Hindawi Publishing Corporation
International Journal of Computer Games Technology
Volume 2011, Article ID 834026, 17 pages
doi:10.1155/2011/834026

Research Article

Determining Solution Space Characteristics for Real-Time Strategy Games and Characterizing Winning Strategies

Kurt Weissgerber, Gary B. Lamont, Brett J. Borghetti, and Gilbert L. Peterson

Department of Electrical and Computer Engineering, Graduate School of Engineering and Management, Air Force Institute of Technology, Wright Patterson AFB, Dayton, OH 45433, USA

Correspondence should be addressed to Gary B. Lamont, cruisede@aol.com

Received 24 September 2010; Revised 7 January 2011; Accepted 2 March 2011

Academic Editor: Alexander Pasko

The underlying goal of a competing agent in a discrete real-time strategy (RTS) game is to defeat an adversary. Strategic agents or participants must define an a priori plan to maneuver their resources in order to destroy the adversary and the adversary's resources as well as secure physical regions of the environment. This a priori plan can be generated by leveraging collected historical knowledge about the environment. This knowledge is then employed in the generation of a classification model for real-time decision-making in the RTS domain. The best way to generate a classification model for a complex problem domain depends on the characteristics of the solution space. An experimental method to determine solution space (search landscape) characteristics is through analysis of historical algorithm performance for solving the specific problem. We select a deterministic search technique and a stochastic search method for a priori classification model generation. These approaches are designed, implemented, and tested for a specific complex RTS game, Bos Wars. Their performance allows us to draw various conclusions about applying a competing agent in complex search landscapes associated with RTS games.

1. Introduction

The real-time strategy (RTS) domain [1] is of interest because it relates to real world problems, for example, determining a "good" military battlefield strategy or defining the "best" strategies for complex RTS video games. A participants/agent strategy is to develop a long-term plan using an agent's resources to win the game. Note that the RTS genre is different than games requiring only real-time tactics (RTT) which deal with making decisions on detailed resource use at each iteration of the game. RTT sometimes are considered a subgenus of real-time strategies. Another way of defining an RTS structure is to consider the terms *macro-management* referring to high-level strategic maneuvering and *micro-management* referring to RTT game interaction.

The objective of a competing agent in an RTS game is to defeat an adversary (or adversaries) by directly and indirectly moving and maneuvering resources in order to destroy the adversary's resources, capture and destroy the adversary, and secure physical regions of the environment [1]. In a gaming situation, it is desired to gather or destroy resources, build physical structures, improve technological development, and control other agents. This is a daunting set of strategic tasks for an RTS game player.

A *comprehensive RTS game could include extensive models of information availability, relations (espionage, diplomacy, intrigue), politics, ingenuity, economics, control (stability), logistics (scarcity), risk management, synchrony, and the scope of complexity, space, and time (speed)*. To incorporate all of these strategic models into an RTS game or simulation is probably next to impossible! Thus, all may not be part of a contemporary RTS game. For example, games such as Ground Control or Company of Heroes do not require resource-gathering. On the other hand, the scope of time and space complexity for each RTS game characteristics are generic areas of interest that may present very difficult problem domains for a dynamic and adaptive RTS agent.

Nevertheless, an existing method for development of a *strategy-based agent* is to employ Artificial Intelligence (AI) techniques in learning while playing. Such an approach

can include genetic algorithms, coevolution, and scripts via a variety of search techniques. Most such learning approaches involve defining an agent architecture, decision variable representation, explicit functional objectives, search exploration and exploitation algorithms, information collection, and a simulation implementation. Any AI architecture must permit an RTS agent to observe its environment and make decisions based on what it observes. An adversarial agent must take actions which allow it to defeat some opponent(s). Many current RTS approaches use AI learning agents, where the agent determines appropriate actions to take in a particular *state* through trial and error; that is, instantiating if-then-else rules or case-based reasoning [1]. The agent must determine some way to collect the required information about the environment and the opponent and then use this information effectively to "beat" the opponent via an action sequence.

We present an AI strategy-based agent which collects information and learns about an opponent by examining its past performance. Past performance can be captured through a collection of *game trace* records from the adversarial agent's state movements. Traces by definition consist of a vector of *snapshots* and each snapshot also contains the value of various types of *features* at a specific point in time. A vector of snapshots which encompass an entire RTS game can be used to reconstruct via search and learn the important strategic features of the game which can lead to victory.

Before the subject of agent generation can be approached, a reliable method of generating a classification model needs to be created. The RTS domain is relatively new; while many different AI search approaches have been applied to agent generation, little research has been done into determining the underlying characteristics of the domain. Are there many different feature combinations which lead to victory? Are they in close proximity to each other, or are they spread out around the domain? Is the solution space (fitness landscape) jagged, where good feature combinations are in very close proximity to bad feature combinations, or are the transitions between the two more gradual? By answering these questions, we can determine an algorithm to use which can leverage the characteristics of the domain to find the better solutions in a reasonable amount of time.

In this paper, related RTS investigations including RTS games are summarized in Section 2 which provides a background for our method. Section 3 formulates the RTS plan and the classification problem along with the generic solution space analysis and the selected information representation. Algorithmic learning techniques based upon deterministic and stochastic search are developed in Section 4 resulting in our AI strategy. The experimental design is provided in Section 5 with results and analysis reflected in Section 6. Conclusions and Future Work are presented in Sections 7 and 8, respectively.

2. Related Work

Related to the development of RTS games are appropriate contemporary RTS agent development methods, some current applications, and supporting generic feature selection and class identification methods.

2.1. Current RTS Game Methods. Over the past three decades, there have been a variety of imperfect information (note that perfect information games include tic-tac-toe, checkers, chess, backgammon, and Go. RTS and RTT approaches have been applied to these games with some success depending upon depth of look-ahead search [2].) RTS games including The Ancient Art of War, Cytron Masters, Utopia, Supremacy, Carrier Command, SimAnt, Dune II. And then Total Annihilation, Age of Empires, Homeworld Cataclysm, Warcraft II & III, and Age of Mythology, Dragonshard, Star Wars: Empire at War, and StarCraft evolved. Newer strategy games include current versions of World in Conflict, Company of Heroes, Civilization 4, Sins of a Solar Empire, Medieval II, Supreme Commander, and the Rise of Nations. Each limits the generic definition of a general RTS game via the previous stated possible characteristic in Section 1. Also, many incorporate RTT templates. Various action and strategy games offer single and multiplayer options as well. Always important are the issues of visualization and animation features of each game regarding ease of use and understanding along with the associated computational and graphic requirements.

Distinct details of these games can usually be found by name via the internet. We address specific RTS game attributes that have a direct consideration in our "optimal" agent algorithmic approach: Case-Based Reasoning, Reinforcement Learning, Dynamic Scripting, and Monte-Carlo planning, along with available RTS software platforms.

A Case-Based Reasoning approach was used by Ontañón et al. [3] in WARGUS, which is an open source implementation of the Blizzard RTS game WarCraft II. They define a state as a 35-feature vector and execute the case in their database closest to the current state. Cases are extracted from expert game traces; humans that were proficient in WARGUS played the game and then annotated each action they took with the goal they were trying to achieve. Each goal is a case, and each action taken to accomplish it is added to the script executed when the case is selected. The AI approach was successful, although on a small scale of only nine games. Note that Ahal et al. [4] also used a Case-Based Reasoning technique for WARGUS which generated successful results.

