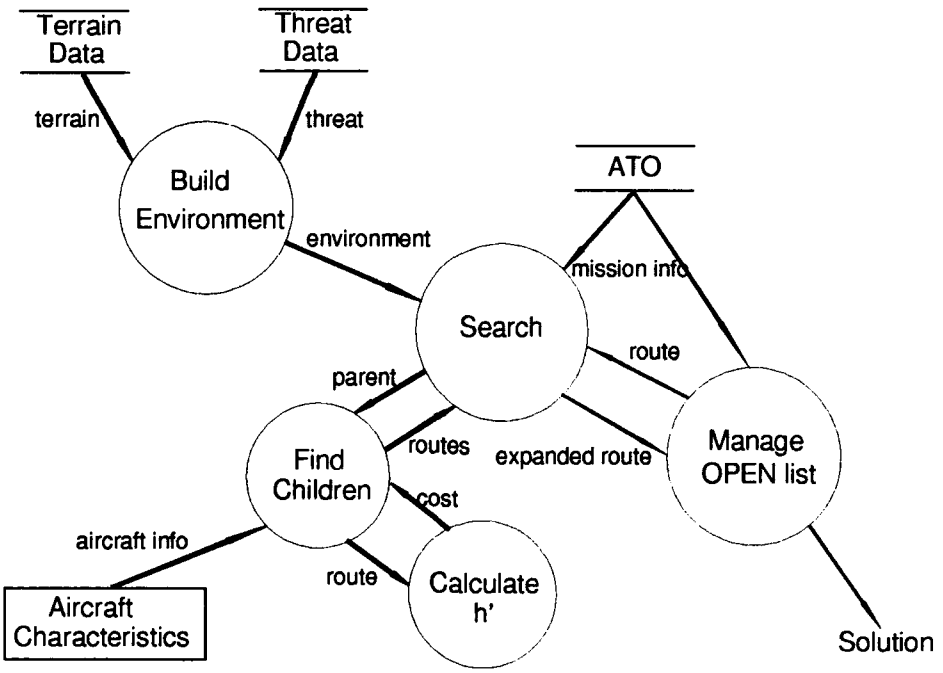

Figure 3.6. Overview of the Entire Mission Routing Package

The heart of the search of the best mission route is the A* search algorithm. The A* algorithm is a search strategy which attempts to explore the route which appears to result in the overall best

solution. Starting with the beginning location all valid "children," or next locations, are found and each one produces a new route which may need to be examined further. As each new route is found it is placed into the open list. The open list is a priority queue sorted on the projected cost of the routes. The route at the head of the queue has the lowest projected cost and thus appears to give the best solution. It is this route at the head of the open list which is removed and has its children found. This process of removing routes, finding children, and placing new routes back into the open list continues until the route at the head of the open list is a solution. Thus as solutions are found they are simply inserted into the open list ensuring that the best route is found. The parallelism of examining routes and finding valid children will be exploited and this is shown in the UNITY description of the A* search algorithm of the next section.

3.4.2 UNITY Description. Before presenting the UNITY description it is first necessary to explain the UNITY notation. The following is an explanation of the UNITY notation (32:2)(7:7-8).

$\langle\ \rangle$	Scope of a statement
$\|\|$	Statements may be executed concurrently
$[]$	Statements must be executed consecutively, though the fairness rule still applies

The following is the UNITY description for the high-level design of the A* algorithm.

Program *A* Search*
  declare
    {Start, Best Vertex, Best Path, Child, and Goal}
    S, BV, BP, C, G : vertices
    path : array[1..N] of vertices
    open : array[1..M] of paths

  always
    {The OPEN list is sorted in ascending order based on f values}
    $\langle\ \|\ \forall i, \forall j\ :\ (0 \leq i < N) \wedge (0 \leq j < N) \wedge (i \leq j)\ ::\ f(open[i]) \leq f(open[j])\ \rangle$

    {The vertex v is always within the search space }

$\langle\ v \in$ Search Space $\rangle$

{The Best Path is always at the head of the Open list }
{and the Best Vertex is the last vertex visited in BP }
$\langle$ BP = open[1] $\rangle$
$\langle$ BV = BP(end) $\rangle$

initially
{Only the Start vertex is initially on the OPEN list}
path = {S}
open = path
found = FALSE

assign
$\langle \| \forall v : v = \text{child(BV)} :: \text{open} := (\text{BP} \wedge v\ )$ if found = FALSE $\rangle$

$[]\ \langle$ found := TRUE if BV = G $\rangle$

end *A* Search*

3.4.3 Formal Proof of UNITY Description Correctness. As mentioned UNITY provides a means to formally prove the correctness of a UNITY description. This section proves the correctness of the A* algorithm description of the previous section.

3.4.4 Fixed Point Exists A simple examination of the UNITY descriptions reveals the fixed point condition. The assignments

BP := OPEN(head)

OPEN := (BP $\wedge$ v)

are made only if found = FALSE, thus as long as found = FALSE then assignments are made and a fixed point is not reached. When found = TRUE then the assignments are not made and the system does not change states; therefore, the fixed point is reached. The last assignment statement of the UNITY descriptions is

found := TRUE

and it is made when Parent(BP) = Goal. When the path at the front of the OPEN list has the goal as the final location, the best path has been found and the fixed point condition is satisfied. Thus, the fixed point condition is found = TRUE (the best path has been found).

Another way to show the solution of the fixed point (FP) condition follows:

$$
\begin{aligned}
\text{FP} \equiv\ & [\text{BP} = \text{OPEN(head)} \ \vee\ \neg\,(\text{found} = \text{FALSE})] \ \wedge \\
& [\text{OPEN} = (\text{BP} \wedge v) \ \vee\ \neg(\text{found} = \text{FALSE})] \wedge \\
& [\text{found} = \text{TRUE} \ \vee\ \neg\,(\text{Parent(BP)} = \text{Goal})]
\end{aligned}
$$

Solving these logic statements it is found that the fixed point occurs when Parent(BP) = Goal resulting in the assignment found = TRUE being made.

3.4.5 Fixed Point Reachable Now that we know the fixed point the next step is to show that it will be reached. A simple way to prove the fixed point is reachable is to establish a metric (m) and then use the induction principle for leads-to (12:72). The principle is

$$\frac{\langle \forall m : m \in W :: p \wedge (M = m) \mapsto (p \wedge (M \prec m)) \vee q \rangle}{p \mapsto q} \tag{3.5}$$

In proving the fixed point is reachable we let p = true and q = the fixed point (FP) which results in

$$\frac{\langle \forall m : m \in W :: true \wedge (M = m) \mapsto (true \wedge (M \prec m)) \vee FP \rangle}{true \mapsto FP} \tag{3.6}$$

We can reduce this equation to

$$\frac{\langle \forall m : m \in W :: (M = m) \mapsto (M \prec m) \vee FP \rangle}{true \mapsto FP} \tag{3.7}$$

We need only show that $(M = m) \mapsto (M \prec m)$ to prove the fixed point is reachable.

As the metric for the A* search we use the value h (not h'). As the search progresses the paths leading to the goal state are explored. Since h is the true value of traveling from the present location to the goal state it is obvious that this value should be decreasing as the search progress. Thus, as assignments are made in the UNITY description the paths should be getting closer to the goal state and h will be decreasing. Thus, $(M = h) \mapsto (M \prec h)$ if assignments are being made and therefore the fixed point is reachable.

3.4.6 Mapping Schema. The read-only schema is used to map the parallelized A* search algorithm to the architecture being used. The read-only schema stipulates that each variable in a program can be modified by at most one processor (12:85). Each variable, in the UNITY description, is found only once on the left side of an assignment statement. Also, since the right side of each assignment statement names a single variable modified by another assignment statement this is an fine-grained read-only schema. In mapping this description to the architecture the following properties hold true:

- the Best Path variable appears on the left side of an equation only on the controller processor, while on the worker processors it appears only as an input parameter, or in other words only on the right side of an equation.
- the boolean variable FOUND also appears on the left side of an equation only on the controller processor and on the right side for the worker processors.

Thus the criteria of the read-only schema holds true and no changes are necessary when mapping the UNITY design to a particular architecture.

3.4.7 Pseudo-Code. A combination of functional and data decomposition are used to decompose the problem and map the UNITY description to an architecture. The function of managing the open list is given to one processor while the rest of the processors find valid children. The data is decomposed by having the controller program send routes to be expanded to the worker programs.

All message communication occurs between the controller and worker nodes, not between any of the worker nodes. The following sections list the high-level pseudo-code for each of the processes.

3.4.7.1 Host Program. A program running on the host processor begins execution of the search and handles all input/output (I/O) between the user and the programs running on the parallel computer. The host program performs no work in the search process, its only function is to act as an interface between the other programs and the user.

```
LOAD Programs Onto Each Node
RECEIVE the Best Route From the Controller Program
RECEIVE Processing Information
DISPLAY the Best Route
PRINT Processing Information
KILL Programs on Each Node
```

3.4.7.2 Controller Program. The management of the open list is performed by a controller program. The controller removes routes at the front of the open list and sends them to an idle worker processor. As expanded routes are returned they are inserted into the open list based on their cost. This process is the assignment to the open list found in the UNITY statment

$$||\forall v : v = \text{child(BV)} :: \text{open} := (\text{BP} \wedge v)$$

from the previous section. Once the optimal route has been found

$$\text{found} := \text{TRUE} \quad \text{if BV} = \text{G}$$

the controller returns the solution, along with processing information to the host program. The criteria for determining the optimal solution has been found is an implementation issue. The processing information supports the analysis of this research effort.

```
INITIALIZE the Open List Queue

PLACE the Base Node Into the Open List Queue

UNTIL Optimal Solution Found

    SEND Route at Front of the Open List Queue to an Idle Processor

    RECEIVE Expanded Route Messages From the Processors

    INSERT Routes Into the Open List Queue

Enduntil


SEND the Solution to the Host Program

SEND Processing Information to the Host Program
```

3.4.7.3 Worker Program. The worker program performs the expansion of a node. It determines the children of a given parent, where the parent is the last location in the route being expanded. The worker determines whether a child is valid and for those that are valid g, h', and f' are calculated. Each of these new routes are returned to the controller program for insertion into the open list. The workers perform the following function from the UNITY description with the assignment to the open list being made by the controller node:

$$||\forall v : v = \text{child(BV)} :: \text{open} := (\text{BP} \wedge v)$$

from the previous section. This process of expanding routes continues until there is no more work to be performed, the optimal solution has been found, at which time the workers send some processing information to the host program. The processing information supports the analysis of this research effort.

```
UNTIL No More Work

    RECEIVE Route to be Expanded

    CALCULATE f', g, and h' for Each Valid Child
```

```
    SEND Expanded Routes to the Controller Program
Enduntil

SEND Processing Information to the Host Program
```

3.5 Tasks

Due to the time constraints imposed on the thesis research a complete and comprehensive software package to solve the mission routing problem is not to be realized. Therefore it is necessary to identify and prioritize the tasks associated with the development and implementation of the software package used to solve the mission routing problem. The following table (Table 3.1) contains all the identified tasks associated with the development of a mission routing system for this research effort. The tasks have been prioritized with a lower number indicating a task with a higher priority.

Table 3.1. Prioritization of Research Tasks

Priority	Task
1	Model terrain
1	Model detection
1	A* Search
1	Distance calculation
1	Radar detection calculation
1	Heuristics
2	Load Defense Mapping Agency (DMA) data
2	Load radar "danger map"
3	Develop a graphical user interface

3.6 Summary

This chapter presents a more detailed description of the mission routing problem than in the previous chapters. From this detailed description a list of primary and supporting requirements is extracted. Also discussed is the methodology used to design and develop a software package used to select routes. With the requirements and methodology defined a description of the high-level design

is presented. The next chapter provides the detailed design and implementation of the high-level design described in this chapter.

IV. Low-Level Design and Implementation

4.1 Introduction

The previous chapter explains the software design methodology and high-level design of the parallelized A* algorithm used to solve the mission routing problem. This chapter discusses the low-level design and implementation of the parallelized A* algorithm from the previous chapter. The low-level design is discussed in detail in section 4.2 while the implementation is discussed in section 4.5. The data structures used in the implementation of that design are discussed in section 4.3. The heuristics used by the A* search algorithm to estimate the cost of continuing searching along a given path are discussed in section 4.4.

4.2 Low-Level Design

4.2.1 English Description. As discussed in the Chapters II and III, the A* search strategy explores the route which appears to result in the overall best solution based on the additive cost function $f' = g + h'$, where g is the cost of reaching end of the route and h' is the projected cost of reaching the goal from the end of the route (46:75). The OPEN list contains all the routes, in ascending order of cost, which are being examined. Initially only the starting location is on the OPEN list. The route at the head of the OPEN list has the lowest projected cost and thus appears to give the optimal solution. For this reason it is removed from the OPEN list and explored further by having all the children of the parent (end location) found. These new routes are placed on the OPEN list in order of cost. This process of removing routes, finding children, and placing the new routes back on the OPEN list continues until the route at the head of the open list is a solution. The A* search algorithm UNITY description in the follow section shows the parallelism of examining routes and finding the valid children for each of these routes.

4.2.2 UNITY Description. The following is the UNITY description for the low-level design of the A* algorithm. The high-level design can be found in the previous chapter.

Program *A* Search*

declare

Start	: vertex	{Start Location (x, y, z coordinates)}
Parent	: vertex	{Parent Location (x, y, z coordinates)}
Child	: vertex	{Child Location (x, y, z coordinates)}
Goal	: vertex	{Goal Location (x, y, z coordinates)}
route	: array[1..N] of vertices	{Set of vertices forming a path}
BR	: route	{Best Route}
OPEN	: array[1..M] of routes	{OPEN list}
found	: boolean	{Flag Indicating if Best Solution Found}

always

{The OPEN list is sorted in ascending order using f values}

$\langle \| \forall i, \forall j \ : \ (0 \leq i < M) \wedge (0 < j \leq M) \wedge (i < j) \ :: \ f(OPEN[i]) \leq f(OPEN[j]) \rangle$

$\langle$Parent = route(tail)$\rangle$ {Parent is the last vertex visited in a given route}

initially

[] OPEN= {S} {Only the Start node is initially in the OPEN list}

[] found = FALSE

assign

$\langle$ BR := OPEN(head) if found = FALSE $\rangle$

[] $\langle$[]$\forall v$: v = Child(Parent(BR)) :: OPEN := (BR $\wedge$ v) if found = FALSE $\rangle$

[] $\langle$ found := TRUE if Parent(BR) = Goal $\rangle$

end *A* Search*

4.2.3 Mapping Schema. A combination of functional and data decomposition is used to map the above UNITY description to the Intel series of hypercube based parallel processing computers. The management of the OPEN list is viewed as running on a controller node, with the other nodes being used to find the children of a parent. Communication, during the search process, occurs only between the controller and worker nodes, not between any of the worker nodes.

The read-only schema is used to map the parallelized A* search algorithm to architecture. This mapping schema stipulates that each variable in a program can be modified by at most one processor (12:85). Each variable, in the UNITY description, is found only once on the left side of an assignment statement. Also, since the right side of each assignment statement names a single variable modified by another assignment statement this is an fine-grained read-only schema. In

mapping this to the actual implementation the following properties are evident, the best route (BR), OPEN list insertion, and FOUND variables appear on the left side of an equation only once. Thus the modification of each of these variables can be assigned to separate processors. The modification of the OPEN list is assigned to the controller processor which incorporates each of the three UNITY assignment statements. The worker processors expand a given route finding the valid children. For each route sent to a worker all the valid children are found and each new route is returned to the controller. Thus the worker performs the following function from the UNITY description with the assignment being made by the controller node:

$$\forall\, v : v = \text{Child}(\text{Parent}(\text{BR})) :: \text{OPEN} := (\text{BR} \wedge v).$$

Each worker program is given a separate best route (BR) to expand which is a data decomposition strategy. The controller program manages the open list and sends routes to the worker programs to expand. This separation of responsibilities is the characteristic of functional decomposition.

4.2.4 Pseudo-Code. If a software engineering approach were not used then the UNITY description would be "coded up" and a "code and fix" strategy would be used to get the software working. This is not the method stressed by software engineering principles since this can lead to improper, non-functional, or non-maintainable software. Before the UNITY description is coded a further functional refinement needs to take place. Pseudo-code is used to further refine and define the functions performed by each of the programs supporting the parallelized A* search algorithm. Also included is the pseudo-code for a sequential version of the A* search algorithm which was developed for comparison of execution times.

4.2.4.1 Host Program. A host program is loaded onto the hypercube's system resource manager (SRM), also known as the host processor. It is not derived from the UNITY description, but from the architecture. The host program handles all input/output (I/O) between the user and the programs running on the parallel computer. The host performs no work in the search process,

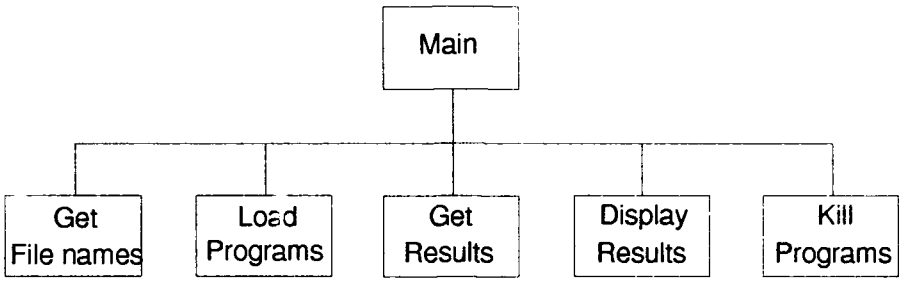

Figure 4.1. Structure Chart for the Host Program

its only function is to act as an interface between the other programs and the user. Before execution it is assumed that the user has allocated a cube, of any dimension, to run the software. The host program asks the user for the file names of the files containing the terrain, radar detection, and Air Tasking Order (ATO) data. The program then loads the controller and worker programs onto the proper nodes of the hypercube. The file names are then sent to each node and the host waits until a solution is returned along with processing information. Lastly, it removes the programs from each of the nodes and releases the cube back to the system. Figure 4.1 is a structure chart for the host program. It shows the functions performed and the dependencies between functions. The following pseudo-code shows the low-level design of the host program.

```
Get cube dimension
Read in file names
Load Nodes
  Load worker program on each node
  Kill program on node 0
  Load control program on node 0
SEND file names to each node
RECEIVE messages
  RECEIVE best route from the controller
  RECEIVE timing, # routes expanded, and efficiency from the controller
  RECEIVE work time and # routes expanded from each worker
Calculate the average worker program efficiency
Print the best route
Print timing, efficiency, and number of routes expanded
```

```
Kill the programs on each node
Release the cube
```

4.2.4.2 Control Program. The controller program acts as the manager of the open list. The controller receives the file names of the files which contain the terrain, radar detection, and Air Tasking Order (ATO) data from the host program. Only data from the ATO file is loaded so the controller has the starting and goal information to begin and terminate the search. The information in the other files is needed during the expansion of a node. The staging base is placed onto the open list queue. The open list is a priority queue based on cost (f'), thus the head of the queue contains the route with the lowest f'. As a worker program requests work the controller programs removes the route at the front of the queue and sends it to the worker. As a worker expands a node it returns routes which are inserted into the queue. As new best routes are found routes on the open list with a cost greater than or equal to the cost of the new best route are deleted from the queue. Thus, the queue only contains routes which appear to be better than the current

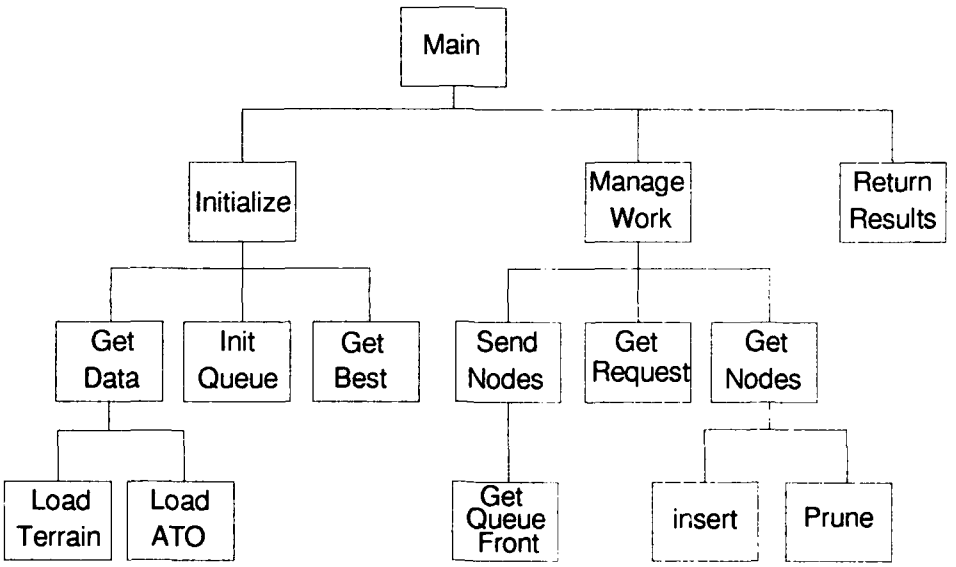

Figure 4.2. Structure Chart for the Control Program

known solution. Figure 4.2 is a structure chart for the controller program. This chart shows the major functions performed by the controller and the dependencies between each of the functions. The following is the pseudo-code describing the algorithm of the controller program.

```
RECEIVE file names from host
Load data from the Air Tasking Order file
Initialize the queue
Place the base location on the queue
Find an initial route to bound the search space
While (queue not empty) or (processors still working) or
      (expand route message waiting)
  While (expand route message waiting) and (queue not full)
    RECEIVE expand route message
    If (route.cost < best.cost)
      Insert route into the queue
      If (route is a solution)
        Set route to best route
        Prune the queue
      Endif
    Endif
  Endwhile
  While (work request message waiting)
    RECEIVE work request message
    Find which processor sent request message
    Set status of that processor available
    Subtract 1 from the number of processors working
  Endwhile
  While (queue not empty) and (queue's front.cost < best.cost) and
        (a processor is not busy)
    Set status of first available processor to busy
    Remove route from front of queue
    Add 1 to the number of processors working
    SEND expand route message to the processor
    Add 1 to the number of routes expanded
  Endwhile
Endwhile

SEND messages to the host
  SEND best route
  SEND timing information to the host program
SEND done message to all worker nodes
```

4.2.4.3 Worker Program. The worker program finds all the next valid locations which can be reached from a given partial route. The worker receives the file names of the files which contain the terrain, radar detection, and Air Tasking Order (ATO) data from the host program.

After the data is loaded the worker requests work from the controller program. Once work is received the children of the last location visited are found. A child is defined as those locations, in the three dimensional search space, which are adjacent to the given location. Adjacent is defined as being one grid location away, including in a diagonal line. The locations directly above or below are excluded. Thus for any given location there is a maximum of 24 neighboring locations (8 at the same altitude, 8 at the altitude below, and 8 at the altitude above). The assumption is made, for simplicity, that the change in altitude is the same as the change in distance at a given altitude. In other words the altitude resolution is the same as the terrain resolution. This assumption greatly reduces the complexity of the distance traveled calculations. The heuristic for h' is described in further detail in section 4.4.

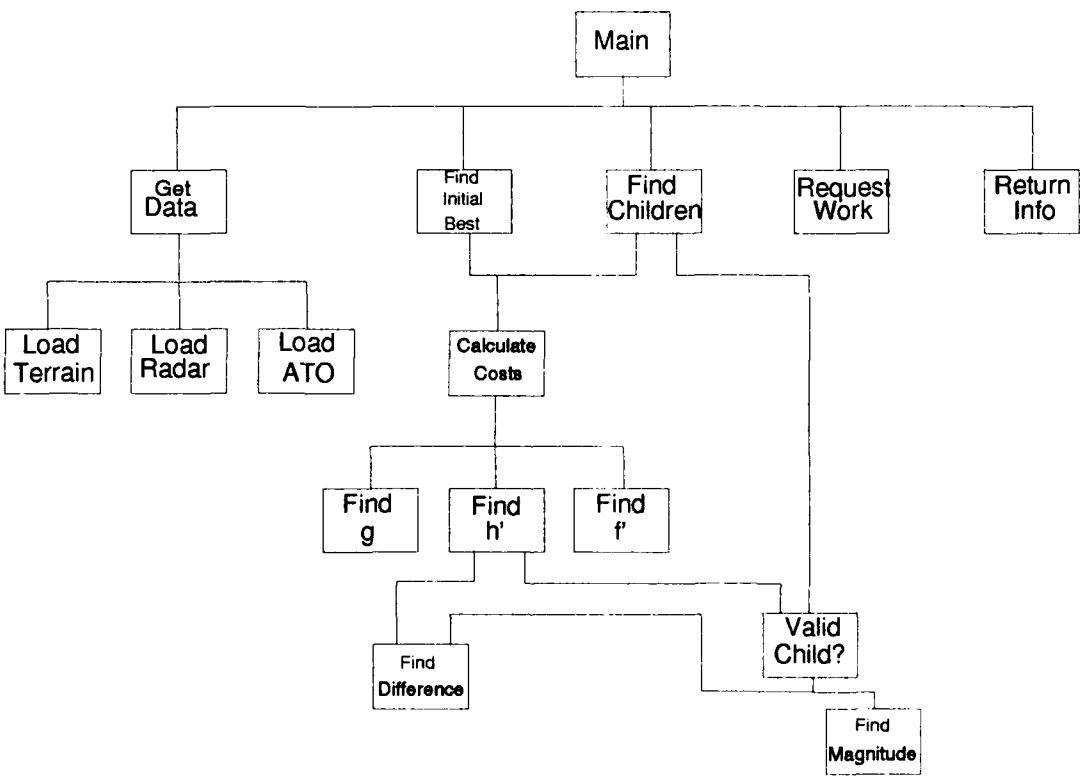

Figure 4.3. Structure Chart for the Worker Program

Each child was checked to ensure it was within the boundaries of the terrain data. As valid children were found, the costs (g, h', and f') are calculated and the partial route were sent back to the controller program for insertion into the open list queue. The worker programs continued requesting work and expanding partial routes until the controller program sent a message telling the worker programs to stop. At this point the worker programs sent timing and expansion information to the host program. Figure 4.3 is a structure chart for the worker program. This structure chart shows the major functions performed by the worker processors and and the dependencies between each functions. The following is the pseudo-code describing the algorithm of the worker program.

```
Initialize data
RECEIVE file names from host
Load data from the terrain file
Load data from the radar detection file
Load data from the Air Tasking Order file

SEND request for work to the controller

Loop Forever
  If (expand route message waiting) then
    RECEIVE expand route message
    Add 1 to number of routes expanded
    For altitude level below, equal and above present altitude
      For each child at that altitude
        If the child is a valid child
          Add child to the route
          Calculate f', g, h'
          SEND route to the controller
        Endif
      Endfor
    Endfor
    SEND request for work to the controller
  Endif
  If (done message waiting) then
    Break out of loop
  Endif
Endloop

SEND number of routes expanded to the host
SEND timing information to the host
```

4.2.4.4 Sequential Version. The algorithm for the sequential version is the same as for the parallel version except for the communication between programs. Whenever the parallel version sent messages the sequential version simply performed a call to the necessary routine. Thus the routines of the host, controller, and worker programs was combined into a single program which would execute on a single processor. Thus the descriptions and structure charts of the host, controller, and worker programs from the previous sections apply to the sequential version as well.

```
Read in file names
Initialize the queue
Initialize data
Load data from the terrain file
Load data from the radar detection file
Load data from the Air Tasking Order file
Place the base location on the queue
Find an initial route to bound the search space

While (queue not empty)
  Remove route from front of queue
  Add 1 to the number of routes expanded
  For altitude level below, equal and above present altitude
    For each child at that altitude
      If the child is a valid child
        Add child to the route
        Calculate f', g, h'
        If (route.cost < best.cost)
          Insert route into the queue
          If (route is a solution)
            Set route to best route
            Prune the queue
          Endif
        Endif
      Endif
    Endfor
  Endfor
Endwhile

Print the best route
Print timing, efficiency, and number of routes expanded
```

4.3 Data Structures

In examining the UNITY description of section 4.2.2 two main data structures are evident, one which is the open list and one which contains information about a route. The implementation of these two data structures is given in the following sections. Two other data structures are also described which are used to represent the terrain and detection information.

4.3.1 OPEN List. The open list is a priority queue based on the results of the additive cost function ($f' = g + h'$) associated with a route. Those routes with a lower cost are placed ahead of those with a higher cost which produces a priority queue. The C language data structure of the open list is

```
PATH    q[Q_SIZE+1];
```

which is simply an array of type PATH. The PATH data type is defined in the next section. The size of the queue is a parameter specified before compilation and the memory is allocated at load time. This data structure is chosen versus a true linked list because the code used to manage the open list had already been developed (22). Rewriting the open list manager is not within the scope of this research effort, though an evaluation of the usefulness of this data structure can be made.

4.3.2 Route Information. Information about a route is stored in the following C language data structure which is given the label PATH.

```
typedef struct
  {
   int      number;
   US       x [MAX_PATH_LENGTH + 1];
   US       y [MAX_PATH_LENGTH + 1];
   US       z [MAX_PATH_LENGTH + 1];
   int      vector_x;
   int      vector_y;
   int      vector_z;
   float    distance;
   float    radar;
```

```
  float     g;
  float     cost;
  int       link;
} PATH;
```

The arrays known as x, y, and z contain the coordinates of each location along the route and number indicates the number of locations, or entries, in the arrays. The length of the arrays is defined by MAX_PATH_LENGTH which is a parameter declared before compilation of this data structure. The direction the aircraft flew to reach the last location, from the next to last location, is stored in vector_x, vector_y, and vector_z. The total distance flown and accumulated radar detection cost are stored in distance and radar, respectively. The actual cost to reach the given location is stored in g while the projected final cost (f') is stored in cost. The variable link is used by the open list management routines.

Table 4.1. Memory Requirements For Data Types

Data Type	Memory Usage
US	2 bytes
integer	4 bytes
float	4 bytes
double	8 bytes

Table 4.1 shows the memory requirements for each type of C language data type. It is evident that an unsigned short requires have the memory as an integer variable. The restriction placed on unsigned shorts are that the range of valid numbers is 0 to 65,536. Since x, y, and z are indices into the terrain and radar matrices values less than 0 are not encountered. Thus, the only restriction is that the terrain and radar matrices not be larger than 65,536. The coordinate arrays (x, y, and z) are declared as unsigned short (US) versus integer to save memory. This data structure is used by the open list because the memory saved for each location stored is 6 bytes. If MAX_PATH_LENGTH were declared as 99 then 600 bytes would be saved. This may not be much, but then this data structure is used to define the open list. If Q_SIZE were declared as 1000 then 6600 bytes would be saved. The structure defined by Garmon used integer declarations (22),

but when it was modified to support this problem the amount of memory being allocated exceeded the processor's memory. Thus, unsigned short data types were needed. The directional vector displacement variables (vector_x, vector_y, vector_z) are declared as integers. Since the directional vector is a unit vector the displacements can take on the values -1, 0, and 1. Since negative numbers are possible the integer data type was needed.

4.3.3 Environment Representation. As discussed in the previous chapter the use of a grid representation is simple and straight forward. Also this is the representation used by the Defense Mapping Agency (DMA) digital terrain database (31:578)(38:172)(37:182-183).

4.3.3.1 Terrain Data. A two dimensional matrix was used to represent the terrain over which the aircraft was to fly. The indices (x, y) corresponded to the location and the entry in the matrix corresponded to the elevation at that location.

```
integer    terrain_matrix[MAX_MATRIX_SIZE+1][MAX_MATRIX_SIZE+1];
```

4.3.3.2 Detection Data. A three dimensional matrix was used to represent the space through which the aircraft was to fly. The indices (x, y) corresponded to the location and were the same as those used for the terrain data. The z index indicated the altitude of the location. Thus the coordinates (x, y, z) gave the location of the aircraft in the three dimensional space with the indices (x, y) being used to determine the elevation at that location.

```
float      radar_matrix[MAX_MATRIX_SIZE+1][MAX_MATRIX_SIZE+1][MAX_MATRIX_SIZE+1];
```

4.4 Heuristics

The heart of the A* search algorithm is the use of an heuristic to predict the cost of continuing along a given search path. The cost of arriving at a given location is known so the heuristic is used to determine the cost of continuing along the path to the solution. In order to guarantee that the

solution returned by the A* search is optimal the heuristic must be admissible. A heuristic function is admissible if

$$h'(n) \leq h(n) \quad \forall n \tag{4.1}$$

where $h(n)$ is the actual cost of going from the present node to the goal node (43:77). To limit the search space and guarantee an answer which is considered optimal it is necessary that h' be as close to h as possible without exceeding h. Also when creating the h' function, the designer must consider the trade-off between the time saved by reducing the search space and the time needed to calculate h'. If not careful it is possible that an h' is calculated which greatly reduces the search space, but which takes a great amount of time to calculate. In this case a simpler, less accurate h' could actually cause the search time to be reduced.

4.4.1 Weighting of Criteria. Before any discussion of heuristics used for this search process it is necessary to first define how the cost value is calculated. For this research only the distance flown and probability of detection are the criteria used to select routes. Since more than one parameter is used to determine the feasibility of a route the cost calculation uses both criteria. Weighting factors are applied to each criteria to allow a valid combination of criteria into a single value. The cost associated with travel between two points is given in the following formula:

$$\text{Cost} = (\text{Radar Weight} \cdot \text{Radar Cost}) + ((1 - \text{Radar Weight}) \cdot \text{Distance}) \tag{4.2}$$

where distance is the distance between the parent location and the child location, not the total distance of the route. The radar cost is defined as

$$\text{Radar Cost} = \text{Probability of Detection} \cdot \text{Distance}. \tag{4.3}$$

This means that for a given probability, at a location, the shortest distance will produce the smallest radar cost. This is true since the longer an aircraft is within a radar's coverage the greater the risk to the aircraft, all things being equal. The value for the radar weighting factor is stored in a header file which is incorporated into the software at compile time.

4.4.2 Straight-Line Distance. A very simple heuristic used for the A* search algorithm used to solve the mission routing problem is finding the distance between a given location and the goal location. Finding the distance between two points in a three dimensional space is given by the following formula (34:814):

$$\overline{|P_1P_2|} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2} \tag{4.4}$$

Since the shortest distance between two points is a straight line this would be an admissible heuristic. This heuristic requires little time to calculate, but it only uses one of the parameters to project a cost which may cause $h' \ll h_{actual}$. This argument holds if the weight of the distance parameter in the cost calculation is small with respect to the combined weights of the rest of the parameters. If, on the other hand, distance is weighted heavily, then h' will not be much less than h_{actual}. Thus use of this heuristic may not reduce the search space or result in the desired decrease of execution time. It may have little impact on the cost and thus the A* search is reduced to a simple best first search. A more accurate heuristic, employing all the parameters of the cost function, is desirable.

4.4.3 Recursive Search with Straight-Line Distance Calculation. As stated in the previous section a heuristic employing all of the parameters of the multicriteria problem is desirable. This type of heuristic more closely matchs the cost calculation than the previous heuristic and may give a result closer to the actual cost (h_{actual}), but it may require more time to calculate.

A limited recursive search is used to calculate a projected cost. From a given location a recursive search finding the minimum actual cost to a specified number of children is conducted.

At the end of the recursive search the straight-line distance heuristic described in the previous section is used to project the cost from the present location in the recursive search to the goal location. This recursive search uses the same algorithm as the actual expansion algorithm only modified to incorporate recursion. The following pseudo-code shows the algorithm employed.

```
If (recursion depth reached) then
  Calculate straight-line distance
  return distance to the calling routine
Endif

If (goal reached) then
  return 0 to the calling routine
Endif

For altitude level below, equal and above present altitude
  For each child at that altitude
    If the child is a valid child
      Add child to the route
      Find h' by recursing with this new partial route
      Calculate g to reach this location
      If f' < best f' at this level
        Set f' to best f'
      Endif
    Endif
  Endfor
Endfor
Return best f' to the calling routine
```

As can be seen from the pseudo-code the routine simply finds the minimum actual cost of traveling to a certain number of children away from the present location and then using the straight-line distance heuristic to estimate continuing along the route. This heuristic uses all the search parameters for part of the heuristic calculation and then the single parameter of distance for the rest. This heuristic is admissible since it combines actual costs along part of the projected route along with the straight-line distance heuristic, which is admissible, for the rest of the route. The recursive heuristic helps to distinguish between routes by "looking" down a route some distance. This does not guarantee that the route will be a good one since areas resulting in high costs may be beyond the range of the recursive search, but it does help to identify routes early in the search

process which may lead to high cost areas. As these types of routes are found they are given a higher h' and are placed further from the front of the open list resulting in the continued exploration of more promising routes. Thus the recursive search finds the local optimum cost from a given location at a given distance away. This heuristic is susceptible to the combinatoric explosion phenomenon since it explores, or actually searches, from a given location all the children to a specified depth. As the depth increases linearly the time to calculate the heuristic increases exponentially because of the exponential increase in the recursive search space.

4.5 Implementation

The system is decomposed into three programs. The host program is the interface between the user and the programs running on the nodes of the parallel computer. The controller program manages the open list used by the A* search algorithm. The last program, the worker, performs the expansion of each location in the search space. The following sections discuss each program in detail. A header file containing information such as the maximum size of the open list queue, maximum length of a route, and maximum sizes of the supporting matrices is included at compile time.

4.5.1 Host Program. The code for the host program is taken from that developed by Garmon, for his thesis, to solve the traveling salesmen problem (TSP) using a parallel A* search (22). The code was modified to ask the user for the file names of the files containing the terrain, radar detection, and Air Tasking Order (ATO) information. The program then loads the controller and worker programs, sends them the file names and waits for the results. Upon completion the program prints the solution route along with timing information for analysis purposes. A listing of the C language code for the host program can be found in Appendix A.1.1.

4.5.2 Control Program. This code is also taken from that developed by Garmon, for his thesis, to solve the traveling salesmen problem (TSP) using a parallel A* search (22). Some problems were encountered during execution which are discussed in detail in section 4.5.5.1.

As discussed in section 4.3.1 the open list is simply a two dimensional matrix where each column is of type route information (section 4.3.2). The link field is used to map the entries in the matrix to a linked list. As routes are inserted into the open list the link fields are adjusted to ensure the route's cost (f') keeps the open list in increasing cost order, a priority linked list scheme. Only routes whose cost (f') is less than the best are inserted into the open list since these routes appear to lead to a solution with a lower cost. As routes are inserted each is checked to see if it is a solution route, if so that route becomes the new best route and the open list is pruned. The pruning routine removes all routes whose cost is equal to or greater than the best route. This results in the best route also being removed from the open list, but its information had been stored in the best route variable which is defined to be of the route information type (section 4.3.2). The purpose for pruning the open list is to free up memory, or in this case free up columns in the open list matrix, since all routes following the best route, in the open list, are known to lead to worse solutions. The following pseudo-code shows the algorithms used to insert routes into the open list, remove the route from the front of the open list, and the routine used to prune the open list.

```
            PRIORITY INSERTION INTO THE OPEN LIST

If (open list is full) then
  Exit this routine
Else
  Get the next available location from the free list
  Adjust the pointer to the front of the free list
  Insert the route into the open list matrix
  If (open list was empty) then
    Set link of route to end of queue
    Set front of queue pointer to the column of the route inserted
    Flag it as busy
  Else
    If (cost of inserted route < cost of route at the front of the open list)
      Set link of route to column of front of open list
```

```
      Set front of queue pointer to the column of the route inserted
    Else
      Set found flag to false
      While ( (not at end of open list) and
              (cost of inserted route > cost of route in the open list) )
        If (the same state)
          Set found flag to true
          Exit the while loop
        Else
          Find next route in the open list
        Endif
      Endwhile
      If (found flag false) then
        Have route inserted point to column pointer to by previous route
        Have previous route in list point to route inserted
      Else
        Set front of the free list back to column of route inserted
      Endif
    Endif
  Endif
If (found flag false) then
  Increment the length of the open list
  Increment number of insertions counter
Endif

If (free list is empty)
  Perform a beam search reduction on the open list
Endif
```

REMOVE THE FRONT ROUTE FROM THE OPEN LIST

```
If (open list is empty) then
  Exit this routine
Else
  Copy the route at the front of the open list to a temporary area
  Set front of the open list to the route pointed to by route removed
  Decrement the length of the open list
  Set link of column of the route removed to the front of the free list
  Set front of the free list to the column of the route removed

  If (open list was full) then
    Flag it as busy
  Else
    If (open list is now empty) then
      Flag it as empty
    Endif
  Endif
  Return the route removed to the calling routine
Endif
```

```
            PRUNE THE OPEN LIST

If (open list is not empty) then
  If (cost of the first entry in the open list >= cost of best route) then
    Add all columns in the open list to the free list
  Else
    While ( (cost of route in the open list < cost of best route) and
            (link of route in the open list is not an end of queue marker) )
      Find the next route in the open list
    Endwhile
    Reset the length of the open list
    If (link of route in the open list is not an end of queue marker) then
      Set previous route's link to end of queue
      Set route's link to front of free list
      Set front of free list to column of route
      Flag the open list as busy
    Endif
  Endif
Endif
```

Normally the termination criteria for the A* search is when the route removed from the front of the open list is a solution route. This is not the case in this implementation. Since the pruning routine removes all entires in the open list with a cost equal to or greater than the best solution route, the termination criteria for the search is when the open list was empty. This approach allowed the prune routine to operate regardless of the order in which routes of equal cost are inserted into the open list. Also since a separate data structure is used to store the best solution, space is saved in the open list by not keeping the solution in both its own data structure and the open list. It is necessary to insert the best solution into the queue before pruning, or ensure the best solution is not inserted. Pruning of the open list takes time, which increases the execution time of the controller program, but this is small compared to the overall execution time and it is the method employed to determine the termination of the search.

The open list is a priority linked list. The routes are inserted in ascending order of cost and links are established pointing to the next route in the list. A matrix representation of the linked list is used versus a true linked list. The code developed by Garmon used the matrix data

structure so this was one reason in selecting the matrix version. Also with the matrix version the amount of memory usage is known at execution time versus memory being allocated during run time. This does cause the open list to have a fixed size, but it does eliminate a possible out of memory run time error. It also eliminates the time associated with allocating and deallocating memory when managing the open list. This type of approach simplifies the algorithms used to perform the management functions.

Since the size of the open list is fixed, a method is needed to ensure space would be available once the open list became full. A beam search reduction strategy is used to reduce the size of the open list once it becomes full. In a beam search only x number of solutions at the front of the priority queue are kept and the rest are discarded. This strategy is employed whenever the open list becomes full thus freeing up space in the open list. Because routes are discarded, not based on cost with respect to a solution, this method is not guaranteed to return an optimal solution. If the open list is very large and a small percent of the routes are discarded then the probability that a partial route leading to an optimal solution is discarded is reduced.

In order to bound the search a solution route was found. The controller simply requested the worker to find a solution, not necessarily the optimal. The cost of this solution was used as the best cost when inserting routes into the open list. The details of the procedure used to find this initial solution are discussed in the following section. A listing of the C language code for the controller program can be found in Appendix A.1.2.

4.5.3 Worker Program. The worker is the heart of the search for the best mission route and the search space is different from that used by Garmon, thus this program is designed and implemented without using any of Garmon's worker program.

The worker program finds all the next possible locations from a given location. A set of rules, or constraints, are used to determine whether a child location can be reached from the parent location. The rules, or constraints, used to determine if a child is valid are:

1. the aircraft must be some minimum altitude above the ground and below its maximum altitude (ceiling),
2. the aircraft must be above a specified altitude and below a specified altitude,
3. the distance flown must be less than the aircraft's combat radius, and
4. the child location must be reachable from the parent location by the aircraft based on the direction of flight. In other words a child location could only be reached if it caused the aircraft to perform a turn which was greater than the minimum turn radius of the aircraft.

To determine if an aircraft has not exceeded it minimum turn radius and likewise climb/dive angle manipulation of directional vectors is used. A single variable, known as field-of-view, is used to represent the greatest change in direction allowed for the aircraft. For simplicity the field-of-view is defined as the maximum change in direction allowed regardless if the change was caused by a climb, dive, turn, or a combination of them. As seen in the route information data structure (section 4.3.2), a directional vector is kept which indicates the direction flown by the aircraft to reach the parent location. A directional vector from the parent location to the child location is then calculated. The following formulas show the method used to find the angle between two 3-dimensional vectors. The dot product of two vectors is given by (34:826)

$$\mathbf{A} \cdot \mathbf{B} = |\mathbf{A}||\mathbf{B}| \cos\theta \tag{4.5}$$

and also by (34:825)

$$\mathbf{A} \cdot \mathbf{B} = a_1 b_1 + a_2 b_2 + a_3 b_3 \tag{4.6}$$

Setting the two equations equal and solving for θ yields

$$\theta = \arccos \frac{a_1 b_1 + a_2 b_2 + a_3 b_3}{|\mathbf{A}||\mathbf{B}|} \tag{4.7}$$

If this angle (θ) is less than or equal to the field-of-view variable the child location is considered reachable from the parent location, thus the child is valid. Otherwise the child is not valid and it is rejected. When moving from one location to another it is guaranteed that the magnitude of that direction vector is not 0. The one time when a direction vector is 0 is when beginning the search. This is because no movement is associated with the beginning state, thus the directional vector's magnitude is 0. A check is made when validating children and if the magnitude of the vector to reach the parent location is zero then the child is valid. If an attempt were made to calculate an angle is this situation a divide by zero error will occur. For all other times both vectors, to reach the parent and from the parent to the child, will be greater than 0 and the calculation can be made.

To bound the search initially, a solution is found which is not necessarily an optimal solution nor is it guaranteed to be a valid solution. The controller sends a message to the worker program containing the beginning location. The routine used to find the initial route tires to find a route which is a straight line between the beginning and ending locations. The only requirement placed on the route is that each location has to be some minimum distance above the ground. If the location selected does not meet this constraint then the location at the same (x, y) coordinate of the child location, but the next higher altitude is checked. If this location meets the constraint then a new straight line is calculated and the search continues. If the location still does not meet the constraint then the (x,y) coordinate of the parent and the next higher altitude (i.e. the aircraft is traveling vertically) is made the next location. This procedure continues until the goal is reached. To compensate for vertical travel the distance traveled is doubled. Since it is possible for the aircraft to travel in a vertical direction the resultant route is not guaranteed to be valid. The following pseudo-code shows the procedure used to find the initial route. This routine is implemented by converting the pseudo-code into C language code. To reduce the message traffic only node number 1 returns the initial solution.

```
If (goal reached) and (I'm node 1) then
```

```
  Return the route to the controller program

Else
  Calculate the vector from the parent to the goal location
  Find the child in this direction

  Calculate the minimum flight altitude at this location
  Determine the actual altitude of the child

  If (actual altitude < minimum altitude)
    Calculate the costs to reach the child
    Place the child location in the route
    Recursively call this routine using this new route
  Else
    Set the found flag to false
    While (change in altitude < 1)
       Increment the change in altitude
       Find the child in this direction
       Determine the actual altitude of the child
       If (actual altitude < minimum altitude)
         Calculate the costs to reach the child
         Place the child location in the route
         Recursively call this routine using this new route
         Set the change in altitude to greater than 1
         Set the found flag to true
       Else
         Make the child directly above the parent location
         Calculate the costs to reach the child using double the distance
         Place the child location in the route
         Recursively call this routine using this new route
       Endif
    Endwhile

Endif
```

The recursion heuristic described in section 4.4.3 is implemented by simply converting the pseudo-code into C language code. The depth of the recursion is defined in the header file. This parameter of the recursion search is used to change the granularity of the worker program. Granularity is the computational time versus the communication time. By increasing the recursion depth the computation time is increased while the communication time remained the same, thus the granularity of the worker program increases. Likewise the granularity can decrease by decreasing the recursive search depth. A listing of the C language code for the worker program is found in Appendix A.1.3.

4.5.4 Sequential Version. The implementation of the sequential version of the parallelized A* search is straightforward. The controller and worker programs are combined into a single program. Whenever communication between processes occurs, the message call is replaced with a function call. This effects the worker little, but the controller functions were modified to remove checking for processors requesting work and waiting messages. The controller's main loop is reduced to simply removing and inserting routes into the open list, along with determining when the search terminates.

The reason the host is include is so this sequential version can run on the hypercube, thus allowing a direct comparison between the sequential and parallel versions. Thus the same host algorithm used for the parallel code can be used for the sequential with the minor change of loading the sequential code on a node versus loading the controller and worker programs. A listing of the C language code for the sequential version, along with the host program, of the parallelized A* search is in Appendix A.2.

4.5.5 Problems Encountered. Implementation of the parallelized mission routing software system was not without its problems, some were design errors and others were system and language implementation errors. The following sections discuss some of the problems encountered during the implementation phase.

4.5.5.1 Termination of the Search. A problem occurring in the determination that the search process had ended. In the original controller program termination of the search occurred when the OPEN list was empty and all the processors were requesting more work. This is a logic completion condition, but there is a possible timing error which manifested itself during execution. The condition occurred when the system began processing. Each processor would initialize itself (i.e., load data and initialize its variables) and then request work from the controller process. The controller would remove the first and only route off the OPEN list (the starting location) and send it to a node. The way the nodes are selected results in node 1 always being selected in this case. Node

1 would read in the route to be expanded, find all the children and send them back to the controller, then request more work. If the controller read in the request for more work from node 1 before it read in the messages containing the children then the controller had an empty OPEN list with all processors requesting work which was the termination condition, so the program terminated. A simple condition was added to the termination condition which checked to see if any children messages were waiting to be read in by the controller program. This check fixed the problem and the system operated correctly.

Another logic problem encounter in the management of the open list was the use of the prune routine. As stated earlier the usual termination criteria for the A* search is when the route removed from the front of the open list is a solution. Thus all subsequent routes are of equal or greater cost meaning all other solutions are also of equal or greater cost. When the parallel version of the code was converted to the sequential version the program would not terminate correctly since the solution route is inserted after the queue has been pruned. This problem is easily remedied once the termination criteria are clearly understood.

4.5.5.2 State Duplication. Another problem encounter with using the existing open list manager for the mission routing problem was that of duplicate states being entered into the open list. A state is defined by not only the parent location, but also by the directional vector. If two routes have the same parent location, but different directional vectors then they are different states since different sets of child locations are possible. If the two routes have the same parent location and same directional vectors then the child are the same and the routes are in the same state. Thus it is possible that routes can be in the same state, regardless of the actual route leading to the location previous to the parent location. If duplicate routes were allowed to exist then the number of routes in the open list would get very large and workers would actual duplicate worker, thus waste processing resources. During the insertion process it is a simple check to see if a state already existed in the open list. As the insertion routine moves through the pseudo linked list

examining the cost a check is added to compare the parent and directional vector of the route being added to the route being examined in the open list. If the routes are in the same state then the insertion routine terminates, not inserting the new route into the open list. This process does not remove a same state whose cost is greater than the one being inserted. This simple check reduces the number of routes being entered into the open list and eliminates duplicate work performed by the worker programs.

