Developing an Effective and Efficient Real Time Strategy Agent for Use as a Computer Generated Force

THESIS

Kurt Weissgerber, Captain, USAF

AFIT/GCS/ENG/10-07

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/10-07

Developing an Effective and Efficient Real Time Strategy Agent for Use as a Computer Generated Force

THESIS

Presented to the Faculty
Department of Electrical and Computer Engineering
Graduate School of Engineering and Management
Air Force Institute of Technology
Air University
Air Education and Training Command
In Partial Fulfillment of the Requirements for the
Degree of Master of Science

Kurt Weissgerber, B.S.C.S.
Captain, USAF

March 2010

AFIT/GCS/ENG/10-07

Developing an Effective and Efficient Real Time Strategy Agent for Use as a Computer Generated Force

Kurt Weissgerber, B.S.C.S.

Captain, USAF

Approved:

[illegible] 22 Feb 2010

Lt Col Brett J. Borghetti, PhD (Chairman) date

[illegible] 22 FEB 2010

Dr. Gilbert L. Peterson (Member) date

Gary B. Lamont 22 FEB 2010

Dr. Gary B. Lamont (Member) date

AFIT/GCS/ENG/10-07

Abstract

Computer Generated Forces (CGF) are used to represent units or individuals in military training and constructive simulation. The use of CGF significantly reduces the time and money required for effective training. For CGF to be effective, they must behave as a human would in the same environment.

Real Time Strategy (RTS) games place players in control of a large force whose goal is to defeat the opponent. The military setting of RTS games makes them an excellent platform for the development and testing of CGF. While there has been significant research in RTS agent development, most of the developed agents are only able to exhibit good tactical behavior, lacking the ability to develop and execute overall strategies.

By analyzing prior games played by an opposing agent, an RTS agent can determine the opponent's strengths and weaknesses and develop a strategy which neutralizes the strengths and capitalizes on the weaknesses. It can then execute this strategy in an RTS game.

This research develops such an RTS agent called the Killer Bee Artificial Intelligence (KBAI). KBAI builds a classifier for an opposing RTS agent which allows it to predict game outcomes. It then takes this classifier, uses it to generate an effective counter-strategy, and executes the tactics required for the strategy. KBAI is both effective and efficient against four high-quality scripted agents: it wins 100% of the time, and it wins quickly. When compared to native artificial intelligence, KBAI has superior performance. It exhibits strategic behavior, as well as the tactics required to execute a developed strategy.

Acknowledgements

First off, thanks to my advisor, Lt Col Brett Borghetti. He allowed me to pursue a subject in which I had significant knowledge, interest and passion. Thank you Dr. Lamont, the most intelligent man I've met. Without his help and guidance, the quality of the search algorithms in this thesis would be lower, and that difference would be statistically significant. Thanks to Lt Col Borghetti's intern Ryan, who ran most of the RTS simulations required for this research but was not allowed to play any games. Finally, thanks to my fiance, who was extremely tolerant of the vast depths of nerdiness this thesis unleashed in me.

Kurt Weissgerber

Table of Contents

Page

List of Figures

List of Tables

Developing an Effective and Efficient Real Time Strategy Agent for Use as a Computer Generated Force

I. Introduction

Strategy requires thought; tactics requires observation.

Max Euwe

The goal of artificial intelligence (AI) is to make a computer perform the actions a human would normally perform [65]. In the military, AI can be used to model opposing forces for realistic training exercises or simulations using Computer Generated Forces (CGF). CGF take the place of humans, reducing the cost and resource requirements of a large-scale military exercise or simulation. Programming CGF so they act in a realistic manner is an open area of research for the military [27].

Real Time Strategy (RTS) games provide a good environment for testing and developing CGF [48]. In an RTS game, each player controls a theater level force, allocating resources and production to create an army and then using this army to destroy the opponent. Human players must think on both tactical and strategic levels to defeat their opponent. The RTS framework allows a single player to control an army consisting of potentially thousands of units, exactly the goal of CGF.

For an RTS agent to be useful as a CGF, it must replicate the behavior of a human [48]. It should gather information to develop a high level strategy and determine the tactics necessary to implement this strategy. The rest of this chapter discusses our approach to the development of a RTS agent which can replicate the behavior and planning aspects of a human player. First, Section 1.1 discusses the way military planners approach a wartime scenario and the parallel to decision making in RTS games. Next, Section 1.2 outlines a proposed method for strategy generation. In Section 1.3 the method of generating an agent from the classifier is outlined. Finally, we conclude with our hypothesis and an overview of the rest of the thesis.

1.1 Military Decision Making

The United States Department of Defense defines strategy as "A prudent idea or set of ideas for employing the instruments of national power in a synchronized and integrated fashion to achieve theater, national, and/or multinational objectives", while tactics are "the employment and ordered arrangement of forces in relation to each other" [44]. Strategy is the overall goal of a military operation, while tactics are the actions which must be taken to accomplish a strategy.

The Joint Planning Process (JPP) details how the US military develops plans to deal with national security threats [43]. In contingency planning, plans are developed to deal with potential threats, while crisis action planning provides direction for dealing with new time-critical situations. Planners develop, evaluate and select courses of action (COA) to support national strategic objectives. Once a COA is selected, an operational level plan is developed to accomplish the strategic objectives. Finally, specific operational objectives are given to small unit commanders, who design tactical plans consisting of the low-level actions and objectives which will attain them. The entire planning process flows from designing an overall strategy to developing the tactics used to accomplish it.

The military has three levels of planning: strategic, operational and tactical, based on the time-frame of the outcomes sought [34]. A strategic plan deals with long-term plans such as national or multinational objectives, risks of military use and the effectiveness of other instruments of power. The operational level of war is where campaigns and major operations are planned. The tactical level focuses on immediate goals [44]. As the time horizon of the operation increases, the level of planning required also increases. Military planners begin with strategic plans, then move onto operational plans designed to accomplish the goals of the strategic plans, and finally tactical plans which will accomplish the operational goals.

Similarly, an RTS agent must make make economic and military decisions at the tactical and operational level, while the strategic level is outside the scope of most RTS games. Economic decisions concern resource allocation among various production alternatives, while military decisions focus on unit placement and attack priorities. Strategies give high level goals, tactics are the low-level actions which are taken to accomplish them. Table 1 shows typical strategies and the tactical decisions that go with them in both the military and economic realm of an RTS game.

Table 1: Strategies and the associated tactical decisions faced by an agent in a typical RTS game.

Strategy	Military Tactical Decision
Impair enemy energy production	Destroy energy production buildings and construction units.
Achieve ground superiority	Destroy ground units and ground unit production buildings. Destroy defensive emplacements which can target ground units.
Increase infrastructure	Defend infrastructure and construction units. Engage enemy forces which seek to target infrastructure.
Strategy	**Economic Tactical Decision**
Impair enemy energy production	Build units which can be used to destroy energy production buildings.
Achieve ground superiority	Build ground units and ground unit production buildings. Build defensive emplacements to protect from ground attack.
Increase infrastructure	Build infrastructure and infrastructure construction units.

The commercial sector mostly relies on script-based RTS agents, which take the same actions no matter how many times they are run [76]. Such agents take a long time to implement and are predictable. A human player, when confronted with a new script, plays a few games to determine the strengths and weaknesses of the script. Then, he devises a strategy to neutralize the strengths and capitalize on the weaknesses. This strategy is refined to the point where tactics can be implemented to accomplish it, just as in the Joint Planning Process.

Many current dynamic RTS agents exhibit good tactical behavior but lack the ability for strategic planning. The quote opening this chapter talks about the different levels of reasoning required for strategy and tactics. Tactics are an easy problem for an agent: observe the state, take an action. However, strategy requires thought. Instead of simply observing the current state, the agent must take a more general view of the overall scenario and determine what over-arching goal should be pursued. While tactics are important, an overall strategy is key to winning an RTS game against a good opponent. An RTS agent should operate just as a human would by examining a script's past performance and determining an effective counter-strategy. Tactics should flow from this strategy.

The question is, how can a computer determine a counter-strategy?

1.2 Generating and Executing Strategies

Any dynamic RTS agent's goal is to gather information about an opponent and then apply this information to generate a counter-strategy. The problem is in determining from where this information comes.

An agent in an RTS could determine the most important aspects of a script by building a predictor for game outcomes. This predictor can be used to determine why a script wins or loses, so the agent can develop a counter-strategy. Such an agent would exhibit the behavior required to be a useful CGF, but how does an agent go about building a predictor?

A predictor can be implemented as a classifier. Given some current state, the classifier determines whether it belongs to the class of states which lead to a win or the class which lead to a loss. Many different methods of building classifiers are available, including principal component analysis, linear discriminant analysis, neural networks and decision trees [41].

When choosing a classification method, it is important to remember for what the classifier will be used. We want a classifier which can be used to develop and execute a strategic plan. Therefore, the classifier should contain elements which can be directly linked to actions in the RTS domain.

Any classifier requires features and samples [41]. In the RTS domain, a feature is some aspect of the environment which is observable to an agent. This could be the number of units, their position, the amount of resources, etc. A sample is just a vector of feature values at a specific point in a game.

Some classifiers take samples and transform them into a different space, while others create new features from the input features [55]. Often, these techniques can lead to good classification accuracies. In creating a CGF in the RTS domain, we want to keep features as close as possible to actions. Transforming feature values or creating new features distances them from actions. While it is easy to determine actions to influence the number of enemy tanks, it is much harder if this value has been changed to a three dimensional vector, or if the number of enemy tanks has been multiplied by two other feature values.

A nearest neighbor classifier can be used on original feature values; it simply takes a novel sample, determines the label of the closest sample in the generated classification model, and assigns the same label to the novel sample. The nearest neighbor classifier could be used to generate a useful predictor by feeding it appropriate features and samples and allowing it to generate a classifier.

To make the generated nearest neighbor classifier useful to an agent, it must go through two processes: feature reduction and exemplar selection. In feature reduction, the number of input features is significantly reduced. The selection of features used as input is tailored to link each feature directly to appropriate actions for an RTS agent, so reducing the number of features given to the agent reduces the action space. Feature reduction allows the agent to focus on actions which will have a significant impact on the game while ignoring actions which are irrelevant.

Exemplar selection reduces the number of samples in the generated classifier by only storing samples which are informative. Informative samples are those which help delineate the boundary between classes of samples [1]. Determining which samples are informative requires some knowledge of the problem domain. For an RTS agent, samples which lead to a win or loss a high percentage of the time are much more informative than samples which have a more even outcome split. The agent wants to attain states which will lead to a successful outcome a high percentage of the time, while avoiding states which lead to an unsuccessful outcome a high percentage of the time. The agent must control the game, avoiding states which do not have a high probability of success. By only keeping the informative samples, the ones which have a high probability of a specific outcome, the agent is forced to control the game.

Determining the best way to generate a classifier in a domain requires an understanding of its characteristics. Are the features dependent? Are there many different sets of features which can be used to generate a good classifier? Are some feature representations significantly superior to others? Answers to these questions must be found to determine the best algorithm to generate a classifier for the domain. This comes down to an analysis of the solution space of a specific problem, and the only way to determine the characteristics of a solution space is to evaluate the performance of different algorithms when applied to the domain. Once the characteristics have been determined, an algorithm can be designed and tailored which will perform well in this solution space.

After the solution space has been analyzed and an algorithm which can generate an appropriate nearest neighbor classifier has been created, an agent which takes this classifier and develops and executes a strategic plan is needed. This agent should be dynamic, able to develop and execute a different strategy based on any input classifier.

1.3 Creating an Agent from a Classifier

When generating the classifier for the agent, the features selected are linked directly to actions in the RTS domain. The classifier contains informative samples which have a high probability of leading to a good or bad outcome for a specific opponent. Our goal is to take this classifier and develop a counter-strategy to a given script-based agent.

Just like any agent, an appropriate action is determined by computing the current state. The feature set in the classifier tell the agent how to compute the current state. As a result of feature reduction, the agent has a much smaller action space to choose from, but still must choose which of these reduced actions to take.

The agent can do this by comparing the current state to the samples left in the classifier. The agent seeks to approach the successful outcome samples, while avoiding the unsuccessful outcome samples. Since the RTS domain is non-deterministic, that is, actions may not always have the expected effect on the state, the agent must predict the state which will be reached if it took a specific action. Then the agent determines which of the possible actions at any particular moment would lead to the best predicted state and take this action.

By using the classifier, the agent can determine an overall strategy which will be effective against a given scripted-agent. This strategy is articulated to the agent through the reduced feature set and the generalized feature values of the classifier. Appropriate use of the classifier guides the agent towards the opponent's centers of gravity, destroying the opponent as efficiently as possible.

1.4 Approach

An agent which can develop strategies and execute them with appropriate tactics should outperform an agent which can only execute good tactical behaviors. This leads to our hypothesis: an agent can use a nearest neighbor classifier to develop a strategy and the appropriate tactics for its execution, outperforming an agent which only makes tactical decisions. We assume the agent has perfect information about the environment, both when collecting game traces and when actually playing an RTS game.

The first step in testing this hypothesis is to generate a classifier. In any new problem domain, the specific characteristics of the problem domain must be determined by analyzing the results of various problem solving methods. To test a problem solving method, the problem must be formally declared so an algorithm to solve it can be generated.

We design the classification problem for the RTS domain as a search problem and formally declare the input and output. The problem declaration is tailored to make it useful to an RTS agent. There are two different families of search algorithm: deterministic and stochastic. A representative instance of each is developed and tested to solve the RTS classification problem. The performance of these two methods is used to illuminate the search landscape of the RTS classification problem. These landscape characteristics are used to create a new evolutionary algorithm which is tuned to the RTS domain. The effectiveness of the developed evolutionary algorithm is validated.

With the classification problem solved, we move to agent development. The Killer Bee Artificial Intelligence (KBAI) is designed to leverage the information contained in the input classifier. It is verified to be both effective and efficient when tested against four high-quality scripted agents in the RTS game Spring. KBAI's performance is superior to both a random choice agent and two full-featured RTS agents packaged with the Spring distribution. This performance validates the hypothesis, showing an agent which uses a strategy is superior to an agent using only tactics.

The rest of this thesis is organized as follows: Chapter II is a literature review of current related work. The history and impact of AI on games is discussed, with a specific focus on RTS games. The relationship between computer games and the military is explored, and then an overview of the classification and prediction problems is given. Finally, how to correctly formulate and solve a general search problem is outlined.

In Chapter III, the classification problem is formally declared. A representative example of each search family is developed and adapted to the classification problem. The performance of these algorithms is used to determine the characteristics of the search space.

An Evolutionary Algorithm is designed to take advantage of the discovered problem domain characteristics in Chapter IV. It is tested on two different data sets which validate its effectiveness for the RTS domain.

Chapter V discusses the development of the Killer Bee Artificial Intelligence (KBAI) from the nearest neighbor classifiers generated through a search algorithm. Four different scripted agents are created for testing. KBAI is tested against the scripted agents and its performance is compared to other existing RTS agents as well as an agent which takes random actions. KBAI is able to significantly outperform the other agents.

Finally, Chapter VI explains the impact of our research, both its contribution to the broad AI field as well as its application to the military.

II. Literature Review

This research develops an automated method to generate counter-strategies to scripted RTS agents. This chapter provides background information related to the conducted research. Section 2.1 gives a broad overview of current game-related Artificial Intelligence work including some goals and results. The current state of RTS agent development is examined, along with a discussion of the use of games in military applications. Section 2.2 explores the classification and feature selection problems. Finally, Section 2.3 discusses various search algorithm strategies along with ways of formally defining them.

2.1 Artificial Intelligence in Games

Artificial Intelligence has been used in games for many years and will continue to be used in the future [66]. AI agents can serve as tactical enemies and partners, support characters or strategic enemies. Agents could be used to control single units in a war simulation, or serve as opponents in a racing game. AI techniques have been used across many genres of computer games, from first person shooter games like Quake® or Doom® to roleplaying games like Baldur's Gate® to strategy games like Warcraft® [49]. "Computer games research is one of the important success stories of AI" [66].

The first AI work in games was done in chess [69]. In fact, probably the most famous game-playing computer is IBM's Deep Blue, which beat the international chess grand champion Gary Kasparov in a six game match in 1997 [40]. This result was a minor international sensation. For many people, it proved computers could perform game-playing tasks much better than humans.

Currently, there are many research efforts across all genres of games. Recent success has been seen in turn based strategy games like Poker [9], Go [12], and Checkers [67]. Non-player characters in Massively Multiplayer Online Games can be created using reinforcement learning [56]. Computer agents in Quake® can use imitation methods to learn effective behavior [60]. An agent in the classic arcade game Ms. Pac-man® can use a simple tree search method [64].

2.1.1 Artificial Intelligence Research in Real Time Strategy Games. RTS games provide a promising environment for many AI research topics. They have hundreds or thousands of objects, obvious multi-agent environments and imperfect information environments, all of which are open AI research fields [15].

RTS games are well-defined, well-tested environments. Results are easily measurable. Research can be tailored to focus on specific problem areas, like pathfinding, targeting, formations, building placement, etc. All these characteristics make RTS games an excellent platform for AI research [15].

Currently, most commercial RTS games use scripted agents for computer-controlled teams. Scripted agents are given a set of actions to perform, and the order in which they should be performed. Attacks are highly predictable: they occur at the same time with the same unit combination every game. While there may be some rule-based actions based on the game state, such as replacing buildings which are destroyed, scripted agents are static from one game to the next. To make them competitive with the human opponent, they are often given extra resources or the ability to build faster as a handicap [76].

Various reasons are given for the use of scripted agents. Some of the more important are [28]:

1. Lack of CPU resources available for AI use
2. Suspicion in development community of the effects of using non-deterministic methods
3. Lack of development time
4. Lack of understanding of AI techniques
5. Most effort is spent on improving game graphics

Many companies believe the amount of money which would be required to generate dynamic agents is not worth the cost. Most players spend very little time with computer controlled opponents, instead opting for the multiplayer version of the game where they can challenge human opponents online. Some researchers believe this is caused by the predictability of the gameplay experience. Perhaps if the gameplay experience was different every time, players would spend more time with computer opponents [28].

Dynamic agent development has been left to the academic community, and a significant amount of research has been done. Dynamic scripting is a type of reinforcement learning agent which generates a script for an agent before every game. The value, or reward, for each script is based on the outcome of the game in which it was used. Rewards are distributed among the actions which make up the script. When tested in WARGUS [75] against static opponents, Dynamic Scripting was able to learn a winning strategy in as few as sixty games against a non-rush opponent [73], but had trouble with a rush strategy, where the opponent attempts to build and attack as quickly as possible.

Dynamic scripting was compared to a Monte Carlo learning algorithm in Bos Wars [7], an open source RTS. While dynamic scripting generates an entire game script before the beginning of the game, the Monte Carlo algorithm split each game into episodes and gave rewards to actions continuously throughout each game. In most cases, the Monte Carlo agent outperformed the Dynamic Scripting approach. However, both had problems with a rush strategy [46].

A Monte Carlo strategy was also used on the open source platform ORTS [14]. This implementation generated and tested plans for an agent in a Capture the Flag Game, where the goal is to capture the opponent's flag while protecting your own. The value of each plan was based on three different measurements on the game state at the end of plan execution: material, exploration/visibility, and flag capture/safety. Plans were selected based on their value in past simulations, and value was updated at the end of every plan execution cycle. This approach had good success against a random opponent, but was again weak when compared to a rush opponent [17].

Case Based Reasoning (CBR) links a state with a goal and the actions which should be taken to achieve it. Multiple goals can be pursued at once, and each goal can have subgoals. CBR was used in the game WARGUS, an open source port of Blizzard Entertainment's Warcraft®. Cases were extracted from games played by "experts": players with significant experience in WARGUS. These experts play games, then explain what goals they were trying to achieve and what actions they took to achieve them. Each goal, the state of the game when it was pursued, and the actions that go with it become a case. Any goal that was a subgoal of another was annotated as such. An agent using the CBR framework was able to successfully defeat the built in WARGUS AI in eight out of nine games [58].

Various full-featured agents have been developed for the open source game Spring. KAI uses the concept of a threat map to guide unit production and attack decisions. Units are produced to counter the enemy's units and attack in areas where the enemy is weak [74]. The Spring agent AAI uses a simple learning algorithm to determine the correct units to build and target [68], while RAI is very similar to KAI but also handles imperfect information [63].

Almost all of these approaches have problems with rush strategies. While they perform well against some opponents, they are vulnerable to others, showing their failure to adapt their strategy to new opponents. The full featured agents fail to exhibit strategic behavior. Actions are taken based on the current state, but are not linked to an overall strategic goal.

2.1.2 Computer Games and the Military. Today, military simulations are a critical aspect of military training and decision making. Military simulations started as a way of predicting the results of combat scenarios at a much smaller cost than a full unit exercise. Computer simulations save on the significant costs of deploying numerous people to conduct an exercise. If the simulation can determine the same result with some degree of accuracy, the cost savings are considerable. Military "modeling and simulation are considered essential to transformation, the remaking of the Armed Forces for the new realities of the 21st century. These tools present a powerful way for military leadership to visualize the future and assess the needs for the new forces" [35].

Computer games have much in common with military simulations. While their goals are different, the end results are often similar. The qualities that make computer games entertaining to players are often the same qualities that provide realistic training environments. The introduction of the World Wide Web led to increased opportunities for multiplayer and cooperative gaming, something which helps in military training. Additionally, significant investment in the commercial gaming industry has led to advances in graphics, simulation reality and cooperative gaming. Today the military is experiencing an "emergence of a culture that accepts computer games as powerful tools for learning, socialization, and training" [35].

Effective strategic level military training requires a complete battlespace of thousands of participants. It can be very costly to conduct a live exercise with this many people [48]. Computer Generated Forces (CGF) can be used to simulate many of these participants, bringing the scenario cost down significantly.

In addition to serving as human entities in military simulations, CGF can be used to test most robotic behavior [53]. The first phase of Demo III, a project to develop autonomous navigation capabilities for military robots, was to test various pathfinding and visual recognition algorithms in a simulator [70]. These tests were conducted using ModSAF, a simulator which provides CGF.

However, development in CGF systems is costly. It requires a substantial investment of both time and money to learn the system, as well as the doctrine which supports it. Most of the current funding is in building and fielding new CGF systems and capabilities, not in research [49].

Computer games provide a good environment for CGF research for many reasons. Many games have a well-defined interface for connecting external agent software. Most come with some sort of editor allowing the user to define the environment in which he/she plays. Finally, the computer game environment is familiar and attractive to many computer science and artificial intelligence researchers [48].

Since computer games are a good platform for AI research, ways to develop agents to perform effectively in these domains should be explored. An effective agent can be developed through classification and search techniques, which will be discussed in the next two sections.

2.2 Classification and Prediction

A classifier is a system created from labeled data which can then be used to classify new data. In a more general sense, building a classifier is the process of learning a set of general rules from specific instances. These rules can be used to assign new samples to classes. Classification falls into the realm of supervised machine learning [47].

A classifier is often generated from an initial dataset, called the training set. This training set is a series of samples of feature values, where a feature is some measurable aspect of a specific problem domain. Each sample has values for all the features and is labeled as to what class in the problem domain it came from.

There are numerous methods of generating classifiers. Logic based algorithms construct decision trees. New data can be classified by following the decision tree from the root to a leaf node and classifying appropriately. Perceptron-based techniques learn weights for each feature value, and then compute a function value for all the training data. Instances are classified based on this function value. Statistical learning algorithms generate probabilities that a sample belongs to a specific class, instead of a simple classification. Common examples of statistical techniques are Linear discriminant analysis [29] and Bayesian networks, which were first used in a machine learning context in 1987 [16].

The family of instance based learning algorithms are the most useful to our research and the next section explores them in depth.

2.2.1 Instance Based Learning Algorithms. Instance based learning (IBL) algorithms assume that similar samples have similar classifications. They derive from the k-Nearest Neighbor (k-NN) classifier, which classifies a sample based on the k closest samples to it in the classifier. IBL algorithms represent each class as a set of exemplars, where each exemplar may be an instance of the class or a more generalized abstraction [55].

Two basic exemplar models are proximity and best examples. A proximity model stores all the training instances with no abstraction, so each new instance is classified based on its proximity to all the samples in the training data. Best example models only store the typical instances of each concept [71]. Best example models can greatly reduce the size of the generated classifier.

One of the methods of creating a best example model from the training set is the K-means clustering algorithm. K-means is a two step algorithm which takes N samples and assigns them to K clusters. Each cluster is represented by a vector over all the features called its mean. K-means is a two step process: in the assignment step, each data point $n \in N$ is assigned to the nearest mean. In the update step, the means are adjusted to match the sample means of all the data points which were assigned to them. This process repeats until the change in the clusters approaches zero [54].

