Parallel Alpha-Beta Search on Shared Memory Multiprocessors

by

Valavan Manohararajah

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

Parallel Alpha-Beta Search on Shared Memory Multiprocessors

Valavan Manohararajah
Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto
2001

The alpha-beta algorithm is a well known method for the sequential search of game trees. Two methods, young brothers wait concept and dynamic tree splitting, have been used successfully in parallel game tree search. First, this work introduces the notion of an exponentially ordered game tree as a model for the game trees encountered in practice. Second, exponentially ordered trees are used in the study of the tree splitting methods used by young brothers wait concept and dynamic tree splitting. Finally, a new tree splitting method based on neural networks is introduced and is found to outperform the other two methods on certain types of trees.

Dedication

To my parents, Amma and Appa.

Acknowledgements

First, I would like to thank my supervisor, Professor Z. G. Vranesic, for his advice, guidance and support. His continual encouragement when I was faced with technical difficulties helped me push through and discover the right solutions. He was always willing to set aside any amount of time for a discussion on thesis matters.

I am grateful to both OGS and NSERC for providing the means to further my education.

Last, but certainly not least, I would like to thank my wife, Abiramy, for her love and support during the course of this thesis.

Contents

Chapter 1

Introduction

The seminal paper by Claude E. Shannon [22] introduced the concept of game trees along with a simple algorithm (MiniMax) for searching them. Shannon was primarily interested in creating a computer player for the game of chess. Here we consider the trees that arise from a particular class of games. Each game in this class has the following properties:

- There are two players involved in the game: player 1 and player 2.
- The two players take turns making moves. At any position in the game, a finite number of moves are available to the player on move.
- The game is deterministic — there are no elements of chance in the game.
- It is a game of *perfect information*. That is, both players know the entire state of the game at all times. For example, chess is a game of perfect information. For the duration of the game, both players know the board position. However, 2-player poker is not a game of perfect information. Although, player 1 can see his cards, he cannot see the cards that player 2 holds.
- There are three possible outcomes in the game: a win for player 1, a win for player 2 or a draw. Games that do not end in a draw are also included in the class. These games have two possible outcomes: a win for player 1 or a win for player 2.

The construction and the subsequent search of game trees forms the basis of many computer programs designed to play two player strategy games. A game tree is a way of representing the possibilities that are available to the players involved in the game. The search of the game tree yields the optimal sequence of play for both sides.

Efficient algorithms for the sequential search of game trees have been in existence for a long time (since 1963 [12]). In a tree where good moves are searched before bad ones, a good

sequential tree search algorithm will examine a fraction of the entire tree. On a parallel system, sequential algorithms are extended to allow several processors to share the search effort. Initial attempts at parallel search had limited success on a large number of processors [17] but more recent efforts have demonstrated that a large number of processors can be used effectively [5]. One of the biggest difficulties in parallel search is in determining where multiple processors can be used to split up the search effort. The main focus of this work is to explore alternative ways of choosing where to split the tree. A comparison of the tree splitting techniques in current use was not previously available. This work produces such a comparison using artificially generated trees. A new tree splitting technique is also introduced and its performance is compared to previously existing techniques.

Chapter 2 explores the different algorithms available for sequential tree search. Three approaches to parallel search are examined in chapter 3. Chapter 4 introduces the concept of artificially generated trees. A new node classification technique is introduced in chapter 5 and its performance is compared to existing techniques using a sequential tree searcher. Chapter 6 studies the performance of three tree splitting techniques within a parallel tree searcher.

Chapter 2

Game Tree Search

2.1 Introduction

A simple tree for a two-player game is presented in Figure 2.1. A node in the tree represents a position in the game while a branch represents a move available at a particular position. Player 1 is on move at nodes with the rectangular graphic and player 2 is on move at nodes with a circle graphic. For example, at the *Root* node, player 1 is on move and the player has two moves available: a and b. Each leaf node has been assigned a score that indicates how valuable that position is. A positive score indicates that player 1 is winning while a negative score indicates that player 2 is winning; a score of 0 indicates a draw. The magnitude of the score conveys important information as well. Higher the score, more favorable a position is for player 1. Similarly, lower the score, more favorable a position is for player 2. Note that this scoring scheme is arbitrary and there are several possible schemes that can be used as long as the scoring scheme allows one to distinguish which of the two players is winning.

While the above discussion involves a tree for a two-player game, a game tree can be constructed for a large class of problems that doesn't involve games. All that a game tree requires is that there be two opposing forces and that they occupy alternate levels in the tree.

The problem in game tree search is to find the *game tree value*. The value of a game tree is the score of the leaf node that is reached when both sides exercise their best options. From a practical viewpoint, what one really needs to find is the option at the root that leads to the game tree value. In the case of the tree in Figure 2.1, the best option is the move that gives player 1 the best chance of winning assuming that both players are playing perfectly. Keeping track of the path that leads to the game tree value is a trivial enhancement to an algorithm that determines this value. In the discussions that follow, for simplicity, it is assumed that the discovery of the game tree value is equivalent to the discovery of the path that leads to that value.

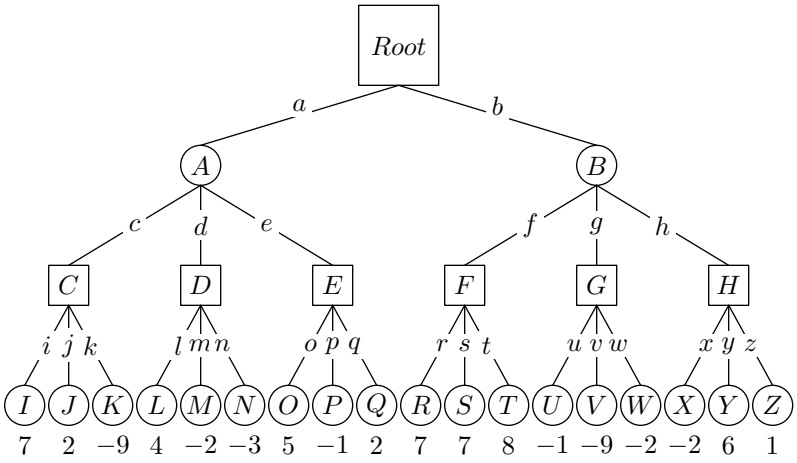

Figure 2.1: A simple game tree.

Consider the problem of finding the game tree value for the tree in Figure 2.1. Ideally, player 1 wants to guide the game towards position T because that has the highest score from his standpoint. Assume that player 1 plays move b in order to start making progress toward position T. As far as player 2 is concerned, move f plays right into player 1's hands. Player 2 obtains a better position by playing move g. If player 2 chooses move g then player 1 follows it up with move u and the final position, U, has a score of -1. Thus, player 1's original plan of guiding the game towards T can be easily thwarted by a careful player 2. Consider what happens when player 1 chooses move a. Now, if player 2 plays c then player 1 chooses i and we obtain a score of 7. However, if player 2 plays d then player 1 chooses l and the final position has a score of 4. A similar analysis on e shows that it leads to a score of 5. Clearly, player 2 should choose d. So the sequence of moves, assuming perfect play by both sides, is a, d, l. This sequence will be referred to as the *principal variation*. The term principal variation is used to describe the sequence of moves that lead to the game tree value. For the tree in Figure 2.1, the game tree value is 4.

2.2 Definitions

All algorithms described in this work expand the branches at a node in a left to right order. A branch or child node, m, is said to come before another branch or child node, n, if m is to the left of n. A branch or child node is said to be the first at a node if it is in the leftmost position.

In addition to the search order defined above, the algorithms search the tree in a depth first manner. Depth first techniques for game tree search require very little memory and the memory requirement does not grow exponentially with the tree size.

MiniMax(*node*)

```
1:  if node.depth = 0 then
2:     return Evaluate(node)
3:  if node.type = max then
4:     score ← −∞
5:  else
6:     score ← +∞
7:  for i ← 1 to node.branch.length
8:        new_node ← Traverse(node, node.branch[i])
9:        value ← MiniMax(new_node)
10:       if node.type = max then
11:          if value > score then
12:             score ← value
13:       else
14:          if value < score then
15:             score ← value
16: return score
```

Figure 2.2: The mini-max algorithm.

2.3 The Mini-Max Algorithm

In the tree of Figure 2.1, at the nodes where player 1 is on move, player 1 will select the move that maximizes his/her score. Similarly, the nodes where player 2 is on move, player 2 will select the move that minimizes his/her score. Thus, we can classify the nodes of a game tree as being one of two types: *maximizing* or *minimizing*. This observation leads directly to the mini-max algorithm in Figure 2.2.

Depending on whether a node is maximizing or minimizing, the algorithm keeps track of the largest or the smallest score, respectively. A leaf node is reached when the remaining depth (*node.depth*) is equal to zero. At a leaf node, the Evaluate function is called to determine the score associated with the node.

In performing its work, the mini-max algorithm explores every node in the game tree. Consider the application of this algorithm to chess. On average, a chess position has 32 possible moves (refer to Section 4.3.2). A tree of depth n would contain 32^n leaf nodes. Clearly, the mini-max algorithm is not practical for a chess tree when the depth exceeds 5.

NegaMax(*node*)

```
1: if node.depth = 0 then
2:    return EvaluateNegaMax(node)
3: score ← −∞
4: for i ← 1 to node.branch.length
5:      new_node ← Traverse(node, node.branch[i])
6:      value ← −NegaMax(new_node)
7:      if value > score then
8:         score ← value
9: return score
```

Figure 2.3: The nega-max algorithm.

2.4 The Nega-Max Formulation

The mini-max algorithm can be simplified by eliminating the distinction between maximizing and minimizing nodes. By simply negating the result returned from the recursive call in Figure 2.2, each node can be treated as a maximizing node. However, another modification is necessary to achieve the same result as the mini-max algorithm. At a leaf node, the Evaluate function has to return a score from the viewpoint of the player on move. For example, consider the case of a leaf node that has a score of -6 in the original mini-max scheme. The score indicates that the position favors player 2. In the new scheme, if player 1 is on move, a score of -6 would be returned. However, if player 2 is on move, then a score of 6 would be returned. The evaluation function that implements this functionality will be referred to as EvaluateNegaMax. Figure 2.3 illustrates the nega-max algorithm [12] that is obtained when one implements the changes described.