4.5.5.3 System Message Buffer. After an hour of processing the system simply hung. Each of the worker programs halted and the controller continued waiting for messages which would never arrive. Thus the system was in a form of deadlock. It was assumed that the open list matrix was getting full then causing the system to hang. The software is supposed to terminate processing whenever the open list gets full, but this did not seem to be the case. The problem was debugged by making the open list matrix very small and monitoring the insertions, removals, and length of the open list. It was observed that once the open list gets full the controller program removes a route and inserts a single route. The way the open list management algorithm has been implemented a message is read if there is room in the open list. The next check is to see if a worker program has requested more work. When the open list got full the controller ended up waiting until a worker requested more work, thus causing the routes coming back from the worker to be left in the system message buffer until there is room in the open list. Once a route is removed from the open list and sent to a worker program, a route in the system message buffer is removed and the route inserted into the open list. This process causes the system message buffers to get full. Once the operating system detects a system message buffer is full the processor trying to send another message is halted and once halted it remains in that state until the cube is released back to the system. This is not a new problem as it had been uncovered by Work and others (50:137-138). The method employed to work around this system error is to implement a beam search reduction scheme which

was discussed in section 4.5.2. This frees up space in the open list thus allowing the controller to continue receiving more than a single route for each route sent for expansion.

4.5.5.4 Type Checking. Unlike the Ada programming language, the C programming language provides no means of performing data type checking. This can lead to problems when working with data types in a single assignment statement. This problem may or may not manifest itself since it could be architecture dependent. This problem caused errors when testing on an architecture other than the hypercube. The matrix indices were declared to be unsigned short integers to reduce memory requirements and still provide for the range of numbers needed. An unsigned short variable requires only 2 bytes of memory while a variable declared to be an integer requires 4 bytes of memory. Using an integer declaration to read in variables declared as unsigned shorts produced no errors on the hypercube, but this was not the case on the sun workstations. This was due to the bit numbering scheme employed by Intel which is the reverse of that used by Sun. Different computers handled the error differently. The Sun3 workstation simply truncates 2 bytes, which are the bytes where the unsigned short data is located, and uses the other two bytes which contain zeros. The Sun SPARCstation (Sun4) aborts execution with a bus error message. Simply using an unsigned short declaration in the read statement solves the error. Programmers must remain aware of the potential for errors not only when reading in information, but also when working with multiply data types.

4.5.5.5 Round-off Error. When testing the calculation of the angle between the two directional vectors invalid results were obtained at the end points of the arc cosine function returned values (i.e. when the angle was either 0 or π radians). The acos function requires the input parameter to be of the type double, so any floats were cast as type double. This seemed to solve some of the errors encountered, but not all of them. Sometimes the arc cosine function returned a correct answer for 0 and π radian angles and sometimes an error was returned. A careful examination of each of the parameters revealed that when the magnitude of each of the

directional vectors was $\sqrt{2}$, a correct answer for the arc cosine calculation was returned, but when the magnitudes were $\sqrt{3}$ then errors were encountered. It was at this point that the calculations for the numerator and denominator (for the arc cosine input parameter) were separated and a print declaration of %18.20f was used in the print statement. It was found that the square root function returned a value with a small error. When performing $\sqrt{3} \cdot \sqrt{3}$ the value 3 was not returned, but the value 2.99999 which caused the error to occur in the arc cosine function. This was not the case when performing $\sqrt{2} \cdot \sqrt{2}$ as the result was equal to 2. Since the input parameter to the arc cosine routine is $-1 \leq x \leq 1$ a value out of this range indicates an overflow/underflow occurred during the calculation and the value needs to be set to the correct value. This check ensures the arc cosine routine is sent a valid value.

4.6 Summary

This chapter described and provided detailed examples of the date structures used by the parallel A* search algorithm designed to solve the mission routing problem. The high-level design from Chapter III was further refined adding communication, architecture specific, and implementation details. Supporting functions were also described in detail. Also presented was a discussion of design and implementation considerations made. The next chapter presents the mission scenarios used to validate the software and the results and supporting analysis of the implemented parallel A* search algorithm.

V. Experimental Testing, Results, and Analysis

5.1 Introduction

This chapter examines the results of executing the system designed and implemented in the previous chapters. Before an analysis of the performance of the software can take place it is necessary to first define the metrics which are used. These definitions are presented in section 5.2. The experimental input data are given in section 5.3. The experiments are described in section 5.4. The results obtained and analysis are found in section 5.5.

5.2 Metrics

In order to determine the viability of using a parallel computer to solve the mission routing problem it is necessary to establish some means of analyzing the performance of the software. Number of nodes expanded, execution time, program efficiency and speed-up are the metrics selected. Each of these metrics are discussed in detail in the following sections.

5.2.1 Nodes Expanded. The number of nodes expanded is defined as the total number of nodes sent to the workers to have children found. As discussed in previous chapters, the goal of a heuristic search is to reduce the search space which in turn minimizes the time needed to find a solution. Since the search is performed in parallel not only is the best appearing route explored, but also the x other best appearing routes. For instance, with 7 worker programs the best 7 routes are sent to worker programs for expansion. In a sequential version it is possible that 6 of those routes would never be expanded if the cost of the routes produced from expanding the best route are less than any of the other 6 routes. Since routes are expanded in parallel the likelihood that routes not leading to an optimal solution are also explored is increased. Thus, the number of nodes expanded is a "good" indication of the effectiveness in reducing the search space though in a parallel processing environment this metric can not be used alone. It can be used to compare different types of heuristics. As long as the number of nodes expanded is used for comparisons for

the same size parallel computer then it is a true measure of the search space. If this metric is used to compare different sizes of parallel computers then it is only an indication because of the amount of "extra" work taking place.

Another use of the number of nodes expanded metric is determining load balancing efficiency. The concept of load balancing is discussed in section 2.2.2.3. In this case, number of nodes expanded refers to the number of nodes expanded for any particular processor in the parallel computer and not to the total nodes expanded for the entire search process. If the system is effectively load balanced then each processor should expand roughly the same number of nodes. This is not a metric to determine the "true" load balancing effectiveness since processors may not spend the same amount of time expanding nodes. For instance if one route being expanded produced 5 children routes and another produced 3 children routes then it is expected that the processor producing 3 children routes would request work sooner thus expanding more nodes. But on the average it is expected that each processor would expand roughly the same number of nodes. This discussion does depend on the algorithm used by the controller processor to handle messages. The controller not only reads in expanded routes, but also sends routes to workers for expansion and receives request for work from the worker processors. Thus this metric may not give a "true" indication on load balancing effectiveness, but it can give a "good" indication especially when used with the average worker efficiency metric discussed in section 5.2.3. The number of nodes expanded per individual processor is not used directly for the analysis, but the data is contained in Appendix C; however, this information is used during implementation and testing of the software to monitor system usage and to detect possible communication problems between the controller and worker processors.

5.2.2 Execution Time. Execution time is ultimately the most important metric since this is what the user is most concerned about, other than the validity of the results. The main requirement of the user is to reduce the execution time so that solutions are found in a "reasonable time". Also as stated in the next section, in a parallel processing environment work actually performed by each

processor has a bearing on the execution time, but for a search the major factors are program efficiency and search space. By increasing program efficiency and reducing the search space the result should be a decrease in the overall execution time. Time can be used to compare different search strategies, number of processors used, and the effect of changing architectures.

There are a number of times associated with a program and the timing information desired depends on the problem being explored. For instance if a new piece of architecture were installed then the time to access that piece of equipment would be important. This research is looking at the overall execution of the software and the effects heuristics and architecture have on that time. For this investigation the time metric is categorized into:

- Time to load the data from the inputs files (terrain, radar, ATO) and initialize any variables
- Time spent finding the initial route used to bound the search space
- Time spent searching for a solution
- Total execution time

The first two timing metrics should be independent of the number of processors used since each processor is identical and there is no interaction taking place during this portion of the software. Thus as the number of processors is changed it is expected that these values will remain the same. They can be used during testing of the software to monitor system performance, but they are not necessary for the overall analysis. The time spent searching for a solution is the total time from when the search is begun until it is terminated, including any idle time. The total execution time is simply the summation of the other three time metrics and since the first two should be "relatively constant" no additional information can be derived on the effect the heuristics or the architecture have on the execution time. Therefore, the time spent searching for a solution is the main metric needed for analysis.

5.2.3 Program Efficiency. For this investigation, program efficiency is defined as the ratio of time spent performing work versus the run time of the process. For the worker programs it is the time spent expanding nodes versus the total search time. This is given in the following formula:

$$\text{Worker Efficiency} = \frac{\text{work time}}{\text{search time}} \tag{5.1}$$

where work time is defined as the time actually spent by a worker program receiving a route, expanding the route, and sending each of the generated new routes to the controller program. Search time is the total time spent by the individual processor in the search phase including idle time waiting for work requests to be fulfilled. This should be the same for each processor, since all start and end at the same times. The average worker efficiency is given by

$$\text{Average Worker Efficiency} = \frac{\sum_{i=1}^{\text{cube dim}} \text{Worker Efficiency}_i}{\text{cube dim} - 1}. \tag{5.2}$$

If the workers are always busy expanding nodes then the average worker efficiency is equal one, but then this number can not be one because of the overhead associated with communication between the worker and controller programs. As the average worker efficiency approaches zero the workers are spending more time waiting for the controller program to respond to their request for work than they do performing actual work. This metric is a good indication of the communication bottleneck at the controller node.

Likewise, the efficiency of the controller program is defined as the time s pent performing management of the open list versus the total search time.

$$\text{Controller Efficiency} = \frac{\text{work time}}{\text{search time}} \tag{5.3}$$

The work time is defined as the time the controller actually spent managing the OPEN list and the search time is the same as for the worker efficiency. If the controller is always busy inserting to and removing information from the open list then this ratio would be equal to one. As with the worker program this ratio can not equal one because of the communication overhead. As this ratio approaches zero it indicates that the worker programs may be spending too much time expanding a route. The desire is for the worker programs' efficiency to be near one while at the same time the controller program's efficiency to also be near one. Neither can be one because of communication and processing overheads. Changing the granularity of the worker program effects the efficiency of the controller which in turn impacts the workers' efficiency. There are interdependencies between the controller's and workers' efficiency. This information in conjunction with the execution time can be used to "fine-tune" the time each worker spends expanding a given route and help to reduce the total execution time by matching the granularity of the problem to the parallel system being used.

5.2.4 Speed-up. The commonly used terms used to describe the performance of parallel processing systems are speed-up and efficiency (45:20). The advantage of parallel computers is their ability to possibly decrease the execution time of software. Speed-up is a measure of improvement which indicates the advantages achieved when running software on a parallel computer versus a single processor computer. Speed-up is defined by

$$S_p = \frac{T_1}{T_p} \tag{5.4}$$

where S_p is the speed-up achieved on a parallel computer with p processors (45:20). T_1 is the time running the algorithm on a single processor and T_p is the time running the algorithm on a p processor parallel computer.

Efficiency is a measure of the speed-up per node realized. Efficiency is given by (45:21)

$$E_p = \frac{S_p}{p}. \tag{5.5}$$

These two metrics provide a means of analytically measuring the effectiveness of the parallel implementation of the software.

5.3 Input Data

The following sections describe the terrain, radar detection, and Air Tasking Order (ATO) data used to test and analyze the performance of the parallel software.

5.3.1 Terrain. A representation of the terrain is shown in Figure 5.1. The terrain is developed so as to have a variety of features such as mountains, hills, ridges, valleys, and relatively flat areas. This terrain is used so as to be a general representation of the type of terrain a pilot might fly over. It also provides more complexity in the search space than a relatively flat terrain. The contour lines represent each 1000 feet in elevation. A two dimensional grid overlay, represented by the dashed lines, is placed over the terrain map. Elevation values for each location are then estimated and stored in an ASCII file. This terrain provides the software with a semi-realistic search space. The contents of the terrain file are contained in Appendix B.1. This information is then read in by the worker program and stored in the data structure described in section 4.3.3.1.

5.3.2 Radar. The radar detection is also modeled using the grid method. In this case, a three dimensional matrix is used with the third dimension corresponding to the altitude above sea level. The model is created by taking the map used to create the terrain data and selecting locations for radar sites. The radar sites are indicated by the large black dots. To represent the radars' coverage, circles are drawn whose centers are located at the radar sites. Figure 5.1 shows the approximate radar coverages. These are approximate because of the limitations of the package

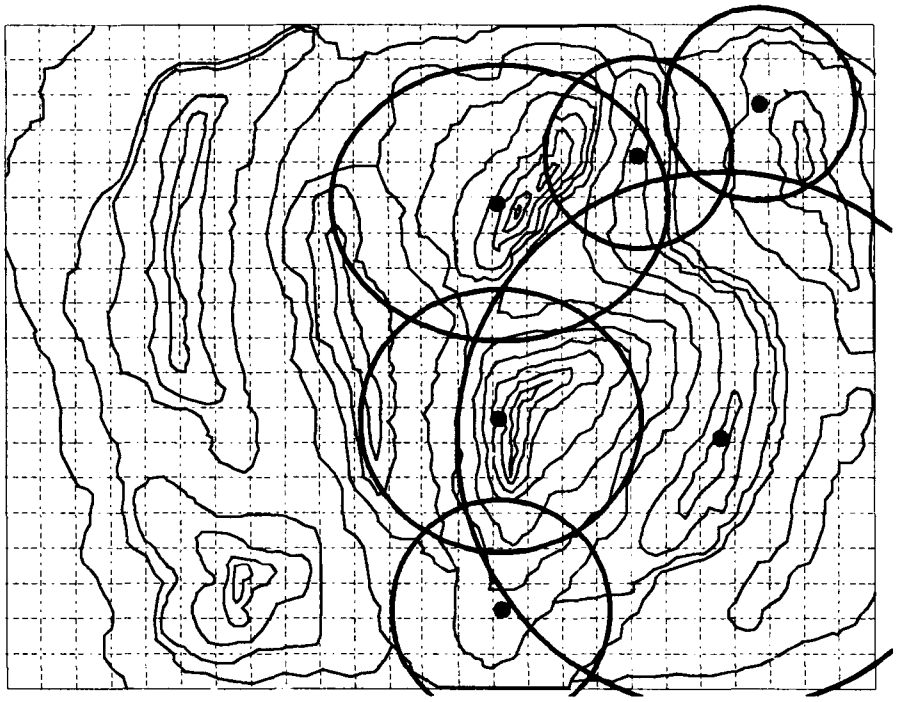

Figure 5.1. The Terrain With Radar Coverage Used

used to produce the figure. Probabilities of detection are then assigned to each location within the three dimensional representation. The probabilities are based on the center, or site's location, having a probability of 1.0 and the locations outside of the circle being 0.0 and the values projecting from the site linearly decreasing. Terrain features effect a radar signal. For instance if a radar site is located at the base of a mountain, then the radar can not detect anything behind the mountain since the mountain blocks the radar's signals. The assumption being made is that tactical radars are modelled, since strategic radars have over-the-horizon capabilities. Terrain masking effects are figured into the estimation of detection values. Areas where overlapping radar coverage occurs the highest probability is assigned to that location. The coverage from one altitude to another is modeled as a cylinder, thus values remained constant when changing altitudes except for those locations where the terrain blocks the field of view of a radar site. The probability of detection values for each location were placed into an ASCII file. The contents of this file are contained in Appendix B.2. Only a portion of the file is given (up to 15,000 feet) because the detection was

modeled as a cylinder and the values at the rest of the altitudes are exactly the same. The worker program reads in this data into the data structure described in section 4.3.3.2. This detection model is simplistic, but it does provide a semi-realistic threat environment. Radar modelling is itself a field of study and any accurate modelling is beyond the scope of this effort.

5.3.3 Air Tasking Order. The Air Tasking Order (ATO) contains information necessary to identify and plan the mission. Information entered into the system is the mission designator, starting location, location of the target, minimum height above ground, and the altitude block assigned to the mission. The altitude block prescribes the minimum and maximum altitudes at which the aircraft may fly. This information is also stored in an ASCII file. Three ATO files are used to test the software. ATO file AFIT-GOA has the starting location at coordinates (1, 1) and the target at (24, 16). This means a route goes from one corner of the grid to the nearly the other, resulting in a larger number of possible routes which need to be searched. Mission AFIT-1A has the same starting location, but the target is at location (17, 17). This reduces the distance between starting location and target. The last ATO file, AFIT-0A, has the starting and target locations close together. This file is used to test the determination of valid children (section 5.5.2). The ATO files used are found in Appendix B.3.

5.4 Experiments

The purpose of this research is to determine the feasibility of solving the multicriteria mission routing problem using a parallel processing environment. As stated in the first chapter, this research is focusing on the following four areas of the mission routing problem:

1. representation of the threat environment,
2. decomposition of the A* search algorithm for use on a parallel processing system,
3. application of heuristics to reduce the search space and the execution time, and

4. effects of parallel computer architecture on the execution time.

The first two areas were addressed during the design phase of this research effort and are discussed in detail in chapters III and IV which cover the design of the software. The last two areas deal with the effects each area has on the execution time. The following sections discuss the experiments and the associated analyses in these two areas. Before the experiments are discussed the test plan is presented.

5.4.1 Test Plan. A simple and straight forward test plan is used to perform each of the experiments. The input data described in section 5.3 is used for the experiments. In order to make comparisons and draw valid conclusions the number of parameters changed is limited to one. Thus for each experiment only a single parameter is altered. Each of the parameters altered are:

- vary the search algorithm by removing the bounding function,
- vary the cube size (change the number of worker nodes),
- vary the granularity of the problem, and
- vary the architecture (use a iPSC/2 and a iPSC/860).

The execution time is a function of the search strategy, with all other things constant. The two strategies are with and without a bounding function. Executing each strategy using the same input data and computer configuration will indicate the effect each has on the overall execution time.

Changing the cube size will provide information into the effects the number of worker nodes has on the performance of the system. Increasing the number of nodes means that more work can be done simultaneously, but it is possible that more "wasted" work will be performed. Changing this parameter of the system will also exercise the controller's ability to manage the open list efficiently and the impact on the workers' efficiency.

A recursive routine is used to calculate h'. The worker programs communicate to the controller each time a child is found and its costs (f', g, and h') are calculated. Thus if the recursion depth is increased the time between communications is increased. By modifying the depth to which the routine searches the granularity of the problem is changed.

Comparing the results obtained on the iPSC/2 and iPSC/860 will provide insight into the effects the architecture (i.e speed of CPU and communication channels) has on the performance of the mission routing software. These two computers were selected because as discussed in Chapter I (see section 1.6) they were made available to support this investigation. There are other types of parallel architectures such as a mesh, pyramid, multistage networks, and "fat trees" (23:1829)(51:29), but access to machines with these types of topologies was not available.

5.4.2 Reduction of the Search Space. A strategy is designed and implemented to reduce the search space and thus the overall execution time. As discussed in the previous chapter (see section 4.5.3), the worker programs find an initial route, though not guaranteed to be valid, and returns this to the controller. This route becomes the known best solution. The controller program compares the cost of a route being inserted into the open list to the cost of the best solution and only inserts the route if its cost is less. This only reduces the routes being inserted into the open list and not the search space itself. To reduce the search space an altitude block is specified for the mission in the ATO. This altitude block specifies the minimum and maximum altitudes at which an aircraft may fly for the mission. This mission restriction was employed during Operation Desert Storm because of the large numbers of aircraft in the same vicinity at the same times and not to reduce the search space (21). All this computation to bound the search space requires time, thus there is a trade-off between the time spent bounding the search and the time spent exploring "bad" nodes. To investigate this trade-off the algorithm without any bounding is executed as well as the same algorithm with the bounding constraints.

5.4.3 Parallel Architecture. In a parallel processing environment one can not only change the architecture upon which the software is executed, but also the number of processors used to solve the problem. This second point deals with the concept of granularity. For a discussion of granularity see section 2.2.2.2. Experiments were developed to investigate both of these concerns.

5.4.3.1 Different Machines. To test the effect of changing architectures the software is run on two hypercubes, each based on a different processor. An Intel 8 processor iPSC/2 and an Intel 8 processor iPSC/860 are used. Table 5.1 (6:10,16) shows the characteristics of each of these parallel computers. The communication network is the same for both architectures as are the

Table 5.1. Characteristics of the iPSC/2 and iPSC/860

	iPSC/2	iPSC/860
CPU	Intel 80386 DX	Intel i860™
Math Coprocessor	Intel 80387	Intel 860 internal fp
Clock	16 MHz	40 MHz
Operating System	Host: AT&T UNIX, System V	Host: AT&T UNIX, System V
	Node: NX/2	Node: NX/2
Memory	Host: 8 Mbytes	Host: 8 Mbytes
	Node: 12 Mbytes	Node: 16 Mbytes
Number of Nodes	8	8
Cube Network	Direct-Connect™ Routing 2.8 Mbytes/sec. bandwidth	Direct-Connect™ Routing 2.8 Mbytes/sec. bandwidth

operating systems on both the host and node processors. The iPSC/860 uses a reduced instruction set computer (RISC) processor (26:1-1) while the iPSC/2 uses a complex instruction set computer (CISC) processor (28:5-381 thru 5-394), the same processor found in many personal computers (PCs). This along with the clock speed effect the execution time of the software. Numerical calculations are used to find children, determine the validity of a child, and to calculate the costs of a child. Thus, another difference affecting the execution of the software is the math coprocessors.

5.4.3.2 Number of Processors Used (Granularity). Testing the effects the number of processors used to solve the problem has on the overall execution time is a simple matter. Each of the hypercubes allow users to specify the size of the cube desired. This allocated cube can contain

all processors of the system or a subset. For the Intel hypercubes the restriction on the size of a subset cube is that the number of processors (n) in the subset must be a power of two ($n = 2^k$) (27:2-4)(6:4). To change the number of processors used all that is needed is to change the size of the allocated cube. Because only a maximum of 8 processors are available only 4 sizes of cubes are possible (cube sizes = 1, 2, 4, 8). Since two programs (worker and controller) are used to solve the problem a cube size of one is not possible. The cube size of one is used to execute the sequential version, thus allowing a direct and valid comparison between the sequential and parallel versions of the software for speed-up analysis.

5.5 Test Results

The results of executing the software for each of the experiments are contained in the following sections. The raw data is found in Appendix C. This information is condensed into tables and graphs. The tables list the relevant data while the graphs are used to show trends and for the comparisons discussed in the previous section.

5.5.1 Reduction of the Search Space. Two algorithms, one not employing any bounding and the same algorithm using a bounding technique were executed. Table 5.2 contains the results of executing the bounded algorithm, with a recursion depth of three while Table 5.3 contains the results of the un-bounded algorithm, with the same recursion depth. Table 5.4 contains the number of nodes, in the search space, which are expanded by each algorithm. The number of nodes expanded is a measure the "amount of work" performed by each strategy.

Table 5.2. Bounded Algorithm with a Depth of 3 on the iPSC/2

Cube Size	Nodes Expanded	Program Efficiency		Timing (seconds)				Improvement	
		Worker	Controller	Initialize	Initial Route	Search	Total	Speed-up	Efficiency
1	7337	—	—	7.636	13.431	49207.111	49228.178	—	—
2	7336	0.896	0.408	0.142	18.806	37282.756	37301.704	1.32	0.66
4	6756	0.632	0.835	0.250	19.177	16306.274	16325.701	3.02	0.75
8	7306	0.304	0.969	0.290	19.457	15677.566	15697.313	3.14	0.39

Table 5.3. Recursion Only Algorithm with a Depth of 3 on the iPSC/2

Cube Size	Nodes Expanded	Program Efficiency		Timing (seconds)		
		Worker	Controller	Initialize	Search	Total
2	7669	0.878	0.450	0.208	37621.738	37621.946
4	6577	0.603	0.857	0.401	15622.431	15622.833
8	7338	0.281	0.975	0.306	16084.236	16084.542

Table 5.4. Work Performed by the Bounded and Un-bounded Algorithms

Cube Size	Nodes Expanded (Bounded)	Nodes Expanded (Un-bounded)
2	7336	7669
4	6756	6577
8	7306	7338

Figure 5.2 shows the execution times on the iPSC/2, using the same input data, for the algorithm without a bounding strategy and for the same algorithm employing bounding. Also the efficiency of the controller and worker programs is calculated and are shown in Figure 5.3. It is interesting to note in the figures that there is very little difference between the two algorithms, both in program efficiency and total execution time. The execution times for each are nearly identical as are the program efficiencies. There is also little difference in the amount of work performed, the total number of nodes expanded (Table 5.4), during the search process. It is evident that no time or work are saved by incorporating the bounding technique into the search. This conclusion is true for the bounding technique implemented and for the Air Tasking Orders used. This may not be true of other mission scenarios or another bounding technique.

5.5.2 Parallel Architecture. To test the effect of changing architectures the software is run on two hypercubes each based on a different processor. An Intel iPSC/2 and an Intel iPSC/860 are used. Tables 5.2 and 5.5 contain the results of executing the bounded algorithm, for recursion depths of three and four, on the iPSC/2. Tables 5.6 and 5.7 contain the results of executing the same algorithm, for the same recursion depths, on the iPSC/860.

It appears from Figure 5.4 that there is little difference between the iPSC/2 and the iPSC/860 running the same program, at least for the total execution time. The iPSC/860 is supposed

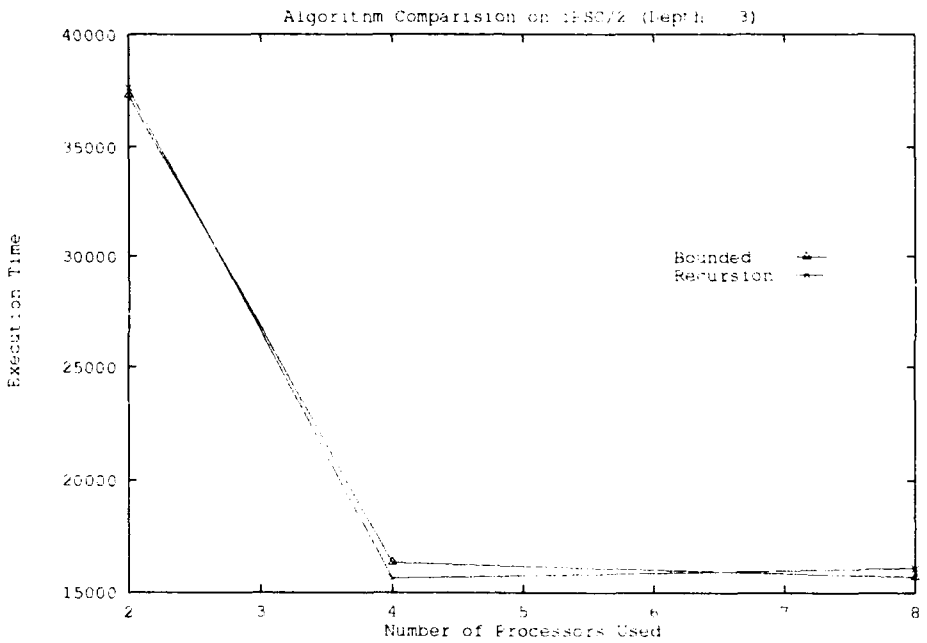

Figure 5.2. Timing Effects of Implementing a Bounding Strategy

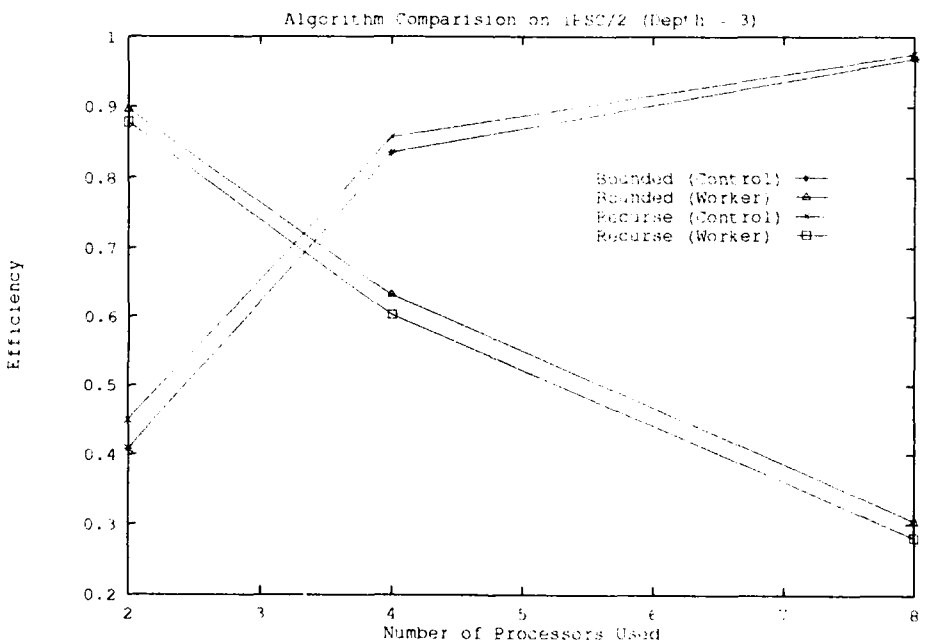

Figure 5.3. Efficiency Effects of Implementing a Bounding Strategy

Table 5.5. Bounded Algorithm with a Depth of 4 on the iPSC/2

Cube Size	Nodes Expanded	Program Efficiency		Timing (seconds)				Improvement	
		Worker	Controller	Initialize	Initial Route	Search	Total	Speed-up	Efficiency
1	5355	—	—	7.657	68.163	145228.101	145303.921	—	—
2	5354	0.988	0.068	0.180	73.364	137152.444	137225.988	1.06	0.53
4	5210	0.981	0.198	0.148	73.664	45295.759	45369.571	3.21	0.80
8	5245	0.957	0.442	0.168	73.936	19856.748	19930.852	7.31	0.91

Table 5.6. Bounded Algorithm with a Depth of 3 on the iPSC/860

Cube Size	Nodes Expanded	Program Efficiency		Timing (seconds)				Improvement	
		Worker	Controller	Initialize	Initial Route	Search	Total	Speed-up	Efficiency
1	16021	—	—	2.177	5.127	46028.364	46035.668	—	—
2	15751	0.875	0.511	0.349	7.511	32636.116	32643.976	1.41	0.70
4	15626	0.519	0.912	0.557	7.498	17960.518	17968.573	2.56	0.64
8	15893	0.242	0.984	1.199	9.236	16998.520	17008.955	2.71	0.34

to be a faster machine than the iPSC/2, though this does not appear to be the case from the results. Examining the contents of Table 5.8 shows that the software running on the iPSC/860 expanded nearly twice the number of nodes than the same software executing on the iPSC/2. This statement presupposes that the data for the iPSC/2 is correct and the results for the iPSC/860 are in error. Just because the software was implemented, tested, and debugged on the iPSC/2 does not guarantee that its results are not in error. The initial thought to the difference was the difference in architecture granularity, but this is dismissed when looking at the results of the sequential version (cube size = 1) which show the same phenomenon. Thus there must be a difference in either the software library routines loaded by the compiler or something to do with the architecture itself. Additional print statements were included in the software to determine what was happening during the execution of the software on each machine. The data collected is contained in Appendix D.2. It is evident from the results (Appendix D.2.1) that for certain parents some children are being

Table 5.7. Bounded Algorithm with a Depth of 4 on the iPSC/860

Cube Size	Nodes Expanded	Program Efficiency		Timing (seconds)				Improvement	
		Worker	Controller	Initialize	Initial Route	Search	Total	Speed-up	Efficiency
1	12446	—	—	2.108	33.228	158901.072	158936.416	—	—
2	12165	0.988	0.076	0.183	34.807	147856.127	147891.117	1.07	0.54
4	11541	0.982	0.221	0.358	34.913	47864.490	47899.761	3.32	0.83
8	11638	0.956	0.503	0.941	37.321	21183.054	21221.316	7.05	0.88

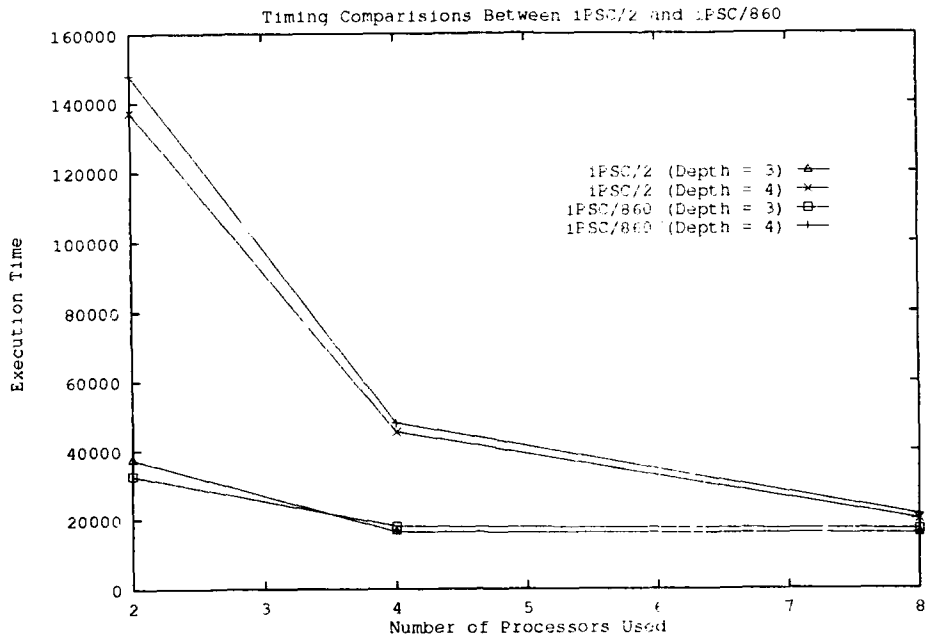

Figure 5.4. Timing Comparison of the iPSC/2 and iPSC/860

Table 5.8. Expansion on the iPSC/2 and iPSC/860

	Depth = 3		Depth = 4	
Cube Size	Nodes Expanded (iPSC/2)	Nodes Expanded (iPSC/860)	Nodes Expanded (iPSC/2)	Nodes Expanded (iPSC/860)
1	7337	16021	5355	12446
2	7336	15751	5354	12165
4	6756	15626	5210	11541
8	7306	15893	5245	11638

Table 5.9. Possible Angles

0
35.26439
45
48.18969
54.73561
60
65.90518
70.52878
90
109.47122
114.09484
120
125.26439
131.81031
135
144.73561
180

accepted as valid on the iPSC/860 while those same children, on the iPSC/2, are deemed as not valid. This indicates the problem exists somewhere in the routine which validates the children. Further investigation (Appendix D.2.2) reveals an accuracy problem between the results of the two machines. It happens that the maximum angle specified is 60° and this is one of the possible angles. Table 5.9 shows all the possible angles and one of those values is 60°. Appendix D.1 has a detailed discussion of how these numbers are obtained. The difference in angle calculation accuracy between the iPSC/2 and iPSC/860 and the fact that one of the possible angles is the same as the value being compared to explains the results. The iPSC/860 accepts more children as valid (its angle is just under 60°) than the iPSC/2 (its angle is just over 60°). This means the search conducted on the iPSC/860 has more children to explore than the search conducted on the iPSC/2 and this is why the iPSC/860 expands more nodes. One solution to this problem is to round-off or truncate the calculated angle before making the comparison. This approach would solve the problem regardless of the architecture used. Instead of modifying the code and rerunning all the test cases, which would have taken many days of execution, a simple test was performed. The maximum angle allowed is reduced to 59° which results in the same children being rejected on both the iPSC/2 and the iPSC/860. The angle could have been increased to 61° instead, but

since the iPSC/860 computer should be faster the test was run on it in order to minimize the time need to perform this test. Tables 5.10 and 5.11 contain the results of re-running the software on the iPSC/860 with the maximum angle allowed changed to 59°. Table 5.12 compares the number of nodes expanded on the iPSC/860, both before and after changing the angle, and on the iPSC/2. The iPSC/2 results are included since the search space on the iPSC/2, using 60°, should be the same as the iPSC/860 using 59°. After setting the maximum angle to 59° the number of nodes expanded during the search on the iPSC/860 is nearly the same as that on the iPSC/2 than the when the angle was 60°. Comparing the results of tables 5.6, bound-i860-d4, bound-i860-d3-59, and bound-i860-d4-59 it is observed that the number of nodes expanded is reduced by a factor of about two. The first inclination is that the time should also be reduced by two, that if half the work is performed then half the time is used. This would indicate a linear relationship, but the results show that the execution time is reduced by a factor of about four. A careful analysis of the problem reveals that this is correct. The problem space is $O(n)$ where n is the size of the open list, those routes being explored. If there is a linear relation then the time complexity would also be $O(n)$. There are n routes placed into the open list. Since the open list is a priority queue an insertion must check, worst case, all n routes before finding where in the queue the route being inserted belongs. Thus for each route the time complexity is $O(n)$ and since there are n routes the overall time complexity is $O(n^2)$. Therefore if the problem space is reduced by two the time is reduced by four, which is observed. This would indicate that if a better branch and bound routine were used then a quadratic reduction in execution time would be realized. Figure 5.5 shows the execution times for each scenario on each machine. In looking at the results in Table 5.12 some interesting things are occurring for a cube size of 4. For a depth of 3, the iPSC/860 software

Table 5.10. Angle = 59.0 & Depth = 3 on the iPSC/860

Cube Size	Nodes Expanded	Program Efficiency		Timing (seconds)				Improvement	
		Worker	Controller	Initialize	Initial Route	Search	Total	Speed-up	Efficiency
1	6954	—	—	5.079	3.188	11350.693	11358.960	—	—
2	6817	0.893	0.413	0.484	6.013	8441.781	8448.278	1.34	0.67
4	7374	0.571	0.865	0.582	5.661	4765.459	4771.702	2.38	0.60

Table 5.11. Angle = 59.0 & Depth = 4 on the iPSC/860

Cube Size	Nodes Expanded	Program Efficiency		Timing (seconds)				Improvement	
		Worker	Controller	Initialize	Initial Route	Search	Total	Speed-up	Efficiency
1	5354	—	—	2.242	16.055	34602.304	34620.600	—	—
2	5353	0.986	0.072	0.323	18.438	32698.327	32717.088	1.06	0.53
4	4284	0.981	0.193	0.549	18.479	8269.536	8288.564	4.18	1.04

Table 5.12. Effect of Changing the Maximum Angle Allowed

Cube Size	iPSC/860				iPSC/2	
	Angle = 60.0		Angle = 59.0		Angle = 60.0	
	Depth = 3	Depth = 4	Depth = 3	Depth = 4	Depth = 3	Depth = 4
1	16021	12446	6954	5354	7337	5355
2	15751	12165	6817	5353	7336	5354
4	15626	11541	7374	4284	6756	5210
8	15893	11638	—	—	7306	5245

expanded about 400 less nodes except for a cube size of 4 where it expanded about 500 more. For a depth of 4, each machine expanded nearly the identical number of nodes except again for a cube size of 4 where the iPSC/860 expanded nearly 1,000 less nodes. Why this is occurring is not clear. The first inclination would be that the granularity of the machines is causing this phenomenon, but it is possible that there are other differences in the software, like the accuracy problem, which are being manifested in these results.

In looking at the speed-up results in Figure 5.6 for a recursion depth of 3 there appears to be little speed-up when increasing the size of the cube from 4 processors to 8 processors. The communication bottleneck occurring at the controller processor would explain this phenomenon. This matches the conclusions of Abdelrahman (1:1496), Garmon (22:6-8), and Rottman(7:105). For a recursion depth of 4 near linear speed-up is achieved. This is because the number of messages being sent to the controller processor during a time period is decreased (i.e. granularity increased). The controller processor is able to process messages so as to keep the worker programs idle for shorter periods of time. As the number of worker processors is increased the efficiency in speed-up should begin to decrease. The same results for a recursion depth of 3 should be seen for a recursion depth of four as the number of processors used is increased. Why is super-linear speed-up seen on the iPSC/860 (angle set to 59 with a recursion depth of four) for a cube size of four? The sequential

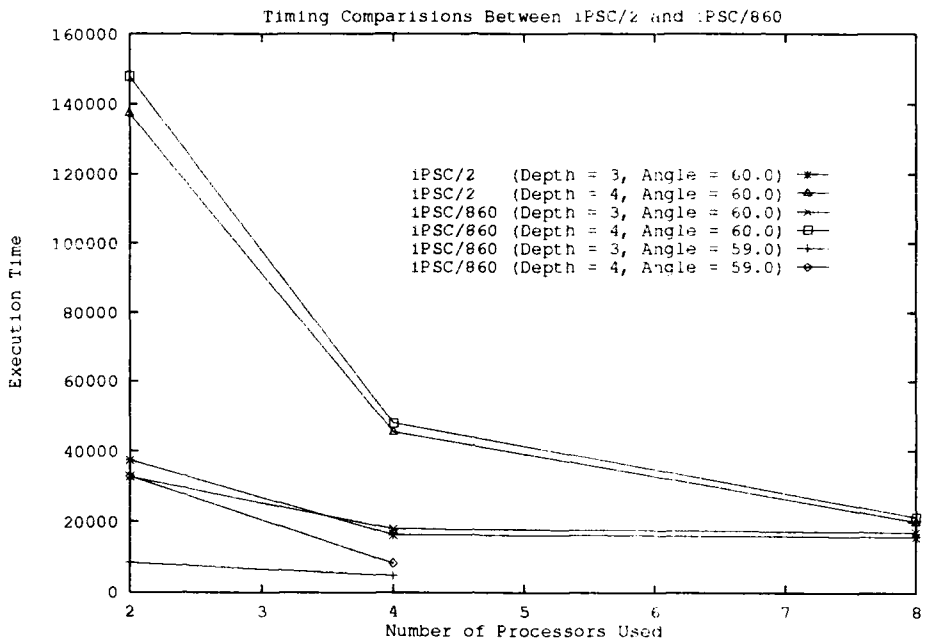

Figure 5.5. Effect of Changing the Maximum Angle Allowed

versions for depths three and four were executed at the same time. Since all I/O goes through the host processor (SRM) if multiple nodes need to perform an I/O function then one performs its operation while the others must block and wait for their turn. Perhaps this occurred for the depth of four software causing its execution time to increase. Since this value is in the nominator of the speed-up equation, increasing the sequential execution time increases the speed-up value. This is a possibility, there may be others.

5.5.3 Number of Processors Used (Granularity). The effect of changing the size of the cube used is examined as is changing the depth of the recursion search for h'. The effect on timing is shown in Figure 5.7 while the effect on program efficiency is shown in Figure 5.8. It is interesting to note from Figure 5.7 that when increasing the cube size from 4 to 8 caused an increase in execution time for a recursion depth of 2, but that the execution times continued to decrease with an increase of cube size for both a depth of 3 and 4. The results shown in Figure 5.8 give a clearer understanding of what is taking place. For a depth of 2, as the cube size increases the efficiency of the controller

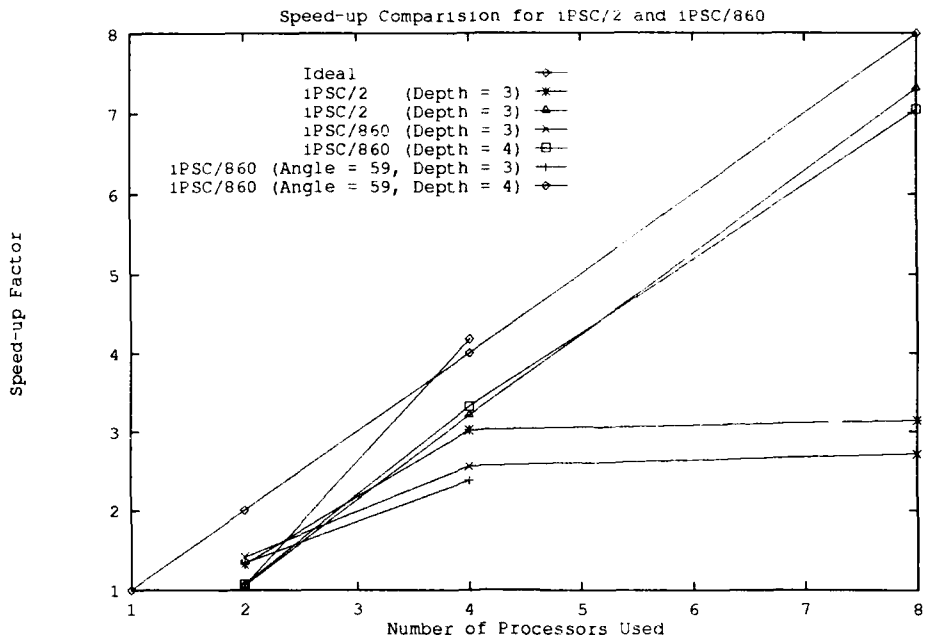

Figure 5.6. Speed-up Comparison of the iPSC/2 and iPSC/860

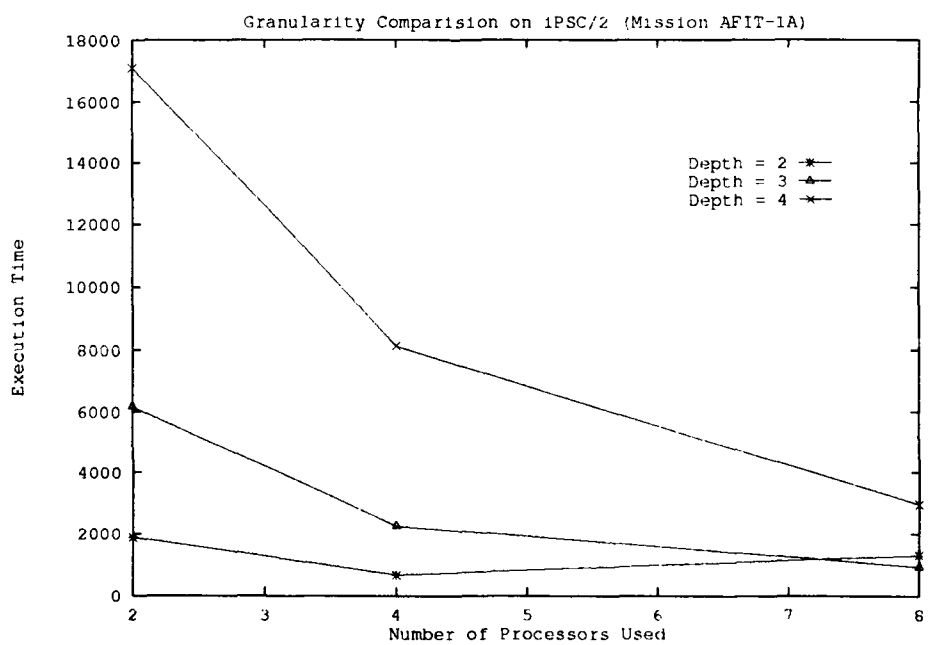

Figure 5.7. Timing Effects of Changing the Recursion Depth

program also increase, to a point, while the efficiency of the worker program decreases. As the cube size increases the number of worker programs increases which results in an increase in the number of messages being sent to the controller processor. The controller is spending most of its time busy handling all this message traffic and the worker programs are spending more time waiting for their requests to be answered. This is evident in Figure 5.8 by the high efficiency of the controller program which increases as the cube size increases. The average worker program efficiency decreases as the cube size increases. Thus, as the cube size increases the controller node is busier and the workers spend more time being idle. As the depth of the recursion is increased the time between sending messages increased, thus the granularity of the problem is increased. This is because as the recursion depth is increased the search space is enlarged thus resulting in an increase in the time to calculate h' which means the time between communication is increased. The communication bottleneck at the controller processor impacts the overall execution time. As the controller's efficiency increases it is desired that the worker's efficiency not decrease resulting in a decrease of the overall execution time. Based on the results for a depth of 2 (Figure 5.7) it is anticipated that for the other depths are the cube size is increased eventually a point will be reached after which the execution time will increase. This is the point where the impact of the communication bottleneck is such that the software actually requires more time to find a solution.

5.6 Summary

This chapter describes the metrics used to analyze the software developed to solve the mission routing problem and its execution on a parallel computer. These metrics consist of the number of nodes expanded and timing information. Timing is used to calculate the efficiency of the controller and worker programs, compare total execution times, and determine speed-up. Also presented is the input data (terrain, radar, and ATO). Experiments were developed and discussed which investigated the execution of the software in a parallel processing environment. The results obtained

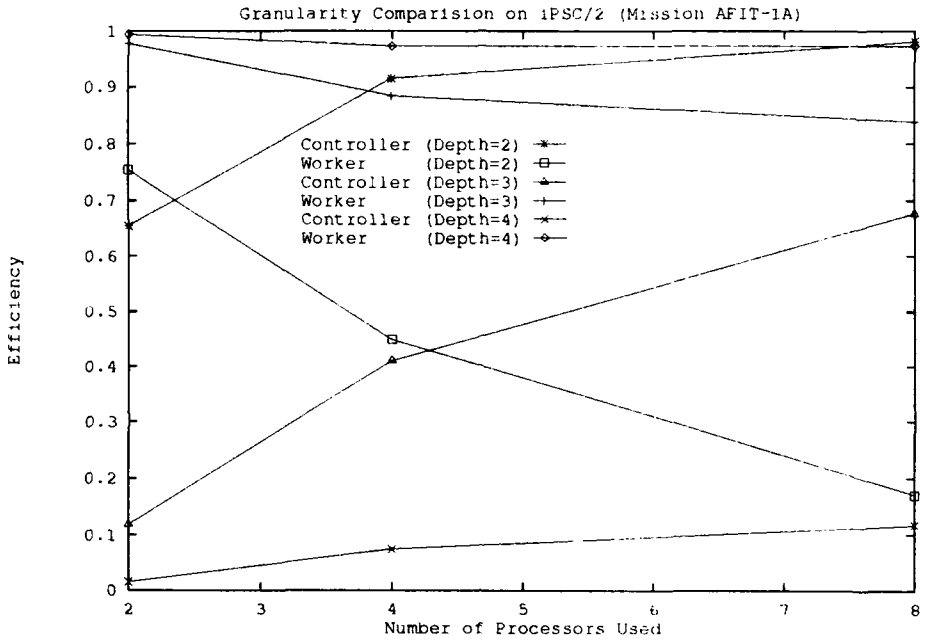

Figure 5.8. Efficiency Effects of Changing the Recursion Depth

are presented, using tables and graphs, along with analysis. Chapter VI presents an overview of the analysis, suggested improvements to the software, and recommendations for further research.

VI. Conclusions and Recommendations

6.1 Introduction

This research examines the feasibility of using a parallel processing environment in selecting mission routes. Selecting a route can be viewed as simply finding the optimal path between the starting point and the target based on some criteria. The problem of finding the route can be reduced to the problem of performing a search within the domain of the threat environment. The search process is combinatoric because a number of parameters such as threats, fuel consumption, time on target, target locations, distance flown, and refueling points must be considered. A parallelized A* search algorithm was used for the search strategy. A number of parameters, such as bounding the search space, architecture, and problem granularity, are changed to study their impact on the execution time.