A Hybrid Case-Based Reasoning/Reinforcement Learning approach was used by Sharma et al. [5] to develop an AI approach for a game called MadRTS, a "commercial RTS game being developed for military simulations". Their technique uses a set of features to determine a game state, such as the number of territories controlled by a given player and the number of units still alive for a given player. Additionally, they incorporate lessons learned from similar tasks to increase learning speed. The developed agent showed significant gains in achieving victory when allowed to transfer knowledge from other domains.

Graepel et al. apply extended Q-learning reinforcement in order to find "good" Markov decision policies for a fighting agent game [6]. The agents are trained using an on-policy algorithm (an on-policy learning algorithm for

an agent interacts with the environment and updates the agent's policy based on current actions taken) for temporal difference learning that employs linear and neural network function approximators. Various selected rewards encourage aggressive or defensive agent behavior. Some acceptable agent policies using these reward functions are found for the author's particular AI game.

Continuous action model-learning in an RTS environment was addressed by Molineaux et al. [7]. They develop the Continuous Action and State Space Learner (CASSL). Their approach is an integrated Case-Based Reasoning/ Reinforcement Learning algorithm. Testing indicated that CASSL significantly outperformed two baseline approaches for selecting agent actions on a task from an RTS gaming environment.

Dynamic Scripting is a method developed by Spronck et al. [8] for third person role-playing games. The generic technique uses a set of rules to define a game state, and the value of these rules determines what actions are added to the script at each turn. This is a way of dealing with the huge decision space of RTS games. However, it prevents this approach from reasoning on actual game conditions. This research was extended to the RTS domain by Kok [9]. Reinforcement learning was used to determine appropriate actions based on states. Instead of using only rule values, the approach allowed use of some knowledge of the actual state which generally leads to success.

A Monte-Carlo planning approach was used by Chung et al. [10] in "Capture the Flag" (CTF) games. In a CTF game, the agent's objective is to obtain the opponent's flag and return it to its base before the opponent is able to do the same. Using a CTF game reduced the complexity of the state; resource collection was unnecessary, and complex strategic level plans were not required. At each step of the game, the designed agent would generate a number of plans (parameter passed to the function), evaluate their performance against the possible actions the opponent could take, and execute the best plan. The success of this approach of course was highly linked to the specific game conditions.

In general, these approaches for solving RTS games do generate acceptable nonoptimal but not robust RTS solutions. This situation is generally due to the characteristics of the highly dimensional RTS search space being jagged and very rough. Moreover, we show this characteristic empirically via more appropriate stochastic search.

Note that contemporary AI techniques in RTS games continue to be in the development stage but with limited implementation. Observe that currently all such RTS games can be beaten by a knowledgeable human opponent, thus, making RTS games quite interesting and one would hope playable. Also, no single AI or human approach has been shown to be better or show more promise than others; therefore, there probably is no generic robust RTS game strategy-based agent that leads to victory in all cases! One can think of this situation as a reflection of the no-free-lunch theorem [11].

2.2. Some Current RTS Platforms. There are a number of RTS platforms on which to implement an RTS game along with collection of algorithmic game data. For example, *Bos Wars* [12] which is an open-source RTS developed as a no-cost alternative to commercial RTS games. Another is *Spring Engine* [13] where perfect knowledge of the environment is not available so a *temporal difference learning* technique is employed. A physics engine called *Havok* Game Dynamics SDK is used in some other RTS games such Age of Empires III and Company of Heroes for realism. Another platform is the NERO game [14], which stands for Neuro-Evolving Robotic Operatives. For the NERO project, a specific neural-net evolutionary algorithm is designed called rtNEAT, real-time Neuro-Evolution of Augmenting Topologies. These RTS platforms operate under Windows or Linux and require high-speed CPUs and extensive graphical interfaces. which stands for Neuro-Evolving Robotic Operatives.

We choose to use the Bos Wars platform for determining general RTS search space characteristics. This choice provides an efficient and effective computational platform for gaining initial insight to the RTS search space. Knowing these characteristics, generic RTS platforms can be used later to explicitly search for RTS strategic solutions using appropriate stochastic AI algorithms.

2.3. General Feature Selection. The goal of generic feature selection is to find a *subset* of features from a data domain (game traces) in order to maximize some identification function a priori. This subset of features can then be used to classify given date at some epoch *(snapshot)*. In the RTS Feature Selection problem, the goal is to classify game states via this feature subset at each snapshot. An initial execution of a selected number of the same RTS game can determine the feature subset. The RTS optimization identification function is derived from a general classification problem; once the appropriate RTS subset features are determined through the RTS training data, game playing state data can be separated quickly with this subset into classes at each snapshot. Note that a method to generalize each class must be determined, so all game states can be classified as well. Those states classified as winning strategies are sought out of course. This is in general a very difficult computational problem. Of course, Generic Feature Selection and Classification continue to be open research areas in engineering and science.

A general overview of feature selection and classification methods is given by Blum and Langley [15]. Although the others listed would also be appropriate, Bos Wars was chosen for ease of analysis. Different ways of defining a *relevant* feature are discussed. One of the most basic is "feature x_i is relevant if there exists some example in the instance space for which twiddling the value of x_i affects the classification." For the remainder of this paper, the term "important" is used synonymously with the definition of relevant.

Blum and Langley [15] also discuss three different general methods of feature selection and classification: filter, wrapper, and embedded. In *filtering methods*, features are selected and then passed to a classification algorithm. This solves the entire problem as a two-step process. In a *wrapper approach*, the two problems are still separate, but multiple solutions are explored. A subset of features is chosen and passed to a classification algorithm, then a different subset is

chosen and its performance in the classification algorithm is compared. This process is repeated many times leading to an acceptable classification. In an *embedded approach*, the two problems are solved concurrently via parallel interaction.

The algorithm designed in this paper takes an embedded approach to a priori feature selection and classification. In each method, possible class separability and clustering functions are based upon a distance function. Such metrics include error probability, interclass distance, k-means clustering, entropy, consistency-based feature selection, and correlation-based feature selection.

A good overview of the feature selection problem domain is presented by Jain et al. [16] in which they define some pertinent terms. "Pattern representation" refers to the number of classes, the number of available patterns, and the number, type, and scale of the features available to the clustering algorithm. Again, the goal of feature selection/classification is to find the specific pattern representation which maximizes (optimizes) the performance of a classifier, in our case, winning game strategies.

Collections of RTS game traces can be used to construct a generalization of a particular game given many runs. By using machine learning techniques, specifically the generation of classification models for the game traces, the feature value combinations which tend to lead to victory and the feature value combinations which tend to lead to defeat can be determined. These good and bad feature values can then be given to an agent that would seek to avoid the bad feature combinations and approach the use of good combinations in the temporal decision process of the game.

There are numerous approaches to feature selection, using many different algorithms and heuristics. For example, search algorithms include deterministic depth-first search and breath-first search (best-first search), and stochastic simulated annealing and genetic algorithm techniques. The Feature Selection problem is known to be NP-Complete [17], with a solution space of $O(n^n)$, where n is the number of possible features which could be selected. Thus, in large feature spaces, stochastic approaches are preferred generating acceptable solutions relatively quickly.