2.5 The Alpha-Beta Algorithm

A closer examination of the mini-max algorithm reveals possible enhancements to the basic technique. Table 2.1 illustrates the progress of the algorithm on the tree of Figure 2.1. It is assumed that the expansion at each node proceeds in a left to right order. We start at the *Root* node, which initially has a score of $-\infty$, and the exploration begins with branch *a*. Node *A* starts with a score of $+\infty$ since it is a minimizing node. The process of recursive calls continues until leaf node *I* is expanded. Here the recursion stops and a value of 7 is returned. At node

C, the old value, $-\infty$, is overwritten with the 7 that is returned by node I. Eventually, node A obtains a score of 4 at step 18. Search at node A proceeds with the traversal of branch e. Node E, being a maximizing node, has an initial score of $-\infty$. On exploring branch o, node E obtains a score of 5. This is where the enhancement can be made. Since node E is a maximizing node, the score can only go higher than 5. However, it is also known that at node A, a minimizing node, the score is 4. Node A will reject any value that is greater than or equal to 4. Thus, the unexplored branches rooted at node E can be eliminated from the search since they will have no effect on the score at node A. Such an elimination is termed a *cut-off* in game tree parlance.

There must be communication between the adjacent levels in the tree in order to determine when search at a node becomes no longer necessary. The enhanced version of the mini-max algorithm, which is referred to as the weak alpha-beta algorithm [12], is illustrated in Figure 2.4. The best score obtained at any node is passed down to the successor as a bound so that cut-offs can be made.

The weak alpha-beta algorithm still misses some cut-offs. Consider the application of the algorithm to the tree of arbitrary depth as illustrated in Figure 2.5. After node A has been explored, the *Root* will have a score of 4. Node B will receive 4 as a bound from the *Root*. Since no search has been done at node B, node C receives an infinite bound. Similarly, node E also receives an infinite bound from node C. However, the bound that was applied at node B still applies to node E since the *Root* will reject any score that is less than or equal to 4. In this particular case, additional cut-offs can be made during the search of the subtree rooted at E if the highest score obtained at a maximizing node is carried downwards as a bound. A similar argument can be made for the smallest score at minimizing nodes. Thus, the algorithm can be enhanced further by maintaining two bounds:

- *alpha (lower bound)*: Keeps track of the highest score obtained at a maximizing node higher up in the tree and is used to perform cut-offs at minimizing nodes.
- *beta (upper bound)*: Keeps track of the lowest score obtained at a minimizing node higher up in the tree and is used to perform cut-offs at maximizing nodes.

The resulting technique is referred to as the alpha-beta [12] algorithm and is summarized in Figure 2.6. To determine the game tree value, the algorithm is invoked with the call AlphaBeta(*Root*, $-\infty, +\infty$). The pair of numbers, $(-\infty, +\infty)$, defines the *search window*.

An illustration of the cut-offs achieved by the alpha-beta algorithm is shown in Figure 2.7. At node E, branches p and q are eliminated. When the search arrives at node B, there is a lower bound of 4. On exploring F, node B obtains a score of 8. A similar exploration of G yields -1. This exceeds the lower bound and the search is terminated at node B; node H is then cut-off.

Step	*Node*	*Score*	*Action*
1	*Root*	$-\infty$	Explore a
2	A	$+\infty$	Explore c
3	C	$-\infty$	Explore i
4	I	7	Return 7
5	C	7	Explore j
6	J	2	Return 2
7	C	7	Explore k
8	K	-9	Return -9
9	C	7	Return 7
10	A	7	Explore d
11	D	$-\infty$	Explore l
12	L	4	Return 4
13	D	4	Explore m
14	M	-2	Return -2
15	D	4	Explore n
16	N	-3	Return -3
17	D	4	Return 4
18	A	4	Explore e
19	E	$-\infty$	Explore o
20	O	5	Return 5
21	E	5	Explore p
22	P	-1	Return -1
23	E	5	Explore q
24	Q	2	Return 2
25	E	5	Return 5
26	A	4	Return 4

Table 2.1: Partial analysis of how the mini-max algorithm explores the tree of Figure 2.1.

WeakAlphaBeta($node, bound$)

```
1:  if node.depth = 0 then
2:     return Evaluate(node)
3:  if node.type = max then
4:     score ← −∞
5:  else
6:     score ← +∞
7:  for i ← 1 to node.branch.length
8:        new_node ← Traverse(node, node.branch[i])
9:        value ← WeakAlphaBeta(new_node, score)
10:       if node.type = max then
11:          if value ≥ bound then
12:             return bound
13:          if value > score then
14:             score ← value
15:       else
16:          if value ≤ bound then
17:             return bound
18:          if value < score then
19:             score ← value
20: return score
```

Figure 2.4: The weak alpha-beta algorithm.

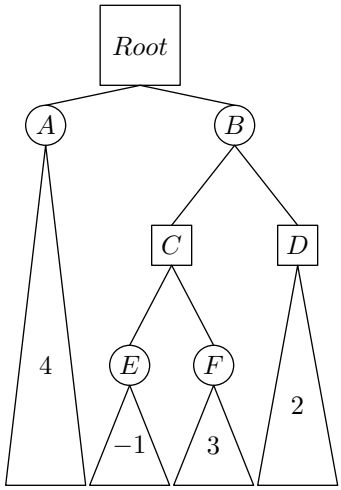

Figure 2.5: A game tree where the weak alpha-beta algorithm misses cut-offs.

The alpha-beta algorithm can be simplified using the nega-max formulation in a manner similar to the simplification of the mini-max algorithm. The reformulated algorithm is presented in Figure 2.8. Each node is treated as a maximizing node, thus *beta* is used as the bound that determines when cut-offs are possible. Furthermore, when making a recursive call, the bounds are reversed (the lower bound becomes the upper bound and vice versa) and negated. This allows the sub-node to be treated as a maximizing node. Note that evaluations are computed by EvaluateNegaMax as required by the nega-max scheme.

The efficiency of the alpha-beta algorithm is dependent on the order in which the branches are searched. If branches that lead to high scores (or low scores in the case of a minimizing node) are expanded first, then tighter bounds will be obtained for the rest of the search. This will result in a higher number of cut-offs. For some problems, the quality of a branch is not known until the leaves are reached, thus it is difficult to control the algorithm so that it searches "good" branches first. However, in many cases, an educated guess can be made regarding the quality of a branch from some preliminary information. Where such information is available, the efficiency of the search is greatly enhanced if the branches are explored in order of decreasing quality.

2.6 A Perfectly Ordered Game Tree and Node Classification

Consider the game tree in Figure 2.9. It is a revised version of the tree in Figure 2.1. The branches have been arranged so that the best branch at each node appears in the leftmost position. Such a tree is perfectly ordered for an alpha-beta search that expands branches in a left to right order. The nodes of a perfectly ordered game tree can be classified into three types

AlphaBeta($node$, $alpha$, $beta$)

```
1:  if node.depth = 0 then
2:      return Evaluate(node)
3:  if node.type = max then
4:      score ← alpha
5:  else
6:      score ← beta
7:  for i ← 1 to node.branch.length
8:          new_node ← Traverse(node, node.branch[i])
9:          if node.type = max then
10:             value ← AlphaBeta(new_node, score, beta)
11:             if value ≥ beta then
12:                 return beta
13:             if value > score then
14:                 score ← value
15:         else
16:             value ← AlphaBeta(new_node, alpha, score)
17:             if value ≤ alpha then
18:                 return alpha
19:             if value < score then
20:                 score ← value
21: return score
```

Figure 2.6: The alpha-beta algorithm.

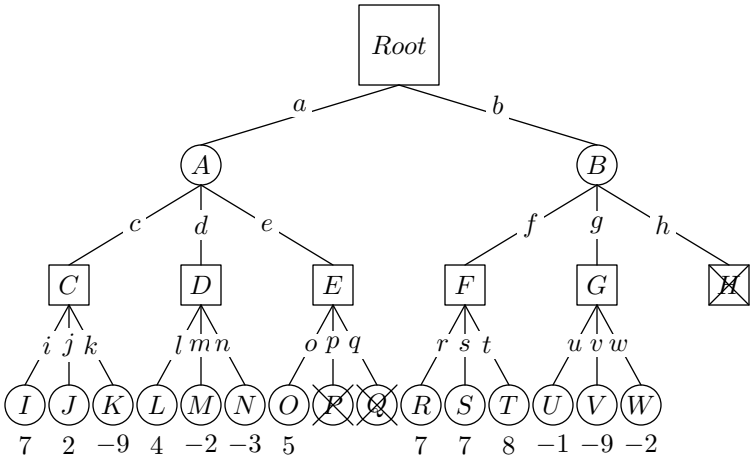

Figure 2.7: Cut-offs made by the alpha-beta algorithm.

AlphaBeta($node, alpha, beta$)

1: **if** $node.depth = 0$ **then**
2: **return** EvaluateNegaMax($node$)
3: **for** $i \leftarrow 1$ **to** $node.branch.length$
4: $new\_node \leftarrow$ Traverse($node, node.branch[i]$)
5: $value \leftarrow -$AlphaBeta($new\_node, -beta, -alpha$)
6: **if** $value \geq beta$ **then**
7: **return** $beta$
8: **if** $value > alpha$ **then**
9: $alpha \leftarrow value$
10: **return** $alpha$

Figure 2.8: The alpha-beta algorithm reformulated using the nega-max scheme.

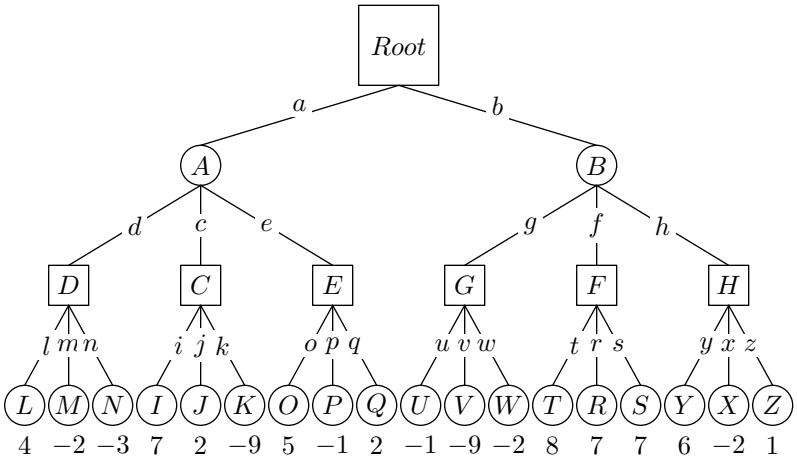

Figure 2.9: A perfectly ordered game tree.