Chapter V presented the experiments, the results obtained, and the associated analysis. This chapter presents a detailed overview of those results and relates them back to the objectives of this research effort as set forth in Chapter I. As with any software development there are always ways to improve the software, section 6.3 identifies areas of improvement for the software. Research is an on-going effort as one problem is solved, or answered, more questions arise which need to be researched further. It is the same with this research effort and section 6.4 discusses areas which require further investigation.

6.2 Interpretations of Results

The last chapter presented the analysis of the results obtained. The following sections present an overview of the analysis and the conclusions drawn.

6.2.1 Reduction of the Search Space. The search space was bounded by finding an initial best cost, using a depth-first greedy approach, and through the use of an altitude block. The cost

of routes being inserted into the open list are compared to the best cost and only those whose cost is less than the best are inserted. The altitude block specifies the minimum and maximum altitudes at which an aircraft may fly for the mission. This information is used when determining the next locations of a route. There is very little difference in the number of nodes expanded and the execution times between the algorithm using the branch and bound strategy and the one not using it. No time or work were saved by incorporating the bounding technique into the search. This does not mean that bounding the search has no effect, it does mean that for the bounding technique implemented and for the test missions used no savings was realized. Reducing the search space results in a decrease of the execution time as was found in testing changing the maximum allowable angle.

At first it was thought that reducing the search space would result in a linear decrease in the execution time, but this was not the case. The thought was that each route being expanded required about the same amount of time and that for each route eliminated a constant amount of execution time would likewise be eliminated. This assumes a linear relationship between the number of routes explored and the time to find a solution. This false assumption can be made when the complexity of the problem is not analyzed. When testing the effects of changing the maximum change in angle on the iPSC/860 the search space was reduced by about a factor of two and the execution time by about four. Since the open list contains n routes, there are n possible routes which may need to be explored. This results in the problem requiring $C \cdot n$ space where C is the amount of space required for a single route. Thus, the problem has a space complexity of $O(n)$. Since the open list is a priority queue, an insertion must check (worst case) all n routes before finding the location in the queue where the route being inserted belongs. Thus for each route the time complexity is $O(n)$ and since there are n routes the overall time complexity is $O(n^2)$.

Bounding the search space requires time, thus there is a trade-off between the time spent bounding the search and the time spent exploring "bad" nodes. The results of the experiment on

the iPSC/860 show that for a reduction in the search space a quadratic decrease in the execution time can be realized. Therefore the extra computation time of bounding the search is out-weighed by the reduced time of searching the "extra" nodes.

6.2.2 Metrics. Metrics are used used to analyze the performance of the software. In analyzing the results of the iPSC/860 one learns the importance of selecting the correct metrics. If only execution time had been used then the conclusion would have been drawn that for this problem there is no difference between the iPSC/2 and iPSC/860. Using the number of nodes expanded metric revealed that the searches progressed differently. This metric indicated a difference in the executable code executing on each computer. The difference was traced to the calculation of the angle between the direction vectors. Metrics must be selected with care. A thorough understanding of the problem and the algorithms used is necessary when selecting metrics. The use of extraneous metrics, those which provide no useful or additional information, is not desirable. Likewise, the selection of all useful metrics is desirable so as to not miss any important information.

6.2.3 Parallel Architecture. To test the effect of changing architectures the software is executed on an Intel iPSC/2 and iPSC/860. The similarities and differences between the two machines was presented in the last chapter (section 5.4.3.1). The same software was placed on each hypercube with the difference being the compiler used. The iPSC/860 uses a cross-compiler to compile and link code to be executed on its node processors. The results of executing the code show similar number of nodes expanded and execution times. This was not logical since the iPSC/860 is reported to be a faster machine than the iPSC/2. After further investigation (Section 5.5.2 and Appendix D.2.2) it was found that a difference in the value of an angle calculation caused the work and times to be similar. This indicates that one can not blindly port code from one computer to another. This is true even when there are no compilation or run time errors encountered. The person porting the code must have a through understanding of the algorithm and the subtle differences between the two computers. Also tests must be run to ensure the results are correct, after porting the software.

When looking at the execution times the iPSC/860 appeared to be just as fast as the iPSC/2, but this was not true. Once the accuracy problem was identified an additional experiment was conducted. The software on the iPSC/860 was modified so the same children accepted on the iPSC/2 would also be accepted on the iPSC/860, likewise the same ones would be rejected on both machines. The results of this experiment reveal that indeed the iPSC/860 is a faster computer. In looking at the execution times for a depth of 4, each computer expanded almost the identical number of nodes, the iPSC/860 was a factor of 4 faster than the iPSC/2. Since each has the identical communication network, the difference must be in the processors. The clock speed on the iPSC/860 is a factor 2.5 faster, this alone does not result in the speed-up achieved. The rest of the speed-up must be in the microprocessor architecture. The iPSC/860 uses a RISC processor along with other hardware techniques to increase the computational speed of computers; such as, instruction pipelines and floating-point units. The iPSC/2 uses a CISC processor with pipelines and floating-point units. Thus, the main difference between the two processors is the time needed to execute a single instruction. There is not enough information to determine the impact of the different math coprocessors on the execution time.

6.2.4 Granularity. By increasing the recursion depth the computation time is increased while the communication time remains the same, thus the granularity of the worker program increases. Likewise the granularity can decrease by decreasing the recursive search depth. Because of the potential communication bottleneck at the controller processor, increasing the granularity reduces the number of messages received by the controller during a time unit. This means that the workers are spending more time processing a given route, but the controller is also able to keep up with the message traffic. Thus the granularity of the worker programs impacts the overall execution time. As the granularity increased the execution time also increased for a given cube size. As the cube size increased the difference in execution time between the different granularities decreased. Thus, for a small cube size the greater recursion depth resulted in a greater execution

time. Figures 5.7 is a good example of the effect changing the problem granularity and the cube size have on the execution times.

As the cube size increased the execution time of the greater recursion depth approached that of the lesser recursion depth. In fact on the iPSC/2 with 8 nodes the recursion depth of 2 had a greater execution time than a recursion depth of 3. Likewise the execution time using 4 nodes was less than the execution time using 8 nodes. This indicates that the program granularity needs to be match to the parallel computer. This algorithm can not be blindly scaled to larger parallel computers. Again this has to do with the trade-off of more computation time per route being expanded versus the communication bottleneck at the controller processor. It is important to examine the granularity of the implemented algorithm and match it to the computer system.

What will happen when the cube size is increased? If the program granularity remains the same then the execution time decreases as the cube size increases. This is true up to a point after which the execution time increases. This is because of the communication bottleneck at the controller processor. The bottleneck reachs a point where the controller program has too many messages which need to be processed before the work requests can be handled. Even though the data collected for recursion depths of 3 and 4 show the execution time to be decreasing as the cube size increases eventually they too reach a point at which the execution time begins to increase.

6.3 Further Improvements to the Software

As with any software package there are always ways to improve the software. The goal of this research was not to produce an operational mission routing software package, but to determine the feasibility of using parallel processing to reduce the time needed to select a route. The software developed in support of this research effort made use of simplifying assumptions. Also because of time constraints portions of the code were not fully developed. The following sections discuss some of the proposed improvements to the code to make it more efficient, flexible, and more user-friendly.

6.3.1 Error Checking. The Air Tasking Order (ATO) data was known to be correct and for this reason no error checking was implemented. This assumption could be made because the user, namely this author, ensured that correct data was entered into the ATO file. This is not necessarily a good assumption as others may produce ATO files and enter them into the system. A simple error checking scheme needs to be incorporated into the software. This error checking should include:

- the x, y, z coordinates are within the bounds of the terrain and radar matrices,
- the terrain and radar matrices are the same size,
- the z coordinate for the starting and goal locations are above the ground,
- the minimum value of the altitude block is less than or equal to the maximum altitude block value,
- negative values are not specified, and
- the altitude type (AGL/MSL) is valid.

These checks are fairly simple to make and ensure the integrity of the parameters used during the search process. They ensure the search can progress correctly and protect the user from entering blatantly incorrect information.

6.3.2 Location Representation. Locations are specified using an x, y, z coordinate system. The x, y, z values corresponded to the location's position within the terrain matrix. This requires the user to have an thorough understanding of the terrain data. The user must know the area the terrain file includes and the reference point for the terrain matrix. This is not a realistic expectation of a user. A better means to specify a location is to use its longitude and latitude coordinates. This is most likely the form the user will have and this divorces the user from a knowledge of the terrain data. The terrain data would include the latitude and longitude of its reference point; this is what the Defense Mapping Agency (DMA) digitized data base uses. The software could convert

any longitude/latitude to the location within the terrain data. The altitude is already specified in feet versus using the z coordinate, so no change is necessary for this parameter.

6.3.3 Load Actual Terrain and Radar Data. As stated previously the routines to include actual DMA were not incorporated because of time constraints. Thus, to provide for a more realistic search these routines need to be used. Use of these routines allow a search to be conducted through an actual terrain. The loading of actual data by the system would require a better model of detection be used. There are system available, such as the Improved Many-On-Many (IMOM) which model radar coverage (2). Incorporating these systems produces a realistic terrain and threat environment through which a mission is flown.

6.3.4 Load Aircraft Information. The maximum change in direction was set to 60°. This is regardless of the aircraft and the terrain resolution. Like the maximum change in direction, the maximum combat radius is also hard-coded in the software. The software should be more flexible and realistic. The aircraft characteristics (i.e. minimum turn radius, maximum climb/dive rates, combat radius, etc.) should be stored in a file and loaded at execution time. This allows the selection of routes for multiple types of aircraft. This could be taken one step further by incorporating a flight dynamics model into the search process.

6.3.5 Route Representation. Presently a route is represented by each location along the route. This requires large memory allocation since the route data structure is uses in the open list as well as many times during the expansion of a single route. This also increases the size of messages being sent between nodes, thus increasing communication time. A better way is to keep only the turn points, those places where the aircraft changes direction. Since directional vectors are used to determine valid children, this information is already available. The software simply needs to compare the old directional vector with the new one. If they are the same then simply remove the parent location with the child location. If they are different then the aircraft has changed directions

and the child location is appended to the route. This produces a route description which is easier to understand, minimizes the information being conveyed, and makes it more realistic. Presently pilots keep track of turn points when planning and flying mission are the turn points and target locations and not all the points in between.

6.3.6 Recursion Efficiency. The present implementation of the recursion routine is inefficient. As the recursion search progresses much work is duplicated. As new children are included no check is performed to see if that state, location and direction vector, had already been explored. This is similar to the open list problem of having duplicate states. As the recursion routine is implemented states may be explored many times trying to determine an admissible h'. This duplication of work results in a longer execution time. A data structure could be used to keep track of each state encountered during the recursion search and a check could be made to determine if a state had been explored and what its cost was. There is a trade-off with this approach since it takes more time to explore each child, but the number of children being explored should be reduced. Which method results in the lower execution time is not clear. It is expected that spending the time eliminating duplicate work does because of the combinatoric property of the search process. As seen in the results of eliminating some of the search space, changing the maximum angle on the iPSC/860, a decrease in space produced a quadratic decrease in execution time. It is assumed that similar results may be obtained by eliminating work duplicated during the h' calculation.

Another technique which can be employed is to send the route being explored each time the recursion search "bottoms out." This is not instead of the original parent/child combination being explored. The original pair still needs to be sent since the recursion routine is not determining the optimality of the path, it is just determining the actual minimum cost to a specific depth. It is still possible that another path may result in an overall lower cost. What is desired is the global best while the recursion search produces a local best. It is possible that the search process continues down the same route as the recursion search. Thus, sending this route to the controller would

eliminate duplicate work. This strategy would increase the message traffic so there is a trade-off between eliminating possible duplicate work and reducing the efficiency of the worker programs.

6.3.7 Initial Route. A logic error is included in the finding of the initial route which is used to bound the search process. This logic error does not effect the selection of the initial only the time necessary to find the route. In examining the structure chart for the worker program (Figure 4.3) one observes that the routine to find the initial route calls the routine which calculates costs. This was done to eliminate duplication of code. Unknowingly this affected the time to find the initial route. The calculate costs routine calls the routine to calculate h' which uses the recursion search routine. Thus, as the depth of the recursion routine is increased the time needed to find the initial route increases. Either the routine to find an initial route should call a separate routine to update costs or the calculate costs routine needs to be modified to eliminate this problem.

6.3.8 Accuracy of Calculations. As discussed in the analysis of the results, an accuracy in the arc cosine function effected the amount of work performed. The code as implemented relied on the accuracy of the calculations to be correct and it was determined that this was incorrect. A simple mechanism of having the software force the desired accuracy should be employed. Simply truncating the calculated value by storing the floating point, double precision, result into an integer or floating point, single precision, variable could be used. Another possible method, though more complex, is to statically determine all the possible angles and check them against the maximum change allowed. If there is a match then the maximum change could be modified. This method is more complex than the first method and almost as flexible. The decision needs to be made as to the desired accuracy of ascertaining if a location is valid. This decision dictates the method and data structures used.

6.3.9 Reporting of Results. The software can be modified to make it more user-friendly by determining both the ingress and egress routes and by providing the pilot with multiple routes. A

single route is found which could be either an ingress or egress route, but not both. The restriction is usually placed upon the ingress and egress routes that they are not to have common locations other than the target and base. Once a solution is found the search could continue finding other solution routes, saving them in a new data structure. The pilot would be given each of these routes. This allows the pilot to select an ingress and egress route and to determine alternate routes should the need arise to deviate from the planned route. This would provide the pilot greater flexibility when planning a mission.

6.4 Recommendations for Further Research

Research is a never-ending process. As a research project progress questions are answered, new insights are gotten, and new facts or principles are discovered, but at the same time new questions arise. It is these questions which perpetuate the research process. This research effort is no different from any other. Questions have been posed and answered and new questions have arisen. Because of time constraints only so many questions could be investigated. Thus, there are still questions which need to be investigated. This section posses some of the questions, which have arisen during this effort, needing further research. Some areas may already be in the process of being researched by other organizations.

6.4.1 Local and Global Bests. All expanded routes were sent back to the controller program for inclusion into the open list. The controller determined whether the route was inserted into the open list by comparing its cost to the cost of the best solution. One area which needs further investigation is if each worker program should use a local best solution when determining whether to send a route back to the controller. This strategy would reduce the message traffic, thus reducing the work load of the controller processor. The controller would still keep the global best and use this information when inserting routes into the open list. This may not be a viable option for this

problem since most of the time a single solution was found at which time the search terminated, but it would still reduce communication between the worker processors and the controller processor.

6.4.2 Centralized versus Distributed Open List. The design and implementation was based on a centralized open list. Garmon examined the differences of a parallelized A* search strategy using a centralized versus a distributed open list when solving the Traveling Salesman Problem (TSP) (22). He determined that for larger size parallel computers the distributed open list was more efficient. Abdelrahman and Mudge also drew the same conclusion (1:1497). Garmon provides some guidelines when making the decision what type of open list implementation to employ on either an iPSC/2 or an iPSC/860 (22:6-17).

6.4.3 Parallel Search Strategy. The main purpose of this research effort was to determine the feasibility of using a parallel processing environment to solve the mission routing problem within a reasonable amount of time. The A* search algorithm was selected for its simplicity, also because a parallelized A* algorithm had been implemented in previous research at the Air Force Institute of Technology. The implemented A* produced long execution times. One area of further work is to examine the implementation of the A* algorithm and determine if they can be made more efficient, such as faster/better management of the open list and faster worker programs. There are other search strategies which may be more efficient in solving the mission routing problem. Research needs to be conducted into the parallelization of other search strategies, such as dynamic programming, to determine which is "best" to apply to this problem.

Another area of research is the direction of the search. What is meant by this statement is the definition of the start and goal states. Beginning the search at the base location and terminating at the target location may not be the most efficiency method. This is because of the multicriteria nature of the mission routing problem. It is believed that the start of the search should begin in the area which contains the heaviest weighted parameters whose values are the greatest. For instance if detection is given the highest weight factor then the area to begin would be the target

area because that is usually where the highest concentration of radar coverage is located. Selecting a single parameter may not be the best method either, but perhaps a combination of parameters will determine the beginning and ending states of the search. By beginning in the area of greatest criteria weighting it would help to ensure that if a beam reduction takes place on the open list then the probability the optimal route is deleted would be reduced.

Changing the search criteria during the search process is another area of mission routing which needs to be studied. As implemented the search using the same heuristics and search process throughout the entire search. With the weighting factors used once a route is found which is no longer within the enemy's radar coverage there is little to distinguish one route from another. Thus many routes appear to be just as good as another and the search space increases. One way to correct this occurrence is once past the enemy's radar coverage use a straight-line route similar to the initial route found for bounding purposes. Then just "walk" along the route making sure it is "free" of radar detection (or other criteria). Another possible method is to change the weighting factors of the criteria during the search. Thus, once past the enemy's radar coverage change the weighting so that in this case distance is weighted greater than detection.

The addition of other criteria such as jamming, aircraft orientation, radar cross section (RCS), fuel consumption, and so forth needs further research. The determination of the impact some of these criteria have upon the success of the mission are beyond the abilities of a pilot. With the advent of "stealth" or low-observable aircraft the detection by enemy has taken on new dimensions. No longer can a simple probability of detection be calculated based on radar equations. The RCS of an aircraft must first be modeled, next the orientation of the aircraft with respect to each of the radar sites must be determined, and then can the probability of detection be calculated. This further refined criteria adds additional complexity to an already complex problem. Also as the number of criteria added to the search the complexity of the problem, search, and software also increases. This can result in an increase in execution time.

6.4.4 Monitoring of the Search. The method used to monitor the search progress is though the use of print statements. This information is difficult to comprehend as it applies to how the search is progressing. This information is important in evaluating the effectiveness of the heuristics, weighting factors, and search algorithm in general. There are other methods available to convey this information such as search graphs and display of the terrain overlaid with routes being explored. The AFIT Algorithm Animation Research Facility (AAARF) already has the ability to display search graphs though modification of the routines may be needed to allow them to work properly with this problem. There are routines available to display terrain data though additional routines may need to be designed and implemented to overlay the routes being explored. Research in this area would determine the best method to monitor the search process and the way this information is to be given to the user, manely the developer.

6.4.5 Reporting of Results. An area associated with monitoring the search is how to report the results of the search. The implemented system simply displays the x, y, z coordinates in order. This requires the pilot to transpose this information back onto a terrain map. Research is being conducted into many aspects of virtual reality. A possible use would be to transfer the terrain, detection, and solution information to such a system and allow the pilot "watch" the mission being flown.

6.4.6 Architectures. Even though two different processor architectures were examined each machine had the same communication network. Other types of communication networks, such as the mesh, need to be explored. The type of communication network employed can effect the execution times of the software. A comparison of the architecture and the search algorithm used needs to be examined. The operation of the search algorithm is dependent on the architecture, thus one algorithm may run faster on one architecture while another algorithm is better suited for another type of architecture. Table 6.1 lists characteristics of some parallel computers. The observations made in section 6.2 must be kept in mind when investigating these other parallel

Table 6.1. Parallel Computer Characteristics

Computer	Classification	Interconnection Network Topology	Memory Topology	Maximum Processors
CAMPUS/800	MIMD	2-Level Crossbar Switch	Shared/ Distributed	800
KSR-1	MIMD	Hierarchy of Rings	Shared	1088
iPSC/860	MIMD	Hypercube	Distributed	128
MasPar MP-2	SIMD	2D Mesh	Shared	16384
nCUBE 2	MIMD	Hypercube	Distributed	8192
Paragon	MIMD	2D Mesh	Shared/ Distributed	4096

computers. The program granularity could be changed, thus the execution time is dependent on both the number of processors and the program granularity. This allows the implemented software to be matched to the parallel computer.

Increasing the computational speed of the computer is an on-going research effort. There are a number of hardware techniques in use to increase the computational speed of computers; such as, reduced instruction set computers (RISC), instruction pipelines, vectorization of instructions, functional units, and parallel processors. If a mission routing system is to support real-time, on-board the aircraft, systems then not only are faster, more efficient algorithms needed, but also faster architectures.

6.4.7 Expert Systems. The use of expert systems on-board the aircraft are on-going research projects sponsored by the Air Force. These types of systems hope to remove many of the tasks from the pilot onto the computer thus allowing the pilot to concentrate on the mission and other tasks. Issues which need to be addressed include: how does the system monitor the flight, how does the system detect changes in the environment (i.e. new threats, planned threats no longer present), how does the system determine if the flight path needs to be changed, and what does the system do if changes are not necessary. The expert system could also be employed in the selection of routes before the mission is flown. As suggested earlier the weighting factors and heuristics could be changed during the search process and this would be an excellent area to use an expert system.

Also an expert system could monitor the routes being generated and determine the feasibility of those routes. For instance a route being explored may remain at a constant altitude, but a pilot may not what to remain at a constant altitude even though that is the best route. The expert system could also compare the routes to previously flown routes. This information could be used to prevent the same routes being flown a number of times and thus having the enemy expecting an aircraft even though there is no known radar in the area.

6.5 Summary

This research examines the feasibility of using a parallel processing environment in selecting mission routes. The research effort designed and implemented a parallelized A* search algorithm. Software engineering principles of top down design, modularity, software reuse, design specification language, and structure charts are used. This helped to reduce the time needed to design and implement the algorithm while also making the maintenance of the software easier. This also makes it easier to understand for those who review and use the algorithms in the future.

Speed-up is realized when the software is executed on parallel processing computers. A version of the software was developed to run on a single processor allowing a valid comparison to be made. From the results found in Chapter V and the discussions in this chapter and Chapter V the parallel versions had lower execution times. The amount of speed-up is dependent on the granularity of the software, the parallel computer, and the number of processors used. The impact the number of processors has on the total execution time is dependent on the granularity of the software. It is important to match these parameters in order to realize the minimum execution time. The software executing on the iPSC/860 had an execution time about 75% less than the the iPSC/2 when performing the same amount of work. It is feasible to use a parallel computer in selecting mission routes and more research is needed in this area; therefore, areas needing further study

are discussed. Also since time constraints did not allow a fully implemented system, suggested improvements to the software are presented.

Appendix A. *Source Code Listings*

A.1 Parallel Version

A.1.1 Host Program

```
/*********************************************************************
--   DATE: 23 Oct 92                                                --
--   VERSION: 3.0                                                   --
--                                                                  --
--   TITLE:  Parallel Mission Routing Host Process                  --
--   FILENAME:  HOST.C                                              --
--   AUTHOR: Capt James J. Grimm  II                                --
--   COORDINATOR:  R. Norris                                        --
--   PROJECT:  Thesis Research Project                              --
--   OPERATING SYSTEM:  System V                                    --
--   LANGUAGE: C                                                    --
--   FILE PROCESSING:  Compile and link with stdio.h and path.h     --
--   FUNCTION:  This program acts as the interface between the programs--
--              executing on the hypercube and the user.  It gets the --
--              file names from the user, then it loads the control   --
--              and worker programs onto the appropiate nodes of the  --
--              hypercube.  After sending the file names it waits to  --
--              receive the solution from the controller node, along  --
--              with timing information.                              --
--                                                                    --
--   HISTORY:  3.0 Modified by Capt James J. Grimm II for the Mission --
--                 Routing Problem.                                   --
--             2.0 Modified by 1Lt Michael S. Rottman for Travelling  --
--                 Salesman Problem                                   --
--             1.0 Written by Capt Rick Mraz for Deadline Job         --
--                 Scheduling Problem                                 --
*********************************************************************/

/*************************************************
*                  Header  Files                 *
*************************************************/

#include  <stdio.h>                  /* Standard IO                  */
#include  "path.h"                   /* Data structures file         */


/*******************************************************
*                   Global Variables                   *
*******************************************************/
char mission[10];                    /* Mission designator           */



/*******************************************************/
/*                       Main Program                  */
/*******************************************************/
main ()
{
 int dim,                            /* Cube dimension               */
     i, k,                           /* Counters                     */
     num_expanded,      /* Nodes sent to processors      */
     total_expanded,              /* Total nodes expanded          */
     node_expanded;                  /* Node expanded by a processor */

 long init_time,                     /* Initialization Time          */
      init_path_time,                /* Time to find initial route   */
```

```
     search_time,                    /* Time performing expansions    */
     run_time;                       /* Execution time from start     */

 float control_efficiency,           /* Efficiency of the controller  */
       amt_work,                     /* Amount work of each processor */
       total_work;                   /* Total work performed          */

 PATH best;                           /* Global Best Solution         */

 char terrainfile[20],                /* File containing terrain data      */
      ATOfile[20],                    /* File containing Air Tasking Order */
      planefile[20],                  /* File containing aircraft info     */
      radarfile[20];                  /* File with radar detection data    */

 FILE *fATO, *fopen();

/*********************************
*       BEGIN Processing        *
*********************************/

 setpid(Host_PID);

/* Get dimension of cube and name of data files                    */

  dim = numnodes();
  printf ("\n\t  PARALLEL MISSION ROUTING PROBLEM  \n");
  printf ("\n\t      A* USING CENTRALIZED LIST     \n");
  printf ("  \t*************************************\n");

  printf ("\nEnter name of file containing the terrain data: ");
  scanf  ("%s", terrainfile);
  printf ("Enter name of file containing the radar data : ");
  scanf  ("%s", radarfile);
  printf ("Enter name of file containing the Air Tasking Order (ATO): ");
  scanf  ("%s", ATOfile);
/*  printf ("Enter name of file containing the aircraft information: ");
  scanf  ("%s", planefile);                                          */

/* Load the Controller process to node 0 and the Worker process to the  */
/* other nodes, then send out the file names to all the nodes.          */

  load  ("worker", ALL_NODES, NODE_PID);
  killproc (CONTROLLER, NODE_PID);
  load  ("control", CONTROLLER, NODE_PID);

  csend(TERRAIN_FILE, terrainfile, sizeof(terrainfile), ALL_NODES, NODE_PID);
  csend(ATO_FILE, ATOfile, sizeof(ATOfile), ALL_NODES, NODE_PID);
  csend(RADAR_FILE, radarfile, sizeof(radarfile), ALL_NODES, NODE_PID);
/*  csend(PLANE_FILE, planefile, sizeof(planefile), ALL_NODES, NODE_PID);    */

  printf ("\n\nWaiting for results ...\n");

  /* While waiting get the mission designator from the ATO file  */
  fATO = fopen (ATOfile, "r");
  fscanf (fATO, "%s",  mission);
  fclose (fATO);

 /*Now wait for results to be returned by the controller node    */
  crecv (BEST_TYPE, &best, sizeof(best));
  crecv (TIME_TYPE, &init_time, sizeof(init_time));
  crecv (TIME_TYPE, &init_path_time, sizeof(init_path_time));
  crecv (TIME_TYPE, &search_time, sizeof(search_time));
  crecv (TIME_TYPE, &run_time, sizeof(run_time));
  crecv (TIME_TYPE, &control_efficiency, sizeof(control_efficiency));
  crecv (NUM_TYPE,  &num_expanded, sizeof(num_expanded));
```

```
  total_work = 0.0;
  total_expanded = 0;

  for (i = 1; i < dim; i++)
    {
     crecv (WORK_TYPE, &amt_work, sizeof(amt_work));
     total_work += amt_work;
    }  /* end for */

  for (k =1; k < dim; k++)
    {
     crecv(EXPANDED, &node_expanded, sizeof(node_expanded));
     total_expanded += node_expanded;
    }  /* end for */

  total_work /= (dim - 1);

  /* Print the best cost route   */

  print_route(best);

  printf("\n\t ***  Timing Information  ***\n\n");
  printf("Initialize = %9.3f (sec) \t Find Initial Route = %9.3f (sec) \n",
          (float)init_time/1000.0, (float)init_path_time/1000.0);
  printf("Searching  = %9.3f (sec) \t Total Execution    = %9.3f (sec) \n\n",
          (float)search_time/1000.0, (float)run_time/1000.0);
  printf("Average worker node efficiency %5.3f\n", total_work);
  printf("The controller efficiency was  %5.3f\n", control_efficiency);
  printf("** %d nodes sent to processors \n", num_expanded);
  printf("** %d total nodes expanded   \n\n", total_expanded);

  waitall(ALL_NODES, NODE_PID);
  killcube(ALL_NODES, NODE_PID);
  relcube();
}    /* end host */

/*************************************************************
-- Subroutine PRINT ROUTE                                   --
-- Passed:  best - best route                               --
-- Returns: none                                            --
-- Function: Prints out the best route                      --
*************************************************************/
print_route(route)
PATH route;                    /* Route to print             */
{
 int i,                        /* Loop counter               */
     num;                      /* Number of nodes in route   */
 US  x, y, z;                  /* Location of each node      */

 float miles_flown;            /* Distance in miles flown    */

 printf("\n The Best Route for mission %s is:\n", mission);
 printf("    x       y       z \n");

 num = route.number;
 for (i = 1; i <= num; i++)
   {
    x = route.x[i];
    y = route.y[i];
    z = route.z[i];
    printf("   %3d     %3d     %3d\n", x, y, z);
   } /* end for */

 printf("\n For a total of %d entries in the route \n", num);
```

```
 miles_flown = route.distance / 5280.0;

 printf("\n  At a distance of %f (%6.2f miles)\n", route.distance, miles_flown);
 printf("\n  With a radar cost of %f \n", route.radar);
 printf("\n  And a computed cost of %f \n", route.g);

} /* end print_route */
```

A.1.2 Controller Program

```
/*************************************************************************
--   DATE: 23 Oct 92                                                    --
--   VERSION: 3.0                                                       --
--                                                                      --
--   TITLE:  Parallel Mission Routing Controller Process                --
--   FILENAME:  CONTROL.C                                               --
--   AUTHOR: Capt James J. Grimm  II                                    --
--   COORDINATOR:  R. Norris                                            --
--   PROJECT:  Thesis Research Project                                  --
--   OPERATING SYSTEM:  System V                                        --
--   LANGUAGE: C                                                        --
--   FILE PROCESSING:  Compile & link with host.c, stdio.h, and path.h --
--   FUNCTION:  This program manages the centralized open list for an  --
--              parallelized A* search algorithm.  An intial route is  --
--              requested from the workers to be used to bounded the   --
--              the search.  The program loops, performing management  --
--              actions, until the open list is empty and no nodes are --
--              working and no messages are waiting to be read in.     --
--              If the open list is not full and expand messages are   --
--              waiting then a loop is performed to read in all the    --
--              expand messages until the list is full or there are no --
--              more expand messages.  The next loop reads in all the  --
--              requests for work messages.  The last loop removes the --
--              route from the front of the open list and sends it to  --
--              the first processor tagged as available.  Once done the--
--              solution along with other processing information is    --
--              sent to the host program and a message is sent to the  --
--              worker programs telling them to exit.                  --
--                                                                     --
--                                                                     --
--   HISTORY:  3.0 Modified by Capt Grimm for the Mission Routing      --
--                 Problem                                             --
--            2.0 Modified by 1Lt Rottman for TSP                      --
--            1.0 Written by Capt Mraz for Deadline Job Scheduling     --
*************************************************************************/

/*************************************************
*                 HEADER  FILES                 *
*************************************************/

#include  <stdio.h>                    /* Standard IO                 */
#include  "path.h"                     /* Data structures file        */


/*******************************************************
*                 Global Variables                    *
*******************************************************/

int  node_status[MAX_CUBE_SIZE+1],      /* Status flag for each node    */
     q_front,                           /* Front of the queue pointer   */
     q_length,                          /* Queue Length                 */
     q_status,                          /* Queue Status                 */
```

```
    q_count,                          /* OPEN list insertion counter  */
    freeptr;                          /* Free List pointer            */

PATH  q[Q_SIZE+1],                    /* Queue of search space nodes  */
      best,                           /* Best route found             */
      E_node;                         /* Next node to expand          */

/*******************************************************
*   Function/Subroutine Prototype Definitions          *
*******************************************************/
int    get_free_node();
PATH   copy_node();
void   q_init();
PATH   delete_q();
void   insert_priority();
int    same_state();
void   beam_search_reduction();
void   prune_q();


/*******************************************************
*                   Main Program                        *
*******************************************************/

main ()
{
 int  x, y, z,                    /* Iteration Counter          */
      ignore,                     /* Dummy parameter            */
      num_nodes,                  /* Number of processors used  */
      scale_factor,               /* Elevation data scale factor */
      num_x,                      /* Number of x coordinates    */
      num_y,                      /* Number of y coordinates    */
      elevation,                  /* Terrain elevation          */
      next_node,                  /* Node that needs work       */
      request,                    /* Work Request Msg Var       */
      work_assigned,              /* Number of processors working*/
      num_expanded;               /* Number of nodes expanded   */

 US   base_x,                     /* X coordinate for base       */
      base_y,                     /* Y coordinate for base       */
      base_z,                     /* Z coordinate for base       */
      base_altitude,              /* Actual altitude of base     */
      target_x,                   /* X coordinate for target     */
      target_y,                   /* Y coordinate for target     */
      target_z,                   /* Z coordinate for target     */
      target_altitude,            /* Actual altitude of target   */
      start_x,                    /* X coord for starting location*/
      start_y,                    /* Y coord for starting location*/
      start_z,                    /* Z coord for starting location*/
      goal_x,                     /* X coord for ending location  */
      goal_y,                     /* Y coord for ending location  */
      goal_z;                     /* Z coord for ending location  */

 long from_node,                  /* Node number of csend processor*/
      start_time,                 /* Start time                   */
      run_time,                   /* Total execution from start   */
      init_time,                  /* Initialization Time          */
      init_path_time,             /* Time to find initial route   */
      search_time,                /* Time performing expansions   */
      total_work_time,            /* Time doing work on OPEN list  */
      start_work_time,            /* Beginning of a work cycle    */
      end_work_time;              /* Ending of a work cycle       */

 float efficiency;                /* Efficiency of the controller  */
```

```
 char terrainfile[20],                /* File containing terrain data  */
      radarfile[20],                  /* File with radar detection data*/
      ATOfile[20],                    /* File with Air Tasking Order   */
      mission[10];                    /* Mission designator            */

 FILE *fterrain, *fradar, *fATO, *fopen();


/*********************************
*      BEGIN PROCESSING          *
*********************************/
 start_time = mclock();

   /* Receive the file name of the terrain data from the host */

 crecv(TERRAIN_FILE, terrainfile, sizeof(terrainfile));

   /* Read in terrain data            */

 fterrain = fopen (terrainfile, "r");
 fscanf (fterrain, "%d", &num_x);
 fscanf (fterrain, "%d", &num_y);
 fscanf (fterrain, "%d", &scale_factor);
 fclose (fterrain);

    /* Receive the file name of the radar detection data from the host */

 crecv(RADAR_FILE, radarfile, sizeof(radarfile));

    /* Receive the file name of the Air Tasking Order data from the host */

 crecv(ATO_FILE, ATOfile, sizeof(ATOfile));

    /* Read in Air Tasking Order (ATO) information      */

 fATO = fopen (ATOfile, "r");
 fscanf (fATO, "%s",  mission);
 fscanf (fATO, "%hu", &base_x);
 fscanf (fATO, "%hu", &base_y);
 fscanf (fATO, "%hu", &base_altitude);
 fscanf (fATO, "%hu", &target_x);
 fscanf (fATO, "%hu", &target_y);
 fscanf (fATO, "%hu", &target_altitude);
 fclose (fATO);

 base_z = base_altitude/scale_factor;
 target_z = target_altitude/scale_factor;

 start_x = base_x;
 start_y = base_y;
 start_z = base_z;
 goal_x  = target_x;
 goal_y  = target_y;
 goal_z  = target_z;

   /* printf("\n Data Loaded by Controller (0) \n");  */
 total_work_time = 0;
 num_nodes       = numnodes();
 work_assigned   = num_nodes - 1;
 num_expanded    = 0;
 q_init();

    /* Place initial node (base node) into the OPEN list queue    */
 E_node.number   = 1;
 E_node.x[1]     = start_x;
```

```
E_node.y[1]     = start_y;
E_node.z[1]     = start_z;
E_node.vector_x = 0;
E_node.vector_y = 0;
E_node.vector_z = 0;
E_node.distance = 0.0;
E_node.radar    = 0.0;
E_node.g        = 0.0;
E_node.cost     = 0.0;
insert_priority(E_node);

init_time = mclock() - start_time;

/* Get initial best to use as a bound though best may not be a valid route */

csend(EXPAND_NODE, &E_node, sizeof(E_node), ALL_NODES, NODE_PID);
crecv(BEST_TYPE, &best, sizeof(best));
printf("\n   Received Initial Path with a computed cost of %f \n\n", best.g);

init_path_time =  mclock() - start_time - init_time;

   /* Main Loop:  while still nodes to expand or work assigned:   */
   /*         1) Collect Work Requests from the Worker Nodes      */
   /*         2) Recieve new nodes from Workers and queue up      */
   /*         3) Hand out nodes to expand                         */
   /*         4) Maintain the Global Best Answer                  */

while ((q_status != EMPTY) || (work_assigned) || (iprobe(NEW_NODE)) )
  {
       /* Collect any new nodes to add to active queue  */
   while ((iprobe (NEW_NODE) ) && (q_status != FULL))
     {
      start_work_time = mclock();
      crecv(NEW_NODE, &E_node, sizeof(E_node));
      from_node = infonode();
      /* printf("NEW_NODE from %d, cost = %f \n", from_node, E_node.cost); */

        /* insert into OPEN list if cost is less than cost of best route */
      if (E_node.cost < best.cost)
        {
         insert_priority(E_node);

            /* See if E_node is a solution  */
         if ( (E_node.x[E_node.number] == goal_x) &&
              (E_node.y[E_node.number] == goal_y) &&
              (E_node.z[E_node.number] == goal_z) )
           {
            printf("\n  A Solution Path Has Been Found \n\n");
            best = copy_node(E_node, best);
            prune_q();
           } /* end if E_node is a goal */
        } /* endif E_node.cost < best.cost */
      end_work_time   = mclock();
      total_work_time = total_work_time + (end_work_time - start_work_time);
     } /*  end while loop   */

        /* Collect work requests and set the status to AVAILable */
   while (iprobe (WORK_REQUEST))
     {
      start_work_time = mclock();
      crecv(WORK_REQUEST, &request, sizeof(request));
      from_node = infonode();
      /* printf(" Received work request from node %d\n", from_node); */
      node_status[from_node] = AVAIL;
      work_assigned--;
```

```
      end_work_time   = mclock();
      total_work_time = total_work_time + (end_work_time - start_work_time);
     } /*  end while loop   */
   /* If there are problems in the queue and if a worker available,     */
   /* mark the worker as BUSY, get an E_node from the active queue and  */
   /* send it to the Worker.                                            */

   while ((q_status != EMPTY) && (q[q_front].cost < best.cost) &&
      ((next_node = get_free_node (num_nodes)) != BUSY))
     {
      start_work_time        = mclock();
      node_status[next_node] = BUSY;
      E_node                 = delete_q();
      work_assigned++;

      /* printf("sending EXPAND_NODE to node %d \n", next_node); */

      csend(EXPAND_NODE, &E_node, sizeof(E_node), next_node, NODE_PID);
      num_expanded++;
      end_work_time   = mclock();
      total_work_time = total_work_time + (end_work_time - start_work_time);
     } /*  end while loop   */

  } /*  end while loop   */

   /* Once the best path has been found, terminate the search by sending  */
   /* a DONE message to all worker nodes and sending the best route along */
   /* with timing information to the host program.                        */

 search_time = mclock() - start_time - init_time - init_path_time;
 run_time    = mclock() - start_time;
 efficiency  = (float) total_work_time / (float) search_time;

 /*  printf("Sending info to the host program \n");   */

 csend (BEST_TYPE, &best, sizeof(best), myhost(), Host_PID);
 csend (TIME_TYPE, &init_time, sizeof(init_time), myhost(), Host_PID);
 csend (TIME_TYPE, &init_path_time, sizeof(init_path_time), myhost(), Host_PID);
 csend (TIME_TYPE, &search_time, sizeof(search_time), myhost(), Host_PID);
 csend (TIME_TYPE, &run_time, sizeof(run_time), myhost(), Host_PID);
 csend (TIME_TYPE, &efficiency, sizeof(efficiency), myhost(), Host_PID);
 csend (NUM_TYPE, &num_expanded, sizeof(num_expanded), myhost(), Host_PID);
 csend (DONE_TYPE, &ignore, sizeof(ignore), ALL_NODES, NODE_PID);
}


/*****************************************************************
--  Subroutine GET FREE PROCESSOR                                --
--  Passed:    n - number of nodes                               --
--  Returns:   BUSY - no worker available  OR                    --
--             i    - number of first available worker           --
--  Called By: main                                              --
--  Calls:     none                                              --
--  Function:  If a worker is available, return its node         --
--             number, otherwise return BUSY.                    --
*****************************************************************/
int get_free_node (n)
int n;
{
 int i;                              /* Loop counter           */

 for (i = 1; i <= n; i++)
   if (node_status[i] ==  AVAIL) return (i);
 return (BUSY);
```

```
}


/*****************************************************************
--  Subroutine COPY NODE                                        --
--  Passed:    n1 - source node                                 --
--             n2 - destination node                            --
--  Returns:   n2 - modified destination node                   --
--  Called By: main, delete_q, insert_priority                  --
--  Calls:     none                                             --
--  Function:  Copy node n1 into node n2 and return n2          --
*****************************************************************/
PATH copy_node (n1, n2)
PATH n1,n2;
{
 int i;                                /* Loop counter          */

 for (i = 0; i <= MAX_PATH_LENGTH; i++)
   {
    n2.x[i] = n1.x[i];
    n2.y[i] = n1.y[i];
    n2.z[i] = n1.z[i];
   }
 n2.number   = n1.number;
 n2.vector_x = n1.vector_x;
 n2.vector_y = n1.vector_y;
 n2.vector_z = n1.vector_z;
 n2.distance = n1.distance;
 n2.radar    = n1.radar;
 n2.g        = n1.g;
 n2.cost     = n1.cost;
 return(n2);
}


/*****************************************************************
--  Subroutine Q INIT                                           --
--  Passed:    none                                             --
--  Returns:   none                                             --
--  Called By: main                                             --
--  Calls:     none                                             --
--  Function:  Initialize array of free nodes by linking them   --
--             together and setting their costs to INFINITY.    --
--             Set q_status to EMPTY                            --
*****************************************************************/
void q_init ()
{
 int i;                                /* Loop counter          */

 for (i = 0; i <= Q_SIZE; i++)
   q[i].link = i + 1;

 q[Q_SIZE].link = EOQ;
 freeptr  = 0;
 q_status = EMPTY;
 q_length = 0;
 q_count  = 0;
 q_front  = 0;
}


/*****************************************************************
--  Subroutine DELETE Q                                         --
--  Passed:    none                                             --
--  Returns:   TEMP - lowest cost node on queue                 --
```

```
--  Called By: main                                    --
--  Calls:     copy_node                                --
--  Function:  Remove first element off queue and modify queue --
--             pointers and flags as needed             --
*************************************************************/
PATH delete_q ()
{
 PATH temp;                        /* Route removed from OPEN list */
 int  n;                           /* Points to new front of list  */

 if (q_status == EMPTY)
   {
    printf("****** WARNING -- THE OPEN LIST QUEUE IS EMPTY ****** \n");
    fflush(stdout);
    fflush(stderr);
   }
 else
   {
    /* get front node from active queue       */

    n = q_front;
    temp = copy_node (q[q_front], temp);
    q_front   = q[q_front].link;
    temp.link = EOQ;
    q_length -= 1;

     /* put old node back on free list       */

    q[n].link = freeptr;
    freeptr   = n;

     /* modify q status as appropriate      */

    if (q_status == FULL)
      q_status = Q_BUSY;
    else
      if (q_front == EOQ)
        q_status = EMPTY;

     return (temp);
   }  /*  endif  */
}  /*  end delete q  */


/*************************************************************
--  Subroutine INSERT PRIORITY                               --
--  Passed:    n - node to insert                            --
--  Returns:   none                                          --
--  Called By: main                                          --
--  Calls:     copy_node                                     --
--  Function:  Insert node n into queue based on its cost    --
*************************************************************/
void insert_priority (n)
PATH n;
{
 int i, j,                         /* Used to set links correctly  */
     num,                          /* Index where inserting route  */
     already;                      /* State already in OPEN list   */

 if (q_status == FULL)
   {
    printf("****** WARNING -- THE OPEN LIST QUEUE IS FULL ****** \n");
    fflush(stdout);
    fflush(stderr);
   }
```

```
else
  {
    /* get free node from free list to put new node into   */
   num     = freeptr;
   freeptr = q[freeptr].link;
   q[num]  = copy_node(n, q[num]);

    /* add new node to empty queue                  */

   if (q_status == EMPTY)
     {
      q[num].link = EOQ;
      q_front     = num;
      q_status    = Q_BUSY;
     }

      /* otherwise insert in cost order, with smallest costs in front */

   else
     if (q[num].cost < q[q_front].cost)
       {
        q[num].link = q_front;
        q_front     = num;
       }
     else
       {
        i = q[q_front].link;
        j = q_front;
        while ((i != EOQ) && (q[num].cost >= q[i].cost))
          {
           /* Checking to see if the state which is being inserted */
           /* into the OPEN list already exists in the OPEN list   */
           /* with a cost less than the one trying to be inserted. */
          already = same_state(q[num], q[i]);
          if (already == TRUE)
            {
             i = EOQ;
            }
          else
            {
             j =i;
             i = q[i].link;
            } /* endif already */
          } /* endwhile */
        if (already == FALSE)
          {
           q[j].link   = num;
           q[num].link = i;
          }
        else
          {
           q[num].link = freeptr;
           freeptr     = num;
          } /* endif already */
       } /* endif q[num].cost < q[q_front].cost */
   if (already == FALSE)
     {
      q_length += 1;
      q_count ++;

      if ((q_count % 500) == 0)
        printf(" ** Queue length = %d with q[q_front].cost = %f ** \n",
                q_length, q[q_front].cost);
     }
```

```
    if (freeptr == EOQ)
       beam_search_reduction();

   }  /* endif */
} /* end insert_priority */



/*****************************************************************
--  Subroutine same_state                                        --
--  Passed:    n1 - route being inserted                         --
--             n2 - route in the OPEN list queue                 --
--  Returns:   TRUE or FALSE                                     --
--  Called By: insert_priority                                   --
--  Calls:     none                                              --
--  Function:  Determines if the end location of a route being --
--             inserted into OPEN list queue was reached from  --
--             the same previous location as a route which is  --
--             already in the OPEN list queue.                  --
*****************************************************************/
int  same_state(n1, n2)
PATH n1, n2;
{
 int  index_n1,                    /* Parent location in route array n1*/
      index_n2;                    /* Parent location in route array n2*/

 index_n1 = n1.number;
 index_n2 = n2.number;

   /* Determination is based on being at the same location */
   /* and having arrived at that location along the same   */
   /* direction vector.                                    */

 if ( (n1.x[index_n1] == n2.x[index_n2]) &&
      (n1.y[index_n1] == n2.y[index_n2]) &&
      (n1.z[index_n1] == n2.z[index_n2]) &&
      (n1.vector_x == n2.vector_x)       &&
      (n1.vector_y == n2.vector_y)       &&
      (n1.vector_z == n2.vector_z) )
   return(TRUE);

 return(FALSE);
} /* end same_state */


/*****************************************************************
--  Subroutine beam_search_reduction                             --
--  Passed:    none                                              --
--  Returns:   none                                              --
--  Called By: insert_priority                                   --
--  Calls:     none                                              --
--  Function:  Performs a beam search type reduction on the    --
--             OPEN list.                                        --
*****************************************************************/
void   beam_search_reduction()
{
 int  pointer,                    /* Location in OPEN list    */
      counter,                    /* Size of the OPEN list    */
      reduction;                  /* Point to begin deleting  */

 printf("   <<<< Performing Beam Search Reduction >>>> \n");
 pointer   = q_front;
 counter   = 1;
 reduction = REDUCE_FACTOR * Q_SIZE;
```

```
 while (counter < reduction)
   {
    counter = counter + 1;
    pointer = q[pointer].link;
   }
 freeptr         = q[pointer].link;
 q[pointer].link = EOQ;
 q_length        = counter;
}



/*****************************************************************
--  Subroutine prune_q                                          --
--  Passed:    none                                             --
--  Returns:   none                                             --
--  Called By: main                                             --
--  Calls:     none                                             --
--  Function:  Prunes the OPEN list priority queue. This        --
--             will prune all routes whose cost is equal to or --
--             greater than the cost of the best route.  The    --
--             assumption made is that the best route is stored--
--             in a separate location.  The call to this        --
--             routine must follow the call to insert the a     --
--             route into the queue, thus when the OPEN list    --
--             is empty the program will terminate and the best--
--             which was stored in another location will be     --
--             displayed.                                       --
*****************************************************************/
void prune_q()
{
 int lead,
     trail,
     n;

 if(q_status != EMPTY)
   {
    q_count = 0;
    lead    = q_front;
    trail   = lead;
    n       = freeptr;

    if (q[q_front].cost >= best.cost)
      {
       while (lead != EOQ)
         {
          trail = lead;
          lead  = q[lead].link;
         }   /* end while */

       q[trail].link = freeptr;
       freeptr       = q_front;
       q_status      = EMPTY;
       q_length      = 0;
      } /* end if  */

    else
      {
       while ((q[lead].cost < best.cost) && (q[lead].link != EOQ))
         {
          q_count ++;
          trail = lead;
          lead  = q[lead].link;
```

```
        }   /*  end while  */

      q_length = q_count;
      if (q[lead].link != EOQ)
        {
         q[trail].link = EOQ;
         freeptr       = lead;
         while (lead != EOQ)
           {
            trail = lead;
            lead  = q[lead].link;
           }   /*  end while  */

         q[trail].link = n;
         q_status      = Q_BUSY;
        }  /*  end if  */
     } /* end else   */
   }  /*  end if  */

}  /*   end prune_q()  */
```

