Akihiro Kishimoto
02B-9007, 112 St., Nw.
Edmonton, Alberta, Canada T6G 2C5

Date: ____________

University of Alberta

Transposition Table Driven Scheduling for Two-Player Games

by

Akihiro Kishimoto

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Spring 2002

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Transposition Table Driven Scheduling for Two-Player Games** submitted by Akihiro Kishimoto in partial fulfillment of the requirements for the degree of **Master of Science**.

Jonathan Schaeffer
Supervisor

Paul Lu

Gordon Swaters
External Examiner

Date:

Abstract

Game-tree search is an important research topic in Artificial Intelligence. Because it is computationally intensive, researchers have turned their attention to parallel game-tree search algorithms in order to improve running time. However, achieving high parallel performance remains a difficult task on distributed-memory systems. Many high-performance systems for two-player games use the $\alpha\beta$ search algorithm, enhanced with transposition tables. Transposition tables store useful information about game-trees from previous searches. When parallelizing the $\alpha\beta$ algorithm, a major problem is sharing transposition table information efficiently among the processors. A processor must communicate in order to look up table entries on other processors, which hurts the performance of the parallel search.

TDS (Transposition-table Driven Scheduling) was recently proposed to overcome this difficulty. TDS was implemented for single-agent search and achieved remarkable results. It is an open question as to whether applying this idea to $\alpha\beta$ can yield good performance, because of the greater complexity of the $\alpha\beta$ algorithm.

This thesis explains why it has been considered hard to combine TDS with $\alpha\beta$. We then develop TDSAB, a parallel algorithm that contains important new techniques which enable TDS to be used with $\alpha\beta$. TDSAB has been tested in two games which have different search characteristics - Awari and Amazons. The performance of TDSAB is evaluated on a network of workstations. TDSAB achieved satisfactory speedups for both games.

Acknowledgements

Without the assistance of many people, I could not have completed this thesis.

First and foremost, I would like to thank my supervisor, Jonathan Schaeffer. He permitted me to study this research topic, carefully read drafts of this thesis, brainstormed new ideas with me at weekly meetings, and gave me much valuable feedback.

Both Martin Müller and Paul Lu permitted me to study parallel search as course projects. As a professor of the GAMES group, Martin gave me a lot of valuable comments from the very beginning of this research. Paul was available to discuss new ideas with Jonathan and me.

I would like to extend my appreciation to Jack van Rijswijck, Theodore Tegos, and Yasushi Tanase. Jack gave me his Awari program and kindly answered questions about the program. Theo generously provided the evaluation function of his Amazons program. Tanase set up his program to extract test positions for my Amazons experiments.

Thomas Lincke of ETH Zürich shared some specific techniques in his Awari program, when he visited the University of Alberta. His Awari endgame databases were also useful for me.

I would like to gratefully acknowledge the people in the Computing and Network Services at the University of Alberta for providing access to their network of PCs.

I have great respect for Yasushi Tanase, Norifumi Gotoh, and Ayumu Nagai, the current and former members of the IS Shogi Project. They are the friends who convinced me to switch my research topic to game-tree search when I was a graduate student at the University of Tokyo in Japan.

Darse Billings, Aaron Davidson, Maria Cutumisu, Anita Petrinjak and other friends often invited me to see movies and to drink some beer, in order to let me escape from my work and refresh myself when I felt tired. Yonghe Niu's funny stories made me laugh when we were working late at night in the AI lab. Emails from my friends in Japan cheered me up whenever I felt lonely. Thanks!

Anne Nield corrected a lot of grammatical errors in this thesis as a proof reader. I appreciate it very much.

Finally, I would like to thank Yngvi Björnsson, Darse Billings, Aaron Davidson, Adi Botea, and the other members of the GAMES group for their beneficial discussions and constructive advice.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Artificial Intelligence and Games

Search plays an essential role in solving many problems in computer science, and search algorithms have many practical applications. For example, database systems, theorem-provers, and game-playing systems have search engines at the core of the application. Games have been regarded as useful test beds for search algorithms research in Artificial Intelligence (AI), for many reasons. First, games provide researchers with strong and clear motivations such as defeating the best human players. Second, because games have simple rules and clear results (wins, draws, or losses), it is easier for researchers to measure improvements than in many real-world applications. Finally, games are intellectually challenging and the skills required to play a game well are skills we want to *teach* a computer.

For the last 40 years, researchers have invested significant resources to improve the quality of play of computers for games such as Othello, checkers, and chess. The results have been impressive, although some games such as Shogi (Japanese chess) and Go still have considerable room for improvement. In Othello, Logistello defeated the human world champion in 1997 [12]. In checkers, Chinook was the first computer program to win a World Man-Machine Championship in 1994 [47]. In chess, Deep Blue [14] is the culmination games research, providing features such as a special hardware chip for chess, massive parallelism, and enhanced search algorithms. Deep Blue defeated Garry Kasparov, the World Chess Champion, in 1997 [49].

Because chess is a popular game in the western world, it is sometimes referred to as one of the *Drosophila* of AI [30]. The Drosophila, which is the scientific name for the common fruit fly, is popularly used for biological experiments. Similarly, chess is used by many researchers to perform experiments in AI. The techniques pioneered in computer chess have been widely applied to other games, such as Othello and checkers. The $\alpha\beta$ search [11, 23] is the most successful algorithm in computer chess. In fact, most of the research in chess has focused on searching chess trees more efficiently, a tool which may be applicable directly to other games.

The strength of a game-playing system is strongly influenced by the depth of the search tree explored [52]. To increase the probability of winning, game-playing systems strive to out-search the opponent. Therefore, researchers have attempted to develop algorithms which search game-trees deeper, subject to time constraints such as those used in tournaments.

1.2 Parallel Systems and Games

Although single processor machines are becoming faster every year, researchers are always looking for increased speed, since that will improve the quality of their game-playing systems. Faster machines provide deeper searches and a better quality of play - which can be taken one step further by using a large number of machines in parallel. Parallel game-playing systems use multiple processors to cooperate in computing the game-tree. One of the key factors in Deep Blue's victory was the use of a massively parallel system.

From the viewpoint of hardware technologies, there are two main kinds of parallel systems. A *shared-memory* system has a global shared address space: each processor can access whichever location it chooses to, in memory. Therefore, when a processor wants to access data created by other processors, it can do so as easily as if it were its own data. On a *distributed-memory* system, each processor has a local address space for its local memory, which cannot be directly accessed by other processors. A processor exchanges messages to obtain information from other processors. For example, when a processor

P_1 wants some information held by another processor P_2: (1) P_1 sends a message to P_2 to request the information; (2) P_2 receives the message from P_1; (3) P_2 sends the information to P_1; and (4) P_1 receives the information. Clearly, communication between processors using a shared-memory is simpler and faster than a distributed-memory system.

There are trade-offs between shared-memory and distributed-memory systems and both types of systems are widely used. From the programmer's point of view, shared-memory systems are attractive for applications which need to frequently share data, because this is more easily done on shared-memory systems than on distributed-memory systems. However, distributed-memory systems are becoming an important architecture. With the increasing importance of the Internet, a large number of computers connected through a network (e.g., NOWs (Network Of Workstations) [3]) are becoming increasingly attractive for parallel computing. In this environment, a computer cannot access information in other computers without incurring communication. Using a network of workstations has two advantages: (a) compared to the expensive shared-memory supercomputers, high-performance workstations are available at low cost; and (b) the system is able to easily incorporate the latest technology as soon as it is available. There has been a strong demand, therefore, to efficiently implement parallel applications in distributed environments.

State-of-the-art game-playing systems most often use the enhanced $\alpha\beta$ algorithm with iterative deepening [50] and transposition tables (basically constructed as hash tables). Iterative deepening performs a shallower search before doing a deeper search in order to have better move ordering information which reduces the search effort. Combined with iterative deepening, transposition tables store useful information on the result of subtrees searched, thereby eliminating duplicate effort. When developing parallel game-playing systems on distributed-memory systems, the effective sharing of transposition tables among processors becomes a serious problem. The fact that the tables cannot be shared without exchanging messages degrades the performance of the parallel search algorithm. In a distributed-memory system, if a processor does not have local table information concerning a position, there are two natural ways to deal with the situation. One is to recalculate the information locally, even though it might be available on another processor. However, this approach incurs an extra computation which a sequential game-playing system does not. The other method is to request another processor, which might have the information on the position, to send it back. Although this approach does not lose the information on the position previously computed, it has a communication latency whenever a processor has no information on the position.

Recently TDS (Transposition-table Driven Scheduling) was proposed as a general scheme for implementing transposition tables efficiently on distributed-memory systems [43]. Many parallel systems keep the work local, and send out requests for data. TDS keeps the data local, and moves the work to the data. The idea of TDS is to always move a position to the processor which has information on the work in its transposition table.

TDS was used to parallelize IDA*, a single-agent search algorithm. Remarkable results were achieved using workstations connected with a fast network.

The $\alpha\beta$ algorithm has a more complicated framework than IDA*. Therefore, in spite of the success of TDS in the single-agent search algorithm, it is considered to be hard to parallelize the $\alpha\beta$ search algorithm using TDS ideas.

1.3 Contribution of this Thesis

This thesis investigates an efficient implementation of parallel $\alpha\beta$ search algorithms on distributed-memory systems. Our contribution is summarized as follows:

- We propose a new parallel $\alpha\beta$ search algorithm, TDSAB (Transposition-table Driven Scheduling Alpha-Beta). TDSAB is an application of TDS to the $\alpha\beta$ search algorithm. The solution addresses the problem of effectively sharing transposition tables on distributed-memory systems.

- Experiments for TDSAB are measured in the games of *Awari* and *Amazons*, two games which have completely different characteristics. Awari programs build narrow game-trees, which makes it difficult to create sufficient work to search for the parallel algorithm. On the other hand, the game-trees of Amazons are bushy, which may create too much work in parallel search algorithms. The speedups observed in our experiments are roughly 23 in Awari and Amazons, on a network of workstations with 64 processors. These speedups are comparable to many reported speedups for other games.

1.4 Organization

Chapter 2 introduces game-tree search algorithms used on a single processor. This chapter is a survey for those who are not familiar with this topic. Chapter 3 surveys the major parallel $\alpha\beta$ search algorithms and addresses some performance issues of the previous work on distributed-memory systems. TDS is introduced in Chapter 3. Chapter 4 gives the details of TDSAB, which uses TDS, and addresses the issues explained in Chapter 3. Chapter 5 presents and analyzes experimental results on TDSAB in Awari and Amazons. Chapter 6 concludes the thesis and indicates further research directions.

Chapter 2

Sequential Game-Tree Search

In this chapter, we describe the basics of sequential game-tree search for two-player games with perfect information. Section 2.1 is an introduction to the concepts of game-trees and minimax search in order to explain how game-playing programs choose a next move. Section 2.2 deals with the $\alpha\beta$ algorithm, which is considerably more powerful than minimax search, while ensuring that the same value as the minimax algorithm is computed. Section 2.3 presents performance enhancements to $\alpha\beta$. In Section 2.4, we introduce MTD(f), which is the state-of-the art $\alpha\beta$ algorithm for two-player games. In Section 2.5, we conclude this chapter.

2.1 Minimax Search

In two-player games, each player in turn chooses one of the legal moves available to them. A player selects the move that *maximizes* their advantage, while the opponent tries to *minimize* the advantage of the other player. The largest gain for the player is to win, while the largest loss is to lose.

In order to build a game-playing program, we need a framework to analyze the possible combination of moves by both players. Therefore, the program builds a *game-tree* to represent positions and moves. A *node* represents a position in a game, and a *branch* corresponds to a move. The *root* node of the tree represents the current position of the game. If the next player to move has no legal move at a node, then the value of that node is determined by the rules of the game. This node is called a *terminal* node. If the program can look ahead all the way from the root to the terminal nodes, it can determine whether the root is a win, lose, or draw. In other words, the ultimate purpose of game-playing programs is to find the best strategy (maximum score) from the root to the terminal node, by considering all the possible move sequences from the root.

However, it is not feasible to generate the whole game-tree in general because the tree size becomes too big. For example, checkers is 10^{20}, and chess has been estimated at 10^{40}. Therefore, we have to stop generating the tree at a certain point for a practical game-playing program that has to make a move in a set amount of time; all we are able to do is to search the game-tree as deep as possible within the restricted time. To represent to what depth the game-tree is explored, the term *ply* was first introduced by Samuel [44]. The ply represents a distance from the root, and is the same notion as the depth of a game-tree. A d-ply search means that the program searches the positions d moves (one play by one player) ahead from the root.

Because the search cannot usually reach a terminal node, we have to approximate the value for a node that the program explored, using a heuristic *evaluation function*. The evaluation function estimates the chance of winning the game. Higher positive numbers represent a more advantageous position for the player, while lower negative numbers represent a more disadvantageous position. Winning positions are usually evaluated to ∞, and losing positions are set to $-\infty$. These evaluation values are negated from the viewpoint of the opponent, because a win (loss) for the player is a loss (win) for the opponent.

The *minimax* framework identifies the best move among the legal moves at a node. In the minimax framework, we have Max nodes and Min nodes. The player at each Max (Min) node is called the Max (Min) player. The children of each Max node are always Min nodes, and the children of each Min node are always Max nodes, because each player makes a move in turn. A node which the program does not expand further is called a *leaf* node. The values assigned to the leaf nodes are from the viewpoint of the Max player. Higher values are advantageous to the Max player, while lower values are advantageous to the Min player. The *minimax value* at each node is naturally calculated from the

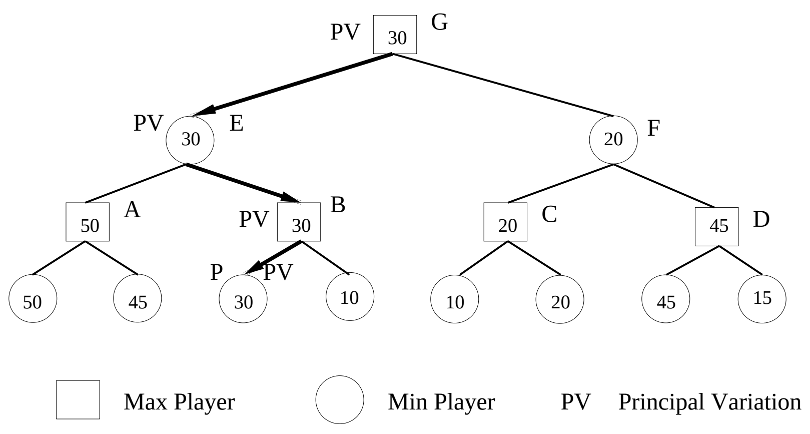


Figure 2.1: Minimax Tree

leaf nodes in a bottom-up manner. Each Max node maximizes the score of the children (Min nodes), while each Min nodes minimizes the values from the child Max nodes. Intuitively, this procedure assumes that both players always try to play the best-scoring move at each position.

Figure 2.1 illustrates the minimax procedure. By convention, the root node is a Max node. The numbers in the leaf nodes are evaluation values, which are computed from the Max player's viewpoint. The scores at the leaf nodes are passed back to their parents. For example, the value for A is equal to 50 because it takes the maximal value of its children, while the value of 30 is passed back to the Min node E after calculating the values for A and B. The root G has a score of 30, the maximum value of its children E and F. At the root G, the best score is reported back by way of P, B, and E through the maximizing/minimizing procedure. This path is called a *principal variation*, which is the best-scoring move sequence for both players. The nodes on this path are called *PV (Principal Variation) nodes*.

Figure 2.2 presents pseudo code for the minimax algorithm. For the sake of simplicity, we present it as the *negamax* form, which is equivalent to the maximizing/minimizing framework. Instead of taking the minimal score for each Min node, the negamax form maximizes the scores by negating the returned values from the children. The scores at the leaf nodes are calculated from the player's own viewpoint in the negamax form. Therefore, all the val-

```
int Minimax(node_t n, int d) {
   int i,  score = $-\infty$;
   if (d == 0 ||  n == terminal) return Evaluate(n);
   for(i = 0; i < n.num_of_children; i++) {
      g = -Minimax(n.child_node[i], d-1)
      score = max(score, g);
   }
   return score;
}
```

Figure 2.2: Negamax form of the Minimax Algorithm

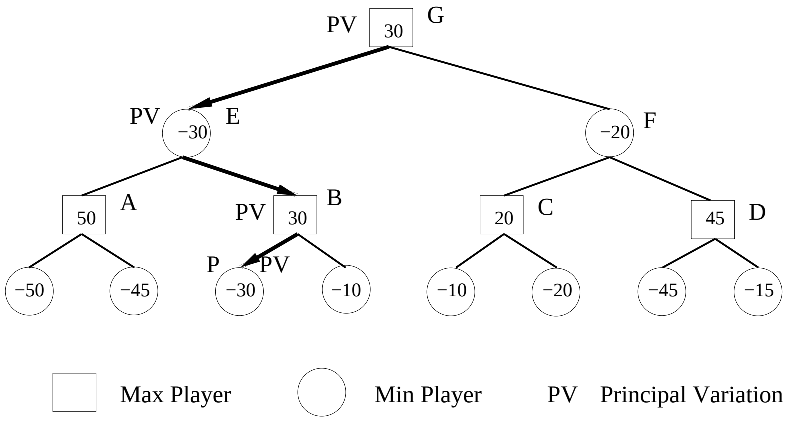


Figure 2.3: Minimax Tree (Negamax Form)

ues assigned in Figure 2.1 are negated in order to compute the same minimax value when using the case in Figure 2.2. (See Figure 2.3, which illustrates the same minimax tree as Figure 2.2). The code has a node n and depth d as a parameter and returns the minimax value for n. At each Max (Min) node, the algorithm looks at the minimax values of all the children of n, and returns the maximal (minimal) value among them to its parent. The minimax algorithm can be implemented as a depth-first search, which requires memory only linear to the search depth.

A d-ply minimax search algorithm visits all the leaf nodes at depth d to determine the minimax value of the search. When searching a d-ply uniform tree with an average of b children at each node, the number of bottom positions

(NBP) (i.e., leaf nodes) visited by the minimax algorithm is:

$$\text{NBP}_{Minimax} = b^d$$

The search grows exponentially as a function of the depth d. This influences the strength of game-playing programs, because deeper search achieves better quality of game play [52].

Fortunately, this work can be reduced because it is not always necessary to visit all the leaf nodes at depth d to determine the minimax value for the root. It can be proved that some nodes cannot affect the value of the root. Knuth and Moore showed the least number of leaf nodes that must be visited to prove the root value is [23],

$$\text{NBP}_{Best} = b^{\lceil \frac{d}{2} \rceil} + b^{\lfloor \frac{d}{2} \rfloor} - 1$$

Although NBP_{Best} is still exponential in d, it is much smaller than $\text{NBP}_{Minimax}$. The best-case minimax algorithm can search roughly twice as deep as the minimax algorithm. Practical algorithms try to achieve the best-case result. The rest of this chapter focuses on reducing the number of leaf nodes visited.

2.2 The $\alpha\beta$ Algorithm

In a minimax tree, it is not necessary to visit every node to compute the correct value at the root. For example, $\max(10, \min(5, \top))$ is always equal to 10, because $\min(5, \cdots)$ is equal to or less than 5. Therefore, from the viewpoint of game-trees, the subtree which corresponds to $\top$ can be pruned (*cut-off*). This is the basic idea of $\alpha\beta$ search. Brudno was the first to publish a paper showing that pruning was possible in minimax search [11], a result which was also independently discovered by other researchers. The $\alpha\beta$ algorithm is a special case of branch-and-bound algorithms. While many branch-and-bound algorithms have only one parameter to bound conditions, $\alpha\beta$ has two bounding parameters (α and β), achieving more cut-offs [23].

Figure 2.4 shows pseudo code for the negamax form of the $\alpha\beta$ algorithm. Unlike the minimax algorithm shown in Figure 2.2, it has a *search window*

```
int AlphaBeta(node_t n, int d, int $\alpha$, int $\beta$) {
  int score = $-\infty$;
  if (d == 0  ||  n == terminal) return Evaluate(n);
  for (i = 0; i < n.num_of_children; i++) {
    score = max(score, -AlphaBeta(n.child_node[i], d-1, $-\beta$, $-\alpha$);
    $\alpha$ = max($\alpha$, score); /* Adjust the search window */
    if ($\alpha \geq \beta$) return $\alpha$; /* Cut-off */
  }
  return score;
}
```

Figure 2.4: Pseudo Code for the $\alpha\beta$ Algorithm (Negamax Form)

(α, β) to detect pruning conditions. α represents a lower bound on the range of relevant values for the player to move, while β indicates an upper bound. Values outside the search window cannot affect the minimax value for the root.

$\alpha\beta$ search starts searching the root node with an initial window $(-\infty, \infty)$, and it traverses a game-tree recursively in a depth-first manner until it reaches a leaf node. The leaf node is evaluated and the value is passed back to its parent node.