2.2.2 Feature Selection. A feature is an observable aspect of some environment. The general problem of feature selection is, given a set of candidate features, select a subset that performs the best under some classification system [42]. Feature selection takes some number of features as input and then outputs a subset of them.

Feature selection is performed to deal with two problems. First, features chosen should be “relevant” in the environment. A relevant feature in an RTS game is one whose value has a large impact on the game outcome. In other words, if a features value is changed slightly, then the game outcome is significantly different [11].

Selecting irrelevant features leads to major problems for classification systems. The presence of irrelevant features in a nearest neighbor classification method leads to an exponential increase in the number of training cases needed to develop an effective classifier [51].

Exhaustively searching all the possible feature combinations of a domain leads to complexity which is exponential in the number of features in the input set [22]. As a result, various methods are used to reduce the search space. Three broad categories of feature selection methods are embedded, filter and wrapper [11]. They differ in their method of evaluating feature subsets.

All three methods are given a set of training data, which consists of samples over all the features, with all samples labeled as to the class to which they belong. Some classification method is used to classify all the samples based on the feature subset picked.

Filter methods rank features according to some correlation measure. The features to use are selected before examining the actual classification accuracy based on these features; the classification method is adapted to the chosen feature subset. Wrapper methods are just the opposite; they select feature subsets based on their usefulness in a given classifier. The feature subset is adapted to the classifier. Embedded methods perform feature selection and classification simultaneously [33].

The Bhattacharrya coefficient is a method of determining the overlap between feature values for two classes. It looks at histograms of the training data and determines the separability of features. The Bhattacharyya coefficient can be used in filter feature selection methods to order features [2].

2.2.3 Measuring Classifier Performance. Classifier performance is generally measured by its prediction accuracy on a set of data. During the learning step, where classifier performance is being adjusted, this prediction accuracy is on the training set. However, when comparing two different classifiers, fairly performing a comparison is not so simple [13].

Optimizing a classifier to fit the training data creates the risk that the classifier will fit to the noise in the data by learning various peculiarities of the training data instead of a general overall rule. This problem is called "overfitting" [25]. It leads to good prediction accuracies on the training set, but lower accuracies on novel data.

Various methods of dealing with the overfitting problem have been proposed. One of the most common is k-fold cross validation. In k-fold cross validation, the input data is split into k equal sized groups, or "folds". The classifier is trained on $k - 1$ folds, then tested on the fold held out. This is repeated for every fold, and then the observed prediction accuracies are averaged. Common methods are $k = 3$, or $k = m$, where m is the number of samples in the dataset. The latter method is referred to as "leave one out". K-fold cross validation has been shown more accurate than other classifier testing methods [10].

All classifiers, especially the instance based ones, can be generated by search algorithms, so various search algorithms are discussed in the next section.

2.3 Search Problems

Search is one of the core areas of artificial intelligence [30]. Search is a method of generating and evaluating solutions to a given problem. When developing an algorithm to solve a problem, it is important to analyze both the problem as well as possible algorithms to solve it.

2.3.1 Defining the Problem. Defining the problem requires two things: an informal explanation and then a formal description. To make problem definition a useful process, it should be clear and precise.

A problem definition has three parts: the representation, the objective, and the evaluation function. The representation defines what a solution to the problem looks like, including all the variables and their values. The objective is the goal of the problem, including any constraints which must be satisfied. Evaluation functions measure the quality of a solution, or how well it satisfies the objectives of a problem. Once all three parts of the problem definition have been defined, the problem can be solved through search [57].

2.3.2 Choosing and Defining the Algorithm.

The two general categories of search algorithm are deterministic and stochastic. The difference between the two classes lies in how they generate the next state for a search problem.

A specific deterministic algorithm, when given a starting state s, always generates the same next state s' no matter how many times the algorithm is run. A specific stochastic algorithm, on the other hand, incorporates some randomness, so the next state s' generated from s may be different in different runs of the algorithm [50].

All search algorithms are subject to the No Free Lunch Theorem, which states "all algorithms that search for an extremum of a cost function perform exactly the same, when averaged over all possible cost functions" [80]. In other words, there is no one search algorithm that is best on all problems. It is important to determine the characteristics of a specific problem domain, and then determine the appropriate algorithm to take advantage of these characteristics [61].

2.3.3 Deterministic Search Algorithms.

Many deterministic search strategies are optimal; they are guaranteed to find a solution to the given problem which is no worse than any other solution. In deterministic search, the only way to do this is to explore every solution to the problem, either explicitly or implicitly. Examples of this type of search are Depth First Search with Backtracking, Breadth First Search, A* Search and Z* Search. Explanations of all these basic deterministic search algorithms can be found in [19, 52].

Deterministic searches are often guided by heuristics, estimates of the fitness of a solution to a problem. When constructing a solution to a problem one piece at a time, the next piece chosen for inclusion may be selected by its expected value, as computed by some heuristic. Traditional heuristic guided searches are all of the best-first algorithms. Choosing a proper heuristic can serve to guide the search towards profitable regions of the space and away from unprofitable regions [59].

A heuristic is said to be admissible if the estimated fitness value for a solution from the heuristic is greater than or equal to its actual value[1]. A best-first search strategy used with an admissible heuristic can guarantee that an optimal solution will be found. If the heuristic is not admissible, then any solution found can not be guaranteed optimal. The search algorithm may be guided towards profitable directions of the search space, but if the entire search space is not explored, then any answer found can not be said to be optimal [59].

Optimal deterministic search strategies are excellent when dealing with small problem instances, or problems where an admissible heuristic exists. They are able to search the entire space and find the best answer. However, they run into problems when dealing with large search landscapes. In these problems, there is not enough time to search the entire space. While insight about the problem domain may lead to an algorithm which can implicitly search the space, as in Dijkstra's shortest path algorithm [26], in most cases the exact problem is too large to be solved through a full enumeration of all possible solutions. Instead, the problem domain is approximated. The problem is modeled as a smaller problem, and then an optimal deterministic algorithm is used to solve this smaller problem instance. The solution is used in the larger problem.

In some cases, the optimal solution to the approximated problem is good enough, but in most cases a near-optimal solution to the full problem is superior [57]. Determining a near optimal solution may require a stochastic search algorithm.

[1]This is in the case of a maximization problem. If the objective is to minimize some function, then the estimated fitness value must be less than or equal to its actual value.

2.3.4 Stochastic Search Algorithms. Stochastic search algorithms have some probabilistic method of determining the next state in a search. Some well known stochastic search algorithms are simulated annealing, tabu search, evolutionary algorithms and particle swarm optimization. Stochastic search algorithms are often called local search algorithms because not all states in the solution space can be reached from the current solution. Generally, the amount of change in the current solution is limited to some neighborhood, to prevent the algorithm from jumping wildly around the search landscape [39].

2.3.5 Evolutionary Algorithms. Evolutionary algorithms are a class of stochastic algorithms based on Darwin's theory of natural selection [20]. Evolutionary algorithms can be split into three types: genetic algorithms, evolutionary programming(not used in this research), and evolution strategies.

Genetic algorithms (GAs) were introduced by Holland [36–38], and subsequently studied by numerous others, including De Jong [23, 24] and Goldberg [31, 32]. GAs work on fixed-length bitstrings which map to solutions to a specific problem. Solutions are combined and mutated at each generation to create better solutions.

Evolution Strategies (ES) were jointly developed by Rechenberg and Schwefel [8]. Problem variables are real valued, making mutation and recombination somewhat more difficult than in genetic algorithms.

Both GAs and ESs balance exploration with exploitation in a search problem through the selective pressure imposed by the algorithm. Selective pressure is influenced by the methods of selection and recombination in a specific algorithm. Algorithms with purely elitist selection and recombination operators are biased towards exploitation, while more stochastic methods can increase exploration of the search space. Correctly tuning selective pressure is a major component in the success of any evolutionary algorithm [18].

2.4 Summary of Related Work

In this section, background info relating to the work discussed later in this thesis was presented. The use of Artificial Intelligence in games was summarized, along with an in-depth discussion of the current state-of-the-field of RTS games as well as the use of computer games for military applications. The classification problem and various methods of feature selection were explored. Finally, general search problems were examined, including problem definitions and different families of search strategies.

A significant amount of research has been done in games. However, developed learning based RTS agents fail to adapt to multiple different strategies, especially rush techniques. Full featured agents fail to exhibit strategic thinking.

Classification algorithms can be used to predict the outcome of a game, and there are many different developed algorithms. When choosing a classification algorithm to solve a specific problem, an analysis of the problem domain informs the selection of a specific search strategy. The best way to determine the characteristics of a problem domain is to examine the results of a specific algorithm's performance. As the problem domain becomes more understood, a specific algorithm can be developed and tailored to solve the specific problem.

The next step in our research is to determine the characteristics of the RTS domain through application of different search methods. In the next chapter, the RTS Prediction Problem (RTSPP) is declared. The RTSPP is tailored to make it useful in agent development. Two different methods of feature selection and classification are developed to solve the RTSPP, one deterministic, one stochastic. Their performance illuminates the characteristics of the search landscape, allowing us to make a selection of appropriate search algorithm to solve the RTSPP.

III. Building an RTS Classifier

Real Time Strategy games have large decision spaces, with hundreds or thousands of units. The size of the action space is a challenge for reinforcement learning type approaches: it may take many iterations to completely explore all the possible actions to find a good strategy.

To deal with this problem, it is useful to be able to determine which actions in an RTS are not useful to a learning algorithm and should not be examined. If the action space can be cut sufficiently, then the agent can effectively reason over the remaining actions.

Distinguishing the important actions of an RTS game from the unimportant ones significantly reduces the action space. One method of approaching this problem is to examine game traces from an RTS game to see which features can be used to separate winning and losing strategies. Samples can be separated by generating a classifier for the game traces and testing its performance. Game samples representative of winning and losing states can be generated using an Instance Based Classification Algorithm to generate a nearest neighbor classification model. This nearest neighbor model can then be used by an agent to determine appropriate actions to take based on the current state. If these features can be linked to actions in the domain, then the reduced feature set can be used to reduce the action space.

Any search problem can be solved using two different types of search: deterministic and stochastic. A deterministic algorithm is not probabilistic. The next search state can be determined from the current search state and the chosen search algorithm. To find an optimal solution, the entire solution space must be searched either explicitly or implicitly using a global search algorithm. This means that the problem domain dimensions must be reduced by decreasing the size of the solution space so it can be searched in a reasonable amount of time. Relaxing the problem domain yields an optimal solution to a smaller problem.

In a stochastic search, the search algorithm is probabilistic. The next state of a search algorithm is not always the same. Instead, the search is guided towards profitable areas using some heuristic. A stochastic algorithm does not search the entire space, instead seeking to exploit characteristics of the problem domain to find good solutions. Stochastic search algorithms require the assumption that the search is allowed to run forever to guarantee optimality. This is clearly unrealistic. However, the solution yielded by a stochastic algorithm is a solution in the original problem domain which may be near optimal.

In some problem domains a near optimal solution to the original problem is better than an optimal solution to a problem where some constraints have been relaxed to reduce the solution space. In others, the converse is true. One way to determine this is to test both approaches on the problem domain. To do this, the problem domain must be explicitly defined. Next, a specific search algorithm can be developed and tailored to the problem. In this chapter, both a deterministic and stochastic search algorithm are developed to solve the RTS classification problem. They are tested on a data set from an RTS application, and their performance is compared. The discovered characteristics of the problem domain are used to select one family for further development.

The rest of this chapter is organized as follows: Section 3.1 discusses the classification problem, giving both the intuition and a formal definition. In Section 3.2, a deterministic solution to the problem is developed, followed by a stochastic solution in Section 3.3. An experimental methodology to compare the two solutions is created in Section 3.4. Results of experiments are presented and analyzed in Section 3.5.

3.1 Problem Description

The classification problem can be formulated as a basic search problem: the Real Time Strategy Prediction Problem (RTSPP). Any search problem can be defined by its input, output, and fitness function.

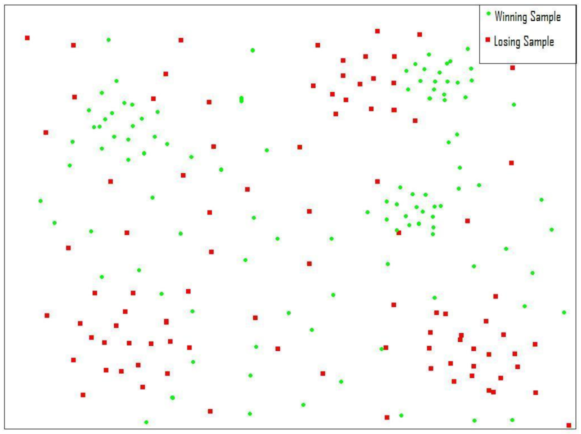
3.1.1 Problem Definition. The input to the RTSPP is a set of game traces from RTS games. Each game trace consists of samples taken at constant intervals of around five seconds. Each sample contains the value of possible features which an agent can observe or derive. In the RTS domain, observed features include the number and type of units, the amount of energy or fuel and the rate at which energy and fuel are collected or used. Derived features include the rate of change of any of the observed features across some time interval. Each sample is labeled as to whether it came from a game which was won or lost by the first player.

All features are defined as the difference between player one's value and player two's value. For example, if at some point in a game player one has two infantry units and player two has three, then the value of the infantry unit feature is negative one. Expressing features as a difference halves the space required to store game traces.

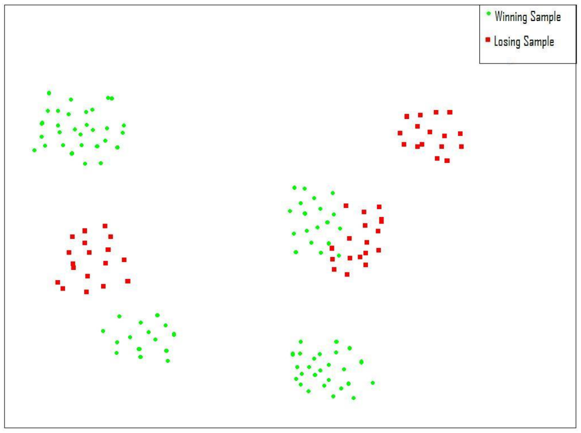
Much of the overlap in state values is caused by noise in the data. Some feature values are unimportant: if one agent is pursuing a strategy which relies on infantry units, while the other is building aircraft, then the number of tanks doesn't have any useful information for the classifier. Removing irrelevant features reduces this noise and makes the space more manageable. Figures 1(a) and 1(b) depict this process on hypothetical game data. The final step is to generalize over all the original data samples. Instead of keeping all of them, which would require significant space, the clusters can be reduced to a single sample representing each cluster. These exemplative samples are referred to as centers.

A solution to the RTSPP is a classifier: a set of features, a set of winning centers and a set of losing centers. The set of features show how to compute the current state. Each center in the set of winning centers is an example of a probable winning state, while each losing center is a probable losing state.

The classifier can then be used to predict the outcome of a game based on the current state. During a game, the values for the features in the solution are measured. Then, the distance to each center in the sets of centers is measured. The closest center is determined. If this center is a winning center, then the game state is predicted to



(a) Game samples before feature reduction



(b) Game samples after feature reduction

Figure 1: (a), (b)

result in a win. If it is a losing center, then the game state results in a loss. There are no ties in our research: all games are run until one player wins outright.

The quality of a solution to the RTSPP can be measured by testing its classification performance. Classification performance is measured as a percentage of correct predictions to total samples, as discussed in Section 2.2.3.

3.1.2 Formal Problem Definition. To remove any possible confusion, the RTSPP is now formally defined: first, the input to the problem is a set of $n \times m$ data, where n is the number of features and m is the number of samples. There is a set F of features and a set S of samples.

The output of the problem is a set of features F', where $F' \subseteq F$, and a set of centers C, where the winning centers are C_w and the losing centers C_l, so $C_w \cup C_l = C$ and $C_w \cap C_l = \oslash$.

The fitness of a solution can be determined by using it to classify all the samples in S. The function $dist(s, c)$ returns the distance from a sample s to a center c, so the value of a prediction function $P(s)$ which returns one for a win and zero for a loss is:

$$P(s) = \begin{cases} 1 & min_{c \in C}(dist(s,c)) \cap C_w = c \\ 0 & min_{c \in C}(dist(s,c)) \cap C_w = \oslash \end{cases}$$

The equation finds the center in the set of centers which is closest to the sample s. If this center is in C_w, then the sample is predicted to result in a win. If it is not in C_w, then the sample is predicted to result in a loss.

Next, a function which determines the accuracy of a prediction is needed. The function $g(s)$ returns one if the prediction is correct, zero if it is not. For ease of notation, the true label of sample s is denoted by $P^*(s)$. $g(s)$ is formally defined as:

$$g(s) = \begin{cases} 1 & P(s) = P^*(s) \\ 0 & P(s) \neq P^*(s) \end{cases}$$

If the predicted value matches the actual value, then the prediction was correct so $g(s)$ is one. Otherwise, the prediction is incorrect and $g(s)$ is zero.

Total fitness $G(S)$ is the sum of $g(s)$ over all samples $s \in S$ divided by the number of samples:

$$G(S) = \frac{\sum_{i=1}^{m} g(s_i)}{m} \tag{1}$$

The objective of the RTSPP is to find the F' and C for which $G(S)$ is maximum.

3.1.3 Solution Space Analysis.

The final step in problem definition is an analysis of the number of possible solutions to the problem. This information is important because it determines the difficulty of the search.

In the RTSPP, there are two components to a solution: the features in the set F' and the centers in C. The number of possible feature subsets is $O(n!) = O(n^n)$ [45].

Center solution space analysis is more complicated. If centers are restricted to being a sample $s \in S$, then the number of possible centers is $O(m!) = O(m^m)$.

However, if center values are not restricted, then the solution space is much larger. If each feature is split into 1,000 possible values, then there are $O(1000^n)$ possible values for a single center. There is no reason to have more than m centers: more than m centers would imply we could create more information than is present in the data. The solution space for real valued centers is $O(1000^n \times m)$.

Combining the two solution spaces leads to a total solution space of $O(n^n \times 1000^n \times m)$.

One of the easiest reductions to the problem domain is to reduce the number of features in F' and centers in C. The overall goal of the RTSPP is to reduce the decision space for an agent. While keeping all the features/samples in a solution may lead to high fitness values, it does not accomplish this goal. Accordingly, the size of F' is limited to some constant j and the size of C is limited to some constant k, leading to these two formal constraints on a solution:

$$|F'| \leq j \tag{2}$$

$$|C| \leq k \tag{3}$$

The two constraints significantly reduce the size of the solution space. The feature selection portion is now $O(n^k)$. The center portion is $O(1000^j \times j)$ for real valued centers and $O(m^j)$ when centers are subject to $C \subset S$. Total solution space size is $O(n^k \times 1000^j \times j)$ or $O(n^k \times m^j)$.

With the reduction based on the constraints, the solution space is polynomial in the number of features and samples in the input data.

In this section, the RTSPP is developed and analyzed. With the problem domain completely defined, a search algorithm to solve it can be designed.

3.2 A Deterministic Approach

In the RTSPP, the goal is to find features which separate the data into two distinct classes. A deterministic solution searches the entire space to find the globally optimal solution to the problem. In a deterministic global search algorithm, the size of the solution space has a huge effect on algorithm performance and developing a heuristic to prune the search space is critical [59].

3.2.1 Algorithm Domain Requirements Specification. A deterministic algorithm consists of the characteristics (adapted from [50]):

1. Variables: X is the set of possible solutions to the problem, Di is the set of candidate solutions still available in the search, Dp is the current solution and Do is the best solution found so far.
2. State: Dp is the current partial solution. The current solution is always a subset of the possible solutions to the problem, $Dp \subseteq X$.
3. Set of Candidates (Di): The set of solutions which have not yet been explored. This is the possible solutions in X not yet in Dp, or $Di = X - Dp$.
4. Set of Constraints: Problem specific conditions on a solution to the problem, η
5. Operations:
    (a) Next State Generator: A function $N(x)$ which maps a state $x \in Di$ to the possible next states $Z \subset Di$.
    (b) Selection Function: A function which takes the possible next states and determines the best next state, according to some heuristic: $S(Z)$. Returns the selected state, $z \in Z$.
    (c) Feasibility Function: Determines whether the current state meets the constraints of the problem, $F(\eta, Dp)$. Returns a truth value.

(d) Objective Function: Determines the fitness of a solution, $G(x \in X)$. The objective is to optimize this function, which could be to find a minimum or maximum value depending on the problem domain.

(e) Solution Function: Determines whether the current solution is a possible solution to the problem, $I(Dp)$. Returns a truth value. If it is a possible solution, then the fitness of this solution, $G(Dp)$, is compared to the fitness of the best solution found so far, $G(Do)$. If better, Dp replaces the current solution in Do.

3.2.2 Algorithm Domain Function Specification. Deterministic algorithms operate in four phases: Initialization, Selection, Feasibility and Evaluation. The integration of the function specification with the domain requirements leads to Algorithm 1.

Algorithm 1 Deterministic Algorithm Specification

```
{Initialize}
$Dp = Do = \oslash$
$Di = X$
while $Di \neq \oslash$ do
  {Selection}
  $Z = N(Dp)$
  $Dp = S(Z)$
  {Feasibility}
  if $!F(\eta, Dp)$ then
    $Di = Di - Dp$
    Return to Selection
  end if
  {Evaluation}
  if $I(Dp)$ AND $G(Dp) > G(Di)$ then
    $Do = Dp$
  end if
end while
```

The current state and the best solution are initialized to the empty set. The set of candidates contains all the possible solutions to the problem. Each step of the search computes the possible next candidates, chooses from these set of candidates and compares to the best solution found so far. The algorithm ends when the set of candidates is empty, and the best solution is contained in Do.

3.2.3 Problem/Algorithm Domain Integration. The next step in algorithm design is to populate the algorithm domain characteristics with specific problem domain information. In the RTSPP, a possible solution is a set of features and centers, so the set of possible solutions is all the possible feature and center combinations, or $X = F \cup S$. A state is a set of features and centers in the current partial solution, $Dp = F' + C$. The set of constraints are listed in Equations 2 and 3, $\eta = \{|F'| \leq j, |C| \leq k\}$

At each step of the algorithm, a feature or center is added to the partial solution. The Neighborhood Function finds all the features/centers not in the solution and provide them as opportunities for selection, $Di = X - Dp$ or $Di = (F \cup S) - (F' + C)$. Since features are selected before centers, the set of candidates Di must be split into two portions, one of features and the other of centers. The feature portion is Di^f and the center portion is Di^c.

Now, based on the problem/algorithm domain integration details, a specific deterministic algorithm can be selected. First, the constraints put an automatic limitation on the size of a solution. Second, the objective of a deterministic global search algorithm is to explore all the possible solutions to the problem. Finally, there is no specific heuristic for estimating the value of a solution without computing the total fitness, and the fitness function is by definition not admissible (see Section 2.3.3), so there is no point in using one of the Best First algorithms. As a result, a good choice for an algorithm is a Depth First Search with Backtracking (DFS-BT). A DFS-BT has time complexity of $O(b^m)$, where b is the branching factor of the search, the maximum number of states which can be generated from another state. m is the maximum level at which a solution could be found. However, the DFS-BT has space complexity of only $O(b \times m)$, which is a significant gain over a breadth first search strategy, which has space complexity of $O(b^m)$ because it must keep track of its current location in the search tree [65].

With the selection of a specific deterministic algorithm, an initial program specification can be constructed. Additionally, since this is a deterministic algorithm which will enumerate all the possible solutions, a level variable l is introduced to keep track of the set of candidates at each level, so Di_l^f and Di_l^c represents the available features and centers at level l. Dp_l^f represents the current feature set at level l, and Dp_l^c is the current center set at level l.

3.2.4 Program Specification. In the algorithm domain, a new feature/center is added to a partial solution at every step. Each feasible possible solution is tested, and if at some point the fitness of the current solution is greater than the best found so far in the search, then the current solution is saved as the new best solution. The search is allowed to run to a depth of $l = j + k$. The backtracking portion of the DFS-BT algorithm means a set of candidates must be stored for each level. The algorithm domain constructs integrate with the function specification to form the high level program specification in Algorithm 2.

This algorithm, in the worst case, is searching every possible solution in the space, which is $O(|F|^k \times |S|^j)$. Additionally, computing the fitness function takes time of complexity $O(k \times |S|^2)$. To compute it, the closest center to every sample must be computed, and the fitness function $G()$ is $O(k \times |S|)$ for a single sample. This leads to total complexity of $O(k \times |F|^k \times |S|^{j+2})$, which is too large to search completely. A method of reducing the search space must be devised.