A.1.3 Worker Program

```
/*********************************************************************
--   DATE: 23 Oct 92                                                --
--   VERSION: 3.0                                                   --
--                                                                  --
--   TITLE:  Parallel Mission Routing Worker Process                --
--   FILENAME:  WORKER.C                                            --
--   AUTHOR: Capt James J. Grimm  II                                --
--   COORDINATOR:  R. Norris                                        --
--   PROJECT:  Thesis Research Project                              --
--   OPERATING SYSTEM:  System V                                    --
--   LANGUAGE: C                                                    --
--   FILE PROCESSING: Compile & link with stdio.h, path.h, and math.h --
--                    Ensure the program is linked with the math    --
--                    library using the -lm switch.                 --
--   FUNCTION:  This is the worker program of the parallel mission  --
--              routing software.  A controller node manages the    --
--              centralized open list of the A* search algorithm.  The --
--              program loads in all the necessary data (terrain,   --
--              radar, and ATO).  An initial route is found which is --
--              used by the controller to bound the search. Once done --
--              the worker request a work from the controller.  Then a --
--              loop is performed until the worker gets a done message --
--              from the controller.  The worker first checks for a --
--              route to be expanded (sent from the controller).  After--
--              the route is expanded a reqest for work is sent to the --
--              controller.  This process is repeated until the control--
--              node terminates the search.                         --
--                                                                  --
--   HISTORY:  3.0 Modified by Capt Grimm for the Mission Routing   --
--                 Problem                                          --
--             2.0 Modified by 1Lt Rottman for Travelling Salesman  --
--                 Problem                                          --
--             1.0 Written by Capt Mraz for Deadline Job Scheduling --
*********************************************************************/

/*******************************************************
*            Header Files                              *
*******************************************************/
```

```
#include  <stdio.h>                    /* Standard IO                 */
#include  <math.h>                     /* Standard math library       */
#include  "path.h"                     /* Data structures file        */

/*******************************************************
*                    Global Variables                  *
*******************************************************/

static int terrain_matrix[MAX_MATRIX_SIZE+1][MAX_MATRIX_SIZE+1];

static float
      radar_matrix[MAX_MATRIX_SIZE+1][MAX_MATRIX_SIZE+1][MAX_ALT_SIZE+1];

int   scale_factor,                    /* Elevation data scale factor */
      altitude_factor,                 /* Altitude data scale factor  */
      delta[8][2],                     /* Matrix for finding children */
      num_x,                           /* Number of x coordinates     */
      num_y,                           /* Number of y coordinates     */
      num_z,                           /* Number of z coordinates     */
      elevation,                       /* Terrain elevation           */
      min_above_ground,                /* Min altitude above ground   */
      lower_alt_block,                 /* Lowest flight altitude      */
      upper_alt_block,                 /* Highest flight altitude     */
      ceiling,                         /* Aircraft's flight ceiling   */
      combat_radius;                   /* Combat radius in miles      */

US    goal_x,                          /* X coord for ending location */
      goal_y,                          /* Y coord for ending location */
      goal_z;                          /* Z coord for ending location */

long  my_node;                         /* My node number              */

float detection,                       /* Radar detection value       */
      field_of_view;                   /* Max angle change for turns  */

char  altitude_type[4],                /* Altitude reference (AGL/MSL)*/
      mission[10];                     /* Mission designator          */

PATH  E_node;                          /* Node received for expansion */


/*******************************************************
*   Function/Subroutine Prototype Definitions           *
*******************************************************/
PATH   copy_node();
float  difference();
double magnitude();
void   Find_initial_path();
float  Find_h_prime();
int    valid_child();
float  recursion_search( );
PATH   Calculate_costs();
void   Find_children();



/*******************************************************
*                      Main Program                     *
*******************************************************/
main ()
{
 int x, y, z,                          /* Iteration counters          */
     num_expanded,                     /* Number of states expanded   */
     ignore;                           /* Dummy used by WORK REQUEST msg */
```

```
US  base_x,                          /* X coordinate for base location   */
    base_y,                          /* Y coordinate for base location   */
    base_altitude,                   /* Actual altitude of base          */
    target_x,                        /* X coordinate for target location */
    target_y,                        /* Y coordinate for target location */
    target_z,                        /* Z coordinate for target location */
    target_altitude;                 /* Actual altitude of target        */

long end_time,                       /* Time end expanding a search node */
     temp_time,                      /* Beginning time of search process */
     start_time,                     /* Start time of node expansion     */
     total_time,                     /* Total time doing search process  */
     work_time,                      /* Actual time spent expanding nodes*/
     num_nodes;                      /* Number of processors being used  */

float amt_work;

char buf[80];

char terrainfile[20],                /* File containing terrain data     */
     radarfile[20],                  /* File with radar detection data   */
     ATOfile[20],                    /* File with Air Tasking Order      */
     aircraft_type[10];              /* Type of aircraft                 */

FILE *fterrain, *fradar, *fATO, *fopen();


/*********************************
*       BEGIN Processing         *
*********************************/

 /* Ensure this is a valid node for the worker program  */

my_node = mynode();
num_nodes = numnodes();

if (my_node == 0)
  exit (1);
if (my_node >= num_nodes)
  exit(0);

   /*  Initialize the delta matrix  */

delta[0][0] =  0  ; delta[0][1] =  1;
delta[1][0] =  1  ; delta[1][1] =  1;
delta[2][0] =  1  ; delta[2][1] =  0;
delta[3][0] =  1  ; delta[3][1] = -1;
delta[4][0] =  0  ; delta[4][1] = -1;
delta[5][0] = -1  ; delta[5][1] = -1;
delta[6][0] = -1  ; delta[6][1] =  0;
delta[7][0] = -1  ; delta[7][1] =  1;

num_expanded     = 0;
field_of_view    = 60.0;
ceiling          = 40000;
combat_radius    = 575;

   /*  Receive the file name of the terrain data from the host */
crecv(TERRAIN_FILE, terrainfile, sizeof(terrainfile));

   /*  Read in terrain data                                    */
fterrain = fopen (terrainfile, "r");
fscanf (fterrain, "%d", &num_x);
fscanf (fterrain, "%d", &num_y);
fscanf (fterrain, "%d", &scale_factor);
```

```
for (y = 1; y <= num_y; y++)
  {
   for (x = 1; x <= num_x; x++)
     {
      fscanf (fterrain, "%d", &elevation);
      terrain_matrix [x][y] = elevation;
     } /* end for y */
  } /* end for x */
fclose (fterrain);

   /* Receive the file name of the radar detection data from the host */
crecv(RADAR_FILE, radarfile, sizeof(radarfile));

   /* Read in radar detection data              */

fradar = fopen (radarfile, "r");
fscanf (fradar, "%d", &num_x);
fscanf (fradar, "%d", &num_y);
fscanf (fradar, "%d", &num_z);
fscanf (fradar, "%d", &altitude_factor);

for (z = 0; z < num_z; z++)
  {
   for (y = 1; y <= num_y; y++)
     {
      for (x = 1; x <= num_x; x++)
        {
         fscanf (fterrain, "%f", &detection);
         radar_matrix [x][y][z] = detection;
        } /* end for y */
     } /* end for x */
  }  /* end for z */
fclose (fradar);

   /*  Receive the file name of the Air Tasking Order data from the host */
crecv(ATO_FILE, ATOfile, sizeof(ATOfile));

   /* Read in Air Tasking Order (ATO) information       */
fATO = fopen (ATOfile, "r");
fscanf (fATO, "%s",  mission);
fscanf (fATO, "%hu", &base_x);
fscanf (fATO, "%hu", &base_y);
fscanf (fATO, "%hu", &base_altitude);
fscanf (fATO, "%hu", &target_x);
fscanf (fATO, "%hu", &target_y);
fscanf (fATO, "%hu", &target_altitude);
fscanf (fATO, "%d",  &min_above_ground);
fscanf (fATO, "%d",  &lower_alt_block);
fscanf (fATO, "%d",  &upper_alt_block);
fscanf (fATO, "%s",  altitude_type);
fclose (fATO);

target_z = target_altitude / altitude_factor;

goal_x  = target_x;
goal_y  = target_y;
goal_z  = target_z;

temp_time = mclock();
work_time = 0;

/* Find initial path so as to create a bound on the search. */
/* This path is not guarenteed to be a valid path.          */
crecv(EXPAND_NODE, &E_node, sizeof(E_node));
```

```
Find_initial_path(E_node);

csend (WORK_REQUEST, &ignore, sizeof(ignore), CONTROLLER, NODE_PID);

   /*  printf(" Node %d sent work request to controller \n", my_node);   */

   /*   Control Loop:                                                    */
   /*         1) If an E_node has arrived, expand it and send any viable */
   /*            children back to controller.  Request more work.        */
   /*         2) If a DONE message arrives, terminate.                   */

for (;;)
  {
     /* 1) check for work: a node to expand.        */

   if (iprobe(EXPAND_NODE) )
     {
      start_time = mclock();
      crecv (EXPAND_NODE, &E_node, sizeof(E_node));
      num_expanded ++;

         /* printf ("Node %d received EXPAND_NODE \n",my_node);  */

      Find_children(E_node);

      end_time = mclock() - start_time;
      work_time += end_time;

        /* Send a request for more work */
      csend (WORK_REQUEST, &ignore, sizeof(ignore), CONTROLLER, NODE_PID);

       /*  printf("Node %d sent work request to controller \n",my_node);     */
     }  /* end if */

     /* 2) check to see if done.              */

   if (iprobe (DONE_TYPE) )
     {
      csend(EXPANDED, &num_expanded, sizeof(num_expanded), myhost(), Host_PID);
      printf("   Node %d expanded %d states\n", my_node, num_expanded);
      break;
     }

  }   /* end for ;;  */

 total_time = mclock() - temp_time;
 sprintf (buf,"total %ld, work %ld", total_time, work_time);
    /*    syslog (NODE_PID, buf);*/
 amt_work = (float) work_time/ (float) total_time;
 csend (WORK_TYPE, &amt_work, sizeof(amt_work), myhost(), Host_PID);

}  /* end main  */


/*****************************************************************
--  Subroutine COPY NODE                                        --
--  Passed:    n1 - source node                                 --
--             n2 - destination node                            --
--  Returns:   n2 - modified destination node                   --
--  Called By: Calculate_costs                                  --
--  Calls:     none                                             --
--  Function:  Copy node n1 into node n2 and return n2          --
*****************************************************************/
PATH copy_node (n1, n2)
```

```
PATH n1,n2;
{
 int i;                                /* Loop counter                */

 for(i = 0; i <= MAX_PATH_LENGTH; i++)
   {
    n2.x[i] = n1.x[i];
    n2.y[i] = n1.y[i];
    n2.z[i] = n1.z[i];
   }
 n2.number   = n1.number;
 n2.vector_x = n1.vector_x;
 n2.vector_y = n1.vector_y;
 n2.vector_z = n1.vector_z;
 n2.distance = n1.distance;
 n2.radar    = n1.radar;
 n2.g        = n1.g;
 n2.cost     = n1.cost;
 return(n2);
}


/*****************************************************************
--  Subroutine difference                                        --
--  Passed:    x1  - x coordinate of first location              --
--             y1  - y coordinate of first location              --
--             z1  - z coordinate of first location              --
--             x2  - x coordinate of second location             --
--             y1  - y coordinate of second location             --
--             z2  - z coordinate of second location             --
--  Returns:   distance                                          --
--  Called By: h_prime, Calculate_costs                          --
--  Calls:     none                                              --
--  Function:  Calculates an h' as the A* heuristic.             --
*****************************************************************/
float difference(x1, y1, z1, x2, y2, z2)
US x1, y1, z1, x2, y2, z2;
{
 int    x_square,                     /* Square of delta x's         */
        y_square,                     /* Square of delta y's         */
        z_square;                     /* Square of delta z's         */
 float  distance;                     /* Distance between locations  */

 x_square = (x1 - x2) * (x1 - x2);
 y_square = (y1 - y2) * (y1 - y2);
 z_square = (z1 - z2) * (z1 - z2);
 distance = (float) sqrt ((double)(x_square + y_square + z_square));
 return(distance);
}


/*****************************************************************
--  Subroutine magnitude                                         --
--  Passed:    x  - length of vector in x direction              --
--             y  - length of vector in y direction              --
--             z  - length of vector in z direction              --
--  Returns:   mag                                               --
--  Called By: valid_child                                       --
--  Calls:     none                                              --
--  Function:  Calculates the magnitude of a vector.             --
*****************************************************************/
double magnitude(x, y, z)
int x, y, z;
```

```
{
 double  mag;                          /* Vectors' magnitude          */

 mag = sqrt ((double)(x * x + y * y + z * z));
 return(mag);
}



/*****************************************************************
--  Subroutine Find_initial_path                                  --
--  Passed:    E_node : path being explored                       --
--  Returns:   none                                               --
--  Called By: main                                               --
--  Calls:     Calculate_costs, Find_initial_path                 --
--  Function:  Find an initial route to be used as the initial --
--             bound on the A* search.  The resultant route is --
--             not guarenteed to be a valid route.                --
*****************************************************************/
void Find_initial_path (E_node)
PATH E_node;
{
 int  index,                           /* Parent location in route array */
      found,                           /* Flag if a child was found      */
      delta_x,                         /* Difference from parent and goal*/
      delta_y,                         /* Difference from parent and goal*/
      delta_z,                         /* Difference from parent and goal*/
      min_altitude,                    /* Minimum flight altitude        */
      actual_altitude,                 /* Actual flight altitude         */
      terrain_elevation;               /* Elevation of the terrain       */

 US   parent_x,                        /* X coordinate of parent location*/
      parent_y,                        /* Y coordinate of parent location*/
      parent_z,                        /* Z coordinate of parent location*/
      child_x,                         /* X coordinate of child location */
      child_y,                         /* Y coordinate of child location */
      child_z;                         /* Z coordinate of child location */

 float radar_cost,                     /* Radar cost from parent to child*/
       cost;                           /* Total cost from parent to child*/

 PATH  temp;                           /* Partial route for recursion     */

/**********************************
*       BEGIN Processing         *
**********************************/

 index    = E_node.number;
 parent_x = E_node.x[index];
 parent_y = E_node.y[index];
 parent_z = E_node.z[index];

 if ((parent_x == goal_x) && (parent_y == goal_y) && (parent_z == goal_z))
   {
    if (my_node == 1)
      csend(BEST_TYPE, &E_node, sizeof(E_node), CONTROLLER, NODE_PID);
   }
 else
   {
    delta_x = goal_x - parent_x;
    if ( abs(delta_x) > 0)
      delta_x = delta_x / abs(delta_x);

    delta_y = goal_y - parent_y;
```

```
    if ( abs(delta_y) > 0)
      delta_y = delta_y / abs(delta_y);

    delta_z = goal_z - parent_z;
    if ( abs(delta_z) > 0)
      delta_z = delta_z / abs(delta_z);

    child_x = parent_x + delta_x;
    child_y = parent_y + delta_y;
    child_z = parent_z + delta_z;

    terrain_elevation = terrain_matrix[child_x][child_y];
    min_altitude      = terrain_elevation + min_above_ground;
    actual_altitude   = child_z * altitude_factor;

    if (actual_altitude > min_altitude)
      {
       temp = copy_node(E_node, temp);
       temp = Calculate_costs(temp, child_x, child_y, child_z);
       Find_initial_path(temp);
      }
    else
     {
      found = FALSE;
      while (delta_z < 1)
{
 delta_z = delta_z + 1;
         child_z = child_z + delta_z;
 actual_altitude = child_z * altitude_factor;
 if (actual_altitude > min_altitude)
   {
    temp = copy_node(E_node, temp);
            temp = Calculate_costs(temp, child_x, child_y, child_z);
            Find_initial_path(temp);
            delta_z = 2;
    found   = TRUE;
   }
} /* end while */
      if (found == FALSE)
{
         /* The only direction left is straight up (vertical climb) which */
         /* is not a valid direction.  This is one reason why the route   */
         /* found by this routine is not guarenteed to be a valid route.  */
 child_x = parent_x;
 child_y = parent_y;
 child_z = parent_z + 1;
 temp    = copy_node(E_node, temp);
         temp    = Calculate_costs(temp, child_x, child_y, child_z);


         /* As a compensation of not being a valid direction the cost of   */
         /* going from the parent to the child location is doubled by      */
         /* recalculating and adding to the total cost the radar detection */
         /* cost while the distance between locations is known to be 1.    */

         radar_cost = radar_matrix[child_x][child_y][child_z]
                      * (float) scale_factor;
         temp.radar = temp.radar + radar_cost;
 cost       = (1 - WEIGHT_RADAR) + (WEIGHT_RADAR * radar_cost);
         temp.g     = temp.g + cost;
         temp.cost  = temp.cost + cost;
 Find_initial_path(temp);
}
     } /* endif first child above ground */
   } /* endif at target */
```

```
} /* end Find_initial_path */


/*****************************************************************
--  Subroutine Find_h_prime                                      --
--  Passed:    x  - x coordinate of child node                   --
--             y  - y coordinate of child node                   --
--             z  - z coordinate of child node                   --
--  Returns:   h_prime                                           --
--  Called By: Calculate_costs                                   --
--  Calls:     difference                                        --
--  Function:  Calculates h' for the A* search algorithm.        --
*****************************************************************/
float Find_h_prime(E_node)
PATH E_node;
{
 float  h_prime;                        /* Calculated A* heuristic     */

 h_prime = recursion_search(E_node, 0);
 return(h_prime);
}


/*****************************************************************
--  Subroutine valid_child                                       --
--  Passed:    E_node  : path with parent                        --
--             child_x : x coordinate of child node              --
--             child_y : y coordinate of child node              --
--             child_z : z coordinate of child node              --
--  Returns:   TRUE or FALSE                                     --
--  Called By: Find_children                                     --
--  Calls:     magnitude                                         --
--  Function:  Determines if a child can be reached from the     --
--             the parent node.  A number of rules are applied --
--             to make this determination.                       --
--               NOTE: The rules are "fired" in sequential       --
--                  sequential order.  If a false condition      --
--                  occurs then firing ceases and a FALSE is     --
--                  returned to the calling routine.             --
--               RULE 1:                                         --
--                  The aircraft is at least some minimum        --
--                  height above the ground and below its        --
--                  ceiling.                                     --
--               RULE 2:                                         --
--                  The aircraft is within the block of          --
--                  altitude specified in the ATO file.          --
--               RULE 3:                                         --
--                  Ensure the aircraft flight distance is not --
--                  greater than its combat radius.  This        --
--                  ensures the aircraft will be able to         --
--                  return to its beginning location (.i.e.      --
--                  no in-flight refueling starting at base      --
--                  location will be assumed).                   --
--               RULE 4:                                         --
--                  The aircraft can rearch the child's          --
--                  location from it's present location and      --
--                  heading.  If the magnitude of the old        --
--                  direction vector is zero then an assumption--
```

```
--                  is made that the parent is the starting    --
--                  location and that any child node being     --
--                  tested can be reached.                     --
*****************************************************************/
int  valid_child (E_node, child_x, child_y, child_z)
PATH E_node;
US   child_x, child_y, child_z;
{
 US   parent_x,                 /* X coordinate for parent node */
      parent_y,                 /* Y coordinate for parent node */
      parent_z;                 /* Z coordinate for parent node */

 int  index,                    /* Array index of parent node            */
      min_altitude,             /* Minumum altitude at child location    */
      actual_altitude,          /* Actual altitude at child location     */
      terrain_elevation,        /* Elevation at child_x, child_y location */
      dir_x,                    /* Direction vector (x) to get to parent  */
      dir_y,                    /* Direction vector (y) to get to parent  */
      dir_z,                    /* Direction vector (z) to get to parent  */
      delta_x,                  /* Direction vector (x) to get to child   */
      delta_y,                  /* Direction vector (y) to get to child   */
      delta_z;                  /* Direction vector (z) to get to child   */

 float distance,                /* Distance (feet) flown by the aircraft  */
       miles_flown;             /* Distance (miles) flown by the aircraft */

 double numerator,              /* Used in angle calculation (acos value)  */
        denominator,            /* Used in angle calculation (acos value)  */
        fraction,               /* Numerator divided by denominator        */
        angle,                  /* Angle between direction vectors (deg)   */
        mag_old_dir,            /* Magnitude of direction vector to parent */
        mag_new_dir;            /* Magnitude of direction vector to child  */


/*********************************************
**                RULE  1                  **
*********************************************/

 terrain_elevation = terrain_matrix[child_x][child_y];
 min_altitude      = terrain_elevation + min_above_ground;
 actual_altitude   = child_z * altitude_factor;

 if (min_altitude > actual_altitude)
   return(FALSE);
 if (ceiling < actual_altitude)
   return(FALSE);


/*********************************************
**                RULE  2                  **
*********************************************/

 if (altitude_type[0] == 'M')
   {
    if (actual_altitude < lower_alt_block)
      return(FALSE);
    if (actual_altitude > upper_alt_block)
      return(FALSE);
   }
 else
   {
    if (actual_altitude < (terrain_elevation + lower_alt_block))
      return(FALSE);
    if (actual_altitude > (terrain_elevation + upper_alt_block))
      return(FALSE);
```

```
  }

/*********************************************
**                RULE  3                  **
*********************************************/

 /* use conversion factor 1 mile = 5280 feet */

 /* This will estimate the minimum miles flown by adding to the     */
 /* distance already flown the straight line distance between the   */
 /* location and the target location.  This is a good approximation */
 /* with minimal calculation time needed.                           */

 index    = E_node.number;
 parent_x = E_node.x[index];
 parent_y = E_node.y[index];
 parent_z = E_node.z[index];

 distance = difference(parent_x, parent_y, parent_z, goal_x, goal_y, goal_z)
            * (float) scale_factor;

 miles_flown = (E_node.distance + distance)  / 5280.0;

 if (miles_flown > (float) combat_radius)
   return (FALSE);


/*********************************************
**                RULE  4                  **
*********************************************/

 dir_x    = E_node.vector_x;
 dir_y    = E_node.vector_y;
 dir_z    = E_node.vector_z;
 delta_x  = child_x - parent_x;
 delta_y  = child_y - parent_y;
 delta_z  = child_z - parent_z;

 mag_new_dir = magnitude (delta_x, delta_y, delta_z);
 mag_old_dir = magnitude (dir_x, dir_y, dir_z);

 if (mag_old_dir == 0.0)  /* for first time thru */
   return (TRUE);

 numerator   = (double)(dir_x * delta_x + dir_y * delta_y + dir_z * delta_z);
 denominator = mag_old_dir * mag_new_dir;
 fraction    = numerator / denominator;

  /*  The following tested are performed to protect against round-off     */
  /*  errors.  When testing the calculation of the angle between the two  */
  /*  directional vectors invalid resutls were obtained at the end points */
  /*  of the acos returned values (i.e when the angle was either 0 or pi  */
  /*  radians.)  The acos function requires the input parameter to be of  */
  /*  the type double, so any floats were changed to double, including    */
  /*  the value of the magnitude function.  This seems to solve some of   */
  /*  the errors, but not all of them.  Sometimes the acos routine        */
  /*  returned a correct answer for 0 and pi radian angles and sometimes  */
  /*  and error was returned.  A careful examination of each of the       */
  /*  parameters revealed that when the magnitude of each of the          */
  /*  directional vectors was sqrt(2) a correct answer for the acos        */
  /*  calculation was returned but when the magnitudes were sqrt(3) then  */
  /*  errors were encountered.  It was at this point that the calcuations */
  /*  for the numerator and denominator (for the acos input parameter)    */
  /*  were separated and the %18.20f print statement used.  It was found  */
```

```
 /*  that the sqrt function returned a value with a small error.  When   */
 /*  performing sqrt(3) * sqrt(3) the value 3 was not returned but the   */
 /*  value 2.99999 which caused the error to occur in the acos routine.  */
 /*  Since the input parameter to the acos routine is -1 <= x <= 1 a    */
 /*  value out of this range indicates an overflow/underflow occurred    */
 /*  during the calculation and the value needs to be set to the correct */
 /*  value.  Thus ensuring the acos routine is sent a valid value.       */

 if (fraction >  1.0) fraction =  1.0;
 if (fraction < -1.0) fraction = -1.0;

 angle = acos(fraction);

 /*  The acos function returns an angle in radians, so the following   */
 /*  calculation converts that angle into degrees.                     */

 angle =  angle * 180.0 / M_PI;

 /*  See if the angle between the two directional vectors is less than the */
 /*  turn/climb/dive angle specified for the aircraft (field_of_view).     */

 if (angle <= field_of_view)
   return(TRUE);

 return(FALSE);

} /* end valid_child */



/*****************************************************************
--  Subroutine recursion_search                                  --
--  Passed:    E_node : path being explored                      --
--             depth  : depth of search                          --
--  Returns:   cost                                              --
--  Called By: Find_h_prime                                      --
--  Calls:     difference, valid_child                           --
--  Function:                                                    --
*****************************************************************/
float recursion_search (E_node, depth)
PATH E_node;
int  depth;
{
 int  c, z,                        /* Loop counters                  */
      index;                       /* Parent location in route array */

 US   parent_x,                    /* X coordinate of parent location */
      parent_y,                    /* Y coordinate of parent location */
      parent_z,                    /* Z coordinate of parent location */
      child_x,                     /* X coordinate of child location  */
      child_y,                     /* Y coordinate of child location  */
      child_z;                     /* Z coordinate of child location  */

 float best_cost,                  /* Best overall cost found         */
       cost,                       /* Total cost from parent to child */
       distance,                   /* Distance from parent to child   */
       radar_cost,                 /* Radar cost from parent to child */
       value;                      /* Determine best cost at parent   */

 PATH  temp;                       /* Partial Route sent for recursion*/

/*********************************
*       BEGIN Processing         *
*********************************/
```

```
best_cost = (float) (INFINITY * scale_factor);
index     = E_node.number;
parent_x  = E_node.x[index];
parent_y  = E_node.y[index];
parent_z  = E_node.z[index];

if (depth >= MAX_DEPTH)
  {
   /* The last location in the recursion search has been found thus */
   /* the heuristic to determine the cost between this location and */
   /* the goal location is the straight line distance between the   */
   /* two locations, assuming node radar detection cost.            */

   distance = difference(parent_x, parent_y, parent_z, goal_x, goal_y, goal_z)
              * (float) scale_factor;
   radar_cost = 0.0;
   cost       = (1.0 - WEIGHT_RADAR) * distance + WEIGHT_RADAR * radar_cost;
   return(cost);
  }

if ((parent_x == goal_x) && (parent_y == goal_y) && (parent_z == goal_z))
  {
   /* The goal location has been encountered during the recursion   */
   /* search, thus 0.0 is returned as the heuristic cost of getting */
   /* to the goal location and the recursion bottoms out.           */

   cost = 0.0;
   return(cost);
  }

for (z = -1; z <= 1; z++)
  {
   child_z = parent_z + z;

             /* ensure child is within altitude boundaries */
   if ( (child_z >= 0) && (child_z <= num_z) )
     {
      for (c = 0; c <= 7; c++)
        {
         child_x = parent_x + delta[c][0];
         child_y = parent_y + delta[c][1];

             /* ensure child is within terrain boundaries */
         if ( (child_x > 0) && (child_x <= num_x) &&
              (child_y > 0) && (child_y <= num_y)   )
           {
            if (valid_child(E_node, child_x, child_y, child_z))
              {
               temp             = copy_node(E_node, temp);
               temp.number      = index + 1;
               temp.x[index +1] = child_x;
               temp.y[index +1] = child_y;
               temp.z[index +1] = child_z;
               temp.vector_x    = child_x - parent_x;
               temp.vector_y    = child_y - parent_y;
               temp.vector_z    = child_z - parent_z;
               value      = recursion_search(temp, depth + 1);
               distance   = difference(parent_x, parent_y, parent_z, child_x,
                                       child_y, child_z) * (float) scale_factor;
               radar_cost = radar_matrix[child_x][child_y][child_z]
                            * distance;
               cost       = (1.0 - WEIGHT_RADAR) * distance
                            + WEIGHT_RADAR * radar_cost;
               value      = value + cost;
```

```
                if (value < best_cost)
                  best_cost = value;
               } /* endif valid_child */
             } /* endif child in terrain boundary */
           }  /* end for c */
        }  /* endif z in altitude boundary */
     }  /* end for z */
   return(best_cost);

}  /* end recursion_search */




/*****************************************************************
--  Subroutine Calculate_costs                                  --
--  Passed:    E_node  : path with parent                       --
--             child_x : x coordinate of child node             --
--             child_y : y coordinate of child node             --
--             child_z : z coordinate of child node             --
--  Returns:   temp    : new route with calculated costs        --
--  Called By: Find_children                                    --
--  Calls:     copy_node, difference, Find_h_prime              --
--  Function:  Calculates the costs associated with moving      --
--             from the parent node to the child node along     --
--             with computing the cumlative values.             --
*****************************************************************/
PATH Calculate_costs (E_node, child_x, child_y, child_z)
PATH E_node;
US   child_x, child_y, child_z;
{
 int  index;                        /* Parent location in route array    */

 US   parent_x,                     /* X coordinate for parent location  */
      parent_y,                     /* Y coordinate for parent location  */
      parent_z;                     /* Z coordinate for parent location  */

 float h_prime,                     /* Projected cost from child to goal */
       radar_cost,                  /* Radar cost from parent to child   */
       cost,                        /* Total cost from parent to child   */
       distance;                    /* Distance from parent to child     */

 PATH temp;                         /* Route, with child, being examined */

/*******************************
*       BEGIN PROCESSING       *
*******************************/
 index    = E_node.number;
 parent_x = E_node.x[index];
 parent_y = E_node.y[index];
 parent_z = E_node.z[index];

 temp             = copy_node(E_node, temp);
 temp.x[index +1] = child_x;
 temp.y[index +1] = child_y;
 temp.z[index +1] = child_z;
 temp.number      = index + 1;

 distance = difference(parent_x, parent_y, parent_z, child_x, child_y, child_z)
            * (float) scale_factor;
 temp.distance = temp.distance + distance;

 radar_cost = radar_matrix[child_x][child_y][child_z] * distance;
```

```
 temp.radar = temp.radar + radar_cost;

 cost = (1.0 - WEIGHT_RADAR) * distance + WEIGHT_RADAR * radar_cost;

 temp.vector_x = child_x - parent_x;
 temp.vector_y = child_y - parent_y;
 temp.vector_z = child_z - parent_z;

 temp.g    = temp.g + cost;
 h_prime   = Find_h_prime(temp);
 temp.cost = temp.g + h_prime;

 return(temp);
}  /* end Calculate_costs */




/*****************************************************************
--  Subroutine Find_children                                     --
--  Passed:    E_node : path with parent needing expansion.      --
--  Returns:   none                                              --
--  Called By: main                                              --
--  Calls:     Calculate_costs                                   --
--  Function:  Finds all the children of a given parent.  Only --
--             paths with valid children are sent back to the  --
--             Controller node for inclusion into the OPEN      --
--             list queue.                                      --
*****************************************************************/
void Find_children (E_node)
PATH E_node;
{
 int  c, z,                        /* Loop counters                    */
      index;                       /* Parent location in route array   */

 US   parent_x,                    /* X coordinate for parent location */
      parent_y,                    /* Y coordinate for parent location */
      parent_z,                    /* Z coordinate for parent location */
      child_x,                     /* X coordinate for child location  */
      child_y,                     /* Y coordinate for child location  */
      child_z;                     /* Z coordinate for child location  */

 PATH temp;                        /* Route, with child, being examined */

/*********************************
*       BEGIN Processing        *
*********************************/

 index    = E_node.number;
 parent_x = E_node.x[index];
 parent_y = E_node.y[index];
 parent_z = E_node.z[index];

 for (z = -1; z <= 1; z++)
   {
    child_z = parent_z + z;

              /* ensure child is within altitude boundaries */
    if ( (child_z >= 0) && (child_z <= num_z) )
      {
       for (c = 0; c <= 7; c++)
         {
          child_x = parent_x + delta[c][0];
          child_y = parent_y + delta[c][1];
```

```
            /* ensure child is within terrain boundaries */
        if ( ( (child_x > 0) && (child_x <= num_x) ) &&
            ( (child_y > 0) && (child_y <= num_y) )   )
          {
           if (valid_child(E_node, child_x, child_y, child_z))
            {

             temp = Calculate_costs(E_node, child_x, child_y, child_z);

                /* send route with valid child to the controller*/
             csend (NEW_NODE, &temp, sizeof(temp), CONTROLLER, NODE_PID);

            } /* endif valid_child */
          } /* endif child in terrain boundary */
        }  /* end for c */
      }  /* endif z in altitude boundary */
    }  /* end for z */
  }  /* end Find_children */
```

A.2 Sequential Version

Many of the routines used by the sequential version are exactly the same as used for the parallel version. Instead of providing a listing of the entire source code for the sequential version only the two routines which were modified (main and find_children) are given. The timing information collected by the host program for the parallel version is collected by the search program as well as the display of the solution. A host program was used to interface with the sequential version of the code running on the parallel computer.

```
/*********************************
*       BEGIN Processing   (main)*
*********************************/

 crecv(TERRAIN_FILE, terrainfile, sizeof(terrainfile));
 crecv(RADAR_FILE, radarfile, sizeof(radarfile));
 crecv(ATO_FILE, ATOfile, sizeof(ATOfile));

 start_time = mclock();

    /*  Initialize the delta matrix  */

 delta[0][0] =  0  ; delta[0][1] =  1;
 delta[1][0] =  1  ; delta[1][1] =  1;
 delta[2][0] =  1  ; delta[2][1] =  0;
 delta[3][0] =  1  ; delta[3][1] = -1;
 delta[4][0] =  0  ; delta[4][1] = -1;
 delta[5][0] = -1  ; delta[5][1] = -1;
 delta[6][0] = -1  ; delta[6][1] =  0;
 delta[7][0] = -1  ; delta[7][1] =  1;

 q_init();

 found         = FALSE;
 num_expanded  = 0;
 field_of_view = 60.0;
 ceiling       = 40000;
 combat_radius = 575;

    /*  Read in terrain data                                    */
 fterrain = fopen (terrainfile, "r");
 fscanf (fterrain, "%d", &num_x);
 fscanf (fterrain, "%d", &num_y);
 fscanf (fterrain, "%d", &scale_factor);

 for (y = 1; y <= num_y; y++)
   {
    for (x = 1; x <= num_x; x++)
      {
       fscanf (fterrain, "%d", &elevation);
       terrain_matrix [x][y] = elevation;
      } /* end for y */
   } /* end for x */
 fclose (fterrain);

    /* Read in radar detection data              */

 fradar = fopen (radarfile, "r");
```

```
fscanf (fradar, "%d", &num_x);
fscanf (fradar, "%d", &num_y);
fscanf (fradar, "%d", &num_z);
fscanf (fradar, "%d", &altitude_factor);

for (z = 0; z < num_z; z++)
  {
   for (y = 1; y <= num_y; y++)
     {
      for (x = 1; x <= num_x; x++)
        {
         fscanf (fterrain, "%f", &detection);
         radar_ma·rix [x][y][z] = detection;
        } /* end for y */
     } /* end for x */
  }  /* end for z */
fclose (fradar);


   /* Read in Air Tasking Order (ATO) information       */
fATO = fopen (ATOfile, "r");
fscanf (fATO, "%s",  mission);
fscanf (fATO, "%hu", &base_x);
fscanf (fATO, "%hu", &base_y);
fscanf (fATO, "%hu", &base_altitude);
fscanf (fATO, "%hu", &target_x);
fscanf (fATO, "%hu", &target_y);
fscanf (fATO, "%hu", &target_altitude);
fscanf (fATO, "%d",  &min_above_ground);
fscanf (fATO, "%d",  &lower_alt_block);
fscanf (fATO, "%d",  &upper_alt_block);
fscanf (fATO, "%s",  altitude_type);
fclose (fATO);

target_z = target_altitude / altitude_factor;
base_z   = base_altitude / altitude_factor;

goal_x  = target_x;
goal_y  = target_y;
goal_z  = target_z;
start_x = base_x;
start_y = base_y;
start_z = base_z;

  /* Place initial node (base node) into the OPEN list queue    */
E_node.number   = 1;
E_node.x[1]     = start_x;
E_node.y[1]     = start_y;
E_node.z[1]     = start_z;
E_node.vector_x = 0;
E_node.vector_y = 0;
E_node.vector_z = 0;
E_node.distance = 0.0;
E_node.radar    = 0.0;
E_node.g        = 0.0;
E_node.cost     = 0.0;
insert_priority(E_node);

init_time = mclock() - start_time;

/* Find initial path so as to create a bound on the search. */
/* This path is not guarenteed to be a valid path.          */

Find_initial_path(E_node);
printf(" Found an initial route with a cost of %f \n", best.cost);
```

```
 init_path_time = mclock() - start_time - init_time;

 temp_time = mclock();

 while(q_status != EMPTY)
   {
    E_node = delete_q();
    num_expanded ++;
    Find_children(E_node);
   } /* endwhile */

 search_time = mclock() - temp_time;
 run_time    = mclock() - start_time;

 print_route(best);

 printf("\n\t     ***  Timing Information  ***\n\n");
 printf("Initialize = %9.3f (sec) \t Find Initial Route = %9.3f (sec) \n",
          (float)init_time/1000.0, (float)init_path_time/1000.0);
 printf("Searching  = %9.3f (sec) \t Total Execution    = %9.3f (sec) \n\n",
          (float)search_time/1000.0, (float)run_time/1000.0);

 printf("** %d Nodes Expanded \n\n", num_expanded);

 csend (DONE_TYPE, &dummy, sizeof(dummy), myhost(), Host_PID);

}  /* end main  */


/*****************************************************************
--  Subroutine Find_children                                     --
--  Passed:    E_node : path with parent needing expansion.      --
--  Returns:   none                                              --
--  Called By: main                                              --
--  Calls:     Calculate_costs                                   --
--  Function:  Finds all the children of a given parent.  Only --
--             paths with valid children are sent back to the  --
--             Controller node for inclusion into the OPEN      --
--             list queue.                                      --
*****************************************************************/
void Find_children (E_node)
PATH E_node;
{
 int  c, z,                        /* Loop counters                    */
      index;                       /* Parent location in route array   */

 US   parent_x,                    /* X coordinate for parent location */
      parent_y,                    /* Y coordinate for parent location */
      parent_z,                    /* Z coordinate for parent location */
      child_x,                     /* X coordinate for child location  */
      child_y,                     /* Y coordinate for child location  */
      child_z;                     /* Z coordinate for child location  */

 PATH temp;                        /* Route, with child, being examined */

/*********************************
*       BEGIN Processing         *
*********************************/

 index    = E_node.number;
 parent_x = E_node.x[index];
 parent_y = E_node.y[index];
 parent_z = E_node.z[index];
```

```
 for (z = -1; z <= 1; z++)
   {
    child_z = (US) ((int) parent_z + z);

              /* ensure child is within altitude boundaries */
    if ( (child_z >= 0) && (child_z <= num_z) )
      {
       for (c = 0; c <= 7; c++)
         {
          child_x = (US) ((int) parent_x + delta[c][0]);
          child_y = (US) ((int) parent_y + delta[c][1]);

          /*printf("Looking at child(%d %d %d)\n",child_x,child_y,child_z);*/

               /* ensure child is within terrain boundaries */
          if ( ( (child_x > 0) && (child_x <= num_x) ) &&
               ( (child_y > 0) && (child_y <= num_y) )   )
            {
             if (valid_child(E_node, child_x, child_y, child_z))
               {

                temp = Calculate_costs(E_node, child_x, child_y, child_z);

                if (temp.cost < best.cost);
                  {
                   insert_priority(temp);

                   if ( (temp.x[temp.number] == goal_x) &&
                        (temp.y[temp.number] == goal_y) &&
                        (temp.z[temp.number] == goal_z) )
                     {
                      printf("\n  A Solution Path Has Been Found \n\n");
                      best = copy_node(temp, best);
                      prune_q();
                     } /* endif goal reached */
                  } /* endif temp.cost < best.cost */
               } /* endif valid_child */
            } /* endif child in terrain boundary */
         }  /* end for c */
      }  /* endif z in altitude boundary */
   }  /* end for z */
}  /* end Find_children */
```

A.3 Support Files

A.3.1 Header File.

```
/*****************************************************************
*                        PATH HEADER FILE                        *
*  Date: 1 Sept 1992                                             *
*  Function:  The following header file defines the solution     *
*             vector and the priority queue functions needed     *
*             by the parallel mission routing system.            *
*****************************************************************/

/*************************************************
*       Define Constants                         *
*************************************************/
#define  US              unsigned short
#define  TRUE                 1            /* True is defined as integer 1 */
#define  FALSE                0            /* False is defined as integer 0*/
#define  MAX_PATH_LENGTH    150            /* Maximum entries in route     */
#define  MAX_MATRIX_SIZE    100            /* Maximum size of terrain/radar*/
#define  MAX_ALT_SIZE        52            /* Maximum altitude entries     */
#define  Q_SIZE            9500            /* Size of the OPEN list queue  */
#define  INFINITY          9999
#define  MAX_CUBE_SIZE        8            /* Largest cube size possible   */
#define  MAX_DEPTH            5            /* Max depth of recursion       */
#define  WEIGHT_RADAR         0.8          /* Weighting of radar detection */
#define  REDUCE_FACTOR        0.6          /* Beam search reduction factor */


/*************************************************
*       Define Flags                             *
*************************************************/
#define  AVAIL               -1            /* Node Available for work      */
#define  BUSY                -2            /* Node is busy                 */
#define  EOQ                 -1            /* End of Linked List Marker    */
#define  EMPTY                1            /* Queue Empty flag (status)    */
#define  Q_BUSY               2            /* Queue Busy flag (status)     */
#define  FULL                 3            /* Queue Full flag (status)     */

/*************************************************
*       Define PID's and Node Assignments        *
*************************************************/
#define  NODE_PID             0            /* Node Process ID              */
#define  ALL_NODES           -1            /* Code to send to all nodes    */
#define  ALL_PIDS             0            /* All processes                */
#define  CONTROLLER           0            /* Controller - Node 0          */
#define  Host_PID             0            /* Host Process ID              */

/*************************************************
*       Define Message Types                     *
*************************************************/
#define  BEST_TYPE           30            /* Message containing a solution*/
#define  TIME_TYPE           40            /* Timing information message   */
#define  EXPAND_NODE         50            /* A partial route to expand    */
#define  DONE_TYPE           60            /* Done with the problem        */
#define  WORK_REQUEST        80            /* Message requesting work      */
#define  NEW_NODE            90            /* Expanded route, to controller*/
#define  WORK_TYPE          110            /* Total work time of worker    */
#define  NUM_TYPE           120            /* Number of locations expanded */
#define  EXPANDED           200     /* Message identifier           */
#define  TERRAIN_FILE       220            /* Terrain filename message     */
#define  ATO_FILE           230            /* ATO filename message         */
#define  RADAR_FILE         240            /* Radar filename message       */
#define  PLANE_FILE         250            /* Plane filename message       */
```

```
/***********************************************
*       Path  Information  Record              *
***********************************************/
typedef struct {
      int    number;                    /* Number of entries in the route  */
      US     x [MAX_PATH_LENGTH+1];     /* Vector of x locations           */
      US     y [MAX_PATH_LENGTH+1];     /* Vector of y locations           */
      US     z [MAX_PATH_LENGTH+1];     /* Vector of z locations           */
      int    vector_x;                  /* Direction vector in x direction */
      int    vector_y;                  /* Direction vector in y direction */
      int    vector_z;                  /* Direction vector in z direction */
      float  distance;                  /* Cumlative distance of the route */
      float  radar;                     /* Cumlative radar detection cost  */
      float  g;                         /* Cost of the given route         */
      float  cost;                      /* Calculated cost (f') of route   */
      int    link;                      /* Forward Links for the OPEN list */
} PATH;
```

A.3.2 Make File.

```
# DATE: 23 Oct 92
# VERSION: 1.0
# TITLE: Makefile for creating and updating the parallelized A* search
#        code for execution on the iPSC/2 hypercube.
# FILENAME: Makefile
# COORDINATOR: Capt Joel Garmon
# PROJECT: EENG 656 Parallel Programming
# OPERATING SYSTEM: XENIX
# HISTORY:
# 05/31/91 Capt Joel Garmon
#          - Initial version
# 10/23/92 Capt James Grimm


help  :
        @echo "This A* Search makefile supports the following:"
        @echo "  make router  - creates executable host, control and worker."
        @echo "  make host    - creates the host part only."
        @echo "  make worker  - creates the node worker part only."
        @echo "  make control - creates the master controller for the nodes."
        @echo "  make both    - creates the controller and worker parts."
        @echo "  make clean   - removes intermediate files."

router : host control  worker

host  : host.o
        cc  -o host host.o -host

control  : control.o
        cc  -o control control.o -node

worker : worker.o
        cc -o worker worker.o -node -lm

both   :  control   worker

clean :
        rm  *.o host control worker
```

A.3.3 Test Angle Calculation.

```
/*********************************************************************
--   DATE: 16 Oct 92                                                 --
--   VERSION: 1.0                                                    --
--                                                                   --
--   TITLE:  Test Calculation of the Angle Between Two Vectors        --
--   FILENAME:  TEST-TRIG.C                                          --
--   AUTHOR: Capt James J. Grimm  II                                 --
--   COORDINATOR:  R. Norris                                         --
--   PROJECT:  Thesis Research Project                               --
--   OPERATING SYSTEM:  System V                                     --
--   LANGUAGE: C                                                     --
--   FILE PROCESSING: Compile & link with stdio.h, path.h, and math.h --
--                    Ensure the program is linked with the math     --
--                    library using the -lm switch.                  --
--   FUNCTION:  This program finds the angle between to directional  --
--              vectors.  The parent location along with the vector  --
--              used to get to the parent are specified.  Each child --
--              location is found and all the information used to    --
--              determine if the child can be reached from the parent --
--              are displayed.  This code was taken from the worker  --
--              program used by the Parallel Mission Routing software. --
--                                                                   --
--   HISTORY:  1.0 Written by Capt James Grimm to debug a calcualtion --
--                 problem while porting the Parallel Mission Routing --
--                 software from the iPSC/2 to the iPSC/860.         --
*********************************************************************/

/*************************************************
*                Header  Files                   *
*************************************************/
#include  <stdio.h>                  /* Standard IO                */
#include  "/usr/include/math.h"      /* Standard math library      */
#include  "path.h"

/*******************************************************/
/*                        Main Program                 */
/*******************************************************/
main()
{
 int delta[8][2], z, c, d_x, d_y, d_z, del_x, del_y, del_z, num_value;

 US p_x, p_y, p_z, c_x, c_y, c_z, old_top, new_top;

 double numerator,          /* Used in angle calculation (acos value)  */
        denominator,        /* Used in angle calculation (acos value)  */
        fraction,           /* Numerator divided by denominator        */
        angle,              /* Angle between direction vectors (deg)   */
        mag_old,            /* Magnitude of direction vector to parent */
        mag_new;            /* Magnitude of direction vector to child  */

 delta[0][0] =  0  ; delta[0][1] =  1;
 delta[1][0] =  1  ; delta[1][1] =  1;
 delta[2][0] =  1  ; delta[2][1] =  0;
 delta[3][0] =  1  ; delta[3][1] = -1;
 delta[4][0] =  0  ; delta[4][1] = -1;
 delta[5][0] = -1  ; delta[5][1] = -1;
 delta[6][0] = -1  ; delta[6][1] =  0;
 delta[7][0] = -1  ; delta[7][1] =  1;

 d_x =  0;            /* x component of directional vector */
 d_y = -1;            /* y component of directional vector */
 d_z = -1;            /* z component of directional vector */
 p_x = 17;            /* Parent's x coordinate  */
```

```
p_y = 16;            /* Parent's y coordinate  */
p_z =  7;            /* Parent's z coordinate  */

printf("\n Starting testing \n\n");

for(z = -1; z <= 1; z++)
 {
  c_z = (US) ((int) p_z + z);

  for(c = 0; c <= 7; c++)
   {
    c_x = (US) ((int) p_x + delta[c][0]);
    c_y = (US) ((int) p_y + delta[c][1]);

    printf(" P(%d %d %d)  C(%d %d %d) ", p_x, p_y, p_z, c_x, c_y, c_z);
    del_x  = (int) c_x - p_x;
    del_y  = (int) c_y - p_y;
    del_z  = (int) c_z - p_z;

    printf(" ND(%2d %2d %2d) ", del_x, del_y, del_z);

    new_top = (del_x * del_x + del_y * del_y + del_z * del_z);
    old_top = (d_x * d_x + d_y * d_y + d_z * d_z);

    mag_new = sqrt ((double) new_top);
    mag_old = sqrt ((double) old_top);

    num_value = (d_x * del_x + d_y * del_y + del_z * d_z);

    printf(" NT(%d) OT(%d) NV(%2d) ", new_top, old_top, num_value);

    numerator   = (double) num_value;
    denominator = mag_old * mag_new;
    fraction    = numerator / denominator;
    if (fraction >  1.0) fraction =  1.0;
    if (fraction < -1.0) fraction = -1.0;

    printf(" Num(%4.1f) Den(%6.4f) Fract(%7.4f) ",
             numerator, denominator, fraction);
    printf(" MO(%5.3f) MN(%5.3f) ", mag_old, mag_new);

    angle= acos(fraction);

    angle =  angle * 180.0 / M_PI;

    printf(" A(%20.16f) ", angle);
    if (angle <= 60.0)
      printf(" TRUE");

    printf("\n");
   }
  }
}
```

Appendix B. *Input Data*

B.1 Terrain Data

25 19 1000

1700	1775	2005	2010	2000	2000	2020	2450	3150	2350	2270	2320	2505	2590	2650	3000	2890	2675	2500	2270	2500	2905	2960	2470	2000
1700	2000	2020	2070	2495	3120	3110	3135	2750	2550	2690	2980	3100	3050	3150	3455	3535	3020	2705	2510	2700	3150	3200	3210	2890
1815	2100	2090	2200	3170	3700	3465	2995	2520	2610	2890	3045	3285	3580	3805	4200	4410	3050	3620	2500	2735	3335	3670	3480	3105
2090	2110	2170	2310	3250	4305	3250	2750	2600	2700	2875	3190	3500	3800	4120	4885	4705	2995	3485	2515	2625	3215	3740	3595	3085
2100	2200	2300	2995	3590	3555	3100	2515	2900	3470	3275	3315	3605	4195	4700	5500	3755	3495	3340	2560	2600	3200	3700	3710	3100
2250	2300	2530	3215	3700	3420	2850	2500	3205	3700	3300	3005	3710	4300	5500	4500	3150	3600	3400	2505	2605	3130	3305	3600	3200
2100	2370	2690	3090	3700	3430	2895	2510	3500	4205	3360	3005	3560	4500	5000	3300	3115	3550	3650	3000	2700	2700	3100	3300	3460
2050	2405	2600	3200	3700	3500	2880	2600	3395	4200	3610	3360	3300	3900	3515	3270	3250	3050	3095	3500	3590	2500	2700	3100	3540
2000	2320	2775	3220	3790	3460	3080	2690	3000	3900	3775	3420	3190	3600	3620	3795	3815	3910	3590	3225	3400	3585	2600	2700	3225
1950	2200	2545	3140	3710	3700	3120	2650	2770	3600	3900	3400	3070	3790	4500	4750	5010	4690	4050	3470	3390	3600	2500	2700	3000
1900	2015	2450	2720	3100	3680	3200	2650	2550	3225	4200	3450	3120	3980	5700	5920	5000	4600	4000	3200	3500	3600	2550	2700	2910
1850	2000	2350	2700	2800	2950	3220	2790	2650	3000	4200	3450	3100	4100	6000	4800	4595	4200	3500	3200	4000	3400	2590	2990	3200
1800	1955	2290	2650	2800	2800	3100	2800	2600	2755	3890	3450	3100	4000	6000	4600	4525	3770	3000	3550	4050	3200	2640	3400	2990
1750	1900	2190	2800	3150	2900	2770	2700	2600	2540	3500	3225	3100	4000	5000	4200	3840	3450	3400	4120	3760	2975	3020	3040	2650
1710	1870	2100	2450	3200	3390	3400	3200	3000	2540	2780	3090	3100	3555	4105	3800	3500	3115	3540	3605	3205	2700	3210	3000	2650
1680	1810	2220	2350	2850	3400	4000	3900	3500	2810	2605	2760	2930	3400	3820	3500	3140	3200	3150	2560	2680	2950	3285	2710	2600
1625	1710	1900	2235	2900	3500	4500	3600	3385	2810	2515	2600	2900	3250	3370	3110	3000	3000	2500	2750	2750	3200	2885	2640	2550
1600	1700	1880	2210	2500	3430	3715	3300	3200	2800	2550	2580	2900	3290	3250	2800	2800	2560	2605	2600	2785	2810	2620	2500	2400
1585	1645	1700	2170	2260	2500	2770	2700	2700	2465	2320	2500	2740	2890	2905	2775	2540	2580	2600	2540	2510	2475	2425	2395	2320