For example, to reduce the problem search space, Somol et al. [18] used heuristics to prevent the expansion of unproductive nodes. By predicting the value of a node instead of computing its actual value, they were able to reduce the amount of time spent evaluating each node. This led to reduced time spent on a search, as well as pruning off nonproductive areas of the search space.

As an example in the marketing domain, feature selection is used to determine customers who are likely to buy a product, based on the other products they have bought. Genetic algorithms were used by Jarmulak and Craw [19] to solve this problem. They assigned weights to each feature selected to take advantage of the relative importance of each feature. Simulated annealing was used by Meiri and Zahavi [20] to solve a similar marketing problem. Feature identification results in both cases were deemed acceptable. Historical motivation for simulated annealing use in optimization problems is discussed by Kirkpatrick et al. [21].

There are numerous examples of feature selection methods, in many different domains. However, feature selection is usually a domain specific problem; a feature selection algorithm which gives a good solution in one problem domain does not necessarily give the same quality of solution in a different domain. Our embedded algorithm uses a priori stochastic feature selection as motivated in the following sections.

2.4. Classification Methods. A classifier is a system created from quantitative labeled data which can then be used to generalize qualitative data. In a more general sense, building a classifier is the process of learning a set of rules from instances. These rules can be used to assign new samples to classes. In an AI taxonomy, classification falls into the realm of supervised machine learning [22]. Note that our *perception* is the process of attaining awareness or understanding of sensory information via classification.

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 or rule-based classifiers for games [23]. New data can be classified by following the decision tree from the root to a leaf node and classifying appropriately. *Perceptron-based* techniques (neural net) 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* and *Probabilistic learning* algorithms generate probabilities that a sample belongs to a specific class, instead of a simple classification. Common examples of these techniques are linear discriminant analysis [24] and Bayesian networks, which were first used in a machine learning context in 1987 [25]. Note that various classifiers can come under a variety of learning algorithm definitions.

The family of *instance-based* learning algorithms are the most useful when developing an agent [26]. 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 [27].

Two basic IBL exemplar models are *proximity* and *best-example*. 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 [28]. Best-example models can greatly reduce the subset size of features.

Another classification method based upon the K-NN approach is the K-winner machine (KWM) model [29]. KWM training uses unsupervised vector quantization and subsequent calibration to label data-space partitions.

A K-winner classifier seeks the largest set of best-matching prototypes agreeing on a test pattern and provides a local-level estimate of confidence. The result leads to tight bounds to generalization performance. The method maybe suitable for high-dimensional multiclass problems with large amounts of data. Experimental results on both synthetic and real domains confirm the approach's effectiveness.

One method 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 are assigned to them. This process repeats until the change in the clusters approaches zero or some defined threshold [30]. Although a spectrum of classification techniques have been introduced for clarification, the classification method selected in the following sections is motivated by the desired to provide insight to RTS search space characteristics. In developing an efficient and effective classification process for a specific RTS game, consideration of the above approaches should be addressed.

3. The Problem

Our Real-Time Strategy Prediction Problem (RTSPP) is a classification problem which is formulated as a basic search problem. Any search problem definition including the RTSPP can be defined by its input, output, and fitness function.

3.1. Problem Definition. The input to the RTSPP is a set of game traces from RTS games. Each game trace consists of "snapshots" taken at constant intervals or epochs. Each snapshot contains the value of all the possible features which an agent can observe. In the RTS domain, features could be the number and type of units, the amount of energy or fuel, or the rate at which energy and fuel are collected or used. Features could also be the rate of change of any of the static features across some time interval. Each snapshot is labeled as to whether it came from a game which was won or lost from player one's perspective.

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 cuts the space required to store game traces in half.

The output (solution) of the RTSPP is a classifier: a subset of features, a set of winning *centers,* and a set of losing centers. The set of features determines which features are used in the classifier. Each center in the set of winning centers gives a set of values across the features which generally result in a winning game. The set of losing centers is the same concept, only from losing games.

The classifier is then used to predict the outcome of a game based on only 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 result in a win. If it is a losing center, then the game state should result in a loss.

The quality of a solution to the RTSPP can be measured by testing its classification performance. *Classification performance* is measured as a percentage of right answers to total samples over various games.

3.2. Formal Problem Definition. The RTSPP is formally defined to remove any ambiguity of understanding. There is a set F of features and a set S of snapshots. 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 snapshots.

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 $C_w \cap C_l = \oslash$. Each center is a representative sample of a snapshot that is a mean of a cluster of minimizing samples.

The fitness of a solution can be determined by using it to classify all the samples in S. The function $\text{dist}(s, c)$ returns the Euclidean distance for example from a sample s to a center c, so the value of a prediction function $P(s)$ is

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

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 actual classification value of sample s is denoted by $P^*(s)$. $g(s)$ is formally defined as

$$g(s) = \begin{cases} 1 & P(s) = P^*(s), \\ 0 & P(s) \neq P^*(s). \end{cases} \quad (2)$$

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

$$G(S) = \frac{\sum_{i=1}^{m} g(s_i)}{m}. \quad (3)$$

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

3.3. RTSPP Solution Space Analysis. The concluding step in the problem definition is an analysis of the number of possible RTSPP solutions. 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 $O(n!) \approx O(n^n)$.

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 $O(m!) \approx O(m^m)$. 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. Since there is no reason to have more than m centers, the solution space for real valued centers is of order $O(1000^n \times m)$.

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. An overall objective of the RTSPP solution 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 objective. 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:

$$\begin{aligned} |F'| &< j, \\ |C| &< k. \end{aligned} \tag{4}$$

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.

4. Feature Subset Search Methods

Any search problem can be solved using one of two general search types: deterministic and stochastic [31]. A deterministic algorithm is not probabilistic. The next search state is only determined from the current search state (partial solution) and the chosen search algorithm. To generate an optimal solution via expanding partial solutions, the entire search space must be searched either explicitly or implicitly. This means that the problem domain could be relaxed to decrease the size of the search space so it can be searched in a reasonable amount of time. Thus, relaxing the problem domain dimensionally yields an optimal solution to a smaller problem.

In a stochastic search, the algorithm is a probabilistic search over the solution space. The next state (solution) of a stochastic 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 solution space; instead, it seeks 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 or at least acceptable.

In some problem domains, a near optimal solution to the original problem is better than an optimal solution. 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 deterministic and stochastic search algorithms are developed to solve the RTS classification problem. They are tested on a data set from an RTS application, and their performance is compared. Finally, a selection is made between the deterministic and stochastic families for further development. To appreciate the subtle aspects of these feature selection search techniques for RTS games, the following sections are provided.

4.1. Deterministic Feature Subset Search. In general, features work in combinations to determine the fitness of a given RTS state. To find a subset of features, deterministic search in the RTS domain faces an immediate problem because of the complexity and roughness of the solution space. There is no way to search the entire problem space in a reasonable amount of time, which would be required to guarantee an optimal classification solution. Moreover, classification, when conducted on a problem with dependent variables, does not lend itself to implicit searching. The RTSPP for example probably has dependent variables.

In problems with independent variables, a solution can be constructed by adding features to a solution one by one, adding the feature at each level which has the greatest positive effect on the classification accuracy of the model. Dependent variables provide no such guarantee; because they work in combinations, the addition or deletion of a feature from a solution can have a large and unpredictable effect on classification model accuracy.

