Shaul Markovitch and Yaron Sella
Computer Science Department, Technion, Haifa 32000, Israel
e-mail : shaulm@cs.technion.ac.il, sella@cs.technion.ac.il
Tel: 972-4-294346

Computational Intelligence, Volume 12, Number 1, 1996

LEARNING OF RESOURCE ALLOCATION STRATEGIES FOR GAME PLAYING

ABSTRACT

Human chess players exhibit a large variation in the amount of time they allocate for each move. Yet, the problem of devising resource allocation strategies for game playing has not received enough attention. In this paper we present a framework for studying resource allocation strategies. We define allocation strategy and identify three major types of strategies: static, semi-dynamic, and dynamic. We then describe a method for learning semi-dynamic strategies from self–generated examples. We present an algorithm for assigning classes to the examples based on the utility of investing extra resources. The method was implemented in the domain of checkers, and experimental results show that it is able to learn strategies that improve game-playing performance.

Key words: Resource–allocation, learning, class-assignment, dynamic-classification, game-playing, time, checkers

1. INTRODUCTION

It is very common to see chess masters spend a considerable amount of time over complicated/crucial board positions, while replying almost instantaneously to others. Indeed, the amount of resources devoted to a move strongly determines its quality. This is the reason why errors are much more frequent in blitz games than in regular games. It is therefore very important for players to allocate their resources wisely. However, determining which board positions deserve more resources is not always easy.

While substantial research has been done about search techniques in game-playing, only a few attempts to study the problem of resource allocation in game playing have been made. The main goal of using iterative-deepening algorithm in game-playing is being able to return from the search with a reasonable decision when the allocated time is over. Hyatt[1984] describes time allocation techniques that were incorporated into Cray-Blitz. The main idea there was to use the opponent's thinking time for planning, and thus save time. The program was also instructed to allocate more time when it observed loss of material. Levy and Newborn [1991] describe a more general algorithm for time control. The procedure employs iterative deepening [Korf, 1985], with breakpoints in predefined time points, where the program considers whether to keep on searching or to halt the search. Although implementation of these methods proved beneficial in practice, none of these works examined the problem of resource allocation thoroughly. No alternative methods were considered and therefore, no comparative study was reported.

The problem of resource allocation was studied more thoroughly in the fields of planning and decision–theoretic control of inference. Horvitz [1987] studies the problem of finding approximate solutions under time constraints. He suggests to use procedures that exhibit graceful responses to diminishing resources. In our work, we assume that minimax is indeed such a procedure and concentrate on deciding how to allocate the resources to it.

At this point it is worth mentioning the phenomenon of pathology in games that was first reported in [Nau, 1980]. The pathology predicts that the minimax procedure will perform worse as the search depth increases, since the errors of the evaluation function will accumulate. Although researchers have taken different approaches when trying to explain why minimax works despite the game searching pathology [Nau, 1982; Pearl, 1983; Scheucher and Kaindel, 1989], most of them agree that in practice the problem rarely can be seen.

Boddy and Dean [1988; 1989] study the problem of time-dependent planning problems. They develop algorithms for solving path-planning problems that return better solutions when allocated more resources (*anytime algorithms*). They also develop algorithms for allocating resources in this domain (which they call *deliberation scheduling*).

The research described here has the following goals:

- Understanding the problem of resource allocation in the context of game playing.
- Studying different types of possible resource allocation strategies.
- Developing a research methodology under which resource allocation strategies could be devised, evaluated and compared.
- Developing a game-independent method for automatically acquiring resource allocation strategies.

A scheme that controls resource allocation must be efficient. This means that the factors that it considers should be easy to compute and that its own execution demands are modest, so that it does not consume too much of the resource while deciding how to use it. Of course, the construction of the scheme can be very expensive, but this is done once prior to game-time and has no effect in terms of resource waste during the game.

We start this work by classifying resource allocation strategies according to their execution pattern. A resource allocation strategy that is executed once at the beginning of the game is very cheap to execute but can use very little knowledge about the course of the game. A resource allocation strategy that is executed at each search step is very knowledgeable but is also very expensive to use. We have concentrated on allocation strategies that are executed once before each move calculation, compromising between cost and knowledge. To make this research feasible we define a simplified framework where a resource allocation strategy is a sequence of binary decisions. Each of these decisions determines whether the next move requires a regular resource allocation or a larger one.

The main part of this work deals with the automatic acquisition of a resource allocation strategy. We would like to develop a learning procedure that can take the move-generator of a game together with a static evaluation function and produce a resource allocation strategy. The work described here presents a good case-study for machine learning researchers. We show that in complex domains a learning system may need to perform tasks that are usually assumed to be carried out by an external teacher, such as examples generation, class assignment and feature generation. Furthermore, we show an example where the traditional way of using classifiers is not appropriate. When using a classifier in the context of resource allocation procedure, we would like the classifier to return a positive decision (for investing extra resource) with higher probability when the player has more resources left to spend. We developed a scheme for learning such a context-sensitive classifier.

Finally, we conducted a set of experiments to test our methodology. The experiments involved the comparison of various resource allocation strategies.

The remainder of this paper is organized as follows : In Section 2 we discuss different types of allocation strategies on a knowledge/cost scale. In Section 3 we describe a general methodology for automatically acquiring semi-dynamic strategies, and describe an implementation of this methodology in Section 4. In Section 5 we describe our methodology for experimentation and Section 6 describes the results. Section 7 concludes.

2. RESOURCE ALLOCATION STRATEGIES

Assume that an agent is facing a sequence of tasks that it intends to perform. Assume that the results of executing a task depend on the amount of resources devoted to the execution. Assume further that the agent has an upper bound on the total amount of resources it can use for the whole sequence. A resource allocation strategy is an algorithm that decides how to distribute the resources among the tasks. Assuming the existence of some criterion for evaluating the performance of the task sequence, we can compare strategies based on the performance that they yield. We assume that the evaluation criterion is non-decreasing monotonic, i.e., that investing more resources cannot lead to deterioration in performance.

The research described here focuses on two-player perfect information games (e.g., Chess, Checkers). Two-players perfect information games are a good domain for studying different strategies of resource allocation because the performance of two strategies can be easily compared by letting programs that use the strategies play against each other.

In the game-playing context, it is very common to limit the total time that players can spend for each k moves. The tasks are the move calculations required while the game is played. A resource allocation strategy for game-playing programs is an algorithm that decides how many resources the program should spend on the calculation of each move. A minimax procedure that is allocated more resources for a search is able to search deeper and therefore reach better decisions.