B.2 Radar Data

```
25 19 51 1000

0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0

0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0

0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
```

0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0

0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.1 0.2 0.0 0.0 0.0 0.2 0.2 0.4 0.6 0.4 0.2 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.5 0.4 0.8 0.0 0.0 0.0 0.2
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.6 0.9 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.2 0.0 0.0 0.0 0.0 0.0 0.0 0.9 0.0 0.9 0.8 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.6 0.4 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.3 0.2 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.3 0.3 0.3 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.3 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.4 0.3 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.5 0.4 0.3
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.7 0.6 0.4
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.7 0.6 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.7 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.7 0.0 0.0 0.3
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.6 0.0 0.0 0.2
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.3 0.5 0.0 0.0 0.0 0.0 0.0 0.0 0.4 0.0 0.4 0.0 0.2 0.1
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.3 0.5 0.0 0.0 0.0 0.5 0.3 0.2 0.3 0.0 0.0 0.2 0.1 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.3 0.5 0.0 0.0 0.7 0.5 0.3 0.1 0.2 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.3 0.5 0.5 0.5 0.3 0.0 0.0 0.1 0.0 0.1 0.0 0.0 0.0

0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.2 0.2 0.0 0.0 0.0 0.2 0.2 0.4 0.6 0.4 0.2 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.3 0.3 0.4 0.3 0.0 0.3 0.5 0.4 0.8 0.9 0.8 0.4 0.2
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.2 0.4 0.4 0.6 0.0 0.0 0.6 0.9 0.6 0.9 1.0 0.9 0.6 0.3
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.2 0.3 0.5 0.5 0.0 0.0 0.0 0.9 1.0 0.9 0.8 0.9 0.8 0.4 0.2
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.1 0.3 0.4 0.6 0.0 0.0 0.0 0.6 0.6 0.9 0.6 0.4 0.6 0.4 0.2 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.2 0.3 0.5 0.0 0.0 0.0 0.0 0.7 0.6 0.5 0.3 0.2 0.3 0.2 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.3 0.0 0.6 0.0 0.0 0.7 0.6 0.5 0.3 0.3 0.3 0.3 0.1 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.2 0.3 0.4 0.5 0.7 0.6 0.3 0.2 0.5 0.4 0.4 0.4 0.5 0.2 0.1
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.2 0.3 0.4 0.6 0.4 0.3 0.0 0.5 0.6 0.6 0.6 0.5 0.4 0.2
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.2 0.3 0.4 0.5 0.0 0.0 0.0 0.0 0.0 0.7 0.7 0.7 0.6 0.5 0.3
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.4 0.5 0.6 0.0 0.0 0.0 0.0 0.7 0.8 0.9 0.8 0.7 0.6 0.4
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.4 0.6 0.0 0.0 0.0 0.0 0.0 0.7 0.9 1.0 0.9 0.7 0.6 0.4
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.1 0.3 0.4 0.5 0.6 0.0 0.0 0.0 0.6 0.7 0.8 0.0 0.8 0.7 0.6 0.4
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.2 0.3 0.4 0.5 0.0 0.0 0.4 0.5 0.6 0.0 0.0 0.7 0.6 0.5 0.3
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.3 0.5 0.0 0.5 0.3 0.4 0.0 0.6 0.0 0.6 0.5 0.4 0.2
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.3 0.5 0.7 0.8 0.7 0.5 0.3 0.3 0.4 0.0 0.4 0.3 0.2 0.1
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.3 0.5 0.8 1.0 0.8 0.5 0.3 0.2 0.3 0.0 0.3 0.2 0.1 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.3 0.5 0.7 0.8 0.7 0.5 0.3 0.2 0.2 0.0 0.2 0.2 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.3 0.5 0.5 0.5 0.3 0.0 0.0 0.1 0.0 0.1 0.0 0.0 0.0

0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.2 0.2 0.0 0.0 0.0 0.2 0.2 0.4 0.6 0.4 0.2 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.3 0.3 0.4 0.3 0.0 0.3 0.5 0.4 0.8 0.9 0.8 0.4 0.2
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.2 0.4 0.4 0.6 0.4 0.0 0.6 0.9 0.6 0.9 1.0 0.9 0.6 0.3
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.2 0.3 0.5 0.5 0.7 0.5 0.5 0.9 1.0 0.9 0.8 0.9 0.8 0.4 0.2
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.1 0.3 0.4 0.6 0.7 0.9 0.0 0.6 0.6 0.9 0.6 0.4 0.6 0.4 0.2 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.2 0.3 0.5 0.7 0.9 0.0 0.9 0.7 0.6 0.5 0.3 0.2 0.3 0.2 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.1 0.3 0.4 0.6 0.7 0.9 0.7 0.6 0.5 0.3 0.3 0.3 0.3 0.3 0.1 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.2 0.3 0.4 0.5 0.7 0.6 0.3 0.2 0.5 0.4 0.4 0.4 0.5 0.2 0.1
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.2 0.3 0.4 0.6 0.4 0.3 0.4 0.5 0.6 0.6 0.6 0.5 0.4 0.2
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.2 0.3 0.4 0.5 0.6 0.2 0.0 0.5 0.6 0.7 0.7 0.7 0.6 0.5 0.3
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.1 0.3 0.4 0.5 0.6 0.0 0.0 0.4 0.6 0.7 0.8 0.9 0.8 0.7 0.6 0.4
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.2 0.3 0.4 0.6 0.8 0.0 0.8 0.5 0.6 0.7 0.9 1.0 0.9 0.7 0.6 0.4
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.1 0.3 0.4 0.5 0.6 0.0 0.6 0.4 0.6 0.7 0.8 0.9 0.8 0.7 0.6 0.4
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.2 0.3 0.4 0.5 0.6 0.5 0.4 0.5 0.6 0.7 0.7 0.7 0.6 0.5 0.3
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.3 0.5 0.5 0.5 0.3 0.4 0.5 0.6 0.6 0.6 0.5 0.4 0.2
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.3 0.5 0.7 0.8 0.7 0.5 0.3 0.3 0.4 0.5 0.4 0.3 0.2 0.1
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.3 0.5 0.8 1.0 0.8 0.5 0.3 0.2 0.3 0.4 0.3 0.2 0.1 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.3 0.5 0.7 0.8 0.7 0.5 0.3 0.2 0.2 0.3 0.2 0.2 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.3 0.5 0.5 0.5 0.3 0.0 0.0 0.1 0.2 0.1 0.0 0.0 0.0

0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.2 0.2 0.0 0.0 0.0 0.2 0.2 0.4 0.6 0.4 0.2 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.3 0.3 0.4 0.3 0.0 0.3 0.5 0.4 0.8 0.9 0.8 0.4 0.2

0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.4	0.4	0.6	0.4	0.0	0.6	0.9	0.6	0.9	1.0	0.9	0.6	0.3
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.5	0.5	0.7	0.5	0.5	0.9	1.0	0.9	0.8	0.9	0.8	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.6	0.7	0.9	0.7	0.6	0.6	0.9	0.6	0.4	0.6	0.4	0.2	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.5	0.7	0.9	1.0	0.9	0.7	0.6	0.5	0.3	0.2	0.3	0.2	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.6	0.7	0.9	0.7	0.6	0.5	0.3	0.3	0.3	0.3	0.3	0.1	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.5	0.7	0.6	0.5	0.2	0.5	0.4	0.4	0.4	0.5	0.2	0.1
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.6	0.4	0.3	0.4	0.5	0.6	0.6	0.6	0.5	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.5	0.6	0.5	0.3	0.5	0.6	0.7	0.7	0.7	0.6	0.5	0.3
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.5	0.6	0.8	0.6	0.4	0.6	0.7	0.8	0.9	0.8	0.7	0.6	0.4
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.6	0.8	1.0	0.8	0.5	0.6	0.7	0.9	1.0	0.9	0.7	0.6	0.4
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.5	0.6	0.8	0.6	0.4	0.6	0.7	0.8	0.9	0.8	0.7	0.6	0.4
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.5	0.6	0.5	0.4	0.5	0.6	0.7	0.7	0.7	0.6	0.5	0.3
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.5	0.5	0.3	0.4	0.5	0.6	0.6	0.6	0.5	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.7	0.8	0.7	0.5	0.3	0.3	0.4	0.5	0.4	0.3	0.2	0.1
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.8	1.0	0.8	0.5	0.3	0.2	0.3	0.4	0.3	0.2	0.1	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.7	0.8	0.7	0.5	0.3	0.2	0.2	0.3	0.2	0.2	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.5	0.5	0.3	0.0	0.0	0.1	0.2	0.1	0.0	0.0	0.0

0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.2	0.0	0.0	0.0	0.2	0.2	0.4	0.6	0.4	0.2	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.3	0.4	0.3	0.0	0.3	0.5	0.4	0.8	0.9	0.8	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.4	0.4	0.6	0.4	0.0	0.6	0.9	0.6	0.9	1.0	0.9	0.6	0.3
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.5	0.5	0.7	0.5	0.5	0.9	1.0	0.9	0.8	0.9	0.8	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.6	0.7	0.9	0.7	0.6	0.6	0.9	0.6	0.4	0.6	0.4	0.2	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.5	0.7	0.9	1.0	0.9	0.7	0.6	0.5	0.3	0.2	0.3	0.2	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.6	0.7	0.9	0.7	0.6	0.5	0.3	0.3	0.3	0.3	0.3	0.1	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.5	0.7	0.6	0.5	0.2	0.5	0.4	0.4	0.4	0.5	0.2	0.1
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.6	0.4	0.3	0.4	0.5	0.6	0.6	0.6	0.5	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.5	0.6	0.5	0.3	0.5	0.6	0.7	0.7	0.7	0.6	0.5	0.3
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.5	0.6	0.8	0.6	0.4	0.6	0.7	0.8	0.9	0.8	0.7	0.6	0.4
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.6	0.8	1.0	0.8	0.5	0.6	0.7	0.9	1.0	0.9	0.7	0.6	0.4
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.5	0.6	0.8	0.6	0.4	0.6	0.7	0.8	0.9	0.8	0.7	0.6	0.4
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.5	0.6	0.5	0.4	0.5	0.6	0.7	0.7	0.7	0.6	0.5	0.3
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.5	0.5	0.3	0.4	0.5	0.6	0.6	0.6	0.5	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.7	0.8	0.7	0.5	0.3	0.3	0.4	0.5	0.4	0.3	0.2	0.1
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.8	1.0	0.8	0.5	0.3	0.2	0.3	0.4	0.3	0.2	0.1	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.7	0.8	0.7	0.5	0.3	0.2	0.2	0.3	0.2	0.2	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.5	0.5	0.3	0.0	0.0	0.1	0.2	0.1	0.0	0.0	0.0

0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.2	0.0	0.0	0.0	0.2	0.2	0.4	0.6	0.4	0.2	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.3	0.4	0.3	0.0	0.3	0.5	0.4	0.8	0.9	0.8	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.4	0.4	0.6	0.4	0.0	0.6	0.9	0.6	0.9	1.0	0.9	0.6	0.3
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.5	0.5	0.7	0.5	0.5	0.9	1.0	0.9	0.8	0.9	0.8	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.6	0.7	0.9	0.7	0.6	0.6	0.9	0.6	0.4	0.6	0.4	0.2	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.5	0.7	0.9	1.0	0.9	0.7	0.6	0.5	0.3	0.2	0.3	0.2	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.6	0.7	0.9	0.7	0.6	0.5	0.3	0.3	0.3	0.3	0.3	0.1	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.5	0.7	0.6	0.5	0.2	0.5	0.4	0.4	0.4	0.5	0.2	0.1
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.6	0.4	0.3	0.4	0.5	0.6	0.6	0.6	0.5	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.5	0.6	0.5	0.3	0.5	0.6	0.7	0.7	0.7	0.6	0.5	0.3
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.5	0.6	0.8	0.6	0.4	0.6	0.7	0.8	0.9	0.8	0.7	0.6	0.4
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.6	0.8	1.0	0.8	0.5	0.6	0.7	0.9	1.0	0.9	0.7	0.6	0.4
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.5	0.6	0.8	0.6	0.4	0.6	0.7	0.8	0.9	0.8	0.7	0.6	0.4
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.5	0.6	0.5	0.4	0.5	0.6	0.7	0.7	0.7	0.6	0.5	0.3
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.5	0.5	0.3	0.4	0.5	0.6	0.6	0.6	0.5	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.7	0.8	0.7	0.5	0.3	0.3	0.4	0.5	0.4	0.3	0.2	0.1
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.8	1.0	0.8	0.5	0.3	0.2	0.3	0.4	0.3	0.2	0.1	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.7	0.8	0.7	0.5	0.3	0.2	0.2	0.3	0.2	0.2	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.5	0.5	0.3	0.0	0.0	0.1	0.2	0.1	0.0	0.0	0.0

0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.2	0.0	0.0	0.0	0.2	0.2	0.4	0.6	0.4	0.2	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.3	0.4	0.3	0.0	0.3	0.5	0.4	0.8	0.9	0.8	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.4	0.4	0.6	0.4	0.0	0.6	0.9	0.6	0.9	1.0	0.9	0.6	0.3
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.5	0.5	0.7	0.5	0.5	0.9	1.0	0.9	0.8	0.9	0.8	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.6	0.7	0.9	0.7	0.6	0.6	0.9	0.6	0.4	0.6	0.4	0.2	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.5	0.7	0.9	1.0	0.9	0.7	0.6	0.5	0.3	0.2	0.3	0.2	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.6	0.7	0.9	0.7	0.6	0.5	0.3	0.3	0.3	0.3	0.3	0.1	0.0

0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.5	0.7	0.6	0.5	0.2	0.5	0.4	0.4	0.4	0.5	0.2	0.1
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.6	0.4	0.3	0.4	0.5	0.6	0.6	0.6	0.5	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.5	0.6	0.5	0.3	0.5	0.6	0.7	0.7	0.7	0.6	0.5	0.3
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.5	0.6	0.8	0.6	0.4	0.6	0.7	0.8	0.9	0.8	0.7	0.6	0.4
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.6	0.8	1.0	0.8	0.5	0.6	0.7	0.9	1.0	0.9	0.7	0.6	0.4
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.5	0.6	0.8	0.6	0.4	0.6	0.7	0.8	0.9	0.8	0.7	0.6	0.4
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.5	0.6	0.5	0.4	0.5	0.6	0.7	0.7	0.7	0.6	0.5	0.3
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.5	0.5	0.3	0.4	0.5	0.6	0.6	0.6	0.5	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.7	0.8	0.7	0.5	0.3	0.3	0.4	0.5	0.4	0.3	0.2	0.1
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.8	1.0	0.8	0.5	0.3	0.2	0.3	0.4	0.3	0.2	0.1	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.7	0.8	0.7	0.5	0.3	0.2	0.2	0.3	0.2	0.2	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.5	0.5	0.3	0.0	0.0	0.1	0.2	0.1	0.0	0.0	0.0

0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.2	0.0	0.0	0.0	0.2	0.2	0.4	0.6	0.4	0.2	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.3	0.4	0.3	0.0	0.3	0.5	0.4	0.8	0.9	0.8	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.4	0.4	0.6	0.4	0.0	0.6	0.9	0.6	0.9	1.0	0.9	0.6	0.3
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.5	0.5	0.7	0.5	0.5	0.9	1.0	0.9	0.8	0.9	0.8	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.6	0.7	0.9	0.7	0.6	0.6	0.9	0.6	0.4	0.6	0.4	0.2	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.5	0.7	0.9	1.0	0.9	0.7	0.6	0.5	0.3	0.2	0.3	0.2	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.6	0.7	0.9	0.7	0.6	0.5	0.3	0.3	0.3	0.3	0.3	0.1	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.5	0.7	0.6	0.5	0.2	0.5	0.4	0.4	0.4	0.5	0.2	0.1
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.6	0.4	0.3	0.4	0.5	0.6	0.6	0.6	0.5	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.5	0.6	0.5	0.3	0.5	0.6	0.7	0.7	0.7	0.6	0.5	0.3
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.5	0.6	0.8	0.6	0.4	0.6	0.7	0.8	0.9	0.8	0.7	0.6	0.4
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.6	0.8	1.0	0.8	0.5	0.6	0.7	0.9	1.0	0.9	0.7	0.6	0.4
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.5	0.6	0.8	0.6	0.4	0.6	0.7	0.8	0.9	0.8	0.7	0.6	0.4
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.5	0.6	0.5	0.4	0.5	0.6	0.7	0.7	0.7	0.6	0.5	0.3
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.5	0.5	0.3	0.4	0.5	0.6	0.6	0.6	0.5	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.7	0.8	0.7	0.5	0.3	0.3	0.4	0.5	0.4	0.3	0.2	0.1
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.8	1.0	0.8	0.5	0.3	0.2	0.3	0.4	0.3	0.2	0.1	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.7	0.8	0.7	0.5	0.3	0.2	0.2	0.3	0.2	0.2	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.5	0.5	0.3	0.0	0.0	0.1	0.2	0.1	0.0	0.0	0.0

0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.2	0.0	0.0	0.0	0.2	0.2	0.4	0.6	0.4	0.2	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.3	0.4	0.3	0.0	0.3	0.5	0.4	0.8	0.9	0.8	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.4	0.4	0.6	0.4	0.0	0.6	0.9	0.6	0.9	1.0	0.9	0.6	0.3
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.5	0.5	0.7	0.5	0.5	0.9	1.0	0.9	0.8	0.9	0.8	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.6	0.7	0.9	0.7	0.6	0.6	0.9	0.6	0.4	0.6	0.4	0.2	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.5	0.7	0.9	1.0	0.9	0.7	0.6	0.5	0.3	0.2	0.3	0.2	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.6	0.7	0.9	0.7	0.6	0.5	0.3	0.3	0.3	0.3	0.3	0.1	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.5	0.7	0.6	0.5	0.2	0.5	0.4	0.4	0.4	0.5	0.2	0.1
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.6	0.4	0.3	0.4	0.5	0.6	0.6	0.6	0.5	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.5	0.6	0.5	0.3	0.5	0.6	0.7	0.7	0.7	0.6	0.5	0.3
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.5	0.6	0.8	0.6	0.4	0.6	0.7	0.8	0.9	0.8	0.7	0.6	0.4
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.6	0.8	1.0	0.8	0.5	0.6	0.7	0.9	1.0	0.9	0.7	0.6	0.4
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.5	0.6	0.8	0.6	0.4	0.6	0.7	0.8	0.9	0.8	0.7	0.6	0.4
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.5	0.6	0.5	0.4	0.5	0.6	0.7	0.7	0.7	0.6	0.5	0.3
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.5	0.5	0.3	0.4	0.5	0.6	0.6	0.6	0.5	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.7	0.8	0.7	0.5	0.3	0.3	0.4	0.5	0.4	0.3	0.2	0.1
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.8	1.0	0.8	0.5	0.3	0.2	0.3	0.4	0.3	0.2	0.1	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.7	0.8	0.7	0.5	0.3	0.2	0.2	0.3	0.2	0.2	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.5	0.5	0.3	0.0	0.0	0.1	0.2	0.1	0.0	0.0	0.0

0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.2	0.0	0.0	0.0	0.2	0.2	0.4	0.6	0.4	0.2	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.3	0.4	0.3	0.0	0.3	0.5	0.4	0.8	0.9	0.8	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.4	0.4	0.6	0.4	0.0	0.6	0.9	0.6	0.9	1.0	0.9	0.6	0.3
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.5	0.5	0.7	0.5	0.5	0.9	1.0	0.9	0.8	0.9	0.8	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.6	0.7	0.9	0.7	0.6	0.6	0.9	0.6	0.4	0.6	0.4	0.2	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.5	0.7	0.9	1.0	0.9	0.7	0.6	0.5	0.3	0.2	0.3	0.2	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.6	0.7	0.9	0.7	0.6	0.5	0.3	0.3	0.3	0.3	0.3	0.1	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.5	0.7	0.6	0.5	0.2	0.5	0.4	0.4	0.4	0.5	0.2	0.1
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.6	0.4	0.3	0.4	0.5	0.6	0.6	0.6	0.5	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.5	0.6	0.5	0.3	0.5	0.6	0.7	0.7	0.7	0.6	0.5	0.3
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.5	0.6	0.8	0.6	0.4	0.6	0.7	0.8	0.9	0.8	0.7	0.6	0.4
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.6	0.8	1.0	0.8	0.5	0.6	0.7	0.9	1.0	0.9	0.7	0.6	0.4

0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.5	0.6	0.8	0.6	0.4	0.6	0.7	0.8	0.9	0.8	0.7	0.6	0.4
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.5	0.6	0.5	0.4	0.5	0.6	0.7	0.7	0.7	0.6	0.5	0.3
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.5	0.5	0.3	0.4	0.5	0.6	0.6	0.6	0.5	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0	0	0.3	0.5	0.7	0.8	0.7	0.5	0.3	0.3	0.4	0.5	0.4	0.3	0.2	0.1
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.8	1.0	0.8	0.5	0.3	0.2	0.3	0.4	0.3	0.2	0.1	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.7	0.8	0.7	0.5	0.3	0.2	0.2	0.3	0.2	0.2	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.5	0.5	0.3	0.0	0.0	0.1	0.2	0.1	0.0	0.0	0.0

0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.2	0.0	0.0	0.0	0.2	0.2	0.4	0.6	0.4	0.2	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.3	0.4	0.3	0.0	0.3	0.5	0.4	0.8	0.9	0.8	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.4	0.4	0.6	0.4	0.0	0.6	0.9	0.6	0.9	1.0	0.9	0.6	0.3
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.5	0.5	0.7	0.5	0.5	0.9	1.0	0.9	0.8	0.9	0.8	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.6	0.7	0.9	0.7	0.6	0.6	0.9	0.6	0.4	0.6	0.4	0.2	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.5	0.7	0.9	1.0	0.9	0.7	0.6	0.5	0.3	0.2	0.3	0.2	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.6	0.7	0.9	0.7	0.6	0.5	0.3	0.3	0.3	0.3	0.3	0.1	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.5	0.7	0.6	0.5	0.2	0.5	0.4	0.4	0.4	0.5	0.2	0.1
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.6	0.4	0.3	0.4	0.5	0.6	0.6	0.6	0.5	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.5	0.6	0.5	0.3	0.5	0.6	0.7	0.7	0.7	0.6	0.5	0.3
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.5	0.6	0.8	0.6	0.4	0.6	0.7	0.8	0.9	0.8	0.7	0.6	0.4
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.6	0.8	1.0	0.8	0.5	0.6	0.7	0.9	1.0	0.9	0.7	0.6	0.4
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.5	0.6	0.8	0.6	0.4	0.6	0.7	0.8	0.9	0.8	0.7	0.6	0.4
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.5	0.6	0.5	0.4	0.5	0.6	0.7	0.7	0.7	0.6	0.5	0.3
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.5	0.5	0.3	0.4	0.5	0.6	0.6	0.6	0.5	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.7	0.8	0.7	0.5	0.3	0.3	0.4	0.5	0.4	0.3	0.2	0.1
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.8	1.0	0.8	0.5	0.3	0.2	0.3	0.4	0.3	0.2	0.1	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.7	0.8	0.7	0.5	0.3	0.2	0.2	0.3	0.2	0.2	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.5	0.5	0.3	0.0	0.0	0.1	0.2	0.1	0.0	0.0	0.0

0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.2	0.0	0.0	0.0	0.2	0.2	0.4	0.6	0.4	0.2	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.3	0.4	0.3	0.0	0.3	0.5	0.4	0.8	0.9	0.8	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.4	0.4	0.6	0.4	0.0	0.6	0.9	0.6	0.9	1.0	0.9	0.6	0.3
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.5	0.5	0.7	0.5	0.5	0.9	1.0	0.9	0.8	0.9	0.8	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.6	0.7	0.9	0.7	0.6	0.6	0.9	0.6	0.4	0.6	0.4	0.2	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.5	0.7	0.9	1.0	0.9	0.7	0.6	0.5	0.3	0.2	0.3	0.2	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.6	0.7	0.9	0.7	0.6	0.5	0.3	0.3	0.3	0.3	0.3	0.1	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.5	0.7	0.6	0.5	0.2	0.5	0.4	0.4	0.4	0.5	0.2	0.1
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.6	0.4	0.3	0.4	0.5	0.6	0.6	0.6	0.5	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.5	0.6	0.5	0.3	0.5	0.6	0.7	0.7	0.7	0.6	0.5	0.3
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.5	0.6	0.8	0.6	0.4	0.6	0.7	0.8	0.9	0.8	0.7	0.6	0.4
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.6	0.8	1.0	0.8	0.5	0.6	0.7	0.9	1.0	0.9	0.7	0.6	0.4
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.5	0.6	0.8	0.6	0.4	0.6	0.7	0.8	0.9	0.8	0.7	0.6	0.4
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.3	0.4	0.5	0.6	0.5	0.4	0.5	0.6	0.7	0.7	0.7	0.6	0.5	0.3
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.5	0.5	0.3	0.4	0.5	0.6	0.6	0.6	0.5	0.4	0.2
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.7	0.8	0.7	0.5	0.3	0.3	0.4	0.5	0.4	0.3	0.2	0.1
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.8	1.0	0.8	0.5	0.3	0.2	0.3	0.4	0.3	0.2	0.1	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.7	0.8	0.7	0.5	0.3	0.2	0.2	0.3	0.2	0.2	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.5	0.5	0.3	0.0	0.0	0.1	0.2	0.1	0.0	0.0	0.0

B.3 Air Tasking Order Data

B.3.1 AFIT-0A.

```
AFIT-0
17  17  8000
10  11  6000
0
4000 14000 MSL
```

B.3.2 AFIT-1A.

```
AFIT-1
17  17  8000
 1   1  5000
0
4000 14000 MSL
```

B.3.3 AFIT-GOA.

```
AFIT-GO
24  16  8000
 1   1  5000
0
4000 14000 MSL
```

Appendix C. *Raw Data*

C.1 iPSC/2 (Mission AFIT-1A)

C.1.1 Bounded (Depth = 2).

C.1.1.1 2 Nodes.

```
        PARALLEL MISSION ROUTING PROBLEM

            A* USING CENTRALIZED LIST
      ************************************

Enter name of file containing the terrain data: terrainA
Enter name of file containing the radar data : radarA
Enter name of file containing the Air Tasking Order (ATO): AFIT-1A


Waiting for results ...

   Received Initial Path with a computed cost of 8429.419922

 ** Queue length = 401 with q[q_front].cost = 6985.279297 **
 ** Queue length = 789 with q[q_front].cost = 7039.680176 **
 ** Queue length = 1127 with q[q_front].cost = 7079.480469 **
 ** Queue length = 1450 with q[q_front].cost = 7104.679199 **
 ** Queue length = 1772 with q[q_front].cost = 7127.026367 **
 ** Queue length = 2068 with q[q_front].cost = 7147.752441 **
 ** Queue length = 2320 with q[q_front].cost = 7166.814941 **
 ** Queue length = 2597 with q[q_front].cost = 7182.460449 **

  A Solution Path Has Been Found

   Node 1 expanded 1431 states

 The Best Route for mission AFIT-1 is:
    x       y       z
    17      17      8
    16      16      8
    15      15      8
    14      15      8
    13      15      8
    12      15      8
    11      15      8
    10      14      8
     9      13      8
     8      12      8
     7      11      8
     6      10      8
     5       9      7
     5       8      7
     5       7      7
     4       6      6
     4       5      6
     3       4      5
     3       3      5
     2       2      5
     1       1      5
```

```
 For a total of 21 entries in the route

  At a distance of 25924.074219 (  4.91 miles)

  With a radar cost of 2497.056396

  And a computed cost of 7182.460449

         ***  Timing Information  ***

Initialize =     0.143 (sec)     Find Initial Route =     7.374 (sec)
Searching  =  1860.298 (sec)     Total Execution    =  1867.815 (sec)

Average worker node efficiency 0.755
The controller efficiency was  0.654
** 1431 nodes sent to processors
** 1431 total nodes expanded
```

C.1.1.2 4 Nodes.

```
         PARALLEL MISSION ROUTING PROBLEM

             A* USING CENTRALIZED LIST
       ************************************

Enter name of file containing the terrain data: terrainA
Enter name of file containing the radar data : radarA
Enter name of file containing the Air Tasking Order (ATO): AFIT-1A


Waiting for results ...

   Received Initial Path with a computed cost of 8429.419922

 ** Queue length = 396 with q[q_front].cost = 7049.107422 **
 ** Queue length = 777 with q[q_front].cost = 7104.868652 **
 ** Queue length = 1129 with q[q_front].cost = 7143.229980 **
 ** Queue length = 1475 with q[q_front].cost = 7165.334473 **
 ** Queue length = 1788 with q[q_front].cost = 7185.517578 **
 ** Queue length = 2087 with q[q_front].cost = 7201.735840 **

  A Solution Path Has Been Found

   Node 2 expanded 308 states
   Node 1 expanded 303 states
   Node 3 expanded 311 states

 The Best Route for mission AFIT-1 is:
    x       y       z
    17      17       8
    16      16       8
    15      15       8
    14      15       8
    13      15       8
    12      15       7
    11      15       7
    10      14       7
     9      13       6
     9      12       6
     8      11       6
     7      10       6
```

```
    6       9       6
    5       8       6
    5       7       6
    4       6       6
    3       5       6
    2       4       5
    1       3       5
    1       2       5
    1       1       5

 For a total of 21 entries in the route

  At a distance of 26020.449219 (  4.93 miles)

  With a radar cost of 2497.056396

  And a computed cost of 7201.735840

         ***  Timing Information  ***

Initialize =     0.209 (sec)     Find Initial Route =     7.717 (sec)
Searching  =   666.375 (sec)     Total Execution    =   674.301 (sec)

Average worker node efficiency 0.449
The controller efficiency was  0.915
** 922 nodes sent to processors
** 922 total nodes expanded
```

C.1.1.3 8 Nodes.

```
          PARALLEL MISSION ROUTING PROBLEM

              A* USING CENTRALIZED LIST
        ************************************

Enter name of file containing the terrain data: terrainA
Enter name of file containing the radar data : radarA
Enter name of file containing the Air Tasking Order (ATO): AFIT-1A


Waiting for results ...

   Received Initial Path with a computed cost of 8429.419922

 ** Queue length = 387 with best.cost = 7165.510742 **
 ** Queue length = 752 with best.cost = 7230.529785 **
 ** Queue length = 1119 with best.cost = 7261.066895 **
 ** Queue length = 1416 with best.cost = 7288.678711 **
 ** Queue length = 1727 with best.cost = 7307.475586 **
 ** Queue length = 2031 with best.cost = 7323.992188 **
 ** Queue length = 2272 with best.cost = 7343.798828 **
 ** Queue length = 2527 with best.cost = 7360.449219 **

  A Solution Path Has Been Found

   Node 1 expanded 215 states
   Node 4 expanded 219 states
   Node 2 expanded 220 states
   Node 7 expanded 220 states
   Node 3 expanded 219 states
```

```
  Node 5 expanded 218 states
  Node 6 expanded 218 states

The Best Route for mission AFIT-1 is:
   x       y       z
   17      17      8
   16      16      7
   15      15      7
   14      15      7
   13      15      7
   12      15      7
   11      15      7
   10      14      7
    9      13      7
    8      12      7
    7      11      7
    7      10      7
    6       9      7
    6       8      7
    5       7      6
    4       6      6
    3       5      6
    3       4      6
    2       3      5
    1       2      5
    1       1      5

For a total of 21 entries in the route

 At a distance of 25924.074219 (  4.91 miles)

 With a radar cost of 2719.542480

 And a computed cost of 7360.449219

        ***  Timing Information  ***

Initialize =     1.179 (sec)     Find Initial Route =     8.758 (sec)
Searching  =  1292.214 (sec)     Total Execution    =  1302.151 (sec)

Average worker node efficiency 0.170
The controller efficiency was  0.982
** 1529 nodes sent to processors
** 1529 total nodes expanded
```

C.1.2 Bounded (Depth = 3).

C.1.2.1 2 Nodes.

```
          PARALLEL MISSION ROUTING PROBLEM

              A* USING CENTRALIZED LIST
       ************************************

Enter name of file containing the terrain data: terrainA
Enter name of file containing the radar data : radarA
Enter name of file containing the Air Tasking Order (ATO): AFIT-1A


Waiting for results ...
```

```
  Received Initial Path with a computed cost of 8429.419922

** Queue length = 401 with q[q_front].cost = 7161.221680 **
** Queue length = 760 with q[q_front].cost = 7223.974609 **
** Queue length = 1101 with q[q_front].cost = 7260.272461 **
** Queue length = 1435 with q[q_front].cost = 7280.907715 **
** Queue length = 1726 with q[q_front].cost = 7305.279297 **
** Queue length = 1990 with q[q_front].cost = 7326.933594 **

 A Solution Path Has Been Found

  Node 1 expanded 1080 states

The Best Route for mission AFIT-1 is:
   x       y       z
   17      17      8
   17      16      8
   16      15      8
   15      15      8
   14      15      8
   13      15      8
   12      15      7
   11      15      7
   10      14      7
    9      13      7
    8      12      7
    7      11      7
    7      10      7
    6       9      7
    6       8      7
    5       7      6
    5       6      6
    4       5      5
    4       4      5
    3       3      5
    2       2      5
    1       1      5

 For a total of 22 entries in the route

  At a distance of 26606.236328 (  5.04 miles)

  With a radar cost of 2507.106934

  And a computed cost of 7326.933594

        ***  Timing Information  ***

Initialize =     0.145 (sec)     Find Initial Route =    16.369 (sec)
Searching  =  6149.665 (sec)     Total Execution    =  6166.179 (sec)

Average worker node efficiency 0.978
The controller efficiency was  0.118
** 1080 nodes sent to processors
** 1080 total nodes expanded
```

C.1.2.2 4 Nodes.

```
PARALLEL MISSION ROUTING PROBLEM
```

```
            A* USING CENTRALIZED LIST
      ************************************

Enter name of file containing the terrain data: terrainA
Enter name of file containing the radar data : radarA
Enter name of file containing the Air Tasking Order (ATO): AFIT-1A


Waiting for results ...

   Received Initial Path with a computed cost of 8429.419922

 ** Queue length = 402 with q[q_front].cost = 7038.552734 **
 ** Queue length = 768 with q[q_front].cost = 7083.830078 **
 ** Queue length = 1105 with q[q_front].cost = 7114.124023 **
 ** Queue length = 1438 with q[q_front].cost = 7142.144531 **
 ** Queue length = 1721 with q[q_front].cost = 7163.830078 **
 ** Queue length = 1989 with q[q_front].cost = 7182.460449 **

  A Solution Path Has Been Found

   Node 1 expanded 356 states
   Node 2 expanded 351 states
   Node 3 expanded 363 states

 The Best Route for mission AFIT-1 is:
    x       y       z
    17      17      8
    16      16      8
    15      15      8
    14      15      8
    13      15      8
    12      15      8
    11      15      8
    10      14      8
     9      13      8
     8      12      8
     7      11      8
     6      10      7
     6       9      7
     6       8      7
     5       7      6
     5       6      6
     4       5      5
     3       4      5
     2       3      5
     1       2      5
     1       1      5

 For a total of 21 entries in the route

  At a distance of 25924.074219 (  4.91 miles)

  With a radar cost of 2497.056396

  And a computed cost of 7182.460449

         ***  Timing Information  ***

Initialize =     1.128 (sec)     Find Initial Route =    18.509 (sec)
Searching  =  2235.942 (sec)     Total Execution    =  2255.579 (sec)

Average worker node efficiency 0.884
The controller efficiency was  0.410
** 1070 nodes sent to processors
```

```
** 1070 total nodes expanded
```

C.1.2.3 8 Nodes.

```
          PARALLEL MISSION ROUTING PROBLEM

              A* USING CENTRALIZED LIST
        ************************************

Enter name of file containing the terrain data: terrainA
Enter name of file containing the radar data : radarA
Enter name of file containing the Air Tasking Order (ATO): AFIT-1A


Waiting for results ...

   Received Initial Path with a computed cost of 8429.419922

 ** Queue length = 377 with best.cost = 7043.543945 **
 ** Queue length = 747 with best.cost = 7097.458008 **
 ** Queue length = 1082 with best.cost = 7125.948242 **
 ** Queue length = 1414 with best.cost = 7149.722656 **
 ** Queue length = 1687 with best.cost = 7176.435547 **

  A Solution Path Has Been Found

   Node 4 expanded 140 states
   Node 1 expanded 134 states
   Node 5 expanded 139 states
   Node 2 expanded 138 states
   Node 6 expanded 140 states
   Node 3 expanded 133 states
   Node 7 expanded 144 states

 The Best Route for mission AFIT-1 is:
    x       y       z
    17      17      8
    16      16      8
    15      15      8
    14      15      8
    13      15      8
    12      15      8
    11      15      8
    10      14      8
     9      13      7
     8      12      7
     7      11      7
     6      10      7
     6       9      7
     6       8      7
     5       7      6
     5       6      6
     4       5      5
     3       4      5
     2       3      5
     1       2      5
     1       1      5

 For a total of 21 entries in the route

  At a distance of 25924.074219 (  4.91 miles)
```

```
With a radar cost of 2497.056396

And a computed cost of 7182.460449

        ***  Timing Information  ***

Initialize =     0.917 (sec)     Find Initial Route =    17.274 (sec)
Searching  =   916.968 (sec)     Total Execution    =   935.159 (sec)

Average worker node efficiency 0.838
The controller efficiency was  0.677
** 968 nodes sent to processors
** 968 total nodes expanded
```

C.1.3 Bounded (Depth = 4).

C.1.3.1 2 Nodes.

```
        PARALLEL MISSION ROUTING PROBLEM

            A* USING CENTRALIZED LIST
      ************************************

Enter name of file containing the terrain data: terrainA
Enter name of file containing the radar data : radarA
Enter name of file containing the Air Tasking Order (ATO): AFIT-1A


Waiting for results ...

   Received Initial Path with a computed cost of 8429.419922

 ** Queue length = 403 with q[q_front].cost = 7229.051270 **
 ** Queue length = 755 with q[q_front].cost = 7282.436523 **
 ** Queue length = 1103 with q[q_front].cost = 7313.320313 **

  A Solution Path Has Been Found

   Node 1 expanded 563 states

 The Best Route for mission AFIT-1 is:
    x       y       z
    17      17      8
    17      16      8
    16      15      8
    15      15      8
    14      15      8
    13      15      8
    12      15      8
    11      15      7
    10      14      6
     9      13      6
     8      12      6
     8      11      6
     7      10      6
     7       9      6
     6       8      6
     6       7      6
     5       6      5
```

```
    5       5       5
    4       4       5
    3       3       5
    2       2       5
    1       1       5

 For a total of 22 entries in the route

  At a distance of 26606.236328 (  5.04 miles)

  With a radar cost of 2507.106934

  And a computed cost of 7326.933594

         ***  Timing Information  ***

Initialize =     0.189 (sec)     Find Initial Route =    60.177 (sec)
Searching  = 17045.360 (sec)     Total Execution    = 17105.727 (sec)

Average worker node efficiency 0.994
The controller efficiency was  0.015
** 563 nodes sent to processors
** 563 total nodes expanded
```

C.1.3.2 4 Nodes.

```
          PARALLEL MISSION ROUTING PROBLEM

              A* USING CENTRALIZED LIST
        ************************************

Enter name of file containing the terrain data: terrainA
Enter name of file containing the radar data : radarA
Enter name of file containing the Air Tasking Order (ATO): AFIT-1A


Waiting for results ...

   Received Initial Path with a computed cost of 8429.419922

 ** Queue length = 394 with q[q_front].cost = 7069.436035 **
 ** Queue length = 756 with q[q_front].cost = 7112.584961 **
 ** Queue length = 1084 with q[q_front].cost = 7146.831055 **
 ** Queue length = 1377 with q[q_front].cost = 7179.375000 **

  A Solution Path Has Been Found

   Node 1 expanded 251 states
   Node 2 expanded 260 states
   Node 3 expanded 265 states

 The Best Route for mission AFIT-1 is:
    x       y       z
   17      17       8
   16      16       8
   15      15       8
   14      15       8
   13      15       8
   12      15       8
   11      15       8
   10      14       8
```

```
    9      13      8
    8      12      8
    7      11      8
    6      10      7
    6       9      7
    5       8      6
    5       7      6
    4       6      5
    4       5      5
    4       4      5
    3       3      5
    2       2      5
    1       1      5

 For a total of 21 entries in the route

  At a distance of 25924.074219 (  4.91 miles)

  With a radar cost of 2497.056396

  And a computed cost of 7182.460449

         ***  Timing Information  ***

Initialize =     0.209 (sec)     Find Initial Route =    61.152 (sec)
Searching  =  8079.764 (sec)     Total Execution    =  8141.125 (sec)

Average worker node efficiency 0.973
The controller efficiency was  0.074
** 776 nodes sent to processors
** 776 total nodes expanded
```

C.1.3.3 8 Nodes.

```
          PARALLEL MISSION ROUTING PROBLEM

              A* USING CENTRALIZED LIST
        ************************************

Enter name of file containing the terrain data: terrainA
Enter name of file containing the radar data : radarA
Enter name of file containing the Air Tasking Order (ATO): AFIT-1A


Waiting for results ...

   Received Initial Path with a computed cost of 8429.419922

 ** Queue length = 370 with q[q_front].cost = 7069.722656 **
 ** Queue length = 732 with q[q_front].cost = 7123.954590 **
 ** Queue length = 1078 with q[q_front].cost = 7156.837402 **
 ** Queue length = 1399 with q[q_front].cost = 7182.460449 **

  A Solution Path Has Been Found

   Node 1 expanded 99 states
   Node 4 expanded 91 states
   Node 2 expanded 93 states
   Node 6 expanded 99 states
   Node 3 expanded 104 states
   Node 5 expanded 100 states
```

```
Node 7 expanded 93 states

The Best Route for mission AFIT-1 is:
   x       y       z
   17      17      8
   16      16      8
   15      15      8
   14      15      8
   13      15      8
   12      15      8
   11      15      8
   10      14      8
    9      13      8
    8      12      8
    7      11      7
    6      10      7
    6       9      7
    5       8      6
    5       7      6
    5       6      6
    4       5      5
    4       4      5
    3       3      5
    2       2      5
    1       1      5

For a total of 21 entries in the route

 At a distance of 25924.074219 (  4.91 miles)

 With a radar cost of 2497.056396

 And a computed cost of 7182.460449

        ***  Timing Information  ***

Initialize =     0.376 (sec)     Find Initial Route =    61.229 (sec)
Searching  =  2884.007 (sec)     Total Execution    =  2945.612 (sec)

Average worker node efficiency 0.974
The controller efficiency was  0.117
** 679 nodes sent to processors
** 679 total nodes expanded
```

C.2 iPSC/2 (Mission AFIT-GOA)

C.2.1 Recursion Only (Depth = 3).

C.2.1.1 2 Nodes.

```
        PARALLEL MISSION ROUTING PROBLEM

            A* USING CENTRALIZED LIST
      ************************************

Enter name of file containing the terrain data: terrainA
Enter name of file containing the radar data : radarA
Enter name of file containing the Air Tasking Order (ATO): AFIT-GOA
```

```
Waiting for results ...
 ** Queue length = 308 with best.cost = 7441.343750 **
 ** Queue length = 562 with best.cost = 7884.716797 **
 ** Queue length = 813 with best.cost = 8179.139160 **
 ** Queue length = 1088 with best.cost = 8414.400391 **
 ** Queue length = 1374 with best.cost = 8586.919922 **
 ** Queue length = 1616 with best.cost = 8767.997070 **
 ** Queue length = 1875 with best.cost = 8899.986328 **
 ** Queue length = 2104 with best.cost = 9047.362305 **
 ** Queue length = 2377 with best.cost = 9152.572266 **
 ** Queue length = 2652 with best.cost = 9253.708984 **
 ** Queue length = 2942 with best.cost = 9332.996094 **
 ** Queue length = 3260 with best.cost = 9388.378906 **
 ** Queue length = 3584 with best.cost = 9421.083984 **
 ** Queue length = 3908 with best.cost = 9449.775391 **
 ** Queue length = 4209 with best.cost = 9470.197266 **
 ** Queue length = 4533 with best.cost = 9490.113281 **
 ** Queue length = 4843 with best.cost = 9508.447266 **
 ** Queue length = 5159 with best.cost = 9522.035156 **
 ** Queue length = 5447 with best.cost = 9534.190430 **
 ** Queue length = 5708 with best.cost = 9550.009766 **
 ** Queue length = 6000 with best.cost = 9560.248047 **
 ** Queue length = 6278 with best.cost = 9571.906250 **
 ** Queue length = 6545 with best.cost = 9581.730469 **
 ** Queue length = 6814 with best.cost = 9591.373047 **
 ** Queue length = 7025 with best.cost = 9603.826172 **
 ** Queue length = 7300 with best.cost = 9610.816406 **
 ** Queue length = 7515 with best.cost = 9619.933594 **
 ** Queue length = 7796 with best.cost = 9627.179688 **
 ** Queue length = 7991 with best.cost = 9635.562500 **
 ** Queue length = 8209 with best.cost = 9643.138672 **
 ** Queue length = 8418 with best.cost = 9652.355469 **
 ** Queue length = 8590 with best.cost = 9660.357422 **
 ** Queue length = 8878 with best.cost = 9664.815430 **

 A Solution Path Has Been Found

  Node 1 expanded 7669 states

The Best Route for mission AFIT-GO is:
```

x	y	z
24	16	8
24	17	8
24	18	8
23	19	8
22	19	8
21	19	8
20	19	8
19	19	8
18	19	8
17	19	8
16	19	8
15	19	8
14	19	8
13	19	8
12	19	8
11	18	8
10	17	8
10	16	8
9	15	8
9	14	8
8	13	8
8	12	8

```
    7      11      8
    6      10      7
    6       9      7
    6       8      7
    5       7      6
    5       6      6
    4       5      5
    4       4      5
    3       3      5
    2       2      5
    1       1      5

 For a total of 33 entries in the route

  At a distance of 37924.074219 (  7.18 miles)

  With a radar cost of 2600.000000

  And a computed cost of 9664.815430

         ***  Timing Information  ***

Initialize =     0.208 (sec)     Find Initial Route =     0.000 (sec)
Searching  = 37621.738 (sec)     Total Execution    = 37621.946 (sec)

Average worker node efficiency 0.878
The controller efficiency was  0.450
** 7669 nodes sent to processors
** 7669 total nodes expanded
```

C.2.1.2 4 Nodes.

```
          PARALLEL MISSION ROUTING PROBLEM

              A* USING CENTRALIZED LIST
        ************************************

Enter name of file containing the terrain data: terrainA
Enter name of file containing the radar data : radarA
Enter name of file containing the Air Tasking Order (ATO): AFIT-GOA


Waiting for results ...
 ** Queue length = 308 with best.cost = 7582.110840 **
 ** Queue length = 567 with best.cost = 8009.636719 **
 ** Queue length = 835 with best.cost = 8309.082031 **
 ** Queue length = 1130 with best.cost = 8521.951172 **
 ** Queue length = 1415 with best.cost = 8690.030273 **
 ** Queue length = 1673 with best.cost = 8848.114258 **
 ** Queue length = 1923 with best.cost = 8998.019531 **
 ** Queue length = 2181 with best.cost = 9114.445313 **
 ** Queue length = 2437 with best.cost = 9225.193359 **
 ** Queue length = 2721 with best.cost = 9318.757813 **
 ** Queue length = 3030 with best.cost = 9388.451172 **
 ** Queue length = 3361 with best.cost = 9431.350586 **
 ** Queue length = 3680 with best.cost = 9463.058594 **
 ** Queue length = 4008 with best.cost = 9488.949219 **
 ** Queue length = 4353 with best.cost = 9505.761719 **
 ** Queue length = 4671 with best.cost = 9523.263672 **
 ** Queue length = 4977 with best.cost = 9539.441406 **
```

```
** Queue length = 5268 with best.cost = 9553.373047 **
** Queue length = 5555 with best.cost = 9566.025391 **
** Queue length = 5808 with best.cost = 9580.183594 **
** Queue length = 6114 with best.cost = 9587.871094 **
** Queue length = 6337 with best.cost = 9601.005859 **
** Queue length = 6622 with best.cost = 9610.727539 **
** Queue length = 6867 with best.cost = 9619.787109 **
** Queue length = 7139 with best.cost = 9629.185547 **
** Queue length = 7307 with best.cost = 9639.515625 **
** Queue length = 7577 with best.cost = 9647.749023 **
** Queue length = 7755 with best.cost = 9656.598633 **
** Queue length = 7994 with best.cost = 9664.815430 **

 A Solution Path Has Been Found

  Node 2 expanded 2176 states
  Node 1 expanded 2174 states
  Node 3 expanded 2227 states

The Best Route for mission AFIT-GO is:
```

x	y	z
24	16	8
24	17	8
24	18	8
23	19	7
22	19	7
21	19	7
20	19	7
19	19	7
18	19	7
17	19	7
16	19	7
15	19	7
14	19	7
13	19	7
12	19	7
11	18	7
10	17	7
10	16	7
9	15	7
9	14	7
8	13	7
7	12	7
7	11	7
6	10	7
5	9	7
4	8	7
4	7	7
3	6	6
2	5	5
2	4	5
2	3	5
1	2	5
1	1	5

```
For a total of 33 entries in the route

 At a distance of 37924.074219 (  7.18 miles)

 With a radar cost of 2600.000000

 And a computed cost of 9664.815430

        ***  Timing Information  ***
```

```
Initialize =     0.401 (sec)     Find Initial Route =     0.000 (sec)
Searching  = 15622.431 (sec)     Total Execution    = 15622.833 (sec)

Average worker node efficiency 0.603
The controller efficiency was  0.857
** 6577 nodes sent to processors
** 6577 total nodes expanded
```

C.2.1.3 8 Nodes.

```
          PARALLEL MISSION ROUTING PROBLEM

              A* USING CENTRALIZED LIST
        ************************************

Enter name of file containing the terrain data: terrainA
Enter name of file containing the radar data : radarA
Enter name of file containing the Air Tasking Order (ATO): AFIT-GOA


Waiting for results ...
 ** Queue length = 296 with best.cost = 7609.605957 **
 ** Queue length = 566 with best.cost = 8041.794922 **
 ** Queue length = 837 with best.cost = 8319.119141 **
 ** Queue length = 1137 with best.cost = 8532.348633 **
 ** Queue length = 1415 with best.cost = 8700.539063 **
 ** Queue length = 1662 with best.cost = 8862.101563 **
 ** Queue length = 1894 with best.cost = 9024.015625 **
 ** Queue length = 2158 with best.cost = 9144.931641 **
 ** Queue length = 2413 with best.cost = 9258.319336 **
 ** Queue length = 2710 with best.cost = 9353.810547 **
 ** Queue length = 3006 with best.cost = 9425.052734 **
 ** Queue length = 3324 with best.cost = 9470.175781 **
 ** Queue length = 3662 with best.cost = 9502.188477 **
 ** Queue length = 4007 with best.cost = 9524.861328 **
 ** Queue length = 4319 with best.cost = 9545.985352 **
 ** Queue length = 4642 with best.cost = 9563.847656 **
 ** Queue length = 4917 with best.cost = 9581.415039 **
 ** Queue length = 5241 with best.cost = 9592.796875 **
 ** Queue length = 5503 with best.cost = 9609.084961 **
 ** Queue length = 5793 with best.cost = 9617.133789 **
 ** Queue length = 6096 with best.cost = 9628.183594 **
 ** Queue length = 6316 with best.cost = 9639.714844 **
 ** Queue length = 6627 with best.cost = 9646.583984 **
 ** Queue length = 6819 with best.cost = 9657.080078 **
 ** Queue length = 7102 with best.cost = 9666.840820 **
 ** Queue length = 7336 with best.cost = 9673.228516 **
 ** Queue length = 7605 with best.cost = 9681.004883 **
 ** Queue length = 7818 with best.cost = 9688.641602 **
 ** Queue length = 8003 with best.cost = 9699.158203 **
 ** Queue length = 8195 with best.cost = 9706.238281 **
 ** Queue length = 8387 with best.cost = 9712.244141 **

  A Solution Path Has Been Found

   Node 4 expanded 1062 states
   Node 2 expanded 1041 states
   Node 1 expanded 1029 states
   Node 6 expanded 1063 states
```

```
Node 3 expanded 1059 states
Node 5 expanded 1033 states
Node 7 expanded 1051 states

The Best Route for mission AFIT-GO is:
   x       y       z
   24      16       8
   24      17       7
   24      18       7
   23      19       6
   22      19       6
   21      19       6
   20      19       6
   19      19       6
   18      19       6
   17      19       6
   16      19       6
   15      19       6
   14      19       6
   13      19       6
   12      19       6
   11      18       6
   11      17       6
   11      16       6
   11      15       6
   10      14       6
    9      13       6
    9      12       6
    9      11       6
    9      10       6
    8       9       6
    7       8       6
    6       7       6
    5       6       6
    4       5       5
    3       4       5
    2       3       5
    1       2       5
    1       1       5

For a total of 33 entries in the route

 At a distance of 38020.453125 (  7.20 miles)

 With a radar cost of 2641.421387

 And a computed cost of 9717.227539

        ***  Timing Information  ***

Initialize =     0.306 (sec)     Find Initial Route =     0.000 (sec)
Searching  = 16084.236 (sec)     Total Execution    = 16084.542 (sec)

Average worker node efficiency 0.281
The controller efficiency was  0.975
** 7338 nodes sent to processors
** 7338 total nodes expanded
```

C.2.2 Bounded (Depth = 2).