Basically, this means there is no admissible heuristic [31] which can be used to trim the search space. An admissible heuristic by definition always generates an optimal solution. However, there are nonadmissible ways which can be used to guide the search. We present one such method, which we use to achieve two different goals: it decreases the solution space so that every possible solution can be tested in a reasonable amount of time, and it guides the search towards profitable areas of the search space. By examining the solutions generated through the use of a heuristic, we can determine characteristics of the solution space, which is one of our objectives. Of course, an admissible heuristic would be more appropriate, but for RTS games, good admissible heuristics are yet to be generated.

When reducing the size of the solution space via classification, we need to find a heuristic which preserves the high fitness solutions of the entire space, while discarding the solutions with low fitness. If we start with the solution space in Figure 1, we would like to find a heuristic which transforms this into the solution space in Figure 2, a desired relaxed or reduced dimensionally problem domain solution space (fitness landscape). The undesired transformed solution space in Figure 3 reflects the removal of some low fitness solutions, but the high fitness solutions have not been retained.

4.1.1. The Heuristic. 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

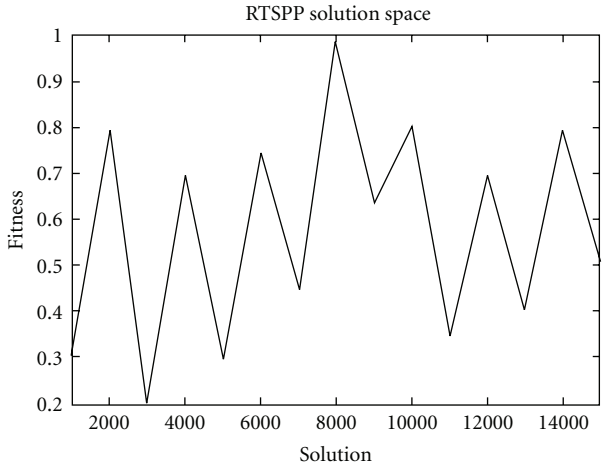


Figure 1: A hypothetical solution space.

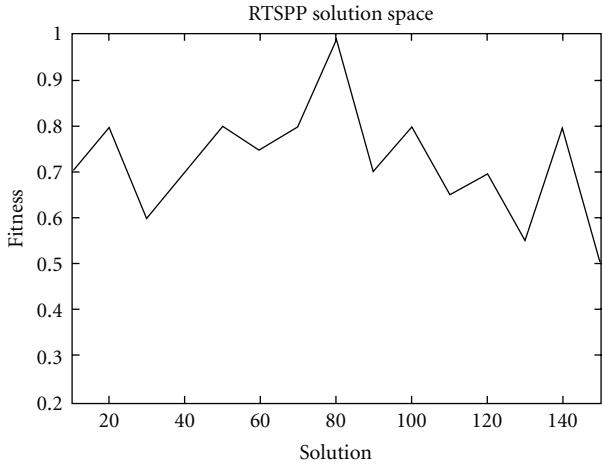


Figure 2: A hypothetical solution space which has been pruned through the use of a heuristic.

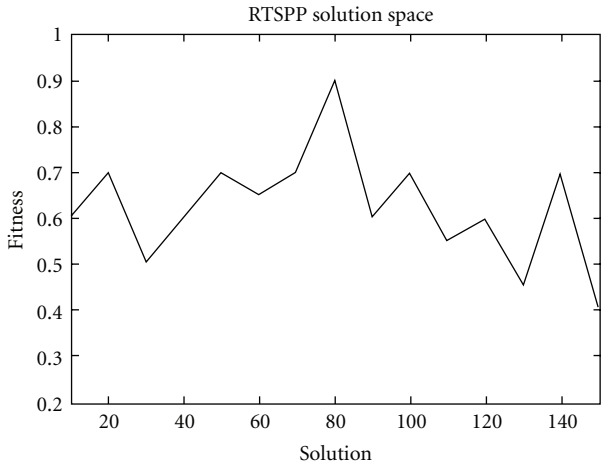


Figure 3: Another hypothetical solution space which has been pruned through the use of a heuristic.

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.

One way of determining good features involves the use of the *Bhattacharyya Coefficient (BC)* [32]. 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

$$\mathrm{BC} = \sum_{i=1}^{I} P(W_i) \times P(L_i), \tag{5}$$

where 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 4 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 can use any type of distribution.

The BC pairs each feature with two centers (one winning, one losing), so at each step of the *depth-first-search with backtracking (DFS-BT)* algorithm, 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 $|F|$.

Of course, BC is not an admissible heuristic. 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 solution samples are close together.

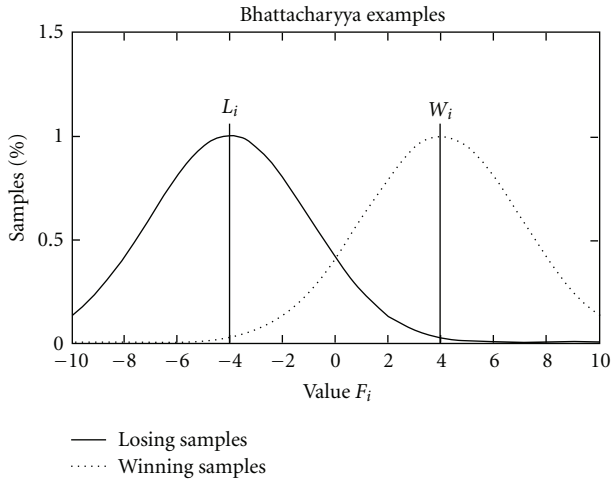


Figure 4: A visualization of the Bhattacharyya coefficient (BC) on feature F_i.

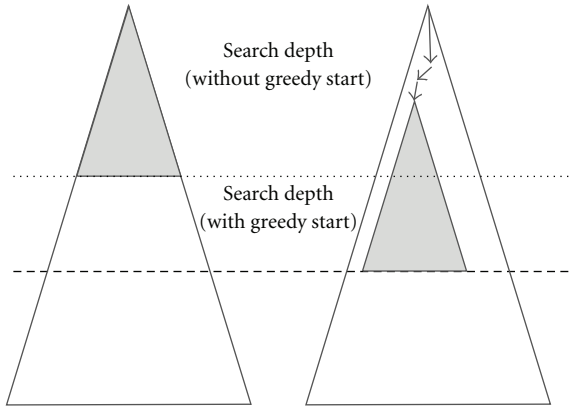


Figure 5: The increased space searchable with greedy search portion.

4.1.2. Choosing a Deterministic Search Algorithm. When choosing a search algorithm, we must keep in mind our goal: to determine the characteristics of the RTSPP solution space. We have a heuristic which we would like to test, the BC. A *best-first* algorithm would allow us to determine the effectiveness of the heuristic, as long as we search the entire domain. If the best solution found by the algorithm is found at the beginning, then the heuristic is good; it guided the search in a profitable direction. However, if the solution found is near the end of the search, then the heuristic is guiding us towards nonoptimal space.

Another way to test the effectiveness of the heuristic is to use a *greedy (DFS)* portion in our overall search. If this greedy portion is at the beginning of the search, then it allows us to increase the depth of our global search, as depicted in Figure 5. Again, the performance of this greedy search can be used to gauge the effectiveness of our heuristic. If better solutions are found when increasing the greedy search depth, then the heuristic guides us towards profitable areas of the search space.