Because we use the negamax form, α and β are negated and exchanged using $-\beta$ $(-\alpha)$ as a lower (upper) bound. At each node, if the returned *score* of a child is greater than α, $\alpha\beta$ updates its lower bound α, because the minimax value is at least equal to *score*. Therefore, at a node as $\alpha\beta$ explores more children, α is monotonically increasing, while β is monotonically decreasing.

When $\alpha \geq \beta$, a proof that the value lies outside the range of relevant values has been achieved. Hence, further work at this node is unnecessary. $\alpha\beta$ returns immediately the lower bound to its parent.

Figure 2.5 shows an example of how the $\alpha\beta$ algorithm works, as it returns the same minimax value as in Figure 2.3. Let us assume that $\alpha\beta$ searches in a left-to-right order. $\alpha\beta$ passed down the initial window $(-\infty, \infty)$ at G to search A via E in a depth-first manner. After evaluating the left child of A, the right child of A is searched with the window $(-\infty, -50)$, because the value for E is at least 50. α at the root node is updated after completing the search of the left child node E. Therefore, the right sibling F is searched

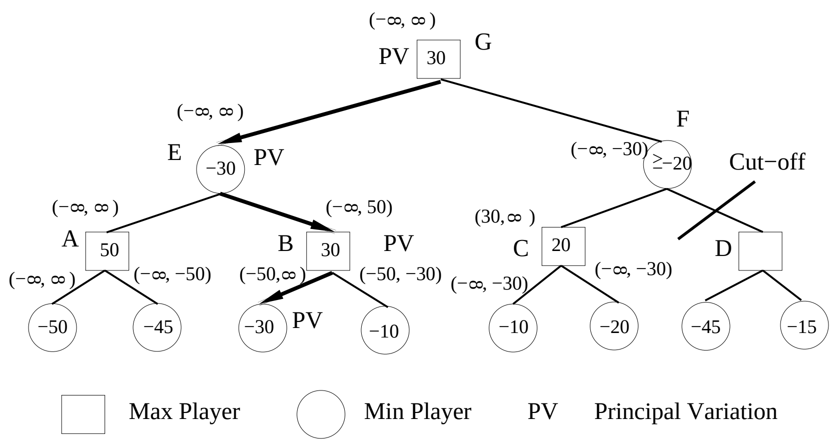


Figure 2.5: Example of $\alpha\beta$ Pruning

with the search window $(-\infty, -30)$. When $\alpha\beta$ backs up -20 to F, the α is adjusted to -20. However, because β is -30, no minimax value can satisfy the search window (α, β). If we do not use the negamax form, this means that $\max(30, \min(20, \top))$ is assured to be equal to or more than 30. Thus, a cut-off occurs at E and $\alpha\beta$ never visits the subtrees of D.

2.3 Enhancements to $\alpha\beta$ Search

$\alpha\beta$ has a best case that is experimentally better than minimax. In this section, we discuss enhancements to $\alpha\beta$ to ensure that the performance is close to the best case.

2.3.1 Move Ordering

Although the $\alpha\beta$ algorithm is efficient compared to the minimax algorithm, its efficiency depends on the order in which nodes are searched. For example, Figure 2.6 shows the best-ordered tree (called *minimal tree*) in which more nodes are pruned than in Figure 2.5. Intuitively, if there are N children, the cut-offs should be obtained with the first child; if this is not the case, then an additional search is done.

Knuth and Moore classified nodes into 3 groups in the best-ordered $\alpha\beta$ search to calculate NBP_{Best} [23] (See Figure 2.7). Type 1 nodes are also called

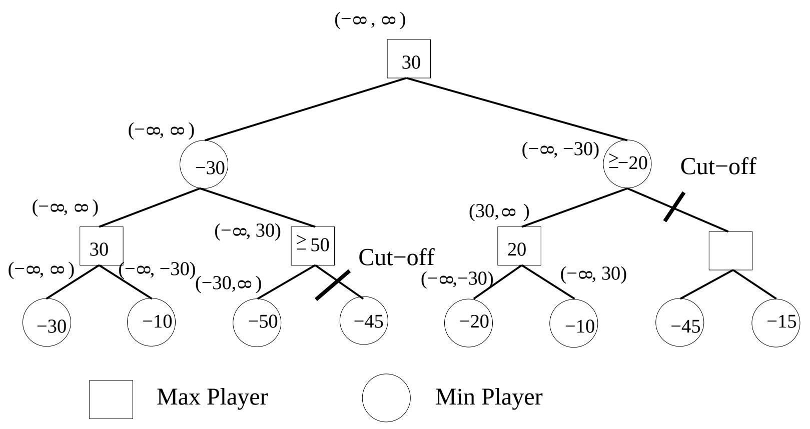


Figure 2.6: Best-Ordered Game-Tree (Minimal Tree)

principal variations (or PV nodes). PV nodes are located in the left-most branch of the tree and they are evaluated first. Cut-offs can occur at type 2 nodes, while all children of the other two types (i.e., type 1 and type 3) are always expanded. Therefore, type 2 nodes are sometimes called CUT nodes, and type 3 nodes are called ALL nodes. In the minimal tree, the $\alpha\beta$ algorithm prunes the rest of the siblings if the left-most child has the best value because, at CUT nodes, cut-offs occur immediately after searching the left-most node, and at PV and ALL nodes $\alpha\beta$ always searches the rest of the siblings.

The $\alpha\beta$ algorithm cannot define the best child at a node in advance. Therefore, before the searching effort is expended, it identifies the likely best child using heuristics. The *killer heuristic* [50] keeps track of moves (*killers*) at each depth which most frequently caused a cut-off. The *history heuristic* [46] is a generalization of the killer heuristic. The history heuristic maps moves to entries in the history table, which contains the weights of the moves. When a cut-off happens at a node with a d-ply search, a weight of 2^d is added to the table entry for the best move. The moves are ordered using their history table scores.

The other techniques widely used are *iterative deepening* and *transposition tables*, which we will explain in the next section.

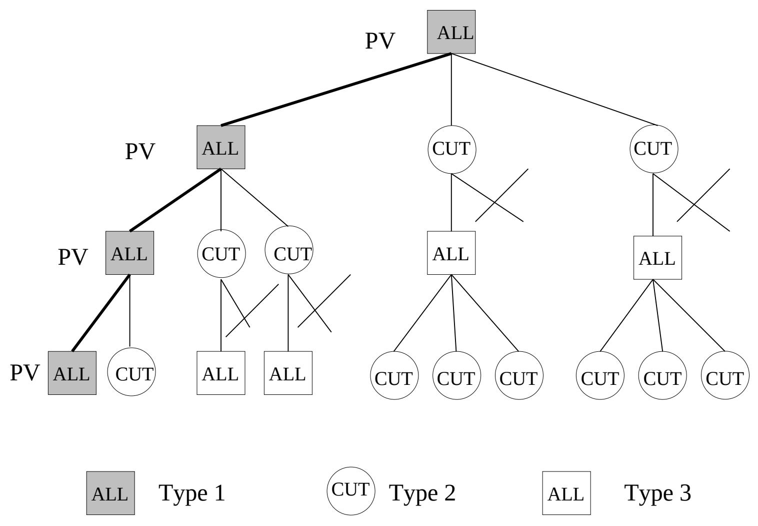


Figure 2.7: Classifications of Nodes in Best-Ordered Trees

2.3.2 Iterative Deepening and Transposition Tables

Early game-playing programs had trouble setting the maximum search depth d, because the computational complexity grows exponentially with d. Slate and Atkin first tried the idea of *iterative deepening* as a better time control scheme, in CHESS 4.5 [50]. Before searching to a large depth d, iterative deepening carries out a series of 1-ply, 2-ply, 3-ply,$\cdots$,$(d-1)$-ply searches. If a new iteration takes too long to return the minimax value, the program aborts searching and returns the best move from the previous iteration.

Although iterative deepening seems inefficient because of the extra cost of expanding interior nodes, it is usually more efficient than those methods using a direct d-ply search, because iterative deepening can improve the move ordering. The best path from the previous iteration is searched first in the new iteration. In effect, the previous iteration provides good move ordering information for the next iteration. The cost of the $d-1$ iterations is relatively small compared to the benefits of having a high probability of searching the best move first with a larger depth d.

In practice, search spaces for many games are not trees, but graphs. A node may be reached by more than one path. The result of a search can be stored in a *transposition table* [19, 50] so that, if the same position recurred in the search, the previously computed value can be reused. In the transposition table, each entry has room for the best score, the depth, and whether the score is exact, an upper bound, or a lower bound. After searching a node, the information on the node is stored in its transposition table entry. The advantages of the transposition table are as follows:

(1) The transposition table can omit searching subtrees that were previously expanded, by checking the table before expanding a node. If a node is found in the transposition table and it was searched to at least the desired depth, the score for the node may be returned without searching the subtrees.

(2) The transposition table is especially useful for iterative deepening. Iterative deepening causes the interior nodes to be repeatedly visited. Even if the transposition table information on the value for a node is not useful, the move that it suggests can improve move ordering. The best move from a previous iteration is searched first with high confidence of being the best move for the new iteration.

Transposition tables are usually constructed as hash tables in order to minimize the response time. The most popular hash function is Zobrist's function, which needs only some XOR operations of random values [54]. If all the transposition entries are full, some information is lost due to replacing an existing entry or discarding new search results. Therefore, the size of the transposition table should be as large as possible.

2.3.3 Aspiration Windows and Principal Variation Search

Let v be the value AlphaBeta(n,d,α,β) returns (see Figure 2.4) and $F(n)$ the minimax value for n. Knuth and Moore showed the following properties of the $\alpha\beta$ algorithm [23]:

(1) **Fail low:** $v \leq \alpha \Longrightarrow v \leq F(n)$.

(2) **Exact value:** $\alpha < v < \beta \Longrightarrow v = F(n)$.

```
int PVS(node_t n, int d, int $\alpha$, int $\beta$) {
   int score;
   if (d == 0 ||  n == terminal) return Evaluate(n);
   /* Full window search */
   score = -PVS(n.child_node[0], d-1, $-\beta$, $-\alpha$);
   for (i = 1; i < n.num_of_children; i++) {
      if (score $\geq \beta$) return score;
      $\alpha$ = max($\alpha$, score);
      /* Null window search */
      value = -PVS(n.child_node[i], d-1, $-\alpha - 1$, $-\alpha$);
      if(value > $\alpha$) {
         if (value < $\beta$)
            /* If fails high, re-search */
            score = -PVS(n.child_node[i], d-1, $-\beta$, $-\alpha$);
         else score = value;
      }
   }
   return $\alpha$;
}
```

Figure 2.8: Pseudo Code for PVS Algorithm (Negamax Form)

(3) **Fail high:** $v \geq \beta \Longrightarrow v \geq F(n)$.

Although the basic $\alpha\beta$ starts with the window of $(-\infty, \infty)$ at the root node, we can still calculate the accurate minimax value v while searching fewer nodes, if we can narrow the search window (called *aspiration window*) $-\infty < \alpha \leq v < \beta < \infty$. If we use a narrower window, we will achieve more cut-offs.

A reasonable estimation for the aspiration window can be derived from iterative deepening. If the minimax value from a $(d-1)$-ply search is v, then we set the aspiration window to be $(v - \epsilon, v + \epsilon)$ for the d-ply search. If the returned value v' is $v - \epsilon < v' < v + \epsilon$, then v' is the precise minimax value for the root node. The fail low value, $v' \leq v - \epsilon$, requires a re-search with $(-\infty, v')$, while the fail high, $v' \geq v + \epsilon$, means the minimax value at the root node is in (v', ∞). Although a re-search in the case of a fail low or fail high may cost more work than that with the initial window $(-\infty, \infty)$, in general with a good evaluation function, aspiration search is usually a winner.

If a window is set to $(\alpha, \alpha+1)$ instead of (α, β), it is called a *null window*. A null window can be used to determine whether or not the score for a node is larger than α. For example, let α be the score for the left-most node. Consider searching the next child with the $(\alpha, \alpha+1)$ window. If the value returned is smaller than or equal to α, then the move is no better than the left-most move and no further work is necessary. A value bigger than α shows that the move searched with the null window is better than the left-most node, and the move must be re-searched with a full window to get its exact value.

Following the theoretical study of Pearl's Scout algorithm [35], Principal Variation Search (PVS) [27] and NegaScout [39] were proposed independently. PVS and NegaScout are improved $\alpha\beta$ algorithms using null windows, and both are widely used search algorithms in game-playing programs. For PVS to work efficiently, moves are required to be strongly ordered by using the techniques described above. Figure 2.8 shows the pseudo code for PVS. PVS first searches the left-most node with a full window. Unlike the basic $\alpha\beta$, after updating a lower bound for the node, it searches the rest of the siblings with a null window $(\alpha, \alpha+1)$, which costs less than a full window (α, β). Intuitively, we see that all the algorithm has to do is to prove the remaining moves are no better than the first move, if the best move is searched first. The only cases in which to re-search the siblings are those when the returned score v holds $\alpha < v < \beta$, because we do not know the exact values of the siblings.

2.4 MTD(f)

As we described in Section 2.3.3, the key idea of PVS was the null window search. A further improvement of the $\alpha\beta$ algorithm can be made by *always* searching with null windows. MTD(f) is one of the variants of such algorithms [38]. MTD(f) itself was originally inspired by the research to reformulate Stockman's best-first search SSS* [51] as a depth-first search MT-SSS* [37].

Figure 2.9 shows the pseudo code for MTD(f). In MTD(f), when searching the root node to depth d, the initial window at the root node is set to a null window $(f-1, f)$, where f is the minimax score for the root node obtained

```
int MTD(node_t n, int d, int f) {
   int score;
   /* Set the initial lower and upper bounds */
   lowerbound = $-\infty$; upperbound = $\infty$;
   if (f == $\infty$) bound = f + 1;
   else bound = f;
   do {
      /* Null window search */
      score = AlphaBeta(n, d, bound-1, bound);
      if (score < bound) upperbound = score;
      else lowerbound = score;
      /* Re-set the bound */
      if (lowerbound == score) bound = score + 1;
      else bound = score;
   } while (lowerbound $\neq$ upperbound);
}
```

Figure 2.9: MTD(f)

with a $d-1$ ply search. Let *score* be a result for a node n with a d-ply search with a null window $(bound-1, bound)$. Each null window search proves whether *score* is less than *bound* or not; if $score < bound$, then the minimax score for n is less than *score*; otherwise the minimax score is greater than or equal to *score*. In other words, a null-window search can determine an upper or lower bound of the minimax score for n. MTD(f) continues performing null window searches, whether a score for n is until the lower bound agrees with the upper bound at the root node - which means that the returned score is an exact value for the root node. For example, let v be the minimax score for n. If a null window search $(f-1, f)$ returns a score f, it holds that $v \geq f$. Then, MTD(f) re-searches n with $(f, f+1)$. If it returns f, $v < f+1$ (i.e., $v \leq f$) is proven. Therefore, the minimax score for n is assured to be f.

The transposition table is essential to the performance of MTD(f) because it prevents the same nodes from having to be searched again and again. Experiments show that MTD(f) outperforms NegaScout on the number of visiting total nodes on execution [36].

2.5 Conclusions

The $\alpha\beta$ search algorithm is used by many game-playing programs. $\alpha\beta$ is able to search close to the minimal tree using the enhancements that we looked at. However, it is still not adequate to enable game-playing programs to improve the quality of play. We need to search game-trees deeper and faster. The next chapter deals with parallel game-tree search algorithms in order to satisfy these requirements.

Chapter 3

Parallel Game-Tree Search

Parallelizing $\alpha\beta$ search is an important research topic, not only in Artificial Intelligence, but also in Parallel Computing. This chapter surveys the previous work on parallel game-tree search algorithms. The first section introduces the primary notions of parallel algorithms - *parallel performance* and *overheads* - to clarify the problems of parallel search. Section 3.2 explains essential problems of parallel search algorithms on distributed-memory systems. The rest of this chapter deals with previous approaches to parallel game-tree search algorithms.

3.1 Preliminaries

3.1.1 Parallel Performance

In order to show the effectiveness of a parallel algorithm, we need to measure how fast it is, in some way. A popular way to measure the relative performance of a parallel algorithm is the *speedup*. The speedup measures how much faster the parallel algorithm solved a problem than the *best* sequential algorithm and is defined as:

$$\text{speedup} = \frac{\text{solution time by the best sequential algorithm}}{\text{solution time by a parallel algorithm}}.$$

For a comparison of a parallel algorithm with a sequential one, using the fastest sequential algorithm on a single processor is very important. For example, when a parallel algorithm achieves a 2-fold speedup with 4 processors, compared with a sequential algorithm which is 4 times slower than the best sequential one, in effect, the speed of the parallel algorithm is half of the best sequential one. However, because we cannot define analytically what the best sequential algorithm is in game-tree search, we use the best optimized sequential algorithm we can write, as a baseline. Therefore, we must be careful in how the speedup is measured so as not to misinterpret the quality of speedups. Unoptimized sequential algorithms can achieve better speedups, compared to those possible with the optimized one. When applied to games, a poor sequential algorithm, which causes fewer cut-offs than the best one, will provide parallel algorithms with more opportunities to search the branches in parallel, which may increase the observed speedup.

The maximum speedup is n with n processors (*linear speedup*). Although a speedup better than n times than that of the sequential algorithm (*super linear speedup*) may happen in parallel game-tree search, on average this should not happen. If it does, then it likely implies that the sequential algorithm is inferior (e.g., the sequential algorithm could be improved by mimicking the parallel algorithm).

Efficiency is a measure that is similar to speedup, and it is defined as:

$$\text{efficiency} = \frac{\text{solution time by the best sequential algorithm}}{\text{solution time by a parallel algorithm } \times \text{ number of processors}}.$$

Efficiency is the fraction of the time that is used by each processor in performing the computation. For example, an efficiency of 0.5 means that the processors are used half of the time on the actual computation, while an efficiency of 1.0 (*linear speedup*) indicates that the processors are used all of the time.

In general, our goal is to achieve as high an efficiency as possible with a large number of processors.

3.1.2 Overheads

In order to improve the execution time through the use of parallelism, it is necessary to divide game-trees into subtrees for the processors to search in parallel. However, although the decomposition of game-trees achieves speedups,

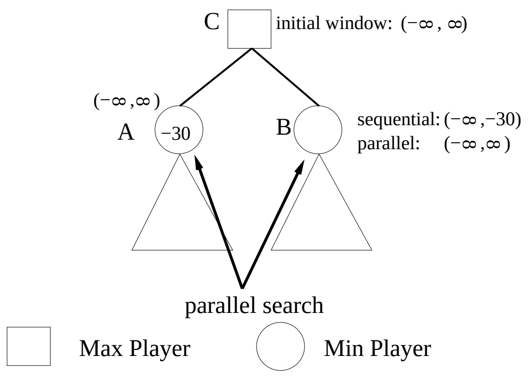


Figure 3.1: Example of Increasing Search Overhead

it causes *extra overheads* which a sequential algorithm does not have. There are three main overheads in parallel search algorithms.

Search overhead is the larger search tree that a parallel algorithm usually builds, compared with its sequential counterpart. It is defined as:

$$\text{search overhead} = \frac{\text{number of nodes searched by a parallel algorithm}}{\text{number of nodes searched by the sequential algorithm}} - 1.$$

There are several causes of search overhead. One possibility is that, at each CUT node, parallel algorithms may search unnecessary subtrees in parallel; these are subsequently proven unnecessary by one of the subtrees causing a cut-off. Another possibility is that the sequential algorithm has a better search window than that used by the parallel algorithm. Figure 3.1 illustrates an example of search overhead in the $\alpha\beta$ algorithm. Assume the initial window at a node C is set to $(-\infty, \infty)$, and the children A and B are searched in parallel. Parallel $\alpha\beta$ searches A and B with the window $(-\infty, \infty)$. On the other hand, sequential $\alpha\beta$ can update the window to $(-\infty, -30)$ by propagating the search result of A. Therefore, sequential $\alpha\beta$ will search B with the window $(-\infty, -30)$, resulting in fewer nodes searched than parallel $\alpha\beta$.

Synchronization overhead is the idle time wasted at synchronization points, where some processors have to wait for the others to finish their search. Some parallel algorithms gather information, such as search windows at a node, to search subsequent nodes with better bounds. Figure 3.2 illustrates an example of synchronization overhead. In order to get a better window than that in

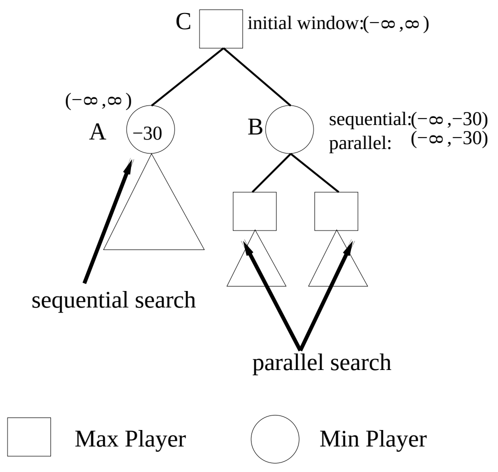


Figure 3.2: Example of Increasing Synchronization Overhead

Figure 3.2, parallel $\alpha\beta$ first searches A sequentially, then searches the children of B in parallel. The other processors have to wait for the minimax value for A, resulting in wasted idle time.