C.2.2.1 2 Nodes.

```
PARALLEL MISSION ROUTING PROBLEM

A* USING CENTRALIZED LIST
************************************

Enter name of file containing the terrain data: terrainA
Enter name of file containing the radar data : radarA
Enter name of file containing the Air Tasking Order (ATO): AFIT-GOA


Waiting for results ...

  Received Initial Path with a computed cost of 15976.822266

 ** Queue length = 333 with q[q_front].cost = 7266.160156 **
 ** Queue length = 564 with q[q_front].cost = 7601.870117 **
 ** Queue length = 788 with q[q_front].cost = 7866.041016 **
 ** Queue length = 1058 with q[q_front].cost = 8053.227539 **
 ** Queue length = 1322 with q[q_front].cost = 8230.949219 **
 ** Queue length = 1580 with q[q_front].cost = 8398.573242 **
 ** Queue length = 1833 with q[q_front].cost = 8538.597656 **
 ** Queue length = 2044 with q[q_front].cost = 8684.550781 **
 ** Queue length = 2285 with q[q_front].cost = 8799.604492 **
 ** Queue length = 2526 with q[q_front].cost = 8910.072266 **
 ** Queue length = 2781 with q[q_front].cost = 9008.322266 **
 ** Queue length = 3014 with q[q_front].cost = 9101.886719 **
 ** Queue length = 3289 with q[q_front].cost = 9173.628906 **
 ** Queue length = 3550 with q[q_front].cost = 9245.269531 **
 ** Queue length = 3812 with q[q_front].cost = 9309.577148 **
 ** Queue length = 4122 with q[q_front].cost = 9353.876953 **
 ** Queue length = 4442 with q[q_front].cost = 9388.384766 **
 ** Queue length = 4735 with q[q_front].cost = 9418.152344 **
 ** Queue length = 5047 with q[q_front].cost = 9443.759766 **
 ** Queue length = 5350 with q[q_front].cost = 9463.954102 **
 ** Queue length = 5659 with q[q_front].cost = 9480.816406 **
 ** Queue length = 5958 with q[q_front].cost = 9495.279297 **
 ** Queue length = 6241 with q[q_front].cost = 9509.697266 **
 ** Queue length = 6536 with q[q_front].cost = 9522.298828 **
 ** Queue length = 6814 with q[q_front].cost = 9533.441406 **
 ** Queue length = 7062 with q[q_front].cost = 9547.046875 **
 ** Queue length = 7314 with q[q_front].cost = 9557.781250 **
 ** Queue length = 7562 with q[q_front].cost = 9567.758789 **
 ** Queue length = 7794 with q[q_front].cost = 9578.824219 **
 ** Queue length = 8027 with q[q_front].cost = 9587.871094 **
 ** Queue length = 8280 with q[q_front].cost = 9597.011719 **
 ** Queue length = 8483 with q[q_front].cost = 9606.309570 **
 ** Queue length = 8745 with q[q_front].cost = 9614.044922 **
 ** Queue length = 8959 with q[q_front].cost = 9621.005859 **
 ** Queue length = 9190 with q[q_front].cost = 9629.029297 **
 ** Queue length = 9369 with q[q_front].cost = 9636.249023 **
  <<<< Performing Beam Search Reduction >>>>
 ** Queue length = 5796 with q[q_front].cost = 9643.409180 **
 ** Queue length = 6065 with q[q_front].cost = 9649.345703 **
 ** Queue length = 6321 with q[q_front].cost = 9655.517578 **
 ** Queue length = 6538 with q[q_front].cost = 9662.332031 **

 A Solution Path Has Been Found

  Node 1 expanded 9837 states

 The Best Route for mission AFIT-GO is:
   x       y       z
```

```
   24      16      8
   24      17      8
   24      18      8
   23      19      8
   22      19      8
   21      19      8
   20      19      8
   19      19      8
   18      19      8
   17      19      8
   16      19      8
   15      19      8
   14      19      8
   13      19      8
   12      19      8
   11      18      8
   10      17      8
    9      16      8
    9      15      8
    8      14      8
    8      13      8
    7      12      8
    7      11      8
    6      10      8
    5       9      7
    5       8      7
    5       7      7
    4       6      6
    4       5      6
    3       4      5
    3       3      5
    2       2      5
    1       1      5

 For a total of 33 entries in the route

  At a distance of 37924.074219 (  7.18 miles)

  With a radar cost of 2600.000000

  And a computed cost of 9664.815430

         ***  Timing Information  ***

Initialize =      0.207 (sec)     Find Initial Route =      8.101 (sec)
Searching  = 26158.732 (sec)     Total Execution    = 26167.040 (sec)

Average worker node efficiency 0.304
The controller efficiency was  0.926
** 9837 nodes sent to processors
** 9837 total nodes expanded
```

C.2.3 Bounded (Depth = 3).

C.2.3.1 Sequential.

```
PARALLEL MISSION ROUTING PROBLEM

A* USING CENTRALIZED LIST
*************************************
```

```
Enter name of file containing the terrain data: terrainA
Enter name of file containing the radar data : radarA
Enter name of file containing the Air Tasking Order (ATO): AFIT-GOA
 Found an initial route with a cost of 15976.822266
 ** Queue length = 331 with q[q_front].cost = 7406.125000 **
 ** Queue length = 600 with q[q_front].cost = 7844.255859 **
 ** Queue length = 851 with q[q_front].cost = 8166.993164 **
 ** Queue length = 1140 with q[q_front].cost = 8406.011719 **
 ** Queue length = 1432 with q[q_front].cost = 8586.857422 **
 ** Queue length = 1689 with q[q_front].cost = 8770.251953 **
 ** Queue length = 1970 with q[q_front].cost = 8904.486328 **
 ** Queue length = 2203 with q[q_front].cost = 9049.735352 **
 ** Queue length = 2470 with q[q_front].cost = 9162.396484 **
 ** Queue length = 2741 with q[q_front].cost = 9267.192383 **
 ** Queue length = 3038 with q[q_front].cost = 9347.765625 **
 ** Queue length = 3366 with q[q_front].cost = 9395.030273 **
 ** Queue length = 3683 with q[q_front].cost = 9430.966797 **
 ** Queue length = 4000 with q[q_front].cost = 9457.712891 **
 ** Queue length = 4310 with q[q_front].cost = 9477.431641 **
 ** Queue length = 4642 with q[q_front].cost = 9495.966797 **
 ** Queue length = 4932 with q[q_front].cost = 9513.947266 **
 ** Queue length = 5238 with q[q_front].cost = 9529.002930 **
 ** Queue length = 5508 with q[q_front].cost = 9544.652344 **
 ** Queue length = 5801 with q[q_front].cost = 9556.470703 **
 ** Queue length = 6093 with q[q_front].cost = 9566.933594 **
 ** Queue length = 6334 with q[q_front].cost = 9579.609375 **
 ** Queue length = 6625 with q[q_front].cost = 9587.034180 **
 ** Queue length = 6851 with q[q_front].cost = 9598.890625 **
 ** Queue length = 7108 with q[q_front].cost = 9608.302734 **
 ** Queue length = 7386 with q[q_front].cost = 9616.025391 **
 ** Queue length = 7623 with q[q_front].cost = 9625.076172 **
 ** Queue length = 7853 with q[q_front].cost = 9634.159180 **
 ** Queue length = 8038 with q[q_front].cost = 9641.689453 **
 ** Queue length = 8257 with q[q_front].cost = 9650.599609 **
 ** Queue length = 8418 with q[q_front].cost = 9660.357422 **
 ** Queue length = 8710 with q[q_front].cost = 9664.815430 **

  A Solution Path Has Been Found


 The Best Route for mission AFIT-GO is:
    x       y       z
   24      16       8
   24      17       8
   24      18       8
   23      19       8
   22      19       8
   21      19       8
   20      19       8
   19      19       8
   18      19       8
   17      19       8
   16      19       8
   15      19       8
   14      19       8
   13      19       8
   12      19       8
   11      18       8
   10      17       8
   10      16       8
    9      15       8
    9      14       8
    8      13       8
    8      12       8
```

```
    7      11      8
    6      10      7
    6       9      7
    6       8      7
    5       7      6
    5       6      6
    4       5      5
    4       4      5
    3       3      5
    2       2      5
    1       1      5

 For a total of 33 entries in the route

  At a distance of 37924.074219 (  7.18 miles)

  With a radar cost of 2600.000000

  And a computed cost of 9664.815430

            ***  Timing Information  ***

Initialize =     7.636 (sec)     Find Initial Route =    13.431 (sec)
Searching  = 49207.111 (sec)     Total Execution    = 49228.178 (sec)

** 7337 Nodes Expanded
```

C.2.3.2 2 Nodes.

```
         PARALLEL MISSION ROUTING PROBLEM

             A* USING CENTRALIZED LIST
       ************************************

Enter name of file containing the terrain data: terrainA
Enter name of file containing the radar data : radarA
Enter name of file containing the Air Tasking Order (ATO): AFIT-GOA


Waiting for results ...

   Received Initial Path with a computed cost of 15976.822266

 ** Queue length = 301 with q[q_front].cost = 7467.415039 **
 ** Queue length = 558 with q[q_front].cost = 7898.970215 **
 ** Queue length = 803 with q[q_front].cost = 8230.949219 **
 ** Queue length = 1093 with q[q_front].cost = 8457.738281 **
 ** Queue length = 1362 with q[q_front].cost = 8638.250000 **
 ** Queue length = 1613 with q[q_front].cost = 8820.671875 **
 ** Queue length = 1865 with q[q_front].cost = 8961.335938 **
 ** Queue length = 2106 with q[q_front].cost = 9095.164063 **
 ** Queue length = 2371 with q[q_front].cost = 9207.203125 **
 ** Queue length = 2653 with q[q_front].cost = 9302.675781 **
 ** Queue length = 2963 with q[q_front].cost = 9370.462891 **
 ** Queue length = 3288 with q[q_front].cost = 9411.371094 **
 ** Queue length = 3615 with q[q_front].cost = 9440.624023 **
 ** Queue length = 3921 with q[q_front].cost = 9467.037109 **
 ** Queue length = 4230 with q[q_front].cost = 9489.152344 **
 ** Queue length = 4560 with q[q_front].cost = 9504.233398 **
 ** Queue length = 4856 with q[q_front].cost = 9520.907227 **
 ** Queue length = 5150 with q[q_front].cost = 9533.441406 **
```

```
** Queue length = 5412 with q[q_front].cost = 9549.185547 **
** Queue length = 5702 with q[q_front].cost = 9559.512695 **
** Queue length = 5984 with q[q_front].cost = 9571.066406 **
** Queue length = 6245 with q[q_front].cost = 9582.477539 **
** Queue length = 6499 with q[q_front].cost = 9592.437500 **
** Queue length = 6709 with q[q_front].cost = 9606.105469 **
** Queue length = 6984 with q[q_front].cost = 9613.125000 **
** Queue length = 7222 with q[q_front].cost = 9621.535156 **
** Queue length = 7482 with q[q_front].cost = 9629.344727 **
** Queue length = 7662 with q[q_front].cost = 9638.539063 **
** Queue length = 7888 with q[q_front].cost = 9648.041992 **
** Queue length = 8079 with q[q_front].cost = 9656.137695 **
** Queue length = 8264 with q[q_front].cost = 9664.815430 **

 A Solution Path Has Been Found

  Node 1 expanded 7336 states

The Best Route for mission AFIT-GO is:
```

x	y	z
24	16	8
24	17	8
24	18	8
23	19	8
22	19	8
21	19	8
20	19	8
19	19	8
18	19	8
17	19	8
16	19	8
15	19	8
14	19	8
13	19	8
12	19	8
11	18	8
10	17	8
10	16	8
9	15	8
9	14	8
8	13	8
8	12	8
7	11	8
6	10	7
6	9	7
6	8	7
5	7	6
5	6	6
4	5	5
4	4	5
3	3	5
2	2	5
1	1	5

```
For a total of 33 entries in the route

 At a distance of 37924.074219 (  7.18 miles)

 With a radar cost of 2600.000000

 And a computed cost of 9664.815430

        ***  Timing Information  ***

Initialize =     0.142 (sec)     Find Initial Route =    18.806 (sec)
```

```
Searching  = 37282.756 (sec)     Total Execution    = 37301.704 (sec)

Average worker node efficiency 0.896
The controller efficiency was  0.408
** 7336 nodes sent to processors
** 7336 total nodes expanded
```

C.2.3.3 4 Nodes.

```
          PARALLEL MISSION ROUTING PROBLEM

              A* USING CENTRALIZED LIST
        ************************************

Enter name of file containing the terrain data: terrainA
Enter name of file containing the radar data : radarA
Enter name of file containing the Air Tasking Order (ATO): AFIT-GOA


Waiting for results ...

   Received Initial Path with a computed cost of 15976.822266

 ** Queue length = 310 with q[q_front].cost = 7661.595703 **
 ** Queue length = 570 with q[q_front].cost = 8116.042969 **
 ** Queue length = 850 with q[q_front].cost = 8418.626953 **
 ** Queue length = 1138 with q[q_front].cost = 8620.270508 **
 ** Queue length = 1397 with q[q_front].cost = 8820.671875 **
 ** Queue length = 1648 with q[q_front].cost = 8988.954102 **
 ** Queue length = 1903 with q[q_front].cost = 9130.094727 **
 ** Queue length = 2164 with q[q_front].cost = 9250.732422 **
 ** Queue length = 2441 with q[q_front].cost = 9359.214844 **
 ** Queue length = 2751 with q[q_front].cost = 9432.746094 **
 ** Queue length = 3083 with q[q_front].cost = 9474.386719 **
 ** Queue length = 3432 with q[q_front].cost = 9503.013672 **
 ** Queue length = 3755 with q[q_front].cost = 9529.002930 **
 ** Queue length = 4062 with q[q_front].cost = 9551.070313 **
 ** Queue length = 4386 with q[q_front].cost = 9567.283203 **
 ** Queue length = 4683 with q[q_front].cost = 9584.219727 **
 ** Queue length = 4992 with q[q_front].cost = 9595.220703 **
 ** Queue length = 5281 with q[q_front].cost = 9610.994141 **
 ** Queue length = 5572 with q[q_front].cost = 9619.841797 **
 ** Queue length = 5863 with q[q_front].cost = 9630.851563 **
 ** Queue length = 6104 with q[q_front].cost = 9643.592773 **
 ** Queue length = 6389 with q[q_front].cost = 9650.250000 **
 ** Queue length = 6640 with q[q_front].cost = 9661.626953 **
 ** Queue length = 6867 with q[q_front].cost = 9672.057617 **
 ** Queue length = 7132 with q[q_front].cost = 9678.925781 **
 ** Queue length = 7363 with q[q_front].cost = 9687.771484 **
 ** Queue length = 7550 with q[q_front].cost = 9698.201172 **
 ** Queue length = 7760 with q[q_front].cost = 9706.238281 **
 ** Queue length = 7931 with q[q_front].cost = 9713.343750 **

  A Solution Path Has Been Found

   Node 1 expanded 2254 states
   Node 2 expanded 2260 states
   Node 3 expanded 2242 states

 The Best Route for mission AFIT-GO is:
    x       y       z
```

```
   24      16      8
   24      17      8
   24      18      8
   23      19      7
   22      19      6
   21      19      6
   20      19      6
   19      19      6
   18      19      6
   17      19      6
   16      19      6
   15      19      6
   14      19      6
   13      19      6
   12      19      6
   11      18      6
   11      17      6
   10      16      6
    9      15      6
    9      14      6
    8      13      6
    7      12      6
    6      11      6
    5      10      6
    5       9      6
    4       8      6
    3       7      6
    3       6      6
    2       5      5
    2       4      5
    2       3      5
    1       2      5
    1       1      5

 For a total of 33 entries in the route

  At a distance of 38020.445313 (  7.20 miles)

  With a radar cost of 2641.421387

  And a computed cost of 9717.227539

         ***  Timing Information  ***

Initialize =      0.250 (sec)     Find Initial Route =     19.177 (sec)
Searching  = 16306.274 (sec)     Total Execution    = 16325.701 (sec)

Average worker node efficiency 0.632
The controller efficiency was  0.835
** 6756 nodes sent to processors
** 6756 total nodes expanded
```

C.2.3.4 8 Nodes.

```
          PARALLEL MISSION ROUTING PROBLEM

              A* USING CENTRALIZED LIST
        ************************************

Enter name of file containing the terrain data: terrainA
Enter name of file containing the radar data : radarA
```

```
Enter name of file containing the Air Tasking Order (ATO): AFIT-GOA


Waiting for results ...

   Received Initial Path with a computed cost of 15976.822266

 ** Queue length = 283 with q[q_front].cost = 7536.276855 **
 ** Queue length = 543 with q[q_front].cost = 7966.524414 **
 ** Queue length = 800 with q[q_front].cost = 8274.081055 **
 ** Queue length = 1085 with q[q_front].cost = 8487.898438 **
 ** Queue length = 1352 with q[q_front].cost = 8673.411133 **
 ** Queue length = 1620 with q[q_front].cost = 8840.985352 **
 ** Queue length = 1868 with q[q_front].cost = 8989.350586 **
 ** Queue length = 2109 with q[q_front].cost = 9119.822266 **
 ** Queue length = 2376 with q[q_front].cost = 9230.203125 **
 ** Queue length = 2671 with q[q_front].cost = 9313.728516 **
 ** Queue length = 2983 with q[q_front].cost = 9376.520508 **
 ** Queue length = 3303 with q[q_front].cost = 9417.193359 **
 ** Queue length = 3627 with q[q_front].cost = 9447.920898 **
 ** Queue length = 3933 with q[q_front].cost = 9469.876953 **
 ** Queue length = 4255 with q[q_front].cost = 9490.113281 **
 ** Queue length = 4560 with q[q_front].cost = 9509.093750 **
 ** Queue length = 4878 with q[q_front].cost = 9522.298828 **
 ** Queue length = 5161 with q[q_front].cost = 9536.858398 **
 ** Queue length = 5417 with q[q_front].cost = 9551.557617 **
 ** Queue length = 5707 with q[q_front].cost = 9561.313477 **
 ** Queue length = 5983 with q[q_front].cost = 9574.377930 **
 ** Queue length = 6252 with q[q_front].cost = 9583.893555 **
 ** Queue length = 6514 with q[q_front].cost = 9593.556641 **
 ** Queue length = 6714 with q[q_front].cost = 9606.309570 **
 ** Queue length = 6990 with q[q_front].cost = 9614.427734 **
 ** Queue length = 7217 with q[q_front].cost = 9623.158203 **
 ** Queue length = 7479 with q[q_front].cost = 9630.943359 **
 ** Queue length = 7657 with q[q_front].cost = 9639.456055 **
 ** Queue length = 7884 with q[q_front].cost = 9648.460938 **
 ** Queue length = 8061 with q[q_front].cost = 9657.726563 **
 ** Queue length = 8280 with q[q_front].cost = 9664.815430 **

  A Solution Path Has Been Found

   Node 1 expanded 1036 states
   Node 4 expanded 1046 states
   Node 2 expanded 1045 states
   Node 5 expanded 1028 states
   Node 3 expanded 1057 states
   Node 6 expanded 1051 states
   Node 7 expanded 1043 states

 The Best Route for mission AFIT-GO is:
    x       y       z
    24      16       8
    24      17       8
    24      18       8
    23      19       8
    22      19       8
    21      19       8
    20      19       8
    19      19       8
    18      19       8
    17      19       8
    16      19       8
    15      19       8
    14      19       8
    13      19       8
```

```
   12      19      8
   11      18      8
   11      17      8
   11      16      8
   10      15      8
    9      14      8
    9      13      8
    8      12      8
    7      11      8
    6      10      8
    6       9      8
    6       8      8
    5       7      7
    4       6      6
    3       5      5
    3       4      5
    2       3      5
    1       2      5
    1       1      5

 For a total of 33 entries in the route

  At a distance of 37924.074219 (  7.18 miles)

  With a radar cost of 2600.000000

  And a computed cost of 9664.815430

         ***  Timing Information  ***

Initialize =     0.290 (sec)     Find Initial Route =    19.457 (sec)
Searching  = 15677.566 (sec)     Total Execution    = 15697.313 (sec)

Average worker node efficiency 0.304
The controller efficiency was  0.969
** 7306 nodes sent to processors
** 7306 total nodes expanded
```

C.2.4 Bounded (Depth = 4).

C.2.4.1 2 Nodes.

```
          PARALLEL MISSION ROUTING PROBLEM

              A* USING CENTRALIZED LIST
       ************************************

Enter name of file containing the terrain data: terrainA
Enter name of file containing the radar data : radarA
Enter name of file containing the Air Tasking Order (ATO): AFIT-GOA


Waiting for results ...

   Received Initial Path with a computed cost of 15976.822266

 ** Queue length = 292 with q[q_front].cost = 7782.239258 **
 ** Queue length = 564 with q[q_front].cost = 8350.895508 **
 ** Queue length = 837 with q[q_front].cost = 8669.387695 **
 ** Queue length = 1131 with q[q_front].cost = 8885.212891 **
```

```
** Queue length = 1401 with q[q_front].cost = 9085.833008 **
** Queue length = 1671 with q[q_front].cost = 9248.910156 **
** Queue length = 1960 with q[q_front].cost = 9351.014648 **
** Queue length = 2297 with q[q_front].cost = 9411.905273 **
** Queue length = 2629 with q[q_front].cost = 9450.676758 **
** Queue length = 2943 with q[q_front].cost = 9480.691406 **
** Queue length = 3275 with q[q_front].cost = 9503.215820 **
** Queue length = 3587 with q[q_front].cost = 9522.298828 **
** Queue length = 3873 with q[q_front].cost = 9544.767578 **
** Queue length = 4192 with q[q_front].cost = 9556.672852 **
** Queue length = 4488 with q[q_front].cost = 9569.970703 **
** Queue length = 4764 with q[q_front].cost = 9582.477539 **
** Queue length = 5045 with q[q_front].cost = 9595.438477 **
** Queue length = 5270 with q[q_front].cost = 9608.302734 **
** Queue length = 5546 with q[q_front].cost = 9616.287109 **
** Queue length = 5803 with q[q_front].cost = 9627.950195 **
** Queue length = 6030 with q[q_front].cost = 9638.064453 **
** Queue length = 6279 with q[q_front].cost = 9648.165039 **
** Queue length = 6472 with q[q_front].cost = 9658.283203 **
** Queue length = 6749 with q[q_front].cost = 9664.815430 **

 A Solution Path Has Been Found

  Node 1 expanded 5354 states

The Best Route for mission AFIT-GO is:
```

x	y	z
24	16	8
24	17	8
24	18	8
23	19	8
22	19	8
21	19	8
20	19	8
19	19	8
18	19	8
17	19	8
16	19	8
15	19	8
14	19	8
13	19	8
12	19	8
11	18	8
11	17	8
10	16	8
10	15	8
9	14	8
9	13	8
8	12	8
7	11	7
7	10	7
7	9	7
6	8	6
6	7	6
5	6	5
5	5	5
4	4	5
3	3	5
2	2	5
1	1	5

```
For a total of 33 entries in the route

 At a distance of 37924.078125 (  7.18 miles)
```

```
With a radar cost of 2600.000000

And a computed cost of 9664.815430

        ***  Timing Information  ***

Initialize =      0.180 (sec)     Find Initial Route =     73.364 (sec)
Searching  = 137152.444 (sec)     Total Execution    = 137225.988 (sec)

Average worker node efficiency 0.988
The controller efficiency was  0.068
** 5354 nodes sent to processors
** 5354 total nodes expanded
```

C.2.4.2 4 Nodes.

```
        PARALLEL MISSION ROUTING PROBLEM

            A* USING CENTRALIZED LIST
      ************************************

Enter name of file containing the terrain data: terrainA
Enter name of file containing the radar data : radarA
Enter name of file containing the Air Tasking Order (ATO): AFIT-GOA


Waiting for results ...

   Received Initial Path with a computed cost of 15976.822266

 ** Queue length = 310 with q[q_front].cost = 7942.277344 **
 ** Queue length = 587 with q[q_front].cost = 8465.471680 **
 ** Queue length = 870 with q[q_front].cost = 8747.261719 **
 ** Queue length = 1149 with q[q_front].cost = 8975.073242 **
 ** Queue length = 1411 with q[q_front].cost = 9171.890625 **
 ** Queue length = 1702 with q[q_front].cost = 9304.623047 **
 ** Queue length = 2010 with q[q_front].cost = 9388.378906 **
 ** Queue length = 2351 with q[q_front].cost = 9433.945313 **
 ** Queue length = 2676 with q[q_front].cost = 9469.876953 **
 ** Queue length = 3020 with q[q_front].cost = 9490.691406 **
 ** Queue length = 3332 with q[q_front].cost = 9519.025391 **
 ** Queue length = 3646 with q[q_front].cost = 9532.197266 **
 ** Queue length = 3940 with q[q_front].cost = 9551.791016 **
 ** Queue length = 4241 with q[q_front].cost = 9566.304688 **
 ** Queue length = 4496 with q[q_front].cost = 9580.396484 **
 ** Queue length = 4812 with q[q_front].cost = 9589.105469 **
 ** Queue length = 5020 with q[q_front].cost = 9606.105469 **
 ** Queue length = 5310 with q[q_front].cost = 9614.427734 **
 ** Queue length = 5555 with q[q_front].cost = 9624.500000 **
 ** Queue length = 5800 with q[q_front].cost = 9635.359375 **
 ** Queue length = 6016 with q[q_front].cost = 9646.025391 **
 ** Queue length = 6249 with q[q_front].cost = 9654.634766 **
 ** Queue length = 6444 with q[q_front].cost = 9664.815430 **
 ** Queue length = 6798 with q[q_front].cost = 9664.815430 **

  A Solution Path Has Been Found

   Node 2 expanded 1721 states
   Node 1 expanded 1745 states
   Node 3 expanded 1744 states
```

```
The Best Route for mission AFIT-GO is:
   x       y       z
   24      16       8
   24      17       8
   24      18       8
   23      19       8
   22      19       8
   21      19       8
   20      19       8
   19      19       8
   18      19       8
   17      19       8
   16      19       8
   15      19       8
   14      19       8
   13      19       8
   12      19       8
   11      18       8
   10      17       8
    9      16       8
    8      15       8
    7      14       8
    6      13       8
    5      12       8
    4      11       7
    3      10       7
    3       9       7
    2       8       6
    2       7       6
    1       6       5
    1       5       5
    1       4       5
    1       3       5
    1       2       5
    1       1       5

 For a total of 33 entries in the route

  At a distance of 37924.070313 (  7.18 miles)

  With a radar cost of 2600.000000

  And a computed cost of 9664.815430

         ***  Timing Information  ***

Initialize =     0.148 (sec)     Find Initial Route =    73.664 (sec)
Searching  = 45295.759 (sec)     Total Execution    = 45369.571 (sec)

Average worker node efficiency 0.981
The controller efficiency was  0.198
** 5210 nodes sent to processors
** 5210 total nodes expanded
```

C.2.4.3 8 Nodes.

```
PARALLEL MISSION ROUTING PROBLEM

A* USING CENTRALIZED LIST
************************************
```

```
Enter name of file containing the terrain data: terrainA
Enter name of file containing the radar data : radarA
Enter name of file containing the Air Tasking Order (ATO): AFIT-GOA


Waiting for results ...

   Received Initial Path with a computed cost of 15976.822266

 ** Queue length = 287 with q[q_front].cost = 7942.958496 **
 ** Queue length = 571 with q[q_front].cost = 8438.163086 **
 ** Queue length = 850 with q[q_front].cost = 8716.149414 **
 ** Queue length = 1143 with q[q_front].cost = 8932.820313 **
 ** Queue length = 1398 with q[q_front].cost = 9127.625977 **
 ** Queue length = 1670 with q[q_front].cost = 9283.896484 **
 ** Queue length = 1961 with q[q_front].cost = 9384.240234 **
 ** Queue length = 2301 with q[q_front].cost = 9451.109375 **
 ** Queue length = 2631 with q[q_front].cost = 9489.152344 **
 ** Queue length = 2961 with q[q_front].cost = 9516.497070 **
 ** Queue length = 3265 with q[q_front].cost = 9541.361328 **
 ** Queue length = 3582 with q[q_front].cost = 9558.315430 **
 ** Queue length = 3871 with q[q_front].cost = 9573.319336 **
 ** Queue length = 4141 with q[q_front].cost = 9587.163086 **
 ** Queue length = 4427 with q[q_front].cost = 9599.671875 **
 ** Queue length = 4708 with q[q_front].cost = 9612.832031 **
 ** Queue length = 4963 with q[q_front].cost = 9623.306641 **
 ** Queue length = 5245 with q[q_front].cost = 9634.142578 **
 ** Queue length = 5487 with q[q_front].cost = 9645.131836 **
 ** Queue length = 5784 with q[q_front].cost = 9652.620117 **
 ** Queue length = 5992 with q[q_front].cost = 9661.923828 **
 ** Queue length = 6192 with q[q_front].cost = 9673.830078 **
 ** Queue length = 6393 with q[q_front].cost = 9682.435547 **

  A Solution Path Has Been Found

   Node 1 expanded 744 states
   Node 4 expanded 751 states
   Node 3 expanded 743 states
   Node 6 expanded 749 states
   Node 2 expanded 744 states
   Node 5 expanded 759 states
   Node 7 expanded 755 states

 The Best Route for mission AFIT-GO is:
    x       y       z
    24      16       8
    24      17       8
    24      18       7
    23      19       6
    22      19       6
    21      19       6
    20      19       6
    19      19       6
    18      19       6
    17      19       6
    16      19       6
    15      19       6
    14      19       6
    13      19       6
    12      19       6
    11      18       6
    10      17       6
     9      16       6
     8      15       6
     8      14       6
```

```
    7      13      6
    6      12      6
    5      11      6
    4      10      6
    4       9      6
    3       8      6
    2       7      6
    1       6      5
    1       5      5
    1       4      5
    1       3      5
    1       2      5
    1       1      5

 For a total of 33 entries in the route

  At a distance of 38020.445313 (  7.20 miles)

  With a radar cost of 2600.000000

  And a computed cost of 9684.090820

         ***  Timing Information  ***

Initialize =     0.168 (sec)     Find Initial Route =    73.936 (sec)
Searching  = 19856.748 (sec)     Total Execution    = 19930.852 (sec)

Average worker node efficiency 0.957
The controller efficiency was  0.442
** 5245 nodes sent to processors
** 5245 total nodes expanded
```

C.3 iPSC/860 (Mission AFIT-1A)

C.3.1 Bounded (Depth = 3).

C.3.1.1 2 Nodes.

```
          PARALLEL MISSION ROUTING PROBLEM

             A* USING CENTRALIZED LIST
        ************************************

Enter name of file containing the terrain data: terrainA
Enter name of file containing the radar data : radarA
Enter name of file containing the Air Tasking Order (ATO): AFIT-1A


Waiting for results ...

   Received Initial Path with a computed cost of 8429.419922

 ** Queue length = 417 with q[q_front].cost = 7140.098145 **
 ** Queue length = 797 with q[q_front].cost = 7214.195313 **
 ** Queue length = 1167 with q[q_front].cost = 7251.303711 **
 ** Queue length = 1506 with q[q_front].cost = 7276.545898 **
 ** Queue length = 1843 with q[q_front].cost = 7293.732422 **
 ** Queue length = 2128 with q[q_front].cost = 7315.572266 **
```

```
 A Solution Path Has Been Found

  Node 1 expanded 1054 states

The Best Route for mission AFIT-1 is:
   x       y       z
   17      17      8
   17      16      8
   16      15      8
   15      15      8
   14      15      8
   13      15      8
   12      15      7
   11      15      7
   10      14      6
    9      13      6
    8      12      6
    7      11      6
    7      10      6
    6       9      6
    6       8      6
    5       7      6
    5       6      6
    4       5      5
    4       4      5
    3       3      5
    2       2      5
    1       1      5

For a total of 22 entries in the route

 At a distance of 26606.234375 (  5.04 miles)

 With a radar cost of 2507.106689

 And a computed cost of 7326.933105

        ***  Timing Information  ***

Initialize =     0.184 (sec)     Find Initial Route =     5.905 (sec)
Searching  =  2890.961 (sec)     Total Execution    =  2897.050 (sec)

Average worker node efficiency 0.983
The controller efficiency was  0.093
** 1054 nodes sent to processors
** 1054 total nodes expanded
```

C.3.1.2 4 Nodes.

```
          PARALLEL MISSION ROUTING PROBLEM

             A* USING CENTRALIZED LIST
       ************************************

Enter name of file containing the terrain data: terrainA
Enter name of file containing the radar data : radarA
Enter name of file containing the Air Tasking Order (ATO): AFIT-1A


Waiting for results ...
```

```
  Received Initial Path with a computed cost of 8429.419922

** Queue length = 417 with q[q_front].cost = 7012.874023 **
** Queue length = 800 with q[q_front].cost = 7065.594727 **
** Queue length = 1164 with q[q_front].cost = 7098.755859 **
** Queue length = 1489 with q[q_front].cost = 7123.954102 **
** Queue length = 1821 with q[q_front].cost = 7143.987305 **
** Queue length = 2135 with q[q_front].cost = 7162.094238 **
** Queue length = 2393 with q[q_front].cost = 7181.497070 **

 A Solution Path Has Been Found

  Node 1 expanded 410 states
  Node 2 expanded 402 states
  Node 3 expanded 403 states

The Best Route for mission AFIT-1 is:
   x       y       z
   17      17       8
   16      16       8
   15      15       8
   14      15       8
   13      15       8
   12      15       8
   11      15       8
   10      14       8
    9      13       8
    8      12       8
    7      11       8
    7      10       8
    6       9       7
    6       8       7
    5       7       6
    5       6       6
    4       5       5
    4       4       5
    3       3       5
    2       2       5
    1       1       5

 For a total of 21 entries in the route

  At a distance of 25924.072266 (  4.91 miles)

  With a radar cost of 2497.056152

  And a computed cost of 7182.459961

        ***  Timing Information  ***

Initialize =     0.444 (sec)     Find Initial Route =     6.953 (sec)
Searching  =  1155.689 (sec)     Total Execution    =  1163.086 (sec)

Average worker node efficiency 0.965
The controller efficiency was  0.307
** 1215 nodes sent to processors
** 1215 total nodes expanded
```

C.3.1.3 8 Nodes.

```
        PARALLEL MISSION ROUTING PROBLEM

            A* USING CENTRALIZED LIST
      ************************************

Enter name of file containing the terrain data: terrainA
Enter name of file containing the radar data : radarA
Enter name of file containing the Air Tasking Order (ATO): AFIT-1A


Waiting for results ...

   Received Initial Path with a computed cost of 8429.419922

 ** Queue length = 409 with q[q_front].cost = 7020.617188 **
 ** Queue length = 801 with q[q_front].cost = 7069.722168 **
 ** Queue length = 1177 with q[q_front].cost = 7100.122070 **
 ** Queue length = 1517 with q[q_front].cost = 7125.947754 **
 ** Queue length = 1850 with q[q_front].cost = 7146.989258 **
 ** Queue length = 2145 with q[q_front].cost = 7166.814453 **
 ** Queue length = 2462 with q[q_front].cost = 7182.459961 **

  A Solution Path Has Been Found

   Node 1 expanded 152 states
   Node 2 expanded 160 states
   Node 4 expanded 164 states
   Node 3 expanded 155 states
   Node 5 expanded 156 states
   Node 6 expanded 154 states
   Node 7 expanded 156 states

 The Best Route for mission AFIT-1 is:
    x       y       z
    17      17      8
    16      16      8
    15      15      8
    14      15      8
    13      15      8
    12      15      8
    11      15      8
    10      14      8
     9      13      8
     8      12      8
     8      11      8
     7      10      8
     6       9      7
     6       8      7
     5       7      6
     5       6      6
     4       5      5
     3       4      5
     2       3      5
     1       2      5
     1       1      5

 For a total of 21 entries in the route

  At a distance of 25924.072266 (  4.91 miles)

  With a radar cost of 2497.056152

  And a computed cost of 7182.459961

         ***  Timing Information  ***
```

```
Initialize =     0.947 (sec)     Find Initial Route =     8.843 (sec)
Searching  =   476.384 (sec)     Total Execution    =   486.174 (sec)

Average worker node efficiency 0.875
The controller efficiency was  0.639
** 1097 nodes sent to processors
** 1097 total nodes expanded
```

C.3.2 Bounded (Depth = 4).

C.3.2.1 2 Nodes.

```
        PARALLEL MISSION ROUTING PROBLEM

            A* USING CENTRALIZED LIST
      ************************************

Enter name of file containing the terrain data: terrainA
Enter name of file containing the radar data : radarA
Enter name of file containing the Air Tasking Order (ATO): AFIT-1A


Waiting for results ...

   Received Initial Path with a computed cost of 8429.419922

 ** Queue length = 421 with q[q_front].cost = 7234.859375 **
 ** Queue length = 808 with q[q_front].cost = 7289.190918 **
 ** Queue length = 1154 with q[q_front].cost = 7326.650391 **
 ** Queue length = 1478 with q[q_front].cost = 7352.232422 **

  A Solution Path Has Been Found

   Node 1 expanded 676 states

 The Best Route for mission AFIT-1 is:
    x       y       z
   17      17       8
   16      16       7
   15      15       7
   14      15       7
   13      15       7
   12      15       7
   11      15       7
   10      14       6
    9      13       6
    8      12       6
    8      11       6
    7      10       6
    7       9       6
    6       8       6
    6       7       6
    5       6       5
    5       5       5
    4       4       5
    3       3       5
    2       2       5
    1       1       5
```

```
For a total of 21 entries in the route

 At a distance of 25924.072266 (  4.91 miles)

 With a radar cost of 2719.542236

 And a computed cost of 7360.449219

        ***  Timing Information  ***

Initialize =     0.348 (sec)     Find Initial Route =    32.277 (sec)
Searching  = 12544.106 (sec)     Total Execution    = 12576.731 (sec)

Average worker node efficiency 0.996
The controller efficiency was  0.011
** 676 nodes sent to processors
** 676 total nodes expanded
```

C.3.2.2 4 Nodes.

```
          PARALLEL MISSION ROUTING PROBLEM

              A* USING CENTRALIZED LIST
        ************************************

Enter name of file containing the terrain data: terrainA
Enter name of file containing the radar data : radarA
Enter name of file containing the Air Tasking Order (ATO): AFIT-1A


Waiting for results ...

   Received Initial Path with a computed cost of 8429.419922

 ** Queue length = 407 with q[q_front].cost = 7186.049805 **
 ** Queue length = 790 with q[q_front].cost = 7226.736328 **
 ** Queue length = 1137 with q[q_front].cost = 7258.778320 **
 ** Queue length = 1463 with q[q_front].cost = 7290.067871 **
 ** Queue length = 1798 with q[q_front].cost = 7307.657715 **

  A Solution Path Has Been Found

   Node 1 expanded 263 states
   Node 2 expanded 257 states
   Node 3 expanded 261 states

 The Best Route for mission AFIT-1 is:
    x       y       z
    17      17      8
    17      16      8
    16      15      8
    15      15      8
    14      15      8
    13      15      8
    12      15      8
    11      15      8
    10      14      8
     9      13      8
     8      12      8
     7      11      7
     7      10      7
```

```
    7       9       7
    6       8       6
    6       7       6
    5       6       5
    5       5       5
    4       4       5
    3       3       5
    2       2       5
    1       1       5

 For a total of 22 entries in the route

  At a distance of 26509.859375 (  5.02 miles)

  With a radar cost of 2507.106689

  And a computed cost of 7307.657715

        ***  Timing Information  ***

Initialize =     0.579 (sec)     Find Initial Route =    32.381 (sec)
Searching  =  5015.077 (sec)     Total Execution    =  5048.037 (sec)

Average worker node efficiency 0.992
The controller efficiency was  0.036
** 781 nodes sent to processors
** 781 total nodes expanded
```

C.3.2.3 8 Nodes.

```
          PARALLEL MISSION ROUTING PROBLEM

             A* USING CENTRALIZED LIST
       ************************************

Enter name of file containing the terrain data: terrainA
Enter name of file containing the radar data : radarA
Enter name of file containing the Air Tasking Order (ATO): AFIT-1A


Waiting for results ...

   Received Initial Path with a computed cost of 8429.419922

 ** Queue length = 407 with q[q_front].cost = 7039.892578 **
 ** Queue length = 790 with q[q_front].cost = 7098.755859 **
 ** Queue length = 1144 with q[q_front].cost = 7125.947754 **
 ** Queue length = 1478 with q[q_front].cost = 7153.207031 **
 ** Queue length = 1786 with q[q_front].cost = 7181.497070 **
 ** Queue length = 2157 with q[q_front].cost = 7182.459961 **

  A Solution Path Has Been Found

   Node 1 expanded 118 states
   Node 2 expanded 122 states
   Node 4 expanded 129 states
   Node 3 expanded 126 states
   Node 5 expanded 117 states
   Node 6 expanded 125 states
   Node 7 expanded 129 states
```

```
The Best Route for mission AFIT-1 is:
   x       y       z
   17      17      8
   16      16      8
   15      15      8
   14      15      8
   13      15      8
   12      15      8
   11      15      8
   10      14      8
    9      13      8
    8      12      8
    7      11      8
    6      10      7
    6       9      7
    6       8      7
    5       7      6
    4       6      5
    3       5      5
    2       4      5
    1       3      5
    1       2      5
    1       1      5

 For a total of 21 entries in the route

  At a distance of 25924.072266 (  4.91 miles)

  With a radar cost of 2497.056152

  And a computed cost of 7182.459961

         ***  Timing Information  ***

Initialize =     0.954 (sec)     Find Initial Route =    33.699 (sec)
Searching  =  2323.707 (sec)     Total Execution    =  2358.360 (sec)

Average worker node efficiency 0.983
The controller efficiency was  0.088
** 866 nodes sent to processors
** 866 total nodes expanded
```

C.4 iPSC/860 (Mission AFIT-GOA).

C.4.1 Bounded With Angle = 60.0 (Depth = 3).