Strategies can be divided into three groups :

- Static strategies – Strategies that decide how the resource should be allocated *before* starting the game.
- Semi-Dynamic strategies – Strategies that decide before each move calculation is performed how many of the resources will be allocated to that move.
- Dynamic strategies – Strategies that can communicate with the move calculation process and update their resource allocation decisions while that process is being carried out.

These types of strategies could be viewed in light of the knowledge available to them and with respect to the resource demands of the strategy execution itself. In the next three subsections we will discuss these three classes of strategies.

2.1. Static Strategies

A static strategy decides upon the resource allocation before the game starts. Naturally, such a strategy has no information about the course of the coming game, and will therefore always yield the same resource allocation sequence. Such a strategy is extremely cheap: It should only be applied once, and the resulting sequence can be used for all future contests. However, it is very unlikely that a single sequence fits all the possible courses of the game. If the strategy is given a model of the opponent [Carmel and Markovitch, 1994] in some form, it can produce an allocation sequence based on the model. Such a static strategy will still be very cheap since it is called only once at the beginning of the game.

2.2. Dynamic Strategies

Dynamic strategies are located on the other end of the knowledge scale. They can use the information gathered during the search to decide the amount of resources allocated for the search. They can thus yield good allocation sequences that are based on a large amount of knowledge. The main problem with such strategies is their cost. In the extreme case the strategy can be called for each node in the search tree.

There are various dynamic strategies employed by existing game-playing programs. The most famous one is the *quiescence search* strategy [Beal, 1990] that keeps on searching branches as long as there are drastic changes in values of nodes in the search tree. Another selective deepening method is called *singular extension* [Anantharaman *et al.*, 1990]. The method conducts a secondary search under a leaf node that dominates its siblings, and has therefore greater influence on the search outcome.

2.3. Semi-dynamic Strategies

Semi-dynamic strategies have access to the board that is at the root of the search tree. Thus, they have much more knowledge than static strategies. Yet, the strategy is executed once for every move and is therefore much cheaper than dynamic strategies.

It is hard to tell what properties of the board should affect the decision of the resource allocation strategy. It is reasonable to devote more resources when the player is in a much inferior position so that it can get out of trouble. But it is also reasonable to devote more resources when the player is in a good position when the right (but hard to find) move will lead it to a victory. Another factor that may affect the resource allocation decision is the complexity of the situation. In a complex situation, the search procedure should probably be allocated more resources.

Finding an algorithm, that can consider all the relevant properties of a board to decide the amount of resources to allocate, is a difficult task. In the following section we will present a methodology that learns such an allocation strategy from examples.

In addition to the board itself, a semi-dynamic strategy can also base its decision on the history of the game. Human players, for example, devote more resources to the computation of a move after the opponent has made an unexpected move. That type of reasoning requires some model of the opponent. We are not considering opponent modeling in this research.

2.4. Research Framework

For a continuous resource, there are an infinite number of ways to partition it into resource sequences. In order to make the research more feasible, we have devised a simplified model of resource allocation strategies:

- The player searches the game tree using minimax procedures with alpha-beta pruning.
- No selective deepening techniques (like quiescence search) are employed.
- The game is stopped after a fixed number of moves (m).
- The player has two resource allocation options: either searching to depth k or searching to depth $k + n$ ($n > 0$).
- The player is allowed to perform the deeper search at most d times, where $d < m$.

Under the above model the output of a strategy is simply a vector $V_1, V_2, \ldots, V_m$, where each V_i can be either *True* or *False*, marking that the strategy decides to perform the deeper search or does not accordingly. The number of V_i's which are *True* must be $\leq d$. Under this model, the total number of allocation sequences that a strategy can produce is $\sum_{i=0}^{d} \left(\begin{array}{c} m \\ i \end{array} \right)$.

In the conclusions we discuss the relaxation of these assumptions.

3. LEARNING SEMI-DYNAMIC STRATEGIES

The knowledge that a strategy uses to make decisions can be domain-independent or domain specific. The advantage of devising a strategy based solely on domain independent knowledge is that such a strategy is very general and is applicable to any game. However, it is clear that a strategy using both domain independent knowledge and domain-specific knowledge can yield better performance. The problem is that obtaining domain-specific knowledge and incorporating it into a strategy is not an easy task.

One way of overcoming these problems is to build a system that learns good strategies automatically. The input for this learning system should be a game, or more specifically a problem solver designed to play a game, and the bounded resource to be allocated. The output of the learning system should be a strategy for allocating the resource.

3.1. Semi-dynamic Strategy Learning as a Classification Problem

We suggest the following general methodology for constructing the learning system :

1. Generate many examples where each example is a board and a correct decision regarding resource allocation for that board.
2. Find general rules that predict which boards require more resources than others.

Assuming that a board is represented by a set of features describing some of its properties, and recalling that in our model all the decisions regarding resource allocation are binary,

the problem can be formulated as a classification problem. Thus, in stage 2 of the proposed methodology, any of the known classification algorithms can be employed. What is left to be determined in further details is stage 1, namely, how should the training examples be generated. Four questions arise:

1. From where will the training examples come?
2. How will the class be determined?
3. How will the features representing a board be determined?
4. How will the learned classifier be used in the context of resource allocation?

Starting with the first question, the examples must obviously be realistic, i.e., boards that are likely to be encountered in real games. Hence, it seems natural that the examples will be generated by the problem solver during the course of real games. The other three questions will be discussed in the following subsections.

3.2. Assigning Classes to Examples

The concept that we want our program to learn is "boards that worth extra search efforts". We use the common terminology of machine learning and call boards that belong to the concept "positive" and those that do not belong "negative". Thus, saying that a board is negative does not imply that using extra resources for this board carries negative utility, but only implies that the it is not a board that worth extra search resources.

Ideally, to determine if a board is positive (should be given extra resource), one should compare the expected outcome of the game when the best move is chosen first with, and then without extra resources, while the rest of the moves stay intact. Unfortunately, this is not possible, because once a different move is chosen, the rest of the game takes a different course.