3.2.5 Program Refinement. One of the easiest ways to reduce solution space size is to determine a way to pair features with centers. If at each step a triple could be selected which consisted of one feature, one winning center and one losing center, the number of combinations would be greatly reduced. This requires a means of determining good feature values when features are selected.

Algorithm 2 DFS-BT

```
$Dp_0^f = Dp_0^c = Do = \oslash$
$l = 0$
$Di_l^f = F$
$Di_l^c = S$
**while** $Di_0^f \neq \oslash$ OR $Dp_0^c \neq \oslash$ **do**
  **if** $l = j + k$ **then**
    Backtrack
  **else if** $Di_l^f \neq \oslash$ **then**
    Select $x \in Di_l^f$
    $Dp_l^f = Dp_l^f \cup x$
    **if** $G(Dp_l) > G(Do)$ **then**
      $Do = Dp_l$
    **end if**
    $Di_{l+1}^f = Di_l^f - x$
    $l = l + 1$
  **else if** $Di_l^c \neq \oslash$ **then**
    Select $z \in Di_l^c$
    $Dp_l^c = Dp_l^c \cup z$
    **if** $G(Dp_l) > G(Do)$ **then**
      $Do = Dp_l$
    **end if**
    $Di_{l+1}^c = Di_l^c - x$
    $l = l + 1$
  **else**
    Backtrack
  **end if**
**end while**
```

One way of determining this involves the use of the Bhattacharyya Coefficient (BC) [2]. The BC can be used to determine the separability of two data sets. It computes the separability of two classes of data, based on a histogram of the data. Values for the coefficient for a feature are between 0 and 1, where values close to zero show the feature is very separable between the two classes, while values close to one show the feature is not very separable for these two classes. Therefore, the BC heuristic can be used to choose the feature with the most separability at each step. Each feature can be paired with a sample in its histogram. The BC finds data distributions that are as far apart as possible; centers should be chosen that best generalize each distribution. Therefore, the median sample of the winning/losing distribution is chosen as the center for each feature.

The BC is calculated by taking a histogram of all the data and determining the probability of a sample falling in a bin for both classes. The two probabilities for each bin are multiplied together and summed over the entire histogram. Formally, this is:

$$BC = \sum_{i=1}^{I} P(W_i) \times P(L_i)$$

I is the number of bins in the histogram, W_i is the set of winning samples, L_i is the set of losing samples and $P()$ is the probability of the samples being in the bin. Figure 2 is a visualization of this idea. The two curves are distributions over the winning and losing samples. The BC is a number between one and zero, expressing the amount of "overlap" of the two distributions; zero represents no overlap, while one represents complete overlap. On this graph, it is the space bounded by both curves. To pair a feature with a winning and losing center, we take the sample at the median of the respective distributions, symbolized by the lines W_i and L_i. We have expressed the win/loss samples for feature F_i as gaussian distributions, but the BC works for any type of distribution.

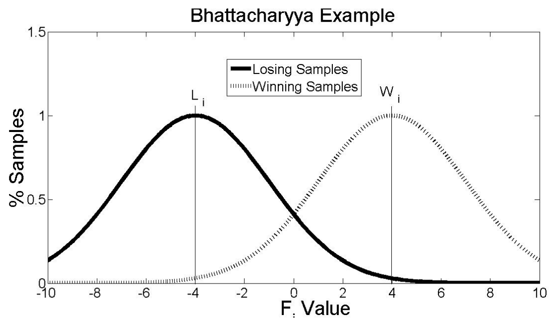


Figure 2: A visualization of the Bhattacharyya coefficient (BC) on feature F_i.

The BC pairs each feature with two centers (one winning, one losing), so at each step of the DFS-BT the set of candidates contains a set of triples, each containing one feature and two centers. Because the BC drives a particular choice of center for each feature, the maximum size of the set of candidates is $|F|$.

The BC is not an admissible heuristic because the optimization function (percent classified correctly) is not directly related to the BC. However, if the triple with the lowest BC is chosen at each step, it should drive the greatest improvement in classification accuracy because the overlap between the winning/losing sets is as small as possible. If the feature with the lowest BC remaining is selected and it does not improve the value of the optimization function, the next one picked should not do any better; the samples are closer together.

Computing a single BC requires sorting all the points based on feature values, which is of complexity $O(|S| \times ln(|S|))$: computing a histogram in $O(|S|)$, then going through the bins of the histograms to compute the BC. Total complexity for a single feature is $O(|S| \times ln(|S|) + |S|) = O(|S| \times ln(|S|))$. Since this must be done for $|F|$ features, total time complexity for the heuristic is $O(|F| \times |S| \times ln(|S|))$. However, this is preprocessing for the algorithm, so it does not need to be computed during the progress of the DFS-BT. If the feature/center triples are ordered by increasing order of BC and enforce the constraint $k = 2 \times j$, then the solution space is $O(|N|^j)$. The optimization function can be computed in $O(j^2 \times |S|)$, only j features are used, and the winning and losing centers can be done at the same time. The optimization function, which must be computed to compare solutions, turns the overall complexity into $O(j^2 \times |F|^j \times |S|)$. The pseudocode is generally the same as Algorithm 2, only now the initial input to the function is $|F|$ feature/center triples, ordered based on BC. This leads to the program refinement in Algorithm 3.

3.2.6 Program Refinement Two. Although much better, the complexity of the problem is still too large to solve in a reasonable amount of time. With the addition of the BC heuristic, a greedy search could be used, in which the DFS tree is traversed by choosing the best node at each step. This type of search has no backtracking; it generates one solution and then terminates. Because the size of the

Algorithm 3 DFS-BT FS/KC Refinement

```
$Dp = Do = \oslash$
$Di^0 = N$
$l = 0$
$Select\, x \in Di^l$
$Di^l = Di^l - x$
$l = l + 1$
$Dp = Dp \cup x$
$Di^l = Di^{l-1} - x$
**if** $l = j$ **then**
    **if** $G^*(Dp) > G^*(Do)$ **then**
        $Do = Dp$
        BACKTRACK
    **end if**
    $z = last\, node\, added\, to\, Dp$
    $Dp = Dp - z$
    $l = l - 1$
    **if** $l = 0$ AND $Di^l = \oslash$ **then**
        TERMINATE
    **else**
        Return to Selection
    **end if**
**end if**
```

solution is limited to j, this search runs in $O(j)$. Since j is a user defined constant, this is constant time.

Alternatively, the search could be performed with some limited amount of backtracking. In the RTSPP, the best looking solutions are the ones with the best BC values. Any good solution should contain the best i feature/center triples. The only difference between good solutions is in the last $j - i$ choices. As a result, a greedy solution could be used until the partial solution has i triples, and then the partial solution could be completed through a DFS/BT search for the next $j - i$ triples. The search now reaches a lower portion of the tree, as depicted in Figure 3. The DFS-BT portion of the search runs in $O((j - i)^2 \times |F|^{j-i} \times |S|)$, which is now the complexity of the program.

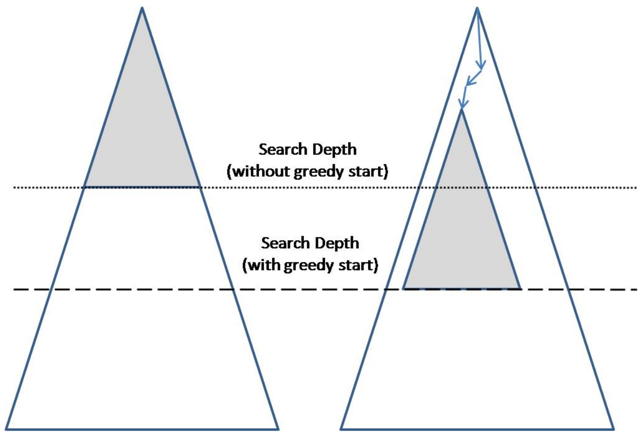


Figure 3: The increased search depth when a greedy search portion is included

The greedy algorithm program specification is in Algorithm 4, whose output is used as input to the DFS/BT search in Algorithm 5. The modified DFS-BT algorithm is Algorithm 5.

Algorithm 4 Greedy DFS

```
$Dp = Do = \oslash$
$Di^0 = N$
$l = 0$
$Select\, x \in Di^l$
$Di^l = Di^l - x$
$l = l + 1$
$Dp = Dp \cup x$
$Di^l = Di^{l-1} - x$
**if** $l = i$ **then**
    $Do = Dp$
    $DFS - BT Di^l, j - i, Dp$
**else**
    Return to selection
**end if**
```

3.2.7 Data Structure Specification. The next step is to pick appropriate implementation data structures and architectures. MATLAB provides an excellent way to compute the histograms needed for the BC: the data is stored as an $|F| \times |S|$ matrix, which can be easily manipulated.

Algorithm 5 DFS-BT 2

```
$Do = \oslash$
$Di^0 = N$
$l = 0$
$Select\, x \in Di^l$
$Di^l = Di^l - x$
$l = l + 1$
$Dp = Dp \cup x$
$Di^l = Di^{l-1} - x$
**if** $l = j$ **then**
  **if** $G^*(Dp) > G^*(Do)$ **then**
    $Do = Dp$
    BACKTRACK
  **end if**
  $z = last\, node\, added\, to\, Dp$
  $Dp = Dp - z$
  $l = l - 1$
  **if** $l = 0$ AND $Di^l = \oslash$ **then**
    TERMINATE
  **end if**
**else**
  Return to Selection
**end if**
```

Once the data is preprocessed it is stored in an array, giving constant time access to any particular cell. An entire row or column can easily be iterated through. This two-dimensional array is referred to as *data*.

In the greedy search a set of candidates is not needed. In fact, there is no need to code the greedy search portion; the value for i and j are passed as parameters to the DFS-BT algorithm, which determines where to start and what features/centers are already in the solution. i is the start level of the DFS-BT, j is the maximum search level. The current level of the search is l.

Feature/center triples are stored as three arrays. These are of size j. The winning/losing center arrays are passed to the DFS function as W and L, respectively. The length of each array is $|N|$, so if the algorithm selects feature $n \in N$, the appropriate winning and losing centers are stored in W_n and L_n. During BC preprocessing the features are ordered from lowest to highest BC. The features are already sorted from lowest to highest BC value in $data$, so there is no need to store the BC values.

The partial solution Dp is stored as three different one-dimensional arrays, all of size j. The first is the features so far, f. The others are the winning and losing centers, w and l. However, since the features and centers are split into feature/center triples, each feature can be used to reference the correct centers. There is no need for w and l, the appropriate entries in W and L can be referenced using the values in f. The best feature set found so far is another one-dimensional array, bf. The accuracy (fitness value) associated with this solution is in the real number variable $best$.

Finally, the set of candidates is stored as the one-dimensional array of integers p. This is of length j, and the value at p_l says our set of candidates at level l goes from p_l to $|N|$. This leads to the termination condition of $p_i = |N|$ and $l = i$.

3.2.8 Final Program Specification. The above data structures can be used to finalize the program specification (Algorithm 6). The final step is to implement the algorithm and test its performance: specific performance metrics are outlined in Section 3.4, with experimental results in Section 3.5.

3.3 A Stochastic Approach

An alternative approach to search the landscape is stochastic search. In a deterministic search, the entire search landscape is explored and a globally optimal solution is found. As a result of the complexity of the solution space, significant reductions to the problem were made to develop a deterministic solution to the RTSPP. In a stochastic search, the entire search space is not enumerated. Instead, the search is guided towards areas of the solution space which appear promising through the fitness function.

Algorithm 6 DFS-BT Final Specification

```
$best = 0$
$bf = \oslash$
$f_{l=1..i} = l$
$p_l = l$
$l = i$
**while** $(p_i \neq |N|$ AND $l \neq i)$ **do**
  **if** $(p_l = |N|)$ **then**
    $l = l - 1$
  **end if**
  $l = l + 1$
  $f_l = p_l$
  $p_l = p_l + 1$
  $p_{l+1} = p_l$
  **if** $l = j$ **then**
    **if** $G^*(f) > best$ **then**
      $bf = f$
      $best = G^*(f)$
    **end if**
    $l = l - 1$
  **end if**
**end while**
```

3.3.1 Algorithm Domain Requirements Specification. The first step in stochastic algorithm development is to formally define the search characteristics (adapted from [39]):

1. The search space $S(\pi)$ of instance π: a finite set of candidate solutions $s \in S$. These are sets of features/centers.
2. Set of feasible solutions: $S'(\pi) \subset S(\pi)$. These are sets of features/centers which meet the constraints given in Equations 2 and 3.
3. Neighborhood relation on $S(\pi)$: $N(\pi) \subset S(\pi) \times S(\pi)$. A neighborhood relation should be developed which can take a solution s and generate a new solution s'. The difference between these two solutions should be small; if it is too large, the search will thrash around the solution space instead of exploiting productive areas.

4. Initialization function *init*(π): $\oslash \rightarrow D(S(\pi))$ which specifies a probability distribution over initial search positions. There are numerous methods of generating an initial solution. It could be randomly generated, or it could be selected because it contains some pre-identified features/centers (based on expert knowledge) which are expected to be in a good solution.

5. Step function *step*(π): $S(\pi) \rightarrow D(S(\pi))$ which maps each search position onto a probability distribution over its neighboring search positions. The step function uses the neighborhood function: a solution is generated, then the step function determines whether it should be accepted.

6. Termination Predicate *terminate*(π): $S(\pi) \rightarrow D(\top, \bot)$ maps each search position to a probability distribution over truth values which indicate whether the search should be terminated upon reaching a specific point in the search space. In most stochastic search, this termination predicate is based upon some convergence criterion. If the solution quality does not increase, or increases less than some fixed constant ϵ, the search is terminated.

3.3.2 Algorithm Domain Function Specification. The next step is to outline how a typical stochastic algorithm flows. A general stochastic search algorithm begins with an initial solution to the problem. A set of new solutions are generated from the initial solution. The step function selects the next state from the solutions generated by the neighborhood function, and then repeats until the termination predicate is satisfied.

Algorithm 7 Stochastic Algorithm Specification

```
{Initialization}
$s = init(\pi)$
while Not $terminate(\pi)$ do
  $s' = N(s)$ {Generate the neighborhood}
  $s = step(s')$ {Select the new solution or not based on the step function}
end while
```

Algorithm 7 general stochastic specification; it can be used for any such algorithm. Determining which specific stochastic algorithm to use requires an analysis of the problem domain.

3.3.3 Problem/Algorithm Domain Integration. We believe the solution landscape of the RTSPP should have many local maximum and minimum points. However, most of these will exist in close proximity to each other; some features should be more closely related to the eventual outcome of a game. For instance, the total number of units for one player compared to the units for another player is one feature which would probably give good prediction accuracies, while the total amount of money or-fuel which could possibly be stored will probably not be in a solution. Local maxima should be near the global maximum, while local minima should be near the global minimum. As a result of this landscape, a stochastic algorithm that is initially biased towards exploration, but then tends to exploitation should be used.

This tentative analysis of the solution space shows the RTSPP may be responsive to a relatively simple stochastic algorithm like simulated annealing. Simulated annealing is very similar to the deterministic search algorithm hill-climbing. Hill-climbing starts with a solution and generates another solution in the neighborhood. If the fitness of this new solution is better, it becomes the solution and the algorithm repeats. If it is not better, then the generated solution is discarded and another solution is generated and tested.

In simulated annealing (SA), the same approach is taken, but worse solutions can be accepted with some probability. Hill-climbing is subject to getting caught in a local maximum since it has no way of escaping. The probabilistic acceptance provided by simulated annealing allows the algorithm to possibly escape from a local maximum. The probability of selecting a worse solution is based on the current temperature, which changes based on a cooling parameter. At the beginning of the algorithm, the temperature is high so almost all solutions are accepted. As the search continues, the temperature falls so lower quality solutions are accepted less frequently. By the end of the algorithm, SA becomes hill climbing.

Simulated annealing is easy to implement and runs quickly. It is a good choice to test the performance of a stochastic algorithm on the RTSPP.

3.3.4 Algorithm Domain Refinement. A complete SA specification requires a solution form, fitness function, neighborhood function and cooling function for the problem domain.

A solution to the RTSPP is a set of features along with a set of centers. There are $|F|$ features, so a solution to the feature selection portion of the RTSPP is a $|F|$-length binary string, where each feature is represented by a location in the string. A zero in the fth position of the string means feature f is not in the solution; a one means it is. Similarly, a solution to the center selection portion of the RTSPP is a binary string of length $|S|$, where a one in the sth position of the string means sample s is a center, while a zero means it is not. A total solution is a binary string of length $|F| + |S|$.

The fitness function determines the value of the chosen solution z. The fitness value is $G(z)$, from Equation 1.

To generate the next solution, the current solution may be mutated in two different ways. Either a bit in the solution is flipped, or two bits of opposite value (a zero and a one) are swapped. This neighborhood function slowly explores the solution space.

The cooling function is a geometric decreasing function defined by a parameter $0 < \alpha < 1$, where $T_{n+1} = \alpha \times T_n$. The probability of choosing a solution with lower fitness is the current temperature divided by the original temperature; T_n/T_0. Termination is when T_n approaches zero.

3.3.5 Program Specification. The combination of the algorithm constructs and specification generate the program specification in Algorithm 8.

Algorithm 8 SA RTSPP Initial Specification

```
$D_o = \oslash$
$x = x_0$
$n = 0$
while $T_n > 0$ do
    $Select\, z \in N(x)$
    if $g(z) > g(x)$ OR $random > (T_n/T_0)$ then
        $x = z$
    end if
    $T_{n+1} = T_n \times \alpha$
    $n = n + 1$
end while
$D_o = x$
```

The algorithm complexity depends on the time it takes to compute the fitness function $G()$. As in the deterministic solution, this takes $O(k \times |S|^2)$. The stochastic algorithm examines a new solution at each step. Since the termination condition is $T_n = 0$, and the current tempurature is a geometric cooling function based on α, SA tests $ln(\epsilon)/ln(\alpha)$ solutions, where ϵ is a number just above zero. The overall problem solution space, from the problem definition, is $O(|F|^k \times |S|^j)$. The stochastic algorithm will not be able to explore the entire solution space, but the SA implementation should guide the search in good directions so the unexplored portions of the space should be uninteresting ones.

3.3.6 Program Specification Refinement. The problem with the program as currently designed is in the neighborhood function. Allowing flipped bits can potentially change the number of features/centers in a solution. Since the two constraints are limits on the number of features and centers, this means the algorithm may generate infeasible solutions. To deal with this problem, a repair function could be introduced to "fix" infeasible solutions, or the neighborhood function could be changed. Since one of the main concerns with the search is complexity, and introducing a repair function increases complexity, changing the neighborhood function is the best course.

Instead of allowing "flipped" bits, only swaps are allowed, and bits must be swapped in the same portion of the binary solution so a bit in the feature portion of the solution is not swapped with a bit in the center portion. Three swaps are made; one in the feature portion and two in the center portion of the solution. For ease of notation, this function is called *swap*(). It takes the current solution x and returns a new solution z. An example of this swap is in Figure 4. s is a general solution, the first $|N|$ numbers are features, the next $|M|$ are samples. x is a possible solution, there are four features and eight samples in this example. The first four samples are winning, the last four are losing. z_1 is a possible nearby solution, one sample and two centers have been swapped. z_2 is not a nearby solution, two samples and four centers have been swapped out.

$$s = F_1..F_N \mid S_1 .... S_M$$

$$x = 0110 \mid 10010110$$

$$z_1 = 0101 \mid 10100101$$

$$z_2 = 1001 \mid 01101001$$

Figure 4: Proximity in solution space.

As already stated, the solution x is a binary string of length $|F|+|S|$. However, this is used to compute the fitness function $G(x)$. To reduce the complexity of this computation, there is a secondary implementation of the solution as three arrays of integers, one of l features and two of k centers. The feature array is F', the winning centers array is C_w and the losing centers array is C_l. When a new solution is accepted, these three sets are updated in constant time by removing the value swapped out and adding the value swapped in.

Additionally, the data array is used to compute the entire fitness function. Like in the deterministic solution, the data is stored in an array for fast access, the array *data*.

The best solution is x_{best}, and its value is $G(x_{best})$. As in the partial solution, this is a binary array of length $|F| + |S|$. In order to quickly print the best solution at the end of the program, the features and centers are stored in integer arrays like in the current solution; F'_{best}, C_w^{best} and C_l^{best}.

Instead of having the user specify the initial solution, it is generated randomly by picking l features, $k/2$ winning centers and $k/2$ losing centers.

The data structures lead to the final program refinement in Algorithm 9. The details of the integer array solutions F', C_w, C_l and their respective *best* values are left out; implementation can be done easily inside the *swap*() function.

Algorithm 9 SA RTSPP Final Specification

```
$x_{best} = 0$
$x = random$
$step = 0$
**while** $T_{step} > \epsilon$ **do**
  **if** $fitness(x) > fitness(x_{best})$ **then**
    $x_{best} = x$
    $z = swap(x)$
  **end if**
  **if** $fitness(z) > fitness(x)$ OR $random < (T_{step}/T_0)$ **then**
    $x = z$
  **end if**
  $T_{step+1} = T_{step} \times \alpha$
  $step++$
**end while**
```

At this point, the algorithm can be implemented and tested. In the next section, the experiments and parameters used to test the implementation are described. The results of the experiments are presented and analyzed in Section 3.5.

3.4 Comparing Two Search Algorithms

The deterministic DFS-BT and stochastic SA algorithms are designed to generate a classifier for the RTSPP, which could then be used to predict the outcome of a Real Time Strategy game long before it is over. Testing a search algorithm requires an exploration of parameter values, a data set and performance metrics. This section outlines all three.

3.4.1 Deterministic Search Parameters. The developed deterministic algorithm is a greedy depth first search with backtracking. It has two search parameters which could be varied: the depth of the local greedy jump start (i) and the max depth of the search (j). At each level, the search adds a feature/center triple, created using a BC, to the solution. At the beginning of the DFS portion of the search, the solution contains i feature/center triples. At the end of the DFS, a full solution has $j = k$ feature/center triples, where j and k are the constraints set out in Equations chap3:eqn:Constraint2 and chap3:eqn:Constraint2, and j is also the max depth of the search.

The depth of the DFS portion of the search is limited to values less than or equal to four because of computational complexity, or the constraint $j - i \leq 4$. Additionally, the goal of a solution to the RTSPP is to reduce the number of features in a solution, leading to the additional constraint $j \leq 8$. To test the performance of the algorithm, the search is run with the parameter combinations in Table 2.

Table 2: Parameter combinations for testing of the deterministic search algorithm.

Run	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
i	0	0	0	0	1	1	1	1	1	2	2	2	2	2	3	3	3	3	3
j	1	2	3	4	1	2	3	4	5	2	3	4	5	6	3	4	5	6	7

Run	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34
i	4	4	4	4	4	5	5	5	5	6	6	6	7	7	8
j	4	5	6	7	8	5	6	7	8	6	7	8	7	8	8

Performance is averaged across all three folds. There are 34 parameter combinations. Since this is a deterministic algorithm, only one run is required for each combination. There are 34 combinations $\times 3$ folds $= 102$ experiments for the DFS-BT algorithm.

3.4.2 Stochastic Search Parameters. The chosen stochastic search algorithm is simulated annealing. The SA algorithm has three search parameters: the initial temperature T_0, the cooling parameter α, and the number of features in a solution l. In the developed SA algorithm, the total number of centers in a solution are equal to the number of features.

Table 3 gives the parameter combinations for the SA tests. Because this is a stochastic algorithm, performance is averaged across fifty runs for each parameter combination. The experimental setup is a full factorial design (every parameter combination is tested) across the three parameters, so there are 343 runs $\times 50$ iterations $\times 3$ folds $= 51450$ experiments for SA.

Table 3: Parameter combinations for testing of the stochastic search algorithm.

Parameter	Range	Step	Unique Values
T_0	50..200	25	7
α	0.2 .. 0.8	0.1	7
l	2 .. 8	1	7

3.4.3 Data. The algorithms are tested on data from a RTS platform called Bos Wars [7]. Bos Wars is an open source RTS developed as a no-cost alternative to commercial RTS games. It has a "dynamic, rate-based economy", making it somewhat different than most other RTS games. Energy (money) and magma (fuel) are consumed at a rate based on the number of units and buildings a player owns. As the size of the player's army increases, more resources must be allocated to sustaining infrastructure. Additionally, Bos Wars has no "tech-tree", so all unit and building types can be created at the beginning of any game.

There are three scripted AIs packaged with the development version of the game: blitz, tankrush and rush. Blitz creates as many buildings and units as possible in the hopes of overwhelming the opponent. Tankrush tries to create tanks as quickly as possible, using a strong unit to beat the weaker units normally created at the beginning of a game. Rush creates as many units as quickly as it can and attacks as soon as possible in order to catch the enemy off guard.