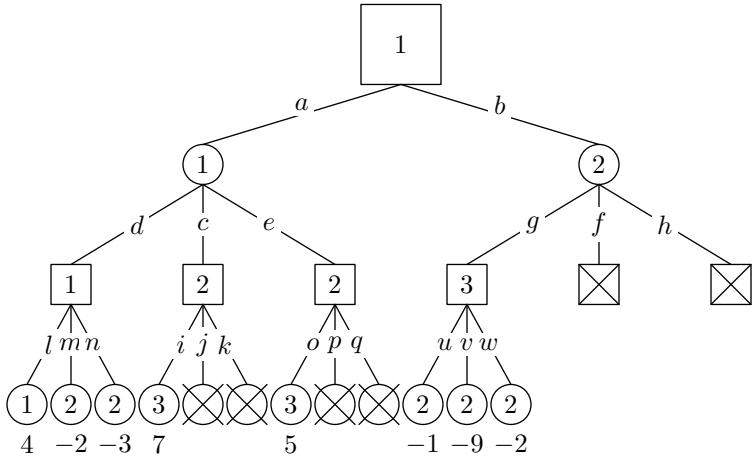

Figure 2.10: Cut-offs achieved by the alpha-beta algorithm on the tree of Figure 2.9.

[12, 17]:

- Type 1 or Principal Variation (PV) Nodes
- Type 2 or CUT Nodes
- Type 3 or ALL Nodes

The type of each node is indicated in Figure 2.10. The figure also illustrates the cut-offs made by an alpha-beta algorithm on the perfectly ordered tree.

2.6.1 Type 1 or PV Nodes

In a perfectly ordered tree, the first sequence of moves searched by the alpha-beta algorithm is also the principal variation. This is the case for any perfectly ordered tree. Each node of the

principal variation is described as being type 1 or PV.

Recall that a full alpha-beta search is initiated with the call AlphaBeta($Root, -\infty, +\infty$). Each PV node receives infinite lower and upper bounds. This stems from the fact that PV nodes constitute the start of the search and scores have yet to be established. Since the bounds are infinite, a cut-off never occurs at a PV node and all branches are searched. However, the search order at a PV node is important. For example, consider a PV node of the maximizing type. Initially the lower bound is $-\infty$ and the upper bound is $+\infty$. The score returned by the first branch will be used as the lower bound for the next branch to be expanded. Clearly, if the score returned by the first branch is the highest possible at that node, the bound will not change for the duration of the search at that node. Furthermore, the search benefits from the high lower bound that was established right at the start.

The first successor to a PV node is also a PV node while the other successors are CUT nodes.

2.6.2 Type 2 or CUT Nodes

CUT nodes are successors to PV nodes and ALL nodes. Since a CUT node is not the first successor at a PV node, it will have a bound as established by the PV node's first branch. Once again, consider a PV node of the maximizing type. The second branch to be expanded at the PV node will lead to a minimizing CUT node. The score returned by the PV node's first branch serves as the lower bound at the CUT node. Note that the CUT node does not have an upper bound since that was never determined at the maximizing PV node — a PV node only determines one of the two bounds. Since the tree being searched is perfectly ordered, the first branch searched at a CUT node immediately leads to a cut-off. Just as with PV nodes, search order is important at CUT nodes.

The first successor to a CUT node is an ALL node whereas the rest of the successors are cut-off.

2.6.3 Type 3 or ALL Nodes

An ALL node is a successor to a CUT node. Being the first branch at a CUT node, the ALL node obtains the same bound information as its parent. Consider the example of a minimizing CUT node. The CUT node will have a valid lower bound, however, the upper bound will be infinite. The first branch to be expanded at the CUT node leads to a maximizing ALL node. Since the upper bound is infinite at the ALL node, no cut-offs can be made and all branches are searched. Due to the perfect ordering of the tree, the scores returned by the ALL node's successors are not high enough to increase the lower bound — the scores returned are worse

than that established by the PV node higher up in the tree. In fact, the search order at an ALL node is irrelevant.

The successors at an ALL node are all CUT nodes.

2.7 Aspiration Search

When the alpha-beta algorithm is invoked with the call AlphaBeta(*Root*, *alpha*, *beta*), it returns a value between *alpha* and *beta* — note that *alpha* and *beta* are valid return values as well. Normally, to determine the game tree value, one would make a call to the algorithm with *alpha* and *beta* set to $-\infty$ and $+\infty$ respectively. Consider the case where the final game tree value is known with some certainty. Greater efficiency can be achieved by employing *aspiration search* [16, 15]. Figure 2.11 illustrates the process. The estimated game tree value is V. An error factor e is used to determine the two initial bounds *alpha* and *beta*. The lower bound is set at one error factor below V and the upper bound is set at one error factor above V. When the alpha-beta algorithm is called with these bounds, the return value is one of three types:

- A return value between *alpha* and *beta*: In this case the game tree value has been determined and further search is not necessary. The tree search was highly efficient due to the narrow search window.
- A return value equal to *alpha*: The search has *failed-low*. The real game tree value is not known; all that is known is that the tree value is less than or equal to *alpha*.
- A return value equal to *beta*: The search has *failed-high*. The game tree value is greater than or equal to *beta*.

In a fail-low or fail-high situation, a new search must be performed with new bounds in order to determine the real tree value. During a new search, many of the nodes that were visited by the initial search may be revisited. Clearly, the new search reduces efficiency; however, note that this situation only arises when the original search fails — a situation that does not arise too often since there is an estimate for the final game tree value.

2.8 The Negascout Algorithm

As mentioned in Section 2.5, when there is some preliminary "quality" information about the branches at a node, the alpha-beta algorithm benefits from the expansion of the branches in order of decreasing quality. If this preliminary information can be used to predict which of the branches is the best at a node with reasonable accuracy, a technique similar to aspiration search can be employed recursively to obtain a highly efficient tree searcher.

1: $alpha \leftarrow V - e$
2: $beta \leftarrow V + e$
3: $score =$ AlphaBeta($Root, alpha, beta$)
4: **if** $score = beta$ **then**
5: $score =$ AlphaBeta($Root, beta, +\infty$)
6: **elsif** $score = alpha$ **then**
7: $score =$ AlphaBeta($Root, -\infty, alpha$)

Figure 2.11: Aspiration search.

The negascout algorithm [20] [1] is illustrated in Figure 2.12. Only the first branch at each node is searched with the full window. The rest of the branches are searched with a *null-window*. A null-window describes the case where *alpha* and *beta* are separated by 1 unit. In this case no real search is performed — the null-window search amounts to a test on the subtree to determine whether its value is less than or equal to *alpha* or greater than or equal to *beta*. In the negascout algorithm, after searching the first branch with the full window, the rest of the branches are simply tested to see whether they have a value that exceeds the best score so far at that node. If any of the branch tests fails-high, a new search is performed with an expanded window to establish the true score of that particular branch.

2.9 Iterative Deepening

Rather than tackle the entire tree at once, depending on the application, it may be advantageous to determine the tree value in steps. First, the game tree value and principal variation for a tree of depth one is determined. Then, a similar process is carried out on a tree of depth two. This *iteratively deepening* process [16, 15] continues until the tree of the required depth has been searched. There are two advantages to this scheme. In a situation where there is a time restriction on the search, the search of the entire tree in a single pass may be too time consuming. However, if the search is carried out in steps, even if the search needs to be stopped at some depth, the previous iteration provides a reasonable solution albeit at a lower search depth. The second advantage is that each iteration can collect useful information about the tree for the next iteration. Consider some examples of this iteration-to-iteration information

[1] The algorithm presented here is a simplified version of the one presented in [20]. In particular, this version does not make use of the *fail-soft* extension and performs new searches even when the height of the subtree is less than two.

```
NegaScout(node, alpha, beta)

1:  if node.depth = 0 then
2:      return EvaluateNegaMax(node)
3:  new_node ← Traverse(node, node.branch[1])
4:  value ← −NegaScout(new_node, −beta, −alpha)
5:  if value ≥ beta then
6:      return beta
7:  if value > alpha then
8:      alpha ← value
9:  for i ← 2 to node.branch.length
10:     new_node ← Traverse(node, node.branch[i])
11:     value ← −NegaScout(new_node, −alpha − 1, −alpha)
12:     if value > alpha and value < beta then
13:         value ← −NegaScout(new_node, −beta, −alpha − 1)
14:     if value ≥ beta then
15:         return beta
16:     if value > alpha then
17:         alpha ← value
18: return alpha
```

Figure 2.12: The negascout algorithm.

```
1:  node ← Root
2:  V ← initial_estimate
3:  for d ← 1 to depth
4:      node.depth ← d
5:      alpha ← V − e
6:      beta ← V + e
7:      score = AlphaBeta(node, alpha, beta)
8:      if score = beta then
9:          score = AlphaBeta(node, beta, +∞)
10:     elsif score = alpha then
11:         score = AlphaBeta(node, −∞, alpha)
12:     V ← score
```

Figure 2.13: An iteratively deepening alpha-beta search that uses the game tree value from the previous iteration in an aspiration search of the current iteration.

sharing:

- The principal variation from the previous iteration is usually a good indicator of what the principal variation from the current iteration will look like. Search efficiency usually improves if the branches from the last principal variation are expanded first.

- The game tree value from the last iteration is a good estimate of what the game tree value will be at the end of the current iteration. Therefore, the game tree value from the previous iteration can be used to guide an aspiration search of the tree to the depth required by the current iteration.

  Figure 2.13 combines iterative deepening with aspiration search. Aspiration search uses a small window centered around the game tree value from the preceding iteration for the search during the current iteration.

- The previous iteration may have retained quality information about certain key branches in the tree. The nature of this information is usually application dependent. This information is then used by the current iteration to determine the ordering of the branches at a node so that good branches will be expanded first.