The BC heuristic also prunes the search space. The BC pairs each feature with a center, as described. This significantly reduces the space, allowing us to completely search the space in a reasonable amount of time. However, we eliminate many possible combinations. To test the effectiveness of the heuristic from this perspective, some other method of search must be used which searches other possibilities missed by the deterministic search.

The best choice for a deterministic search algorithm is to begin with a greedy search which chooses some number of feature/center triples for a partial solution. Then, we begin a best first search which tries all the possible combinations of triples which can be used to form a solution, subject to the constraints on the number of features in a solution.

These algorithm choices lead to two different search parameters: the depth of the greedy search and the total number of features in a solution. By varying these parameters, we can gauge the effectiveness of the heuristic, as well as determine some characteristics of the solution space. But, because of the deterministic algorithm computational characteristics, a stochastic local search algorithm is selected.

4.2. Stochastic Feature Subset Search. It is assumed because of the combinatorics that the solution landscape of the RTSPP has many local maximum and minimum points. Most of these would 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 is probably not in a solution. Local maxima should be near the global maximum, while local minima should be near the global minimum. As a result of these search landscape characteristics, a stochastic algorithm that is initially biased towards exploration, but then tends to exploitation is suggested.

This tentative analysis of the solution space shows the RTSPP may be responsive to a relatively simple stochastic algorithm like *simulated annealing (SA)* [33]. Simulated annealing is very similar to the deterministic search algorithm hill-climbing [31]. 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, 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 such that 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.

4.3. SA Algorithm Domain Refinement. In order to appreciate the important design evolution of our SA method, the SA algorithm refinement is presented. Initially, we need to consider a complete formal SA specification, which 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 $n = |F|$ features, so a solution to the feature selection portion of the RTSPP is an n-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 in the solution. 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 *chromosome* solution is a binary string of length $|F| + |S|$.

The fitness function is determined by the chromosome string representing the current solution z. The fitness value is of course $G(z)$, from (3).

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. The generic neighborhood function permits a slow exploration of the solution space (landscape) with the use of this mutation operator.

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 reaches zero.

4.4. Program Specification. The combination of the algorithm constructs and specification generates the program specification in Algorithm 1.

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 temperature is selected to be a geometric cooling function based on α, SA tests $\ln(\epsilon)/\ln(\alpha)$ solutions, where ϵ is a very small number, say 0001. The overall problem solution space, from the problem definition, is $O(|F|^k \times |S|^j)$. The stochastic algorithm is not able to explore the entire solution space, but the SA initialization of solutions should "cover" all the various search space regions. The SA implementation should guide the search in good directions so the unexplored portions of the space should be uninteresting ones.

4.5. Program Specification Refinement. The problem with the program as currently designed is in the neighborhood function. Allowing flipped bit s 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.

```
D_o = ∅ "String Solution Domain"
x = x_0 "initial string"
n = 0
while T_n > 0 do
    Select z ∈ N(x) "Select string in ngbr of x via mutation"
    if g(z) > g(x) OR random > (T_n/T_0) then
        x = z
    end if
    T_{n+1} = T_n × α
    n = n + 1
end while
D_o = x "Final string solution"
```

Algorithm 1: SA RTSPP Initial Specification.

$s = F_1 \cdots F_N \mid S_1 \cdots S_M$
$x = 0110 \mid 10010110$
$z_1 = 0101 \mid 10100101$
$z_2 = 1001 \mid 01101001$

Figure 6: Proximity in solution space.

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 based upon problem insight: 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 6. 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.

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.

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

Algorithm 2: SA RTSPP Final Specification.

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 $G(x_{\text{best}})$. 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 2. 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. The algorithm is implemented, tested, and analyzed via experimental design.

5. Experimental Setup

RTS problem domain data is used to test the two designed classification search algorithms, the parameters used in each algorithm, and the performance metrics used to gauge their performance.

5.1. Data: Bos Wars Game. The algorithms are tested on data from the RTS platform *Bos Wars* [12]. Bos Wars is an open source RTS developed as a no-cost alternative to commercial RTS games. There are eight maps or game environments packaged with the game. In most maps, starting conditions for both players are similar. Each player has the same resource amount and the same access to resources and starts with the same number and type of units. Three different two-player maps are used: two have similar starting conditions and one had a line of cannons (defensive buildings) for one player. Bos Wars 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 AI search techniques packaged with the development version of the game: *Blitz, Tank Rush, and Rush*. Blitz creates as many buildings and units as possible in the hopes of overwhelming the opponent. Tank Rush 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.

Additionally, there are three different difficulty levels for the game: *Easy, Normal, and Hard*. Changing the difficulty level allows the AI search 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. As indicated, three Bos Wars maps or different environmental games are executed and evaluated: *Battlefield, Island Warfare, and Wetlands*.

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:* including 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, Medics, Rocket Tanks, Tanks, Harvesters, Training Camps, Vehicle Factories, Gun Turrets, Big Gun Turrets, Cameras, Vaults, Magma Pumps, Power Plants, and Nuclear Power Plants. Additionally, thirty delta values for all the features based on the snapshot taken 25 seconds before are created, so there are *sixty features*.

Altogether, *eighty-one games* are recorded. For the three maps, three iterations are run for selected combinations of the Bos Wars AI search techniques (Tank Rush Rush, Tank Rush Blitz, Rush versus 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 1 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 2 shows the average standard deviation in game length for a specific agent combination at a specific difficulty level on a specific map. This standard deviation is an average of the standard deviation across the three difficulty settings. These statistics show the deterministic nature of the Bos Wars scripts. In a given agent combination on a given map, the

Table 1: Records for each agent combination on listed map (1st agent wins—2nd agent wins).

Map/Agent combination	Battlefield	Island Warfare	Wetlands
Rush versus Blitz	6–3	9–0	9–0
Tank Rush versus Blitz	9–0	9–0	9–0
Rush versus Tank Rush	0–9	2–7	9–0

Table 2: Average standard deviation in game length (seconds) for agent combinations on specific maps.

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

Table 3: 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%

same agent tends to win every time. The game length is almost the same every time.

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 3. 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.

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.

5.2. Deterministic Search Parameters. The developed deterministic algorithm is a *greedy, depth-first search with backtracking* combined search. It has two search parameters which could be varied: the depth of the 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 (4) 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 all possible parameter combinations subject to these constraints, a total of *34 different test combinations*.

Table 4: 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

5.3. 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 is equal to the number of features.

Table 4 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.

5.4. 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 finishes 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.

6. 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.

6.1. 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 5 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 are averaged.

6.2. Effect of Deterministic Search Parameters. In the deterministic search, there are two parameters: the greedy search depth and the total search depth, i and j, respectively. DFS depth is equal to $j - i$. The two graphs in Figure 7 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 observed. 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 is reflected 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.

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 are compared to the best stochastic algorithm solutions on a novel data set in Section 6.5.

6.3. The Bhattacharyya Metric. The Bhattacharyya Metric (BC) is computed for each training set in the Bos Wars Data before beginning the deterministic search. In Figure 8, 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.