The notion of *load balancing* is an important influence on synchronization overhead, and is defined as:

$$\text{load balancing} = \frac{\text{maximal number of nodes searched by a processor}}{\text{average number of nodes searched by each processor}}.$$

Load balancing measures how evenly the work is distributed among the processors. Ideally, each processor should be given work that takes exactly the same amount of time. In practice, this does not happen, and some processors are forced to wait for others to complete their tasks. Although effective load balancing does not necessarily mean that there is a low synchronization overhead, in general bad load balancing implies that a parallel algorithm has a large synchronization overhead.

Communication overhead is caused by the cost of exchanging information between processors. Obviously, communication overhead never occurs in sequential algorithms because there is no need to exchange information.

Our purpose is to decompose the game-trees to maximize the speedup of our parallel algorithm. In other words, we want to minimize the overheads mentioned above. However, we have to note that these overheads are not independent of each other. For example, if we reduce the communication and synchronization overheads, the search overhead will likely be increased. On the other hand, reducing search overhead may increase the other two. In fact, because it is difficult to find the best case which minimizes the combination of these overheads, most researchers attempt to maximize the performance of their parallel $\alpha\beta$ algorithms by running a large number of experiments and choosing the combination of algorithmic features that gives the best performance. Researchers also study theoretical frameworks of parallel $\alpha\beta$ algorithms. However, the performance estimated by theoretical analyses is usually different from that achieved by experiments, because theoretical approaches sometimes assume the conditions which are unlikely satisfied in practice. For example, Hsu not only measures the experiments for his parallel $\alpha\beta$ algorithm, but also studies the theoretical analysis [20]. However, Hsu's theoretical analysis assumes that the nodes are perfectly ordered at CUT nodes, while nodes are not always ordered perfectly in practical parallel $\alpha\beta$ algorithms.

The notion of *granularity* is important in parallel computing. Granularity is defined as the amount of work that is done between communications. Coarse granularity implies large computation, resulting in relatively fewer communications - which may cause a load imbalance. On the other hand, fine granularity usually achieves good load balancing. However, it may have more communication overhead than coarse granularity, because it divides a larger number of tasks among the processors by exchanging messages. Therefore, granularity depends on systems such as CPUs, and network latencies. In game-tree search, the term granularity is used to refer to the search depth of a subtree, since this determines the size of the computation.

3.2 Issues of Distributed-Memory Systems

From the viewpoint of hardware technologies, we basically have two types of parallel machines: *shared-memory* and *distributed-memory* systems. A shared-memory multiprocessor system has a global address space for main memory, which means that each processor can access all of the main memory. Therefore, when a processor wants to access data created by another processor, the data is accessed as easily as if it were its own data. In a distributed-memory system, each processor has a local address space for its local memory, and can access only its own memory. Because a processor cannot directly access the memory of other processors, it has to do so by sending messages over an interconnection network.

As described in Chapter 2, practical $\alpha\beta$ search algorithms utilize transposition tables. Therefore, when parallelizing $\alpha\beta$ search on distributed-memory machines, the efficient implementation of transposition tables becomes a serious problem. Because each processor does not share its memory on a distributed-memory machine, it cannot access the transposition table entries of the other processors without any communication.

There are three naive ways to implement transposition tables on distributed-memory machines.

(1) **Partitioned transposition table:** Each processor keeps a disjoint subset of the table entries. This can be seen as a large transposition table divided among all the processors (e.g., [16]). Let L be the total number of table entries with p processors, then each processor usually has $\frac{L}{p}$ entries. When a processor P needs a table entry, it sends a message to ask the processor Q, which keeps the corresponding entry, to return the information to P. P has to wait for Q to send back the information on the table entry to P. When the processor P updates a table entry, P sends a message to the corresponding processor to update the entry. Updating messages can be done asynchronously.

(2) **Replicated transposition table:** Each processor has a copy of the same transposition table. Looking up a table entry can be done by a local access because each processor has its own local copy of the information. Updating an entry requires a broadcast to all the other processors so that they can update their tables with the new information.

(3) **Local transposition table:** Each processor has its own transposition

table. No table entries are shared among processors. Looking up and updating an entry can be done by a local access.

These implementations of the transposition table produce serious bottlenecks, which negates some of the benefits of the parallelization of sequential $\alpha\beta$:

(1) The partitioned transposition table has a high overhead because communication occurs for most of the table accesses. Looking up an entry is expensive because it incurs both synchronization and communication overheads. The communication delay for lookup operations is at least twice as high as the network latency.

(2) In the replicated transposition table, updating an entry has a large overhead because of broadcast updates. Even if messages for updates can be sent asynchronously, and multiple messages can be sent at one time by combining them as a single message, the communication overhead increases as the number of processors increases.

(3) The replicated transposition table has fewer entries than a partitioned transposition table. Though one of the advantages of using distributed-memory machines is having more memory, the replicated transposition table does not make use of this advantage.

(4) Though the local transposition table has no communication and synchronization overhead, it usually produces a large search overhead. A processor may end up computing a piece of work that already exists in the transposition table of another processor. Therefore, local transposition tables are only good for a small number of processors [29].

(5) These approaches may perform redundant search in the case of a DAG (Directed Acyclic Graph). In a DAG, a node may have more than one parent, while every node except the root has only one parent in a tree. A search result is stored in the transposition table *after* the search is complete. If two identical nodes are allocated to two different processors

in the case of a DAG, then duplicate search may occur, which increases the search overhead.

Because the efficient implementation of transposition tables in a distributed environment is a challenging problem, researchers have been looking for better solutions [45, 8]. We will describe an efficient way to implement transposition tables in a distributed environment in the case of single-agent search in Section 3.4.

3.3 Previous Work on Parallel $\alpha\beta$ Search

Numerous parallel $\alpha\beta$ search algorithms have previously been proposed. (See [7] for a broad survey in this field). Although these parallel algorithms differ, there are two classes of algorithms widely used - *synchronous* and *asynchronous*. Synchronous algorithms force some nodes in the game-tree to be completed before other nodes can be searched; this reduces the search overhead, at the expense of synchronization overhead. Asynchronous algorithms allow processors to search nodes independently of other processors, which can reduce the synchronization overhead, but has the drawback of increasing the search overhead. Section 3.3.1 presents synchronous parallel $\alpha\beta$ search algorithms, using YBWC (Young Brothers Wait Concept) as an example. Section 3.3.2 deals with asynchronous algorithms, using APHID (Asynchronous Parallel Hierarchical Iterative Deepening) as an example.

3.3.1 Synchronous Algorithms

Highly optimized sequential $\alpha\beta$ search algorithms have good move ordering schemes. A well-ordered tree has the property that if a cut-off is to occur at a node, the first move has a high probability of achieving it. YBWC [15] states that the left-most branch at a node must be searched before the other branches at the node are searched. This notion can be generalized so that all the promising branches must be searched before the rest of the branches at a node are searched. This generalization is applied to Feldmann's YBWC* algorithm [15]. Although there are many variants of YBWC, the only differences are in the implementation details [25, 53, 21]. The idea of YBWC originates from PV-Split [28], which applies the above strategy only at PV nodes. YBWC can be seen as a generalized version of PV-Split because the strategy of YBWC is applied to all nodes. YBWC has the following benefits: First, the minimal game-trees can be parallelized with no search overhead in YBWC. This is because parallelization is applied based on the properties of best-ordered trees, in which cut-offs are caused by the values of the left-most nodes. Second, YBWC may limit search overhead even if the game-trees are not best-ordered. At each CUT node, the left-most branch of the node can cause a cut-off with a high probability, while searching the left-most branch first at each PV node can improve the search window for searching the rest of the branches.

However, synchronous algorithms such as YBWC and PV-Split present two problems. First, these synchronous algorithms initiate parallelism too quickly at CUT nodes. For example, if the second left-most branch causes a cut-off at a CUT node in YBWC, searching the other branches is unnecessary. YBWC* can delay parallelism if some branches also have a high probability of causing a cut-off. Application-dependent information can be used to initiate parallelism. Second, YBWC suffers from severe overhead at synchronization points (Figure 3.3 is helpful in understanding synchronization points). Assume that the strategy of YBWC is applied to all nodes. At the beginning, no parallelism is allowed until the search reaches a leaf node. After evaluating that leaf node, the right siblings of the leaf node are searched in parallel. However, the number of siblings is usually smaller than the number of processors. As the search continues, subtrees allocated to the processors become larger and these subtrees themselves are split into smaller pieces, enabling more processors to work. Thus, YBWC inherently causes a synchronization point for every node of the left-most branch; one processor is busy and the remaining processors will be idle. Furthermore, by using iterative deepening, the number of synchronization points increases, which hurts performance.

Because synchronization overhead degrades the speedups, researchers have attempted to reduce the number of synchronization points. Synchronization points become a more serious problem in a game-tree where the branching

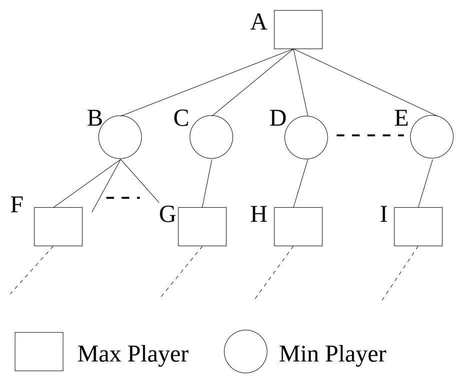


Figure 3.3: Synchronization Overhead in YBWC

factor is small, such as in checkers [26]. Feldmann's YBWC* searches all the branches at each type 3 node in order to reduce the number of synchronization points - based on the fact that no cut-off happens at type 3 nodes in $\alpha\beta$. The chess program, Zugzwang, was implemented based on YBWC* [15]. Deep Blue also allows parallelization of all the branches at type 3 nodes [14]. However, this approach may increase the search overhead, even though it reduces the number of synchronization points. Furthermore, it is clear that we need a way to determine if a node is a type 3 node. An educated guess can be made, as to which are the type 3 nodes, but may be wrong and thereby incur extra parallel overhead. Zugzwang and Deep Blue parallelized NegaScout, which must traverse all the children of each type 3 node. Searching all the children at a type 3 node in parallel may incur extra search overhead in parallel MTD(f), because MTD(f) can cause cut-offs, even at type 3 nodes.

Another problem in using synchronous search algorithms is load balancing. Even if each processor is allocated the same number of nodes, some processors may finish searching their nodes faster than others. To solve this, some programs rely on *work-stealing*. The work-stealing framework is a popular approach in two-player games (e.g., Zugzwang [15], Multigame's runtime system [41], *Socrates [22], and its successor Cilkchess [4]). Each processor has a local work queue containing nodes which can be distributed to other processors. If the local work queue of a processor is not empty, it dequeues a node from its local work queue, traverses the node, and enqueues new children of the node into its own local work queue. When the local work queue has no work to do, the processor steals a node from another processor. The processor to steal from is selected randomly. Although the work-stealing framework improves the load balancing, it has a trade-off on distributed-memory machines, because of the transposition table. Assume that a node is moved from a processor which has information on the node locally in its transposition table, to a processor which does not have any transposition table information. In this case, using a local transposition table will increase the search overhead because of the lack of relevant transposition entries at the processor which received the stolen work. Using a partitioned transposition table causes a communication latency in looking up the transposition entry for the node. Replicated tables also have extra communication overhead when broadcasting the result of searching the node.

ABDADA is a variant of YBWC [53]. However, ABDADA does not use work-stealing. Instead, ABDADA uses a shared transposition table to control the parallel search. All the processors start searching the root node simultaneously. Each transposition table entry has a field for the number of processors entering a node, which is used to determine the order in which to search children of that node. ABDADA achieved greater speedups than YBWC in chess and Othello on a shared-memory machine. However, it is hard to implement ABDADA on distributed-memory machines because of the necessity of sharing the transposition tables.

3.3.2 Asynchronous Algorithms

Because synchronous approaches have synchronization points where some processors have to wait for others to finish searching, researchers have attempted to get rid of the synchronization points by using asynchronous approaches. Newborn's UIDPABS first tried this approach in his chess program, but the speedups were worse than PV-Split [33]. However, Brockington's APHID, which generalizes UIDPABS, recently showed better speedups than YBWC in Othello and checkers, and comparable speedups to YBWC on two chess programs [8, 10].

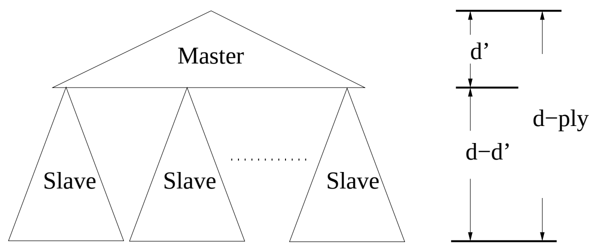


Figure 3.4: APHID

APHID [8, 10] is constructed with a *master* processor and a series of *slave* processors (Figure 3.4). Roughly speaking, when conducting a d-ply search, the master repeatedly searches the first $d'(< d)$ ply of the game-tree from the root node, using the $\alpha\beta$ algorithm, while the slaves independently search the remaining $d - d'$ ply with the iterative deepening $\alpha\beta$ algorithm. If a node at depth d' is not searched by the slave to the depth $d - d'$, the master cannot use the precise value of the node. Therefore, when the master visits a node at depth d' for the first time, the node is marked *uncertain* and assigned to a slave. The master uses the guessed scores for uncertain nodes until a slave searches the node to depth $d - d'$. The master traverses the root node to depth d' until no node visited by the master is assigned an uncertain mark. The slave processors simply search nodes allocated by the master in an iterative-deepening manner, and report the result back to the master for each iteration. Because no information is exchanged among them, it has no communication and synchronization overheads among the slaves.

Load imbalances can happen in APHID. The master processor tries to effectively manage load balancing. There are two reasons why load imbalance can occur. One is that the number of nodes assigned to the slaves is not always equal; the other is that some pieces of work are much larger than others. For example, work on the principal variations is usually larger than other pieces of work in the search. APHID has two strategies for load-balancing. If there is an *underworked* slave which has no node to search, the master moves nodes to the underworked slave from an *overworked* slave that has a large number of nodes to work on. There is a trade-off based on the implementation of transposition tables in a distributed environment, because moving nodes may result in not reusing the information in the transposition tables. The other load balancing technique used in APHID is that it breaks a large piece of work into smaller pieces in order to facilitate distributing them to multiple processors. The approach of APHID is that the master extends its search horizon to a depth greater than d'. The master stores the information on the largest pieces of work explored for the last few iterations, as well as the average size of each piece of work. If the largest piece of work is v times bigger than the average size, the master searches that node 2-ply deeper, by splitting the work into multiple pieces and distributing them to the slaves. Therefore, Figure 3.4 is misleading because the depth to which the master searches is not always d'.

Compared to synchronous algorithms, APHID has the following advantages:

(1) APHID has no synchronization points, which are the serious bottlenecks in YBWC within an iteration and between iterations.

(2) APHID does not require a shared transposition table for distributed-memory machines, because the master always stores the most important transposition entries - the first d'-ply of the search. The table access to these important entries is controlled locally by the master.

(3) APHID solves a major problem of initiating parallelism at CUT nodes. In YBWC*, application-dependent knowledge is used to avoid parallelizing the branches that have high probabilities of causing cut-offs. On the other hand, APHID can parallelize the only nodes which are unlikely to be pruned by using the best information available, based on application-independent criteria (i.e., guessed scores). Therefore, APHID can limit the increase of search overhead at CUT nodes (see [10] for detailed descriptions on this problem).

(4) APHID is designed to easily integrate parallelism into existing sequential $\alpha\beta$ search algorithms. For example, while APHID was integrated into the checkers program, Chinook [48], with one afternoon of effort, the

parallel version of Chinook used in the 1994 match against Tinsley took 4 to 6 weeks to implement [9].

Although APHID provided the answer to the question of whether asynchronous algorithms could outperform synchronous algorithms, there is still further work to be done. The sequential counterpart Brockington *et al.* parallelized is NegaScout. Brockington mentions in his thesis that APHID does not perform well on a sequential search algorithm enhanced with an aspiration window at the root of the tree, such as MTD(f) [8]. It is still an open question whether or not APHID can outperform YBWC using MTD(f).

3.4 TDS

In Section 3.2, the bottlenecks of parallel search algorithms on distributed systems were described. For single-agent search, such as solving the sliding tile puzzles and Rubik's cube, Romein *et al.* investigated a novel parallel algorithm called TDS (Transposition-table Driven Scheduling) [43]. TDS is a new scheduling algorithm on distributed-memory systems. Its idea can be applied to any parallel search algorithms which use a transposition table, although TDS was used by Romein *et al.* to parallelize IDA* [43, 41]. In this section, we first describe IDA*, a sequential algorithm for single-agent search. We then present TDS. Finally, we discuss the applicability of TDS to two-player games.

3.4.1 IDA*

While the goal of two-player search is to find the best move by traversing game-trees, the goal of single-agent search algorithms is to find a path from a given problem to a solution, with minimal cost. IDA* [24] and A* [24] are used for single-agent search. While A* is a best-first search algorithm, IDA* is a modification of A* based on depth-first iterative deepening searches [50]. IDA* , therefore, uses less memory than A*.

IDA* has two heuristic values. $g(n)$ is the cost already spent to reach a node n from the root. The *heuristic function* $h(n)$ is an estimated cost to a goal node from n. The total estimated cost $f(n)$ at n is $f(n) = g(n) + h(n)$.

IDA* contains a bound to cut off searching nodes which exceed the bound. At first, the bound is set to the estimated cost at the root node, $h(root)$. For each iteration, IDA* traverses nodes in a depth-first manner until it exceeds the bound or finds a solution. When an iteration fails to find a solution, the bound is increased to the minimum value that exceeded the previous bound. This continues until IDA* finds a solution, or resources are exhausted.

Figure 3.5 presents pseudo code for IDA*. For the sake of simplicity, we assume that the cost from a node to its child is always 1. The algorithm consists of two parts: IDA* controls the search bound at the root node for each iteration, and *DFS* expands nodes in a depth-first manner. In *DFS*, if the estimated cost from a branch of a node to a solution exceeds the bound, it does not expand the branch, and goes to the next branch. Otherwise, *DFS* dives into a child of the node by recursively calling *DFS*.

The heuristic function $h(n)$ plays an important role in IDA*. $h(n)$ is analogous to an evaluation function in two-player games. If $h(n)$ never overestimates the cost to the goal state, IDA* will find an optimal solution whose cost is minimal. This property is called *admissibility*.

Similar enhancements used for the $\alpha\beta$ algorithm, such as transposition tables, can also be used in IDA* to improve the performance of the algorithm [40]. Hence, sharing the transposition tables becomes an issue when parallelizing IDA* on distributed-memory machines.

3.4.2 TDS Algorithm

In a traditional approach, such as work-stealing, it is hard to share a transposition table on distributed-memory systems, because which processor a piece of work ends up on is decided dynamically. Therefore, the processor receiving a piece of work may not have any relevant transposition table entries in its local memory. TDS is a parallel IDA* algorithm which uses a different approach for solving the transposition table problem [43, 41]. In TDS, transposition tables are partitioned over the processors. Then, whenever a node is expanded, its children are scattered to the processors which keep their transposition table entries, instead of utilizing remote lookups to the processors. Figure 3.6 il-

```
int solved = false;

int IDA*(node_t root) {
   int new_bound = $h$(root);
   do {
      /* Perform a depth-first iterative deepening search */
      new_bound = DFS(root,new_bound);
   } while (!solved || new_bound $\neq\infty$);
}

int DFS(node_t n, int bound) {
   int i, b,  new_bound = $\infty$;
   if ($h$(n) == 0) {
      solved = true;
      return 0;
   }
   for (i = 0; i < n.num_of_children; i++) {
      if (1 + $h$(n.child_node[i]) $\leq$ bound)
         /* Search deeper */
         b = 1 + DFS(n.child_node[i],bound-1);
      else b = $h$(n) + 1;
      if (solved)  return b;
      /* Update the bound for next iteration */
      new_bound = min(new_bound,b);
   }
   return new_bound;
}
```

Figure 3.5: Pseudo Code for IDA* (the cost is always 1)

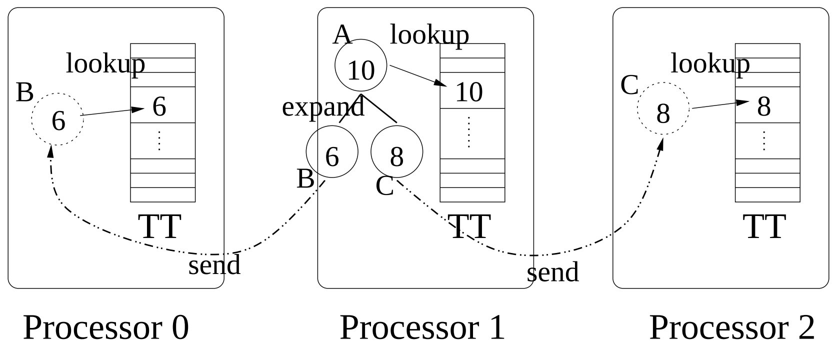


Figure 3.6: Transposition-table Driven Scheduling

lustrates the ideas of TDS. In this figure, we show 3 processors, and a circle stands for a node. Each position is assigned a number k (see integers inside the nodes). In practice, the transposition table key of a node is used to assign the unique number to the node. $F(k)$ determines the processor that the position should be moved to. In this example, we define F as $F(k) = k \pmod 3$. Transposition tables are partitioned to each processor. The transposition entry of a position with a number k is assigned to Processor $F(k)$, so as to enable looking up this table entry locally. Let A be a node that is expanded now, and let us assume that B and C are the children of A. When B and C are created, they are moved to the processors which have their transposition table entries. Therefore, B (6 is assigned to this position) is sent to Processor 0, because $F(6) = 0 \pmod 3$, and C is sent to Processor 2.