Chapter 3

Parallel Alpha-Beta Search

3.1 Introduction

Parallelization of the alpha-beta algorithm and its variants has proven to be difficult. The overheads in a parallel implementation of the algorithm can be classified into three categories [21]:

- Communication overhead
- Synchronization overhead
- Search overhead

First, consider the communication overhead. Communication between processors is normal in any parallel algorithm; however, there is one type of communication that is unique to parallel alpha-beta. The sequential alpha-beta algorithm updates its two bounds, *alpha* and *beta*, as the search of a game tree progresses. When searching in parallel, if one processor finds an improvement to *alpha* or *beta*, it informs the other processors working below that node so that they can make use of the tighter bound that was just discovered.

Synchronization overhead results when a processor sits idle while waiting for some event to occur. For example, if four processors, P1, P2, P3 and P4, are working together at a node, processors P2, P3 and P4 may be waiting for the result of a search being conducted by processor P1 on some branch at that node.

Search overhead is a consequence of the parallel alpha-beta algorithm examining nodes that would have been avoided by the sequential version. When parallel search is initiated at a node, the best score might not have been discovered as yet. As a result, parallel search is conducted with a wider window than in the sequential case. Furthermore, after parallel search has been initiated, one processor may discover that search is no longer necessary at the node due to a cut-off condition — the other processors have essentially performed useless work.

The three overheads are not independent of each other; they are related in a complex manner. Let us consider a few examples of this complex relationship. To reduce search overhead, the processors may continuously update each other with the latest search information thereby increasing communication overhead. In another strategy to reduce search overhead, we may require that a certain number of branches be explored in a sequential fashion before attempting parallel search at a node. When the parallel search is actually started, it is highly likely that the score at the node will have stabilized. Communication overhead is also reduced as messages carrying bound information will be less frequent. However, synchronization overhead will increase since there will be several processors waiting idly while a single processor completes the required number of branches.

In this chapter, important work in the area of alpha-beta parallelization is presented. While the literature describes several methods [1], only three are described here. These three methods have been chosen carefully from the several that are available. The first method, principal variation splitting (PVSplit) [16], is the result of some of the earliest attempts at parallelizing alpha-beta. Although it is a relatively old idea, PVSplit has been the subject of much research and it has been the source of inspiration for several newer methods. The other two methods, young brothers wait concept (YBWC) [6] and dynamic tree splitting (DTS) [9], are more recent and spectacular speed-ups have been reported. These two methods embody two different design goals: YBWC was developed in a distributed environment where communication costs are high, whereas DTS was developed in an environment where communication is quite cheap.

3.2 Super-Linear Speed-Up?

Although it rarely ever happens in practice, parallel alpha-beta may yield super-linear speed-ups. Consider the tree of Figure 3.1. It is assumed that this tree is part of a much bigger tree. When the search arrives at the *Root* node, *alpha* has a value of -5 and *beta* has a value of 0.[1] It is assumed that each major operation in alpha-beta can be completed in one time unit. Table 3.1 illustrates the progress of a single processor executing the alpha-beta algorithm on the tree. It takes 25 time units to complete the search. Now, consider the situation where two processors, P1 and P2, are working together at the *Root* node. The first processor handles the subtree rooted at node A while the second handles the subtree rooted at node B. The progress of processors P1 and P2 is illustrated in Tables 3.2 and 3.3 respectively. Since move ordering is bad within the subtree rooted at A, all nodes will have to be explored. However, there is perfect ordering in the subtree rooted at B and cut-offs are plentiful. The second processor completes its search in a short period of time and it discovers that a cut-off condition exists at the *Root*.

[1] This discussion uses mini-max conventions as opposed to nega-max conventions.

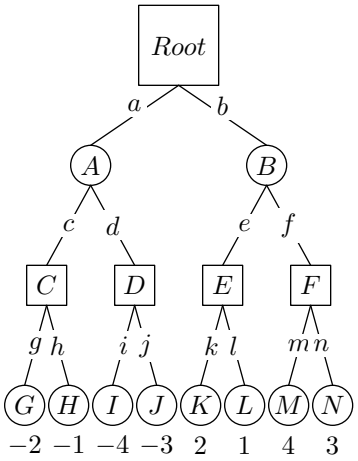

Figure 3.1: Parallel search on this tree yields super-linear speed-ups.

Processor 1 is then stopped and the search of the *Root* node is complete. With two processors, the search takes a mere 11 time units. Clearly, super-linear speed-up is a possibility, but the rather contrived nature of the tree in Figure 3.1 and the bounds used should be indicative of the fact that super-linear speed-up rarely occurs in practice.

3.3 Principal Variation Splitting

In PVSplit [16], the rule is that the first branch at a PV node must be searched before parallel search of the remaining branches may begin. All processors travel down the first branch at each PV node until they reach the PV node that is one level above the leaf nodes. Here one processor searches the first branch while the other processors wait. Once the first branch has been examined, all processors join the search effort. Each processor takes away a branch at a time and determines its value. If a processor discovers an improvement to the score at the node, it informs the other processors of the updated value. When there aren't any unassigned branches, a processor that runs out of work remains idle until the other processors finish. Once all branches have been examined, the search effort moves the current node's parent. Since the value of the parent node's first branch was just computed, parallel search can be started at the parent as well. This process continues upwards in the tree until all the branches at the *Root* node have been examined.

Figure 3.2 illustrates the progress of two processors, P1 and P2, as they use PVSplit on a small tree. Both processors travel down the leftmost path until they reach node D. At this node, P2 remains idle while P1 explores branch g. Once the exploration of g is complete, parallel search is started; P1 explores h while P2 explores i. When D has been completely evaluated,

Time	*Node*	*Alpha*	*Beta*	*Score*	*Action*
0	*Root*	-5	0	-5	Explore a
1	A	-5	0	0	Explore c
2	C	-5	0	-5	Explore g
3	G	-5	0	-2	Return -2
4	C	-5	0	-2	Explore h
5	H	-2	0	-1	Return -1
6	C	-5	0	-1	Return -1
7	A	-5	0	-1	Explore d
8	D	-5	-1	-5	Explore i
9	I	-5	-1	-4	Return -4
10	D	-5	-1	-4	Explore j
11	J	-4	-1	-3	Return -3
12	D	-5	-1	-3	Return -3
13	A	-5	0	-3	Return -3
14	*Root*	-5	0	-3	Explore b
15	B	-3	0	0	Explore e
16	E	-3	0	-3	Explore k
17	K	-3	0	2	Return 2
18	E	-3	0	-3	Cut-off
19	B	-3	0	0	Explore f
20	F	-3	0	-3	Explore m
21	M	-3	0	4	Return 4
22	F	-3	0	-3	Cut-off
23	B	-3	0	0	Return 0
24	*Root*	-5	0	-3	Cut-off

Table 3.1: Progress of the alpha-beta algorithm on the tree of Figure 3.1.

Time	*Node*	*Alpha*	*Beta*	*Score*	*Action*
0	*Root*	-5	0	-5	Explore a
1	A	-5	0	0	Explore c
2	C	-5	0	-5	Explore g
3	G	-5	0	-2	Return -2
4	C	-5	0	-2	Explore h
5	H	-2	0	-1	Return -1
6	C	-5	0	-1	Return -1
7	A	-5	0	-1	Explore d
8	D	-5	-1	-5	Explore i
9	I	-5	-1	-4	Return -4
10	D	-5	-1	-4	Explore j

Table 3.2: Progress of P1 on the tree of Figure 3.1.

Time	*Node*	*Alpha*	*Beta*	*Score*	*Action*
0	*Root*	-5	0	-5	Explore b
1	B	-5	0	0	Explore e
2	E	-5	0	-5	Explore k
3	K	-5	0	2	Return 2
4	E	-5	0	-5	Cut-off
5	B	-5	0	0	Explore f
6	F	-5	0	-5	Explore m
7	M	-5	0	4	Return 4
8	F	-5	0	-5	Cut-off
9	B	-5	0	0	Return 0
10	*Root*	-5	0	-5	Cut-off

Table 3.3: Progress of P2 on the tree of Figure 3.1.

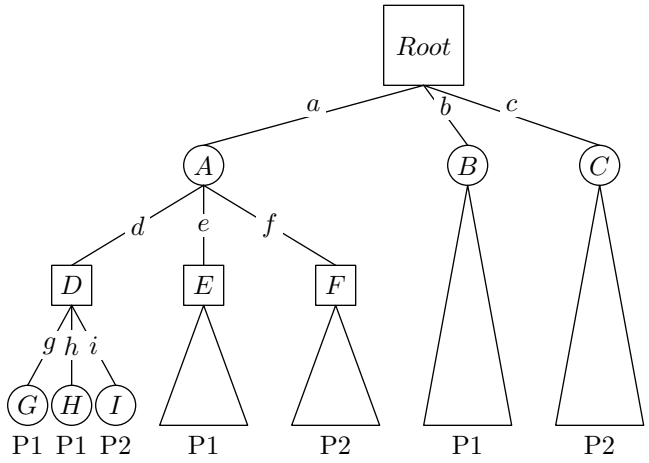

Figure 3.2: Two processors using PVSplit to divide up the work in a small tree.

the search moves to its parent, A. Since the eldest brother, d, has been examined at A, parallel search can be employed once again. Processor P1 evaluates e while P2 evaluates f. Once A has been evaluated, parallel search begins at the *Root* node with P1 evaluating b and P2 evaluating c.

Let us examine some of the reasoning behind PVSplit. First, at a PV node all branches have to be searched, thus parallel search is a good idea at this type of node. Second, by requiring that the first branch be examined before parallel search is started at a node, parallel search starts only when a bound has been determined. If the branches have been ordered according to some preliminary quality information, then the score returned by the first branch may be the best possible at that node. In fact, the original work [16] described PVSplit as a method for searching strongly-ordered trees (refer to Section 4.3) where the first branch at any node is the best 70 percent of the time. If the best possible score is obtained when the first branch is examined, parallel search will examine precisely the same nodes as the sequential version. Therefore, there is no search overhead if at each PV node the first branch is also the best. Third, search at a PV node examines more nodes than the search of any other node of equivalent height because a PV node has no bound information — alpha and beta are at negative infinity and positive infinity respectively. Thus, a PV node is a reasonable choice as a site for parallel search.