There are eight maps packaged with the game. In most maps starting conditions for both players are similar. Each player has the same amount and access to resources and starts with the same number and type of units. Three different two-player maps were used: two had similar starting conditions, one had a line of cannons (defensive buildings) for one player.

Additionally, there are three different difficulty levels for the game: Easy, Normal and Hard. Changing the difficulty level allows the AI to execute its script faster, so it progresses farther in its strategy in a given time period during a Hard game than a Normal game and Normal progresses further than Easy.

To collect data, the Bos Wars source code is modified to take a snapshot of the game state at intervals of five seconds and output the feature values to a text file. Each snapshot consists of thirty different statistics: Energy Rate; Magma Rate; Stored Energy; Stored Magma; Energy Capacity; Magma Capacity; Unit Limit; Building Limit; Total Units; Total Buildings; Total Razings; Total Kills; Engineers; Assault Units; Grenadiers; Bazoo; Dorcoz; Medics; Rocket Tanks; Tanks; Harvesters; Training Camps; Vehicle Factories; Gun Turrets; Big Gun Turrets; Cameras; Vaults; Magma Pumps; Power Plants; Nuclear Power Plants. Additionally, thirty delta values for all the features based on the snapshot taken 25 seconds before were created, so there are sixty features to choose from.

Altogether, eighty-one games are recorded. For the three maps, three iterations are run for each combination of AI (tankrush v rush, tankrush v blitz, rush v blitz) at each difficulty level, so each map has twenty-seven game traces.

Win/loss prediction is easier the closer one gets to the end of the game, and almost impossible at the beginning. The goal of the RTSPP is to capture the important part of a game, where one player obtains an advantage over the other. To facilitate this, only game states in the third quarter of a game, the ones starting after 50% of the game had elapsed and before 75% of the game had elapsed, are used as input. The shortest game was about ten minutes long, while the longest was more than forty minutes. Predictions ranged from samples 2.5 minutes from the end of the game to 20 minutes from the end of the game. Table 4 gives the records of each scripted agent match on a specific map. Results are summed for each agent, no matter what difficulty level, since both agents have the same advantage. Table 5 shows the average standard deviation in game length for a specific agent combination at a specific difficulty level on a specific map. These statistics show the deterministic nature of the Bos Wars scripts. In a given agent combination on a given map, the same agent tends to win every time. The game length is almost the same every time.

Table 4: Records for each agent combination on the listed map. The first number is wins for the first agent, the second number is wins for the second agent.

Map/Agent Combination	Battlefield	Island Warfare	Wetlands
Rush v Blitz	6-3	9-0	9-0
Tank Rush v Blitz	9-0	9-0	9-0
Rush v Tank Rush	0-9	2-7	9-0

Table 5: Average standard deviation in game length (seconds) for agent combinations on specific maps. This standard deviation is an average of the standard deviation across the three difficulty settings.

Map/Agent Combination	Battlefield	Island Warfare	Wetlands
Rush v Blitz	0.00	31.30	38.10
Tank Rush v Blitz	0.00	0.00	34.78
Rush v Tank Rush	0.77	30.41	0.26

Extracting all the third quarter samples from the game leads to a sample size of about 4500. This data is split into two portions: the first, of around 3000 samples, is used by both algorithms to develop classifiers. This data is referred to as the Bos Wars Training Set. The remaining 1500 samples are held out and used to compare the best classifiers found by the two algorithms. This data is referred to as the Bos Wars Testing Set. Holding out a portion of the data so neither algorithm is allowed to train on it leads to a fair comparison. The percentage of winning samples in each data set is presented in Table 6. When analyzing results, the win/loss bias of the data determines how good the accuracy is when compared to an uninformed algorithm which simply assigns the majority label to every sample.

Table 6: Number of winning samples for each fold of the Bos Wars Training Set and the number of winning samples in the Bos Wars Test Set.

Data Set	Winning Samples	Samples	Percentage
Fold One	311	998	31.2%
Fold Two	311	998	31.2%
Fold Three	310	997	31.1%
Bos Wars Test Set	452	1463	30.9%

When generating classifiers, both algorithms use 3-fold cross validation to develop their classifiers. In 3-fold cross validation, the data is split into three sections. The algorithm takes two of these sections to train a classifier, and then uses the final third to test the performance of the classifier.

The Bos Wars Training Set is used to determine the best search parameters for each algorithm. Solutions obtained using the best search parameters on the Training Set are then tested on the Bos Wars Testing Set.

3.4.4 Hardware. Searches were run on one of two hardware configurations:

- **Configuration A**
  - Processor: AMD Athlon 64 x2 Dual Core Processor 5600+, 2.81 GHz
  - Memory: 2.00 GB
  - Operating System: Microsoft Windows Vista Home Premium, Service Pack One
  - Compiler: Geany v0.18

- **Configuration B**
  - Processor: AMD Phenom 8600B, 2.29 GHz
  - Memory: 3.48 GB
  - Operating System: Microsoft Windows XP Professional, Service Pack Three
  - Compiler: Microsoft Visual Studio 2008

In cases where search time is presented, the specific hardware configuration used to generate these times is denoted.

3.4.5 Performance Metrics. To assess the performance of each classification algorithm, two metrics are used: the fitness of the generated classifiers and the time to complete a search. The fitness of a classifier is its classification accuracy on the test set.

For the deterministic solution, every time the algorithm is run with the same parameter settings on the same data set, it will finish with the same solution. Repeated iterations are not required. For each parameter setting, the algorithm is run on each of the three folds in the data set. The best classifier found is tested on the appropriate fold, and the fitness across all three folds is averaged, giving an average classification accuracy for the parameter setting. The time to complete each search is expressed in seconds required for the search; this is also averaged across all three folds for the specific parameter setting.

In the stochastic search, subsequent runs of the algorithm do not necessarily result in the same answer, so one hundred iterations are run for each parameter combination on each fold. The average time required to complete one iteration is computed for each fold.

Finally, to compare the two algorithms, the classifiers for the top five parameter settings are tested on the Bos Wars Test Set. The average fitness for each parameter setting is computed and can be used for comparison of the performance of the two algorithms, along with the average time to complete a search.

This section outlined the search parameters, data, and performance metrics for search algorithm comparison. In the next section, performance of the algorithms on the Bos Wars Training Set is presented and analyzed. The best performing parameters are determined, and solutions generated using these parameters are tested on the Bos Wars Testing Set. The algorithms are directly compared on this testing data, and one search family is chosen for further development.

3.5 Classifier Comparison Results and Analysis

This section displays the results of the deterministic and stochastic search algorithms and compares their performance. First, the best performing deterministic search parameters are determined by examining algorithm performance on the Bos Wars Training Set. The process is repeated for the stochastic search algorithm. Next, the classifiers generated using the best performing parameters are compared on the Bos Wars Training Set. Finally, one algorithm family is selected for further development.

3.5.1 Deterministic Search. Deterministic search algorithm performance is measured in terms of time to search and classification performance. The chosen deterministic search algorithm was a Depth First Search with Backtracking (DFS-BT). 3-fold cross validation was used on the Bos Wars data set. Table 7 shows an average and a standard deviation for search time and classification accuracy across all the folds. i is the greedy search depth and j is the full search depth. Fitness is the average classification accuracy for the solution found in the training data on the appropriate test set for each fold. Time is the average length of the search rounded to the nearest second. St Dev is the standard deviation for the three measurements which were averaged.

3.5.2 Effect of Deterministic Search Parameters. In the deterministic search, there were two parameters: the greedy search depth and the total search

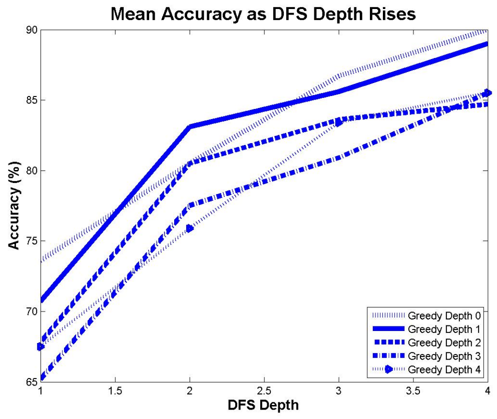
Table 7: Results for the DFS-BT across all folds on the Bos Wars Training Set.

i	j	Fitness	St Dev	Time (s)	St Dev
0	1	73.6%	0.039	<1	0.000
0	2	80.5%	0.050	1.33	0.577
0	3	86.7%	0.027	46.67	0.577
0	4	90.0%	0.004	1013.00	5.292
1	2	70.7%	0.060	<1	0.000
1	3	83.1%	0.046	3.00	0.000
1	4	85.6%	0.039	69.33	1.155
1	5	89.0%	0.019	1345.33	17.786
2	3	67.8%	0.060	<1	0.577
2	4	80.5%	0.040	3.33	0.577
2	5	83.6%	0.036	90.67	0.577
2	6	84.7%	0.008	1673.00	51.176
3	4	65.2%	0.050	<1	0.577
3	5	77.5%	0.020	4.67	0.577
3	6	80.9%	0.023	113.00	1.000
3	7	85.5%	0.022	1962.33	15.535
4	5	67.5%	0.010	<1	0.577
4	6	75.9%	0.031	6.00	0.000
4	7	83.4%	0.019	141.67	0.577
4	8	85.5%	0.009	2279.00	6.557
5	6	73.0%	0.017	1.00	0.000
5	7	83.4%	0.019	7.00	0.000
5	8	85.5%	0.009	164.00	0.000
6	7	72.9%	0.016	<1	0.577
6	8	83.1%	0.009	9.00	0.000
7	8	75.7%	0.082	<1	0.577

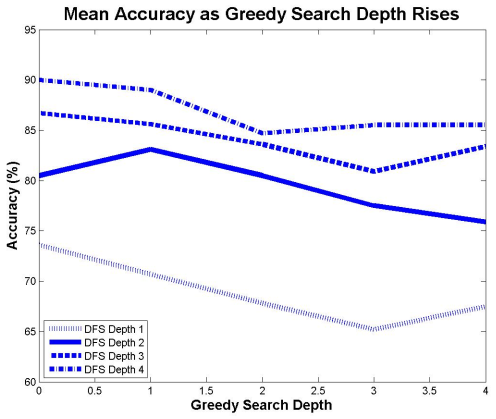
depth, i and j respectively. DFS depth is equal to $j - i$. The two graphs in Figure 5 show the effect of the two search parameters on classifier performance.

In the first, the direct relationship between classification accuracy and DFS depth $j - i$ can be clearly seen. No matter what the greedy search depth, the classification accuracy of the solution increases when the DFS is allowed to search deeper.

However, the greedy search portion, which can be seen in the second graph, is not as effective. Although not as definitive, the trend in the classification accuracy as i increases but $j-i$ is held constant appears to be downward. This can be validated by looking at the best performing parameter sets, as determined by mean classification accuracy: the top four parameter sets are where the greedy search depth is zero or one.



(a) Effect of DFS depth on classification accuracy



(b) Effect of greedy search depth on classification accuracy

Figure 5: (a), (b)

The solutions with the best fitness are generated for the parameter values $j-i = 4$ and $i = 0$. The results of the classifiers determined with these parameter values will be compared to the best stochastic algorithm solutions on a novel data set in Section 3.5.5.

3.5.3 The Bhattacharyya Metric. The Bhattacharyya Metric (BC) was computed for each training set in the Bos Wars Data before beginning the deterministic search. In Figure 6, the value of the BC for each feature in the training set is displayed, in order of lowest to highest. The best BC for any set is 37%, which quickly rises. The BC determines separability of a feature: its high values lead to the conclusion that the Bos Wars data is not very separable.

As a heuristic for the greedy search portion of the deterministic algorithm, the BC is ineffective. In almost all cases, adding more levels to the greedy search decreased performance. However, using the BC to pair features with centers was effective: using these triples, the deterministic search was able to attain accuracies over 90% in some cases.

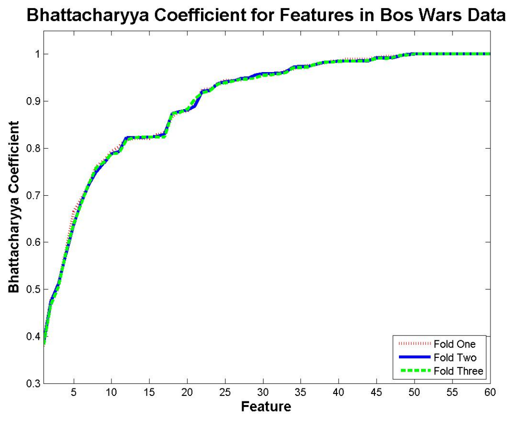


Figure 6: The BC for the features in the Bos Wars training sets.

3.5.4 Stochastic Search. The chosen stochastic search algorithm was simulated annealing. To fine-tune a stochastic algorithm, the effect of various parameters on solution fitness must be explored. Figure 7 depicts the effect of the number of features in the solution l, the cooling parameter α and the initial temperature T_0 on both solution fitness and search time across all three folds of the Bos Wars data.

The number of features in a solution and the cooling parameter have a direct relationship with both classification accuracy and search time. For alpha values, the relationship appears to be linear. An increase of 0.1 in α results in an average fitness increase of 2%. Two-sample t-tests for comparisons of the average fitness values for different alpha values all yield p-values of 0.000, giving significant statistical evidence that these averages are different. However, the increase in search time looks exponential. Increasing alpha exponentially increases the number of iterations for the simulated annealing algorithm. In Section 3.3, the number of simulated annealing iterations was derived as $ln(\epsilon)/ln(\alpha)$, so the exponential relationship is to be expected.

The number of features in a solution has a large impact on fitness at the low ends, but less at the high ends. Again, two-sample t-tests yield p-values of 0.000, giving significant statistical evidence of a difference in average fitness value for different feature values. The effect on search time is linear. This is also expected. The

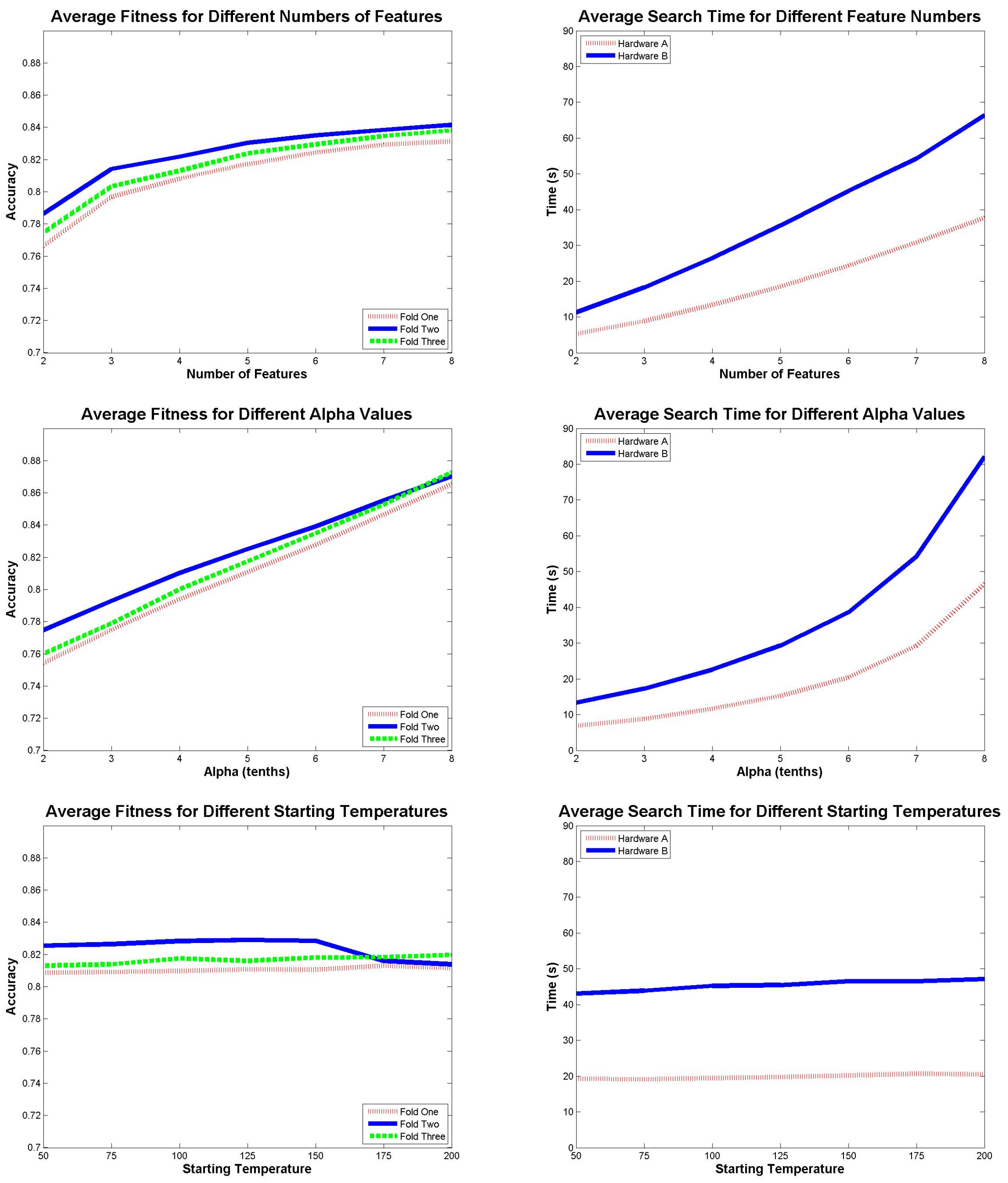


Figure 7: Effect of different parameter settings on overall classification accuracy and search time for Simulated Annealing on the Bos Wars data set.

complexity of the fitness computation is linear in the number of features, so an increase has a linear effect on complexity.

The starting temperature has a negligible effect on classification accuracy and search time. Two-sample t-tests for a difference in average fitness are less definitive, with p-values ranging from 0.6 to 0.000. The largest difference between average fitness is $< 3\%$, showing the starting temperature has little effect on overall fitness. This is because of the cooling function, which multiplies the current temperature by α to get the next temperature. For temperature to have a larger effect, the steps between values would have to be much larger. Basically, this would increase the number of iterations for the search. Since changing the value of α already does this, there is no real reason to adjust the starting temperature as well.

The detailed analysis of the effect of the parameter values leads to a selection of the best values for the Bos Wars data set. In this case, those values are $T_0 = 200$, $\alpha = 0.8$ and $l = 8$. In the next section, the results of the stochastic and deterministic algorithms are compared on the Bos Wars Test Set.

3.5.5 Comparing Deterministic and Stochastic Search. To choose whether to develop a deterministic or stochastic algorithm, we must compare the solutions found by each. For each algorithm, the best performing search parameters were determined. In the deterministic algorithm, these parameters are $i = 0$ (greedy search depth) and $j = 4$ (total search depth). For the stochastic algorithm, the parameters are $T_0 = 200$ (starting temperature), $\alpha = 0.8$ (cooling parameter) and $l = 8$ (number of features in solution).

Instead of comparing the results of the algorithms on the data sets already seen, they are tested on a different Bos Wars data set on which neither was allowed to train. The deterministic algorithm uses the three different solutions developed for the parameter settings. Each solution is the result of a DFS on a different fold of the Bos Wars training set. The stochastic algorithm was run fifty times on each fold, so there are 150 different solutions for the best parameter set. All these solutions are tested on the novel data set.

The classification accuracy, along with the time which was required to generate each solution from the training data, is presented in Table 8.

Table 8: Results for the solutions found with the best parameters by the deterministic and stochastic algorithm. Accuracy is the average accuracy on the Bos Wars test data, on which neither algorithm was allowed to train. Search Time is the average time in seconds of an average search with the best performing parameters on Hardware Configuration A.

Algorithm	Accuracy	Search Time
Deterministic	91.4%	1013.0
Stochastic	96.2%	75.0

The results are unequivocal: the stochastic algorithm outperforms the deterministic algorithm on both performance metrics. Additionally, the SA solution exposes information about the problem domain. Figure 8 shows the number of times each feature appears in one of the 150 SA solutions tested on the Bos Wars Test Set. Although some features are clearly used more than others, no single set of features appears to dominate all the solutions. The standard deviation of the fitness for each iteration is 0.000197, showing all the solutions found have similar fitness values.

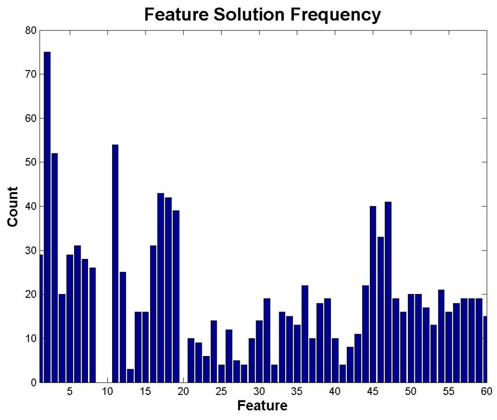


Figure 8: The frequency of each feature in the 150 SA solutions evaluated on the Bos Wars Test Set.

We conclude there is no single feature representation which is obviously better. Good feature representations are spread out around the space, with many different local maximums which appear to have similar accuracy. While the exact difference between the fitness of these solutions and the fitness of the optimal solution is unknown, the max fitness is 100%, so they can not be more than 4% below. Good solutions can be found in many different sections of the solution space. The solution space of the RTSPP on the Bos Wars data set is jagged, but in a particular way: there are many local maximums, and their fitness is close to the global maximum.

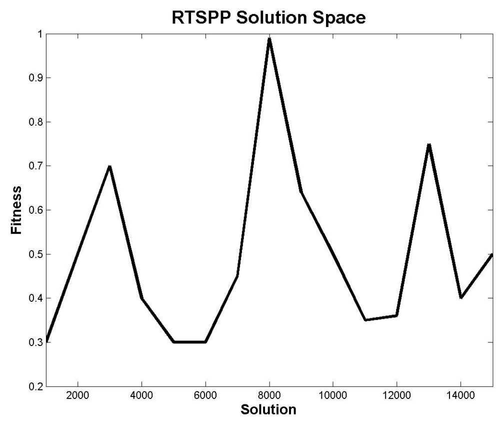
Traditionally, SA performs well on a solution space with few local maximums, as in Figure 9(a). In this space, there is one global and two local maximums. By preinitializing the SA algorithm multiple times, eventually it will start in the right portion of the space and find the global maximum. Even if fails to start near the global maximum, the stochastic nature of the algorithm gives it a chance to escape the local maximum and transfer to the global.

However, if there are too many local maximums, as in Figure 9(b), the SA algorithm has to have a very specific starting location to avoid getting trapped. Initially, SA can deal with this type of solution space by increasing the number of times it is restarted, but as the number of local maximums increases, this solution is no longer effective. The required starting located is severely restricted, and finding it requires too many SA iterations for the algorithm to run in a reasonable amount of time.

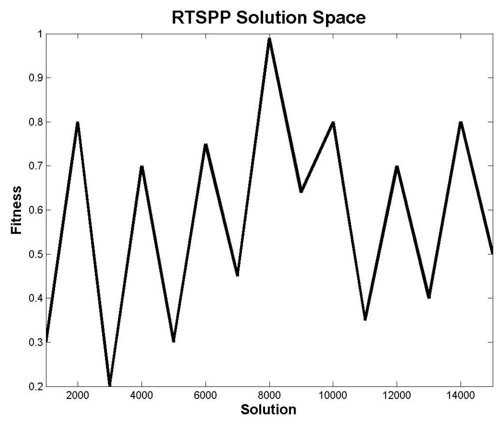
In the RTSPP there are many local maximums, but the fitness of these local maximums is close to the fitness of the global maximum, as in Figure 9(c). This means an SA algorithm can start in almost any place in the domain and find a near-optimal solution.

The deterministic algorithm attempts to reduce the solution space in such a way that it preserves the high fitness value solutions and prunes away the low fitness value solutions. Such a space would look like Figure 9(d), where the best solutions from Figure 9(c) have been preserved. This space is still jagged, but the number of solutions have been significantly reduced, and the low fitness solutions have been eliminated.