Intuitively, TDS seems to have much more communication overhead than the work-stealing framework. In fact, however, TDS achieves lower communication overhead because all the transposition table accesses are local. As well it eliminates the synchronous communication of the remote table lookups such as occurs in partitioned transposition tables.

In TDS, each processor has a local work queue which contains nodes to be searched, and part of the transposition table accessed by its own processor. The local work queue is implemented as a stack to maintain the depth-first manner of IDA*.

Figure 3.7 presents a sketch of TDS. In *MainLoop*, each processor repeatedly attempts to dequeue a node from its local work queue, and then expands the children of the node. The cost to reach a solution for each child is calculated by the heuristic function $h(n)$. If the estimated cost of the child does not exceed the search bound, the child is sent to a target processor (called a *home processor*) which has the transposition table entry for the child (assigned by the function *HomeProcessor*). The home processor of a node is computed from the transposition table key of the node. Transposition table keys are calculated by the function *TTKey*. After expanding all the children, their parent node is erased from the local work queue because the search results of the children do not need to be reported back to their parent in IDA*. Each processor continues the above procedure as long as it has any nodes to be expanded in its local queue. If a local queue is empty, it may mean that it has expanded all the nodes for that IDA* iteration. Therefore, processors check whether a solution has been found; if not, they set a new search bound to start a new iteration of IDA* (called *global termination*). *ReceiveNode* is called repeatedly to receive work. When a processor receives a node, it checks its transposition table entry by doing a local lookup to check if the node was previously searched with an adequate bound. If the node has already been adequately searched, the processor does not need to re-search the node; therefore, the node is discarded and not stored in the local queue. Otherwise, the information on the node is stored in the transposition table in order to avoid duplicate search, and then it is pushed onto the local queue to be searched.

TDS has the following advantages [43]:

(1) All transposition table accesses are local. Nodes are always moved to the processor which may have information on the node in its transposition table.

(2) All communications are asynchronous. Processors do not have to wait for messages. Therefore, TDS achieves close to linear speedups with a large number of processors.

(3) No duplicate search is performed as long as a processor does not use

```
void MainLoop() {
  int i, dest, solved = FALSE;
  while (!solved) {
    node_t n = GetLocalJob();
    if (n $\neq$ NOT_FOUND)
      for (i = 0; i < n.num_of_children; i++) {
        if ($h$(n.child_node[i]) + 1 $\leq$ n.bound) {
          n.child_node[i].bound = n.bound - 1;
          dest = HomeProcessor(TTKey(n.child_node[i]));
          SendNode(n.child_node[i],dest);
        }
    }
    else solved := CheckGlobalTermination();
  }
}

void ReceiveNode(node_t n) {
  trans_entry_t entry = TransLookup(n);
  if (!entry.hit || entry.bound $\leq$ n.bound) {
    TransStore(n);
    PutLocalJob(n);
  }
}
```

Figure 3.7: Simplified TDS Algorithm

up all the transposition table entries. Even if a node is reached from different paths, the node is always allocated to the same processor.

(4) TDS achieves more stable execution times than work-stealing frameworks.

(5) It is not necessary to have a separate load balancing scheme. When using IDA*, Zobrist's method [54] is used to calculate the transposition table keys, which are uniformly distributed. Uniformly distributed transposition table keys distribute the work evenly to the processors.

The performance of TDS was measured on the 15-puzzle, the double-blank puzzle, and Rubik's cube. In these experiments, TDS achieves speedups between 109 and 122 on 128 processors, while the traditional work-stealing framework with replicated/partitioned transposition tables achieves speedups between 8.7 and 62. On a system containing multiple clusters, connected by high-latency and low-bandwidth wide-area links, TDS with some modifications (Wide-Area TDS) also achieves better performance than the work-stealing framework, by a wide margin [42].

3.4.3 Applicability of TDS to $\alpha\beta$ Algorithms

The idea of TDS seems to be easily applied to two-player games. However, there are important differences between IDA* and $\alpha\beta$ algorithms that complicate the issue.

(1) IDA* does not have $\alpha\beta$-like pruning. When traversing the children of a node, IDA* merely checks their bounds. In other words, the results of the children are not reported back to its parent. On the other hand, $\alpha\beta$ checks if a cut-off happens, after searching a branch of a node. When a cut-off occurs, we need a mechanism to not only receive the scores reported from the children, but also to tell the other processors to stop searching other branches, in order to avoid unnecessary search.

(2) $\alpha\beta$ search has a window which IDA* does not have. Search windows make the implementation of TDS complicated because (a) the window

may be narrowed after searching a node, and (b) a node reached through more than one path may be searched with different windows.

(3) The search order of nodes in $\alpha\beta$ is much more important than in IDA*. In IDA*, the implementation of the work queue is a stack to allow TDS to behave in a depth-first manner. In parallel $\alpha\beta$ search, we need a more complex scheme in order to allow the left-most and shallowest nodes to be searched first.

For these reasons, parallel $\alpha\beta$ based on TDS has been considered hard to parallelize. The next chapter describes integrating TDS into an $\alpha\beta$ framework.

Chapter 4

TDSAB

In this chapter, we propose a new parallel $\alpha\beta$ algorithm, TDSAB (Transposition-table Driven Scheduling Alpha-Beta). TDSAB is an application of TDS to the $\alpha\beta$ algorithm. This chapter first gives an explanation of the basic TDSAB algorithm, and then it deals with the implementation details for the games of Awari and Amazons. Finally, it discusses the difficulty in design and implementation of parallel search algorithms.

4.1 Basic Algorithm

4.1.1 Overview

As we explained in the last chapter, applying TDS to the $\alpha\beta$ algorithm presents the following difficulties:

(1) **Pruning:** $\alpha\beta$'s scheme for pruning is different from that of IDA*.

(2) **Search windows:** $\alpha\beta$ may search identical nodes with different windows if the search space is a DAG.

(3) **Priority of nodes:** The order in which nodes are considered is more important in $\alpha\beta$ than in IDA*. In parallel $\alpha\beta$ based on TDS, a more complex scheme is required to manage the order in which nodes are searched than is the case in IDA*. In IDA*, a stack is sufficient; in TDSAB, this method is inadequate.

The solution to the above issues is facilitated by the fact that TDSAB consists of the following three combinations of algorithms:

$$\text{TDSAB} = \text{MTD}(f) + \text{YBWC} + \text{TDS}.$$

- **MTD(f):** TDSAB uses MTD(f) as its sequential version of $\alpha\beta$, which solves the problem of searching identical nodes with different search windows. MTD(f) consists of null window searches initialized at the root; therefore, each search has the same search window at each node. If we always synchronize parallelization at the root, we do not have to be concerned with searching nodes with different search windows. Using MTD(f) as the sequential algorithm makes it simpler to parallelize $\alpha\beta$. However, parallel MTD(f) may achieve fewer speedups than parallel $\alpha\beta$, because MTD(f) causes more cut-offs, in general, and requires more synchronization.

- **YBWC:** Parallelism is restricted by YBWC. If the left-most branch of a node does not cause a cut-off, TDSAB traverses the rest of the branches in parallel. Because the left-most branch of a node has a high probability of causing a cut-off, YBWC can avoid the huge increase in search overhead that arises from simply searching all the branches in parallel.

- **TDS:** The distribution of nodes to processors is based on TDS, and therefore, TDSAB retains the important merits of TDS. Nodes to be searched are moved to the processors which may have transposition table entries for those nodes.

TDSAB addresses to the TDS scheme of moving work to the data. Several issues need to be resolved in TDSAB:

(1) **Priority queue:** To search the nodes in approximately the same order as sequential $\alpha\beta$ does, we keep the nodes in a *priority queue*, similar to that used in Brockington's APHID [8].

(2) **Signatures:** For the purpose of solving the issue of cut-offs, we propose a *signature*, which expresses, roughly speaking, a path from the root to a node. A signature reduces the number of messages by only broadcasting to all the processors when a cut-off happens; as well, it detects cyclic positions when searching cyclic graphs such as in Awari.

(3) **Synchronization of nodes:** The search order of identical nodes must be considered carefully in the case of cyclic graphs, in order to avoid deadlock. Our strategy for synchronizing identical nodes guarantees that deadlock cannot occur.

4.1.2 Priority Queues

In sequential game-tree search, moves are ordered from the best to the worst, in order to prune as many nodes as possible. Good move ordering must be kept in the parallel search as well. Our solution in TDSAB for selecting the next node to be searched is similar to that used in APHID [8]. Each node has a priority based on how "left-sided" it is, as Brockington explains in [8]. To compute the priority of a node, the path from the root to that node is considered. There are 3 priority scores for a node, based on the type of node. The priority of a node is the sum of priority scores along the path from the root to that node. For example, assume that we wish to calculate the priority of a node A. For each PV node which is on the path from the root node to A, the highest score is added to the priority of A. For each left-most node that is not a PV node, the second highest score is added to the priority of A. For the other nodes, nothing is added to the priority of A. For our current implementation, the highest score is 4 and the second highest is 2. We note that the calculation of the priorities can be done inexpensively. When a node A creates a child B, the priority of B is computed by the sum of the priority of A and the score of B. This scheme is simple and could be improved by calculating the order of nodes more precisely, which would require more expensive computation.

TDSAB maintains a doubly-linked list of priority buckets for each depth. A bucket contains a fixed number of nodes with the same priority. For each depth, priority buckets are sorted in decreasing order, so as to easily find a node with the highest priority and the shallowest (farthest from the root) depth. The strategy of TDSAB to determine which node is to be expanded next is as follows:

(1) A node with deeper (closer to the root) depth and the highest priority is chosen in the following case: Let *high* be the highest priority of the nodes in the depth d queue. A node P searching to depth d with the priority *high* is selected if (a) any node whose priority is less than *high* is currently being searched, and (b) P is not currently being searched.

(2) Otherwise, a node with the shallowest depth and the highest priority is selected among the nodes which (a) have not been expanded yet, and (b) have received the score for the first left-most branch with no cut-off occurring. The shallowest depth is considered first, and the priority is considered next.

We note that (1) is important in order to keep the left-sided order of exploring game-trees, which happens in the case of a processor receiving a node with a higher priority, after expanding a node with a lower priority.

4.1.3 Signatures

Let P be a node and Q be a child of P. When searching the children of P in parallel, if Q returns a score that causes a cut-off at P, searching other children of P is not necessary. If searching any other children is currently under way, then it must be stopped. TDSAB, therefore, has to stop any useless searches in order to avoid increasing the search overhead. However, because all the descendants of P are not always on the same processor in the TDS framework, we have to consider an efficient implementation for cut-offs in TDSAB. In a naive implementation, when a cut-off happens at a node, the processor searching the node has to send a message to all the processors searching its children, asking them to stop searching. Then, the processors searching the children of the node send messages to the processors searching

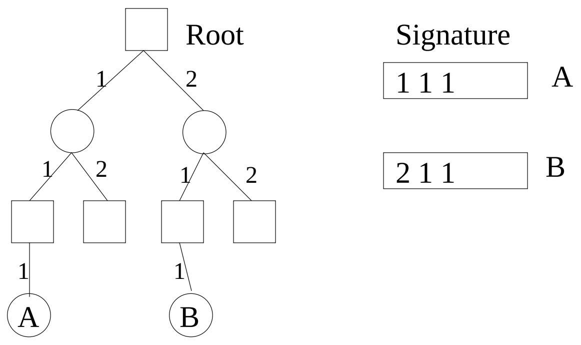


Figure 4.1: Signatures

the grandchildren of the node asking them to abort searching, and so on. This approach clearly results in the exchange of many messages which can lead to a large increase in communication overhead, and also to delay in killing unnecessary work - which, in turn, results in more search overhead.

In TDSAB, when a cut-off occurs, we reduce the number of messages exchanged by using a *signature*. Intuitively, the signature for P is the path traversed from the root node to P. Every branch of a node has a tag which differentiates it from other branches at that node; a signature of P is seen as a sequence of these tags from the root to P. Figure 4.1 illustrates an example of signatures. The decimal number on each branch between two nodes is the tag. The signature of A is 111 derived from the path from the root to A; the signature of B is 211.

When a cut-off happens at a node P, all TDSAB is required to do is to broadcast the signature of P to all the processors. When a processor receives a cut-off signature, it examines its local priority queue, and deletes all the nodes which have the same paths from the root to P. For example, in Figure 4.1, if TDSAB wants to prune all the children of A, the signature 111 is broadcast and each processor prunes all the nodes that begin with the signature "$111\cdots$".

For our implementation, we express signatures as n moves each of m bits, because we can compress each move at a node into a unique integer by using the index of the moves generated. For example, in Awari each move can be encoded into a 3 bit tag because there are at most 6 moves in each position.

Therefore, a signature with 96 bits indicates a 32-ply search from the root. In Amazons, our implementation encodes each move into a 16-bit tag and expresses a signature as a 160-bit integer for a 10-ply search.

With cyclic graphs, we need a method for detecting cycles because a practical implementation of $\alpha\beta$ scores a cycle as a draw and does not perform any additional search.[1] We will explain how cycles are detected in the Section 4.2.2.

4.1.4 TDSAB Algorithm

Figures 4.2 and 4.3 present the pseudo code of the Negamax form of TDSAB. For the sake of simplicity, we explain TDSAB without YBWC. Like the IDA* version of TDS, TDSAB repeatedly (a) receives a node from other processors, (b) enqueues it into its local priority queue, (c) selects a node to expand from its local queue, and (d) sends the children of the node to the appropriate processor.

The function *ParallelNWS* does one iteration of a null window search $(\gamma - 1, \gamma)$ in parallel, until the score for the root becomes a fail high or fail low. The end of the iteration is checked by the function *FinishedSearchingRoot*, which can be implemented by broadcasting a message when the score for the root has been decided. The function *RecvNode* checks regularly to determine if new information has arrived at a processor. We note that messages are exchanged asynchronously. *RecvNode* receives 3 kinds of information:

(1) **NEW_WORK:** New node created by traversing a game-tree.

(2) **CUT_OFF:** Signature used for cut-offs.

(3) **SEARCH_RESULT:** Minimax score of a node decided after searching its descendants.

If new information arrives at a processor, *GetNode*, *GetSignature*, and *GetSearchResult* acquire information on a node, signature, and score for a node, re-

[1]This approach causes the GHI (Graph-History-Interaction) problem [34, 13, 6], which may lead to a different score with different paths for the same position. Nevertheless, games programmers ignore the GHI problem because it rarely happens in practice.

spectively. *GetLocalJob* determines a node to be expanded from its local priority queue, while *DeleteLocalJob* deletes a node from its queue. *SendNode* is a function to send a node to the processor chosen by the function *HomeProcessor*, which returns the processor having a transposition table entry to the node.

There are some extra lines in TDSAB compared to the IDA* version of TDS. (*), (+), (-), and (#) indicate code which does not appear in TDS. TDSAB needs to receive a score from the children and deal with cut-offs. (*) is the code for receiving a cut-off message, and a cut-off is implemented by using a signature. If a processor receives a signature, it examines its local queue and discards all the nodes traversed by the same path as the signature (*CutAllDescendant*), as explained in Section 4.1.3. A terminal node or small piece of work is immediately searched locally (i.e., this small piece is called *granularity* of the work) with a null window $(\gamma - 1, \gamma)$, and sent to its parent node by using the function *SendScore*. That node is then deleted because the node need not be searched further (code at (-) in *ParallelNWS*). When receiving a search result (code at (+)), TDSAB negates the score because of the Negamax form; it then has to consider two cases (*StoreSearchResult*): fail high and fail low. If a score proves a fail high, TDSAB does not need to search the rest of the branches, therefore, a fail high score is saved in the transposition table (*TransFailHighStore*). The node is then dequeued from the priority queue, and the fail high score is sent to the processor having its parent (*SendScore*). We note that TDSAB keeps information on nodes being searched, unlike the IDA* version of TDS. Only after a processor has completed searching a node is it discarded. Because searching the rest of the branches has already started, the processor broadcasts a signature to abort all useless search (see the code at (#)). When a fail low happens, a processor stores the maximal score of the branches. If all the branches of a node are searched, the fail low score for the node is stored in the transposition table (*TransFailLowStore*), and the score is reported back to its parent.

```
int $\gamma$; /* A search window is set to $(\gamma - 1, \gamma)$. */
const int granularity; /* Granularity depends on machines, networks, and so on. */

/* Null window search in parallel. */
void ParallelNWS() {
  int type, value;
  node_t p;
  signature_t signature;
  do {
    if (RecevNode(&type) == TRUE) { /* some information arrives */
      switch(type) {
        case NEW_WORK:  /* New work is stored in its priority queue. */
          GetNode(&p);
          Enqueue(p); break;
(*)     case CUT_OFF:  /* Obsolete nodes are deleted from its priority queue. */
(*)       GetSignature(&signature);
(*)       CutAllDescendant(signature); break;
(+)     case SEARCH_RESULT:
(+)       GetSearchResult(&p,&value);
(+)       /* The value is negated because of the negamax form. */
(+)       StoreSearchResult(p,-value); break;
        default:
          Error();
      }
    }
    /* Find the next node to be traversed. */
    GetLocalJob(&p);
    if (p $\neq$ NOT_FOUND) {
      if (p == terminal || p.depth $\leq$ granularity) {
        value = AlphaBeta(p,p.depth,$\gamma - 1$,$\gamma$); /* Local search is done for small work. */
(-)     SendScore(p.parent,value);
(-)     DeleteLocalJob(p);
      } else {  /* Do one-ply search in parallel. */
        for (int i = 0; i < p.num_of_children; i++) {
          p.child_node[i].depth = p.depth - 1;
          SendNode(p.child_node[i],HomeProcessor(TTKey(p.child_node[i])));
        }
      }
    }
  } while (!FinishedSearchingRoot());
}
```

Figure 4.2: Simplified Pseudo Code for TDSAB (Negamax, without YBWC)

```
/* A minimax value is computed and stored in the transposition table. */
void StoreSearchResult(node_t p, int value) {
  if (value $\geq \gamma$) { /* Fail high */
    /* Put the search result in the transposition table. */
    TransFailHighStore(p,value);
(-) SendScore(p.parent,value);
    /* Discard useless search. */
(#) SendPruningMessage(p.signature);
(-) DeleteLocalJob(p);
  } else { /* Fail low */
    p.score = MAX(p.score,value);
    p.num_received ++;
    if (p.num_received == p.num_of_children) {
      /* All the scores for its children are received. */
      TransFailLowStore(p,p.score);
      /* Send the minimax value to its parent. */
(-)   SendScore(p.parent,p.score);
(-)   DeleteLocalJob(p);
    }
  }
}
```

Figure 4.3: Simplified Pseudo Code for TDSAB (cont.)

4.2 Implementation Details

TDSAB has been implemented for the games of Awari and Amazons. These games have different properties that exhibit themselves in the different characteristics of a parallel search. The main differences between Awari and Amazons are:

(1) The average number of children at each node (*average branching factor*) in Awari is small (less than 6), while Amazons has bushy trees. For example, the initial position in Amazons has 2,176 moves. We need different strategies to deal with these different search spaces.

(2) The search space of Amazons is a DAG, while Awari has cyclic positions. In Awari, we need a mechanism for cycle detection. In both games we need to delay searching a node if more than one path leads to the node so as not to do a redundant search. Furthermore, we have to carefully treat the search order of these nodes in Awari, because some synchronization strategies may cause deadlock.

This section deals with the specific techniques applied to each game.

4.2.1 Amazons

Modification of YBWC

In YBWC, if the left-most branch of a node does not cause a cut-off, the rest of the branches are searched in parallel. However, YBWC distributes too many nodes to the processors in Amazons, which increases search overhead because of the large branching factor. Therefore, if the first branch of a node does not cause a cut-off, a smaller number of children at a time are searched in parallel. If all these branches return fail lows, a processor tries to expand another small number of children in parallel, and so on. For our current implementation, the number of children to be searched in parallel is set to 30 at PV (or ALL) nodes, 20 at CUT nodes with up to 15 processors, 50 at PV (or ALL) nodes, and 20 at CUT nodes with more than 15 processors. These numbers are tuned by hand to achieve the best speedup with some test positions.