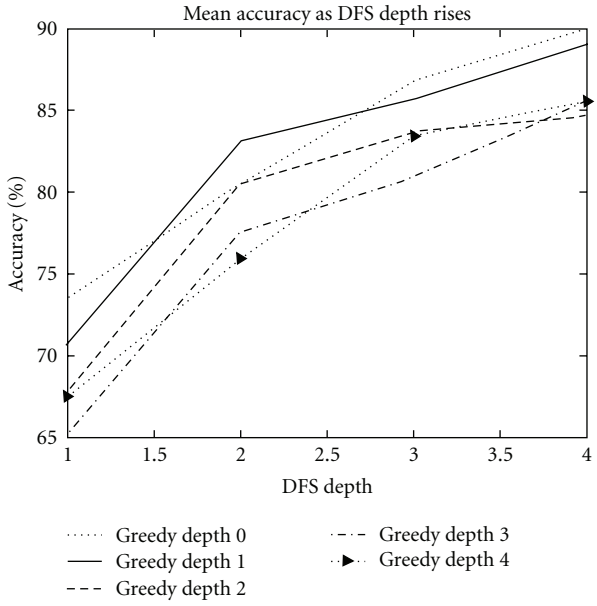
As a heuristic for the greedy search portion of the deterministic algorithm, the BC is ineffective. In almost all cases, adding more levels to the greedy search decreased performance. However, using the BC to pair features with centers is effective: using these triples, the deterministic search is able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

Table 5: Results for the DFS-BT across all folds on the Bos Wars Training Set.

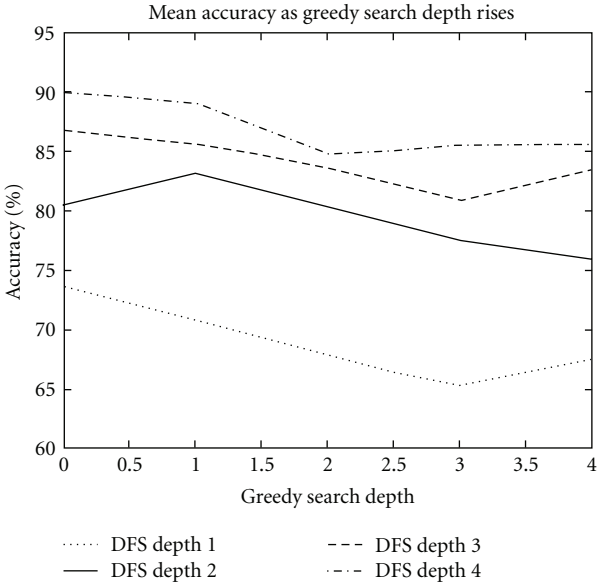
i	j	Fitness	St Dev	Time (s)	St Dev
0	1	73.6%	0.039	<1	0.00
0	2	80.5%	0.050	1.33	0.58
0	3	86.7%	0.027	46.67	0.58
0	4	90.0%	0.004	1013.00	5.29
1	2	70.7%	0.060	<1	0.00
1	3	83.1%	0.046	3.00	0.00
1	4	85.6%	0.039	69.33	1.16
1	5	89.0%	0.019	1345.33	17.79
2	3	67.8%	0.060	<1	0.58
2	4	80.5%	0.040	3.33	0.58
2	5	83.6%	0.036	90.67	0.58
2	6	84.7%	0.008	1673.00	51.18
3	4	65.2%	0.050	<1	0.58
3	5	77.5%	0.020	4.67	0.58
3	6	80.9%	0.023	113.00	1.00
3	7	85.5%	0.022	1962.33	15.54
4	5	67.5%	0.010	<1	0.58
4	6	75.9%	0.031	6.00	0.00
4	7	83.4%	0.019	141.67	0.58
4	8	85.5%	0.009	2279.00	6.56
5	6	73.0%	0.017	1.00	0.00
5	7	83.4%	0.019	7.00	0.00
5	8	85.5%	0.009	164.00	0.00
6	7	72.9%	0.016	<1	0.58
6	8	83.1%	0.009	9.00	0.00
7	8	75.7%	0.082	<1	0.57

6.4. Stochastic Search. To fine-tune the simulated annealing stochastic algorithm, the effects of various parameters on solution fitness are explored. Figure 9 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.

Both 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



(a) Effect of DFS depth on classification accuracy



(b) Effect of greedy search depth on classification accuracy

Figure 7: Effect of search depth on classification accuracy, (a) and (b).

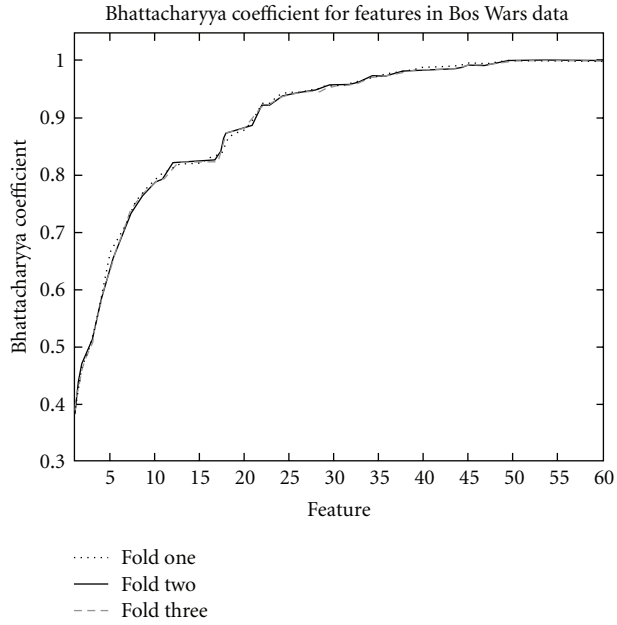


Figure 8: The BC for the features in the Bos Wars training sets.

comparisons of the average fitness values for different alpha values all yield very small P values, 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 4.4, the number of simulated annealing iterations is derived as $\ln(\epsilon)/\ln(\alpha)$, so the exponential relationship was 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 .000, giving significant statistical evidence of a difference in average fitness value for different feature values. The effect on search time is linear. This was also expected. The 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 due to α. Two-sample t-tests for the difference in average fitness are less definitive, with P values ranging from .6 to .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.

6.5. 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 are 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 observed, 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