Alternatively, one can compare the expected outcome on a more local basis. Namely, compare the expected outcomes of the best moves with and without extra resources. Let b be a board, $move_k(b)$ be the best move in a search to depth k and $class(b)$ be the class assigned to b. We suggest the following methodology to determine the class of a board:

$$class(b) = \begin{cases} \text{positive} & move_k(b) \neq move_{k+n}(b) \\ \text{negative} & \text{otherwise} \end{cases} \quad (1)$$

The above method assigns each board a discrete class, either positive or negative. More generally, we can estimate the utility of a depth $k+n$ search, and use it as a measure for the positiveness of the board. Let us mark by $V_k(m)$ the minimax value assigned to move m by a depth k search. The class assigned to b will be :

$$class(b) = V_{k+n}(move_{k+n}(b)) - V_{k+n}(move_k(b)) \quad (2)$$

We measure the utility as being the difference between the values of the best moves chosen by depth $k+n$ and k searches. The values are expressed in terms of the evaluation function of the problem solver. Note that if $move_k(b) = move_{k+n}(b)$, the utility will be 0. The values of both moves should be taken from the best estimates available. Assuming that a search to depth $k+n$ "knows better" than a search to depth k, we take both values from the former. Since $move_{k+n}(b)$ is the best move of a $k+n$ depth search, $class(b)$ can never be negative. Note also that this measure is unlikely to be discrete. Therefore, incorporating it into the system as the class requires a classification algorithm that can process continuous classes.

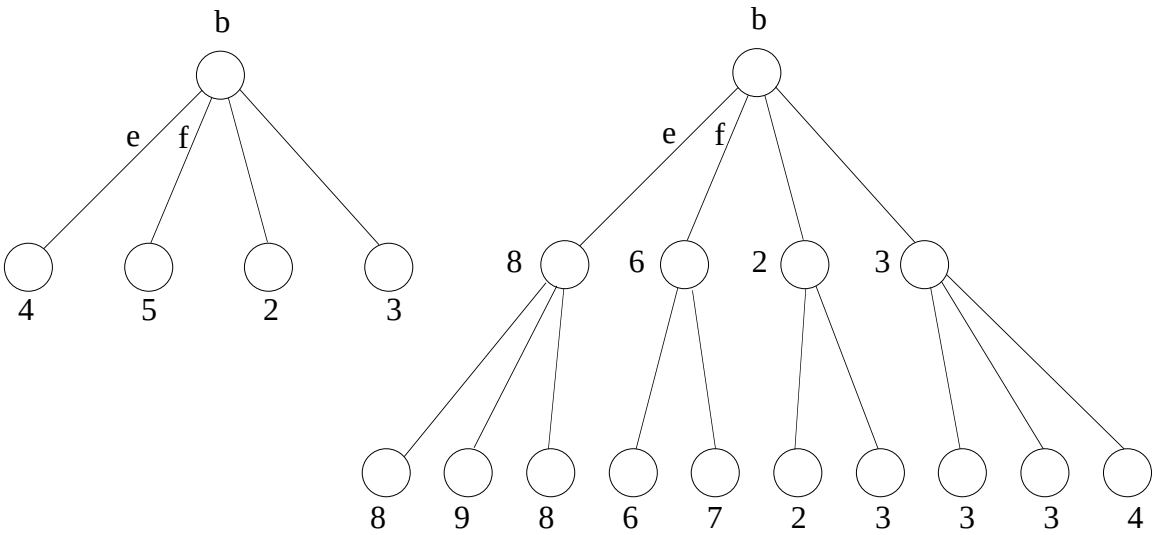

Figure 1. Example for the operation of the class assignment procedure. A search to depth 1 selects move f while a search to depth 2 selects move e. Therefore, this board is a positive example.

Figure 1 demonstrates how the class assignment procedure works when only one best move can be chosen after the search. For this example we assume that $k = n = 1$. The two game–trees show how, from the same board–position b, the recommended move can change as the depth of search varies. In a search to depth 1, the move f turns out to be the best move, therefore $move_1(b) = f$. In a search to depth 2, the move e turns out to be the best move, therefore $move_2(b) = e$. Hence $move_1(b) \neq move_2(b)$. Thus, according to equation (1), $class(b)$ is set to be positive. Applying equation (2) to the same example, we get $V_2(e) = 8$, $V_2(f) = 6$ which yields $class(b) = 2$.

Yet more generally, it is possible that upon searching to some depth there will be more than one best move,i.e., two or more moves will have the same (best) minimax values. We shall change the notation to $moves_k(b)$ and $moves_{k+n}(b)$ to mark that these entities are sets. Within the discrete class paradigm, the following criterion should be used in assigning a class to a board:

$$class(b) = \begin{cases} \text{positive} & moves_k(b) - moves_{k+n}(b) \neq \emptyset \\ \text{negative} & \text{otherwise} \end{cases} \tag{3}$$

The board is positive if and only if depth $k + n$ search can be beneficial or, in other words, there is a chance that a depth k search will choose one of the wrong moves. Note that using $moves_k(b) \neq moves_{k+n}(b)$ is not good enough because it includes the case $moves_k(b) \subset moves_{k+n}(b)$. In the latter, although depth k search can choose from a smaller set, this set does not include any wrong moves, thus yielding that any move chosen by a depth k search is correct.

Within the continuous class paradigm, the difference is now measured between sets and for each of these sets we have to calculate the value of the expected outcome. For the set

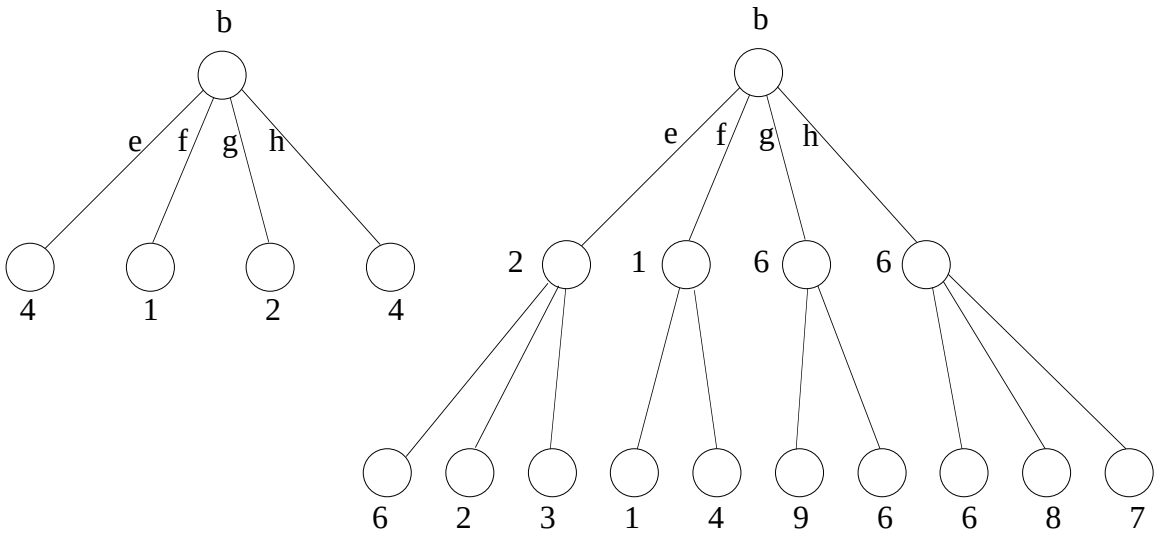

Figure 2. Example for the operation of the class assignment procedure. A search to depth 1 can lead to the selection of move e which is suboptimal. Therefore, this board is a positive example