Synchronization of Identical Nodes

TDSAB can easily detect identical nodes with different paths, because they are always kept on the same processor. When more than one path leads to an identical node in Amazons, the lengths of the paths are always the same. Assume A and B are identical nodes, and A is chosen for a search. For our implementation, if the expansion of B has already begun, A is not expanded. A waits for the score of B to be stored in the transposition table. On the other hand, if B has not yet been searched, A is expanded and B will wait for the score of A. Once a minimax score for the identical nodes is in the transposition table, the processor searching the identical nodes can immediately send this score to the processors having the parents of the nodes, because A is always searched as deep as B with the same search window.

4.2.2 Awari

Parallelism at the Root

In MTD(f), there are 2 possible ways to determine the minimax score at the root: one is to continuously lower the scores at the root in each iteration by returning fail lows, then return a fail high score in the last iteration. The other is to raise up the scores at the root by proving fail highs, and finally return a fail low score in the last iteration. Fail low searches expand more nodes than fail high searches, in general. While it is necessary to traverse all the children to prove a fail low for a node, MTD(f) may explore only one of the children for a fail high.

Because Awari has a narrow search tree, processors are often starved for work. Our implementation always expands all the children of the root node in parallel, increasing the amount of work and reducing the frequency of starvation.

This clearly decreases the idle time of each processor by generating a large amount of work to do, while possibly increasing search overhead. Searching all nodes in parallel can be a win when MTD(f) lowers the score, for both the synchronization and the search overhead. For fail low searches, all the children of the root node are always expanded to prove a score for the root is lower than the current window; this can reduce the synchronization points without increasing the search overhead. In the last iteration, MTD(f) proves an exact score for the root by returning a fail high score. Speculatively searching all the children of the root in parallel could increase search overhead, when compared to simply applying the normal YBWC strategy to the root. However, it is worth paying the price of extra search in the last iteration, because proving fail lows is more difficult than proving fail highs.

Even when MTD(f) is raising the score, searching all children of the root in parallel does not increase the search overhead as much as one might imagine. First, even in this case, MTD(f) needs to return a fail low in the last iteration in order to determine the score for the root - which requires more work than proving fail highs. If the number of fail high searches before the last fail low is small, reducing the idle time for the fail low reduces the execution time in total. For example, if a fail low happens immediately after a fail high, and proving that fail low is much heavier than proving the fail high, searching the root in parallel will achieve improvement - although some extra nodes may be searched in order to prove the fail high. Moreover, when we search all children in parallel at the root and expect a fail high, the search overhead is generally low because our priority queue scheme guarantees that the left-most branch is searched first.

However, we need to note that the parallelism at the root may change the search order from that obtained when applying the YBWC strategy to the root. Figure 4.4 illustrates an example. Let PV be the principal variations in the last iteration, and the number inside a node be a priority computed using the priority scheme explained in Section 4.1.2. Assume that P and Q are located in the same processor in the current iteration. If we allow parallelism at the root, the processor expands Q before P, because Q has a higher priority. In the basic YBWC, this situation never occurs because Q is always expanded after searching P is completed.

If P does not cause a fail high at the root and the root receives a fail high score because of Q, searching all the children in parallel may improve

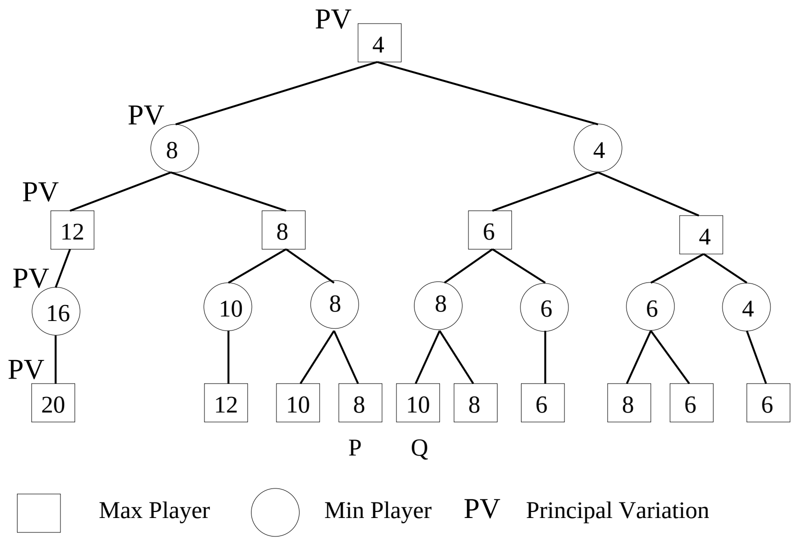


Figure 4.4: Problem of Parallelism at the Root

the running time. However, if P proves a fail high at the root, while Q does not, the search overhead will be increased. This problem will provide a further research topic for TDSAB.

Cycle Detection Using Signatures

The search space in Awari is a cycle. Therefore, $\alpha\beta$ search immediately returns an evaluation score of a draw when it detects a cycle; zero is usually used for the evaluation score of a draw. Although the ancestors of a node are not stored in the same processor in TDSAB, TDSAB is able to detect cycles by using signatures in the following way: First, TDSAB can always detect whether two nodes are identical, and if so, they are always allocated to the same processor. Then, TDSAB can detect whether the relations between two nodes are cyclic or not, by using the properties of signatures. The node with shallower depth should go by way of the identical node with deeper depth, if they are cyclic. Figure 4.5 shows an example of signatures with cycles, in which we assume B and B' are cyclic. In this case, the signature of B' should be "$1\cdots$", because B is on the way to B' and the signature of B is 1.

We point out that we do not suggest a solution to the GHI problem, but simply mention that TDSAB can detect cycles by using signatures.

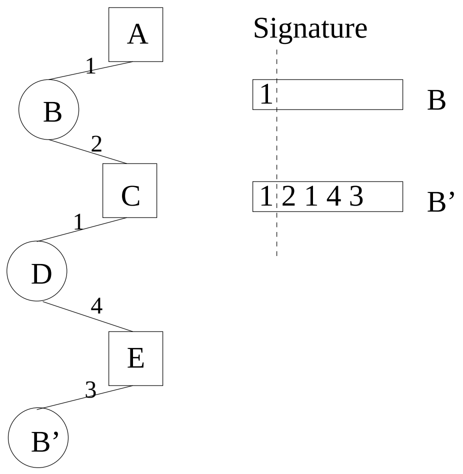


Figure 4.5: Cycle Detection

Synchronization of Identical Nodes

In Amazons, when a processor receives identical nodes with different paths, it expands only one of the nodes. The remaining nodes wait for the return of the score of the node, so as to avoid redundantly expanding these nodes. In Awari, identical nodes can have a different search depth, and may even have a cyclic position. Careless strategies for avoidance of a redundant search may cause deadlock. For instance, assume that n_1 and n_2 are identical nodes and n_1 has already been expanded. Then, if a processor always waits until it receives the score for n_1 before searching n_2, it may cause deadlock in cyclic graphs. Figure 4.6 illustrates an example of deadlock: Suppose that B and B' are identical nodes. If these nodes are searched in the following order, deadlock will occur:

(1) A is expanded, and B and E are sent to their home processors.

(2) B is expanded, and C is sent to its home processor. If the home processor of B receives a node B' identical to B, searching B' is delayed until it receives a score for B.

(3) E is expanded, and D is sent.

(4) D is expanded, and B' is sent. Searching B' is complete after finishing B.

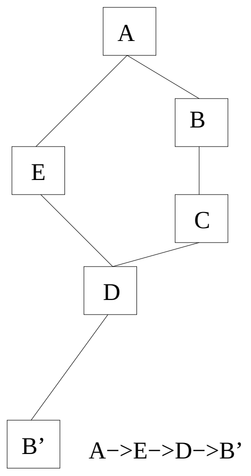


Figure 4.6: Deadlock with Cycles

(5) C is expanded, and D is sent.

In this case, B waits for the score for C, C waits for D, D waits for B', and B' waits for B. Therefore a cyclic wait has been created and deadlock ensues.

Looking at the above example again, B' is a repeated node of B if it comes by a path $A \rightarrow B \rightarrow C \rightarrow D \rightarrow B'$, which can immediately return a score using the technique explained above. However, B' is not a cyclic node if it is reached by the path $A \rightarrow E \rightarrow D \rightarrow B'$. Therefore, considering the path from A to B', B' must be searched in order to determine a score for E. We note that this problem is very similar to the GHI problem, which may return a wrong score for a node with a cyclic position, because the transposition table does not consider the path by which the node was reached.

To eliminate the possibility of deadlock, TDSAB searches the shallower of the identical nodes first. This prevents the search of a shallower node being delayed until a deeper identical node returns its score. The reason deadlock happens in Figure 4.6 is that B' waits for the result of B, even though the search depth of B' is 3-ply shallower than B. More specifically, let n_1 be a node for a d_1-ply search chosen by the priority queue strategy explained in Section 4.1.2. TDSAB expands the nodes as follows:

(1) If a processor has no identical interior nodes to n_1, n_1 is expanded for a

d_1-ply search.

(2) If n_1 is a cyclic node, it immediately returns the minimax score 0. The cyclic detection is done by using the signature of n_1.

(3) Let n_2 for a d_2-ply search be identical to n_1.

   (a) If $d_1 \geq d_2$ and n_2 has not been searched yet, n_2 is chosen to be expanded, and n_1 waits for n_2 to return its score.

   (b) If $d_1 \geq d_2$ and the search of n_2 has already begun, n_1 is searched after n_2 has finished being searched.

   (c) If $d_1 < d_2$, then n_1 is expanded, regardless of the state of n_2.

Note that this is similar to the solution for the GHI Problem (e.g., [6]). The good news is that freedom from deadlock is assured in our strategy. The proof is given in Appendix A. However, some nodes are searched more than once even if deadlock is not caused. For example, if no descendant of D is a repeated node of the ancestors of D in Figure 4.6, the above node synchronization scheme cannot avoid searching D more than once in the case of (3)(c).

4.3 Difficulty in TDSAB Implementation

Basically it is a difficult task to design and implement parallel programs. They have inconsistencies in execution, and sometimes deadlock occurs, making it much more difficult to debug parallel programs than sequential programs. Furthermore, TDSAB has asynchronous aspects of computation, which makes it more difficult to implement, although asynchronous communications reduce the idle time. Therefore, we need to write a TDSAB program very carefully. For example, debugging the implementation of signatures was difficult. TDSAB sends signatures and nodes asynchronously, and does not control which messages a processor receives first. In a practical environment, a signature for a cut-off sometimes arrives earlier than a node which should be pruned by that signature, because a node which will be pruned by that signature may have already distributed its children, which should be pruned as well. These children may arrive at a processor after that processor receives that signature. Therefore, it was necessary to keep signatures even after processors received them. In fact, even with our knowledge of parallel programming as well as game-tree search algorithms, it took several weeks to achieve stable TDSAB implementations for Awari and Amazons. This does not include the time to tune the performance, for example, setting the best granularity for the target environment.

There are several ways that games programmers may parallelize their sequential $\alpha\beta$ algorithms with a minimal amount of effort. Cilk [17] is an extension of the C language, with some primitives to support parallelism. Given a sequential $\alpha\beta$ algorithm, Cilk is a choice for implementing synchronous parallel $\alpha\beta$ algorithms, such as YBWC. However, we note that the runtime system of Cilk is based on the work-stealing framework, which has serious bottlenecks that affect transposition table performance on distributed-memory systems. Another choice for games programmers is to incorporate parallelism using the APHID library [8]. In this case, all we have to do to achieve parallelization is to add a small number of lines of code to the sequential programs. However, APHID does not achieve a very satisfactory speedup in a game with a larger branching factor, when compared to YBWC. Moreover, APHID does not perform well when parallelizing MTD(f). On the other hand, we need to implement TDSAB from scratch with the help of message-passing libraries. It is therefore necessary to help programmers to easily incorporate TDSAB into their game-playing programs. This will be a further research topic of TDSAB.

Chapter 5

Experimental Results

This chapter presents some performance results for TDSAB. Two different applications, Awari and Amazons, have been implemented in order to investigate the characteristics of TDSAB. Awari has a small branching factor which enables game-tree search algorithms to search deeper, while a large average branching factor in Amazons forces them to build shallower game-trees. We will show that TDSAB succeeds in achieving good results in both these games. Section 5.1 describes the experimental methodology used. The rest of this chapter analyzes the performance and bottlenecks of two applications. Section 5.2 deals with the experiments in Awari and Amazons, and compares our results to those reported by other researchers.

5.1 Methodology

5.1.1 Hardware Environments and Applications

Our environment consists of 50 dual Processor Pentium III PCs at 933 MHz, provided by Computing and Network Services at the University of Alberta. Each PC has 512 MB of memory and the machines are connected through a 100 Mb/s Ethernet. The shared memory between dual processors was not exploited. (p / 2) ($p \geq 2$) PCs were used for the experiment with p processors. The experiments used up to 64 processors (i.e., 32 PCs were used) at a time.

We implemented TDSAB in two applications. The TDSAB framework was modified to use move generators for Awari and for Amazons.

- **Awari:** The evaluation function of BamBam, written by Jack van Rijswijck, was used.

- **Amazons:** The evaluation function of Antiope,written by Theodore Tegos, was incorporated into our implementation.

Information on these games, as well as on BamBam and Antiope, can be found at `http://www.cs.ualberta.ca/~games/`. TDSAB could have been implemented in chess and checkers instead of Awari and Amazons. However, we chose these games, because they are simpler to implement and are of more research interest currently than chess and checkers.

Both applications are written in C. PVM [18] (its version is 3.4) is used for the message passing between processors.

5.1.2 Test Sets and Experimental Conditions

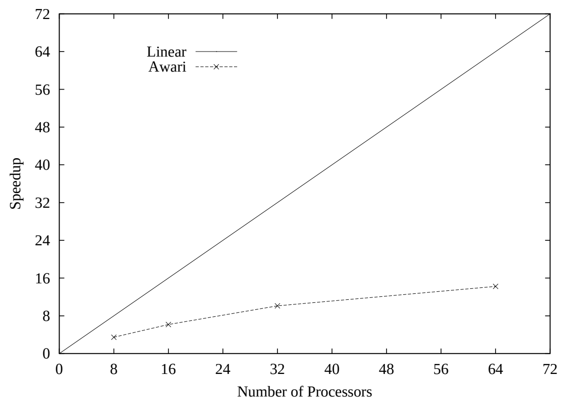
Care is needed in selecting a test suite for benchmarking parallel performance. The test suite should be representative of the conditions that are present during a competitive game. Larger sizes of game-trees improve the speedups. This is well-known result and is one way to create misleading results. Any $\alpha\beta$ speedup must be interpreted in the context of the size of the search performed. The sizes should be limited, because game-playing programs must select a move within a limited amount of time. In our experiments, the test suites are chosen to enable parallel searches to finish within 2-3 minutes on average, when using the largest number of processors in an environment. The time constraint is 180 seconds on 64 processors. We prepared 20 test positions selected from games played by BamBam against itself in Awari, and 20 positions through the self-play of Yasushi Tanase's Amazons program. The positions selected for our test suite are between 10 and 20 moves (half moves in chess terminology), from the initial position in Awari, and between 25 and 35 moves in Amazons. We ran all the test positions to the same fixed search depth, which is set to 24 in Awari and 5 in Amazons. At present, no search extensions, such as singular extension [2] and quiescence search [5], are implemented in the sequential counterparts. This prevents the parallel searches from returning different minimax scores from those of the sequential searches. For example, a varying search order of nodes could cause different search extensions to be turned on, thereby causing different minimax scores. Having no search extensions allowed the applications to return the identical minimax scores in most of the test positions (97.5% in Awari and 100% in Amazons), measuring meaningful observed speedups.[1]

The performance of game-playing programs is sensitive to the size of the transposition table used. Our experiments used 200 MB memory for transposition tables for both sequential applications. Some researchers measure the experiments under the condition that the parallel test uses exactly the same amount of memory for transposition tables as does the sequential test. Others use additional memory, with an increase in the number of processors. The first methodology measures the scalability of the number of processors, rather than the scalability of memory [8], while the second measures the practical performance on distributed-memory systems. Because both approaches must be considered, we measured the results under the conditions that the total size of memory used for the transposition tables of parallel programs is 200 MB, and that each processor used 200 MB memory for its transposition table. Throughout this chapter we call the former the *constant* size memory (or *constant*), and the latter the *increased* size memory (or *increased*).

5.2 Results

This section describes the main results for both applications. We will start with Awari, then show the result in Amazons and, finally, compare our results to those reported by other researchers.

[1]In Awari the minimax score of a parallel search sometimes does not match that of a sequential search, because a transposition table entry for a node that has been searched to *at least* the required depth immediately returns a score. There may be a difference of the search order between the sequential and parallel searches, returning a different score. However, the ratio of returning the different score is small (97.5% of the test positions returned identical scores in our experiment).



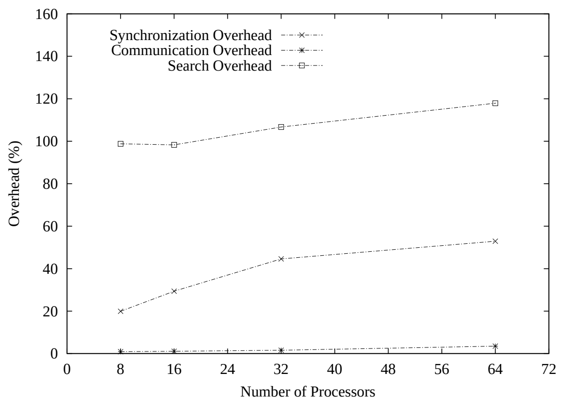


Figure 5.1: Speedups and Overheads (Awari, Constant)

Number of Processors	Execution Time (seconds)	Speedup	Search Overhead (%)	Synch. Overhead (%)	Comm. Overhead (%)
1	2177.2	-	-	-	-
8	628.7	3.46	98.8	19.9	0.9
16	363.4	6.16	98.3	29.4	1.1
32	215.4	10.11	106.7	44.6	1.6
64	152.9	14.24	117.9	52.9	3.5

Table 5.1: Performance and Overheads (Awari, Constant)

5.2.1 Awari

Figure 5.1 illustrates the graphs of the average speedups and overheads with 20 positions in Appendix C.1, when using the constant size memory. Our sequential Awari program expands 300,000 nodes per second. Table 5.1 gives the corresponding performance data. Execution time is computed by dividing the total execution time by the number of positions experimented with (i.e., 20). Synchronization overhead is measured as the ratio of the average idle time of a processor to the average execution time. Communication overhead is the ratio of the average amount of time which a processor spends in exchanging data with other processors, to the average execution time. Search overhead is measured by the definition given in Chapter 3.

To measure synchronization and communication overheads, we used different programs, which have additional operations to those used to measure speedups and search overhead. Therefore, we note that the theoretical speedups calculated by these overheads do not always reflect the observed speedups in each game.

Figure 5.1 shows that scalability goes down as the number of processors increases. The best speedup achieved with the constant size memory is 14.24 on 64 processors, for a speedup efficiency of 0.22. This result looks disappointing, however, game-tree search is notoriously hard to parallelize. In particular, it is difficult to parallelize Awari because of its small branching factor: there often is not enough parallel work to keep 64 processors busy.

Communication overhead is a minor factor in the performance of TDSAB. Although this overhead increases slightly as we use more processors, it is still relatively small. Transposition table entries are accessed locally by the processors in TDSAB. Therefore, processors communicate only when they distribute nodes, exchange scores, and broadcast signatures asynchronously, resulting in the small communication overhead.

Search and synchronization overheads hurt the performance of TDSAB. The synchronization overhead grows almost linearly with the number of processors, which becomes 52.9 % with 64 processors. The large synchronization overhead limits the speedup. For example, if 64 processors spend half of the execution time being idle, the speedup is theoretically at most 32-fold. This is not surprising, given that there are 12 iterations (the program iterates in steps of two ply at a time), each of which contains an average of 34 synchronization points at the root in the experiments. Moreover, because we parallelized MTD(f), it clearly has more synchronization points than are obtained by parallelizing other $\alpha\beta$ variants such as PVS and NegaScout.

In order to analyze the synchronization overhead, we implemented a tool to monitor the idle time of each processor. Figure 5.2 shows a graph of processor idle time (white space) for Position 1 in Appendix C.1. The Y-axis is the processor number (0-31) and the X-axis is time spent. The vertical lines show where a synchronization point occurred. The blue vertical line (darker line for grey-scale print) shows a fail low at the root, while the green line (lighter line) shows a fail high. Clearly, the last few synchronization points resulted in a large amount of idle time, limiting the speedup.