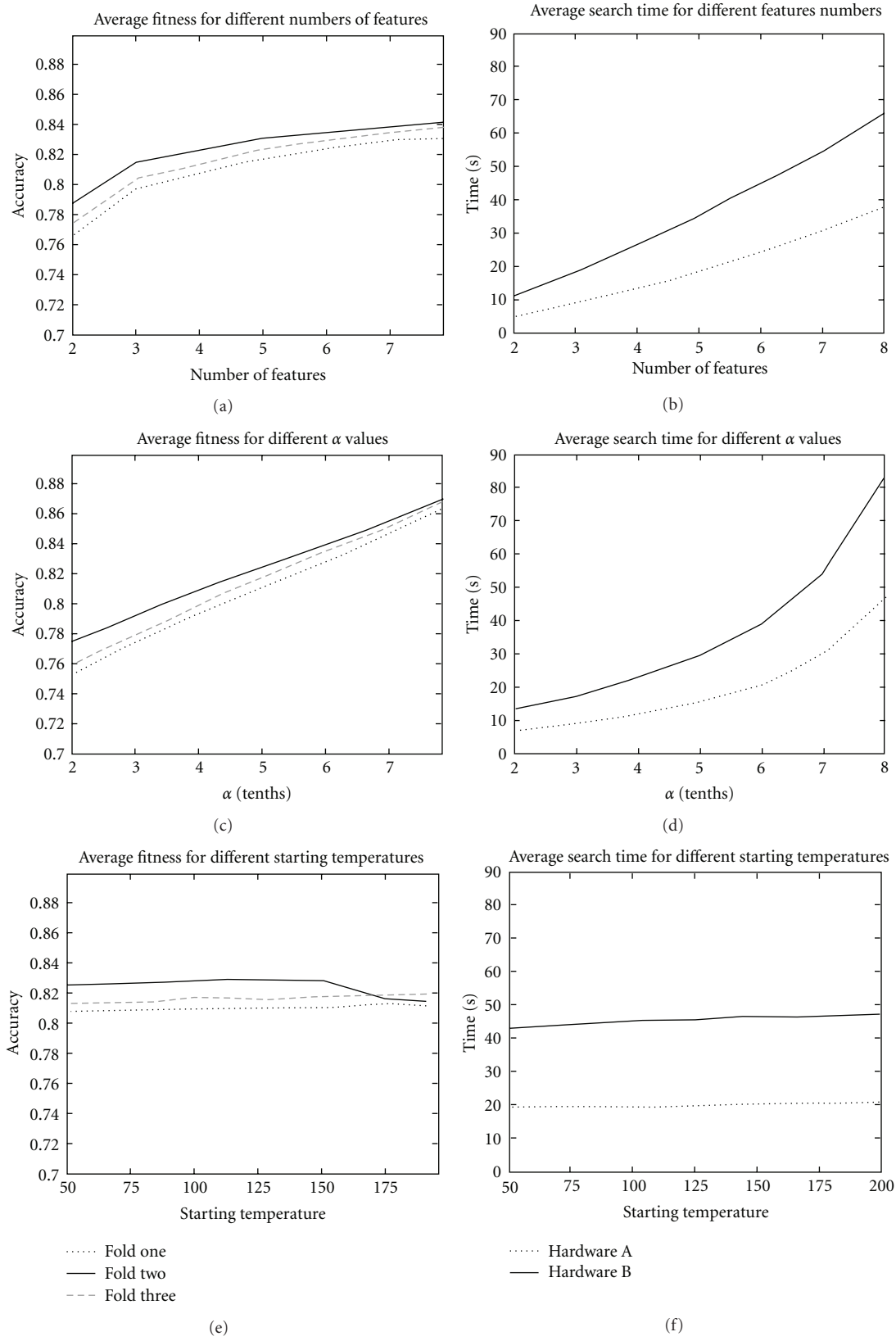


Figure 9: Effect of different parameter settings on overall classification accuracy and search time for Simulated Annealing on the Bos Wars data set.

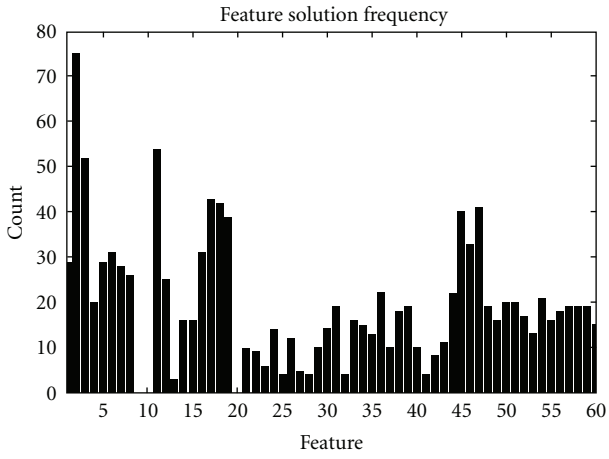


Figure 10: The frequency of each feature in the 150 SA solutions evaluated on the Bos Wars Test Set.

Table 6: Results for the solutions found with the best parameters by the deterministic and stochastic algorithms.

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

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 is 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 is required to generate each solution from the training data, is presented in Table 6. Accuracy is the average accuracy on the Bos Wars test data, on which neither algorithm is allowed to train. Search Time is the average time in seconds of an average search with the best performing parameters on Hardware Configuration A.

7. Conclusion

The results are unequivocal: the stochastic algorithm outperforms the deterministic algorithm on both performance metrics. In the RTSPP domain, a near-optimal solution to the original problem is better than an optimal solution to the reduced-dimension problem.

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 more complicated to refine or tune the algorithm for a specific RTSPP search landscape.

On the other hand, the SA solution exposes information about the problem domain. Figure 10 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 subset 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.

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 cannot be more than 5% below this value. Good solutions can be found in many different sections of the solution space since the RTSPP solution space landscape is jagged.

The failure of the BC metric to generate good classification accuracies for the deterministic solution indicates that the features are dependent. Features work in combinations to determine the outcome of an RTS game.

This study was conducted to determine the characteristics of the RTSPP. While the stochastic search method was able to find good classification accuracies that was not our main objective, instead, we used the results to determine the characteristics of the space, which allows us to develop a search algorithm tailored to our specific RTSPP problem.

The deterministic search tries to find a heuristic. In many searches, a heuristic is used to guide the search in profitable directions. If admissible, it can also be used to implicitly search much of the domain, using a best-first search strategy like A^* or Z^* [31]. The heuristic could reduce search time, allowing the entire domain to be explored in a reasonable amount of time through pruning.

In the RTSPP, we do not have that luxury. No admissible heuristic could be found. Instead, we used a heuristic to reduce the size of the solution space. Our hope was the heuristic would preserve the high fitness solutions in the space, while discarding the lower fitness solutions. For example, if the entire problem domain looked as in Figure 1, then the reduced solution space looks as in Figure 2. In this pedagogical example, we accomplish our goal. The heuristic makes it so the reduced solution space can be completely explored, and the reduced solution space retains all the high fitness solutions from the original solution space.

Our results show this does not work for the RTSPP. The stochastic algorithm is allowed to search the entire space. Even though it is only able to explore a small portion of solutions on each run, it finds solutions superior to those from the deterministic solution. Instead of the ideal reduced solution space, we have found a space looking more like Figure 3. We have removed some of the low fitness solutions, but have not retained the high fitness solutions.

The stochastic search results tell us the solution space is quite jagged and rough. However, it also tells us the fitness of the solution at the top of each *ridge* is similar. While we do not know the fitness of the optimal solution in the domain, we know we can use a simple hill climbing approach to find a high fitness solution. The solution found is composed of different features and centers on every iteration, but has a similar fitness, as demonstrated by the low standard deviation between the fitness of the SA solutions. In the RTS domain, this is an intuitive result: there are many different

strategies which can be pursued to win an RTS game, each one equally valid!

8. Future Work

Our goal is to use the understanding of the solution space characteristics determined in this study and develop a more complicated RTSPP algorithm. This innovative generic RTSPP method would employ a hybrid genetic algorithm/evolutionary strategy [34, 35]. This algorithm would be tested on the Bos Wars data as well as data obtained from the more complicated RTS game platform called *Spring* [13] or another available platform.

Specific to the RTS game domain, Bakkes et al. [13] created an evaluation function for the RTS platform called *Spring Engine*, where perfect knowledge of the environment is not available. *Temporal difference learning* is used to create an appropriate weighting for two features, "number of units observed of each type" and "safety of tactical positions". In [36], the same authors used five different features to accomplish the same basic goal. Like us, they hope to use their evaluation function to help drive improvements in adaptive RTS games. We hope to develop a more formal method of feature selection and allow this feature selection to correctly determine an appropriate strategy for an RTS game. Additionally, instead of temporal difference learning to determine appropriate weights for the features discovered, we desire to characterize winning/losing game states in terms of their location in n-space, where n is the number of features selected: a strategic approach. We would take classifiers generated for the Spring platform and use them as the foundation for a strategy-based agent which would generate and execute counter-strategies for a given opponent. Also, using a time-delay window of the past n snapshots should be address instead of the single snapshot.