$moves_{k+n}(b)$, all its members have the same minimax value (in terms of depth $k+n$ search) by definition. Therefore, the value of the expected outcome of a depth $k+n$ search can be taken as the value of any member $m' \in moves_{k+n}(b)$. For the set $moves_k(b)$, although all its members have identical V_k values, they might have different V_{k+n} values. Therefore, the class assigned to b will be:

$$class(b) = \ V_{k+n}(m') - \frac{\sum_{m \in moves_k(b)} V_{k+n}(m)}{|moves_k(b)|} \tag{4}$$

Figure 2 demonstrates how the class assignment procedure works when more than one best move can be chosen after the search. Again, we assume for the example that $k = n = 1$. After a search to depth 1, the moves $\{e, h\}$ form the set of possible best moves. Therefore $moves_1(b) = \{e, h\}$. After a search to depth 2, the moves $\{g, h\}$ form the set of possible best moves. Therefore $moves_2(b) = \{g, h\}$. Hence, $moves_1(b) - moves_2(b) \neq \emptyset$ holds, and thus, according to equation (3), $class(b)$ is set to be positive. Applying equation (4) to the same example, we get $V_2(h) = 6$, $V_2(e) = 2$ which yields $class(b) = 2$. Note that calculating $moves_{k+n}(b)$ requires a minimax search to depth $k+n$ *without* alpha-beta cut-offs at the top level nodes. In this research a discrete class paradigm was used, and equation 3 was adopted as the class assignment procedure.

3.3. Extracting Features from Examples

Features can be domain-independent (e.g., number of possible moves) or domain-specific (e.g., number of kings on the last row). Determining which features should be extracted from the board, especially for domain-specific features, is not trivial. Some of the problems arising

in that matter are:

- The features must have some predictive power regarding the class.
- The features should be easy to compute to prevent high overhead of the strategy execution.
- There should not be too many features. Too many features (especially irrelevant ones) might damage the quality of the learned rules (over-fitting [Schaffer, 1992]), and they also increase strategy execution overhead since their values have to be calculated during game-time.

It should be noted at this point, that the existence of a relatively-small, cheap-to-compute and highly-predictive set of features is not guaranteed. Ideally, a learning system should be able to generate features automatically (a process called *constructive induction*). In this research, some of the features are given as input to the learning system and some are self generated. The method that was used for generating features is best explained in the context of the checkers domain and will therefore be described in the next section.

Usually, one does not know in advance which features of an example are relevant to the target concept and which are not. Many candidates for features can be thought of, but due to bad effects of irrelevant features it is important that such features will be filtered out.

In recent years, the problem of irrelevant features in inductive learning received more attention, and some feature-selection algorithms emerged (such algorithms perform *attention filtering* according to the framework described in [Markovitch and Scott, 1993]). In the experiments described in section 5, we used a feature selection algorithm called RELIEF [Kira and Rendell, 1992] that attempts to eliminate features statistically irrelevant to the class.

3.4. Using Context-sensitive Classifiers in Resource Allocation Context

As noted earlier, many classification algorithms are known, and any of them can be used as the learning module of the system. For the experiments described in the next section, we used a variant of ID3 [Quinlan, 1986] coupled with RELIEF. Decision-trees are relatively cheap classifiers making them an attractive choice for resource allocation usage. In order to decide whether to invest extra resources, the program need only pass the current board through the decision tree and invest the extra resources only when the board is classified as positive.

There is one problem with the above method. It does not take into account the amount of resources still available. If many resources are available relative to the time left till the end of the game, we would like our allocation strategy to be less selective and to classify boards as positive more often. If the resources are scarce we would like the strategy to be more careful before deciding that boards are positive.

We have devised a method that uses probabilistic decision trees to make the allocation strategy behave in the way described above. The main idea is to have a global soft threshold that determines which leaves are positive and which are negative. The value of the threshold is determined according to the remaining resources. We call such a decision tree a *context-sensitive classifier*.

Assume that each leaf l of the decision tree contains a pair of numbers $\langle W(l), P(l)\rangle$ where $0 \leq W(l) \leq 1$ is the probability of an individual board to belong to the leaf and $0 \leq P(l) \leq 1$ is the probability of a board that belongs to the leaf to be positive. Let $L(T)$ be the set of all leaves of a decision tree T. We know that $L(T)$ forms a partition of the board population (every board belongs to exactly one leaf). Therefore, $\sum_{l \in L(T)} W(l) = 1$.

Assume that $0 \leq b \leq 1$ is a global threshold. A leaf l classifies a board as positive iff $P(l) \leq b$. When b increases the number of "positive" leaves increases as well. The a-priory probability of an arbitrary member of the board population to be classified as positive is therefore directly dependent on the threshold b.

Assume that during a game, $\overline{m}$ moves are left to play with $\overline{d}$ deep searches still available. We would like the resource allocation strategy to decide on making a deep search with probability $\frac{\overline{d}}{\overline{m}}$. We compute what value of threshold makes the portion of the population that is classified as positive equal to $\frac{\overline{d}}{\overline{m}}$. This is the minimal b that satisfies

$$\left[\sum_{P(l)\geq b} W(l)\right] \geq \frac{\overline{d}}{\overline{m}} \tag{5}$$

When the resource allocation procedure is called, it computes b and passes the current board through the decision tree. Only if the leaf that it reaches satisfies $P(l) \geq b$, will the procedure allocate extra resources. After the move is performed, the values of $\overline{d}$ and $\overline{m}$ are recalculated, and the process continues. This process can be viewed as placing a dynamic threshold on the positiveness of leaves that is changed according to $\overline{d}$ and $\overline{m}$. Since computing b using equation (5) before every move is too expensive, we perform a preprocessing stage after acquiring the decision tree. A table is constructed that specifies what is the positiveness threshold for a range of possible values of $\frac{\overline{d}}{\overline{m}}$.