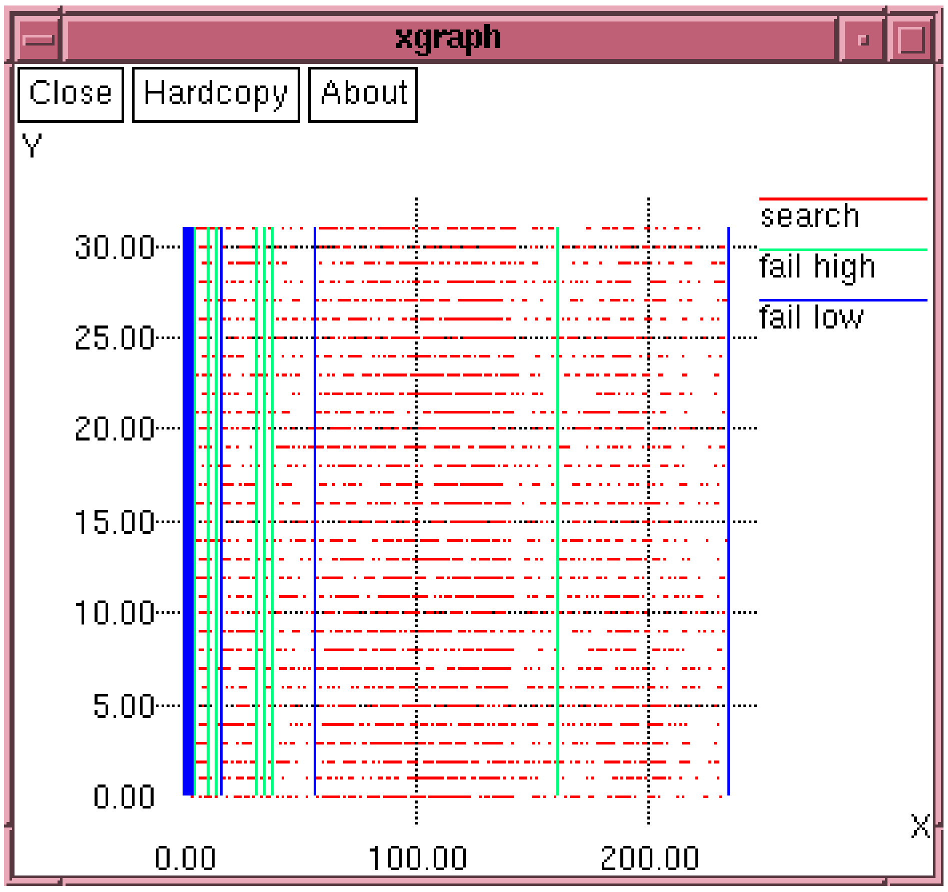


Figure 5.2: Idle Time (Awari, Constant)

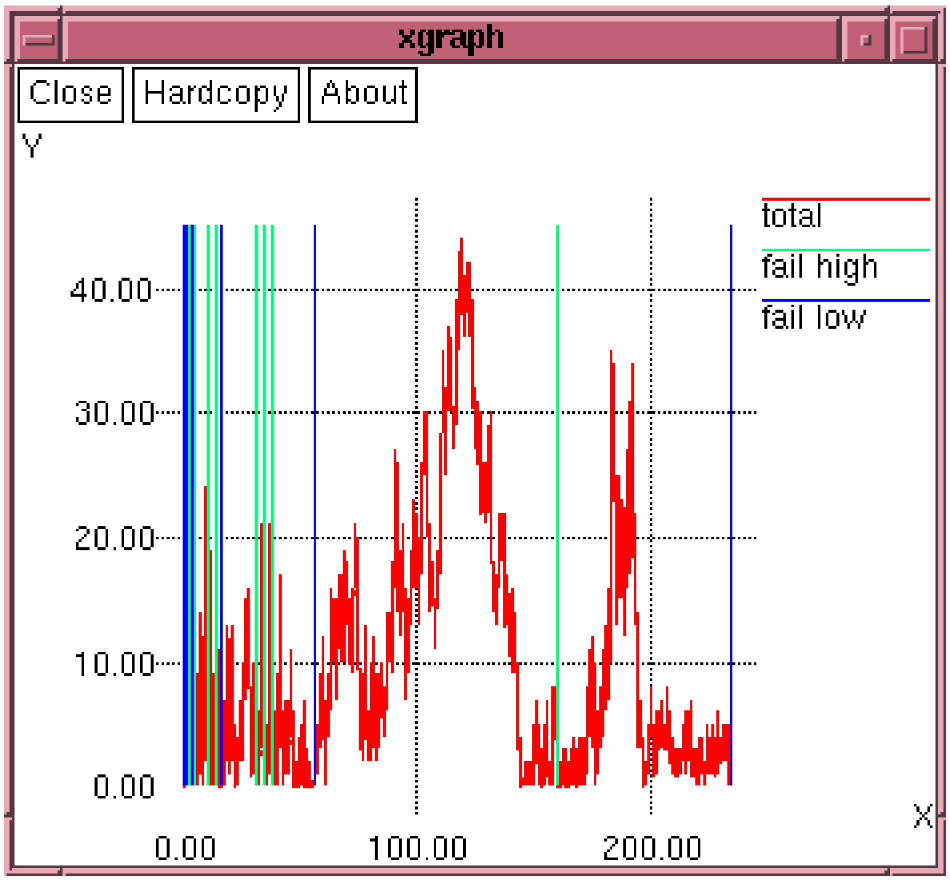


Figure 5.3: Priority Queue (Awari, Constant)

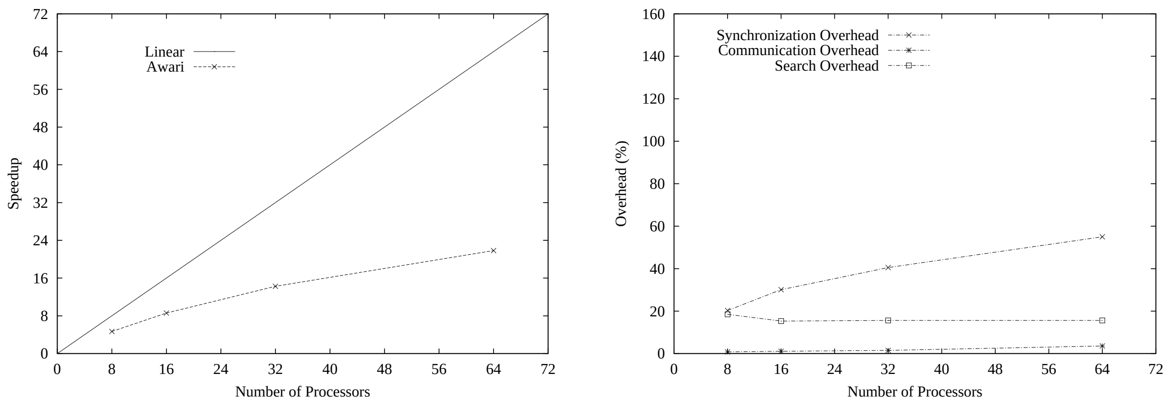


Figure 5.4: Speedups and Overheads (Awari, Increased)

Number of Processors	Execution Time (seconds)	Speedup	Search Overhead (%)	Synch. Overhead (%)	Comm. Overhead (%)
1	2177.2	-	-	-	-
8	463.81	4.69	18.5	20.3	0.8
16	253.48	8.59	15.3	30.1	1.1
32	152.67	14.26	15.6	40.5	1.5
64	99.80	21.82	15.6	55.0	3.6

Table 5.2: Performance and Overheads (Awari, Increased)

Figure 5.3 illustrates the number of nodes enqueued in the priority queue. The X-axis represents time spent, while the Y-axis indicates the number of nodes contained in the queue of a typical processor. The vertical lines show where a synchronization point occurred, as defined in Figure 5.2. This graph indicates that the processor is running out of work at the beginning and the end of iterations.

The search overhead starts at 98.8 % and increases gradually with an increase in the number of processors; it becomes 117.9 % with 64 processors. Clearly, TDSAB suffers from a large search overhead.

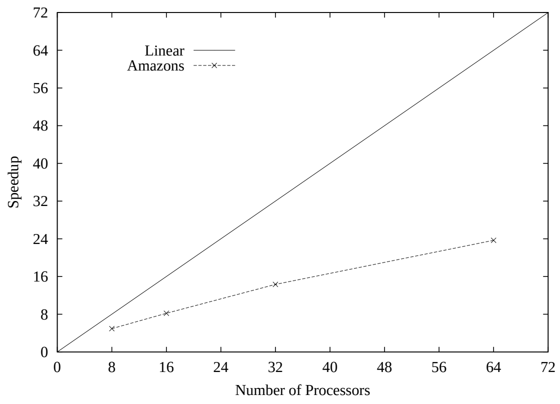
Our hypothesis to explain a large search overhead when using the constant size memory is that the size of transposition tables for the search is too small; valuable information is overwritten - potentially serious issue, because parallel search builds larger trees than does sequential search.

To test this hypothesis, experiments were done with an increased amount of memory Figure 5.4 shows the resulting speedups and overheads, and Table 5.2 shows the corresponding numbers of these graphs. The result proves our hypothesis of a large increase of search overhead with the constant size memory. Compared to the constant size memory, additional memory improves the speedup (21.82-fold on 64 processors). Increasing memory reduces the search overhead by a large margin (117.9 % versus 15.6 %), although it still has considerable synchronization overhead. This result demonstrates that one of the properties of TDSAB - that the transposition table scales well with more memory - is an important advantage that can dramatically influence the speedup.

5.2.2 Amazons

Figure 5.5 and Table 5.3 show the performance and overheads in Amazons using the constant size memory. Amazons has superior performance (23.68-fold speedup) to Awari, using the constant size memory, because Amazons has a very large branching factor (and, hence, no shortage of work to be done). The large branching factor, however, turns out to be a liability. At nodes where parallelism can be initiated, many pieces of work are generated, creating a lot of concurrent activity (which is desirable). If a cut-off occurs, many of these pieces of work may prove to have been unnecessary, resulting in increased search overhead (which is undesirable). In this case, search overhead limits the performance, suggesting that the program should be more prudent than it currently is in initiating parallel work. Other parallel implementations have adopted a similar policy of searching subsets of the possible moves at a node, precisely to limit the impact of unexpected cut-offs (for example, [53]).

The synchronization overhead in Amazons is smaller than in Awari (37.2 % versus 52.9 %). Figure 5.6 illustrates a graph of processor idle time for Position 1 in Appendix C.2 with 32 processors. This clearly illustrates there are fewer synchronization points compared to the number obtained in Awari (See Figure 5.2). Amazons' 3 iterations (the Amazons program also iterates in steps of two ply at a time) contain 13 synchronization points on average at the root in the experiments, while Awari has 34 synchronization points.



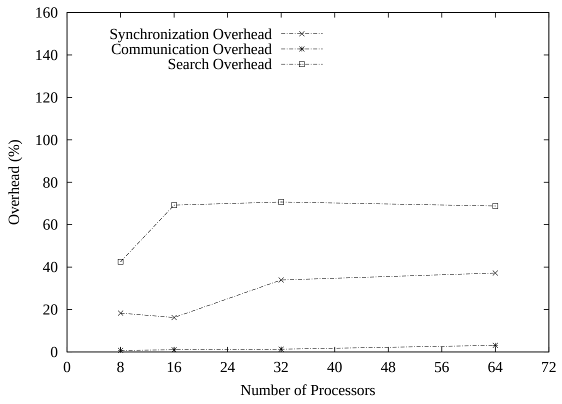


Figure 5.5: Speedups and Overheads (Amazons, Constant)

Number of Processors	Execution Time (seconds)	Speedup	Search Overhead (%)	Synch. Overhead (%)	Comm. Overhead (%)
1	1604.1	-	-	-	-
8	324.6	4.94	42.5	18.3	0.7
16	195.6	8.20	69.2	16.2	1.0
32	112.0	14.32	70.7	33.9	1.3
64	67.7	23.68	68.8	37.2	3.1

Table 5.3: Performance and Overheads (Amazons, Constant)

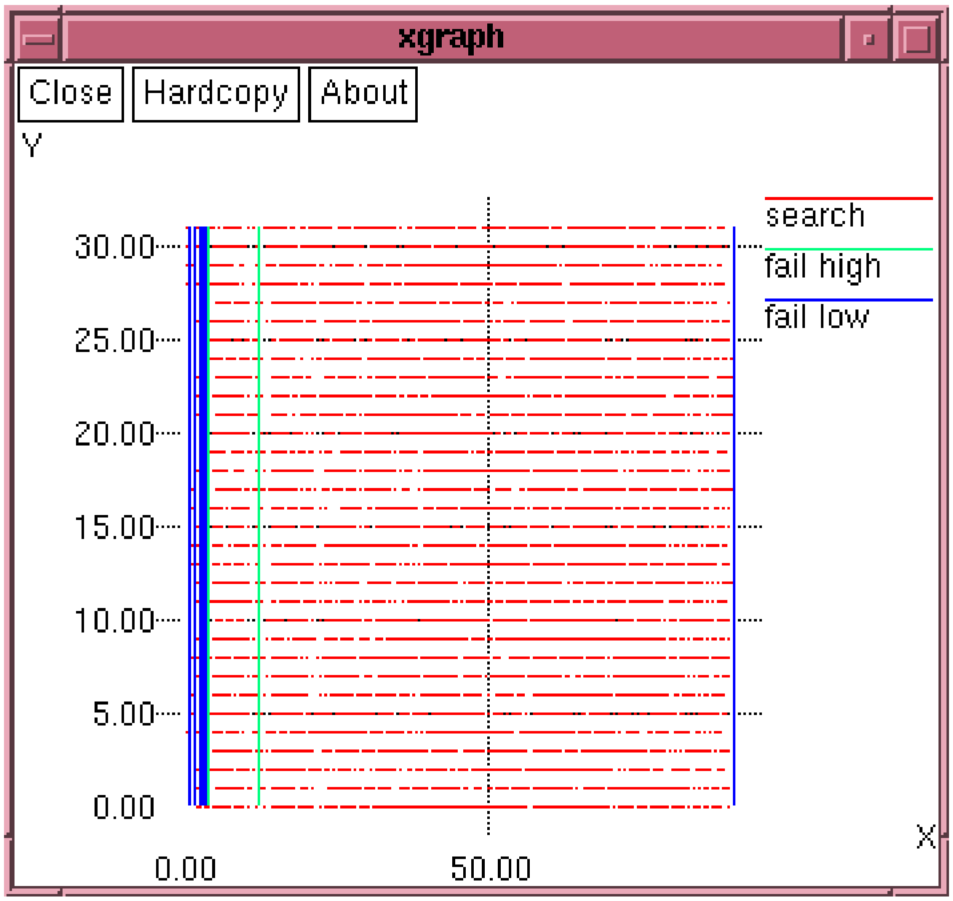


Figure 5.6: Idle Time (Amazons, Constant)

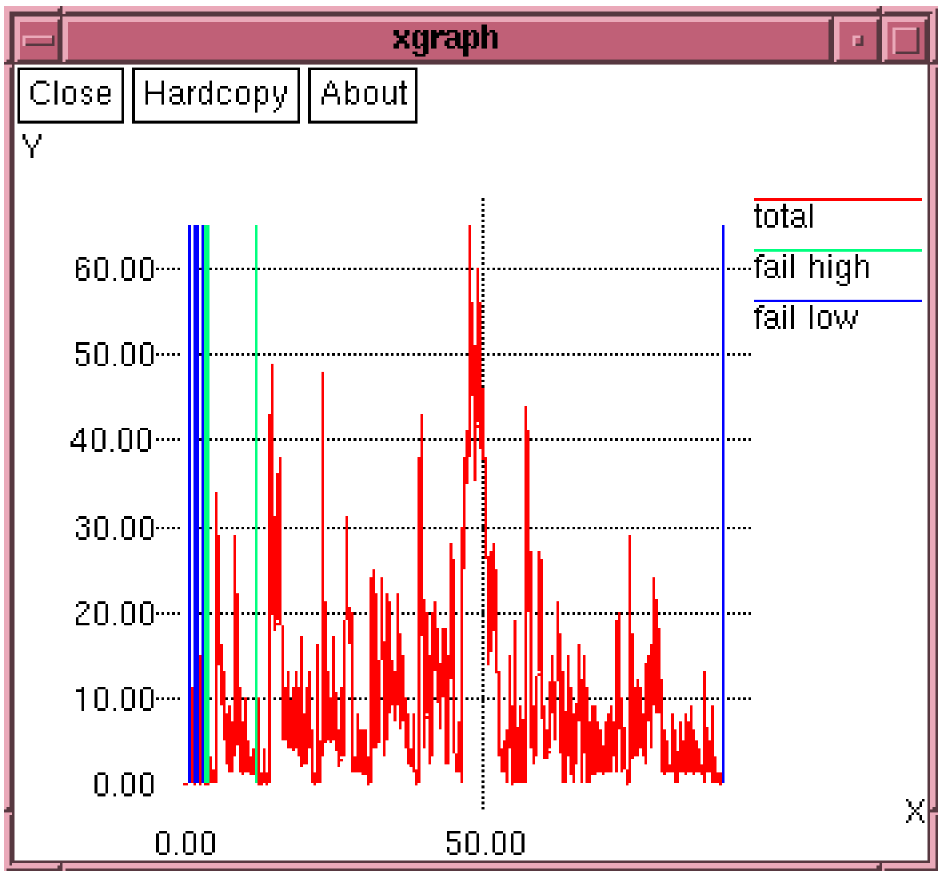
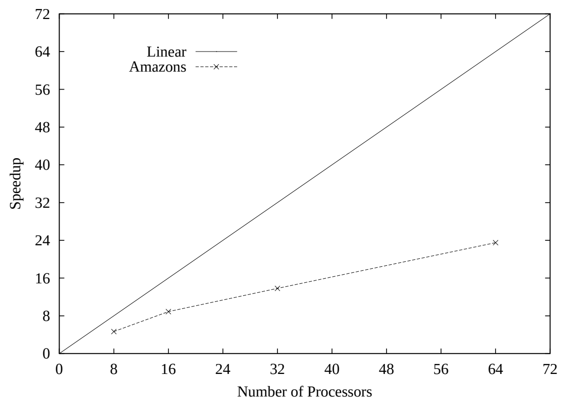


Figure 5.7: Priority Queue (Amazons, Constant)



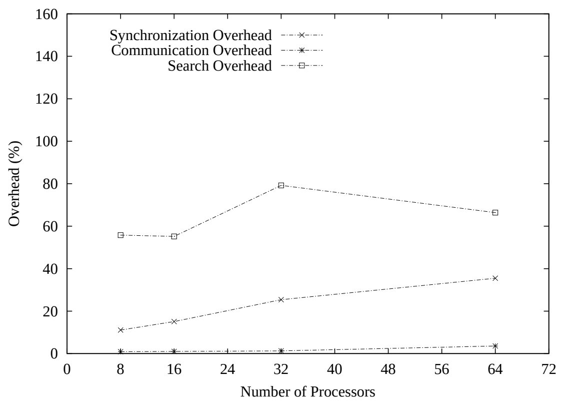


Figure 5.8: Speedups and Overheads (Amazons, Increased)

Number of Processors	Execution Time (seconds)	Speedup	Search Overhead (%)	Synch. Overhead (%)	Comm. Overhead (%)
1	1604.1	-	-	-	-
8	343.58	4.67	55.8	11.1	0.9
16	180.11	8.90	55.2	15.1	1.0
32	116.11	13.81	79.2	25.4	1.3
64	68.25	23.50	66.4	35.5	3.6

Table 5.4: Performance and Overheads (Amazons, Increased)

Figure 5.7 is the profile of the number of nodes in the priority queue of a typical processor, measured under the same conditions as in Figure 5.6. We note that the size of the queues is kept small because of Amazons' enhancement of initiating only some of the children in parallel at a time. The number of nodes allocated to the processor increases and decreases suddenly, continuing this phenomenon until the search ends. This demonstrates that a larger number of nodes (20 at CUT nodes, and 50 at other nodes) are immediately searched in parallel when a fail low happens at a node, and deleted when a cut-off occurs, because of the larger branching factor of Amazons. The result is less synchronization overhead but more search overhead in Amazons than in Awari.

Figure 5.8 and Table 5.4 show the performance and overheads with the increased size memory. Interestingly the performance in Amazons does not improve at all when the memory size is increased (23.68-fold with constant versus

23.50-fold with increased). Unlike the results in Awari, using additional memory does not reduce the search overhead. Our Amazons implementation has a much more expensive evaluation function than that of Awari (300,000 nodes per second in Awari versus 20,000 in Amazons). Therefore, our Amazons implementation expends more time in using up all the transposition table entries. Using 200 MB transposition tables in total is sufficient for both sequential and parallel search algorithms.

5.2.3 Comparison to Other Experiments

At present, TDSAB achieves only comparable speedups to those reported by other researchers, using the conventional parallel $\alpha\beta$ algorithms. The advantages of TDSAB are partially offset by synchronization overhead, in Awari and by search overhead in Amazons. However, it is difficult to make a fair comparison to the results reported by other researchers, because the speedups depend on factors such as the hardware configurations, sequential counterparts to parallelize, implementations, experimental conditions, and games.

In a sense, implementing a new parallel idea using MTD(f) was not useful. When comparing results with previous work, it is clear that two parameters have changed: the sequential algorithm (MTD(f) versus $\alpha\beta$) and the parallel algorithm (TDS versus some variants of YBWC). Thus, it is not obvious which component is responsible for any parallel inefficiencies. As the synchronization overhead for Awari demonstrates, MTD(f) has more synchronization points at the root and, hence, more synchronization overhead.