The PVSplit method is not without its faults. When the first branch is not the best, search overhead increases as parallel search is conducted with a bound that is not as tight as in a sequential search. Depending on the branching factor, the method may not be able to use a large number of processors effectively. For example, in chess, where the average branching factor is 32, the method does not have a mechanism for handling more than 31 processors. Synchronization overhead is a significant problem in PVSplit. Consider what happens as parallel search at a

Processors	1	2	4	8	16
Speed-Up	1.0	1.8	3.0	4.1	4.6

Table 3.4: Performance of PVSplit on chess trees.

Processors	1	2	4	8	16
Speed-Up	1.0	1.9	3.4	5.4	6.0

Table 3.5: Performance of EPVS on chess trees.

node nears its end. Most of the processors in the search effort will be idle waiting for a few processors that are searching "difficult" branches. A difficult branch is one that requires a larger search effort because more nodes are examined in the subtree generated by that branch compared to the other branches at the node.

Experiments with PVSplit have shown that speed-up is limited to a large extent by synchronization overhead [17, 21, 9]. Searching chess trees on a Cray C90, the technique produces the speed-ups in Table 3.4 [9]. The speed-up seems to be limited to an upper bound of 5. This has led to some interesting work that tries to reduce the synchronization overhead in PVSplit. In enhanced principal variation splitting (EPVS) [9], when a processor becomes idle, all processors move to the subtree being searched by one of the busy processors and a site for parallel search is created two levels below the original site. Experiments with this method produced the speed-ups in Table 3.5 [9]. Although this method is a little better than plain PVSplit, it is still not very efficient when a large number of processors are involved in the search effort. In Dynamic PVSplit (DPVS) [21], each processor runs a version of PVSplit. However, the difference is that a controller process dynamically assigns idle processors to help the busy processors in the system. Searching chess trees, this technique obtained a speed-up of 7.64 on a network of 19 Sun 3/75s.

3.4 Young Brothers Wait Concept (YBWC)

There are two different versions of YBWC. The earliest one is referred to as the weak YBWC [6]. A more recent version that modifies the technique slightly is referred to as the strong YBWC [5]. Whenever a distinction needs to be made between the two versions, the qualifications weak and strong will be used. However, if the discussion applies to both techniques, then the method will be referred to as simply YBWC.

At any node, the first branch to be expanded is referred to as the *eldest brother* while the

other branches are referred to as the *younger brothers*. In weak YBWC, the rule is as follows: the eldest brother has to be examined before parallel search of the younger brothers is possible. Although this is similar to PVSplit, in YBWC parallel search is possible at any node not just at PV nodes.

Before delving into the method's details, the concept of node ownership is introduced. A processor that owns a node is responsible for its evaluation. It is also responsible for returning the node's evaluation to its parent. Note that this may involve communication if the owner of the evaluated node is different from that of its parent. Usually, a node and its successors have the same owner. However, multiple processors can collaborate on a node if some of the successors have different owners. In YBWC, once a processor is given ownership of a node, the ownership of that node is not transferable to another processor.

At the start, one processor is given ownership of the *Root* node while the other processors remain in an idle state. A processor, P1, that is idle selects another processor, P2, at random and transmits a message requesting work. Processor P2 has work available if there is at least one node in the subtree it is examining that satisfies the weak YBWC criterion. That is, P2 has work available if it owns a node at which the eldest brother has been evaluated. The node that satisfies the criterion becomes the *split-point*; if there are many nodes that satisfy the criterion, then the node that is the highest in the tree is selected as the split-point. A split-point is a node that has been chosen as a site for parallel search.

If P2 has work available then a master-slave relationship is established between P2 and P1. Note that P1 may be one of many slaves to P2. The master and its slaves share the search effort at the split-point. Each processor takes away a branch at a time until the search at the split-point is complete. As in PVSplit, if one processor finds an improvement to the score at the split-point then the new score is transmitted to the other processors involved. A processor may also discover a cut-off condition at the split-point. In this case, the search is complete and the slaves return to their idle state. A slave may also return to its idle state if there isn't any work left at the split-point. If the master returns from the search of some branch to find no work at the split-point, it should not remain idle while waiting for the busy processors to finish because this would increase synchronization overhead. Instead, the master acts as a slave to one of the busy processors. This is referred to as the *helpful master* concept.

When an idle processor P1 transmits a message requesting work to a processor P2, the latter may not have any work available. Processor P2 forwards the request message to another randomly selected processor. However, if the message has already traveled through a certain number of processors, P2 throws away the message and informs P1 that no work is available. Processor P1 then begins requesting again.

In strong YBWC, the nodes of a game tree are classified into three types:

- *Y-PV*: The root node is of type Y-PV. The first successor at a Y-PV node is of type Y-PV while the rest of the successors are Y-CUT.
- *Y-CUT*: The first successor is a Y-ALL node while the rest are Y-CUT nodes.
- *Y-ALL*: All successors are Y-CUT nodes.

Note that the definition of a Y-PV node is the same as that of a PV node except that it produces Y-PV and Y-CUT nodes as successors. Furthermore, a Y-ALL node is similar to an ALL node except that it produces Y-CUT nodes as successors. However, a Y-CUT node is quite different compared to a CUT node. Recall that a CUT node is defined as having only one successor and that the lone successor is of type ALL. While that definition is suitable for a tree that is perfectly ordered, when a tree is imperfectly ordered, a CUT node may have more than one successor. The new node classification specifies that these additional successors at CUT nodes are of type Y-CUT.

Strong YBWC uses the weak YBWC criterion at Y-PV and Y-ALL nodes. However, at Y-CUT nodes, strong YBWC enforces a different rule: all "promising" branches must be examined before parallel search is possible. A promising branch is one that is likely to produce a cut-off based on some preliminary quality information. The exact definition of a promising branch is application dependent. In strong YBWC there is a longer wait at CUT nodes before parallel search is possible. Although this reduces the potential parallelism, the search overhead is greatly reduced and in practice, strong YBWC produces better speed-ups than weak YBWC.

On a Parsytec SC 320 machine (based on the T800 Transputer), weak YBWC obtained a speed-up of 137 when searching chess trees with 256 processors [5]. Strong YBWC obtained a speed-up of 142 on the same system. Experiments were also conducted on a Parsytec GCel machine (based on the T805 Transputer) with 1024 processors. With 1024 processors, strong YBWC produced a speed-up of 344.

3.5 Dynamic Tree Splitting (DTS)

DTS [8, 9] uses a peer-to-peer approach rather than a master-slave approach as in YBWC. Node ownership takes on a different meaning in DTS. While many processors may collaborate on a node, the processor that finishes its search last is responsible for returning the node's evaluation to its parent.

At the start, one processor is set to search the *Root* node while the other processors are in an idle state. An idle processor consults a global list of active split-points (SP-LIST) to find work to do. If a split-point with work is found, the idle processor joins the other processors that are working at that split-point and the work at that node is shared. However, if no work can be

found in SP-LIST, the idle processor broadcasts a HELP message to all processors. On receipt of the HELP message, a processor that is busy copies the state of the subtree it is examining to a shared area. The idle processor then examines the shared area to find a suitable split-point. If a split-point can be found, the split-point is first copied into SP-LIST and the idle processor then shares the work at the split-point with the processor that originally expanded the node. If a suitable split-point cannot be found in the shared area, the idle processor rebroadcasts the HELP message after a small delay.

When a processor returns from the search of some branch to find no work at a split-point, the processor simply enters the idle state where it can try to find work at another node. However, if a processor returning from the search is the last processor at the split-point, then the processor is responsible for returning the node's evaluation to its parent. Furthermore, this processor does not enter the idle state but continues working at the parent node.

Similar to PVSplit and YBWC, if one processor discovers an improved score, the score is shared with the other processors working at the split-point. Instead of an improved score, if a cut-off condition is discovered, a single processor is left at the node as its owner while the other processors return to their idle states.

Finding a suitable split-point after having broadcast the HELP command is rather complicated. The selection procedure is not as simple as the one found in YBWC. First, the type of each node is determined. The set of rules that is used to determine node type in DTS is quite different from YBWC, therefore different names are used to avoid any confusion. A node is classified into three types:

- *D-PV*: A node that has the same *alpha* and *beta* values as the *Root*.
- *D-CUT*: A minimizing node with the same *beta* as the *Root* or a maximizing node with the same *alpha* as the *Root*.
- *D-ALL*: Any node that does not fit the D-PV and D-CUT criteria.

The types, D-PV and D-CUT, are equivalent to the normal types, PV and CUT, respectively. However, the D-ALL node is much broader in scope than the ALL type. Although every ALL node is also a D-ALL node, the D-ALL type also encompasses those nodes that are searched due to imperfect ordering. After determining the node type, there are two override phases. During the first override phase, a node's type is changed from D-CUT into D-ALL if more than three nodes have been examined at the node without having achieved a cut-off. The second override phase deals with the situation where there are several D-ALL nodes at consecutive levels in the tree. DTS only allows two D-ALL nodes to be consecutive in the tree. Following the second D-ALL node, nodes are forced into an alternating sequence of D-CUT and D-ALL

Processors	1	2	4	8	16
Speed-Up	1.0	2.0	3.7	6.6	11.1

Table 3.6: Performance of DTS on chess trees.

nodes. There is also a confidence factor associated with each D-CUT and D-ALL node. If many moves (up to a limit of three) have been searched at a D-CUT node, then the confidence that it is a D-CUT node is lowered. If several moves have been searched at a D-ALL node, then the confidence that it is a D-ALL node increases. A node's suitability as a split-point is based on four factors:

- The node must be of type D-PV or D-ALL.
- The height of the node. Nodes that are higher up in the tree (closer to the root) represent more work.
- If it is a D-PV node, its first branch must have been searched.
- If it is a D-ALL node, the confidence factor should be relatively high.

The process of selecting a split-point is quite complicated but all this effort is in an attempt to reduce the search overhead.

Searching chess trees on a Cray C916/1024 machine, DTS produced the speed-ups in Table 3.6 [9]. Since DTS was designed with shared memory in mind, experiments with a large number of processors are not available as shared memory multiprocessors are hard to find in large configurations.