Acknowledgment

This investigation is a research effort of the AFIT Center for Cyberspace Research (CCR), Director: Dr. Rick Raines.

References

[1] B. Geryk, *A History of Real-Time Strategy Games*, GameSpot, 2008.

[2] S. M. Lucas and G. Kendall, "Evolutionary computation and games," *IEEE Computational Intelligence Magazine*, vol. 1, no. 1, pp. 10–18, 2006.

[3] S. Ontañón, K. Mishra, N. Sugandh, and A. Ram, "Case-based planning and execution for real-time strategy games," in *Proceedings of the 7th International Conference on Case-Based Reasoning*, vol. 4626 of *Lecture Notes in Computer Science*, pp. 164–178, Springer, Berlin, Germany, 2007.

[4] D. W. Aha1, M. Molineaux, and M. Ponsen, "Learning to win: casebased plan selection in a RTS game," in *Proceedings of the 6th International Conference on Case-Based Reasoning (ICCBR '05)*, H. Muoz-Avila and F. Ricci, Eds., pp. 5–20, Springer, 2005.

[5] M. Sharma, M. Holmes, J. Santamaria, A. Irani, C. Isbell, and A. Ram, "Transfer learning in real-time strategy games using hybrid CBR/RL," in *International Joint Conference on Artificial Intelligence*, 2007.

[6] T. Graepel, R. Herbrich, and J. Gold, "Learning to fight," in *Proceedings of Computer Games: Artificial Intelligence, Design and Education (CGAIDE '04)*, Q. Mehdi, N. Gough, and D. Al-Dabass, Eds., pp. 193–200, 2004.

[7] M. Molineaux, D. W. Aha, and P. Moore, "Learning continuous action models in a real-time strategy environment," in *Proceedings of the 21th International Florida Artificial Intelligence Research Society Conference (FLAIRS '08)*, pp. 257–262, AAAI Press, May 2008.

[8] P. Spronck, M. Ponsen, I. Sprinkhuizen-Kuyper, and E. Postma, "Adaptive game AI with dynamic scripting," *Machine Learning*, vol. 63, no. 3, pp. 217–248, 2006.

[9] E. Kok, *Adaptive reinforcement learning agents in RTS games*, M.S. thesis, University Utrecht, Utrecht, The Netherlands, 2008.

[10] M. Chung, M. Buro, and J. Schaeffer, "Monte Carlo planning in rts games," in *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 2005.

[11] D. H. Wolpert and W. G. Macready, "No free lunch theorems for optimization," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 67–82, 1997.

[12] F. Beerten, J. Salmon, L. Taulelle, F. Loeffler, N. Mistry, and T. Penfold, "Bos wars. Open Source Software," 2008, http://www.boswars.org/.

[13] S. Bakkes, P. Kerbusch, P. Spronck, and J. van den Herik, "Automatically evaluating the status of an rts game," in *Proceedings of the Workshop on Reasoning, Representation, and Learning in Computer Games (IJCAI '05)*, 2005.

[14] R. Miikkulainen, B. D. Bryant, R. Cornelius, I. V. Karpov, K. O. Stanley, and C. H. Yong, "Computational intelligence in games," in *Computational Intelligence: Principles and Practice*, G. Y. Yen and D. B. Fogel, Eds., IEEE Computational Intelligence Society, Piscataway, NJ, USA, 2006.

[15] A. L. Blum and P. Langley, "Selection of relevant features and examples in machine learning," *Artificial Intelligence*, vol. 97, no. 1-2, pp. 245–271, 1997.

[16] A. K. Jain, M. N. Murty, and P. J. Flynn, "Data clustering: a review," *ACM Computing Surveys*, vol. 31, no. 3, pp. 316–323, 1999.

[17] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman, New York, NY, USA, 1979.

[18] P. Somol, P. Pudil, and J. Kittler, "Fast branch & bound algorithms for optimal feature selection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, no. 7, pp. 900–912, 2004.

[19] J. Jarmulak and S. Craw, "Genetic algorithms for feature selection and weighting," in *Proceedings of the Workshop on Automating the Construction of Case Based Reasoners (IJCAI '99)*, 1999.

[20] R. Meiri and J. Zahavi, "Using simulated annealing to optimize the feature selection problem in marketing applications," *European Journal of Operational Research*, vol. 171, no. 3, pp. 842–858, 2006.

[21] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[22] S. B. Kotsiantis, "Supervised machine learning: a review of classification techniques," *Informatica*, vol. 31, no. 3, pp. 249–268, 2007.

[23] A. Champandard, *AI Game Development: Synthetic Creatures with Learning and Reactive Behaviors*, New Riders, 2003.

[24] J. H. Friedman, "Regularized discriminant analysis," *Journal of the American Statistical Association*, vol. 84, no. 405, pp. 165–175, 1989.

[25] B. Cestnik, I. Kononenko, and I. Bratko, "Assistant 86: a knowledgeelicitation tool for sophisticated users," in *Proceedings of the 2nd European Working Session on Learning*, pp. 31–45, 1987.

[26] E. Larry Bull, *Advances in Learning Classifier Systems*, Springer, New York, NY, USA, 2004.

[27] R. L. de Mantaras and E. Armengol, "Machine learning from examples: inductive and lazy methods," *Data & Knowledge Engineering*, vol. 25, no. 1-2, pp. 99–123, 1998.

[28] E. E. Smith and D. Medin, *Categories and Concepts*, Harvard University Press, Cambridge, Mass, USA, 1981.

[29] S. Ridella, S. Rovetta, and R. Zunino, "K-winner machines for pattern classification," *IEEE Transactions on Neural Networks*, vol. 12, no. 2, pp. 371–385, 2001.

[30] D. J. C. MacKay, *Information Theory, Inference, and Learning Algorithms*, Cambridge University Press, Cambridge, UK, 2003.

[31] J. Pearl, *Heuristics*, Addison-Wesley, New York, NY, USA, 1984.

[32] F. J. Aherne, N. A. Thacker, and P. I. Rockett, "The Bhattacharyya metric as an absolute similarity measure for frequency coded data," *Kybernetika*, vol. 34, no. 4, pp. 363–368, 1998.

[33] E. Aarts and J. K. Lenstra, *Local Seach in Combinatorial Optimization*, Wiley, New York, NY, USA, 1997.

[34] K. Weissgerber, B. Borghetti, G. Lamont, and M. Mendenhall, "Towards automated feature selection in real time strategy games," in *GAMEON-NA Conference*, August 2009.

[35] K. Weissgerber, B. J. Borghetti, and G. L. Peterson, "An effective and efficient real time strategy agent," in *Proceedings of the 23rd Annual Florida Artificial Intelligence Research Society Conference*, 2010.

[36] S. Bakkes, P. Spronck, and J. van den Herik, "Phase-dependent evaluation in RTS games," in *Proceedings of the 19th Belgian-Dutch Conference on Artificial Intelligence*, pp. 3–10, 2007.