The Awari results can be compared to previous work using checkers, because checkers also has a small branching factor. For example, APHID achieves a speedup of 14.35-fold with the constant size memory, using an Origin 2000 with 64 processors [8]. This result is similar to TDSAB's speedup (14.24-fold) in Awari using the constant size memory. However, APHID's result is obtained using a different sequential $\alpha\beta$ (PVS versus MTD(f)), and a different environment (distributed shared-memory versus distributed-memory).

Multigame is the only previous attempt to parallelize MTD(f) using the increased size memory [41]. Multigame's performance at checkers (21.54-fold speedup) is comparable to TDSAB's result in Awari. In chess, Multigame achieved a 28.42-fold speedup using partitioned transposition tables - better than TDSAB's results in Amazons. However, this is not a fair comparison. The Multigame results were obtained using slower machines (Pentium Pros at 200 Mhz versus Pentium IIIs at 933 Mhz), a faster network (Myrinet 1.2 Gb/s duplex network versus 100 Mb/s Ethernet), longer execution times (roughly 33% larger), and different games. Chess and checkers could have been used for our TDSAB implementations, allowing for a fairer comparison between our work and the existing literature. However, chess and checkers no longer seem to be of interest to the research community, while both Awari and Amazons are the subject of active research efforts, indicating greater interest in these games, currently.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this thesis we tried to answer the question raised in Romein's thesis [41] concerning the application of TDS to two-player games: "Since this kind of (minimax) algorithm requires propagation of search results upward in the tree, it is not yet clear whether TDS will be successful for this class of algorithms [41]."

In Chapter 4, TDSAB was developed as a combination of TDS with MTD(f). Based on TDS, TDSAB reversed the relationship of the computation and the data. While the traditional approach sends data to the work that needs it, TDSAB sends work to where the data is located. Two important new techniques (priority queues and signatures) were proposed to overcome the more complex framework of $\alpha\beta$ than that found in a single-agent search algorithm (IDA*). The synchronization strategy of identical nodes in TDSAB avoids deadlock when searching a cyclic graph.

In Chapter 5, the results of the first implementations of TDSAB in Awari and Amazons were reported for a network of workstations with 64 processors. The best speedups in these two application domains were roughly 23, which are comparable to what others have achieved using conventional parallel algorithms.

Our answer to Romein's question is both *yes* and *no.* Clearly, the TDS framework offers important advantages for a high-performance search application, including asynchronous communication and effective use of memory.

However, these advantages are partially offset by the increased synchronization overhead of MTD(f). The end result of this work is to achieve speedups that are comparable to what others have obtained. This is disappointing since, given the obvious advantages of transposition table driven scheduling, better results might have been expected. On the other hand, this is the first attempt to apply TDS to the two-player domain, and TDSAB opens up new opportunities for further performance improvements.

6.2 Future Work

Although a lot of effort has been put into TDSAB, there are a number of things to be done in the future.

- **Speculative Search**
  Although TDSAB achieved satisfactory speedups on a network of workstations, the results were not impressive. In order to achieve better performance in Awari, we need to reduce the number of synchronization points. At present, in an attempt to solve the starvation problem, our Awari implementation searches the children of the root in parallel. This is an *ad hoc* approach that does not generalize to old application domains. We need a general way to allow a speculative search only when the processors are starved for work - while not increasing the search overhead.

- **TDS Implementation of $\alpha\beta$**
  Because MTD(f) clearly has more synchronization points at the root than PVS (or NegaScout), it might not be the best algorithm to parallelize. Parallel PVS based on TDS could possibly achieve a better speedup by reducing the synchronization points. In order to parallelize PVS we need to deal with the case of receiving identical nodes with different ranges of search windows, when searching a DAG or cyclic graph.

- **Better Priority Queue Ordering**
  Our current scheme for priority queues is not as "left-sided" as would

be the case for a sequential search - which will increase search overhead. Better priority queue ordering will reduce the search overhead and, hence, improve the speedup.

- **Initiating Parallelism**

  Our Amazons implementation suffered from large search overhead, which decreased performance. When a fail low score returned to a node, the Amazons implementation allowed only a (small) constant number of branches to be searched in parallel. This constant was decided by human programmers. We might need a dynamic method to control the amount of parallelism initiated at a node, in order to reduce the search overhead.

- **Assistance for Programmers**

  As we explained in Section 4.3, it is a difficult task to implement TDSAB from scratch. Therefore, it would be highly desirable to be able to provide other programmers with useful tools for developing TDSAB (e.g., debugging aids).

- **Applicability of TDS to AND/OR Tree Search**

  Recently AND/OR tree search algorithms based on proof and disproof numbers have achieved remarkable results in solving endgames [1, 31, 32]. TDS ideas can be used for parallelizing these algorithms, however, this creates a serious problem. When a node is expanded, the sequential algorithms using proof and disproof numbers look up the table entries of the node's children. In TDS, the processor expanding a node does not always keep the table entries of its children. Therefore, we need further research to resolve this issue.

Bibliography

[1] L. V. Allis, M. van der Meulen, and H. J. van den Herik. Proof-Number Search. *Artificial Intelligence*, 66:91–124, 1994.

[2] T. S. Anantharaman, M. Campbell, and F.-h. Hsu. Singular Extensions: Adding Selectivity to Brute-Force Searching. *Artificial Intelligence*, 43(1):99–109, 1990.

[3] T. E. Anderson, D. E. Culler, D. A. Patterson, and the NOW Team. A Case for Networks of Workstations: NOW. *IEEE Micro*, 15(1):54–64, 1995.

[4] R. Barton, D. Beal, D. Dailey, M. Frigo, C. Joerg, C. Leiserson, P. Lisiecki, R. Porter, H. Prokop, and D. Venturini. Cilkchess homepage: `http://supertech.lcs.mit.edu/chess/`.

[5] D. F. Beal. A Generalised Quiescence Search Algorithm. *Artificial Intelligence*, 43(1):85–98, 1990.

[6] D. M. Breuker, H. J. van den Herik, J. W. H. M. Uiterwijk, and L. V. Allis. A Solution to the GHI Problem for Best-First Search. In *Computers and Games (CG'98)*, volume 1558 of *Lecture Notes in Computer Science*, pages 25–49. Springer-Verlag, 1998.

[7] M. G. Brockington. A Taxonomy of Parallel Game-Tree Search Algorithms. *International Computer Chess Association Journal*, 19(3):162–174, 1996.

[8] M. G. Brockington. *Asynchronous Parallel Game-Tree Search.* PhD thesis, Department of Computing Science, University of Alberta, 1998.

[9] M. G. Brockington and J. Schaeffer. APHID Game-Tree Search. In H. J. van den Herik and J. W. H. M. Uiterwijk, editors, *Advances in Computer Chess 8*, pages 69–91. Universiteit Maastricht, 1997.

[10] M. G. Brockington and J. Schaeffer. APHID: Asynchronous Parallel Game-Tree Search. *Journal of Parallel and Distributed Computing*, 60:247–273, 2000.

[11] A. L. Brundo. Bounds and Valuations for Abridging the Search for Estimates. *Problems of Cybernetics*, 10:225–261, 1963. Translation of Russian original in Problemy Kibernetiki, 10:141-150, May 1963.

[12] M. Buro. The Othello Match of the Year: Takeshi Murakami vs. Logistello. *International Computer Chess Association Journal*, 20(3):189–193, 1997.

[13] M. Campbell. The Graph-History Interaction: On Ignoring Position History. In *1985 Association for Computing Machinery Annual Conference*, pages 278–280, 1985.

[14] M. S. Campbell, A. J. Hoane Jr., and F.-h. Hsu. Deep Blue. *Artificial Intelligence*, to appear in 2002.

[15] R. Feldmann. *Spielbaumsuche auf Massiv Parallelen Systemen.* PhD thesis, University of Paderborn, 1993. English translation titled *Game Trees on Massively Parallel Systems* is available.

[16] R. Feldmann, P. Mysliwietz, and B. Monien. Distributed Game Tree Search on a Massively Parallel System. In *Data Structures and Efficient Algorithms: Final Report on the DFG Special Joint Initiative*, volume 594 of *Lecture Notes in Computer Science*, pages 270–288, 1992.

[17] M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *ACM SIGPLAN Conferences on Programming Language Design and Implementation (PLDI'98)*, pages 212–223, 1998.

[18] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing.* The MIT Press, 1994. ISBN 0-262-57108-0.

[19] R. D. Greenblatt, D. E. Eastlake, and S. D. Crocker. The Greenblatt Chess Program. In *Proceedings of ACM Fall Joint Computing Conference*, volume 31, pages 801–810, San Francisco, 1967. Springer-Verlag.

[20] F.-h. Hsu. *Large Scale Parallelization of Alpha-Beta Search: An Algorithmic and Architectural Study with Computer Chess.* PhD thesis, Carnegie Mellon University, 1990. Also available as a technical report CMU-CS-90-108.

[21] R. M. Hyatt. The Dynamic Tree Splitting Parallel Search Algorithm. *International Computer Chess Association Journal*, 20(1):3–19, 1997.

[22] C. F. Joerg and B. C. Kuszmaul. Massively Parallel Chess. In *Third DIMACS Parallel Implementation Challenge*, 1994.

[23] D. E. Knuth and R. W. Moore. An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, 6:293–326, 1975.

[24] R. E. Korf. Depth-First Iterative Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence*, 27(1):97–109, 1985.

[25] B. C. Kuszmaul. *Synchronized MIMD Computing.* PhD thesis, Massachusetts Institute of Technology, 1994.

[26] C.-P. P. Lu. Parallel Search of Narrow Game Trees. Master's thesis, Department of Computing Science, University of Alberta, 1993.

[27] T. A. Marsland. Relative Performance of Alpha-Beta Implementations. In *Proceedings of International Joint Conferences on Artificial Intelligence (IJCAI'83)*, pages 763–766, Karlsruhe, Germany, 1983.

[28] T. A. Marsland and M. S. Campbell. Parallel Search of Strongly Ordered Game Trees. *ACM Computing Surveys*, 14(4):533–551, 1982.

[29] T. A. Marsland and F. Popowich. Parallel Game-Tree Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 7(4):442–452, 1985.

[30] J. McCarthy. Chess as the Drosophila of AI. In T. A. Marsland and J. Schaeffer, editors, *Computers, Chess, and Cognition*, pages 227–237. Springer-Verlag, 1990.

[31] A. Nagai and H. Imai. Proof for the Equivalence between Some Best-First Algorithms and Depth-First Algorithms for AND/OR Trees. In *KOREA-JAPAN Joint Workshop on Algorithms and Computation*, pages 163–170, 1999.

[32] A. Nagai and H. Imai. Application of df-pn Algorithm to a Program to Solve Tsume-Shogi Problems (in Japanese). In *IPSJ Algorithmic Colloquium AL-75-2*, pages 9–16, 2000.

[33] M. Newborn. Unsynchronized Iteratively Deepening Parallel Alpha-Beta Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 10(5):687–694, 1988.

[34] A. J. Palay. *Searching with Probabilities.* PhD thesis, Boston University, 1985.

[35] J. Pearl. Asymptotic Properties of Minimax Trees and Game-Searching Procedures. *Artificial Intelligence*, 14(2):113–138, 1980.

[36] A. Plaat. *Research Re:Search & Re-search.* PhD thesis, Erasmus University, 1996.

[37] A. Plaat, J. Schaeffer, W. Pijls, and A. de Bruin. SSS* $= \alpha - \beta +$ TT. Technical Report TR 94-17, Department of Computing Science, University of Alberta, 1994.

[38] A. Plaat, J. Schaeffer, W. Pijls, and A. de Bruin. Best-First Fixed-Depth Minimax Algorithms. *Artificial Intelligence*, 87:55–293, 1996.

[39] A. Reinefeld. An Improvement of the Scout Algorithm. *International Computer Chess Association Journal*, 6(4):4–14, 1983.

[40] A. Reinefeld and T. A. Marsland. Enhanced Iterative-Deepening Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 16(7):701–710, 1994.

[41] J. W. Romein. *Multigame - An Environment for Distributed Game-Tree Search.* PhD thesis, Vrije Universiteit Amsterdam, 2001.

[42] J. W. Romein and H. E. Bal. Wide-Area Transposition-Driven Scheduling. In *IEEE International Symposium on High Performance Distributed Computing*, San Francisco, 2001.

[43] J. W. Romein, A. Plaat, H. E. Bal, and J. Schaeffer. Transposition Table Driven Work Scheduling in Distributed Search. In *16th National Conference on Artificial Intelligence (AAAI'99)*, pages 725–731, Orlando, 1999.

[44] A. L. Samuel. Some Studies in Machine Learning. *IBM Journal of Research and Development*, 3(3):210–229, 1959.

[45] J. Schaeffer. Distributed Game-Tree Searching. *Journal of Parallel and Distributed Computing*, 6:90–114, 1989.

[46] J. Schaeffer. The History Heuristics and Alpha-Beta Search Enhancements in Practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 11(1):1203–1212, 1989.

[47] J. Schaeffer. *One Jump Ahead: Challenging Human Supremacy in Checkers*. Springer-Verlag, 1997.

[48] J. Schaeffer, R. Lake, C.-P. P. Lu, and M. Bryant. Chinook: The World Man-Machine Checkers Champion. *AI Magazine*, 17(1):21–29, 1996.

[49] J. Schaeffer and A. Plaat. Kasparov versus Deep Blue: The Re-match. *International Computer Chess Association Journal*, 20(2):95–101, 1997.

[50] D. J. Slate and L. R. Atkin. CHESS 4.5 - The Northwestern University Chess Program. In P. Frey, editor, *Chess Skill in Man and Machine*, pages 82–118. Springer-Verlag, 1977.

[51] G. C. Stockman. A Minimax Algorithm Better than Alpha-Beta? *Artificial Intelligence*, 12(2):179–196, 1979.

[52] K. Thompson. Computer Chess Strength. In M. Clarke, editor, *Advances in Computer Chess 3*, pages 55–56. Pergamon Press, 1982.

[53] J.-C. Weill. The ABDADA Distributed Minimax-Search Algorithm. *International Computer Chess Association Journal*, 19(1):3–16, 1996.

[54] A. L. Zobrist. A New Hashing Method with Applications for Game Playing. Technical report, Department of Computer Science, University of Wisconsin, Madison, 1970. Reprinted in *International Computer Chess Association Journal*, 13(2):169-173, 1990.

Appendix A

Proof that TDSAB is Deadlock-Free

A.1 Proof

We assert that TDSAB is free from deadlock, since the selection scheme explained in Chapter 4 cannot have a circular wait. In other words, if a node A waits for the score of a node B, B cannot wait for the score of A.

Here we prove that TDSAB is deadlock-free. The relation $(n_1, d_1) \xrightarrow{\text{w}} (n_2, d_2)$ means that a node n_1 for a d_1-ply search waits for n_2 for a d_2-ply search to finish. W maintains the relations of the synchronizations between two nodes. W is initialized to ϕ to indicate that no node is waiting for a score of some other nodes initially (See (#) in Figure A.1). We note that W is shared among the processors, and updated as soon as it is overwritten.

Figures A.1 and A.2 present the pseudo-code of TDSAB used for the proof. In this Figure, the data structure **node_t** contains an element for the search depth *depth*. The idea of the proof is to insert extra operations into the TDSAB code to maintain the search order of the nodes. (+) and (-) insert a relation $(n_1, d_1) \xrightarrow{\text{w}} (n_2, d_2)$ to W to indicate that searching n_1 will be done after accomplishing n_2. (+) appears in *GetLocalJob*. The function *ExistIdenticalNode* picks up a node n_2 from its local priority queue, which is identical to n_1 and whose depth is equal to or shallower than the depth for n_1. *ExistIdenticalNode* first tries to find a node n_2 which has already begun to be expanded. If there is no node already expanded, *ExistIdenticalNode* chooses the shallowest node (and not deeper than n_1) which is not yet searched. *GetLocalJob* first chooses the next node to be expanded, n_1, as explained in Section 4.1.2 (*FindCandidate*). A cyclic node checked by *IsCycle* returns zero immediately. Then, if there exists a node n_2 searched with the same or a shallower depth, searching n_1 is delayed until after the completion of the search of n_2, as described in Section 4.2.2. Therefore, a relation at (+) is added to W to check that n_1 is searched after n_2. We note that no relation is added in *GetLocalJob* if (a) there are no nodes identical to n_1, or (b) even if there exists an identical node whose depth is deeper than that of n_1. A relation is also added inside *ParallelNWS* (See (-) in *ParallelNWS*). Whenever a node n for a d-ply search expands its child n_c, searching n is completed after receiving the score of n_c. Therefore, a relation $(n, d) \xrightarrow{\text{w}} (n_c, d-1)$ is added to W.

From Figure A.1 and A.2, the following two lemmas are proven:

Lemma A.1.1 *Let n_1 and n_2 be nodes and d_1, d_2 be depths. Then, for any $(n_1, d_1) \xrightarrow{\text{w}} (n_2, d_2) \in W$ it holds that*

$$(n_1, d_1) \xrightarrow{\text{w}} (n_2, d_2) \Longrightarrow d_1 \geq d_2.$$

Proof. Straightforward from the pseudo code in Figure A.1 and A.2. (+) and (-) are the only cases in which relations are added to W. It holds $d_1 \geq d_2$ at (+), while $d_2 = d_1 - 1$ at (-).

■

Lemma A.1.2 *Let n_1, n_2 be nodes and d_1, d_2 be depths. Then, for any $(n_1, d_1) \xrightarrow{\text{w}} (n_2, d_2) \in W$ it holds that*

$$(n_1, d_1) \xrightarrow{\text{w}} (n_2, d_2) \wedge d_1 = d_2 \Longrightarrow n_1 \text{ and } n_2 \text{ are identical nodes.}$$

Proof. The only case where $(n_1, d_1) \xrightarrow{\text{w}} (n_2, d_1)$ is added to W is at (+) in *GetLocalJob*. Here n_1 is assumed to be identical to n_2.

■

Now the theorem for freedom from deadlock is proven by using the above two lemmas:

```
    /* $W$ maintains the relations of the synchronizations between two nodes.
       $W$ is shared among all the processors. */
(#) **relation_t** $W = \phi$;

    /* A search window is set to $(\gamma - 1, \gamma)$. */
    **int** $\gamma$;
    /* Granularity depends on machines, networks, and so on. */
    **const int** granularity;

    /* Find a node to expand next. */
    **void** GetLocalJob(**node_t** n) {
      **node_t** $n_1$, $n_2$;
      **do** {
        /* Select a node in the local priority queue. */
        FindCandidate(&$n_1$);
        **if** ($n_1$ == NOT_FOUND) {
          n = NOT_FOUND; **return**;
        }
        **if** (IsCycle($n_1$))  {
          /* Cyclic positions immediately return a score to its parent,
             then find another candidate to be searched. */
          SendNode($n_1$.parent,0);
          DeleteLocalJob($n_1$); **continue**;
        } **else if** (ExistIdenticalNode($n_1$,$n_2$)) {
          /* There exists a node $n_2$ which holds $n_1$.depth $\geq$ $n_2$.depth. */
(+)       $W+ = \{(n_1, n_1.\text{depth}) \xrightarrow{w} (n_2, n_2.\text{depth})\}$;
          **if** (!IsSearching($n_2$)) {
            n = $n_2$; **return**;
          } **else** {
            /* Find another node to be expanded. */
            **continue**;
          }
        } **else** {
          n = $n_1$; **return**;
        }
      } **while** (1);
    }
```

Figure A.1: Simplified Pseudo Code for TDSAB with W

```
    /* $W$ is declared in Figure A.1, which is initialized to $\phi$. */
(#) **extern relation_t** $W$;

    /* Null window search in parallel. */
    **void** ParallelNWS() {
      **int** type, value;
      **node_t** n;
      **signature_t** signature;
      **do** {
        ...
        /* Find a node traversed next. */
        GetLocalJob(&p);
        **if** (p $\neq$ NOT_FOUND) {
          **if** (p == terminal || p.depth $\leq$ granularity) {
            ...
          } **else** {  /* Do one-ply-search in parallel. */
            **for** (**int** i = 0; i < p.num_of_children; i++) {
(-)           $W+=\{(\mathrm{p}, \mathrm{p.depth}) \xrightarrow{\mathrm{w}} (\mathrm{p.child\_node[i]}, \mathrm{p.depth}-1)\}$;
              p,child_node[i].depth = p.depth - 1;
              SendNode(p.child_node[i],HomeProcessor(TTKey(p.child_node[i])));
              ...
            }
          }
        }
      } **while** (!FinishedSearchingRoot());
    }

    /* A minimax value is computed and stored in the transposition table. */
    **void** StoreSearchResult(**node_t** p, **int** value) {
      ...
    }
```

Figure A.2: Simplified Pseudo Code for TDSAB with W (cont.)