Chapter 4

Artificial Game Trees

4.1 Introduction

Although a wide variety of game trees arise in practice, as far as the alpha-beta algorithm is concerned, a tree's average branching factor and the quality of its branch ordering usually governs the length of time spent searching the tree. Searching artificially generated trees allows one to examine the algorithm's behavior for a wide variety of branching factors and branch orderings. When artificial trees are used as a model for the trees encountered in practice, they should mimic the behavior exhibited by real trees. This chapter introduces the notion of an *exponentially ordered* tree. A method for generating such trees is also described.

4.2 Generating Artificial Trees

Any method used in artificial tree generation should satisfy three requirements. First, the method must be able to generate a wide variety of trees. Second, when identical input parameters are used, identical trees should be produced. This is particularly important because the artificial trees are used to compare different techniques and identical trees must be presented to each technique. Third, the order of branch expansion should not affect the tree generated. In sequential search this requirement has no effect, but in parallel search the order of branch expansion is usually dependent on the parallel search method. The method described below satisfies all three requirements.

Three routines used for random number generation in the artificial tree generator are examined first before a treatment of the method itself. Figure 4.1 shows these three routines. The RANDOM function is the main random number generator. It is of the linear congruential type. The RANDOMSEED function is used to seed the main generator. In reality, this is also another linear congruential generator. Normally, it would have been sufficient to simply copy the seed

1: **constant** *random_limit* ← 2147483647

RandomSeed(*rnd*, *seed*)

2: *rnd.state* ← (16807 · *seed*) **mod** *random_limit*

Random(*rnd*)

3: *rnd.state* ← (48271 · *rnd.state*) **mod** *random_limit*
4: **return** *rnd.state*

RandomRange(*rnd*, *lower*, *upper*)

5: **return** Random(*rnd*) · (*upper* − *lower* + 1)/*random_limit* + *lower*

Figure 4.1: Random number routines used by the artificial tree generator.

into the generator's state variable. Furthermore, if the seeding function had to use a multiplier, it could have used the same multiplier as the main generator. However, a standard seeding function is not adequate for the artificial tree generator because it uses multiple random number streams and each stream is initialized using a seed generated by another stream. If either no multiplier or the same multiplier was used, then the same sequence of random numbers would be generated by a stream and the stream that produced its seed. The random tree method [4], which is used for parallel random number generation, provides the necessary insight to solve the problem of similar sequences — a second linear congruential generator can be used to separate the two sequences so that they appear different. Additionally, note that the multipliers used, 16807 and 48271, and the modulus, 2147483647, are known to have good randomness properties [11]. The RandomRange function is an extension of Random to provide random numbers in a certain range, [*lower*, *upper*]. It simply scales and shifts a random number generated by Random into the appropriate range before returning the result.

Virtually all of the tasks carried out by the artificial tree generator can be combined into a single routine. This routine will be referred to as BranchGenerate and is presented in Figure 4.2. Every internal node invokes BranchGenerate to generate its successors. The generation of successors at each node is controlled by two variables, *seed* and *pscore*. Both of these variables are set by a node's parent. The generation of successors is as follows. First,

a new stream of random numbers is initialized using *seed* (lines 1 – 2). Each successor needs values for its *seed* and *pscore* variables. It also needs values for its *depth* (which represents the height of the tree at the node) and *type* (maximizing or minimizing) variables. Strictly speaking, *depth* and *type* are not related to artificial tree generation — they are integral to any tree expansion routine, artificial or not. Each successor's *seed* is generated by a call to Random (line 17). Determining a value for a successor's *pscore* is more involved. A node's *pscore* value represents the score that is obtained when the results returned by the successors are maximized or minimized as required at the node. The *pscore* for the first successor is set equal to the node's *pscore*. The smallest and largest score that can be generated is held in the constants, *score_min* and *score_max*, respectively. If a node is of the maximizing type, then the value of the *pscore* variable for each of the remaining successors is obtained by generating random numbers in the range $[score\_min, pscore]$ (line 10). However, if the node is of the minimizing type, then random numbers in the range, $[pscore, score\_max]$, are used instead (line 14). Consider what has been accomplished so far. The best score at a node is always in its first position. If the node being expanded is of the maximizing type, the remaining successors will all have scores that are smaller than or equal to the first successor. If the node is of the minimizing type, the remaining successors will have scores that are greater than or equal to the first successor. This essentially produces a perfectly ordered tree. However, the ordering produced by the artificial tree generator is of no concern since the BranchGenerate routine will be followed by a BranchOrder routine whose sole purpose is to order the successors in a more realistic manner.

In Figure 4.3, the artificial tree generator has been incorporated into an AlphaBeta search routine. At a leaf node, the value of *pscore* is returned as the node's score. At all other nodes, BranchGenerate is called to generate the node's successors. Once the execution of BranchGenerate is complete, BranchOrder is called to order the successors. A possible BranchOrder routine is described in Section 4.3.3.

Note that the entire tree is dependent on only two values: the *Root* node's *seed* and *pscore*. In the experiments conducted, different seeds are used to generate different trees, however, the *pscore* value is fixed. The value of the *pscore* variable is fixed to the value $(score\_max - score\_min)/2$. If the *Root* node's *pscore* is too low, then the range used at the root, $[score\_min, pscore]$, will be too small. Similarly, if the value of *pscore* is too high, then the first successor's range, $[pscore, score\_max]$, will be too small. Therefore, a value for *pscore* somewhere in the middle of the range of scores that are possible offers the greatest flexibility. In the experiments described in this text, two sets of seeds are used to generate artificial trees. The sets will be referred to as tg_1 and tg_2. The first set, tg_1, contains 100 seeds while the second set, tg_2, contains 200 seeds. Both sets are described in Appendix 1.

```
1:  score_min ← 0
2:  score_max ← random_limit − 1

BranchGenerate(node)

3:  node.rnd ← new random_type
4:  RandomSeed(node.rnd, node.seed)
5:  for i ← 1 to node.child.length
6:      node.child[i] ← new node_type
7:      if i = 1 then
8:          node.child[i].pscore ← node.pscore
9:      if node.type = max then
10:         node.child[i].type ← min
11:         if i ≠ 1 then
12:             node.child[i].pscore ← RandomRange(node.rnd, score_min, node.pscore)
13:     else
14:         node.child[i].type ← max
15:         if i ≠ 1 then
16:             node.child[i].pscore ← RandomRange(node.rnd, node.pscore, score_max)
17:     node.child[i].depth ← node.depth − 1
18:     node.child[i].seed ← Random(node.rnd)
```

Figure 4.2: Generation of successors using the artificial tree generator.

AlphaBeta($node, alpha, beta$)

```
1:  if node.depth = 0 then
2:     return node.pscore
3:  BranchGenerate(node)
4:  BranchOrder(node)
5:  if node.type = max then
6:     score ← alpha
7:  else
8:     score ← beta
9:  for i ← 1 to node.child.length
10:    if node.type = max then
11:       value ← AlphaBeta(node.child[i], score, beta)
12:       if value ≥ beta then
13:          return beta
14:       if value > score then
15:          score ← value
16:    else
17:       value ← AlphaBeta(node.child[i], alpha, score)
18:       if value ≤ alpha then
19:          return alpha
20:       if value < score then
21:          score ← value
22: return score
```

Figure 4.3: Incorporating the artificial tree generator into an alpha-beta search procedure.

4.3 Modeling Branch Ordering

To mimic the branch orderings observed in practice, Marsland and Campbell [16] propose the model of a *strongly ordered* tree where the branches are ordered according to two rules:

- The first branch at any node is the best 70 percent of the time.
- There is a 90 percent chance that the branch with the best score is located within the first quarter of the branches at a node.

In [18], to generate a strongly ordered tree with a branching factor of 20, each branch is given a weight:

$$w_1 = 70, \quad w_2 = 5, \quad w_3 = 5, \quad w_4 = 5, \quad w_5 = 5, \quad w_6 = \frac{2}{3}, \quad \ldots, \quad w_{20} = \frac{2}{3}$$

Each weight represents the probability of that move being chosen as the best at that node.

Experiments with the author's chess program, RajahX [14], indicate that the model of a strongly ordered tree is not an accurate representation of the branch orderings observed in practice. A new model, the *exponentially ordered* model is introduced to overcome the deficiencies in the strongly ordered model. If necessary, the exponential model can be adjusted so that it satisfies the criteria of the strongly ordered model.

4.3.1 A Description of RajahX

Experience with RajahX has shown that the program is quite a strong chess player. It has competed successfully in two tournaments. At the 1996 Dutch Computer Chess Championships the program placed 13th in a field of 20 participants. After several modifications to the program, at the 1997 Aegon Man-Machine tournament the program finished 17th in a field of 100. Playing on a Pentium 166, the program's blitz rating hovers around the 2500 mark on the Internet Chess Club.

RajahX uses a negascout search routine with numerous enhancements. At a leaf node, a quiescence search routine is used to refine the node before it is evaluated. The structure of the tree generated by the program is not uniform. Variations with moves that seem "interesting" are searched more deeply in order to uncover any gains or losses that are outside the normal search depth. The program uses several methods to improve the efficiency of its search routine:

- Iterative deepening.
- Aspiration search.

- *Killer table* [16, 15]: A move that is found to produce a cut-off is stored in a killer table. If the move comes up again in another part of the tree, it should be placed near the front of the move list since it is likely to produce a cut-off again.

- *History list* [15]: A history list contains 4096 entries (64×64). It contains an entry for each ($from\_square, to\_square$) pair in chess. The entry indicates how often a move from one square to another is the best or is able to produce a cut-off at a node. This table is maintained as the search progresses and is used to order the branches at every node in the tree.

- *Transposition table* [16, 15]: The score determined for each position encountered during the search can be stored in a hash table that is accessed using a key generated from that board position. If the position arises again through a different sequence of moves, the table can immediately provide a score for the position and search is completely avoided.

- *Null move search* [3]: In chess, a "pass" is not a legal move. However, if such a move was legal, in most positions, the player on move would not pass since the other player could potentially improve his position greatly. This observation can be used during the search as follows. At a node in the tree, if the player on move makes a pass and the resulting score is higher than the upper bound then making a non-pass move would result in a significantly higher score. Therefore, when the score for a pass type move is higher than the upper bound, the node can be cut-off right away since it is likely to contribute nothing to the overall search effort.

RajahX also implements a simple learning method [23]. Positions that are found to be problematic for the program are stored in a table and the program "learns" to avoid these positions when they arise again during tree search.