The values of $W(l)$ are calculated during the creation of the decision tree. If the number of available examples is E and the number of examples that reach leaf l is $E(l)$ then we set $W(l) = \frac{E(l)}{E}$. If $P(l)$ positive examples reach a leaf l, then we set $P(l) = \frac{P(l)}{E(l)}$. To make these numbers more reliable, we add a stage after building the tree. In this stage a large number of examples are generated and passed through the tree, updating $P(l)$ and $W(l)$.

4. IMPLEMENTATION

The above methodology was implemented in a system whose architecture is illustrated in Figure 3. The learning system generates boards by having two instances of the player playing against each other. The boards are then assigned a class using equation 3. The system then extracts the features from the boards and feeds the classification program with the classified examples. The classification program generates classification rules. The performance system receives a board as input. It calls the resource allocation procedure that uses the learned classification rules to decide the resource allocation for the given board. The search procedure then conducts a minimax search using the resources allocated.

4.1. Domain

The domain that was chosen as a platform for the experiments is the game of Checkers. The free parameters, defined in subsection 2.4, were set as follows: basic search depth $(k) = 4$, additional search depth $(n) = 2$, maximum number of deep searches per game $(d) = 8$, and maximum number of paired moves per game $(m) = 40$. m was selected as a low number of moves that can still allow interesting games. d was selected to be a relatively small portion of m. We have considered using $k = 2$, but such a shallow search leads to uninteresting games. We also considered $k = 6$, but too many computation resources were required for experimentation with such a deep search.

When a game does not end after 40 paired moves, we call a judge who determines the

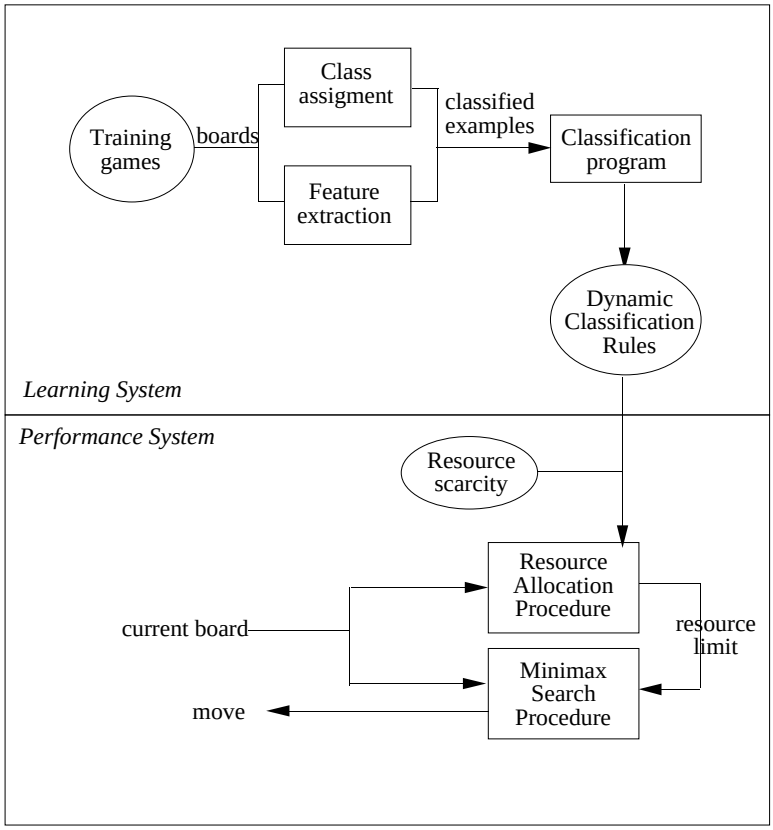

Figure 3. An architecture for a system that learns resources allocation strategies.

result according to the final position. If one side has a significant piece advantage (4 in our experiments) it is declared a winner, otherwise the game is a tie. A simple evaluation function was used on the leaves of the game-tree which relied only on piece advantage. It also made the player with an advantage more aggressive.

4.2. Automatic Feature Generation

We have implemented a scheme for automatic feature generation based on board patterns. These patterns were learned for the purpose of learning resource allocation strategy only - they were not used for the minimax evaluation function.

We have restricted the shape and size of patterns to make the search for good patterns feasible. A pattern consists of a 3×3 sub-board that contains 5 checkers squares. There are 18 positions on an 8×8 Checkers board in which a possible match for this pattern can be found. Each square in a pattern can have one of the following values : don't-care, free, any-white, any-black, white-pawn, white-king, black-pawn, black-king. The value don't-care is more general than the value any-white, which in turn is more general than the value white-pawn. The generalization relationship between all possible values of the squares is expressed by a tree whose description appears in figure 4.

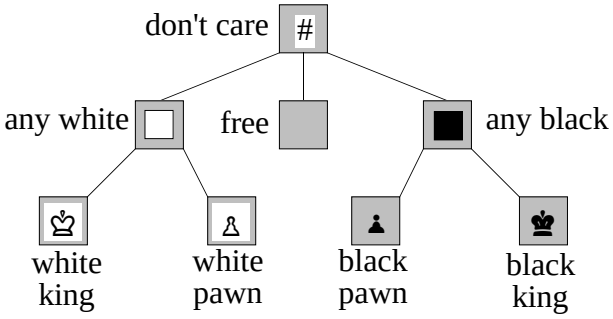

Figure 4. Generalization relationship of pattern squares values

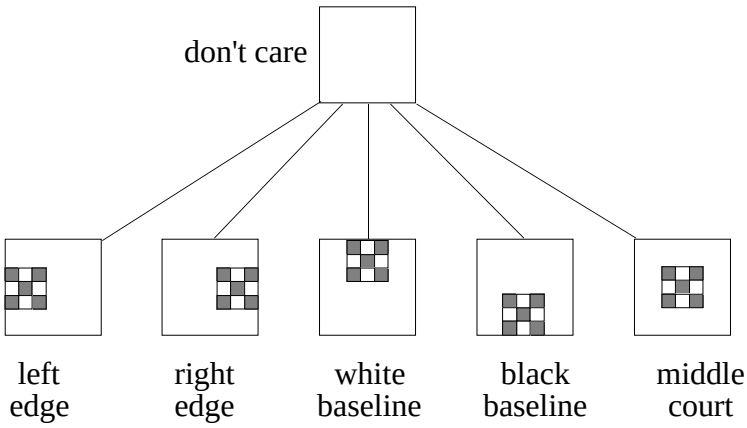

Figure 5. Generalization relationship of pattern position values

Another part of a pattern is its position on the board. The position component can take one of the following values : don't-care, white-baseline, black-baseline, middle-court, right-edge, left-edge. Figure 5 shows the position hierarchy. Figure 6 shows an example of a pattern where a white piece attacks a black piece.