Theorem A.1.1 *(Freedom from Deadlock)*
Let $n_1, \cdots, n_k$ be nodes and $d_1, \cdots d_n$ be depths. W never holds the following relation:

$$(n_1, d_1) \xrightarrow{\text{w}} \cdots \xrightarrow{\text{w}} (n_k, d_k) \xrightarrow{\text{w}} (n_1, d_1).$$

In other words, the node selection scheme in TDSAB does not have a circular wait.

Proof. Assume that deadlock occurs in the selection scheme. Then, there exists relations in W such that:

$$(n_1, d_1) \xrightarrow{\text{w}} \cdots \xrightarrow{\text{w}} (n_k, d_k) \xrightarrow{\text{w}} (n_1, d_1).$$

- Case $d_k < d_1$:
  By Lemma A.1.1, $d_k \geq d_1$ holds. This contradicts $d_k < d_1$.
- Case $d_k > d_1$:
  By using Lemma A.1.1 repeatedly, it holds that $d_1 \geq d_2, d_2 \geq d_3, \cdots, d_{k-1} \geq d_k$. Therefore, $d_1 \geq d_k$, which contradicts $d_k > d_1$.
- Case $d_k = d_1$:
  Lemma A.1.1 deduces $d_1 \geq d_2 \geq \cdots d_k$. Because $d_k = d_1$, it follows that $d_1 = d_2 = \cdots = d_k$. By using Lemma A.1.2 repeatedly, we can prove that $n_1, \cdots, n_k$ are identical nodes. Without loss of generality, we can assume that n_1 is chosen last among these nodes. Before n_1 is chosen, the relation should be $(n_2, d_1) \xrightarrow{\text{w}} (n_3, d_1) \xrightarrow{\text{w}} \cdots \xrightarrow{\text{w}} (n_k, d_1)$, because there is no relation $(n_j, d_k) \xrightarrow{\text{w}} (n_l, d_k)$ in W if n_l is not expanded. Then, in *GetLocalJob*, (a) only the relation $(n_1, d_1) \xrightarrow{\text{w}} (n_i, d_1)(i \neq 1)$ is added when n_i is being searched, or (b) no relation is added to W if all the identical nodes have finished being searched. Clearly, because neither (a) nor (b) adds the relation $(n_k, d_1) \xrightarrow{\text{w}} (n_1, d_1)$, it does not have $(n_1, d_1) \xrightarrow{\text{w}} \cdots \xrightarrow{\text{w}} (n_k, d_k) \xrightarrow{\text{w}} (n_1, d_1)$.

Thus, it is proven that TDSAB is free from deadlock.

■

Appendix B

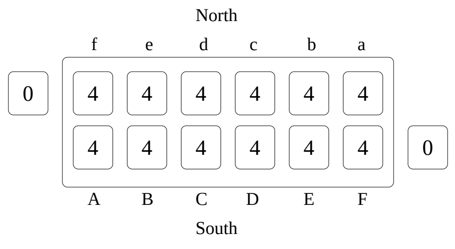
Brief Summary of Games

This Appendix is for those who are not familiar with the games of Awari and Amazons. We briefly explain the rules of these games.

B.1 Awari

Awari is an African board game played for more than 3500 years. Because there are several variations of the rules, we only explain the version implemented in the program we used for our experiments. More information can be found at `http://www.cs.ualberta.ca/~awari/rules.html`.

Awari has 48 *stones* and 12 *pits*. The goal of Awari is to capture as many stones as possible. Each pit contains stones, which are sowed by each player (*north* or *south* player) by turns. The north plays first. Figure B.1 illustrates the starting position. Four stones per pit are assigned at the initial position of the Awari board. The two boxes outside the board are used to keep the stones captured by the players. Because no stones are captured at first, no



North to Move

Figure B.1: Starting Position in Awari

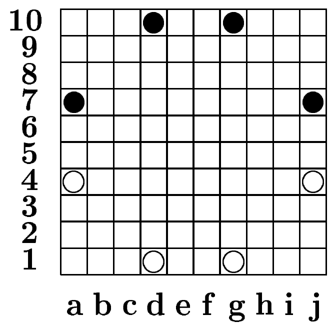


Figure B.2: Starting Position in Amazons

player has any stones. The rules of Awari are summarized as follows:

(1) On each turn, a player selects a non-empty pit on the side of that player. The player then sows the stones in that pit, dropping one stone at a time anti-clockwise into each pit. If a pit with 12 or more stones is chosen, the original pit is skipped and left empty.

(2) If the last stone lands in the opponent's pit, leaving 2 or 3 stones, these stones are captured. If the previous pit also contains 2 or 3 stones, they are captured as well. This continues with consecutive previous pits on the opponent's side.

(3) The player may not play a move that makes all the pits of the opponent empty.

(4) When a player captures more than 24 stones, it is a win for that player. If a position is repeated or both players take 24 stones, it is a draw.

B.2 Amazons

The game of Amazons was invented by Argentinian Walter Zamkauskas in 1988. It is played on a 10×10 board. Each player has 4 queens (called *amazons*), initially placed on the board as in Figure B.2. The rules are summarized in the following way:

(1) The players alternate playing moves, white moving first. A move consists of two mandatory parts. First, a player chooses an amazon to move. An amazon moves like a chess queen. After the amazon has moved, it shoots an *arrow* from the destination square to an empty square. An arrow is shot like an chess queen. The square which an arrow lands on is marked for the rest of the game.

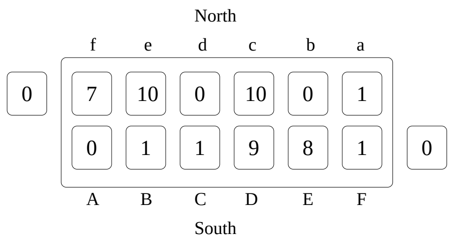
(2) Amazons and arrows cannot go over or onto marked squares.

(3) The last player who can make a move is a winner.

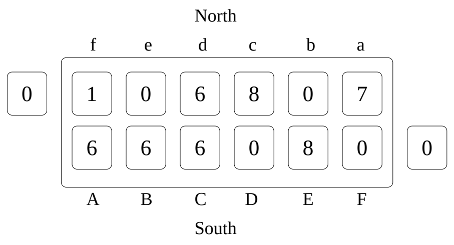
Appendix C

Test Positions

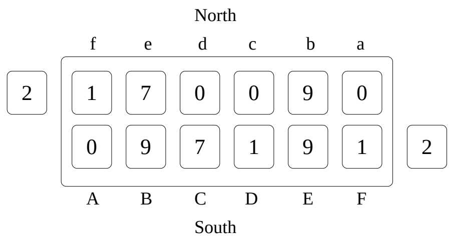
C.1 Awari



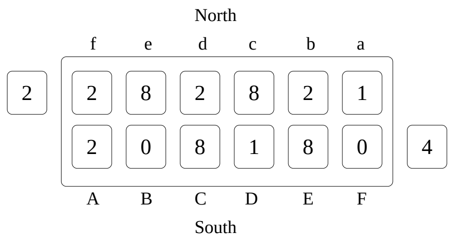
Position 1
North to Move



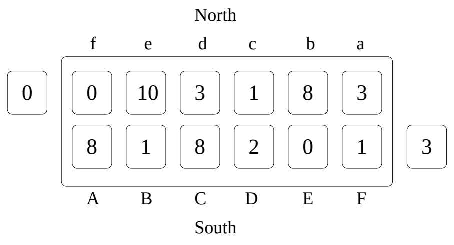
Position 2
North to Move



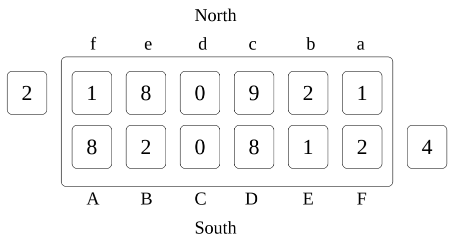
Position 3
North to Move



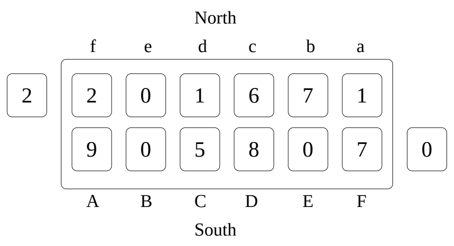
Position 4
North to Move



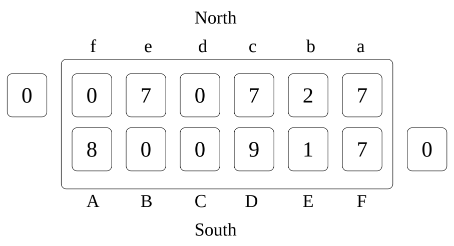
Position 5
North to Move



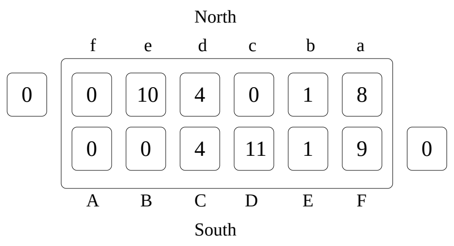
Position 6
North to Move



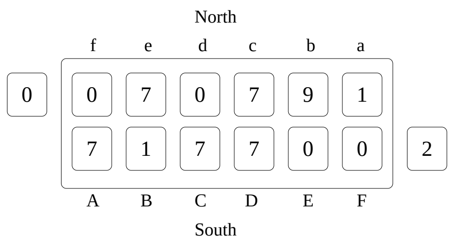
Position 7
South to Move



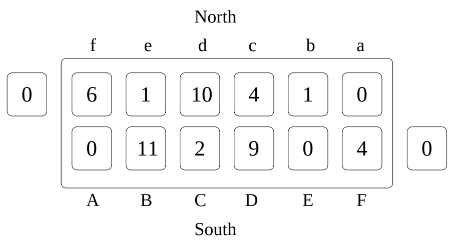
Position 8
North to Move



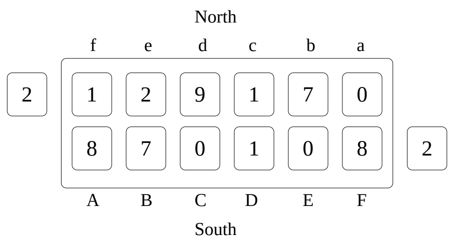
Position 9
North to Move



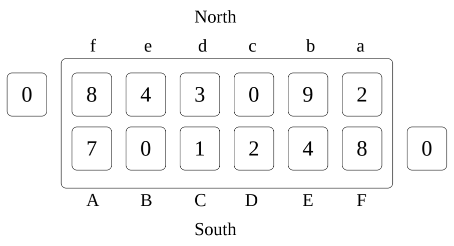
Position 10
South to Move



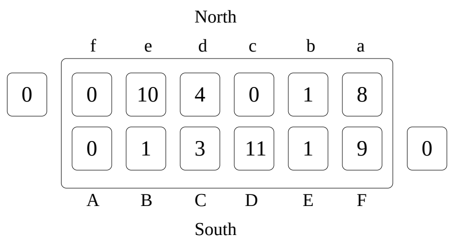
Position 11
North to Move



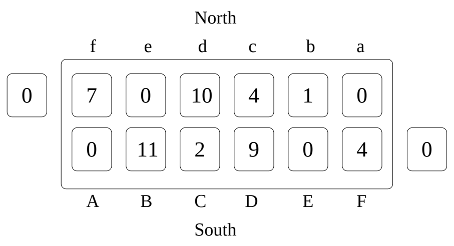
Position 12
North to Move



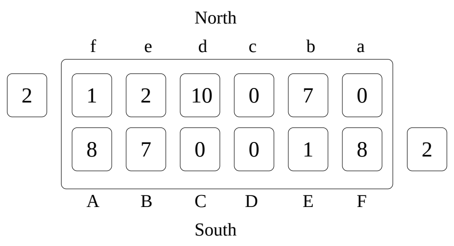
Position 13
North to Move



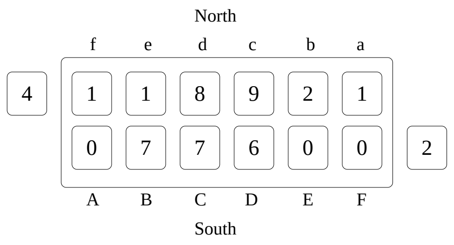
Position 14
South to Move



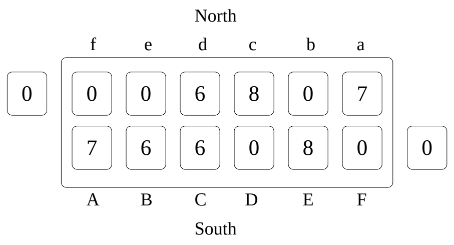
Position 15
South to Move



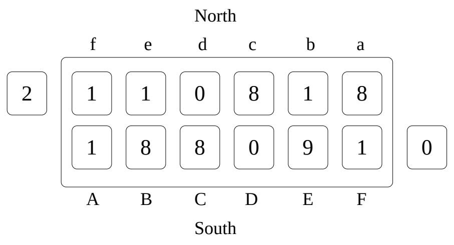
Position 16
North to Move



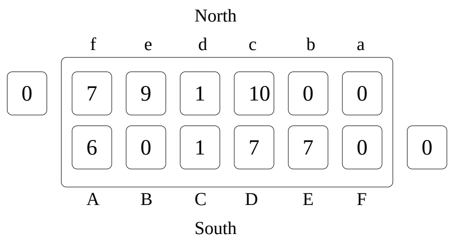
Position 17
North to Move



Position 18
South to Move

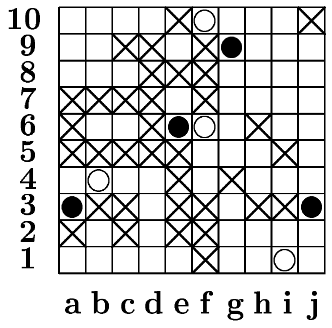


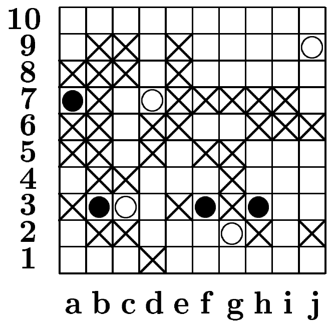
Position 19
South to Move

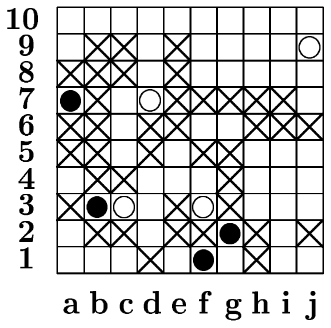


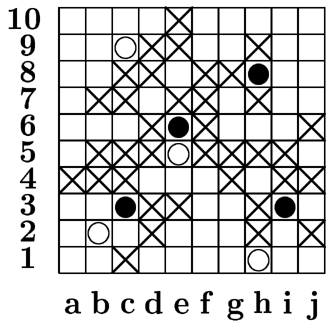
Position 20
South to Move

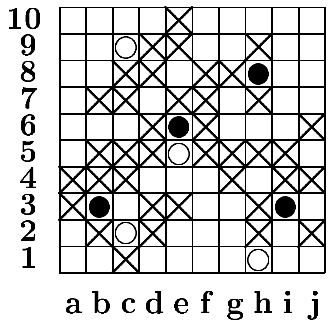
C.2 Amazons

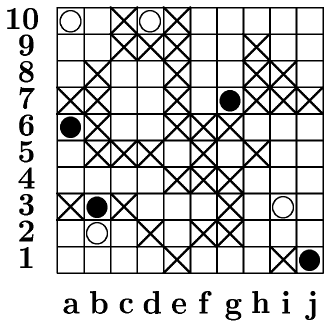

Position 1
White to Move

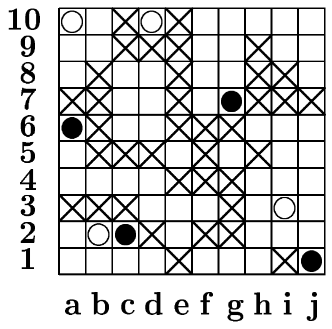

Position 2
Black to Move

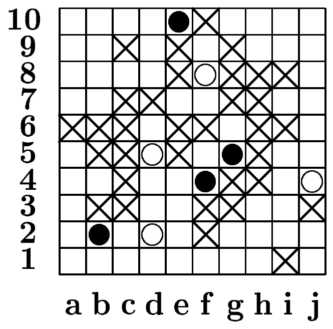

Position 3
White to Move

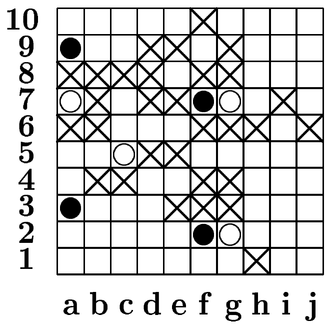

Position 4
Black to Move

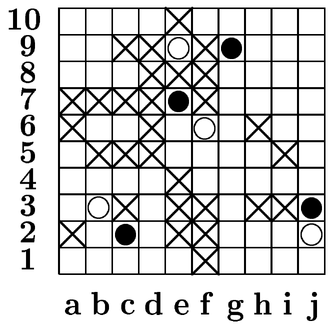

Position 5
Black to Move

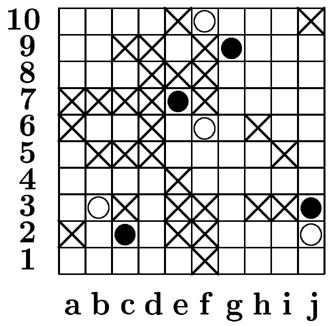

Position 6
Black to Move

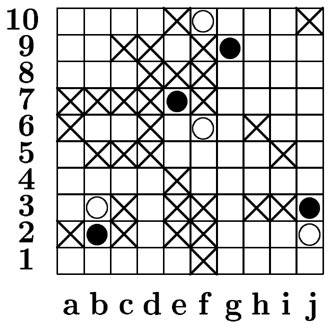

Position 7
White to Move

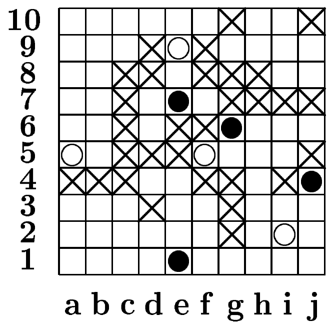

Position 8
White to Move

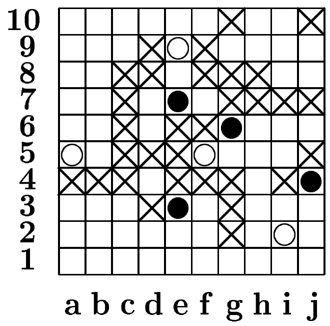

Position 9
Black to Move

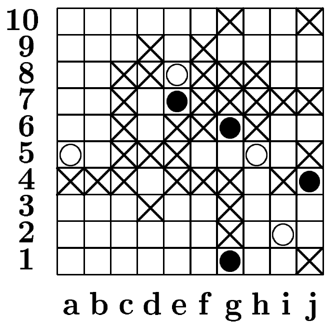

Position 10
White to Move

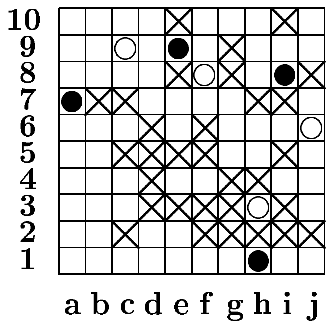

Position 11
Black to Move

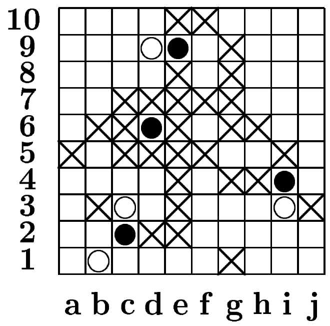

Position 12
White to Move

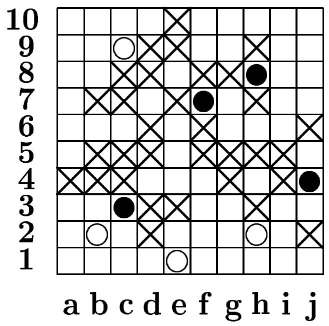

Position 13
Black to Move

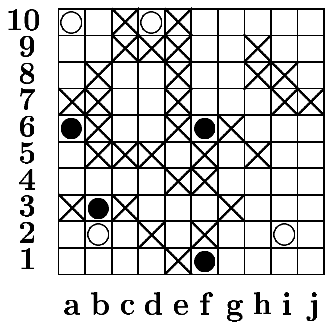

Position 14
White to Move

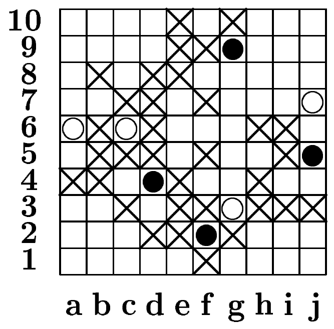

Position 15
Black to Move


Position 16
White to Move


Position 17
Black to Move


Position 18
Black to Move


Position 19
White to Move


Position 20
White to Move