C.4.1.1 2 Nodes.

```
          PARALLEL MISSION ROUTING PROBLEM

              A* USING CENTRALIZED LIST
        ************************************

Enter name of file containing the terrain data: terrainA
Enter name of file containing the radar data : radarA
Enter name of file containing the Air Tasking Order (ATO): AFIT-GOA
```

```
Waiting for results ...

  Received Initial Path with a computed cost of 15976.821289

** Queue length = 345 with q[q_front].cost = 7321.625977 **
** Queue length = 614 with q[q_front].cost = 7572.889648 **
** Queue length = 882 with q[q_front].cost = 7772.815430 **
** Queue length = 1144 with q[q_front].cost = 7940.270508 **
** Queue length = 1384 with q[q_front].cost = 8077.243164 **
** Queue length = 1666 with q[q_front].cost = 8178.366211 **
** Queue length = 1921 with q[q_front].cost = 8275.594727 **
** Queue length = 2192 with q[q_front].cost = 8354.332031 **
** Queue length = 2459 with q[q_front].cost = 8429.144531 **
** Queue length = 2700 with q[q_front].cost = 8508.234375 **
** Queue length = 2981 with q[q_front].cost = 8565.270508 **
** Queue length = 3230 with q[q_front].cost = 8627.852539 **
** Queue length = 3496 with q[q_front].cost = 8683.722656 **
** Queue length = 3708 with q[q_front].cost = 8743.944336 **
** Queue length = 3936 with q[q_front].cost = 8796.912109 **
** Queue length = 4175 with q[q_front].cost = 8842.523438 **
** Queue length = 4392 with q[q_front].cost = 8890.786133 **
** Queue length = 4588 with q[q_front].cost = 8934.691406 **
** Queue length = 4795 with q[q_front].cost = 8977.369141 **
** Queue length = 5016 with q[q_front].cost = 9017.466797 **
** Queue length = 5209 with q[q_front].cost = 9059.417969 **
** Queue length = 5446 with q[q_front].cost = 9094.762695 **
** Queue length = 5651 with q[q_front].cost = 9132.886719 **
** Queue length = 5821 with q[q_front].cost = 9176.864258 **
** Queue length = 6024 with q[q_front].cost = 9209.489258 **
** Queue length = 6207 with q[q_front].cost = 9245.529297 **
** Queue length = 6370 with q[q_front].cost = 9279.855469 **
** Queue length = 6582 with q[q_front].cost = 9307.747070 **
** Queue length = 6789 with q[q_front].cost = 9336.828125 **
** Queue length = 7028 with q[q_front].cost = 9360.974609 **
** Queue length = 7256 with q[q_front].cost = 9385.375000 **
** Queue length = 7512 with q[q_front].cost = 9405.291016 **
** Queue length = 7787 with q[q_front].cost = 9420.639648 **
** Queue length = 8019 with q[q_front].cost = 9439.339844 **
** Queue length = 8298 with q[q_front].cost = 9453.063477 **
** Queue length = 8567 with q[q_front].cost = 9468.422852 **
** Queue length = 8803 with q[q_front].cost = 9482.679688 **
** Queue length = 9100 with q[q_front].cost = 9493.128906 **
** Queue length = 9341 with q[q_front].cost = 9505.552734 **
  <<<< Performing Beam Search Reduction >>>>
** Queue length = 5817 with q[q_front].cost = 9514.939453 **
** Queue length = 6162 with q[q_front].cost = 9522.248047 **
** Queue length = 6497 with q[q_front].cost = 9529.062500 **
** Queue length = 6810 with q[q_front].cost = 9537.083984 **
** Queue length = 7103 with q[q_front].cost = 9545.895508 **
** Queue length = 7425 with q[q_front].cost = 9551.791016 **
** Queue length = 7757 with q[q_front].cost = 9557.912109 **
** Queue length = 8040 with q[q_front].cost = 9565.529297 **
** Queue length = 8347 with q[q_front].cost = 9571.066406 **
** Queue length = 8629 with q[q_front].cost = 9578.438477 **
** Queue length = 8938 with q[q_front].cost = 9584.368164 **
** Queue length = 9219 with q[q_front].cost = 9589.790039 **
** Queue length = 9473 with q[q_front].cost = 9597.320313 **
  <<<< Performing Beam Search Reduction >>>>
** Queue length = 6010 with q[q_front].cost = 9601.845703 **
** Queue length = 6348 with q[q_front].cost = 9606.309570 **
** Queue length = 6692 with q[q_front].cost = 9609.179688 **
** Queue length = 7032 with q[q_front].cost = 9614.427734 **
** Queue length = 7313 with q[q_front].cost = 9617.783203 **
** Queue length = 7637 with q[q_front].cost = 9623.195313 **
** Queue length = 7954 with q[q_front].cost = 9627.179688 **
```

```
** Queue length = 8275 with q[q_front].cost = 9632.107422 **
** Queue length = 8530 with q[q_front].cost = 9636.320313 **
** Queue length = 8820 with q[q_front].cost = 9640.810547 **
** Queue length = 9116 with q[q_front].cost = 9646.219727 **
** Queue length = 9405 with q[q_front].cost = 9650.085938 **
  <<<< Performing Beam Search Reduction >>>>
** Queue length = 5924 with q[q_front].cost = 9654.554688 **
** Queue length = 6220 with q[q_front].cost = 9658.597656 **
** Queue length = 6535 with q[q_front].cost = 9662.877930 **
** Queue length = 6900 with q[q_front].cost = 9664.815430 **

 A Solution Path Has Been Found

  Node 1 expanded 15751 states

The Best Route for mission AFIT-GO is:
   x       y       z
   24      16       8
   24      17       8
   24      18       8
   23      19       8
   22      19       8
   21      19       8
   20      19       8
   19      19       8
   18      19       8
   17      19       8
   16      19       8
   15      19       8
   14      19       8
   13      19       8
   12      19       8
   11      18       8
   10      17       8
   10      16       8
    9      15       8
    9      14       8
    8      13       8
    8      12       8
    7      11       8
    6      10       7
    6       9       7
    6       8       7
    5       7       6
    5       6       6
    4       5       5
    4       4       5
    3       3       5
    2       2       5
    1       1       5

For a total of 33 entries in the route

 At a distance of 37924.074219 (  7.18 miles)

 With a radar cost of 2600.000000

 And a computed cost of 9664.815430

        ***  Timing Information  ***

Initialize =     0.349 (sec)     Find Initial Route =     7.511 (sec)
Searching  = 32636.116 (sec)     Total Execution    = 32643.976 (sec)

Average worker node efficiency 0.875
```

```
The controller efficiency was  0.511
** 15751 nodes sent to processors
** 15751 total nodes expanded
```

C.4.1.2 4 Nodes.

```
          PARALLEL MISSION ROUTING PROBLEM

              A* USING CENTRALIZED LIST
        ************************************

Enter name of file containing the terrain data: terrainA
Enter name of file containing the radar data : radarA
Enter name of file containing the Air Tasking Order (ATO): AFIT-GOA


Waiting for results ...

   Received Initial Path with a computed cost of 15976.821289

 ** Queue length = 342 with q[q_front].cost = 7295.641602 **
 ** Queue length = 605 with q[q_front].cost = 7548.078125 **
 ** Queue length = 867 with q[q_front].cost = 7752.152344 **
 ** Queue length = 1112 with q[q_front].cost = 7930.727051 **
 ** Queue length = 1347 with q[q_front].cost = 8079.891113 **
 ** Queue length = 1633 with q[q_front].cost = 8176.913086 **
 ** Queue length = 1884 with q[q_front].cost = 8274.081055 **
 ** Queue length = 2160 with q[q_front].cost = 8356.519531 **
 ** Queue length = 2442 with q[q_front].cost = 8426.970703 **
 ** Queue length = 2700 with q[q_front].cost = 8495.869141 **
 ** Queue length = 2976 with q[q_front].cost = 8560.510742 **
 ** Queue length = 3245 with q[q_front].cost = 8617.153320 **
 ** Queue length = 3487 with q[q_front].cost = 8677.940430 **
 ** Queue length = 3735 with q[q_front].cost = 8729.084961 **
 ** Queue length = 3988 with q[q_front].cost = 8780.481445 **
 ** Queue length = 4205 with q[q_front].cost = 8829.189453 **
 ** Queue length = 4398 with q[q_front].cost = 8879.531250 **
 ** Queue length = 4617 with q[q_front].cost = 8922.499023 **
 ** Queue length = 4828 with q[q_front].cost = 8963.523438 **
 ** Queue length = 5040 with q[q_front].cost = 9000.816406 **
 ** Queue length = 5225 with q[q_front].cost = 9044.284180 **
 ** Queue length = 5446 with q[q_front].cost = 9080.322266 **
 ** Queue length = 5630 with q[q_front].cost = 9119.196289 **
 ** Queue length = 5816 with q[q_front].cost = 9160.290039 **
 ** Queue length = 6014 with q[q_front].cost = 9194.660156 **
 ** Queue length = 6177 with q[q_front].cost = 9231.916016 **
 ** Queue length = 6336 with q[q_front].cost = 9264.474609 **
 ** Queue length = 6524 with q[q_front].cost = 9296.173828 **
 ** Queue length = 6747 with q[q_front].cost = 9321.712891 **
 ** Queue length = 6947 with q[q_front].cost = 9349.755859 **
 ** Queue length = 7185 with q[q_front].cost = 9374.731445 **
 ** Queue length = 7421 with q[q_front].cost = 9395.030273 **
 ** Queue length = 7655 with q[q_front].cost = 9415.109375 **
 ** Queue length = 7903 with q[q_front].cost = 9432.028320 **
 ** Queue length = 8142 with q[q_front].cost = 9449.147461 **
 ** Queue length = 8414 with q[q_front].cost = 9463.954102 **
 ** Queue length = 8652 with q[q_front].cost = 9478.705078 **
 ** Queue length = 8929 with q[q_front].cost = 9491.339844 **
 ** Queue length = 9186 with q[q_front].cost = 9502.972656 **
 ** Queue length = 9433 with q[q_front].cost = 9514.648438 **
   <<<< Performing Beam Search Reduction >>>>
```

```
** Queue length = 5973 with q[q_front].cost = 9522.298828 **
** Queue length = 6295 with q[q_front].cost = 9530.350586 **
** Queue length = 6634 with q[q_front].cost = 9537.341797 **
** Queue length = 6950 with q[q_front].cost = 9546.695313 **
** Queue length = 7268 with q[q_front].cost = 9552.005859 **
** Queue length = 7592 with q[q_front].cost = 9557.977539 **
** Queue length = 7879 with q[q_front].cost = 9566.185547 **
** Queue length = 8184 with q[q_front].cost = 9573.007813 **
** Queue length = 8464 with q[q_front].cost = 9580.919922 **
** Queue length = 8779 with q[q_front].cost = 9586.035156 **
** Queue length = 9063 with q[q_front].cost = 9592.123047 **
** Queue length = 9353 with q[q_front].cost = 9599.110352 **
  <<<< Performing Beam Search Reduction >>>>
** Queue length = 5866 with q[q_front].cost = 9604.585938 **
** Queue length = 6198 with q[q_front].cost = 9608.302734 **
** Queue length = 6553 with q[q_front].cost = 9612.832031 **
** Queue length = 6868 with q[q_front].cost = 9616.813477 **
** Queue length = 7191 with q[q_front].cost = 9621.819336 **
** Queue length = 7497 with q[q_front].cost = 9626.884766 **
** Queue length = 7816 with q[q_front].cost = 9632.060547 **
** Queue length = 8085 with q[q_front].cost = 9636.320313 **
** Queue length = 8365 with q[q_front].cost = 9641.240234 **
** Queue length = 8674 with q[q_front].cost = 9646.460938 **
** Queue length = 8943 with q[q_front].cost = 9651.353516 **
** Queue length = 9220 with q[q_front].cost = 9656.503906 **
** Queue length = 9479 with q[q_front].cost = 9662.414063 **
  <<<< Performing Beam Search Reduction >>>>
** Queue length = 6034 with q[q_front].cost = 9664.815430 **

 A Solution Path Has Been Found

  Node 2 expanded 5183 states
  Node 1 expanded 5220 states
  Node 3 expanded 5223 states

The Best Route for mission AFIT-G0 is:
   x       y       z
   24      16       8
   24      17       8
   24      18       8
   23      19       8
   22      19       8
   21      19       8
   20      19       8
   19      19       8
   18      19       8
   17      19       8
   16      19       8
   15      19       8
   14      19       8
   13      19       8
   12      19       8
   11      18       8
   11      17       8
   10      16       8
    9      15       8
    9      14       8
    8      13       8
    8      12       8
    7      11       8
    6      10       7
    6       9       7
    5       8       7
    4       7       6
    3       6       6
```

```
    2       5       5
    2       4       5
    1       3       5
    1       2       5
    1       1       5

 For a total of 33 entries in the route

  At a distance of 37924.070313 (  7.18 miles)

  With a radar cost of 2600.000000

  And a computed cost of 9664.815430

         ***  Timing Information  ***

Initialize =     0.557 (sec)     Find Initial Route =     7.498 (sec)
Searching  = 17960.518 (sec)     Total Execution    = 17968.573 (sec)

Average worker node efficiency 0.519
The controller efficiency was  0.912
** 15626 nodes sent to processors
** 15626 total nodes expanded
```

C.4.1.3 8 Nodes.

```
          PARALLEL MISSION ROUTING PROBLEM

              A* USING CENTRALIZED LIST
        ************************************

Enter name of file containing the terrain data: terrainA
Enter name of file containing the radar data : radarA
Enter name of file containing the Air Tasking Order (ATO): AFIT-GOA


Waiting for results ...

   Received Initial Path with a computed cost of 15976.821289

 ** Queue length = 335 with q[q_front].cost = 7324.488281 **
 ** Queue length = 604 with q[q_front].cost = 7575.170898 **
 ** Queue length = 859 with q[q_front].cost = 7780.726563 **
 ** Queue length = 1122 with q[q_front].cost = 7945.687988 **
 ** Queue length = 1364 with q[q_front].cost = 8079.891113 **
 ** Queue length = 1621 with q[q_front].cost = 8185.544922 **
 ** Queue length = 1874 with q[q_front].cost = 8284.203125 **
 ** Queue length = 2147 with q[q_front].cost = 8366.806641 **
 ** Queue length = 2429 with q[q_front].cost = 8435.746094 **
 ** Queue length = 2674 with q[q_front].cost = 8514.046875 **
 ** Queue length = 2939 with q[q_front].cost = 8574.901367 **
 ** Queue length = 3200 with q[q_front].cost = 8634.073242 **
 ** Queue length = 3474 with q[q_front].cost = 8685.714844 **
 ** Queue length = 3716 with q[q_front].cost = 8736.662109 **
 ** Queue length = 3958 with q[q_front].cost = 8793.707031 **
 ** Queue length = 4186 with q[q_front].cost = 8842.523438 **
 ** Queue length = 4393 with q[q_front].cost = 8889.791016 **
 ** Queue length = 4626 with q[q_front].cost = 8926.709961 **
 ** Queue length = 4815 with q[q_front].cost = 8970.699219 **
 ** Queue length = 5024 with q[q_front].cost = 9010.871094 **
 ** Queue length = 5207 with q[q_front].cost = 9052.476563 **
```

```
** Queue length = 5425 with q[q_front].cost = 9090.109375 **
** Queue length = 5618 with q[q_front].cost = 9129.294922 **
** Queue length = 5796 with q[q_front].cost = 9171.066406 **
** Queue length = 5983 with q[q_front].cost = 9204.410156 **
** Queue length = 6168 with q[q_front].cost = 9243.416016 **
** Queue length = 6319 with q[q_front].cost = 9278.300781 **
** Queue length = 6532 with q[q_front].cost = 9306.005859 **
** Queue length = 6737 with q[q_front].cost = 9336.372070 **
** Queue length = 6962 with q[q_front].cost = 9363.667969 **
** Queue length = 7203 with q[q_front].cost = 9385.583984 **
** Queue length = 7458 with q[q_front].cost = 9403.602539 **
** Queue length = 7712 with q[q_front].cost = 9422.628906 **
** Queue length = 7950 with q[q_front].cost = 9439.339844 **
** Queue length = 8204 with q[q_front].cost = 9454.852539 **
** Queue length = 8466 with q[q_front].cost = 9468.913086 **
** Queue length = 8702 with q[q_front].cost = 9483.596680 **
** Queue length = 8992 with q[q_front].cost = 9493.122070 **
** Queue length = 9261 with q[q_front].cost = 9503.337891 **
  <<<< Performing Beam Search Reduction >>>>
** Queue length = 5765 with q[q_front].cost = 9513.602539 **
** Queue length = 6118 with q[q_front].cost = 9520.907227 **
** Queue length = 6429 with q[q_front].cost = 9529.002930 **
** Queue length = 6753 with q[q_front].cost = 9535.935547 **
** Queue length = 7071 with q[q_front].cost = 9544.583984 **
** Queue length = 7354 with q[q_front].cost = 9551.201172 **
** Queue length = 7679 with q[q_front].cost = 9556.960938 **
** Queue length = 7962 with q[q_front].cost = 9563.821289 **
** Queue length = 8273 with q[q_front].cost = 9568.460938 **
** Queue length = 8556 with q[q_front].cost = 9576.163086 **
** Queue length = 8846 with q[q_front].cost = 9581.949219 **
** Queue length = 9129 with q[q_front].cost = 9587.375000 **
** Queue length = 9433 with q[q_front].cost = 9594.171875 **
  <<<< Performing Beam Search Reduction >>>>
** Queue length = 5915 with q[q_front].cost = 9600.625977 **
** Queue length = 6268 with q[q_front].cost = 9605.172852 **
** Queue length = 6593 with q[q_front].cost = 9608.302734 **
** Queue length = 6946 with q[q_front].cost = 9612.832031 **
** Queue length = 7260 with q[q_front].cost = 9616.287109 **
** Queue length = 7564 with q[q_front].cost = 9621.755859 **
** Queue length = 7861 with q[q_front].cost = 9625.895508 **
** Queue length = 8186 with q[q_front].cost = 9630.810547 **
** Queue length = 8462 with q[q_front].cost = 9635.562500 **
** Queue length = 8724 with q[q_front].cost = 9639.828125 **
** Queue length = 9000 with q[q_front].cost = 9646.185547 **
** Queue length = 9290 with q[q_front].cost = 9649.708984 **
  <<<< Performing Beam Search Reduction >>>>
** Queue length = 5751 with q[q_front].cost = 9655.906250 **
** Queue length = 6069 with q[q_front].cost = 9659.177734 **
** Queue length = 6394 with q[q_front].cost = 9664.203125 **

 A Solution Path Has Been Found

  Node 1 expanded 2263 states
  Node 2 expanded 2264 states
  Node 5 expanded 2262 states
  Node 3 expanded 2279 states
  Node 4 expanded 2272 states
  Node 6 expanded 2266 states
  Node 7 expanded 2287 states

The Best Route for mission AFIT-GO is:
   x       y       z
   24      16       8
   24      17       8
   24      18       8
```

```
   23      19      8
   22      19      8
   21      19      8
   20      19      8
   19      19      8
   18      19      8
   17      19      8
   16      19      8
   15      19      8
   14      19      8
   13      19      8
   12      19      8
   11      18      8
   11      17      8
   11      16      8
   10      15      8
    9      14      8
    8      13      8
    8      12      8
    7      11      8
    7      10      8
    6       9      7
    5       8      7
    5       7      7
    4       6      6
    3       5      5
    3       4      5
    2       3      5
    1       2      5
    1       1      5

 For a total of 33 entries in the route

  At a distance of 37924.070313 (  7.18 miles)

  With a radar cost of 2600.000000

  And a computed cost of 9664.815430

        ***  Timing Information  ***

Initialize =     1.199 (sec)     Find Initial Route =     9.236 (sec)
Searching  = 16998.520 (sec)     Total Execution    = 17008.955 (sec)

Average worker node efficiency 0.242
The controller efficiency was  0.984
** 15893 nodes sent to processors
** 15893 total nodes expanded
```

C.4.2 Bounded With Angle = 60.0 (Depth = 4).

C.4.2.1 2 Nodes.

```
        PARALLEL MISSION ROUTING PROBLEM

            A* USING CENTRALIZED LIST
      ************************************

Enter name of file containing the terrain data: terrainA
Enter name of file containing the radar data : radarA
```

```
Enter name of file containing the Air Tasking Order (ATO): AFIT-GOA

Waiting for results ...

  Received Initial Path with a computed cost of 15976.821289

 ** Queue length = 323 with q[q_front].cost = 7376.946289 **
 ** Queue length = 571 with q[q_front].cost = 7698.872070 **
 ** Queue length = 843 with q[q_front].cost = 7938.537109 **
 ** Queue length = 1102 with q[q_front].cost = 8135.414063 **
 ** Queue length = 1386 with q[q_front].cost = 8283.530273 **
 ** Queue length = 1655 with q[q_front].cost = 8403.310547 **
 ** Queue length = 1894 with q[q_front].cost = 8520.799805 **
 ** Queue length = 2132 with q[q_front].cost = 8619.304688 **
 ** Queue length = 2427 with q[q_front].cost = 8687.996094 **
 ** Queue length = 2706 with q[q_front].cost = 8745.828125 **
 ** Queue length = 2983 with q[q_front].cost = 8806.241211 **
 ** Queue length = 3236 with q[q_front].cost = 8868.263672 **
 ** Queue length = 3480 with q[q_front].cost = 8927.907227 **
 ** Queue length = 3706 with q[q_front].cost = 8987.048828 **
 ** Queue length = 3946 with q[q_front].cost = 9035.540039 **
 ** Queue length = 4181 with q[q_front].cost = 9082.395508 **
 ** Queue length = 4404 with q[q_front].cost = 9133.377930 **
 ** Queue length = 4644 with q[q_front].cost = 9172.050781 **
 ** Queue length = 4840 with q[q_front].cost = 9221.103516 **
 ** Queue length = 5074 with q[q_front].cost = 9255.421875 **
 ** Queue length = 5262 with q[q_front].cost = 9294.814453 **
 ** Queue length = 5471 with q[q_front].cost = 9329.046875 **
 ** Queue length = 5670 with q[q_front].cost = 9362.250000 **
 ** Queue length = 5929 with q[q_front].cost = 9386.564453 **
 ** Queue length = 6174 with q[q_front].cost = 9411.462891 **
 ** Queue length = 6448 with q[q_front].cost = 9432.028320 **
 ** Queue length = 6697 with q[q_front].cost = 9453.063477 **
 ** Queue length = 6973 with q[q_front].cost = 9469.876953 **
 ** Queue length = 7224 with q[q_front].cost = 9488.949219 **
 ** Queue length = 7515 with q[q_front].cost = 9501.388672 **
 ** Queue length = 7779 with q[q_front].cost = 9514.825195 **
 ** Queue length = 8046 with q[q_front].cost = 9527.974609 **
 ** Queue length = 8318 with q[q_front].cost = 9539.441406 **
 ** Queue length = 8553 with q[q_front].cost = 9551.052734 **
 ** Queue length = 8866 with q[q_front].cost = 9557.927734 **
 ** Queue length = 9116 with q[q_front].cost = 9568.031250 **
 ** Queue length = 9357 with q[q_front].cost = 9579.812500 **
  <<<< Performing Beam Search Reduction >>>>
 ** Queue length = 5886 with q[q_front].cost = 9584.863281 **
 ** Queue length = 6194 with q[q_front].cost = 9591.085938 **
 ** Queue length = 6483 with q[q_front].cost = 9600.386719 **
 ** Queue length = 6790 with q[q_front].cost = 9606.309570 **
 ** Queue length = 7122 with q[q_front].cost = 9611.364258 **
 ** Queue length = 7384 with q[q_front].cost = 9617.248047 **
 ** Queue length = 7691 with q[q_front].cost = 9625.031250 **
 ** Queue length = 7990 with q[q_front].cost = 9630.723633 **
 ** Queue length = 8230 with q[q_front].cost = 9638.064453 **
 ** Queue length = 8522 with q[q_front].cost = 9643.056641 **
 ** Queue length = 8786 with q[q_front].cost = 9648.606445 **
 ** Queue length = 9050 with q[q_front].cost = 9655.595703 **
 ** Queue length = 9288 with q[q_front].cost = 9662.012695 **
  <<<< Performing Beam Search Reduction >>>>
 ** Queue length = 5829 with q[q_front].cost = 9664.815430 **

 A Solution Path Has Been Found

  Node 1 expanded 12165 states
```

```
The Best Route for mission AFIT-GO is:
   x       y       z
   24      16      8
   24      17      8
   24      18      8
   23      19      8
   22      19      8
   21      19      8
   20      19      8
   19      19      8
   18      19      8
   17      19      8
   16      19      8
   15      19      8
   14      19      8
   13      19      8
   12      19      8
   11      18      8
   11      17      8
   10      16      8
   10      15      8
    9      14      8
    9      13      8
    8      12      8
    7      11      7
    7      10      7
    7       9      7
    6       8      6
    6       7      6
    5       6      5
    5       5      5
    4       4      5
    3       3      5
    2       2      5
    1       1      5

 For a total of 33 entries in the route

  At a distance of 37924.074219 (  7.18 miles)

  With a radar cost of 2600.000000

  And a computed cost of 9664.815430

         ***  Timing Information  ***

Initialize =      0.183 (sec)     Find Initial Route =     34.807 (sec)
Searching  = 147856.127 (sec)     Total Execution    = 147891.117 (sec)

Average worker node efficiency 0.988
The controller efficiency was  0.076
** 12165 nodes sent to processors
** 12165 total nodes expanded
```

C.4.2.2 4 Nodes.

```
          PARALLEL MISSION ROUTING PROBLEM

             A* USING CENTRALIZED LIST
        ************************************
```

```
Enter name of file containing the terrain data: terrainA
Enter name of file containing the radar data : radarA
Enter name of file containing the Air Tasking Order (ATO): AFIT-GOA


Waiting for results ...

   Received Initial Path with a computed cost of 15976.821289

 ** Queue length = 326 with q[q_front].cost = 7503.325195 **
 ** Queue length = 607 with q[q_front].cost = 7832.151855 **
 ** Queue length = 883 with q[q_front].cost = 8067.336914 **
 ** Queue length = 1166 with q[q_front].cost = 8244.138672 **
 ** Queue length = 1461 with q[q_front].cost = 8371.456055 **
 ** Queue length = 1704 with q[q_front].cost = 8496.833008 **
 ** Queue length = 1949 with q[q_front].cost = 8607.810547 **
 ** Queue length = 2242 with q[q_front].cost = 8685.939453 **
 ** Queue length = 2518 with q[q_front].cost = 8747.978516 **
 ** Queue length = 2792 with q[q_front].cost = 8817.796875 **
 ** Queue length = 3058 with q[q_front].cost = 8880.573242 **
 ** Queue length = 3293 with q[q_front].cost = 8941.316406 **
 ** Queue length = 3546 with q[q_front].cost = 8995.882813 **
 ** Queue length = 3768 with q[q_front].cost = 9051.890625 **
 ** Queue length = 4007 with q[q_front].cost = 9105.364258 **
 ** Queue length = 4251 with q[q_front].cost = 9148.771484 **
 ** Queue length = 4490 with q[q_front].cost = 9189.117188 **
 ** Queue length = 4721 with q[q_front].cost = 9234.304688 **
 ** Queue length = 4964 with q[q_front].cost = 9269.062500 **
 ** Queue length = 5169 with q[q_front].cost = 9308.505859 **
 ** Queue length = 5351 with q[q_front].cost = 9344.503906 **
 ** Queue length = 5595 with q[q_front].cost = 9373.716797 **
 ** Queue length = 5864 with q[q_front].cost = 9397.166016 **
 ** Queue length = 6131 with q[q_front].cost = 9419.016602 **
 ** Queue length = 6392 with q[q_front].cost = 9440.634766 **
 ** Queue length = 6657 with q[q_front].cost = 9463.057617 **
 ** Queue length = 6953 with q[q_front].cost = 9477.013672 **
 ** Queue length = 7229 with q[q_front].cost = 9492.023438 **
 ** Queue length = 7484 with q[q_front].cost = 9508.224609 **
 ** Queue length = 7773 with q[q_front].cost = 9520.907227 **
 ** Queue length = 8045 with q[q_front].cost = 9531.462891 **
 ** Queue length = 8261 with q[q_front].cost = 9547.046875 **
 ** Queue length = 8553 with q[q_front].cost = 9555.260742 **
 ** Queue length = 8808 with q[q_front].cost = 9566.185547 **
 ** Queue length = 9073 with q[q_front].cost = 9576.163086 **
 ** Queue length = 9342 with q[q_front].cost = 9583.721680 **
   <<<< Performing Beam Search Reduction >>>>
 ** Queue length = 5850 with q[q_front].cost = 9589.790039 **
 ** Queue length = 6161 with q[q_front].cost = 9597.539063 **
 ** Queue length = 6450 with q[q_front].cost = 9606.105469 **
 ** Queue length = 6785 with q[q_front].cost = 9610.175781 **
 ** Queue length = 7070 with q[q_front].cost = 9616.287109 **
 ** Queue length = 7362 with q[q_front].cost = 9624.122070 **
 ** Queue length = 7671 with q[q_front].cost = 9629.186523 **
 ** Queue length = 7931 with q[q_front].cost = 9635.562500 **
 ** Queue length = 8193 with q[q_front].cost = 9641.604492 **
 ** Queue length = 8491 with q[q_front].cost = 9648.041992 **
 ** Queue length = 8765 with q[q_front].cost = 9653.536133 **
 ** Queue length = 9007 with q[q_front].cost = 9660.012695 **
 ** Queue length = 9305 with q[q_front].cost = 9664.815430 **
   <<<< Performing Beam Search Reduction >>>>

  A Solution Path Has Been Found

   Node 2 expanded 3887 states
   Node 1 expanded 3855 states
```

```
Node 3 expanded 3799 states

The Best Route for mission AFIT-GO is:
   x       y       z
   24      16       8
   24      17       8
   24      18       8
   23      19       8
   22      19       8
   21      19       8
   20      19       8
   19      19       8
   18      19       8
   17      19       8
   16      19       8
   15      19       8
   14      19       8
   13      19       8
   12      19       8
   11      18       8
   11      17       8
   10      16       8
   10      15       8
    9      14       8
    9      13       8
    8      12       8
    7      11       7
    6      10       7
    6       9       7
    5       8       6
    5       7       6
    4       6       5
    4       5       5
    4       4       5
    3       3       5
    2       2       5
    1       1       5

 For a total of 33 entries in the route

  At a distance of 37924.074219 (  7.18 miles)

  With a radar cost of 2600.000000

  And a computed cost of 9664.815430

         ***  Timing Information  ***

Initialize =     0.358 (sec)     Find Initial Route =    34.913 (sec)
Searching  = 47864.490 (sec)     Total Execution    = 47899.761 (sec)

Average worker node efficiency 0.982
The controller efficiency was  0.221
** 11541 nodes sent to processors
** 11541 total nodes expanded
```

C.4.2.3 8 Nodes.

```
PARALLEL MISSION ROUTING PROBLEM

   A* USING CENTRALIZED LIST
```

```
        ************************************

Enter name of file containing the terrain data: terrainA
Enter name of file containing the radar data : radarA
Enter name of file containing the Air Tasking Order (ATO): AFIT-GOA


Waiting for results ...

   Received Initial Path with a computed cost of 15976.821289

 ** Queue length = 304 with q[q_front].cost = 7493.672852 **
 ** Queue length = 565 with q[q_front].cost = 7817.017090 **
 ** Queue length = 833 with q[q_front].cost = 8058.233398 **
 ** Queue length = 1112 with q[q_front].cost = 8225.800781 **
 ** Queue length = 1404 with q[q_front].cost = 8359.893555 **
 ** Queue length = 1677 with q[q_front].cost = 8488.294922 **
 ** Queue length = 1931 with q[q_front].cost = 8587.010742 **
 ** Queue length = 2213 with q[q_front].cost = 8666.118164 **
 ** Queue length = 2501 with q[q_front].cost = 8730.224609 **
 ** Queue length = 2767 with q[q_front].cost = 8796.770508 **
 ** Queue length = 3041 with q[q_front].cost = 8859.888672 **
 ** Queue length = 3280 with q[q_front].cost = 8924.345703 **
 ** Queue length = 3501 with q[q_front].cost = 8983.913086 **
 ** Queue length = 3738 with q[q_front].cost = 9035.605469 **
 ** Queue length = 3959 with q[q_front].cost = 9086.886719 **
 ** Queue length = 4205 with q[q_front].cost = 9138.521484 **
 ** Queue length = 4449 with q[q_front].cost = 9179.313477 **
 ** Queue length = 4677 with q[q_front].cost = 9222.627930 **
 ** Queue length = 4908 with q[q_front].cost = 9259.384766 **
 ** Queue length = 5123 with q[q_front].cost = 9298.426758 **
 ** Queue length = 5342 with q[q_front].cost = 9332.226563 **
 ** Queue length = 5562 with q[q_front].cost = 9363.702148 **
 ** Queue length = 5813 with q[q_front].cost = 9390.601563 **
 ** Queue length = 6074 with q[q_front].cost = 9413.194336 **
 ** Queue length = 6331 with q[q_front].cost = 9433.788086 **
 ** Queue length = 6594 with q[q_front].cost = 9455.077148 **
 ** Queue length = 6887 with q[q_front].cost = 9470.712891 **
 ** Queue length = 7150 with q[q_front].cost = 9489.152344 **
 ** Queue length = 7429 with q[q_front].cost = 9503.215820 **
 ** Queue length = 7685 with q[q_front].cost = 9519.367188 **
 ** Queue length = 7968 with q[q_front].cost = 9529.500977 **
 ** Queue length = 8228 with q[q_front].cost = 9541.574219 **
 ** Queue length = 8490 with q[q_front].cost = 9551.791016 **
 ** Queue length = 8758 with q[q_front].cost = 9561.313477 **
 ** Queue length = 9037 with q[q_front].cost = 9571.353516 **
 ** Queue length = 9265 with q[q_front].cost = 9581.714844 **
   <<<< Performing Beam Search Reduction >>>>
 ** Queue length = 5827 with q[q_front].cost = 9587.034180 **
 ** Queue length = 6142 with q[q_front].cost = 9595.438477 **
 ** Queue length = 6424 with q[q_front].cost = 9603.039063 **
 ** Queue length = 6742 with q[q_front].cost = 9608.302734 **
 ** Queue length = 7077 with q[q_front].cost = 9614.427734 **
 ** Queue length = 7349 with q[q_front].cost = 9620.373047 **
 ** Queue length = 7639 with q[q_front].cost = 9627.178711 **
 ** Queue length = 7949 with q[q_front].cost = 9634.159180 **
 ** Queue length = 8177 with q[q_front].cost = 9639.456055 **
 ** Queue length = 8463 with q[q_front].cost = 9646.185547 **
 ** Queue length = 8733 with q[q_front].cost = 9651.791992 **
 ** Queue length = 8981 with q[q_front].cost = 9658.558594 **
 ** Queue length = 9253 with q[q_front].cost = 9664.815430 **
   <<<< Performing Beam Search Reduction >>>>
 ** Queue length = 5828 with q[q_front].cost = 9664.815430 **

  A Solution Path Has Been Found
```

```
Node 1 expanded 1679 states
Node 2 expanded 1670 states
Node 4 expanded 1670 states
Node 3 expanded 1668 states
Node 5 expanded 1623 states
Node 6 expanded 1639 states
Node 7 expanded 1689 states

The Best Route for mission AFIT-G0 is:
   x       y       z
  24      16       8
  24      17       8
  24      18       8
  23      19       8
  22      19       8
  21      19       8
  20      19       8
  19      19       8
  18      19       8
  17      19       8
  16      19       8
  15      19       8
  14      19       8
  13      19       8
  12      19       8
  11      18       8
  10      17       8
   9      16       8
   9      15       8
   9      14       8
   8      13       8
   7      12       8
   6      11       7
   5      10       7
   4       9       7
   3       8       6
   3       7       6
   2       6       5
   2       5       5
   2       4       5
   2       3       5
   1       2       5
   1       1       5

For a total of 33 entries in the route

 At a distance of 37924.074219 (  7.18 miles)

 With a radar cost of 2600.000000

 And a computed cost of 9664.815430

        ***  Timing Information  ***

Initialize =     0.941 (sec)     Find Initial Route =    37.321 (sec)
Searching  = 21183.054 (sec)     Total Execution    = 21221.316 (sec)

Average worker node efficiency 0.956
The controller efficiency was  0.503
** 11638 nodes sent to processors
** 11638 total nodes expanded
```

C.4.3 Bounded With Angle = 59.0 (Depth = 3).

C.4.3.1 2 Nodes.

```
PARALLEL MISSION ROUTING PROBLEM

A* USING CENTRALIZED LIST
************************************

Enter name of file containing the terrain data: terrainA
Enter name of file containing the radar data : radarA
Enter name of file containing the Air Tasking Order (ATO): AFIT-GOA


Waiting for results ...

  Received Initial Path with a computed cost of 15976.821289

 ** Queue length = 309 with q[q_front].cost = 7609.132813 **
 ** Queue length = 563 with q[q_front].cost = 8092.900391 **
 ** Queue length = 853 with q[q_front].cost = 8383.039063 **
 ** Queue length = 1136 with q[q_front].cost = 8600.023438 **
 ** Queue length = 1386 with q[q_front].cost = 8806.270508 **
 ** Queue length = 1650 with q[q_front].cost = 8960.724609 **
 ** Queue length = 1918 with q[q_front].cost = 9108.349609 **
 ** Queue length = 2181 with q[q_front].cost = 9232.999023 **
 ** Queue length = 2472 with q[q_front].cost = 9323.435547 **
 ** Queue length = 2788 with q[q_front].cost = 9386.617188 **
 ** Queue length = 3111 with q[q_front].cost = 9422.214844 **
 ** Queue length = 3446 with q[q_front].cost = 9452.700195 **
 ** Queue length = 3758 with q[q_front].cost = 9474.687500 **
 ** Queue length = 4079 with q[q_front].cost = 9494.670898 **
 ** Queue length = 4371 with q[q_front].cost = 9512.826172 **
 ** Queue length = 4675 with q[q_front].cost = 9529.002930 **
 ** Queue length = 4954 with q[q_front].cost = 9545.173828 **
 ** Queue length = 5248 with q[q_front].cost = 9556.621094 **
 ** Queue length = 5535 with q[q_front].cost = 9567.283203 **
 ** Queue length = 5766 with q[q_front].cost = 9581.111328 **
 ** Queue length = 6049 with q[q_front].cost = 9589.790039 **
 ** Queue length = 6271 with q[q_front].cost = 9602.035156 **
 ** Queue length = 6548 with q[q_front].cost = 9609.772461 **
 ** Queue length = 6771 with q[q_front].cost = 9618.826172 **
 ** Queue length = 7037 with q[q_front].cost = 9627.578125 **
 ** Queue length = 7217 with q[q_front].cost = 9636.536133 **
 ** Queue length = 7437 with q[q_front].cost = 9646.185547 **
 ** Queue length = 7656 with q[q_front].cost = 9654.634766 **
 ** Queue length = 7808 with q[q_front].cost = 9664.366211 **

 A Solution Path Has Been Found

  Node 1 expanded 6817 states

The Best Route for mission AFIT-GO is:
   x      y      z
   24     16     8
   24     17     8
   24     18     8
   23     19     8
   22     19     8
   21     19     8
   20     19     8
   19     19     8
```

```
   18      19      8
   17      19      8
   16      19      8
   15      19      8
   14      19      8
   13      19      8
   12      19      8
   11      18      8
   10      17      8
   10      16      8
    9      15      8
    9      14      8
    8      13      8
    8      12      8
    7      11      8
    6      10      7
    6       9      7
    6       8      7
    5       7      6
    5       6      6
    4       5      5
    4       4      5
    3       3      5
    2       2      5
    1       1      5

 For a total of 33 entries in the route

  At a distance of 37924.074219 (  7.18 miles)

  With a radar cost of 2600.000000

  And a computed cost of 9664.815430

         `**  Timing Information  ***

Initialize =     0.484 (sec)     Find Initial Route =     6.013 (sec)
Searching  =  8441.781 (sec)     Total Execution    =  8448.278 (sec)

Average worker node efficiency 0.893
The controller efficiency was  0.413
** 6817 nodes sent to processors
** 6817 total nodes expanded
```

C.4.3.2 4 Nodes.

```
         PARALLEL MISSION ROUTING PROBLEM

             A* USING CENTRALIZED LIST
       ************************************

Enter name of file containing the terrain data: terrainA
Enter name of file containing the radar data : radarA
Enter name of file containing the Air Tasking Order (ATO): AFIT-GOA


Waiting for results ...

   Received Initial Path with a computed cost of 15976.821289

 ** Queue length = 308 with q[q_front].cost = 7691.256836 **
```

```
** Queue length = 573 with q[q_front].cost = 8158.927734 **
** Queue length = 857 with q[q_front].cost = 8428.045898 **
** Queue length = 1149 with q[q_front].cost = 8635.816406 **
** Queue length = 1422 with q[q_front].cost = 8812.970703 **
** Queue length = 1671 with q[q_front].cost = 8984.738281 **
** Queue length = 1937 with q[q_front].cost = 9112.332031 **
** Queue length = 2195 with q[q_front].cost = 9231.916016 **
** Queue length = 2463 with q[q_front].cost = 9340.158203 **
** Queue length = 2752 with q[q_front].cost = 9432.745117 **
** Queue length = 3058 with q[q_front].cost = 9488.003906 **
** Queue length = 3399 with q[q_front].cost = 9529.167969 **
** Queue length = 3699 with q[q_front].cost = 9563.847656 **
** Queue length = 4020 with q[q_front].cost = 9587.947266 **
** Queue length = 4332 with q[q_front].cost = 9611.217773 **
** Queue length = 4672 with q[q_front].cost = 9624.118164 **
** Queue length = 4955 with q[q_front].cost = 9643.644531 **
** Queue length = 5264 with q[q_front].cost = 9655.515625 **
** Queue length = 5539 with q[q_front].cost = 9672.458008 **
** Queue length = 5820 with q[q_front].cost = 9682.289063 **
** Queue length = 6105 with q[q_front].cost = 9693.055664 **
** Queue length = 6349 with q[q_front].cost = 9705.886719 **
** Queue length = 6624 with q[q_front].cost = 9714.297852 **
** Queue length = 6850 with q[q_front].cost = 9725.752930 **
** Queue length = 7110 with q[q_front].cost = 9733.441406 **
** Queue length = 7362 with q[q_front].cost = 9741.497070 **
** Queue length = 7610 with q[q_front].cost = 9750.850586 **
** Queue length = 7854 with q[q_front].cost = 9758.968750 **
** Queue length = 8060 with q[q_front].cost = 9765.359375 **
** Queue length = 8261 with q[q_front].cost = 9774.593750 **
** Queue length = 8449 with q[q_front].cost = 9783.105469 **
** Queue length = 8700 with q[q_front].cost = 9789.323242 **

 A Solution Path Has Been Found

  Node 1 expanded 2478 states
  Node 2 expanded 2440 states
  Node 3 expanded 2456 states

The Best Route for mission AFIT-GO is:
```

x	y	z
24	16	8
23	17	8
22	18	8
21	19	8
20	19	8
19	19	8
18	19	8
17	19	8
16	19	8
15	19	8
14	19	8
13	19	8
12	19	8
11	18	8
10	17	8
9	16	8
9	15	8
8	14	8
8	13	8
7	12	8
6	11	8
6	10	8
5	9	7
4	8	7
3	7	6

```
    3       6       6
    2       5       5
    2       4       5
    1       3       5
    1       2       5
    1       1       5

 For a total of 31 entries in the route

  At a distance of 36752.500000 (  6.96 miles)

  With a radar cost of 3048.528076

  And a computed cost of 9789.323242

         ***  Timing Information  ***

Initialize =     0.582 (sec)     Find Initial Route =     5.661 (sec)
Searching  =  4765.459 (sec)     Total Execution    =  4771.702 (sec)

Average worker node efficiency 0.571
The controller efficiency was  0.865
** 7374 nodes sent to processors
** 7374 total nodes expanded
```

C.4.4 Bounded With Angle = 59.0 (Depth = 4).

C.4.4.1 2 Nodes.

```
          PARALLEL MISSION ROUTING PROBLEM

              A* USING CENTRALIZED LIST
        ************************************

Enter name of file containing the terrain data: terrainA
Enter name of file containing the radar data : radarA
Enter name of file containing the Air Tasking Order (ATO): AFIT-GOA


Waiting for results ...

   Received Initial Path with a computed cost of 15976.821289

 ** Queue length = 292 with q[q_front].cost = 7782.239258 **
 ** Queue length = 564 with q[q_front].cost = 8350.895508 **
 ** Queue length = 837 with q[q_front].cost = 8669.387695 **
 ** Queue length = 1131 with q[q_front].cost = 8885.211914 **
 ** Queue length = 1397 with q[q_front].cost = 9086.886719 **
 ** Queue length = 1671 with q[q_front].cost = 9248.910156 **
 ** Queue length = 1958 with q[q_front].cost = 9352.745117 **
 ** Queue length = 2298 with q[q_front].cost = 9411.905273 **
 ** Queue length = 2630 with q[q_front].cost = 9450.676758 **
 ** Queue length = 2944 with q[q_front].cost = 9480.691406 **
 ** Queue length = 3276 with q[q_front].cost = 9503.215820 **
 ** Queue length = 3587 with q[q_front].cost = 9522.298828 **
 ** Queue length = 3873 with q[q_front].cost = 9544.767578 **
 ** Queue length = 4193 with q[q_front].cost = 9556.672852 **
 ** Queue length = 4488 with q[q_front].cost = 9569.969727 **
 ** Queue length = 4765 with q[q_front].cost = 9582.477539 **
 ** Queue length = 5046 with q[q_front].cost = 9595.438477 **
```

```
** Queue length = 5271 with q[q_front].cost = 9608.302734 **
** Queue length = 5547 with q[q_front].cost = 9616.287109 **
** Queue length = 5804 with q[q_front].cost = 9627.950195 **
** Queue length = 6031 with q[q_front].cost = 9638.064453 **
** Queue length = 6280 with q[q_front].cost = 9648.165039 **
** Queue length = 6467 with q[q_front].cost = 9658.556641 **
** Queue length = 6748 with q[q_front].cost = 9664.815430 **

 A Solution Path Has Been Found

  Node 1 expanded 5353 states

The Best Route for mission AFIT-GO is:
   x       y       z
   24      16       8
   24      17       8
   24      18       8
   23      19       8
   22      19       8
   21      19       8
   20      19       8
   19      19       8
   18      19       8
   17      19       8
   16      19       8
   15      19       8
   14      19       8
   13      19       8
   12      19       8
   11      18       8
   11      17       8
   10      16       8
   10      15       8
    9      14       8
    9      13       8
    8      12       8
    7      11       7
    7      10       7
    7       9       7
    6       8       6
    6       7       6
    5       6       5
    5       5       5
    4       4       5
    3       3       5
    2       2       5
    1       1       5

 For a total of 33 entries in the route

  At a distance of 37924.074219 (  7.18 miles)

  With a radar cost of 2600.000000

  And a computed cost of 9664.815430

        ***  Timing Information  ***

Initialize =     0.323 (sec)     Find Initial Route =    18.438 (sec)
Searching  = 32698.327 (sec)     Total Execution    = 32717.088 (sec)

Average worker node efficiency 0.986
The controller efficiency was  0.072
** 5353 nodes sent to processors
** 5353 total nodes expanded
```

C.4.4.2 4 Nodes.

```
PARALLEL MISSION ROUTING PROBLEM

A* USING CENTRALIZED LIST
*************************************

Enter name of file containing the terrain data: terrainA
Enter name of file containing the radar data : radarA
Enter name of file containing the Air Tasking Order (ATO): AFIT-GOA


Waiting for results ...

  Received Initial Path with a computed cost of 15976.821289

 ** Queue length = 303 with q[q_front].cost = 7999.010742 **
 ** Queue length = 591 with q[q_front].cost = 8468.818359 **
 ** Queue length = 865 with q[q_front].cost = 8754.843750 **
 ** Queue length = 1148 with q[q_front].cost = 8980.826172 **
 ** Queue length = 1424 with q[q_front].cost = 9156.640625 **
 ** Queue length = 1699 with q[q_front].cost = 9312.875000 **
 ** Queue length = 1998 with q[q_front].cost = 9414.377930 **
 ** Queue length = 2328 with q[q_front].cost = 9474.386719 **
 ** Queue length = 2676 with q[q_front].cost = 9511.164063 **
 ** Queue length = 3012 with q[q_front].cost = 9541.574219 **
 ** Queue length = 3317 with q[q_front].cost = 9567.021484 **
 ** Queue length = 3621 with q[q_front].cost = 9586.769531 **
 ** Queue length = 3934 with q[q_front].cost = 9599.423828 **
 ** Queue length = 4236 with q[q_front].cost = 9614.621094 **
 ** Queue length = 4526 with q[q_front].cost = 9627.723633 **
 ** Queue length = 4769 with q[q_front].cost = 9642.463867 **
 ** Queue length = 5093 with q[q_front].cost = 9651.353516 **
 ** Queue length = 5298 with q[q_front].cost = 9664.279297 **
 ** Queue length = 5532 with q[q_front].cost = 9676.298828 **
 ** Queue length = 5794 with q[q_front].cost = 9684.090820 **

 A Solution Path Has Been Found

  Node 2 expanded 1433 states
  Node 1 expanded 1412 states
  Node 3 expanded 1439 states

 The Best Route for mission AFIT-GO is:
   x      y      z
   24     16     8
   24     17     8
   24     18     7
   23     19     6
   22     19     6
   21     19     6
   20     19     6
   19     19     6
   18     19     6
   17     19     6
   16     19     6
   15     19     6
   14     19     6
   13     19     6
   12     19     6
```

```
   11      18      6
   10      17      6
    9      16      6
    8      15      6
    8      14      6
    8      13      6
    7      12      6
    7      11      6
    6      10      6
    5       9      6
    5       8      6
    5       7      6
    4       6      5
    4       5      5
    4       4      5
    3       3      5
    2       2      5
    1       1      5

 For a total of 33 entries in the route

  At a distance of 38020.449219 (  7.20 miles)

  With a radar cost of 2600.000000

  And a computed cost of 9684.090820

         ***  Timing Information  ***

Initialize =     0.549 (sec)     Find Initial Route =    18.479 (sec)
Searching  =  8269.536 (sec)     Total Execution    =  8288.564 (sec)

Average worker node efficiency 0.981
The controller efficiency was  0.193
** 4284 nodes sent to processors
** 4284 total nodes expanded
```

Appendix D. *Angles Between Directional Vectors*

D.1 Problem Analysis

Once it was determined there was a difference in values between the calculated angles possible sources of this error were identified as either hardware or software. Even though the i860 is advertised as a 64-bit microprocessor its instruction set and integer registers are 32-bits wide, while the floating point units are 64-bits (26:1-2). The i860 does support both 32- and 64-bit representation of real number which are in accordance with ANSI/IEEE standard 754-1985 (26:1-4,2-2). The 386 has 32-bit instructions with a 32-bit data bus (28:5-355). The architecture does support 32-, 64-, and 80-bit real numbers (28:5-305). The format used to represent the 32- and 64-bit data is not given, nor is there any indication whether any of the representations conform to any ANSI/IEEE standard. The assumption can not be made that because the microprocessors are made by the same company that they use the same representations or conform to the same standards. Thus, it is possible that the differences in hardware could be causing the accuracy problem.

Chapter III presented the equations used to derive the formula for calculating the angle between two 3-dimensional vectors which is

$$\theta = \arccos \frac{a_1 b_1 + a_2 b_2 + a_3 b_3}{|\mathbf{A}||\mathbf{B}|} \quad . \tag{D.1}$$

If θ is less than or equal to the field-of-view variable then the child location is considered reachable from the parent location, thus the child is valid. Otherwise the child is not valid and is rejected. Since the directional vectors are unit vectors, a and b can only take on the values -1, 0, and 1. Because order does not matter $a_i b_i$ also can only take on the values -1, 0, and 1. The representation of the environment is a cube meaning all edges have the same length, this means that $\mathbf{A}$ and $\mathbf{B}$ can only take on the values 1, $\sqrt{2}$, and $\sqrt{3}$ which means that $\mathbf{A} \cdot \mathbf{B}$ will have the values 1, $\sqrt{2}$, $\sqrt{3}$, $\sqrt{6}$, 2, and 3. This reduces the number of combinations which needed to be examined. A spreadsheet is

used to generate the results contained in Table D.1. Since all possible combinations are examined there are some combinations which are not physically possible. For instance if $a_i b_i$ are all -1 then the two vectors go from one corner to the opposite. Both vectors have a magnitude of $\sqrt{3}$. Thus for $a_i b_i$ equal all ones $\mathbf{A} \cdot \mathbf{B}$ can only be equal to 3, all other possible values of $\mathbf{A} \cdot \mathbf{B}$ are not physically possible. Figure D.1 is an example of some possible directional vectors.