Our pattern language is significantly weaker than the pattern language used by MORPH [Levinson, 1991]. We intend to use the reacher language of Levinson in our future research.

There are $6 \times 5^8 = 2343750$ different patterns. The search for good patterns is conducted using the best-first strategy. Starting with the most general pattern, the search procedure expands the best pattern and inserts its successors into the list of candidate patterns. The expansion function specializes the pattern in all possible ways. When the allocated search resources are exhausted, the best patterns can be taken for use in the classification task.

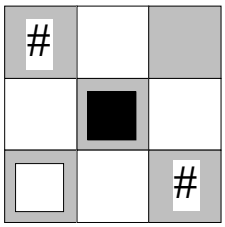

Figure 6. An example of a pattern: a white piece attacking a black piece

Patterns are evaluated using information theoretic methods similar to ID3's. Assume the existence of a large set of boards, SB, for which the class is known. In our case the class is either positive or negative. Let p be the pattern whose quality is to be evaluated. Let $max(p)$ be the maximum number of times that the pattern p can occur in a board. For each of the possible number of occurrences of p, $i \in [0..max(p)]$, let $pos(i)$ be the number of positive boards in SB in which p occurred exactly i times. Let $neg(i)$ be defined similarly for negative boards. The formula that gives a score to the pattern p with respect to the set SB is :

$$\text{Info}(p) = \sum_{i=0}^{max(p)} \frac{pos(i) + neg(i)}{|SB|} * I(pos(i), neg(i)) \tag{6}$$

$$I(p,n) = -\frac{p}{p+n} log_2(\frac{p}{p+n}) - \frac{n}{p+n} log_2(\frac{n}{p+n}) \tag{7}$$

Info(p) returns a real number in the range $[0.0, 1.0]$, which represents how much information will be needed to store SB when p is used as a discriminating feature regarding the class. When info(p) is closer to 0.0, it means that p is more informative with respect to the class of the examples in SB.

5. EXPERIMENTATION METHODOLOGY

In order to test the methodology that we have developed for learning resource allocation strategies, we have conducted a set of experiments in the Checkers domain. The next subsection will discuss the problem of evaluating the performance of game-playing programs. We then describe experiments that were performed to establish a lower and an upper bound on the improvement that a resource allocation strategy can possibly yield. Finally, we describe a sequence of experiments that tested the effectiveness of the various mechanisms described in previous sections.

5.1. Strategy Evaluation

In the work presented in this paper we evaluate the merit of the proposed algorithm for learning and using resource-allocation strategies by comparing its performance with that of the default resource allocation strategy that spreads its resources randomly. To make

the experiment more informative, we created a sequence of versions of our program with an increasing sophistication. Each successive version in the sequence contains additional mechanism.

To make such experimentation valid we need to pass these successive versions through a reliable test, and compare the results. The obvious choice for such a test is a sequence of games against some fixed opponent. The fixed opponent was set to be a player that searches to a fixed depth k (4 in our experiments) and uses the same evaluation function as all the players in our experiments. We call this opponent $F4$.

Most of the experimental work in the field indeed uses the results of actual games between programs to evaluate their performance. Many researchers use the USCF rating as a measurement for a program's ability based on games against programs with a known rating. However, the successive versions of the programs are different only in the resource allocation strategies that they use. Therefore, the performance of successive versions of the program is likely to be very similar and the test must have high reliability.

Let WHITE and BLACK be two game-playing programs. The performance of WHITE relative to BLACK based on n games is $\frac{W}{n}$, where W is the number of times that WHITE won[1]. The performance of WHITE relative to BLACK is $R(WHITE, BLACK) = \lim_{n\to\infty} \frac{W}{n}$.

In order to estimate $R(WHITE, BLACK)$, we need to perform a finite set of games between the two players. We can consider such a tournament as a sequence of binomial experiments and use standard statistical methods to calculate the number of experiments needed to achieve accurate estimates with high confidence. Based on initial experiments, we have decided to generate a sample that will be large enough to yield a confidence interval of 0.01 with confidence of 0.95. The number of experiments needed to achieve such confidence is 9604. Therefore, each test point in our experiments consisted of 10,000 games.

For any given resource allocation strategy R We can now define a quality value $Q(R)$ which is its relative performance to $F4$ based on 10,000 games. In order to prevent repetition of a small set of games throughout all the tournaments, whenever the minimax procedure scans the top–level–nodes of the game–tree, which are actually the list of possible moves a player can make in a given position, it does it in random order. This means that if some of the possible moves a player can make in a given position have the same minimax value, any one of them can be chosen.

5.2. Upper and Lower Bounds

One of the basic assumptions of this work is that a smart resource allocation strategy can improve the quality of play. Let G be the set of all possible resource allocation strategies (that satisfy the assumptions we set for our domain). In the first stage of our experiment, we would like to estimate the maximal and minimal elements of the set $Q(G) = \{Q(g)|g \in G\}$. The values of $\max(Q(G))$ and $\min(Q(G))$ will determine the possible range of improvement for our learning program.

It is easy to find $\min(Q(G))$. We measure the performance of a program that never uses extra resources. We call this strategy the *never* strategy. Since such a program is identical to $F4$ we expect its performance to be very close to 0.5. The assumptions in subsection 2.4 require that a strategy will decide on using extra resources *at most* d time, therefore $never \in G$. Its minimality depends on our monotonicity assumption which states that using additional resources can never decrease the performance of a program.

It is much harder to find $\max(Q(G))$. We can find an upper bound on $\max(Q(G))$ by

[1]For games that allow draw, we define W to be the number of times white wins plus half of the number of draws.

measuring the performance of the *always* strategy. A strategy that always decides on using extra resources. Note that $always \notin G$ since it does not satisfy our requirement of using extra resources d times at most.

