Learning of Resource Allocation Strategies for Game Playing

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

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 did not receive 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 proceed to describe a method for learning semi-dynamic strategies from self generated examples. The method assigns 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.

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 deserves more resources is not always easy.

Substantial research efforts were allocated for devising sophisticated mechanisms for selecting the right move (minimax, alpha-beta, windowing). However, there were only few attempts to study the problem of resource allocation in game playing throughly [Levy and Newborn, 1991; Hyatt, 1984].

The research described here has the following goals:

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

The rest 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 experiments conducted using the implementation and their results. Section 6 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 for 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 between 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 2-player perfect information games (e.g., Chess, Checkers). 2-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 much resource the program should spend on the calculation of each move. A minimax procedure that is allocated more resources for 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 much of the resource 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 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 resulted sequence can be used for all games. However, it is very unlikely that a single sequence fits all possible games. If the strategy is given a model of the opponent 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 in order 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* a/., 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 in order 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 to 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 procedure 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 K's which are *True* must be < *d.* Under this model, the total number of allocation sequences that a strategy can produce is $\sum_{i=0}^{d} \binom{m}{i}$.

3 Learning Semi-dynamic Strategies

The knowledge that a strategy uses in order 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 getting domain specific knowledge and incorporating it into a strategy is not an easy task. Even human expert players find it hard to formulate explicitly the reasoning behind their resource allocation decisions.

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 exactly will the training examples be generated. Four questions arise:

1. Where will the training examples come from ?
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 should 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

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 once with, and once without extra resource, 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 resource. Let 6 be a board, movek (6) 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 Vk(m) the minimax value assigned to move m by a depth *k* search. The class assigned to 6 will be :

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

We measure the utility as 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* (6) = movejb+n(6) 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 Ar, we take both values from the former. Note also that this measure is unlikely to be discrete, so incorporating it into the system as the class, requires a classification algorithm that can process continuous classes.

More generally still, it is possible that upon searching to some depth, there will be more than one best move. We shall change the notation to *movesk(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:

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

The board is positive only if depth *k* + *n* search can be beneficial or, in other words, there is a chance that a depth ib search will choose one of the wrong moves. Note that using $moves_k(6)$ = $movesk+_n\{b)$ is not good enough because it includes the case $mores_k$ (b) C $movesk+_n(b)$. In the latter, although depth k search can choose from a smaller set, this set does not include any wrong moves, yielding that any move chosen by 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 $moves_{k+n}$ (6), all its members have the same minimax value (in terms of depth *k* -h n search) by definition. Therefore, the value of the expected outcome of a depth ib -I- n search, can be taken as the value of any member *ml E* mot;e$fc+$_n$(&). For the set moves* (6), although all its members have identical V* values, they might have different V_{k+n} values. Therefore, the class assigned to 6 will be:

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

Note that calculating $moves_{k+n}(6)$ 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, so 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., center control),

and 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 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 resource, the program need only to pass the current board through the decision tree, and invest the extra resource 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 a large resource is 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 resource is 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 in order to make the allocation strategy behave in the way described above. We divide the training set into two portions. The first portion is used to build a decision tree in the traditional way. The tree is then fixed and the second portion of *T* examples is used in order to assign probabilities to the leaves. If the number of examples that reach a leaf / is 7} out of which *P1* are positive, then the leaf is assigned the probability:

$$PP_l = \begin{cases} \frac{P_l - 0.5}{T_l} & T_l \neq 0 \\ 0.0 & T_l = 0 \wedge l \text{ was negative} \\ \frac{P}{T} & T_l = 0 \wedge l \text{ was positive} \end{cases}$$

Assume that during a game, $\overline{m}$ moves are left to play with *d* deep searches still available. We would like the resource allocation strategy to decide on making deep search with probability $\frac{\bar{d}}{\overline{m}}$. We order the set of leaves according to their probabilities and mark the longest prefix of the ordered set (with highest probabilities) that satisfies

$$\frac{\sum T_l}{T} < \frac{\bar{d}}{\overline{m}}$$

When the resource allocation procedure is called, it passes the current board through the decision tree. Only if the leaf that it reaches is marked, the procedure will allocate extra resource.

After the move is performed, the values of $\bar{d}$ and 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 *d* and m. Since gathering the leaves with highest probability before every move is too expensive, we perform a preprocessing stage after acquiring the decision tree. A table is constructed that specifies for a range of possible values of $\frac{\bar{d}}{\overline{m}}$ what is the positiveness threshold.

4 Implementation

The above methodology was implemented in a system whose architecture is illustrated in Figure 1. The learning system generates boards by having two instances of the player play 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 were set as follows: basic search depth (k) = 4, additional search depth (n) = 2, maximum number of deep searches per game *(d)* = 8, maximum number of paired moves per game (m) = 40.

A simple evaluation function was used on the leaves of the game-tree which relied only on piece advantage.