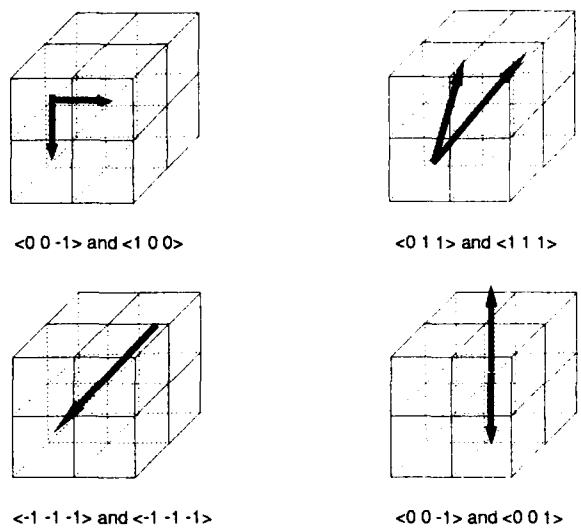

Figure D.1. Examples of Directional Vectors

D.2 Execution Results

D.2.1 Insertions Into the Open List. The following sections contain a partial output listing. The sequential version of the code is executed for ease of determining the operation of the algorithm. The following abbreviations are used in the output:

P	-	Parent (Last entry in the Route)
L	-	Number of entries in the route
D	-	Directional vector used to each parent
C	-	Cost (f')
G	-	Cost (g)
R	-	Accumulated radar cost
D	-	Distance traveled
BC	-	Best cost

D.2.1.1 iPSC/2 (Mission AFIT-0A).

```
PARALLEL MISSION ROUTING PROBLEM

A* USING CENTRALIZED LIST
************************************

Enter name of file containing the terrain data: terrainA
Enter name of file containing the radar data : radarA
Enter name of file containing the Air Tasking Order (ATO): AFIT-0A
 Found an initial route with a cost of 5690.290527
REMOVED - P(17 17 8) L( 1) D( 0 0 0) C( 0.0000) G( 0.0000) R( 0.0000) D( 0.0000) BC(5690.2905)
  INSERTED- P(17 18 7) L( 2) D( 0 1 -1) C(4236.1880) G( 707.1068) R( 848.5282) D( 1414.2136) BC(5690.2905)
  INSERTED- P(18 18 7) L( 2) D( 1 1 -1) C(4412.5815) G( 519.6152) R( 762.1024) D( 1732.0508) BC(5690.2905)
  INSERTED- P(18 17 7) L( 2) D( 1 0 -1) C(4340.4175) G( 424.2641) R( 622.2540) D( 1414.2136) BC(5690.2905)
  INSERTED- P(18 16 7) L( 2) D( 1 -1 -1) C(3973.9595) G( 519.6152) R( 762.1024) D( 1732.0508) BC(5690.2905)
  INSERTED- P(17 16 7) L( 2) D( 0 -1 -1) C(3798.2222) G( 707.1068) R( 848.5282) D( 1414.2136) BC(5690.2905)
  INSERTED- P(16 16 7) L( 2) D(-1 -1 -1) C(4224.6904) G(1212.4355) R(1316.3586) D( 1732.0508) BC(5690.2905)
  INSERTED- P(16 17 7) L( 2) D(-1 0 -1) C(4904.7998) G(1131.3710) R(1187.9395) D( 1414.2136) BC(5690.2905)
  INSERTED- P(16 18 7) L( 2) D(-1 1 -1) C(4925.3516) G(1212.4355) R(1316.3586) D( 1732.0508) BC(5690.2905)
  INSERTED- P(17 18 8) L( 2) D( 0 1 0) C(3955.1392) G( 500.0000) R( 600.0000) D( 1000.0000) BC(5690.2905)
  INSERTED- P(18 18 8) L( 2) D( 1 1 0) C(4294.6421) G( 424.2641) R( 622.2540) D( 1414.2136) BC(5690.2905)
  INSERTED- P(18 17 8) L( 2) D( 1 0 0) C(4181.4131) G( 300.0000) R( 440.0000) D( 1000.0000) BC(5690.2905)
  INSERTED- P(18 16 8) L( 2) D( 1 -1 0) C(3877.6450) G( 424.2641) R( 622.2540) D( 1414.2136) BC(5690.2905)
  INSERTED- P(17 16 8) L( 2) D( 0 -1 0) C(3589.9670) G( 500.0000) R( 600.0000) D( 1000.0000) BC(5690.2905)
  INSERTED- P(16 16 8) L( 2) D(-1 -1 0) C(4040.3633) G( 989.9495) R(1074.8024) D( 1414.2136) BC(5690.2905)
  INSERTED- P(16 17 8) L( 2) D(-1 0 0) C(4393.5005) G( 800.0000) R( 840.0000) D( 1000.0000) BC(5690.2905)
  INSERTED- P(16 18 8) L( 2) D(-1 1 0) C(4718.3232) G( 989.9495) R(1074.8024) D( 1414.2136) BC(5690.2905)
  INSERTED- P(17 18 9) L( 2) D( 0 1 1) C(4358.2202) G( 707.1068) R( 848.5282) D( 1414.2136) BC(5690.2905)
  INSERTED- P(18 18 9) L( 2) D( 1 1 1) C(4470.6230) G( 519.6152) R( 762.1024) D( 1732.0508) BC(5690.2905)
  INSERTED- P(18 17 9) L( 2) D( 1 0 1) C(4401.9604) G( 424.2641) R( 622.2540) D( 1414.2136) BC(5690.2905)
  INSERTED- P(18 16 9) L( 2) D( 1 -1 1) C(4087.1873) G( 519.6152) R( 762.1024) D( 1732.0508) BC(5690.2905)
  INSERTED- P(17 16 9) L( 2) D( 0 -1 1) C(3903.3296) G( 707.1068) R( 848.5282) D( 1414.2136) BC(5690.2905)
  INSERTED- P(16 16 9) L( 2) D(-1 -1 1) C(4371.0771) G(1212.4355) R(1316.3586) D( 1732.0508) BC(5690.2905)
  INSERTED- P(16 17 9) L( 2) D(-1 0 1) C(5051.1860) G(1131.3710) R(1187.9395) D( 1414.2136) BC(5690.2905)
  INSERTED- P(16 18 9) L( 2) D(-1 1 1) C(5015.9639) G(1212.4355) R(1316.3586) D( 1732.0508) BC(5690.2905)
REMOVED - P(17 16 8) L( 2) D( 0 -1 0) C(3589.9670) G( 500.0000) R( 600.0000) D( 1000.0000) BC(5690.2905)
  NOT VALID- P(17 16 8) C(17 17 7)
  NOT VALID- P(17 16 8) C(18 17 7)
  NOT VALID- P(17 16 8) C(18 16 7)
  INSERTED- P(18 15 7) L( 3) D( 1 -1 -1) C(4796.6821) G(1192.8203) R(1500.6664) D( 2732.0508) BC(5690.2905)
  INSERTED- P(17 15 7) L( 3) D( 0 -1 -1) C(4290.5396) G( 924.2641) R(1222.2540) D( 2414.2136) BC(5690.2905)
  INSERTED- P(16 15 7) L( 3) D(-1 -1 -1) C(4299.0342) G(1366.0254) R(1639.2305) D( 2732.0508) BC(5690.2905)
  NOT VALID- P(17 16 8) C(16 16 7)
  NOT VALID- P(17 16 8) C(16 17 7)
  NOT VALID- P(17 16 8) C(17 17 8)
  NOT VALID- P(17 16 8) C(18 17 8)
  NOT VALID- P(17 16 8) C(18 16 8)
  INSERTED- P(18 15 8) L( 3) D( 1 -1 0) C(4677.4321) G(1065.6854) R(1335.3911) D( 2414.2136) BC(5690.2905)
  INSERTED- P(17 15 8) L( 3) D( 0 -1 0) C(4149.6938) G( 800.0000) R(1040.0000) D( 2000.0000) BC(5690.2905)
  INSERTED- P(16 15 8) L( 3) D(-1 -1 0) C(4165.5610) G(1207.1068) R(1448.5282) D( 2414.2136) BC(5690.2905)
  NOT VALID- P(17 16 8) C(16 16 8)
  NOT VALID- P(17 16 8) C(16 17 8)
  NOT VALID- P(17 16 8) C(17 17 9)
  NOT VALID- P(17 16 8) C(18 17 9)
  NOT VALID- P(17 16 8) C(18 16 9)
  INSERTED- P(18 15 9) L( 3) D( 1 -1 1) C(4916.0576) G(1192.8203) R(1500.6664) D( 2732.0508) BC(5690.2905)
  INSERTED- P(17 15 9) L( 3) D( 0 -1 1) C(4398.4829) G( 924.2641) R(1222.2540) D( 2414.2136) BC(5690.2905)
  INSERTED- P(16 15 9) L( 3) D(-1 -1 1) C(4445.4209) G(1366.0254) R(1639.2305) D( 2732.0508) BC(5690.2905)
  NOT VALID- P(17 16 8) C(16 16 9)
  NOT VALID- P(17 16 8) C(16 17 9)
REMOVED - P(17 16 7) L( 2) D( 0 -1 -1) C(3798.2222) G( 707.1068) R( 848.5282) D( 1414.2136) BC(5690.2905)
  NOT VALID- P(17 16 7) C(17 17 6)
  NOT VALID- P(17 16 7) C(18 17 6)
  NOT VALID- P(17 16 7) C(18 16 6)
  INSERTED- P(18 15 6) L( 3) D( 1 -1 -1) C(5029.4966) G(1399.9271) R(1749.1946) D( 3146.2644) BC(5690.2905)
  INSERTED- P(17 15 6) L( 3) D( 0 -1 -1) C(4524.9961) G(1131.3710) R(1470.7822) D( 2828.4272) BC(5690.2905)
  INSERTED- P(16 15 6) L( 3) D(-1 -1 -1) C(4527.7588) G(1573.1322) R(1887.7587) D( 3146.2644) BC(5690.2905)
  NOT VALID- P(17 16 7) C(16 16 6)
  NOT VALID- P(17 16 7) C(16 17 6)
  NOT VALID- P(17 16 7) C(17 17 7)
  NOT VALID- P(17 16 7) C(18 17 7)
```

```
 NOT VALID- P(17 16  7)  C(18 16  7)
 NOT VALID- P(17 16  7)  C(18 15  7)
 INSERTED- P(17 15  7)  L( 3)  D( 0 -1  0)  C(4356.8140) G(1007.1068) R(1288.5282) D( 2414.2136)  BC(5690.2905)
 NOT VALID- P(17 16  7)  C(16 15  7)
 NOT VALID- P(17 16  7)  C(16 16  7)
 NOT VALID- P(17 16  7)  C(16 17  7)
 NOT VALID- P(17 16  7)  C(17 17  8)
 NOT VALID- P(17 16  7)  C(18 17  8)
 NOT VALID- P(17 16  7)  C(18 16  8)
 NOT VALID- P(17 16  7)  C(18 15  8)
 NOT VALID- P(17 16  7)  C(17 15  8)
 NOT VALID- P(17 16  7)  C(16 15  8)
 NOT VALID- P(17 16  7)  C(16 16  8)
 NOT VALID- P(17 16  7)  C(16 17  8)
REMOVED -  P(17 16  9)  L( 2)  D( 0 -1  1)  C(3903.3296) G( 707.1068) R( 848.5282) D( 1414.2136)  BC(5690.2905)
 NOT VALID- P(17 16  9)  C(17 17  8)
 NOT VALID- P(17 16  9)  C(18 17  8)
 NOT VALID- P(17 16  9)  C(18 16  8)
 NOT VALID- P(17 16  9)  C(18 15  8)
 NOT VALID- P(17 16  9)  C(17 15  8)
 NOT VALID- P(17 16  9)  C(16 15  8)
 NOT VALID- P(17 16  9)  C(16 16  8)
 NOT VALID- P(17 16  9)  C(16 17  8)
 NOT VALID- P(17 16  9)  C(17 17  9)
 NOT VALID- P(17 16  9)  C(18 17  9)
 NOT VALID- P(17 16  9)  C(18 16  9)
 NOT VALID- P(17 16  9)  C(18 15  9)
 INSERTED- P(17 15  9)  L( 3)  D( 0 -1  0)  C(4464.7573) G(1007.1068) R(1288.5282) D( 2414.2136)  BC(5690.2905)
 NOT VALID- P(17 16  9)  C(16 15  9)
 NOT VALID- P(17 16  9)  C(16 16  9)
 NOT VALID- P(17 16  9)  C(16 17  9)
 NOT VALID- P(17 16  9)  C(17 17 10)
 NOT VALID- P(17 16  9)  C(18 17 10)
 NOT VALID- P(17 16  9)  C(18 16 10)
 INSERTED- P(18 15 10)  L( 3)  D( 1 -1  1)  C(5261.2490) G(1399.9271) R(1749.1946) D( 3146.2644)  BC(5690.2905)
 INSERTED- P(17 15 10)  L( 3)  D( 0 -1  1)  C(4735.5898) G(1131.3710) R(1470.7822) D( 2828.4272)  BC(5690.2905)
 INSERTED- P(16 15 10)  L( 3)  D(-1 -1  1)  C(4808.3833) G(1573.1322) R(1887.7587) D( 3146.2644)  BC(5690.2905)
 NOT VALID- P(17 16  9)  C(16 16 10)
 NOT VALID- P(17 16  9)  C(16 17 10)
```

D.2.1.2 iPSC/860 (Mission AFIT-0A).

PARALLEL MISSION ROUTING PROBLEM

A* USING CENTRALIZED LIST
****

```
Enter name of file containing the terrain data: terrainA
Enter name of file containing the radar data : radarA
Enter name of file containing the Air Tasking Order (ATO): AFIT-0A
 Found an initial route with a cost of 5690.290039
REMOVED -  P(17 17  8)  L( 1)  D( 0  0  0)  C(   0.0000) G(   0.0000) R(   0.0000) D(   0.0000)  BC(5690.2900)
 INSERTED- P(17 18  7)  L( 2)  D( 0  1 -1)  C(4180.4175) G( 707.1068) R( 848.5281) D( 1414.2135)  BC(5690.2900)
 INSERTED- P(18 18  7)  L( 2)  D( 1  1 -1)  C(4134.7300) G( 519.6152) R( 762.1024) D( 1732.0508)  BC(5690.2900)
 INSERTED- P(18 17  7)  L( 2)  D( 1  0 -1)  C(3908.9822) G( 424.2641) R( 622.2540) D( 1414.2135)  BC(5690.2900)
 INSERTED- P(18 16  7)  L( 2)  D( 1 -1 -1)  C(3973.9595) G( 519.6152) R( 762.1024) D( 1732.0508)  BC(5690.2900)
 INSERTED- P(17 16  7)  L( 2)  D( 0 -1 -1)  C(3798.2219) G( 707.1068) R( 848.5281) D( 1414.2135)  BC(5690.2900)
 INSERTED- P(16 16  7)  L( 2)  D(-1 -1 -1)  C(4224.6904) G(1212.4355) R(1316.3586) D( 1732.0508)  BC(5690.2900)
 INSERTED- P(16 17  7)  L( 2)  D(-1  0 -1)  C(4684.2109) G(1131.3708) R(1187.9393) D( 1414.2135)  BC(5690.2900)
 INSERTED- P(16 18  7)  L( 2)  D(-1  1 -1)  C(4925.3516) G(1212.4355) R(1316.3586) D( 1732.0508)  BC(5690.2900)
 INSERTED- P(17 18  8)  L( 2)  D( 0  1  0)  C(3955.1392) G( 500.0000) R( 600.0000) D( 1000.0000)  BC(5690.2900)
 INSERTED- P(18 18  8)  L( 2)  D( 1  1  0)  C(3968.6196) G( 424.2641) R( 622.2540) D( 1414.2135)  BC(5690.2900)
 INSERTED- P(18 17  8)  L( 2)  D( 1  0  0)  C(3777.5442) G( 300.0000) R( 440.0000) D( 1000.0000)  BC(5690.2900)
 INSERTED- P(18 16  8)  L( 2)  D( 1 -1  0)  C(3877.6450) G( 424.2641) R( 622.2540) D( 1414.2135)  BC(5690.2900)
 INSERTED- P(17 16  8)  L( 2)  D( 0 -1  0)  C(3589.9668) G( 500.0000) R( 600.0000) D( 1000.0000)  BC(5690.2900)
 INSERTED- P(16 16  8)  L( 2)  D(-1 -1  0)  C(4040.3633) G( 989.9495) R(1074.8022) D( 1414.2135)  BC(5690.2900)
 INSERTED- P(16 17  8)  L( 2)  D(-1  0  0)  C(4393.5005) G( 800.0000) R( 840.0000) D( 1000.0000)  BC(5690.2900)
 INSERTED- P(16 18  8)  L( 2)  D(-1  1  0)  C(4718.3232) G( 989.9495) R(1074.8022) D( 1414.2135)  BC(5690.2900)
 INSERTED- P(17 18  9)  L( 2)  D( 0  1  1)  C(4212.4355) G( 707.1068) R( 848.5281) D( 1414.2135)  BC(5690.2900)
 INSERTED- P(18 18  9)  L( 2)  D( 1  1  1)  C(4357.3335) G( 519.6152) R( 762.1024) D( 1732.0508)  BC(5690.2900)
 INSERTED- P(18 17  9)  L( 2)  D( 1  0  1)  C(3968.6196) G( 424.2641) R( 622.2540) D( 1414.2135)  BC(5690.2900)
 INSERTED- P(18 16  9)  L( 2)  D( 1 -1  1)  C(4087.1873) G( 519.6152) R( 762.1024) D( 1732.0508)  BC(5690.2900)
 INSERTED- P(17 16  9)  L( 2)  D( 0 -1  1)  C(3903.3296) G( 707.1068) R( 848.5281) D( 1414.2135)  BC(5690.2900)
 INSERTED- P(16 16  9)  L( 2)  D(-1 -1  1)  C(4371.0771) G(1212.4355) R(1316.3586) D( 1732.0508)  BC(5690.2900)
```

```
INSERTED- P(16 17  9)  L( 2)  D(-1  0  1)  C(4830.5972) G(1131.3708) R(1187.9393) D( 1414.2135)  BC(5690.2900)
INSERTED- P(16 18  9)  L( 2)  D(-1  1  1)  C(5015.9639) G(1212.4355) R(1316.3586) D( 1732.0508)  BC(5690.2900)
REMOVED - P(17 16  8)  L( 2)  D( 0 -1  0)  C(3589.9668) G( 500.0000) R( 600.0000) D( 1000.0000)  BC(5690.2900)
NOT VALID- P(17 16  8)  C(17 17  7)
NOT VALID- P(17 16  8)  C(18 17  7)
NOT VALID- P(17 16  8)  C(18 16  7)
INSERTED- P(18 15  7)  L( 3)  D( 1 -1 -1)  C(4796.6821) G(1192.8203) R(1500.6664) D( 2732.0508)  BC(5690.2900)
INSERTED- P(17 15  7)  L( 3)  D( 0 -1 -1)  C(4290.5396) G( 924.2640) R(1222.2539) D( 2414.2134)  BC(5690.2900)
INSERTED- P(16 15  7)  L( 3)  D(-1 -1 -1)  C(4299.0342) G(1366.0254) R(1639.2305) D( 2732.0508)  BC(5690.2900)
NOT VALID- P(17 16  8)  C(16 16  7)
NOT VALID- P(17 16  8)  C(16 17  7)
NOT VALID- P(17 16  8)  C(17 17  8)
NOT VALID- P(17 16  8)  C(18 17  8)
NOT VALID- P(17 16  8)  C(18 16  8)
INSERTED- P(18 15  8)  L( 3)  D( 1 -1  0)  C(4677.4321) G(1065.6854) R(1335.3911) D( 2414.2134)  BC(5690.2900)
INSERTED- P(17 15  8)  L( 3)  D( 0 -1  0)  C(4149.6938) G( 800.0000) R(1040.0000) D( 2000.0000)  BC(5690.2900)
INSERTED- P(16 15  8)  L( 3)  D(-1 -1  0)  C(4165.5610) G(1207.1067) R(1448.5281) D( 2414.2134)  BC(5690.2900)
NOT VALID- P(17 16  8)  C(16 16  8)
NOT VALID- P(17 16  8)  C(16 17  8)
NOT VALID- P(17 16  8)  C(17 17  9)
NOT VALID- P(17 16  8)  C(18 17  9)
NOT VALID- P(17 16  8)  C(18 16  9)
INSERTED- P(18 15  9)  L( 3)  D( 1 -1  1)  C(4916.0576) G(1192.8203) R(1500.6664) D( 2732.0508)  BC(5690.2900)
INSERTED- P(17 15  9)  L( 3)  D( 0 -1  1)  C(4398.4829) G( 924.2640) R(1222.2539) D( 2414.2134)  BC(5690.2900)
INSERTED- P(16 15  9)  L( 3)  D(-1 -1  1)  C(4445.4209) G(1366.0254) R(1639.2305) D( 2732.0508)  BC(5690.2900)
NOT VALID- P(17 16  8)  C(16 16  9)
NOT VALID- P(17 16  8)  C(16 17  9)
REMOVED - P(18 17  8)  L( 2)  D( 1  0  0)  C(3777.5442) G( 300.0000) R( 440.0000) D( 1000.0000)  BC(5690.2900)
NOT VALID- P(18 17  8)  C(18 18  7)
INSERTED- P(19 18  7)  L( 3)  D( 1  1 -1)  C(4204.6396) G( 646.4102) R(1063.5383) D( 2732.0508)  BC(5690.2900)
INSERTED- P(19 17  7)  L( 3)  D( 1  0 -1)  C(4307.1060) G( 582.8427) R( 949.1169) D( 2414.2134)  BC(5690.2900)
INSERTED- P(19 16  7)  L( 3)  D( 1 -1 -1)  C(4706.8594) G( 819.6152) R(1202.1023) D( 2732.0508)  BC(5690.2900)
NOT VALID- P(18 17  8)  C(18 16  7)
NOT VALID- P(18 17  8)  C(17 16  7)
NOT VALID- P(18 17  8)  C(17 17  7)
NOT VALID- P(18 17  8)  C(17 18  7)
NOT VALID- P(18 17  8)  C(18 18  8)
INSERTED- P(19 18  8)  L( 3)  D( 1  1  0)  C(4080.8318) G( 582.8427) R( 949.1169) D( 2414.2134)  BC(5690.2900)
INSERTED- P(19 17  8)  L( 3)  D( 1  0  0)  C(4396.2583) G( 500.0000) R( 800.0000) D( 2000.0000)  BC(5690.2900)
INSERTED- P(19 16  8)  L( 3)  D( 1 -1  0)  C(4605.5835) G( 724.2640) R(1062.2539) D( 2414.2134)  BC(5690.2900)
NOT VALID- P(18 17  8)  C(18 16  8)
NOT VALID- P(18 17  8)  C(17 16  8)
NOT VALID- P(18 17  8)  C(17 17  8)
NOT VALID- P(18 17  8)  C(17 18  8)
NOT VALID- P(18 17  8)  C(18 18  9)
INSERTED- P(19 18  9)  L( 3)  D( 1  1  1)  C(4418.6919) G( 646.4102) R(1063.5383) D( 2732.0508)  BC(5690.2900)
INSERTED- P(19 17  9)  L( 3)  D( 1  0  1)  C(4452.6396) G( 582.8427) R( 949.1169) D( 2414.2134)  BC(5690.2900)
INSERTED- P(19 16  9)  L( 3)  D( 1 -1  1)  C(4807.7012) G( 819.6152) R(1202.1023) D( 2732.0508)  BC(5690.2900)
NOT VALID- P(18 17  8)  C(18 16  9)
NOT VALID- P(18 17  8)  C(17 16  9)
NOT VALID- P(18 17  8)  C(17 17  9)
NOT VALID- P(18 17  8)  C(17 18  9)
REMOVED - P(17 16  7)  L( 2)  D( 0 -1 -1)  C(3798.2219) G( 707.1068) R( 848.5281) D( 1414.2135)  BC(5690.2900)
NOT VALID- P(17 16  7)  C(17 17  6)
NOT VALID- P(17 16  7)  C(18 17  6)
INSERTED- P(18 16  6)  L( 3)  D( 1  0 -1)  C(4885.4834) G(1131.3708) R(1470.7820) D( 2828.4270)  BC(5690.2900)
INSERTED- P(18 15  6)  L( 3)  D( 1 -1 -1)  C(5029.4966) G(1399.9270) R(1749.1945) D( 3146.2642)  BC(5690.2900)
INSERTED- P(17 15  6)  L( 3)  D( 0 -1 -1)  C(4524.9956) G(1131.3708) R(1470.7820) D( 2828.4270)  BC(5690.2900)
INSERTED- P(16 15  6)  L( 3)  D(-1 -1 -1)  C(4527.7588) G(1573.1321) R(1887.7585) D( 3146.2642)  BC(5690.2900)
INSERTED- P(16 16  6)  L( 3)  D(-1  0 -1)  C(4811.8584) G(1697.0562) R(1923.3303) D( 2828.4270)  BC(5690.2900)
NOT VALID- P(17 16  7)  C(16 17  6)
NOT VALID- P(17 16  7)  C(17 17  7)
NOT VALID- P(17 16  7)  C(18 17  7)
NOT VALID- P(17 16  7)  C(18 16  7)
INSERTED- P(18 15  7)  L( 3)  D( 1 -1  0)  C(4879.9351) G(1272.7922) R(1583.9192) D( 2828.4270)  BC(5690.2900)
INSERTED- P(17 15  7)  L( 3)  D( 0 -1  0)  C(4356.8135) G(1007.1068) R(1288.5281) D( 2414.2134)  BC(5690.2900)
INSERTED- P(16 15  7)  L( 3)  D(-1 -1  0)  C(4356.8604) G(1414.2135) R(1697.0562) D( 2828.4270)  BC(5690.2900)
NOT VALID- P(17 16  7)  C(16 16  7)
NOT VALID- P(17 16  7)  C(16 17  7)
NOT VALID- P(17 16  7)  C(17 17  8)
NOT VALID- P(17 16  7)  C(18 17  8)
NOT VALID- P(17 16  7)  C(18 16  8)
NOT VALID- P(17 16  7)  C(18 15  8)
NOT VALID- P(17 16  7)  C(17 15  8)
NOT VALID- P(17 16  7)  C(16 15  8)
NOT VALID- P(17 16  7)  C(16 16  8)
NOT VALID- P(17 16  7)  C(16 17  8)
```

```
REMOVED - P(18 16  8) L( 2) D( 1 -1  0) C(3877.6450) G( 424.2641) R( 622.2540) D( 1414.2135) BC(5690.2900)
 NOT VALID- P(18 16  8) C(18 17  7)
 NOT VALID- P(18 16  8) C(19 17  7)
 INSERTED- P(19 16  7) L( 3) D( 1  0 -1) C(4659.2095) G( 848.5281) R(1244.5079) D( 2828.4270) BC(5690.2900)
 INSERTED- P(19 15  7) L( 3) D( 1 -1 -1) C(5292.8438) G(1290.2894) R(1661.4844) D( 3146.2642) BC(5690.2900)
 INSERTED- P(18 15  7) L( 3) D( 0 -1 -1) C(4460.5239) G( 989.9495) R(1357.6450) D( 2828.4270) BC(5690.2900)
 NOT VALID- P(18 16  8) C(17 15  7)
 NOT VALID- P(18 16  8) C(17 16  7)
 NOT VALID- P(18 16  8) C(17 17  7)
 NOT VALID- P(18 16  8) C(18 17  8)
 NOT VALID- P(18 16  8) C(19 17  8)
 INSERTED- P(19 16  8) L( 3) D( 1  0  0) C(4768.6616) G( 724.2640) R(1062.2539) D( 2414.2134) BC(5690.2900)
 INSERTED- P(19 15  8) L( 3) D( 1 -1  0) C(5142.4141) G(1131.3708) R(1470.7820) D( 2828.4270) BC(5690.2900)
 INSERTED- P(18 15  8) L( 3) D( 0 -1  0) C(4390.5278) G( 824.2640) R(1142.2539) D( 2414.2134) BC(5690.2900)
 NOT VALID- P(18 16  8) C(17 15  8)
 NOT VALID- P(18 16  8) C(17 16  8)
 NOT VALID- P(18 16  8) C(17 17  8)
 NOT VALID- P(18 16  8) C(18 17  9)
 NOT VALID- P(18 16  8) C(19 17  9)
 INSERTED- P(19 16  9) L( 3) D( 1  0  1) C(4796.4678) G( 848.5281) R(1244.5079) D( 2828.4270) BC(5690.2900)
 INSERTED- P(19 15  9) L( 3) D( 1 -1  1) C(5397.9512) G(1290.2894) R(1661.4844) D( 3146.2642) BC(5690.2900)
 INSERTED- P(18 15  9) L( 3) D( 0 -1  1) C(4684.5239) G( 989.9495) R(1357.6450) D( 2828.4270) BC(5690.2900)
 NOT VALID- P(18 16  8) C(17 15  9)
 NOT VALID- P(18 16  8) C(17 16  9)
 NOT VALID- P(18 16  8) C(17 17  9)
```

D.2.2 Accuracy of Angle Calculations. For each test case the parent location used is (17, 16, 7) and the directional vector reaching the parent is $< 0 \quad -1 \quad -1 >$. These values are selected because on the iPSC/2 four children from this state were accepted as valid next locations while on the iPSC/860 those same four children plus an additional four other children were accepted. Thus for this state, the iPSC/860 accepted twice the number of children resulting in twice the number of possible routes being explored. This state allows a direct comparison of the angle calculations, therefore providing insight into the differences between the two executable codes. The following abbreviations are used in the output:

C	-	Child coordinates
ND	-	Directional vector from parent to child
NT	-	$x^2 + y^2 + z^2$ of ND
OT	-	$x^2 + y^2 + z^2$ of old direction vector
Num	-	Numerator
Den	-	Denominator
Fract	-	Fraction sent to arc cosine function
MO	-	Magnitude of old directional vector
MN	-	Magnitude of old directional vector
A	-	Calculated angle
TRUE	-	Child is valid

D.2.2.1 iPSC/2.

Starting testing

```
C(17 17 6) ND( 0 1 -1) NT(2) OT(2) Num( 0.0) Den(2.0000) Fract( 0.0000) MO(1.414) MN(1.414) A( 90.0000000000000000)
C(18 17 6) ND( 1 1 -1) NT(3) OT(2) Num( 0.0) Den(2.4495) Fract( 0.0000) MO(1.414) MN(1.732) A( 90.0000000000000000)
C(18 16 6) ND( 1 0 -1) NT(2) OT(2) Num( 1.0) Den(2.0000) Fract( 0.5000) MO(1.414) MN(1.414) A( 60.0000000000000070)
C(18 15 6) ND( 1 -1 -1) NT(3) OT(2) Num( 2.0) Den(2.4495) Fract( 0.8165) MO(1.414) MN(1.732) A( 35.2643896827546680) TRUE
C(17 15 6) ND( 0 -1 -1) NT(2) OT(2) Num( 2.0) Den(2.0000) Fract( 1.0000) MO(1.414) MN(1.414) A( 0.0000012074182697) TRUE
C(16 15 6) ND(-1 -1 -1) NT(3) OT(2) Num( 2.0) Den(2.4495) Fract( 0.8165) MO(1.414) MN(1.732) A( 35.2643896827546680) TRUE
C(16 16 6) ND(-1 0 -1) NT(2) OT(2) Num( 1.0) Den(2.0000) Fract( 0.5000) MO(1.414) MN(1.414) A( 60.0000000000000070)
C(16 17 6) ND(-1 1 -1) NT(3) OT(2) Num( 0.0) Den(2.4495) Fract( 0.0000) MO(1.414) MN(1.732) A( 90.0000000000000000)
C(17 17 7) ND( 0 1 0) NT(1) OT(2) Num(-1.0) Den(1.4142) Fract(-0.7071) MO(1.414) MN(1.000) A(135.0000000000000000)
C(18 17 7) ND( 1 1 0) NT(2) OT(2) Num(-1.0) Den(2.0000) Fract(-0.5000) MO(1.414) MN(1.414) A(119.9999999999999800)
C(18 16 7) ND( 1 0 0) NT(1) OT(2) Num( 0.0) Den(1.4142) Fract( 0.0000) MO(1.414) MN(1.000) A( 90.0000000000000000)
C(18 15 7) ND( 1 -1 0) NT(2) OT(2) Num( 1.0) Den(2.0000) Fract( 0.5000) MO(1.414) MN(1.414) A( 60.0000000000000070)
C(17 15 7) ND( 0 -1 0) NT(1) OT(2) Num( 1.0) Den(1.4142) Fract( 0.7071) MO(1.414) MN(1.000) A( 45.0000000000000070) TRUE
C(16 15 7) ND(-1 -1 0) NT(2) OT(2) Num( 1.0) Den(2.0000) Fract( 0.5000) MO(1.414) MN(1.414) A( 60.0000000000000070)
C(16 16 7) ND(-1 0 0) NT(1) OT(2) Num( 0.0) Den(1.4142) Fract( 0.0000) MO(1.414) MN(1.000) A( 90.0000000000000000)
C(16 17 7) ND(-1 1 0) NT(2) OT(2) Num(-1.0) Den(2.0000) Fract(-0.5000) MO(1.414) MN(1.414) A(119.9999999999999800)
C(17 17 8) ND( 0 1 1) NT(2) OT(2) Num(-2.0) Den(2.0000) Fract(-1.0000) MO(1.414) MN(1.414) A(179.9999987925817400)
C(18 17 8) ND( 1 1 1) NT(3) OT(2) Num(-2.0) Den(2.4495) Fract(-0.8165) MO(1.414) MN(1.732) A(144.7356103172453100)
C(18 16 8) ND( 1 0 1) NT(2) OT(2) Num(-1.0) Den(2.0000) Fract(-0.5000) MO(1.414) MN(1.414) A(119.9999999999999800)
C(18 15 8) ND( 1 -1 1) NT(3) OT(2) Num( 0.0) Den(2.4495) Fract( 0.0000) MO(1.414) MN(1.732) A( 90.0000000000000000)
C(17 15 8) ND( 0 -1 1) NT(2) OT(2) Num( 0.0) Den(2.0000) Fract( 0.0000) MO(1.414) MN(1.414) A( 90.0000000000000000)
C(16 15 8) ND(-1 -1 1) NT(3) OT(2) Num( 0.0) Den(2.4495) Fract( 0.0000) MO(1.414) MN(1.732) A( 90.0000000000000000)
C(16 16 8) ND(-1 0 1) NT(2) OT(2) Num(-1.0) Den(2.0000) Fract(-0.5000) MO(1.414) MN(1.414) A(119.9999999999999800)
C(16 17 8) ND(-1 1 1) NT(3) OT(2) Num(-2.0) Den(2.4495) Fract(-0.8165) MO(1.414) MN(1.732) A(144.7356103172453100)
```

D.2.2.2 iPSC/860.

Starting testing

```
C(17 17 6) ND( 0 1 -1) NT(2) OT(2) Num( 0.0) Den(2.0000) Fract( 0.0000) MO(1.414) MN(1.414) A( 90.0000000000000000)
C(18 17 6) ND( 1 1 -1) NT(3) OT(2) Num( 0.0) Den(2.4495) Fract( 0.0000) MO(1.414) MN(1.732) A( 90.0000000000000000)
C(18 16 6) ND( 1 0 -1) NT(2) OT(2) Num( 1.0) Den(2.0000) Fract( 0.5000) MO(1.414) MN(1.414) A( 59.9999999999999850) TRUE
C(18 15 6) ND( 1 -1 -1) NT(3) OT(2) Num( 2.0) Den(2.4495) Fract( 0.8165) MO(1.414) MN(1.732) A( 35.2643896827546390) TRUE
C(17 15 6) ND( 0 -1 -1) NT(2) OT(2) Num( 2.0) Den(2.0000) Fract( 1.0000) MO(1.414) MN(1.414) A( 0.0000000000000000) TRUE
C(16 15 6) ND(-1 -1 -1) NT(3) OT(2) Num( 2.0) Den(2.4495) Fract( 0.8165) MO(1.414) MN(1.732) A( 35.2643896827546390) TRUE
C(16 16 6) ND(-1 0 -1) NT(2) OT(2) Num( 1.0) Den(2.0000) Fract( 0.5000) MO(1.414) MN(1.414) A( 59.9999999999999850) TRUE
C(16 17 6) ND(-1 1 -1) NT(3) OT(2) Num( 0.0) Den(2.4495) Fract( 0.0000) MO(1.414) MN(1.732) A( 90.0000000000000000)
C(17 17 7) ND( 0 1 0) NT(1) OT(2) Num(-1.0) Den(1.4142) Fract(-0.7071) MO(1.414) MN(1.000) A(135.0000000000000000)
C(18 17 7) ND( 1 1 0) NT(2) OT(2) Num(-1.0) Den(2.0000) Fract(-0.5000) MO(1.414) MN(1.414) A(120.0000000000000100)
C(18 16 7) ND( 1 0 0) NT(1) OT(2) Num( 0.0) Den(1.4142) Fract( 0.0000) MO(1.414) MN(1.000) A( 90.0000000000000000)
C(18 15 7) ND( 1 -1 0) NT(2) OT(2) Num( 1.0) Den(2.0000) Fract( 0.5000) MO(1.414) MN(1.414) A( 59.9999999999999850) TRUE
C(17 15 7) ND( 0 -1 0) NT(1) OT(2) Num( 1.0) Den(1.4142) Fract( 0.7071) MO(1.414) MN(1.000) A( 44.9999999999999920) TRUE
C(16 15 7) ND(-1 -1 0) NT(2) OT(2) Num( 1.0) Den(2.0000) Fract( 0.5000) MO(1.414) MN(1.414) A( 59.9999999999999850) TRUE
C(16 16 7) ND(-1 0 0) NT(1) OT(2) Num( 0.0) Den(1.4142) Fract( 0.0000) MO(1.414) MN(1.000) A( 90.0000000000000000)
C(16 17 7) ND(-1 1 0) NT(2) OT(2) Num(-1.0) Den(2.0000) Fract(-0.5000) MO(1.414) MN(1.414) A(120.0000000000000100)
C(17 17 8) ND( 0 1 1) NT(2) OT(2) Num(-2.0) Den(2.0000) Fract(-1.0000) MO(1.414) MN(1.414) A(180.0000000000000000)
C(18 17 8) ND( 1 1 1) NT(3) OT(2) Num(-2.0) Den(2.4495) Fract(-0.8165) MO(1.414) MN(1.732) A(144.7356103172453700)
C(18 16 8) ND( 1 0 1) NT(2) OT(2) Num(-1.0) Den(2.0000) Fract(-0.5000) MO(1.414) MN(1.414) A(120.0000000000000100)
C(18 15 8) ND( 1 -1 1) NT(3) OT(2) Num( 0.0) Den(2.4495) Fract( 0.0000) MO(1.414) MN(1.732) A( 90.0000000000000000)
C(17 15 8) ND( 0 -1 1) NT(2) OT(2) Num( 0.0) Den(2.0000) Fract( 0.0000) MO(1.414) MN(1.414) A( 90.0000000000000000)
C(16 15 8) ND(-1 -1 1) NT(3) OT(2) Num( 0.0) Den(2.4495) Fract( 0.0000) MO(1.414) MN(1.732) A( 90.0000000000000000)
C(16 16 8) ND(-1 0 1) NT(2) OT(2) Num(-1.0) Den(2.0000) Fract(-0.5000) MO(1.414) MN(1.414) A(120.0000000000000100)
C(16 17 8) ND(-1 1 1) NT(3) OT(2) Num(-2.0) Den(2.4495) Fract(-0.8165) MO(1.414) MN(1.732) A(144.7356103172453700)
```

Table D.1. All Possible Angles

$a_i b_i$			$\mathbf{A} \cdot \mathbf{B}$	$a_1b_1 + a_2b_2 + a_3b_3$	$\frac{a_1b_1+a_2b_2+a_3b_3}{\mathbf{A} \cdot \mathbf{B}}$	Angle
0	0	1	1	1	1	0
0	1	1	1	2	2	ERR
1	1	1	1	3	3	ERR
0	0	-1	1	-1	-1	180
0	-1	-1	1	-2	-2	ERR
-1	-1	-1	1	-3	-3	ERR
0	1	-1	1	0	0	90
1	1	-1	1	1	1	0
1	-1	-1	1	-1	-1	180
0	0	1	$\sqrt{2}$	1	0.707107	45
0	1	1	$\sqrt{2}$	2	1.414214	ERR
1	1	1	$\sqrt{2}$	3	2.121320	ERR
0	0	-1	$\sqrt{2}$	-1	-0.707107	135
0	-1	-1	$\sqrt{2}$	-2	-1.414214	ERR
-1	-1	-1	$\sqrt{2}$	-3	-2.121320	ERR
0	1	-1	$\sqrt{2}$	0	0	90
1	1	-1	$\sqrt{2}$	1	0.707107	45
1	-1	-1	$\sqrt{2}$	-1	-0.707107	135
0	0	1	$\sqrt{3}$	1	0.577350	54.73561
0	1	1	$\sqrt{3}$	2	1.154701	ERR
1	1	1	$\sqrt{3}$	3	1.732051	ERR
0	0	-1	$\sqrt{3}$	-1	-0.577350	125.26439
0	-1	-1	$\sqrt{3}$	-2	-1.154701	ERR
-1	-1	-1	$\sqrt{3}$	-3	-1.732051	ERR
0	1	-1	$\sqrt{3}$	0	0	90
1	1	-1	$\sqrt{3}$	1	0.577350	54.73561
1	-1	-1	$\sqrt{3}$	-1	-0.577350	125.26439
0	0	1	$\sqrt{6}$	1	0.408248	65.90518
0	1	1	$\sqrt{6}$	2	0.816497	35.26439
1	1	1	$\sqrt{6}$	3	1.224745	ERR
0	0	-1	$\sqrt{6}$	-1	-0.408248	114.09484
0	-1	-1	$\sqrt{6}$	-2	-0.816497	144.73561
-1	-1	-1	$\sqrt{6}$	-3	-1.224745	ERR
0	1	-1	$\sqrt{6}$	0	0	90
1	1	-1	$\sqrt{6}$	1	0.408248	65.90518
1	-1	-1	$\sqrt{6}$	-1	-0.408248	114.09484
0	0	1	2	1	0.5	60
0	1	1	2	2	1	0
1	1	1	2	3	1.5	ERR
0	0	-1	2	-1	-0.5	120
0	-1	-1	2	-2	-1	180
-1	-1	-1	2	-3	-1.5	ERR
0	1	-1	2	0	0	90
1	1	-1	2	1	0.5	60
1	-1	-1	2	-1	-0.5	120
0	0	1	3	1	0.333333	70.52878
0	1	1	3	2	0.666667	48.18969
1	1	1	3	3	1	0
0	0	-1	3	-1	-0.333333	109.47122
0	-1	-1	3	-2	-0.666667	131.81031
-1	-1	-1	3	-3	-1	180
0	1	-1	3	0	0	90
1	1	-1	3	1	0.333333	70.52878
1	-1	-1	3	-1	-0.333333	109.47122

Bibliography

1. Abdelrahman, Tarek S. and Trevor N. Mudge. "Parallel Branch and Bound Algorithms on Hypercube Multiprocessors." *The Third Conference on Hypercube Concurrent Computers and Applications, Volume II*. New York: ACM Press, 1988.
2. Air Force Electronic Warfare Center (AFEWC/SA). *Users/Operation Manual for the Improved Many-On-Many (IMOM)*. Contract F41621-86-C5005, Alexandria VA: ENTEK, Inc., June 1987.
3. Bahnij, Maj Robert B. *A Fighter Pilot's Intelligent Aide for Tactical Mission Planning*. MS thesis, AFIT/GCS/ENG/85D-1, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1985.
4. Barr, Avron and Edward A. Feigenbaum, editors. *The Handbook of Artificial Intelligence, Volume 1*. William Kaufmann, Inc., 1981.
5. Barr, Avron and Edward A. Feigenbaum, editors. *The Handbook of Artificial Intelligence, Volume 2*. William Kaufmann, Inc., 1982.
6. Beard, R. Andrew, et al. *AFIT/ENG Intel Hypercube Quick Reference Manual Version 2.1*. Technical Report, Wright-Patterson AFB, OH: Air Force Institute of Technology, 20 July 1992.
7. Beard, R. Andrew, et al. *Compendium of Parallel Programs for the Intel iPSC Computers, Volume I, Version 1.5*. Technical Report, Wright-Patterson AFB, OH: Air Force Institute of Technology, 1990.
8. Bharadwaj, Sudy, et al. "Issues in Dynamic Parallelization," *AI Expert*, *5* (February 1992).
9. Bradshaw, 2Lt Jeffrey S. *A Pilot's Planning Aid for Route Selection and Threat Analysis in a Tactical Environment*. MS thesis, AFIT/GCS/ENG/86D-11, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1986.
10. Brassard, Gilles and Paul Bratley. *Algorithmics: Theory and Practice*. Prentice-Hall, Inc., 1988.
11. Burgess, Lisa. "Services Embrace Commercial Specs for Mission Planning," *Military & Aerospace Electronics*, *3*:35–37 (1992).
12. Chandy, Mani K. and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1989.
13. Cobb, Richard H. and Harlan D. Mills. "Engineering Software Under Statistical Quality Control." *IEEE Software*. Los Alamitos, CA: IEEE Computer Socitey, November 1990.
14. Cohen, Paul R. and Edward A. Feigenbaum, editors. *The Handbook of Artificial Intelligence, Volume 3*. William Kaufmann, Inc., 1982.
15. Cormen, Thomas H., et al., editors. *Introduction to Algorithms*. McGraw-Hill, Inc., 1989.
16. DeCegama, Angel L. *Parallel Processing Architectures and VLSI Hardware, Volume 1*. Prentice-Hall, Inc., 1989.
17. Denton, Richard V. and Peter L. Froeberg. "Applications of Artificial Intelligence in Automated Route Planning." *Proceedings of SPIE - Applications of Artificial Intelligence, vol 485*. Bellingham, WA: SPIE, May 1984.
18. Duncan, Ralph. "A Survey of Parallel Computer Architectures." *Computer, vol. 23*. New York: IEEE Computer Society, February 1990.

19. Felten, Edward W. "Best-First Branch-and-Bound on a Hypercube." *The Third Conference on Hypercube Concurrent Computers and Applications, Volume II*. New York: ACM Press, 1988.

20. Flynn, Michael J. "Very High-Speed Computing Systems." *Proceedings of the IEEE, vol. 54*. New York: IEEE Press, December 1966.

21. Gardner, Capt Michael T. Personal interview. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, 25 August 1992.

22. Garmon, Capt Joel S. *Implementation and Analysis of NP-Complete Algorithms on a Distributed Memory Computer*. MS thesis, AFIT/GE/ENG/92-M, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 1992.

23. Hayes, John P. and Trevor N. Mudge. "Hypercube Supercomputers." *Proceedings of the IEEE, vol. 77*. New York: IEEE Press, December 1989.

24. Hennessy, John L. and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990.

25. Humphrey, Watts S. *Managing the Software Process*. Addison-Wesley, 1990.

26. Intel. *i860™ 64-bit Microprocessor Programmer's Reference Manual*. Intel, 1990.

27. Intel. *iPSC/2 and iPSC/860 User's Guide*. Intel, June 1990.

28. Intel. *Microprocessors, Volume II*. Intel, 1991.

29. Isensee, CPT Ernst K. *Multicriteria Network Routing of Tactical Aircraft in a Threat Radar Environment*. MS thesis, AFIT/GST/ENS/91M-01, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 1991.

30. Keirsey, David M., et al. "Multilevel Path Planning for Autonomous Vehicles." *Proceedings of SPIE - Applications of Artificial Intelligence, vol 485*. Bellingham, WA: SPIE, May 1984.

31. Kuan, Darwin T. "Terrain Map Knowledge Representation for Spatial Planning." *Proceedings of the First Conference of Artificial Intelligence Applications*. Denver, CO: IEEE Computer Society, December 1984.

32. Lamont, Gary B. Class handout distributed in CSCE 656, UNITY Program Syntax, revision 2. School of Engineering, Air Force Institute of Technology (AU),Wright-Patterson AFB OH, 1992.

33. Lamont, Gary B. and Jeffrey Simmers. *Predicate and Temporal Logic (an overview/a review), Version I*. Technical Report, Wright-Patterson AFB, OH: Air Force Institute of Technology, 16 May 1991.

34. Leithold, Louis. *The Calculus with Analytic Geometry* (3rd Edition). Harper & Row, 1976.

35. Lewis, Ted G. and Hesham El-Rewini. *Introduction to Parallel Computing*. Prentice-Hall, 1992.

36. Meystel, A. and E. Koch. "Computation Simulation of Autonomous Vehicle Navigation." *Proceedings of SPIE - Applications of Artificial Intelligence, vol 485*. Bellingham, WA: SPIE, May 1984.

37. Mitchell, Joseph S. B. "An Algorithmic Approach to Some Problems in Terrain Navigation," *Artificial Intelligence*, *37*:171–201 (1988).

38. Mitchell, Joseph S. B. and David M. Keirsey. "Planning Strategic Paths Through Variable Terrain Data." *Proceedings of SPIE - Applications of Artificial Intelligence, vol 485*. Bellingham, WA: SPIE, May 1984.

39. Nolen, Troy. "Parallel Processing for Problem Solving," *AI Expert*, 5 (February 1992).

40. Paker, Yakup. *Multi-microprocessor Systems*. Academic Press, 1983.

41. Papadimitriou, Christos H. "Shortest-Path Motion." *Foundations of Software Technology and Theoretical Computer Science: Sixth Conference*. New York: Springer-Verlag, December 1986.

42. Parodi, Alexandre M. "A Route Planning System for an Autonomous Vehicle." *Proceedings of the First Conference of Artificial Intelligence Applications*. Denver, CO: IEEE Computer Society, December 1984.

43. Pearl, Judea. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1984.

44. Pressman, Roger S. *Software Engineering: A Practitioner's Approach* (2nd Edition). McGraw-Hill, Inc., 1987.

45. Ragsdale, Susann, editor. *Parallel Programming*. McGraw-Hill, Inc., 1991.

46. Rich, Elaine and Kevin Knight. *Artificial Intelligence* (2nd Edition). McGraw-Hill, Inc., 1991.

47. Schwan, Karsten, et al. "Process and Workload Migration for a Parallel Branch-and-Bound Algorithm on a Hypercube Multicomputer." *The Third Conference on Hypercube Concurrent Computers and Applications, Volume II*. New York: ACM Press, 1988.

48. Spear, Capt Jon L. *Improvements to the AFIT Tactical Mission Planner (TMP)*. MS thesis, AFIT/GCS/ENG/88D-20, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1988.

49. Subcommittee on Investigations and Oversight; US House of Representatives Committee on Science, Space, and Technology. *Bugs in the Program: Problems in Federal Government Computer Software Development and Regulation*. Staff Study 101st Congress, First Session, Serial G. Washington: Government Printing Office, September 1989.

50. Work, Paul R., et al. "New Computational and Communications Results on the Intel iPSC/860 with the Intel System Software Release 3.3." *Proceedings of the Intel Supercomputers Users' Group*. Beaverton, OR: Intel Corporation, November 1991.

51. Zorpettes, Glenn. "The Power of Parallelism," *IEEE Spectrum* (September 1992).

Vita

James J. Grimm II was born in New Haven, Connecticut on March 2, 1961. He graduated from Eldorado High School, Las Vegas, Nevada in June, 1979. He attended Kansas State University and Brigham Young University before enlisting in the USAF. Upon completion of technical school at Kessler AFB. MS he was assigned to Ramstein Air Base, West Germany, as a computer programmer. He worked with multinational intelligence personnel in defining software requirements for Allied Forces Central Europe exercise directing staff as well as providing software supporting for other NATO intelligence systems. While in Germany he took classes from the University of Maryland, European Campus. He applied for and was accepted into the Airman's Education and Commissioning Program (AECP). He was assigned to Kansas State University were he completed a Bachelor of Science in Electrical Engineering degree in May 1988. At Kansas State University he joined Tau Beta Pi and Eta Kappa Nu honor societies. After attending Officer Training School he received a reserve commission in the USAF and was assigned to the Air Force Weapons Laboratory at Kirtland AFB in Albuquerque, New Mexico. He was responsible for testing and analyzing the effects electromagnetic pulse (EMP) have upon USAF weapon systems. He was also responsible for data processing activities in support of the B-1B Phase 3 and EC-135 System Level Electromagnetic Pulse Test programs. He remained in this position until his assignment to the School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, OH.

Permanent address: 916 Apache Ln
Las Vegas, NV 89110

REPORT DOCUMENTATION PAGE

December 1992 Master's Thesis

Solution to a Multicriteria Aircraft Routing Problem
Utilizing Parallel Search Techniques

James J. Grimm II, Captain, USAF

Air Force Institute of Technolcgy, WPAFB OH 45433-6583

AFIT/GCE/ENG/92D-04

Capt Kevin A. Cox
Electronic Systems Center (AFMC)
Mission Planning Systems (ESC/YV)
Hanscom AFB, MA 01731-5000

Approved for public release; distribution unlimited

Pilots select routes based on factors such as threats, fuel, time on target, distance, and refueling points. This is a time consuming task. This thesis presents the software engineering synthesis of a software tool, based on a parallelized A* search algorithm, to select routes. For simplicity only threats and distance are used.
A centralized open list is used with one processor managing the list while the other processors perform the node expansions. This decomposition results in a dynamically load balanced system. A number of parameters are changed to study their impact on the execution time. The use of a branch and bound technique and its impact on the execution time is studied. Other parameters examined are the size of the supercomputer and granularity of the algorithm. It is important to match the software granularity to the architecture to ensure maximum utilization of the supercomputer and minimize execution time. Tests were run on both an iPSC/2 and iPSC/860 to determine the effects of the architecture upon the execution time. In conjunction with execution time, the efficient usage of the parallel computer was also examined.

Parallel Processing, Search Algorithms, Heuristic Search, Aircraft Routing, Mission Planning, Software Engineering

246

UNCLASSIFIED UNCLASSIFIED UNCLASSIFIED UL