4.3.2 Quality of Branch Ordering in RajahX

To measure the quality of the first branch expanded at each node, a method suggested in [10] measures the frequency with which the first branch is the best or is able to produce a cut-off. This frequency is referred to as f_{best}. The method assumes that the branch that causes the cut-off is also the best. This assumption produces a highly optimistic estimate of the quality of branch ordering because a branch that produces a cut-off at a node is not necessarily the node's best. There is a large degree of uncertainty in the ranking of the cut-off branch among the other branches at the node, because an exact score is unavailable for that branch and several branches may be unexplored when the cut-off occurs. However, f_{best} still provides useful information

about the tree structure. Specifically, f_{best} can be used as a criterion for comparing the trees produced by RajahX with the trees produced by the artificial tree generator.

To compute the value of f_{best} for RajahX, a minor program modification was necessary. The negascout search algorithm is replaced by the simpler alpha-beta algorithm. Most of the time, the negascout algorithm searches with a null window. Cut-offs are more frequent in a null window search and the greater frequency of cut-offs will skew the value of f_{best} even further.

Statistics are collected while RajahX conducts searches on the Bratko-Kopec set of 24 test positions [13]. The program is asked to search each position for 10 seconds. Tables 4.1, 4.2 and 4.3 summarize the statistics collected. Every node in the tree does not contribute to the computation of f_{best}. For example, at a maximizing node, if every branch returns a score equal to the lower bound then it is not clear which branch is the best. Furthermore, a cut-off never occurs at such a node. The second column in Table 4.1 gives the number of nodes that contribute to the computation of f_{best}. To determine the average branching factor, the number of *pseudo-legal* moves at every node that contributes to f_{best} is totaled. For efficiency reasons, most chess programs generate pseudo-legal moves. The set of moves generated is described as being pseudo-legal since the set may include illegal moves which leave the king in check. Moves that are illegal are detected and removed during the search process. The number of pseudo-legal moves is a good estimate of the average branching factor since illegal moves are not generated for most positions. However, this estimate is slightly higher than the real branching factor due to the inclusion of the illegal moves. The third column in Table 4.1 indicates the total number of pseudo-legal moves for each test position. Using the data in the second and third columns, the average branching factor is determined to be approximately 32. An expanded version of the original method in [10] was implemented to obtain a more detailed view of the tree structure. The expanded version calculates the frequency with which a move in one of the first 10 positions is the best or is able to produce a cut-off. This produces a set of frequencies: $f_{best}(1), f_{best}(2), \ldots, f_{best}(10)$. Table 4.2 presents the first five values of f_{best} and Table 4.3 presents the last five. The term, $p_{best}(n)$, refers to the probability with which a move in a position, n, is the best and is obtained by dividing $f_{best}(n)$ by the number of nodes that contributed to $f_{best}(n)$ (Table 4.1).

For the trees searched by RajahX, the first branch has a very high probability ($p_{best}(1) = 0.878521$) of being the best or being able to produce a cut-off. On a perfectly ordered game tree $p_{best}(1) = 100$. The branch ordering techniques used by RajahX help it achieve something quite close to a perfectly ordered tree. In the next section, the challenge is to reproduce the behavior observed in RajahX using artificially generated trees. If the artificially generated trees are to be used as a model of the trees encountered in practice they should produce similar values for $p_{best}(1), p_{best}(2), \ldots, p_{best}(10)$.

Position	*Nodes Where Statistic is Valid*	*Pseudo-Legal Moves*
1	237	10427
2	76567	2212805
3	48817	1658962
4	80879	2435516
5	72022	2949829
6	353732	6885046
7	69248	2561168
8	231888	4255581
9	62636	2389516
10	83923	3187410
11	55373	2211242
12	97282	3935812
13	70682	2780559
14	79563	3017889
15	119038	4110521
16	81574	2944782
17	53895	2032046
18	57534	2452978
19	81410	3295496
20	96656	3421319
21	77374	2714680
22	46893	1849020
23	69700	2757646
24	23204	944301
Total	2090127	67014551
Average Branching Factor	$\frac{67014551}{2090127} = 32.06243$	

Table 4.1: A count of the nodes that contribute to f_{best} and average branching factor in the Bratko-Kopec test positions.

	Frequency With Which Move is Best or Produces a Cut-off				
Position	$f_{best}(1)$	$f_{best}(2)$	$f_{best}(3)$	$f_{best}(4)$	$f_{best}(5)$
1	206	10	9	5	1
2	63684	5366	3172	1388	757
3	42317	1811	1503	1285	554
4	67370	6305	2964	1370	771
5	62754	3324	2057	1209	634
6	335578	10165	3273	1113	658
7	60394	3029	1869	1165	946
8	205706	12945	4276	2331	2060
9	50785	4413	3051	1369	935
10	75596	3594	1481	985	659
11	46712	1898	1282	1660	1282
12	84100	4589	2007	1427	1197
13	56714	5664	2977	1615	990
14	66364	4673	2469	2032	1413
15	109542	4389	1892	937	508
16	69683	5139	1635	1312	1427
17	45531	2727	1991	1081	809
18	48190	3249	1406	1135	998
19	71547	3397	2003	1267	804
20	80469	4517	2670	2869	1834
21	71920	1892	786	877	581
22	42647	1854	739	478	297
23	58615	3478	2222	1961	1061
24	19796	1556	769	390	162
Total	1836220	99984	48503	31261	21338
$p_{best}(n)$	0.878521	0.047836	0.023206	0.014957	0.010209

Table 4.2: The first five values of f_{best} derived from the search of the Bratko-Kopec test positions.

	Frequency With Which Move is Best or Produces a Cut-off				
Position	$f_{best}(6)$	$f_{best}(7)$	$f_{best}(8)$	$f_{best}(9)$	$f_{best}(10)$
1	0	1	0	0	0
2	402	211	230	177	170
3	289	240	111	110	90
4	450	343	222	151	134
5	553	274	164	128	98
6	637	552	445	365	278
7	635	321	163	143	74
8	1155	763	614	509	417
9	560	334	230	176	159
10	313	198	115	108	109
11	777	440	286	169	138
12	941	588	335	214	154
13	520	335	275	228	167
14	750	317	239	172	132
15	272	245	193	145	96
16	792	375	230	155	157
17	594	300	168	106	88
18	512	354	263	216	223
19	623	280	187	189	142
20	1272	673	460	309	280
21	392	203	92	88	72
22	205	98	78	40	32
23	464	277	187	152	135
24	146	62	51	27	29
Total	13254	7784	5338	4077	3374
$p_{best}(n)$	0.006341	0.003724	0.002554	0.001951	0.001614

Table 4.3: The last five values of f_{best} derived from the search of the Bratko-Kopec test positions.

4.3.3 An Exponential Branch Ordering Model

Imagine a function whose sole purpose is to order the branches at a node in a best to worst order. Although such perfect ordering is impossible, the function makes an educated attempt at creating a best to worst order. There are two branch lists; one is unordered and the other is ordered. Initially, the unordered list is full and has b (branching factor) members. The ordered list is empty. Until the unordered list is empty, the function moves branches, one at a time, from the unordered list into the ordered list. Branches must be carefully selected from the unordered list since they are being moved into an ordered list. To aid the selection process, the function uses domain dependent knowledge to predict which of the remaining moves in the unordered list is the best. In the *exponential branch ordering* model, two weights, w_a and w_b, represent the effect of the domain dependent knowledge on the resultant ordering. When selecting the first move, the knowledge assisted selection mechanism is successful in picking the best move with a probability of $w_a/100$. For all other moves, the knowledge assisted selection mechanism is successful in picking the best remaining move from the unordered list with a probability of $w_b/100$. When the domain dependent knowledge is not good enough to select the next best move, the probability that the move picked is still the best is given by: $1/(b-n+1)$. This is the probability when a simple random pick is used to select the nth branch. The routine in Figure 4.4 produces an exponential branch ordering. Both the ordered and unordered list are stored within the *node.child* array. When selecting the first move, the ordered list is empty and the entire array stores the elements of the unordered list. When selecting branch n, the elements in the range $[1, n-1]$ are members of the ordered list and elements in the range $[n, b]$ are members of the unordered list.

An exponential branch ordering has the following properties:

- The ordering generated is controlled by three parameters: b, w_a and w_b. Parameter, b, represents the branching factor. Parameters, w_a and w_b, represent the effect of the knowledge component in the ordering function and are values in the range $[0, 100]$. In Figure 4.4, the code in the first path (lines 8 – 19) selects the branch with the best score among the remaining branches, while the second path (line 21) selects one of the remaining branches at random. The selected branch is then placed in position i. Parameters, w_a and w_b, are used to determine which of the two paths is taken.

- The first branch at a node is best with probability, p_1, given by:

$$p_1 = \frac{w_a}{100} + \left(1 - \frac{w_a}{100}\right)\frac{1}{b} \tag{4.1}$$

- A branch at position n, where $2 \leq n \leq b$, is best with probability, p_n, given by:

$$p_n = \left(1 - \frac{w_a}{100}\right)\left(1 - \frac{w_b}{100}\right)^{n-2}\frac{1}{b}\left(\frac{(b-n+1)w_b}{100} + \left(1 - \frac{w_b}{100}\right)\right) \tag{4.2}$$

Branch Position	*Branch is Best or Produces a Cut-off*	
	$f_{best}(n)$	$p_{best}(n)$
1	14732328	0.873821
2	893225	0.052980
3	416419	0.024699
4	249527	0.014800
5	154453	0.009161
6	86787	0.005148
7	60687	0.003600
8	48289	0.002864
9	29638	0.001758
10	26563	0.001576

Table 4.4: Values of f_{best} and p_{best} derived from the search of 100 artificially generated trees.

We now address the question of whether an exponentially ordered model is an adequate representation of the trees encountered in practice. In other words, can an exponentially ordered tree produce the same values for f_{best} and p_{best} as the trees searched by RajahX? Table 4.4 illustrates the values of f_{best} and p_{best} collected during an alpha-beta search of 100 artificially generated trees (generated using set tg_1). Each tree was searched to a depth of 7. The exponential ordering employed by the artificial tree generator used the parameters: $b = 32$, $w_a = 79$ and $w_b = 5$. Figure 4.5 compares the probability curve produced by RajahX (generated using Tables 4.2 and 4.3) with the probability curve produced by the artificial tree generator. Clearly, when the appropriate parameters are used, the trees produced by an exponential ordering closely resemble the trees searched by RajahX.