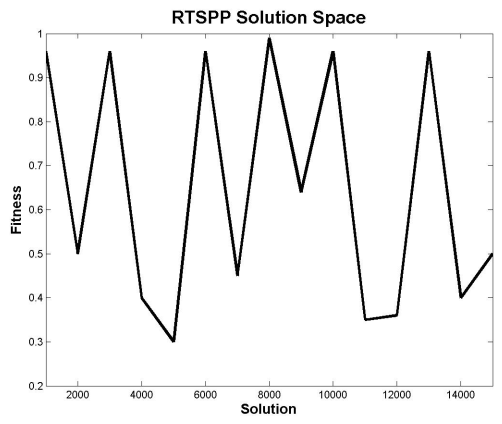
However, the reduction made with the BC actually results in a solution space like Figure 9(e). Some of the low fitness solutions have been discarded, but the highest fitness solutions have not been preserved. The entire space can be searched,



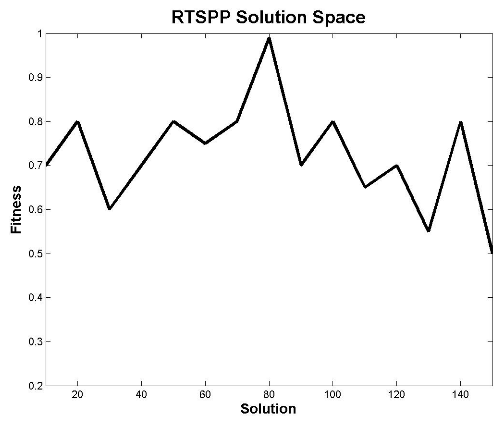
(a) Ideal SA solution space



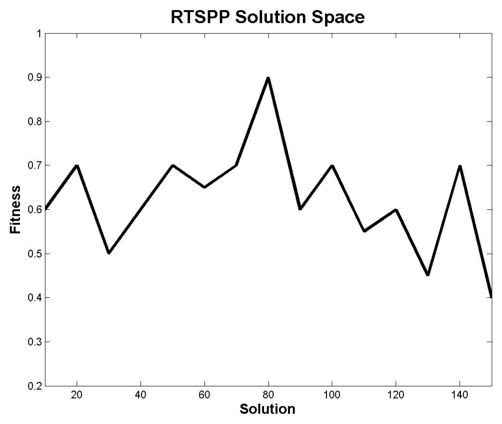
(b) Unideal SA solution space



(c) Actual RTSPP SA solution space



(d) Ideal deterministic solution space



(e) Actual deterministic solution space

Figure 9: Possible Solution Spaces

but since the best solutions are no longer in the space, the optimal solution in this space is significantly below the near-optimal solution found by SA when searching the complete space.

This solution space analysis, enabled by the performance of the two algorithms on the Bos Wars data set, can be used to inform the development of a more complex algorithm to solve the RTSPP.

3.6 Conclusion

The simulated annealing solution gives good performance on this data set. However, simulated annealing is a simple stochastic search algorithm which was chosen for the ease with which it could be implemented. It would be complicated to refine the algorithm to better suit the RTSPP.

Additionally, the failure of the BC metric to generate good classification accuracies for the deterministic solution show that features are dependent. Feature combinations determine the outcome of an RTS game, not single features.

An analysis of the performance of the two algorithms leads to the conjecture that the solution space of the RTSPP looks like Figure 9(c), and there is no heuristic which can be used to effectively prune the size of the solution space.

Therefore, we would like to develop a more complex algorithm which could be specifically tailored to the discovered solution space characteristics of the RTSPP. We choose an evolutionary algorithm, because it has a population of solutions which can be initialized in different areas of the solution space, finding many different feature represenations. In the next chapter, this algorithm is developed, refined and tested to solve the RTSPP.

IV. An Evolutionary Algorithm

In Chapter III we demonstrated a stochastic algorithm is better than a deterministic algorithm when solving the RTSPP. Additionally, an analysis of the solutions found by simulated annealing showed the solution space is jagged, with many local maximums whose fitness is similar to the global maximum. The failure of the BC heuristic for the deterministic search showed that features are dependent.

We would like to find good solutions from different portions of the solution space so many different state representations can be explored. One way to do this is through an evolutionary algorithm (EA) which evaluates a population of solutions. In Section 4.1 an EA is developed and tailored to the RTSPP. Section 4.2 outlines an experimental methodology to test the algorithm's performance. Finally, the results of experiments are presented and analyzed in Section 4.3.

4.1 An Evolutionary Algorithm

EAs are based on the idea of natural selection. A general overview of EAs can be found in Chapter II.

Each state in an EA is called a generation. A generation is a set of solutions to the specific search problem. The goal of an EA is to find good solutions at each generation and use them to create better solutions for the next generation. At each generation, solutions are combined and mutated together to produce new solutions. The fitness of these new solutions is evaluated, and some are selected for the next generation. As the generations go on, the solutions in the population improve.

There are many different types of EAs. The development of an evolutionary algorithm to solve the RTSPP begins with an exploration of the general EA domain.

4.1.1 Requirements Specification. The general EA is an 8-tuple (taken from [3]):

$$EA = (I, \Phi, \Omega, \Psi, s, \iota, \mu, \lambda)$$

- I: the space of individuals (possible solutions) to the problem

- Φ: function which assigns a fitness value to an individual in I
- Ω: set of probabilistic genetic operators
- Ψ: generation transition function
- s: selection operator
- ι: termination criterion
- μ: number of individuals in current generation
- λ: number of individuals created using Ω

4.1.2 Function Specification. The general outline of an evolutionary algorithm is given in Algorithm 10 [3]. In this function specification, $P(t)$ is the population at generation t and p_i represents the ith member of P.

Algorithm 10 Evolutionary Algorithm

```
{Initialize}
$t = 0$
Select $P(0) \in I$
{Evaluate}
Compute $\Phi(p_i), i = 1, 2, ..., \mu$
while NOT $\iota$ do
    {Recombine}
    $P'(t) = \Omega_r(P(t))$
    {Mutate}
    $P''(t) = \Omega_m(P'(t))$
    {Evaluate}
    Compute $\Phi(p''_i), i = 1, 2, ..., \lambda$
    {Select}
    $P(t+1) = s(P''(t))$
    {Iterate}
    $t = t + 1$
end while
```

The algorithm above goes through the basic phases of any evolutionary algorithm. The EA initializes a population, evaluates that population, does recombination, mutation and selection and then repeats. This is done until the termination condition is reached. The particular way in which the specific algorithm performs recombination, mutation and selection determines what specific type of evolutionary algorithm is used. The choice of evolutionary algorithm type is done based on an analysis of the problem domain. The RTSPP lends itself to a hybrid genetic algorithm/evolutionary strategy.

An evolutionary strategy is chosen for the center portion of the problem because its recombination and mutation operators can be used to manipulate real valued variables. Instead of limiting centers to samples in S, real values can be used to find center values. This means each center can be adapted to the specific feature it encodes, maximizing the use of each value.

However, features cannot be real values. There is a feature 1 and 2, but no feature 1.5. Therefore, a genetic algorithm formulation is used on the feature portion of the problem.

4.1.3 Evolutionary Strategy Specification. A general evolutionary strategy formulation, taken from [3], is shown in Algorithm 11.

Algorithm 11 Evolutionary Strategy

```
$t = 0$
{Initialize}
$P(0) = \overrightarrow{a_0}, ...., \overrightarrow{a_\mu} \in I^\mu$
{Evaluate}
Compute $\Phi(\overrightarrow{a_0}, ...., \overrightarrow{a_\mu} \in I^\mu)$
while NOT $\iota$ do
    {Recombine}
    $P'(t) = r'(P(t))$
    {Mutate}
    $P''(t) = m'(P(t))$
    {Evaluate}
    Compute $\Phi(\overrightarrow{a_0}, ...., \overrightarrow{a_\mu} \in I^\mu)$
    {Select}
    $P(t+1) = s'(P''(t))$
    $t = t + 1$
end while
```

An evolutionary strategy uses the recombination operator r', mutation operator m' and selection operator s', which are defined based on the specific problem domain.

4.1.4 Genetic Algorithm Specification. A general genetic algorithm function specification, from [3], is given in Algorithm 12.

Algorithm 12 Genetic Algorithm

```
$t = 0$
Initialize $P(0)$
Evaluate $P(0)$ based on fitness function
while Not termination condition $\iota$ do
    Recombine $P(t)$ to produce $P'(t)$
    Mutate $P'(t)$ to produce $P''(t)$
    Evaluate $P''(t)$
    Select $P(t+1)$ from $P''(t)$
    $t+ = 1$
end while
```

Additionally, a genetic algorithm has a mutation rate, p_m, a recombination operator, r, the length of each object variable, l, the scaling window, ω, and the population size, μ. The next step is to populate the algorithm domain specifications with problem domain specific information.

4.1.5 Problem/Algorithm Domain Integration. The first step in evolutionary algorithm development is to define a population member. In the RTSPP, a population member is a binary string representing each feature which shows which of the possible input features are contained in the specified solution. Additionally, each member contains centers which are real number vectors specifying a value for each possible feature. Only some of these values are active at any particular time, as determined by the features which are contained in the specified solution. The center value for each solution is determined by the features which are active in the solution. The number of centers is determined by the constraint in Equation 3.

In an evolutionary strategy, recombination is done using some combination of two members of the current population. For the RTSPP, two solutions are randomly selected from the population using a probability distribution based on fitness, where solutions with higher fitness values have a greater chance of selection but all solutions have some chance of selection.

These two parent solutions are used to generate a new solution. First, the feature portion of the first parent is copied to a new solution. Then, the center values are copied. If only one of the two parent solutions contains a feature, then its center values for that feature are copied to the new solution. If both parents contain the feature, then the center values of the parent with higher fitness are copied. If neither parent contains a feature, then the center values are randomly generated.

For the feature selection portion, some specific characteristics of the problem domain are utilized. First, features should not be chosen one by one. Switching out a single feature may have a huge effect, but only because of its interaction with some other feature in the representation. Similarly, adding a feature to a subset has a greater effect if it interacts with some feature already in the subset.

At first, this would appear to lend itself to a crossover type recombination operator. However, it is important to keep the number of features in a solution small, and crossover does not guarantee this. A better approach is to use a mutation type operator, where mutation is done by shifting all the bit values of the feature selection solution to the right some number of bits, completely changing the features in the solution. If the initial population contains many distinct feature sets, this mutation operator will lead to good exploration of the search space.

Selection is done based on the fitness value of the solutions in the population. Total elitism is used in selection, to increase the convergence speed of the algorithm. The population at each generation contains μ solutions, and λ more solutions are generated through recombination and mutation. During selection, the μ solutions with the highest fitness are selected from the population for the next generation.

4.1.6 Data Structure Selection. In an evolutionary algorithm, a solution is a vector of variable values which completely define a solution to the problem. In the RTSPP, there are two separate sections of this vector: the feature selection section and the center value section.

The feature selection section is a binary string of length $|F|$, where each bit corresponds to a feature in F. If the bit is a one, then the feature is in the solution. If it is a zero, then the feature is not in the solution. The number of features in a solution is limited to some integer $w \leq |F|$, as in Equation 2.

The center value section consists of $2 \times w$ different real valued vectors, each containing $|F|$ feature values. The first w vectors correspond to winning centers, while the last w correspond to losing centers. Each of the values in a vector corresponds to the value of a specific feature. The feature values actually used in a solution are determined by the feature selection section, but all values are present. This ensures center values are linked to specific features, so a change in the feature selection section does not throw off the gains which may have been made through genetic selection.

Formally, $s_i \in P_t$ refers to the ith solution in the population P at generation t. Similarly, $x_j^k \in s_i$ refers to the value of feature j in center k. The feature selection section of the solution is $n_i \in s_i$, where n_i is the $|F|$ bit binary string representing the features in solution i.

The data for the RTSPP is in the $|F| \times |S|$ data array $DATA$, so $DATA_{ij}$ refers to the value of the ith feature in the jth sample.

4.1.7 Program Specification. With appropriate data structures defined, we can move to a general program specification, contained in Algorithm 13. ρ is used to represent a unique random number, generated when appropriate.

This program specification is basically the same as the general evolutionary algorithm specification. A complete definition of the program specification requires a detailed declaration of the mutation and recombination functions.

The parameter λ_r determines how many solutions will be generated by recombination. λ_r should be sufficiently large to so it generates enough solutions for the next population, and each solution in $P(t)$ has a good chance of contributing something to $P(t+1)$. However, the fitness function must be computed for every solution

Algorithm 13 RTSPP EA Search

```
{Initialize}
$t = 0$
$P(0) = random$
{Evaluate}
Compute $\Phi(P(0))$
**while** NOT $\iota$ **do**
  $P(t) = P(t) \cup Recombine(P(t))$
  **for** $\forall s_i \in P(t)$ **do**
    **if** $\rho < p_m$ **then**
      $P(t) = P(t) \cup Mutate(s_i)$
    **end if**
  **end for**
  Compute $\Phi(P''(t))$
  $P(t+1) = s(P''(t))$
**end while**
```

Algorithm 14 RTSPP Recombine

```
**for** $i = 1$ TO $\lambda_r$ **do**
  Select $p_1 \in P(t)$
  Select $p_2 \in P(t)$
  **for** $j = 1$ TO 12 **do**
    **for** $k = 1$ TO $|F|$ **do**
      **if** $n_i \in p1$ AND $n_i \in p2$ **then**
        **if** $\Phi(p1) > \Phi(p2)$ **then**
          $x_j^k = x_j^k \in p1$
        **else**
          $x_j^k = x_j^k \in p2$
        **end if**
      **else if** $n_i \in p1$ **then**
        $x_j^k = x_j^k \in p1$
      **else if** $n_i \in p2$ **then**
        $x_j^k = x_j^k \in p2$
      **else**
        $x_j^k = random()$
      **end if**
    **end for**
  **end for**
**end for**
```

generated, so it is important to keep λ_r from getting too large to prevent excessive computational effort. The recombination specification is in Algorithm 14.

Next, the mutation function must be defined. Instead of providing a probability for mutation, since this is the only way to generate different feature combinations after the initial population has been formed, a certain number of solutions are mutated, just as in recombination. The number of solutions to be formed by mutation is λ_m. Mutated solutions are selected from a fitness based probability distribution. Solutions with higher fitness have a greater chance of selection. The mutation specification is in Algorithm 15.

Algorithm 15 RTSPP Mutation
```
for $i = 1$ to $\lambda_m$ do
  Select $p_1 \in P(t)$
  $r \leftarrow randint()$
  for $j = 1$ to $|F|$ do
    $n_j \leftarrow n_{(j+r)\%|F|} \in p1$
  end for
  $x = x \in p1$
end for
```

4.1.8 Complexity Analysis. To understand the algorithm completely, its complexity must be analyzed.

First, the solution space changes because of the modifications to the problem domain. There are no more than w features in a solution, so there are $O(|F|^w)$ possible combinations of features. Centers are now real valued, which increases the complexity of this portion. However, they are all on the interval [-1, 1]. If some granularity is defined, so we only look at decimal places out to the thousandth, there are $O(2000^{|S|})$ possible center combinations. Since a stochastic global search algorithm was developed, there is no plan to completely explore the solution space, but deriving its total complexity gives insight into the level of problem which must be solved, as well as how many solutions should be included in each generation.

In terms of algorithm complexity, the first thing to look at is the fitness function. Computing the fitness function for one solution takes $O(|S| \times 16 \times |F|) = O(|S| \times |F|)$, since it must go through every sample and determine to which center it is closest. There are $\mu + \lambda, \lambda = \lambda_r + \lambda_m$ solutions at each generation. If there are G generations, the total complexity is $O(|S| \times |F| \times (\mu + \lambda) \times G)$.

4.1.9 Memetic Refinement. The implementation of this algorithm is highly biased towards exploitation. On a large scale test problem, it converges at $G \leq 50$, with low fitness values. However, it does show a quick increase in average population fitness from one generation to the next. Additionally, elitism makes it sensitive to start value; the best fitness in the initial population determines how much the average fitness increases before convergence.

In order to take advantage of this quick convergence and elitism bias, we use a memetic algorithm to diversify the population after some number of generations. A memetic algorithm alternates between performing global search and local search. The local search is used to improve on the solutions in the current population. It can also serve to shake the search out of convergence by injecting new information into the population. After a local search iteration, global search is restarted in a new area, leading it to explore new areas and find better solutions.

In a classification problem, local search can be implemented using a k-means clustering algorithm, which modifies the centers in each solution. K-means is a local search algorithm. It takes a certain number of points among all the samples. Each sample is assigned to the closest point, and then the point is adjusted to be the mean of all the values of samples assigned to it. This process is repeated until the change from one iteration to the next approaches zero.

By using the centers found in solutions as the points for a k-means algorithm, solutions can be improved and diversified. Once completed, the new solutions are fed back to the evolutionary algorithm, which again improves the solution. The average fitness of solutions at the end of this memetic algorithm is greatly improved.

For each solution in the population, the local search splits all the samples in S into sets K_i based on the closest appropriate center (winning or losing), where K_i represents all the samples closest to center i. The next step is to take all the samples in each set and, for each active feature in the current solution, sets the feature values in each center to the average of all the values in the sample set. The K-means specification is in Algorithm 16.

Algorithm 16 K-means Optimization

```
for $\forall p \in P(t)$ do
    for $\forall s \in S$ do
        $K_i = K_i \cup s$ Where $x_i$ is the closest center to $s$ in $p$
    end for
    for $j = 1$ TO 12 do
        for $k = 1$ TO $|F|$ do
            if $n_k \in p$ then
                $x_k^j = Mean(\forall DATA_{ij} \in K_i)$
            end if
        end for
    end for
end for
```

Local search is performed every ten generations. Instead of using a convergence criterion, local search is run for twenty iterations, and then the global search continues.

This section described the development of a Memetic Evolutionary Algorithm to solve the RTSPP. The next section outlines the experiments and parameters used to test the algorithm. Results and analysis of the algorithm's performance on two different data sets are discussed in Section 4.3.

4.2 Evaluating Evolutionary Algorithm Performance

The evolutionary algorithm was developed to leverage the problem domain characteristics of the RTSPP. An evolutionary algorithm is a stochastic algorithm, so its performance over a number of iterations must be measured just as in the SA algorithm. In this section, the data used to test the algorithm, the parameter settings for the tests and the performance metrics used to evaluate performance are outlined.

Table 9: The parameter combinations used in tests of the Evolutionary Algorithm. λ_m is the number of solutions generated through mutation, λ_r is the number of solutions generated through recombination. The search hardware provides the constraint $\lambda_m + \lambda_r \leq 300$

Experiment	1	2	3	4	5
λ_m	50	100	150	200	250
λ_r	250	200	150	100	50

4.2.1 Evolutionary Algorithm Parameters. The developed evolutionary algorithm has six different parameters: the population size μ, the number of solutions to be generated by mutation λ_m, the number of solutions to be generated by recombination λ_r, the convergence criterion ι, the maximum number of generations G and the number of features in a solution w. Additionally, the algorithm can be run with or without the local search option.

In an evolutionary algorithm, the value of μ should be set represent as many solutions as possible. The hardware used for the search limits this to $\mu = 200$. λ_m and λ_r can be tuned to the problem domain by evaluating their performance at different values. These values are set out in Table 9. The termination criterion ι should be set so as to allow the algorithm to continue as long as is useful. The maximum number of generations never comes into play because ι is satisfied before the number of generations reaches G. The termination criterion is based on change in the average fitness values of all the solutions in the population. If the change in the average fitness is less than 0.0005 over ten generations, then the algorithm terminates. The number of features in a solution is set to $w \leq 6$, ensuring there are no more than six features in a solution. While both the previous solutions were allowed to use up to eight features, this is actually too many for the eventual RTS agent to reason over.

Fifty runs of the evolutionary algorithm were performed for each parameter combination with and without the local search option, so there are 5 combinations $\times 50$ runs $\times 3$ folds $\times 2$ options $= 1500$ runs for each data set.

4.2.2 Data. There are two sources of data used to test the RTSPP evolutionary algorithm: Bos Wars and Spring. The Bos Wars data is the same as described in Section 3.4.3. Three fold cross validation is used on the Bos Wars Training Set, and solutions from the best performing parameter sets are tested on the Bos Wars Test Set.

In addition, data was collected from the open source game Spring. Spring, similar to Bos Wars, is a dynamic resource based game set in a futuristic combat environment. The Spring source code was modified to take snapshots at five second intervals across fifteen different features: Solar Collectors, Metal Extractors, Advanced Solar Collectors, Energy Rate, Metal Rate, KBotlabs, Vehicle Plants, Advanced Vehicle Plants, Construction Vehicles, Light Infantry, Heavy Infantry, Light Tanks, Medium Tanks, Heavy Tanks, Light Laser Turrets. Then, two different delta values were computed for each feature: the change from ten seconds before and 25 seconds before. In all, there were forty-five different features collected from Spring.

Only one map was used, MetalHeckv2, designed for two person competitive games. MetalHeckv2 is a map which simplifies resource collection because metal can be collected at the same rate anywhere on the map. It is mostly flat, with some cliff-like projections which can be used for cover.

Four different scripted agents were used: Infantry Rush, Tank Rush, Blitz and Turtle. All scripts are outlined in Appendix B. Each script was played eight times as player one against all the agents, so ninety-six games were collected.

Just as in Bos Wars, data from the third quarter of games was extracted, and three-fold cross validation was used to test algorithm performance.

For comparison to the Bos Wars data, statistics about win percentage and game length for the Spring data are presented in Table 10 and 11. The bias of the data is in Table 12.

As compared to the Bos Wars data set, the Spring data set has much higher variance. No one agent is dominant. Additionally, there is high variance in game

Table 10: Records for Spring agent combinations. All games were played on a single map. The agent in the row is player one, while the agent in the column is player two. The first number is wins for player one, the second is wins for player two.

Agent	Blitz	Infantry Rush	Tank Rush	Turtle
Blitz	-	6-2	3-5	4-4
Infantry Rush	5-3	-	3-5	2-6
Tank Rush	5-3	1-7	-	4-4
Turtle	4-4	7-1	4-4	-

Table 11: Average standard deviation in game length (seconds) for agent combinations. The agent in the row is player one, the agent in the column is player two.

Agent	Blitz	Infantry Rush	Tank Rush	Turtle
Blitz	-	282.6	162.8	209.4
Infantry Rush	265.9	-	320.9	1136.9
Tank Rush	239.4	153.1	-	299.3
Turtle	353.7	744.5	226.5	-

Table 12: Number of winning samples for each fold of the Spring data set.

Data Set	Winning Samples	Samples	Percentage
Fold One	687	1444	47.58%
Fold Two	694	1444	48.06%
Fold Three	691	1444	47.88%

length for each agent combination. This makes the Spring data set a much more complicated classification problem. Finally, the Spring data set has much less bias then the Bos Wars set. All of the folds are around 48% winning samples, compared to 30% for the Bos Wars data.

4.2.3 Hardware. All EA tests are run on the same hardware configurations outlined in section 3.4.4.

4.2.4 Performance Metrics. In an evolutionary algorithm, performance can be measured through four factors: convergence speed, unique solutions in the population, the fitness of the best solution in the population and the average fitness of the population. Convergence speed is the number of generations it takes the algorithm to terminate. The number of unique solutions is the number of solutions in the population which are significantly different from each other: in the RTSPP, significantly different solutions are those which have a different feature set. The average fitness of the population is the average fitness of all the different solutions in the population.

To determine the quality of the evolutionary algorithm each parameter combination is run fifty times on each fold, both with and without the local search iterations. Then, the mean number of unique solutions, the mean fitness of the population, the fitness of the best solution in the population, the convergence speed (number of generations before convergence criterion is satisfied) and average search time are measured. The results on the Bos Wars data are compared to the results obtained for the stochastic solutions.

4.3 Evolutionary Algorithm Results and Analysis

In this section, the results of the EA on both the Bos Wars and Spring data sets are presented and analyzed. The optimal parameter values for number of solutions created by mutation and recombination are determined.

4.3.1 Bos Wars Data. Table 13 presents the average fitness of the unique solutions in the population and the max fitness of a solution in the population for the five parameter settings on the Bos Wars data. T-tests for each of the pairwise combinations give some evidence of a statistical difference in average fitness based on parameter settings. For example, the p-value for a test between the $\lambda_r = 50$ and $\lambda_r = 200$ rows is 0.000, which gives significant evidence of a difference. However, a test between the $\lambda_r = 100$ and $\lambda_r = 150$ rows yields a p-value of 0.940, giving no evidence of a difference in means. Even when there is statistical evidence for a difference in mean, the actual difference is no more than 0.8%. There is no real effect on the performance metrics based on changing the parameter settings. This could be due to the low variance in the Bos Wars data set. The EA is able to find good solutions no matter what parameters are used, because it has a large population size and the mutation and recombination operators are tuned to the domain.

Table 13: Average fitness of the population and a standard deviation for various parameter settings on the Bos Wars data with the memetic option. Averages are across the three folds. λ_r is the number of solutions created by recombination, λ_m is the number of solutions created by mutation.

λ_r	λ_m	Avg Fitness	St Dev
50	250	71.2%	12.7%
100	200	71.7%	12.5%
150	150	71.7%	12.5%
200	100	72.0%	12.4%
250	50	71.5%	12.5%

Table 14: Average fitness of the population and the average fitness of the best solution in the population at termination of the EA on Bos Wars data with and without the memetic option. The standard deviation for average fitness is for the entire population across the five different parameter settings. The standard deviation for the max fitness is for best fitness found after each run across the five different parameter settings.