We will also try to find a tight lower bound on $\max(Q(G))$. The strategy that will be used for setting the lower bound on $\max(Q(G))$ is the *oracle* strategy. This strategy calls $class(b)$, the class assignment procedure, for every root board. It performs a deep search for each board b with positive $class(b)$ as long as there are remaining extra resources. Therefore we know that $Q(oracle) \leq \max(Q(G)) \leq Q(always)$. Note that it is possible to have a strategy that is better than *oracle*. While the oracle strategy avoids investing extra resources when such an investment is useless, it does not necessarily pick the optimal set of k moves where extra resources should be invested. Note also that the while *oracle* is a legal member of G it is not acceptable as a practical resource allocation strategy since the resources used by the strategy itself significantly exceed the resource limit.

For control we also measure the performance of the *random* strategy which decides randomly whether to search deeper. The procedure selects at the beginning of the game a random set of k (=8) move indexes between 1 and m (=40). It then performs a deeper search for each move that was selected.

5.3. Learning Semi-dynamic Strategies

We have conducted a series of learning sessions to test the various components of the learning system. We started with the basic ID3 procedure, fed with $100,000$ examples, with 23 hand-crafted features and no filtering. The features were similar to those used by Samuel [1959]. The features were used only by the resource allocation strategy. Table 1 in the next section lists the 23 features. The evaluation function was fixed for the whole stage of experimentation and was based solely on piece advantage. We then enabled the RELIEF feature filter (using a set of 10,000 examples). The next step was to use a context-sensitive classifier. In order to make the information in the tree leaves more reliable we enriched it by passing a set of 500,000 examples through it. Finally, we allowed the system to learn patterns. The search procedure was allotted 1000 expansions and generated 8000 patterns. The best patterns were added to the given features.

6. RESULTS

The results of the experiments are summarized in table 2. One of the most notable result in the table is the performance of the *oracle* strategy. Its performance is very close to that of *always*. That means that a good resource allocation strategy can maintain similar performance while saving 80% of the resources (the *always* strategy uses extra resources in each move while the oracle strategy does so only 20% of the time).

These results also verifies our class assignment methodology. Based on the results, it is clear that if the system can acquire good classification rules its performance can get close to optimal (with respect to the set G of possible allocation strategies). The oracle strategy also helped us setting tight bounds on $\max(Q(G))$. Based on its performance we know that $0.683 \leq \max(Q(G)) \leq 0.700$. The RELIEF feature filter retained 17 relevant features. Table 1 shows the original features and those selected by RELIEF.

The range of possible values for $Q(R)$ is $[0.499, 0.700]$. Therefore, we can say that our full system achieved 50% of the possible improvement (34% of the possible improvement over the random strategy). The importance of finding the possible range of improvement can be appreciated when we look at the results of earlier set of experiments performed in

feature name	range	RELIEF
white legal moves	2 – 15	+
white total moves	2 – 16	+
black total moves	1 – 17	+
white mobile pieces	1 – 10	+
black mobile pieces	1 – 10	+
num of white pawns	0 – 12	+
num of white kings	0 – 4	+
num of black pawns	0 – 12	+
num of black kings	0 – 5	–
evaluation function	1 – 17	+
white on threat	0 – 1	–
white sacrifice	0 – 9	+
black sacrifice	0 – 15	+
white crown1	0 – 3	+
white crown2	0 – 3	+
white crown3	0 – 3	+
lonely black pawn	0 – 2	–
white triangle trap	0 – 2	–
black triangle trap	0 – 2	–
white center pieces	0 – 6	+
black center pieces	0 – 6	+
white base control	0 – 2	–
black base control	0 – 2	+

Table 1. The hand-crafted features used for learning resource allocation strategies. Features that were filtered out by RELIEF are marked by "–" in the right column.

the domain of Fives (an extension of Tic–Tac–Toe to a 10×10 board, with the purpose of creating a sequence of five consecutive symbols). We have found experimentally that the range of possible values for that domain was $[0.57, 0.59]$. Once we have found how small this range is, it became clear that the domain was not suitable for experimenting with resource allocation strategies.

7. CONCLUSIONS

This paper presents the first stage of a research that studies resource allocation strategies for game playing. We defined allocation strategy and developed a framework for classifying strategies according to the amount of knowledge they can use and according to their computational requirements.

We developed a methodology for automatically acquiring semi-dynamic strategies. The methodology generates examples (boards) and assigns them classes by measuring the utility of investing additional resources. Therefore, the strategy learning task is reduced to a classification problem. We developed a method for building and using context-sensitive classifiers. When many resources remain, the classifier recommends the investment of extra resource

	white wins	black wins	ties	probability
never	2109	2139	5752	0.499 ± 0.01
random	2518	1580	5902	0.547 ± 0.01
Basic ID3	2906	1813	5281	0.555 ± 0.01
ID3+RELIEF	2942	1311	5747	0.582 ± 0.01
Context-sensitive classifier	3053	1314	5633	0.587 ± 0.01
DC + self-generated features	3225	1242	5533	0.599 ± 0.01
oracle	4305	648	5047	0.683 ± 0.009
always	4551	557	4892	0.700 ± 0.009

Table 2. Summary of results. Confidence level is 0.95.

with higher probability. Finally, we developed a method for automatic generation of features based on patterns of sub-boards. A best-first search procedure searches the space of patterns using information-gain to evaluate the generated patterns.

The methodology has been implemented in the domain of checkers. We conducted an extensive set of experiments that showed a significant improvement over the random allocation strategy. We found experimentally the maximal improvement possible for that domain under the assumptions that we set, and showed that the improvement that we achieved is one half of the maximal possible improvement.

Is it possible to get better results? If indeed the major obstacle is the limited set of features that we use, we could try using a richer pattern language like the one used by Levinson [1991]. However, such patterns are more expensive to use which makes their use for resource allocation decisions less attractive. Another possible direction for improving the results is the use classification algorithms that can deal with continuous classes. Using such algorithms we could use the more accurate class assignment formula (equation 4).

The reader should note that it is unlikely that a resource allocation strategy will have a performance close to that of the *oracle* strategy. The *oracle* strategy decides whether to invest extra resources by trying both alternatives and comparing the results. This is analogue to a search program that decides what operator to apply by conducting a full-depth lookahead. A useful resource allocation strategy can not use such a method and is therefore bound to use some approximation. Even dynamic strategies can not approach the performance of the *oracle* strategy. To know the consequences of searching deeper the program must carry out the search, but at this point it is too late to decide that the extra search is useless.