Search Option	Avg Fitness	St Dev	Max Fitness	St Dev
Memetic	71.3%	12.7%	89.8%	0.18%
No Memetic	60.0%	12.7%	86.8%	0.23%

Table 14 shows a comparison of the performance of the EA with and without the memetic option. The memetic option is able to increase both the average and max fitness of the population upon search termination. Using a local search component injects more information into the population, allowing it to continue to find better solutions. A two-sample t-test for a difference in the max fitness yields a p-value of 0.000, giving significant statistical evidence of a difference in average fitness. The p-value for a difference in the max fitness also yields a p-value of 0.000, again giving significant statistical evidence of a difference in max fitness [62].

Figure 10 shows the effect on the number of unique solutions in the population at convergence caused by the different parameter settings with the memetic option. The parameters again appear to have no effect on the average number of unique solutions.

For the stochastic algorithm from Chapter III using the optimal cooling and initial temperature and six features in the solution, the average accuracy on the Bos

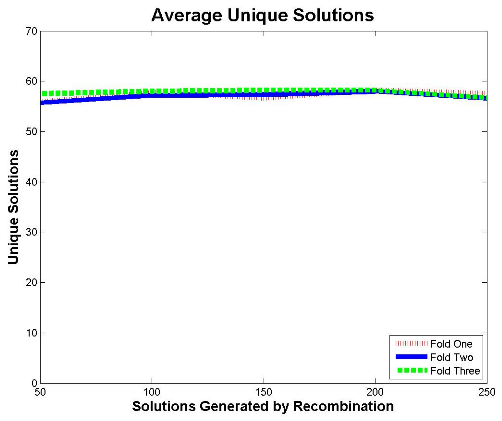


Figure 10: The effects of different parameter settings on the number of unique solutions in the population for EA searches on the Bos Wars data. The horizontal axis shows the number of solutions created by recombination (λ_r). The number of solutions created by mutation is $\lambda_m = 300 - \lambda_r$. Each line represents a different fold of the data.

Wars Training Set is 87.8%. The lowest average max accuracy for the EA on the same data set is 89.6%. When the search algorithm is run on the Bos Wars Test Set (defined in Section 3.4.3) with these parameters, these averages change to 88.3% and 89.9% respectively. A two-sample t-test yields a p-value of 0.000, giving significant evidence of a difference in these two means. The EA is able to find at least one solution with better fitness than the stochastic algorithm no matter what parameters are used.

The results on the Bos Wars data validate the effectiveness of the EA. However, they give no clue as to the best parameter values. In the next section, we will examine the results of the EA on the Spring data to see if this will give insight into better parameter values.

4.3.2 Spring Data.

Table 15 presents the average fitness of the unique solutions in the population and the max fitness of a solution in the population for the five parameter settings on the Spring data. Just as on the Bos Wars data, T-tests for each of the pairwise combinations give some evidence of a statistical difference in average fitness based on parameter settings. The lowest p-value is 0.003, found when comparing the $\lambda_r = 50$ and $\lambda_r = 250$ rows. This row has an actual difference in means

Table 15: Average fitness of the population and a standard deviation for various parameter settings on the Spring data with the memetic option. Averages are across the three folds. λ_r is the number of solutions created by recombination, λ_m is the number of solutions created by mutation.

λ_r	λ_m	Avg Fitness	St Dev
50	250	60.7%	6.9%
100	200	60.7%	7.0%
150	150	60.8%	7.0%
200	100	60.9%	7.0%
250	50	61.0%	7.0%

Table 16: Average fitness of the population and the average fitness of the best solution in the population at termination of the EA on Spring data with and without the memetic option. The standard deviation for average fitness is for the entire population across the five different parameter settings. The standard deviation for the max fitness is for best fitness found after each run across the five different parameter settings.

Search Option	Avg Fitness	St Dev	Max Fitness	St Dev
Memetic	60.6%	7.0%	71.5%	0.17%
No Memetic	55.2%	6.9%	69.1%	0.19%

of 0.3%, which is not a large difference in fitness. Again, there is no real effect on performance from changing the parameter settings.

Table 16 shows a comparison of the performance of the EA with and without the memetic option. Just as on the Bos Wars data, the memetic option is able to increase both the average and max fitness of the population upon search termination. Using a local search component injects more information into the population, allowing it to continue to find better solutions. A two-sample t-test for a difference in the average fitness yields a p-value of 0.000, giving significant statistical evidence of a difference in average fitness. The p-value for a difference in the max fitness also yields a p-value of 0.000, again giving significant statistical evidence of a difference in max fitness [62].

Figure 11 shows the convergence speed of the EA on the Spring data with the memetic option. Again, the recombination operator increases the convergence speed, just as in the Bos Wars data.

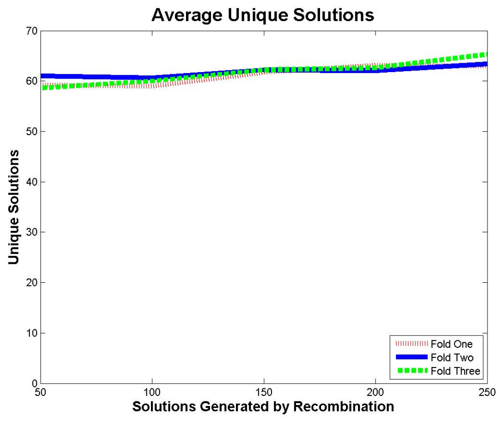


Figure 11: The effects of different parameter settings on the number of unique solutions in the population for EA searches on the Spring data. The horizontal axis shows the number of solutions created by recombination (λ_r). The number of solutions created by mutation is $\lambda_m = 300 - \lambda_r$. Each line represents a different fold of the data.

As there does not appear to be a large effect on fitness from different parameters, we choose the one which will minimize search time. The number of solutions created by recombination should be set to 250, and the number of solutions created by mutation should be set to 50.

While the classification accuracies of the solutions found on the Spring data are much lower than on the Bos Wars data, this is to be expected. Only about 30% of the Bos Wars data is winning samples, while 48% of the Spring data is winning samples. While the EA is able to obtain solutions with a fitness of 90% on the Bos Wars data, this is only a 20% improvement over an algorithm which simply predicts every sample is a losing sample. For the Spring data, similar results are found. Solutions with 70% accuracy can be found, which represents an 18% improvement over a losing prediction.

Additionally, there is much more variance in the Bos Wars data. While the Bos Wars agent matches tended to have the same outcome every time, there was much higher variation in the Spring data, showing it is a harder classification problem. The results on the Spring data are sufficient to validate the effectiveness of the created EA.

4.4 Conclusion

In this chapter, an evolutionary algorithm was developed and tailored to the RTSPP. Performance of the EA is comparable to the performance of the stochastic algorithm developed in Chapter III.

Additionally, the EA is not limited to values seen in the input game traces. It can use any real number as a feature value in a center, so it can better fit the solution to the problem.

In the next chapter, we take classifiers generated by the EA and use them as input to an agent. This agent will use the classifier to generate and execute a successful strategy.

V. Designing an RTS Agent

In the last chapter, an evolutionary algorithm was developed to solve the RTS Prediction Problem. The goal of the RTSPP is to generate a nearest neighbor classifier which can be used as the framework for an RTS agent. An agent which can appropriately use this classifier should be able to develop a strategy and implement appropriate tactics to defeat a scripted opponent.

This chapter outlines the different aspects of agent development required to test the overall hypothesis of this research, that an agent can generate strategies and appropriate tactics using a nearest neighbor classifier. Section 5.1 discusses an RTS platform, Spring, and different problems and decisions encountered by players and agents. The development of the Killer Bee Artificial Intelligence (KBAI) is presented in section 5.2. A methodology for testing KBAI's performance is outlined in Section 5.3. Finally, the results of this methodology are presented and analyzed in Section 5.4.

5.1 Spring

Spring is an open source RTS platform based on the commercial game Total Annihilation. Spring is maintained by freelance programmers in their freetime. Development is in C++, and most is done on a Windows platform. The Spring AI Interface is constructed to allow for quick and easy AI development. A more detailed description of the interface can be found in Appendix A.

Spring is setup to allow the user community to develop both maps and mods for the game. A mod determines the unit/building types which can be constructed: their strengths and weakness, costs, etc, as well as the way each looks. Giving the average Spring player this much control over the game leads to an extremely active user community.

Balanced Annihilation (BA) is one of the most popular Spring mods currently available. Set in a futuristic combat environment, the infantry units are "KBots", short for killer robots. Vehicles are more traditional, with both tanks and artillery vehicles available. All attack units may have either projectile or laser weapons.

BA has two different resources: energy and metal. Each unit costs a certain amount of energy and metal to produce, and upon completion may either produce or consume a certain amount as upkeep. Energy/metal production is expressed as a rate. A certain amount of energy and fuel can be stockpiled. If the amount of energy produced is greater than the amount consumed, this stockpile rises up to a certain cap. Conversely, the stockpile falls if the energy consumed is greater than the energy produced. Metal works the same way.

The object of a Balanced Annihilation game is to destroy the opponent's commander. Just like the King in Chess, all other units are secondary to the commander. It's death ends the game.

5.2 KBAI Development

In this section, the nearest neighbor classifier which serves as the foundation of the Killer Bee Artificial Intelligence is presented. Then, determining the correct set of actions to consider, along with a means of determining what buildings/units to produce and which units to target are described.[1]

5.2.1 State Representation. KBAI requires a compact representation of the state space of the RTS game. A full state representation that contains the quantity and owner of each unit type and each building type, their locations on the map, and a history of their actions would provide too large a space for efficient decision making. Instead, the state which KBAI uses is a vector of features F, where each feature is identified by the unit or building it pertains to, and the value of each feature is the difference between the number of that type that the agent owns has minus the number of that type that the opponent owns.

[1]Parts of this section adapted from [79]

In order to make decisions about what actions to pursue in the game, KBAI requires a nearest neighbor classifier that differentiates winning states from losing states. To do this, the classifier maps the values of the features of F to a set of winning centers W and losing centers L. Formally, the output of a nearest neighbor classifier consists of three different parts: a set of features F, a set of winning centers W and a set of losing centers L. A center is an exemplar of a winning or losing state in the domain.

The centers are labeled similarly: W_j is the jth winning center. Each center is a vector of feature values, one for each feature in F, so W_j^i is the value of the ith feature in the jth winning center.

The agent must reason on the three input sets F, W and L. It is also given a set of normalization parameters α, which are used to normalize the feature values of the current state to the interval $[-1, 1]$.

KBAI must make two types of decisions: military and economic. Military decisions determine when attacks should happen, the enemy units to target and the units to attack with. Economic decisions consider what units and buildings should be produced. When faced with a decision, KBAI determines the current state using the feature set and the normalization parameters and takes some action.

5.2.2 Determining the Active Feature.

A simple way to determine the appropriate action to take is to pick a feature and somehow attempt to increase its value, but how should this feature be selected? The agent knows the current state, which is the specific values of the features in F. It also knows good and bad value combinations for these features: the centers in W and L.

When choosing which feature to influence, the agent is trying to get from the current state S to some future state S' because it believes S' is the best possible future state. All the possible S' candidate states could be computed by determining the effect of focusing on a particular feature for some amount of time.

The next question is how to determine the effect on a feature's value if it is targeted for increase for fifteen seconds. The cost of each unit is different, and each takes a different amount of time to build. Since this is a stochastic domain, the agent cannot be assured it will be able to complete one unit or half a unit, or whether it could simply go destroy an enemy unit. The actual effect on a feature if it is targeted for increase is hard to estimate.

The normalization parameters in α add to the difficulty. The observed non-normalized values for the infantry units feature are actually integers on the range $[-21, 18]$, while the values for the construction vehicle feature are integers on the range $[-2, 2]$. Obviously, building a single construction vehicle can have a much greater impact on the state than building a single infantry unit.

The best approach is to determine some constant increase amount δ to use for each feature value, and then normalize it with the parameters in α to account for the relative weights. Then, the potential next state S' which could be attained if feature k was chosen as the active feature can be computed across all the features in S using the iterator l:

$$S'_l = \begin{cases} S_l & k \neq l \\ S_l + (\delta/\alpha_l) & k = l \end{cases}, l = 1..|F|$$

By using this equation, the agent can compute all the possible next states. To pick the best next state, and thus choose the active feature, some evaluation function must be used to determine the expected value of each S'. The centers give a natural way to do this: the candidate state which is closest to a center in W and farthest from the closest center in L is the best possible next state.

Distance from the ith candidate state to winning/losing center j is computed with a Euclidean distance function:

$$distance(S'_i, W_j) = \sqrt{\sum_{k=1}^{|F|}(S'_{i,k} - W_j^k)^2}$$

The closest losing center can be computed by swapping L for W. For each candidate state, the agent determines the distance to the closest winning/losing center:

$$w_{dist} = min_{i,j}(distance(S'_i, W_j)), j = 1..|W|, i = 1..|S'|$$

$$l_{dist} = min(distance(S'_i, L_j)), j = 1..|L|, i = 1..|S'|$$

The fitness of candidate state i is:

$$fitness_i = w_{dist} - l_{dist}$$

The candidate state i with the highest fitness is selected. Next, the agent determines the feature k in S'_i which is furthest below the values in W_j:

$$k = min(S_i - W_j^i), i = 1..|F| \tag{4}$$

Now k becomes the "active feature".

There may be a point where the feature values of the closest center are all less than the feature values of the current state. In this case, the active feature becomes the one which has the smallest difference between its value in the current state and the closest center using equation 4.

Finally, the time interval at which to reset the active feature must be determined. If this time period is too short, the agent may constantly switch between many different features, leading to action thrashing with no specific goal, as well as having to do computationally expensive calculations whenever a decision is needed. However, if the time period is too long, the agent may miss a drastic state change requiring different decisions. To balance these two concerns, the agent recomputes the active feature every fifteen seconds.

5.2.3 Economic Decisions. Many human players will confirm the most important aspect at the beginning of an RTS game is to build units as quickly as possible. The player who can produce units quicker often wins the game.

Agents are subject to the same problem: failing to focus on unit production at the beginning of a game results in a tactical disadvantage, which turns into a potentially disastrous strategic disadvantage as the game progresses. Often the game becomes unrecoverable by the time the agent realizes the problem and changes the active feature.

A naive approach to construction would simply build to increase the active feature: if the active feature is metal extractors, build metal extractors. If it is light laser turrets, build light laser turrets. Problems could occur when the agent focuses on non-unit features for long periods of time, like metal or energy production buildings, light laser towers or energy and metal usage/consumption. If these features compose a significant part of F, an agent which only produced non-unit production buildings would be vulnerable to an opponent. Along the same lines, while thrashing is reduced by only computing the active feature every fifteen seconds, there are still problems when different types of units are produced.

For example, a simple feature representation could contain three features: light infantry units (LIUs), metal extractors and light laser towers (LLTs). When LIUs are the active feature they are produced and, when there are enough, sent to attack the enemy. However, LIUs are extremely weak against LLTs; LLTs have much more powerful weapons and a greater range. When sent to attack an LLT, LIUs are often destroyed before they are in range to attack.

If LIUs are the active feature for four reasoning cycles, and then the active feature changes to LLTs, the agent's forces are quickly destroyed. Even if LLTs are not the active feature the opponent may have some, which destroy the agent's infantry units when they attempt to attack the opponent's base.

To prevent these problems, the agent must focus on unit construction right from the beginning. KBAI agents determine what type of units to produce as soon as it loads the classifier by looking at the features in F. The agent looks through the state representation for features which require heavy tanks. If any are present, then heavy tanks are selected as the production unit and the game begins. If no heavy tank features are in F, then the agent looks for features which require heavy infantry, then light tanks, and finally light infantry units.

Putting the possible unit construction types in this order balances the two goals of the agent: first, ensure the game is won. Second, win as quickly as possible. The lighter the unit produced, the faster it can be produced. The agent seeks to choose the lightest unit possible, while still providing enough firepower to fight off the opponent.

Once the production unit is determined, the agent produces as many of the appropriate unit production buildings as possible. It checks to make sure enough energy and metal are being produced to support another production building. If yes, then the production building is produced. If not, then energy or metal production are produced, then the agent checks again. This cycle continues throughout the game.

Any production buildings completed produce the chosen production unit until the game ends or they are destroyed.

5.2.4 Military Decisions. The agent's military decisions identify which of the opponent's units should be attacked first. In order to achieve the goal of winning efficiently, the agent should attack the opponent's "centers of gravity": the units/buildings whose destruction will have a large impact on the opposing strategy being pursued. For example, destroying resource production buildings could prevent the opponent from producing units by reducing the resources available for construction.

The agent has two different targeting scenarios: when its units can see some of the opponent's units and when they cannot. In the former case, the agent is in a situation where some of its units are already close to the enemy. In this case, a target should be selected from the units which are visible. This is done through a lookup table linked to the active feature. Each active feature has a target set which consists of the five possible attack units as well as a single building. The attack units are listed in some order in the lookup table based on the active feature: if the active feature is a weak building/unit, then the lighter units are listed first in the table, because they are more of a threat to this building type. If the active feature is a strong building/unit, then the stronger units are listed first. After the five attack units, a single building is listed. The agent will always target attack units before buildings. If none of the possible targets are visible, the agent will pick a visible unit at random as the target. All the agent's units attack the target until it is destroyed, then a new target is selected.

In the case where the agent can not see any units, it will use perfect information to get a list of all of the enemy's units. This list is searched for *strategic targets*. Strategic targets are based on the active feature: if the active feature is an infantry unit, the strategic target is infantry unit production buildings. If the active feature is metal production, then the strategic target is metal production buildings. The agent uses a three step targeting algorithm:

```
if There is a strategic target for the active feature then
    Target the closest one
else if There is a strategic target for any feature in $F$ then
    Target the closest one
else
    Target the opponent's commander
end if
```

The closest strategic target is computed by determining the center of mass for all of the units in the agent's attack group. This center of mass is just an average of the Euclidean locations for all of the units. The closest strategic target to the center of mass is selected as the target.

This targeting algorithm leverages the active feature along with the other features in F. The visible portion ensures that direct threats are dealt with first, while the non-visible portion ensures the agent pursues the overall strategy dictated by the classifier.

5.2.5 Discussion.

The agent is implemented based on this program refinement. The classifier is used to determine which units should be produced to end the game as quickly as possible with a high probability of success. The active feature is used to determine the direction in which the agent should attack. If the active feature has no strategic targets, then F' determines any other possible strategic targets.

With the design of KBAI complete, the next step is to determine how to measure its performance. In the next section, an experimental methodology is designed to accomplish this goal.

5.3 Evaluating Dynamic Agent Performance

Agent development has two goals: the agent should be both effective and efficient. The dynamic agent KBAI uses a classifier to guide economic and military decisions. In this section, the specific way in which the classifier was generated are discussed. Then, the experimental setup is described. Finally, the performance metrics for determining effectiveness and efficiency are defined.

5.3.1 Generating the Classifier.

KBAI's goal is to use a classifier to learn how to beat a particular scripted opponent. It pursues strategies which have been proven successful against the script. To facilitate this, the data used to generate the model must come from games which involve the particular scripted opponent, and this opponent must be in the player two position, so a winning sample represents a strategy which was successful against the agent and a losing sample represents a strategy which was unsuccessful.

The data is the same as in Section 4.2.2. It is split into four sets, one for each of the scripted agents, where each set consists of all the games which were played by the studied agent in the player two position. There is no reason to hold out data, so all the data from each set is used to generate an appropriate classifier. For each agent, their are twenty-four games from which to learn. Each KBAI agent uses classifiers generated from the appropriate set using the evolutionary algorithm developed in Chapter IV. This classifier is the best one in the population after a single run of the algorithm on the data set, using the parameters determined by an analysis of the experimental results from Section 4.3. Initial experimental results on classifiers generated from the third quarter of RTS game trace data were unimpressive, so all classifiers used by agents were instead generated from the second quarter of Spring game trace data. A complete exploration of this issue and possible ways to solve it in future research can be found in Appendix C.

5.3.2 Experiments. KBAI performance is evaluated by running a fifty game match against four different scripted opponents: Infantry Rush, Tank Rush, Blitz and Turtle. KBAI uses a different classifier for each opponent.

For comparison, three other dynamic agents are also tested against the scripted opponents: Dynamic Random Agent (DRA), KAI and RAI (all described below). Each comparison agent is tested in a fifty game match against each of the scripted agents.

None of the comparison agents are learning agents. This leads to problems with the comparisons. KBAI is able to learn from the opponent's past behavior to develop a strategy; none of the other agents is able to change its behavior. A complete discussion of our reasons for choosing these comparison agents is contained in Appendix D.

However, our hypothesis is an agent can use a nearest neighbor classifier to develop a strategy and the appropriate tactics for its execution, outperforming an agent which only makes tactical decisions. Both KAI and RAI are well suited to validate this hypothesis: they make tactical decisions in the environment because they can not develop an overall strategy by learning. Their developers have examined and solved each problem faced by an RTS agent based only on the current state. Sophisticated pathfinding, group movement, attack formation, build decision and placement, and targeting algorithms are present in both KAI and RAI.

5.3.2.1 The Scripted Opponents. Testing the performance of KBAI requires some scripted opponents. To facilitate this, four different scripted agents were developed: Infantry Rush, Tank Rush, Blitz and Turtle.

The **infantry rush** is a very simple strategy. The goal of an infantry rush is to quickly mass produce cheap infantry units, and then use these to attack the opponent before he is ready to defend. An initial group of infantry units prevents the opponent from creating an economy, and then additional groups of attackers are used to follow up on this advantage and complete the victory. The infantry rush, in fact any rush strategy, is very dependent on the success of its first few waves of units. If it can not significantly impair the initial production capabilities of the opponent, it usually loses. The opponent builds stronger units, which can destroy the weaker units produced by the infantry rush. Eventually, the infantry rush is overwhelmed.

In a **tank rush**, the goal is similar to an infantry rush. The agent attempts to quickly build units and attack the enemy. However, as opposed to an infantry rush, the agent builds more powerful units; while some opponents are prepared for a simple infantry rush, they may not be able to handle the more powerful onslaught of a tank rush. Tank rush has the same weakness as infantry rush: it is very dependent on the success of the first few waves.

In a **blitz** strategy, the goal is to create an unstoppable army. The agent builds as many troops as possible, but does not attack until the army is sufficiently large that it can do serious damage to the opponent. Initial unit production is directed towards weak units, then stronger units as the game progresses. Each successive blitz wave is stronger. A blitz strategy is initially vulnerable to a rush attack. However, if it can fight off the first few rush attacks, it overwhelms the opponent through sheer power.

A **turtle** strategy builds defensive buildings, such as laser towers, along with general assault units. The agent waits for the opponent to attack first, in the hopes of using the defensive buildings to destroy the attacking units, and then sending its own assault units to quickly follow up and destroy the enemy. The turtle strategy is less vulnerable to a rush. The light laser towers prevent the relatively weak rush units from doing much damage. It is, however, vulnerable to a blitz attack. The light laser towers cannot hold off a strong attack.

All the scripted opponents use a simple targeting algorithm: if no units are visible, the enemy unit closest to the script's commander is targeted. If units are visible, the target is selected at random from the visible units. Building placement is also random: the nearest suitable build site to the commander or construction vehicle which is far enough away from the other buildings is selected. There is no group movement algorithm, so any units ordered to attack take the shortest possible route to the target. Slower units take longer to reach the target.

A more detailed descripted of the pathfinding, targeting and group movement algorithms and each scripted agent can be found in Appendix B.

5.3.2.2 Dynamic Random Agent. We developed DRA to assess the impact of the KBAI architecture on agent performance, as compared to the impact of decisions made using the classifier. DRA uses the same underlying architecture as KBAI, but makes a random choice whenever KBAI would reason on the current state and the nearest neighbor classifier.

DRA randomly selects one of the four unit types to produce at the beginning of the game from a uniform distribution; just as in KBAI, this unit type will be produced throughout the game. Where KBAI uses the active feature to determine the next target, DRA selects the next target randomly from the available targets. If any enemy units are visible, it randomly selects one of these. If none are visible, it randomly selects a target from all the enemy units.

5.3.2.3 KAI & RAI. KAI, a perfect information Spring agent, was developed in [74]. KAI comes preloaded with the Spring game. It uses the concept of a threat map to determine where it should attack the opponent, and does sophisticated reasoning to determine what type of units should be produced to counter the units being produced by the opponent. For group movement, a flocking algorithm is used to ensure all KAI's attacking units remain close together.

RAI is an imperfect information Spring agent which also comes preloaded with the Spring game [63]. RAI documentation is not as complete as for KAI, however, it appears RAI also uses a threat map and similar reasoning to determine the correct units to produce.

We restrict KAI and RAI to prevent them from building Aircraft. During testing, KAI and RAI matches crashed over 80% of the time when allowed to build aircraft. The flocking algorithm used for group movement is extremely computationally expensive. Aircraft travel quickly, requiring frequent updates and overtaxing the processor until the game fails. Other than this restriction, KAI and RAI are allowed to build any unit/building type available in the game. In all matches, KAI and RAI difficulty was set to the highest possible setting, five out of five.

5.3.3 Performance Metrics. Performance metrics are collected to measure the effectiveness and efficiency of each dynamic agent against the four scripted agents. Effectiveness is measured by win percentage in each fifty-game match. Efficiency is measured by examining the length of games with successful outcomes for the dynamic agent.

A higher win percentage means the agent is more effective. For an agent to be considered at all effective, it must win 60% of the games in its fifty game match. Effectiveness is the most important factor for an agent.

If an agent is effective, its next goal is to be efficient. If the agent can win faster than some other equally effective agent, then it is more efficient. The length of winning games is recorded at the end of each game. The mean game length and standard deviation of successful games for each effective agent is calculated. A two-sample t-test is used to compare the efficiency of agents.

5.3.4 Conclusion. This section outlined a set of experiments which can be used to measure the performance of KBAI. In the next section, the results of the experiments are presented and analyzed.

5.4 The Killer Bee Artificial Intelligence Agent

KBAI has two goals: to be effective and efficient. Effectiveness is measured by win percentage, while efficiency is measured by time to win.

Because KBAI relies on a classifier generated from a stochastic search algorithm, the first part of this section will look at the performance of KBAI on three different generated classifiers. We show that similar performance is obtained from each generated classifier, demonstrating the robustness of our method. Next, the results of KBAI are compared to the results of other agents.

5.4.1 KBAI Consistency. In this section, KBAI is tested using three different classifiers for each script, generated from game traces where the opposing script was player two. Table 17 presents the classification accuracy of each classifier. Each classifier was the top performing classifier found at the end of a complete search.

The results of a fifty game match show all the classifiers are equally effective. KBAI won 100% of the games played for every classifier. To measure efficiency, Figure 12 shows a 95% confidence interval for the game length of each classifier against

Table 17: Classification accuracies for the classifiers used in KBAI testing. Each classifier is the top performing classifier generated during a single run of the evolutionary algorithm on the appropriate script's dataset.

Classifier/Script	Blitz	Infantry Rush	Tank Rush	Turtle
One	68.3%	65.8%	69.0%	76.1%
Two	68.9%	68.0%	67.6%	76.6%
Three	70.2%	69.3%	66.1%	76.0%

the appropriate scripted agent. In some cases, the standard deviation is different, which results in different interval widths. However, a two-sample t-test for a difference in mean game length yields p-values $\geq .1$ except in two cases: classifier three on Tank Rush and two on Turtle. These p-values show there is no evidence of a difference in mean-game length, except in two cases. On Tank Rush, classifier three is statistically different than one and two, the p-values in both cases are 0.000. For turtle, classifier two is different than classifier one (p-value of .007), but neither is different from three.

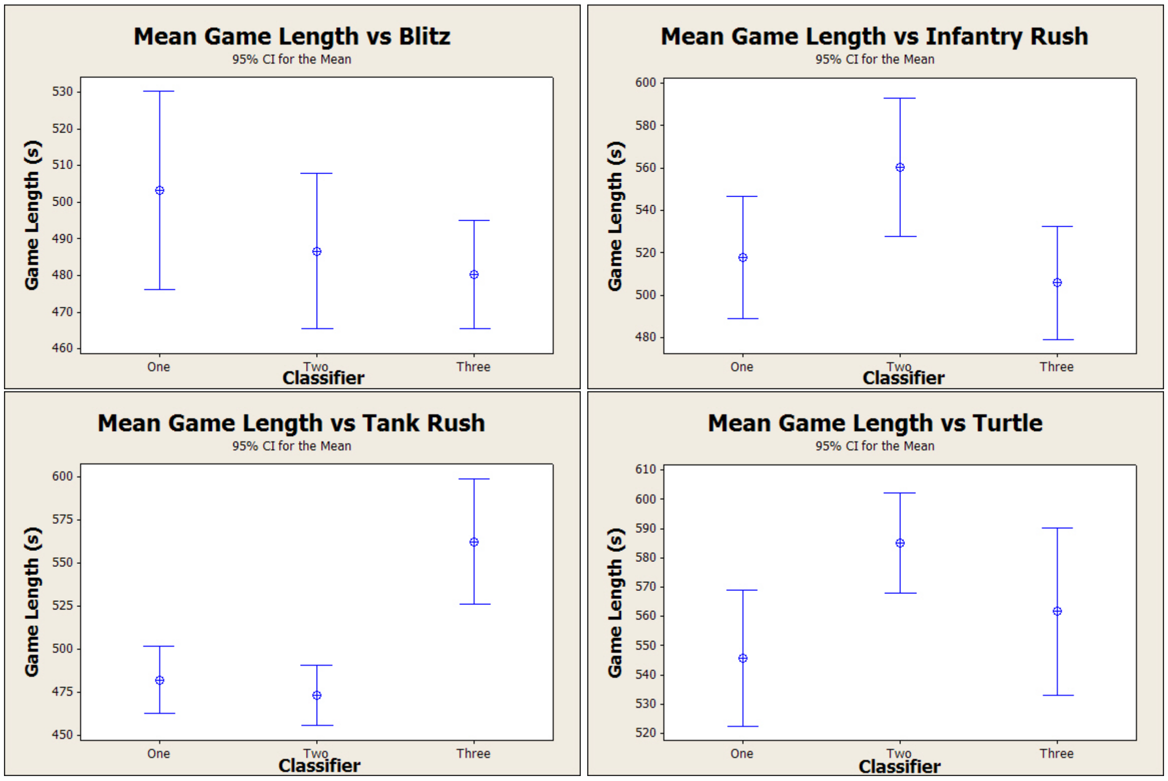


Figure 12: 95% Confidence Intervals for KBAI's mean game length for each classifier.

While statistical differences in mean game lengths for classifiers on Tank Rush and Turtle show that some classifiers are more efficient, the results in the next section, which discusses the performance of other dynamic agents against the scripts, validates the better efficiency of all the classifiers used.

KBAI never loses a game, no matter which classifier it uses. Additionally, its performance is similar, no matter which classifier it uses. The underlying architecture, which depends on the stochastic evolutionary algorithm, is able to take the input classifier and execute the appropriate strategy.

5.4.2 KBAI Versus Comparison Agents. In this section of KBAI's performance analysis, we show KBAI is superior to both a random agent using the same undelying architecture as KBAI as well as two other dynamic agents currently available for Spring. These two dynamic agents are presented as a comparison to KBAI's method because they are available with every Spring distribution.

Table 18 shows the winning percentage for all agents versus the scripted agents. KBAI clearly outperforms all other dynamic agents, winning 100% of the time against all the scripted agents. The only other effective dynamic agent is DRA, which uses the same framework as KBAI: it builds as many units as it can as fast as possible. As discussed earlier, this is generally the determining factor of an RTS game.

Table 18: Dynamic agent win rate against the four scripted agents (50 games for each combination).

Agent	Blitz	Infantry Rush	Tank Rush	Turtle
KBAI	100%	100%	100%	100%
DRA	74%	90%	68%	70%
KAI	36%	18%	30%	38%
RAI	8%	66%	14%	20%

If an agent is effective, is it also efficient? An effective agent is one which wins at least 60% of its games. Table 19 has the mean game length for the effective dynamic agents, calculated only from games which the dynamic agent won. KBAI has a lower mean game length against all opponents. In Figure 13, the 95% confidence interval for game length of KBAI is compared to that of DRA. Not only is the CI for KBAI lower, it encompasses a much smaller range. For RAI vs. Infantry Rush, the only case where the other two dynamic agents were effective against a script, the mean game length is much higher than for KBAI and DRA. Although RAI seems to be moderately effective against Infantry Rush, it is certainly not efficient.

Table 19: Mean time to win for the four dynamic agents versus the four scripted agents. Units are seconds of game time, and all measurements are rounded to the nearest second. Results are only displayed for effective agents: those with a winning percentage of at least 60%. The asterisks mean the agent was not effective against the script.

Agent	Blitz	IR	TR	Turtle
KBAI	507	540	512	548
DRA	604	750	724	905
KAI	*	*	*	*
RAI	*	1617	*	*

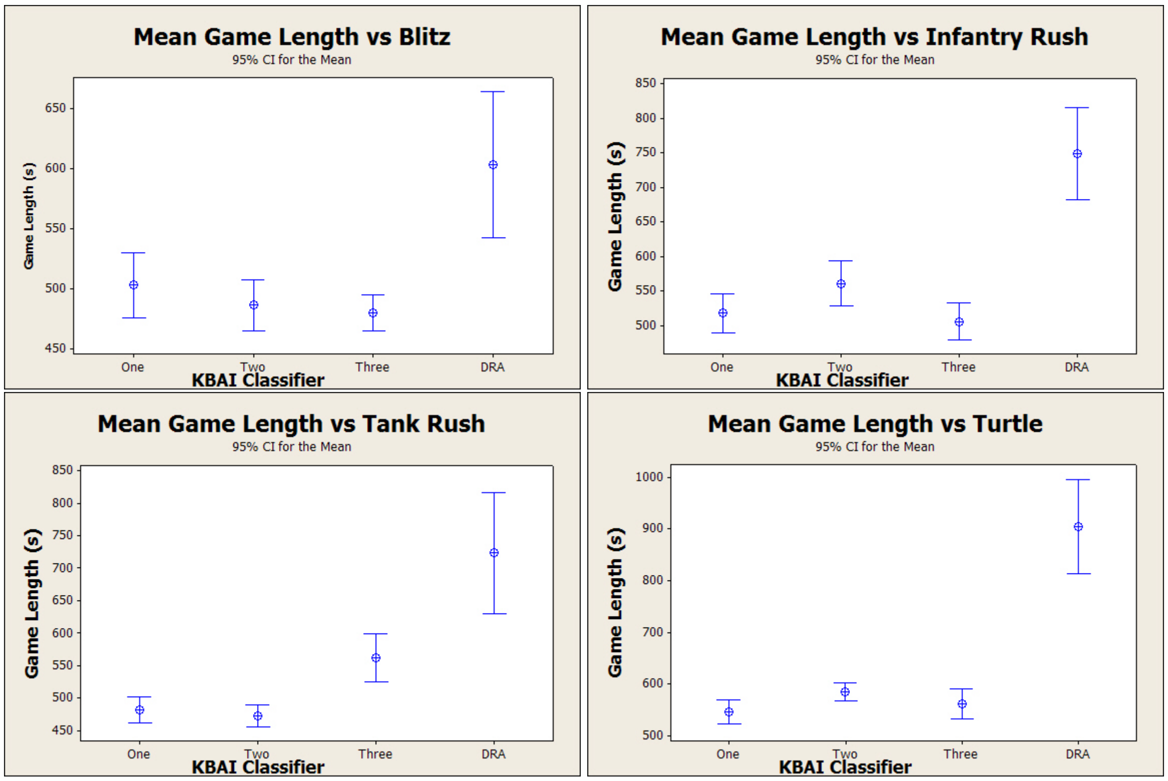


Figure 13: Interval plots for all three KBAI classifier and DRA game length against the four scripted agents.

The experimental results verify that KBAI is both effective and efficient in combatting the scripted opponents. DRA is still able to win a majority of the time, even though it has an unguided targeting scheme and uninformed unit selection. This is due to dynamic resource allocation: if a scripted agent loses infrastructure, it can not go back and rebuild. It continues with its build script, with production hampered because of a lack of resources. If DRA loses infrastructure, it realizes that resources are low and builds more infrastructure, ensuring a steady stream of attack units are available.

However, the lack of informed targeting goals in DRA can lead to long wars of attrition. KBAI has a predictable game length: its mean confidence interval is lower and smaller than DRAs in all cases, showing there is much less variation in game length for KBAI. A two sample t-test for a difference in the mean game length for DRA and KBAI leads to p-values ≤ 0.005, giving strong statistical evidence of a difference in mean game-length [62].

KAI and RAI are mostly ineffective against the scripts: seven out of the eight matches yield win percentages below 40%. These results are caused because both fail to focus on a particular strategy. Multiple unit types are produced and used to attack, and significant resources are expended on production units at the beginning of the game. This allows the scripted agents to gain a unit advantage. When attacking, RAI and KAI look for the weakest point of a base to attack. However, this may not be the right location: if the targets in this sector were important, they would be protected.

KBAI outperforms all of the other tested dynamic agents in terms of both effectiveness and efficiency. It is a robust system which can be used to defeat different types of scripted agent, without the weakness to rush strategies of reinforcement learning approaches outlined in Section 2.1.1.

5.5 Conclusion

In this chapter, an agent which makes decisions based on a generated nearest neighbor classification model was developed and tested. The results validate our hypothesis: an agent can use a nearest neighbor classifier to develop a strategy and the appropriate tactics for its execution, outperforming an agent which only makes tactical decisions.

In the next chapter, the contributions of this work to the military and the overall AI field is discussed, as well as areas of future work.

VI. Conclusion

This research is conducted to validate the hypothesis that an agent can use a nearest neighbor classifier to develop a strategy and the appropriate tactics for its execution, outperforming an agent which only makes tactical decisions. In Chapter III we formally declare the specific search problem which must be solved to generate the classifier. A deterministic and a stochastic algorithm are developed and evaluated to determine which is better suited to the RTS domain. The stochastic algorithm has better classification performance on actual RTS game trace data than a deterministic algorithm which searches a reduced solution space. An attempt to use the Bhattacharrya Coefficient as a heuristic to guide a deterministic search was ineffective, showing features in the RTS domain are dependent. Analysis of the performance of the two algorithms leads to a conjecture about the characteristics of the solution space: it is jagged, with many local maximums with fitness similar to the global maximum. These characteristics are used to create an algorithm which can develop a nearest neighbor classifier in the RTS domain, required to validate our hypothesis.

In Chapter IV we develop and tailor an algorithm to take advantage of the discovered characteristics of the RTS domain and solution space. An evolutionary algorithm (EA) was selected and tailored because of the population of solutions created by an EA, which are used to explore the different areas of the solution space, finding multiple local maximums. This algorithm is tested on data sets from two different RTS platforms, demonstrating the ability to generate a classifier for the RTS domain.

In Chapter V we take classifiers generated with the evolutionary algorithm and design the Killer Bee Artificial Intelligence (KBAI) RTS agent. KBAI outperforms three other RTS agents in terms of both effectiveness and efficiency. It is able to beat rush opponents, unlike traditional learning-based agent approaches (as discussed in Chapter II), and is able to develop different strategies to suit the input classifier. KBAI's performance is superior to the performance of native Spring AI, validating our hypothesis. This research has uses in both the military domain and the broader field of Artificial Intelligence.

6.1 Artificial Intelligence Impact

This research has two portions: classification and agent development. In classification, we reformulate the traditional nearest neighbor classification problem so its output is useful in RTS agent development.

Three different algorithms are presented to solve this classification problem, taking an embedded approach to feature selection and classification. We believe the best features and their representation are problems which must be solved at the same time in the RTS domain. Two different data sets are created to test our hypothesis. The performance of the evolutionary algorithm on both was good. However, since these are new data sets, it would be hard to say whether performance is better or worse than other methods. We hope that the successful application of our solution to agent development will spur further research in this area.

In agent development, KBAI, a new type of RTS learning agent was created. Instead of attempting multiple different action scripts or copying strategies obtained from a case base, KBAI uses information from other games which the opponent has played. It takes this data, runs the developed classification algorithm on it, and uses the output nearest neighbor classifier to direct and execute a winning strategy. Success is obtained against multiple different strategies, showing the robustness of the KBAI system. Where current approaches have struggled with rush strategies, KBAI is able to develop an appropriate strategy for all the opponents it faced. It is dependent on info from the problem domain, but learning is done offline before actually playing against the adversary.

KBAI, which relies on the classification model, outperforms two native agents packaged with the Spring distribution. It is effective and efficient, able to generate a strategy and execute the tactics to achieve good performance over the four different scripts used as opponents.

6.2 Military Impact

As discussed in our literature review, Computer Generated Forces can be developed and tested in the RTS domain. For CGF to be useful, they must be able to act as a human would when presented with the same problem [48].

KBAI generates a strategy by examining the past performance of an opponent script. It determines the important characteristics of an opponent and uses these characteristics to develop a strategy. This mirrors military doctrine, as laid out in the Joint Planning Process [43]. The classifier algorithm is used to determine a strategy, and then KBAI determines the tactics required to achieve it. It is able to effectively and efficiently control the military and economic decisions of a simulated theater level force.

CGF often require human controllers at some level because of their inability to exhibit strategic behavior. While current approaches can make good low-level tactical decisions, their overall strategy must be guided by a human. KBAI is able to make a CGF which can exhibit strategic behavior at the operational level of war. This strategic ability can be used to significantly reduce the required human presence.

6.3 Future Work

There are many areas for future work, both in the classification problem and the agent development portion.

For classification, we have illuminated some of the RTS problem domain characteristics. However, there is still much left to be done. While we examined two different RTS platforms, there are many others. If other platforms can be found which have similar characteristics, where feature combinations can be used to predict game outcome, the same methods could be applied.

Additionally, there are many other algorithm families which could be applied to the classification problem. We developed a novel evolutionary algorithm, but other classification approaches could also be used such as principal component analysis or linear discriminant analysis.

All the algorithms focused on data generated from a specific time-period of each RTS game, based on an assumption about where the important decisions in an RTS game are made. This assumption may not be valid, and should be tested. Determining the correct portion of the RTS games and how wide this portion should be is an interesting problem. An in-depth exploration of this problem is presented in Appendix C.

KBAI is a new approach to RTS agent development, different from current approaches. Determining whether our approach is superior to other learning approaches and under what assumptions and limitations is an area of future work. KBAI was successful against scripted agents when given twenty-four games to learn from. However, whether it would be successful allowed more learning information, or what the effect of limited learning information would be on the developed agent is also interesting. Potential methods of approaching this problem are discussed in Appendix D.

In agent development, we still seek to combat more dynamic agents. The implemented Spring scripts do not look at the current state when making economic decisions, and they have a simple targeting algorithm. Refining KBAI so it can combat more sophisticated agents is a possible next step.

Finally, testing whether KBAI could learn from its own games would allow it to improve its results online. This would increase KBAI's robustness and potentially make it a more useful training tool by allowing it to adapt to new strategies tried by a human opponent during a game.

Appendix A. Spring AI Interface

Spring comes with a well-defined AI interface, allowing for quick and relatively painless agent development. An agent is compiled into a linked library file, which the Spring game interacts with during the game.

Spring has AI interfaces called wrappers for both Java [78] and C++ [77]. The C++ wrapper was used for all agents in this research.

The C++ wrapper provides two way communication between the current game and the agent. Game to agent communication is done through a series of notification functions. The agent is notified when:

1. A new game is started
2. A new unit is created
3. Construction of a unit is completed
4. A unit is idle
5. A unit is destroyed
6. A unit is damaged
7. An enemy unit enters the line of sight (LOS) of one of the agent's units
8. An enemy unit is destroyed
9. An enemy unit is damaged
10. A new game cycle has started (thirty times/second)

All these functions provide some information to the agent. If the notification is about a specific unit being created or finished, a reference to the unit is provided. If the notification also pertains to the enemy, like one of the agents units was damaged by the enemy, a reference to the enemy unit is also provided.

The agent is notified at the start of every game cycle through the *Update* function, so if there are certain actions which should be taken at some regular interval, like computing the state, these can be implemented in the *Update* function.

Communication from the agent to the game is though the callback interface, which can be used to issue commands to the agent's units or to obtain information about the game state. This callback is implemented as an imperfect information interface: the agent can only obtain information about things which are within LOS of one of it's units. However, the agent can use a cheat interface to get information about any units over which it does not have LOS.

Agents are coded by determining what actions should be taken when events happen. The agent does not have to deal with every notification. The wrapper provides the capability, whether used or not.

Appendix B. Spring Scripted Agent Descriptions

Four different scripts were developed to test the performance of an agent which uses a classifier to make strategies and tactical decisions. This section outlines some of the algorithms which are the same for each agent, and then briefly outlines the different strategy pursed by each of the four scripted agents.

B.1 Pathfinding

Spring provides a simple pathfinding algorithm. When a unit is given a command to move, it determines the shortest straight line path and moves in that direction. If an obstacle is reached, then the unit tries to go around the obstacle and then continue on the shortest path to the destination.

The test map has no large mountain ranges or other barriers, so this pathfinding algorithm works well.

B.2 Targeting

Targeting is done based on the greatest threat to the team. Spring games end when a team's commander is destroyed, so the greatest threat is the unit closest to the agent's commander. Anytime a unit is selected to attack, the distance from the commander to every enemy unit is computed, and the closest unit is selected as the target. All attacking units attack the target until it is destroyed. When the target is destroyed, the new enemy unit closest to the commander is selected as the new target.

This is an expensive calculation which can slow down the game if done during large scale battles. To deal with this problem the agent checks to see if any of its units have LOS on any enemy units. If they do, then the agent attacks the first enemy seen. If it can not see any enemies, then the full calculation is done. During a large engagement, this speeds up the determination of the next target, significantly reducing the computational time required.

B.3 Group Movement/Attack Formations

Group movement is the way units attacking at the same time move. Should the faster units wait for the slower units? Should light units stay behind heavy units?

There is no sophisticated group movement algorithm implemented. Units travel as fast as they possibly can, and attack as soon as they arrive at the target. Slower moving units get to the target later than fast moving units.

Attack formations are a similar concept, only they specifically deal with where units place themselves when attacking the enemy. Units are static in an attack formation. There is no sophisticated maneouvering algorithm. Units fire as soon as they are in range of their target. They do not move inside their maximum range to the target, but do not maneuover to stay outside the target's maximum range (if it is less). Units with smaller ranges must move closer to their target. Since light units tend to have shorter ranges, this leads to attack formations with small units closest to the target and heavy units farthest away.

B.4 Scripted Agent Descriptions

With the constant algorithms specified, a Spring agent script can be fully specified by determining what buildings the production units will produce, which units the production buildings will produce, and how many units will be used to compose a group.

Four scripted agents were created for experimental purposes. In this section, the general strategy pursued by each agent is described along with a description of the units and buildings it produces. Production is split into two categories: vehicles and buildings. Production vehicles produce buildings, while production buildings produce units.

B.4.1 Infantry Rush. The infantry rush is a very simple strategy. The goal of an infantry rush is to quickly mass produce cheap infantry units, and then use these to attack the opponent before he is ready to defend. An initial group of infantry units prevents the opponent from creating an economy, and then additional groups of attackers are used to follow up on this advantage and complete the victory.

This script is implemented using a single construction unit, the commander. The first building constructed is a K-Bot Lab, then enough power stations and energy production facilities to support it, then another K-Bot Lab, and so on. This continues until the game ends. The infantry rush attacks in groups of six units. As soon as a group of six units is completed it is ordered to attack.

The infantry rush, in fact any rush strategy, is very dependent on the success of its first few waves of units. If it can not significantly impair the initial production capabilities of the opponent, it usually loses. The opponent builds stronger units, which can destroy the weaker units produced by the infantry rush. Eventually, the infantry rush is overwhelmed.

B.4.2 Tank Rush. In a tank rush, the goal is similar to an infantry rush. The agent attempts to quickly build units and attack the enemy. However, as opposed to an infantry rush, the agent builds more powerful units; while some opponents are prepared for a simple infantry rush, they may not be able to handle the more powerful onslaught of a tank rush.

More powerful units require greater infrastructure support. The Tank Rush agent must initially allocate some resources to increase infrastructure support. Instead of using a single construction unit, the first unit produced is a construction vehicle, which can be used to augment the construction capabilities of the commander. The commander builds some energy/metal production buildings, then a vehicle plant. Then more energy/metal production, another vehicle plant and so on.

As soon as the first vehicle plant is completed it produces a construction vehicle and then some light tanks. After it produces ten light tanks, it switches to medium tanks and produces these for the rest of the game. Any subsequent vehicle plants completed produce medium tanks. The construction vehicle builds high end energy production buildings and vehicle plants.

The production of stronger units leads to smaller group sizes. The Tank Rush agent forms groups of four, and attacks as soon as a group is completed. Tank rush has the same weakness as infantry rush: it is very dependent on the success of the first few waves.

B.4.3 Blitz Agent. In a blitz strategy, the goal is to create an unstoppable army. The agent builds as many troops as possible, but does not attack until the army is sufficiently large that it can do serious damage to the opponent.