In subsection 2.4 we list a set of simplifying assumptions that we have made for this research. Assumption 1, which specifies that the basic search technique will be minimax, is reasonable for most game-playing programs. Assumptions 2–5 could be relaxed by adopting the following realistic model of time allocation:

1. Each player is allocated t minutes per m moves (in chess, for example, each player is allocated 90 minutes for the first 40 moves).
2. The search procedure can get a time limit as a parameter and can select a move within this time limit. It is not easy to modify a depth-first procedure such as minimax to cause it to behave in this way. We used the regular minimax procedure which can take a depth limit as a parameter. However, the time spent by the procedure grows exponentially with the depth resource. A common method for time–sensitive search is iterative deepening

which can take a time limit and can guarantee to return a move selection within the given time limit[2].

3. Selective deepening methods can be used when a time-sensitive search procedure is employed. The time used for them will be added to the time counter.
4. The player can make any time-allocation decision providing that it is not greater than the remaining time.

We can make relatively small changes in our methodology to deal with such a model. The simplest way is to change the set of classes learned from $\{k, k+n\}$ (for depth k and depth $k+n$) to the set $\{t/m - l\epsilon, t/m - (l-1)\epsilon, \ldots, t/m, t/m + \epsilon, \ldots, t/m + q\epsilon\}$ for some l, q and ϵ. The classification problem would then be a $l + q + 1$ class classification task instead of a two-class classification task. The tagging of examples will be carried out using the same method as before, but instead of performing two searches to determine the class of an example, the learner will perform $l + q + 1$ searches.

The semi-dynamic strategy proposed in this paper can be combined with other types of strategies. A common dynamic method that is based on iterative deepening [Korf, 1985] (such as the one described in [Levy and Newborn, 1991]) can call the semi-dynamic strategy to decide the initial search depth. Such a combination can significantly reduce the resources wasted during the first iterations.

The work presented here contributes to two communities; To the game-playing research community it brings a deeper understanding of the problem of resource allocation and presents a domain-independent methodology of acquiring resource allocation strategies; To the machine learning community it demonstrates the difficulties involved in the application of machine learning techniques to real domains. Most of the works in the field assume that they are supplied with a set of classified examples. However, for many problems, such as the problem described here, the main difficulty is to generate examples and assign them the right classes. Once a classifier is constructed its use is also not necessarily trivial. The resource allocation domain is an example where the classification of objects should depend upon the context of their use. This work shows a methodology for building a context-sensitive classifier that behaves in this way.

The methodology described in this work is easily applicable to other games. The only change needed is the design of features that may be relevant to resource allocation decisions. If automatic feature generation is required we need to design appropriate pattern language. The class assignment procedure should work for any game. We can also use similar learning techniques to other domains such as planning. If we have a way to evaluate the state of the system, we can assign a class to a state by comparing the values successor states with and without investing extra resources.

REFERENCES

[Anantharaman *et al.*, 1990] T. Anantharaman, M. Campbell, and F. H. Hsu. Singular extensions : Adding selectivity to brute–force searching. *Artificial Intelligence*, 43:99–109, 1990.

[Beal, 1990] D. F. Beal. A generalised quiescence search algorithm. *Artificial Intelligence*, 43:85–98, 1990.

[Boddy and Dean, 1988] Mark Boddy and Thomas Dean. An analysis of time-dependent planning.

[2]While iterative deepening can take any time limit as input, since it returns the move selection according to the last completed iteration, the *effective* amount of resources it uses grows exponentially just like the simpler minimax procedure with depth limit.

In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 49–54, St. Paul, Minnesota, 1988. Morgan Kaufmann.

[Boddy and Dean, 1989] Mark Boddy and Thomas Dean. Solving time-dependent planning problems. In *Proceedings of The Eleventh International Joint Conference for Artificial Intelligence*, pages 979–984, Detroit, Michigan, 1989.

[Carmel and Markovitch, 1994] David Carmel and Shaul Markovitch. The m* algorithm: Incorporating opponent models into adversary search. TR 9402, The Center for Intelligent Systems, Technion, 1994.

[Horvitz, 1987] E. J. Horvitz. Reasonong abount beliefs and actions under computational resource constraints. In *Proceedings of the Third Workshop on Uncertainty in Artificial Intelligence*, pages 301–324, Seattle, WA, 1987. Morgan Kaufmann.

[Hyatt, 1984] R. M. Hyatt. Using time wisely. *ICCA Journal*, 1:4–9, 1984.

[Kira and Rendell, 1992] K. Kira and L. A. Rendell. A practical approach to feature selection. In *Proceedings of The Ninth International Conference on Machine Learning*, Aberdeen, Scotland, 1992. Morgan Kaufmann.

[Korf, 1985] Richard E. Korf. Depth–first iterative–deepening : An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

[Levinson, 1991] Robert Levinson. Pattern associativity and the retrieval of semantic networks. *Computers and Mathematics*, Special issue on Semantic Netwrok in Artificial Intelligence, 1991.

[Levy and Newborn, 1991] David Levy and Montey Newborn. *How Computers Play Chess*. Computer Science Press, New York, New York, 1991.

[Markovitch and Scott, 1993] Shaul Markovitch and Paul D. Scott. Information filtering: Selection mechanisms in learning systems. *Machhine Learning*, 10:113–151, 1993.

[Nau, 1980] D. S. Nau. Pathology on game trees : a summary of results. In *Proceedings of the First National Conference on Artificial Intelligence*, pages 102–104, Stanford, California, 1980. Pitman.

[Nau, 1982] D. S. Nau. An investigation of the causes of pathology in games. *Artificial Intelligence*, 19:257–278, 1982.

[Pearl, 1983] J. Pearl. On the nature of pathology in game searching. *Artificial Intelligence*, 20:427–453, 1983.

[Quinlan, 1986] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[Samuel, 1959] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal*, 3(3):211–229, 1959.

[Schaffer, 1992] C. Schaffer. Overfitting avoidance as bias. In *Proceedings of The Ninth International Conference on Machine Learning*, Aberdeen, Scotland, 1992. Morgan Kaufmann.

[Scheucher and Kaindel, 1989] Anton Scheucher and Hermann Kaindel. The resaon for the benefits of minimax search. In *Proceedings of The Eleventh International Joint Conference for Artificial Intelligence*, pages 322–327, Detroit, Michigan, 1989.

List of Figures