This script has two construction units, the commander and a construction Vehicle, which is produced later than in the Tank Rush script. The commander builds energy/metal production buildings, a KBot lab, more energy/metal production, and a vehicle plant. It then continues to produce energy/metal production and vehicle plants/KBot labs. A construction vehicle is produced by the first vehicle plant completed. The construction vehicle produces advanced solar collectors and advanced vehicle plants.

Three different production buildings are created: KBot labs, vehicle plants and advanced vehicle plants. KBot labs produce ten small infantry units, and then medium infantry units. Vehicle plants produce ten light tanks and then medium tanks. The first vehicle plant produced creates the construction vehicle as the first unit in its build queue. Advanced vehicle factories produce heavy tanks.

The script attacks in waves of nine assault units. Additionally, the production buildings initially build light assault units and then transition to medium/heavy assault units. This means each successive wave is stronger.

A blitz strategy is initially vulnerable to a rush attack. However, if it can fight off the first few waves, it overwhelms the opponent through sheer power.

B.4.4 Turtle Agent. A turtle strategy builds defensive buildings, such as laser towers, along with general assault units. The agent waits for the opponent to attack first, in the hopes of using the defensive buildings to destroy the attacking units, and then sending its own assault units to quickly follow up and destroy the enemy.

As in the blitz strategy there are two production vehicles, the commander and a construction vehicle. The commander produces energy/metal production, a KBot lab, more infrastructure, a vehicle lab, then metal production buildings and light laser towers in a ratio of three to one. Once the construction vehicle is finished it produces advanced solar collectors to support the high energy consumption needs of the light laser towers.

Only two production buildings are created. The KBot lab produces ten light infantry units and then medium infantry units. The Vehicle Plant produces a construction vehicle, ten light tanks and then medium tanks.

Turtle holds all its units until attacked, then begins a counterattack as soon as one of its units/buildings is damaged. Every unit finished after this point is sent to attack the enemy, along with all of the initially held units.

The turtle strategy is less vulnerable to a rush. The light laser towers prevent the relatively weak rush units from doing much damage. It is, however, vulnerable to a blitz attack. The light laser towers can not hold off a strong attack.

Appendix C. Finding the Turning Point

The goal of classifier generation in our research is to capture the "turning point" of an RTS game, the critical decisions or actions taken by an agent which allowed it to defeat it's opponent or caused it to be defeated. At the beginning of our research, we assumed the turning point was contained in the third quarter of RTS games, but agent performance using a classifier generated from this quarter was sub-standard. The agent failed to win a majority of the time in a small pilot test of around ten games. As a result, we generated classifiers from the second quarter of games and used this as the foundation of KBAI agents. These agents were able to win 100% of the time.

This performance leads to the conclusion that the turning point of an RTS game is not contained in the second half of the game. However, we can not say this is always the case. Each agent was allowed to generate a classifier from information on twenty-four different games. It may be that the turning point for a majority of these was contained in the second quarter, but some were not.

To find the actual turning point, the performance of agents using classifiers generated from different portions of RTS games must be compared. When we wanted to gauge the performance of an agent, we ran fifty game matches against the particular opponent under test. Even when running at a high speed, it can take more than six hours to complete fifty games. Since each classifier is generated through a stochastic process, the performance of agents different classifiers from each section should be averaged. Clearly, this becomes a time-consuming process.

Isolating the turning point of an RTS game may also be dependent on the particular opponent which is being studied. Again, testing this question would require many matches and classifiers. Additionally, turning points may change based on map size, resource availability or some other unknown variable.

Finally, we assumed 25% of a game is sufficient to capture all the turning points. This assumption may have over or undershot the actual value. Perhaps all the turning points for a specific agent are contained in only 10% of the time, or are so varied they occur in 50% of the time. This question is subject to the same issues: the size of the portion may be dependent on other variables. Fully exploring and testing these questions are left as future work.

Appendix D. Finding Comparison Agents for RTS Research

When performing research in RTS agent development, there appears to be no consensus in the field on exactly how to present and compare results. In this appendix, we look at the strengths and weaknesses of three different methods of assessing RTS agent performance: comparisons to other approaches, comparisons to random agents and combatting native AIs. Finally, we conclude with a summary of our method of assessing performance and our reasons for doing so.

D.1 Comparing to Another Approach

One of the big decisions which must be made before embarking on RTS agent research is in the choice of platform. Numerous platforms are available and have been used by researchers, including Bos Wars, Spring, ORTS, WARGUS and madRTS. All have different agent interfaces, strengths and weaknesses. Some were designed specifically for research, others are ports of commercial games or were developed by free-lance programmers as an alternative to expensive commercial games.

When choosing an RTS platform, researchers may be guided by their capabilities in specific programming languages, or they may be directed towards a specific platform because their group has already done research on that platform and tools for agent interface have already been developed.

Kok developed a Monte-Carlo reinforcement learning algorithm in the game Bos Wars. When selecting a platform, ORTS was also examined but discarded because of “the lack of a fully functioning agent to start development with and compare results to” [46]. Bos Wars was selected because it had a native AI, one which relied on scripts for tactical decisions. To get an agent into Bos Wars, an interface was written between the 2apl agent programming language [21] and Bos Wars. For a comparison agent, Spronck’s Dynamic Scripting (DS) was implemented in Bos Wars [72]. Dynamic Scripting was originally implemented in WARGUS, and the port to Bos Wars “provided some challenges in resolving model incompatibilities” [46].

In the RTS field, this is the only current paper which compared two approaches. The problems with this method of validating results are in the different platforms and in agent complexity. Because he used a different platform, Kok mentioned having to change DS to make it work in his research. The original creator of DS was not consulted, and the fairness of Kok's implementation can not be verified.

Complexity is always a problem for RTS agents. The action space of an RTS is huge: each unit can move to almost any location on the map and take many different ways to get there, and may attack, defend, or produce new buildings. Units can be grouped, attacks can happen in many different locations. All RTS agents must, at some point, place a limit on the exact actions over which they will reason. The pathfinding, movement, group formation and building placement algorithms are generally not part of a learning agent's decision space, instead implemented as external functions which the agent calls. However, these algorithms are generally different from agent to agent, making it hard to determine their effect on overall game outcome. Most RTS players would agree these four factors of an RTS game, if done correctly, can have a significant impact on the game. When attempting to recreate a researcher's approach, these algorithms can change the effectiveness of the approach.

These two concerns are the reason most researchers do not choose to compare to other approaches in the field. It is very hard to make this a fair comparison, and will remain so until all research is done on a single platform and agent code is made widely available.

D.2 Comparing to a Random Agent

Some researchers choose to compare their agent to a random agent. This random agent must use the same framework as the created agent, but makes a random choice when the created agent would make an informed choice. The researcher can conclude his/her approach is superior to a completely uninformed decision, and it allows them to show the impact of their architecture on their success. Comparing to a random agent removes the concern discussed in the previous section about the effect of the external algorithms which are left out of an agent's reasoning space. The random agent is allowed to use these algorithms.

Chung et al designed a Monte-Carlo agent which generated and tested plans in the RTS game ORTS [17]. Actions were selected for plan inclusion based on their associated fitness value, which was modified after each game. The random agent used for comparison selected actions with no regard for their value. Chung was able to conclude his agent outperformed the random agent in most cases, and then focused on the effect of changing the different parameters used as input to his agents.

We compare KBAI agents to DRA, a random agent using the same architecture, in Chapter V. This allows us to show the impact of the domain knowledge included in the architecture on the success of KBAI agents.

Comparing to a random agent allows the researcher to show informed choice is better than uninformed choice using his framework. Additionally, it removes the effect of external algorithms. However, he can not say his approach is better than any other out there.

D.3 Combatting Native AI

Native AI is the term used when referring to the computer opponent made available to a RTS game user with the original distribution of a game. In Bos Wars and WARGUS, the native AI is mostly script based. The Spring distribution comes packaged with various dynamic agents, including KAI, RAI and AAI. Spring is frequently updated, and agent development is done by a different group of programmers, so often their is a lag between game updates and agent updates, meaning the old agent's may not work when a new game update is released. Many Spring agents are referred to as "legacy": development on them has ceased, and the agent's will not work with the current distribution.

Bakkes et al have published numerous papers on the Spring platform [4–6]. In 2005 and 2007, they generated a prediction function for three different Spring agents. Of these three agents, two have become legacy. In 2008, they created an agent to combat the one remaining agent, AAI. The goal of this created agent was to maintain a tie in a game, with the purpose of keeping a human player entertained. The validation of this goal was presented as the length of time which the agent was able to uphold a tie against AAI and a random agent. No comparison was made to another approach.

Ontanon et al presented a method of extracting cases from expert traces in a game called WARGUS [58]. The developed agent was pitted against the native WARGUS AI. Results were presented as the record of the agent against the native AI. No comparison was made to any other method.

The first researcher to engage in a new goal in RTS who uses native AI is forced to conclude they have created a way of doing something, but is unable to say their method is superior to any other. Results against native AI are useful to follow on researchers, because they can test their approaches against this publicly available agent to compare to other research. However, they are forced to use the same platform when conducting their research, which may require a significant investment of time to learn. Additionally, they must rely on the continued availability of the native AI. If the RTS platform undergoes frequent updates, they either have to go back to the version on which previous research was performed, or assume that any updates to the game did not affect the strength of the native AI or the assumptions made in previous research.

D.4 Our Approach

We have outlined the three basic methods of presenting results in RTS games. In our research, we used a combination of comparing to a random agent and a different method, comparison to the performance of native AI. Our goal is to show the impact of the domain knowledge included in the architecture, as well as provide a method for other domain experts to gauge how significant those results are. By comparing to the performance of KAI and RAI against static scripts, we enable other researchers to go back and determine how effective their performance is when compared to these two agents.

We think RTS research is still new. No single platform is used, and no one has shown that all RTS games are the same, so a technique which is successful on one platform will retain its success when ported to a new platform. These are open areas of research.

Bibliography

1. Aha, David W., Dennis Kibler, and Marc K. Albert. "Instance-based Learning Algorithms". *Machine Learning*, 6:37–66, January 1991.

2. Aherne, F., N. Thacker, and P. Rockett. "The Bhattacharyya Metric as an Absolute Similarity Measure for Frequency Coded Data". *Kybernetika*, 32:1–7, 1997.

3. Back, Thomas. *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, 1996.

4. Bakkes, Sander, Philip Kerbusch, Pieter Spronck, and Jaap van den Herik. "Automatically Evaluating the Status of an RTS game". *Proceedings of the IJCAI-05 Work- shop on Reasoning, Representation, and Learning in Computer Games.* 2005.

5. Bakkes, Sander, Pieter Spronck, and Jaap van den Herik. "Phase-dependent Evaluation in RTS games". *Proceedings of the 19th Belgian-Dutch Conference on Artificial Intelligence*. 2007.

6. Bakkes, Sander, Pieter Spronck, and Japp van den Herik. "Rapid Adaptation of Video Game AI". *IEEE Symposium on Computational Intelligence and Games*, 79–86. 2008.

7. Beerten, Franois, Jimmy Salmon, Los Taulelle, Frank Loeffler, Nehal Mistry, and Tejay Penfold. "Bos Wars". Open Source Software, 2008. URL `http://www.boswars.org/`.

8. Beyer, Hans-Georg and Hans-Paul Schwefel. "Evolution Strategies –A Comprehensive Introduction". *Natural Computing: An International Journal*, 1:3–52, 2002.

9. Billings, D., N. Burch, A. Davidson, R. Holte, T. Schauenberg, and D. Szafron. "Approximating Game-Theoretic Optimal Strategies for Full-scale Poker". *Eighteenth International Joint Conference on Artificial Intelligence*, 661–668. 2003.

10. Blum, Avrim, Adam Kalai, and John Langford. "Beating the Hold-Out: Bounds for K-fold and Progressive Cross-Validation". *Twelfth Annual Conference on Computational Learning Theory*, 203–208. 1999.

11. Blum, Avrim L. and Pat Langley. "Selection of Relevant Features and Examples in Machine Learning". *Artificial Intelligence*, 97:245–271, 1997. ISSN 0004-3702.

12. Bouzy, B. and B. Helmstetter. "Monte-Carlo Go Developments". *Tenth Conference on Advances in Computer Games*, 159–174. 2003.

13. Browne, Michael W. "Cross-Validation Methods". *Journal of Mathematical Psychology*, 44(1):108 – 132, 2000. ISSN 0022-2496.

14. Buro, Michael. "ORTS: A Hack-Free RTS Game Environment". *Third International Conference on Computers and Games*, 156–161. 2003.

15. Buro, Michael. "Call for AI Research in RTS Games". *AAAI Workshop on AI in Games*, 139–141, 2004.

16. Cestnik, Bojan, Igor Kononenko, and Ivan Bratko. "ASSISTANT 86: A Knowledge-Elicitation Tool for Sophisticated Users". *Second European Working Session on Learning*, 31–45. 1987.

17. Chung, Michael, Michael Buro, and Jonathan Schaeffer. "Monte Carlo Planning in RTS Games". *IEEE Symposium on Computational Intelligence and Games*. 2005.

18. Coello, Carlos A., Gary B. Lamont, and David A. Van Veldhuizen. *Evolutionary Algorithms for Solving Multi-Objective Problems (Genetic and Evolutionary Computation)*. Springer-Verlag New York, Inc., 2006.

19. Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (2nd Edition)*. The MIT Press, 2001.

20. Darwin, Charles. *The Origin of Species*. Gramercy, May 1995.

21. Dastani, Mehdi, Dirk Hobo, and John-Jules Meyer. "A Platform For Evolving Characters in Competitive Games". *The Congress on Evolutionary Computation*, 1420–1426. 2004.

22. Davies, Scott and Stuart Russell. "NP-completeness of Searches for Smallest Possible Feature Sets". *1994 AAAI Fall Symposium on Relevance*, 41–43. 1994.

23. De Jong, Kenneth A. "On Using Genetic Algorithms to Search Program Spaces". *Second International Conference on Genetic Algorithms and their Applications*, 210–216. L. Erlbaum Associates Inc., Hillsdale, NJ, USA, 1987. ISBN 0-8058-0158-8.

24. De Jong, Kenneth Alan. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*. Ph.D. thesis, University of Michigan, Ann Arbor, MI, 1975.

25. Dietterich, Tom. "Overfitting and Undercomputing in Machine Learning". *ACM Computing Surveys*, 27:326–327, 1995.

26. Dijkstra, E. W. "A Note on Two Problems in Connexion With Graphs". *Numerische Mathematik*, 1:269–271, 1959.

27. Dompke, Uwe. "Computer Generated Forces - Background, Definition and Basic Technologies". *Simulation of and for Military Decision Making*. North Atlantic Treaty Organisation/Research and Technology Organisation, 2001.

28. Fairclough, Chris, Michael Fagan, Brian Mac Namee, and Pdraig Cunningham. "Research Directions for AI in Computer Games". *Twelfth Irish Conference on Artificial Intelligence and Cognitive Science*, 333–344. 2001.

29. Friedman, Jerome H. "Regularized Discriminant Analysis". *Journal of the American Statistical Association*, 84:165–175, 1989.

30. Gardner, Ann. "Search: An Overview". *AI Magazine*, 2:2–6, 1981.

31. Goldberg, David E. "Genetic Algorithms and Rules Learning in Dynamic System Control". *First International Conference on Genetic Algorithms*, 8–15. 1985.

32. Goldberg, David E. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Longman Publishing Co., Inc., 1989.

33. Guyon, Isabelle and André Elisseeff. "An Introduction to Variable and Feature Selection". *The Journal of Machine Learning Research*, 3:1157–1182, 2003.

34. Headquarters, Department of the Army. *United States Army Field Manual 3-0: Operations.* 2001.

35. Herz, J.C. and Michael R. Macedonia. "Computer Games and the Military: Two Views". *Defense Horizons*, 11:1–8, April 2002.

36. Holland, John H. "Outline for a Logical Theory of Adaptive Systems". *The Journal of the Association for Computing Machinery*, 9:297–314, 1962.

37. Holland, John H. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence.* MIT Press, 1992.

38. Holland, John H. and Judith S. Reitman. "Cognitive Systems Based on Adaptive Algorithms". *The ACM Special Interest Group on Artifical Intelligence Bulletin*, 63:49–49, 1977.

39. Hoos, Holger and Thomas Sttzle. *Stochastic Local Search: Foundations & Applications.* Morgan Kaufmann Publishers Inc., 2004.

40. IBM Research. "Kasparov vs. Deep Blue; the Rematch". Web, May 1997. URL `http://www.research.ibm.com/deepblue/`.

41. Jain, A.K., M.N. Murty, and P.J. Flynn. "Data Clustering: A Review". *ACM Computing Surveys*, 31:264–323, 1999.

42. Jain, Anil and Douglas Zongker. "Feature Selection: Evaluation, Application, and Small Sample Performance". *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19:153–158, 1997.

43. Joint Chiefs of Staff. *Joint Publication 5-0, Joint Operational Planning.* United States Department of Defense, 2006.

44. Joint Doctrine Division, J-7, Joint Staff. *Joint Publication 1-02, Department of Defense Dictionary of Military and Associated Terms.* United States Department of Defense, 2001, amended 2009.

45. Keller, Joseph B. *Stirling's Formula Derived Simply.* Technical Report arXiv:0711.4412v2, Nov 2007.

46. Kok, Eric. *Adaptive Reinforcement Learning Agents in RTS Games.* Master's thesis, University Utrecht, The Netherlands, 2008.

47. Kotsiantis, S. B. "Supervised Machine Learning: A Review of Classification Techniques". *Informatica*, 31:249–268, 2007.

48. Laird, John E. "An Exploration into Computer Games and Computer Generated Forces". *Eighth Conference on Computer Generated Forces and Behavior Representation.* 2000.

49. Laird, John E. and Michael van Lent. "Human-Level AI's Killer Application: Interactive Computer Games". *Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, 1171–1178. 2000. ISBN 0262511126.

50. Lamont, Gary B. "Course Notes, CSCE 686, Advanced Algorithm Design". Air Force Institute of Technology, WPAFB, Ohio, 2009.

51. Langley, Pat and Wayne Iba. "Average-Case Analysis of a Nearest Neighbor Algorithm". *Thirteenth International Joint Conference on Artificial Intelligence*, 889–894. 1993.

52. Lewis, Harry R. and Larry Denenberg. *Data Structures and Their Algorithms.* Addison-Wesley Longman Publishing Co., Inc., 1991.

53. von der Lippe, Sonia, Robert Bialczak, Jonathan Nida, and Michelle Kalphat. "A Robotic Army: The Future is CGF". *Tenth Conference on Computer Generated Forces and Behavioral Representation.* 2001.

54. MacKay, David J. C. *Information Theory, Inference, and Learning Algorithms.* Cambridge University Press, 2003.

55. de Mantaras, Ramon Lopez and Eva Armengol. "Machine Learning From Examples: Inductive and Lazy methods". *Data & Knowledge Engineering*, 25:99–123, 1998.

56. Merrick, Kathryn and Mary Lou Maher. "Motivated Reinforcement Learning for Non-player Characters in Persistent Computer Game Worlds". *2006 ACM SIGCHI International Conference on Advances in Computer Entertainment Technology*, 3. 2006.

57. Michalewicz, Zbigniew and David B. Fogel. *How to Solve It: Modern Heuristics.* Springer, December 2004.

58. Ontanon, Santiago, Kinshuk Mishra, Neha Sugandh, and Ashwin Ram. "Case-based Planning and Execution for Real-time Strategy Games". *Seventh International Conference on Case-Based Reasoning*, 164–178. 2007.

59. Pearl, Judea. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley Longman Publishing Co., Inc., 1984.

60. Priesterjahn, Steffen and Alexander Weimer. “An Evolutionary Online Adaptation Method for Modern Computer Games Based on Imitation”. *Ninth Annual Conference on Genetic and Evolutionary Computation*, 344–345. 2007.

61. Radcliffe, Nicholas J. and Patrick D. Surry. “Fundamental Limitations on Search Algorithms: Evolutionary Computing in Perspective”. *Lecture Notes in Computer Science 1000*, 275–291. Springer-Verlag, 1995.

62. Ramsey, Fred L. and Daniel W. Schafer. *The Statistical Sleuth: A Course in Methods of Data Analysis (2nd ed.).* Duxbury Press, 2002.

63. Reth. “RAI: Reth’s Artificial Intelligence for Spring”, 2009. URL `http://springrts.com/wiki/AI:RAI`. Http://springrts.com/wiki/AI:RAI.

64. Robles, David and Simon M. Lucas. “A Simple Tree Search Method for Playing Ms. Pac-Man”. *IEEE Symposium on Computational Intelligence and Games*, 249–255. 2009.

65. Russell, Stuart J. and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003.

66. Schaeffer, Jonathan. “A Gamut of Games”. *AI Magazine*, 22:29–46, 2001.

67. Schaeffer, Jonathan, Neil Burch, Yngvi Bjornsson, Akihiro Kishimoto, Martin Muller, Robert Lake, Paul Lu, and Steve Sutphen. “Checkers Is Solved”. *Science*, 317:1518–1522, 2007.

68. Seizinger, Alexander. “AI:AAI for Spring.”, 2006. URL `http://spring.clan-sy.com/wiki/AI:AAI`.

69. Shannon, C. E. “Programming a computer for playing chess”. *Philosophical Magazine*, 41:256–275, 1950.

70. Shoemaker, C.M. and J.A. Bornstein. “The Demo III UGV Program: a Testbed for Autonomous Navigation Research”. *IEEE International Symposium on Intelligent Control*, 644–651. 1998.

71. Smith, E. E. and D. Medin. *Categories and Concepts*. Harvard University Press, 1981.

72. Spronck, Pieter. *Adaptive Game AI*. Ph.D. thesis, Mastricht University, The Netherlands, 2005.

73. Spronck, Pieter, Marc Ponsen, Ida Sprinkhuizen-Kuyper, and Eric Postma. “Adaptive Game AI with Dynamic Scripting”. *Machine Learning*, 63:217–248, 2006.

74. Stene, Sindre Berg. *Artificial Intelligence Techniques in Real-Time Strategy Games - Architecture and Combat Behavior*. Master’s thesis, Norwegian University of Science and Technology, 2006.

75. The WARGUS Team. “WARGUS”, 2002-2007. URL `http://wargus.sourceforge.net/`.

76. Tozour, Paul. “The Perils of AI Scripting”. *AI Game Programming Wisdom*, 541–547. Charles River Media, Inc., 2002.

77. Vobruba, Robin. “Spring C++ Wrapper”. Open Source Software, 2008.

78. Vobruba, Robin. “Spring Java Wrapper”. Open Source Software, 2008.

79. Weissgerber, Kurt, Brett J. Borghetti, and Gilbert L. Peterson. “An Effective and Efficient Real Time Strategy Agent”. *Twenty-third Annual Florida Artificial Intelligence Research Society Conference (submitted)*. 2010.

80. Wolpert, David H. and William G. Macready. *No Free Lunch Theorems for Search*. Working Papers 95-02-010, Santa Fe Institute, February 1995. URL `http://ideas.repec.org/p/wop/safiwp/95-02-010.html`.

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)* 25–2–2010	**2. REPORT TYPE** Master's Thesis	**3. DATES COVERED** *(From — To)* Sept 2008 — Mar 2010

4. TITLE AND SUBTITLE

Developing an Effective and Efficient Real Time Strategy Agent for Use as a Computer Generated Force

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Weissgerber, Kurt, Capt, USAF

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management
2950 Hobson Way
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GCS/ENG/10-07

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Intentionally left blank

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approval for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Computer Generated Forces (CGF) are used as units or people in military training and simulation. The use of CGF significantly reduces the time and money required for effective training. Real Time Strategy (RTS) games place players in control of a large force whose goal is to defeat the opponent. The military setting of RTS games makes them an excellent platform for the development and testing of CGF. While significant research has been done into RTS agent development, most of the developed agents are only able to exhibit good tactical behavior. By analyzing prior games played by an opposing agent, an RTS agent could determine the opponent's strengths and weaknesses and develop a strategy which neutralizes the strengths and capitalizes on the weaknesses. It could then execute this strategy in an RTS game. This research develops the Killer Bee Artificial Intelligence (KBAI). KBAI takes a classifier for the RTS domain, uses it to generate an effective counter-strategy, and executes the tactics required for the strategy.

15. SUBJECT TERMS

Computer Generated Forces, Real Time Strategy, Classification, Strategy Generation

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Brett J. Borghetti, Lt Col, USAF (ENG)
U	U	U	UU	137	**19b. TELEPHONE NUMBER** *(include area code)* (937) 255–6565, x4612 brett.borghetti@afit.edu