The exponential model can satisfy the strongly ordered model's criteria when supplied with the right parameters. A strongly ordered tree with a branching factor of 20 is produced when the parameters, $b = 20$, $w_a = 69$ and $w_b = 19$, are used. The data in Table 4.5 was generated using these parameters. The second column is the result of applying Equations 4.1 and 4.2 for various values of n; this column presents the probability with which a move in a position, n, is the best. A running total of the probabilities in the second column is maintained in the third column; this column presents the probability with which a move in one of the first n positions is the best. The probability of the first move being the best is 0.705500. This is slightly higher than the 0.70 required by the strongly ordered model. The best branch is within the first quarter of the branches at a node with a probability of 0.899916. This figure is slightly lower

BranchOrder($node$)

```
1:  for i ← 1 to node.child.length
2:      if i = 1 then
3:          w_i ← w_a
4:      else
5:          w_i ← w_b
6:      w ← RandomRange(node.rnd, 0, 99)
7:      if w < w_i then
8:          if node.type = max then
9:              v ← score_min − 1
10:             for j ← i to node.child.length
11:                 if node.child[j].pscore > v then
12:                     p ← j
13:                     v ← node.child[j].pscore
14:         else
15:             v ← score_max + 1
16:             for j ← i to node.child.length
17:                 if node.child[j].pscore < v then
18:                     p ← j
19:                     v ← node.child[j].pscore
20:     else
21:         p ← RandomRange(node.rnd, i, node.child.length)
22:     swap node.child[i].pscore ↔ node.child[p].pscore
```

Figure 4.4: A routine that produces an exponential branch ordering.

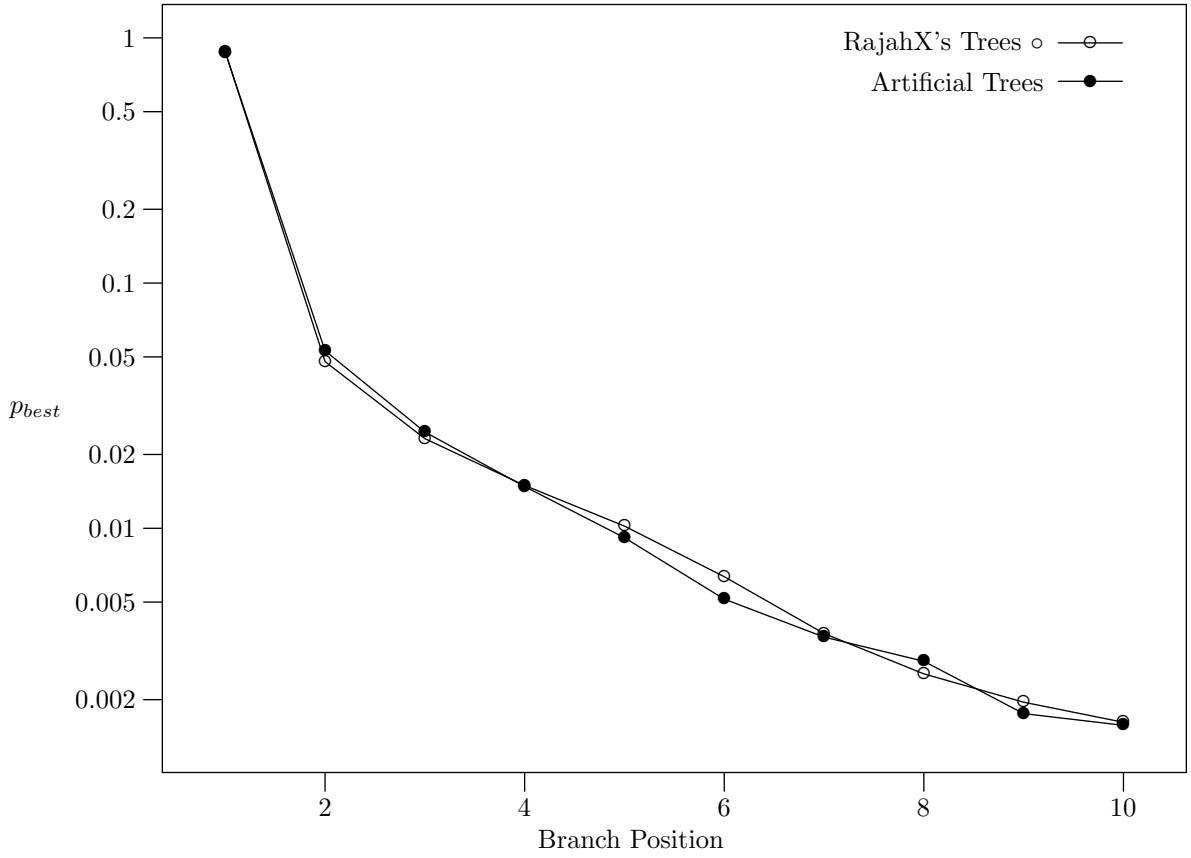

Figure 4.5: Comparing the values of p_{best} from RajahX with the values of p_{best} from artificially generated trees.

n	p_n	$p_1 + \ldots + p_n$
1	0.705500	0.705500
2	0.068510	0.774010
3	0.053108	0.827118
4	0.041085	0.868203
5	0.031714	0.899916
6	0.024420	0.924337
7	0.018754	0.943090
8	0.014359	0.957449
9	0.010957	0.968406
10	0.008329	0.976735
11	0.006305	0.983040
12	0.004749	0.987789
13	0.003557	0.991345
14	0.002646	0.993991
15	0.001953	0.995944
16	0.001428	0.997372
17	0.001032	0.998403
18	0.000734	0.999138
19	0.000513	0.999651
20	0.000349	1.000000

Table 4.5: An exponential ordering with parameters, $b = 20$, $w_a = 69$ and $w_b = 19$.

than the required value of 0.90. Highly precise values for w_a and w_b can be used to produce the exact values required by the strongly ordered model, but the routine in Figure 4.4 uses integer arithmetic and it only accepts integral values in the range $[0, 100]$ for w_a and w_b. This is only a minor limitation and it does not have a significant impact on the performance experiments to be described in Chapter 5 and 6.

The performance experiments in this work will consider five different branch orderings as generated by the following parameter sets:

1. $b = 32$, $w_a = 79$, $w_b = 5$.

2. $b = 20$, $w_a = 69$, $w_b = 19$.

3. $b = 32$, $w_a = 66$, $w_b = 66$.

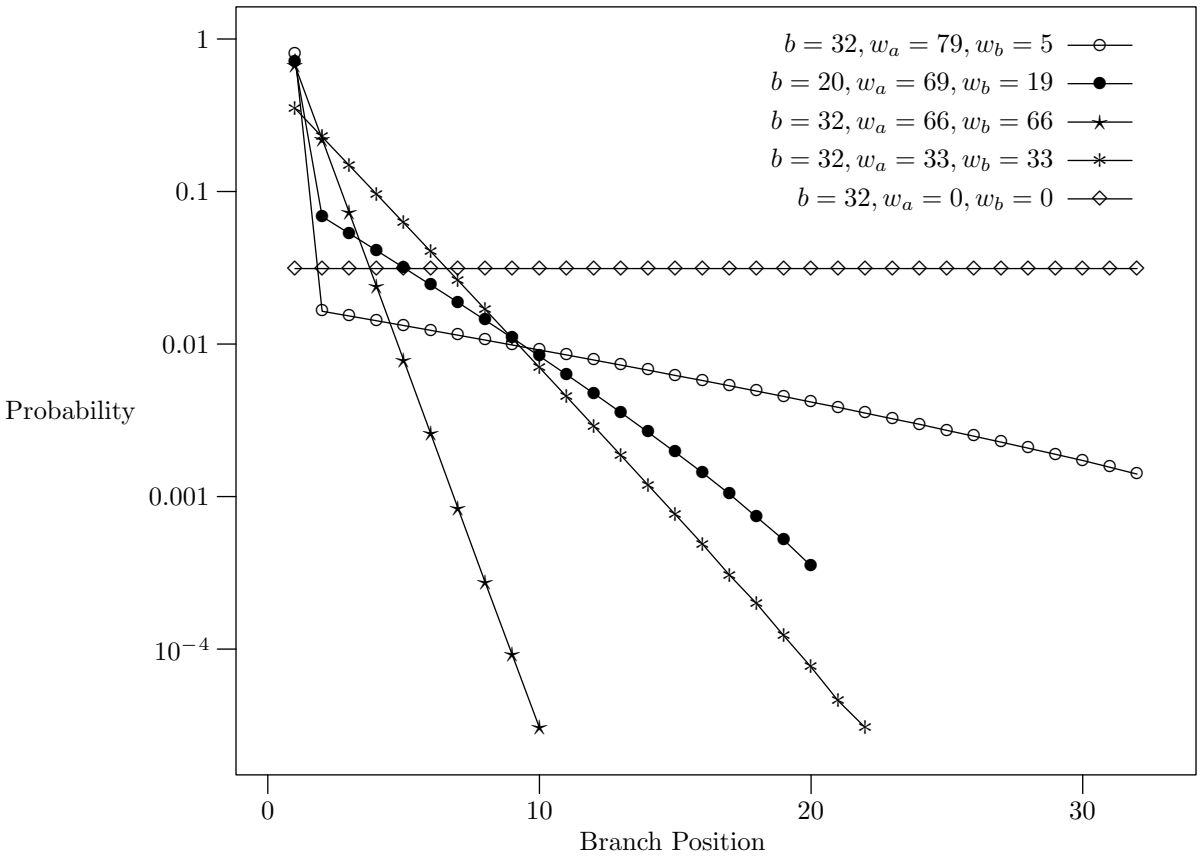

Figure 4.6: The probability curves generated by the five parameter sets used in the experiments.

4. $b = 32,\ w_a = 33,\ w_b = 33.$

5. $b = 32,\ w_a = 0,\ w_b = 0.$

The first parameter set generates trees that resemble the trees searched by RajahX, while the second generates strongly ordered trees. The last three parameter sets are designed to illustrate the effect of decreasing branch ordering accuracy. The fifth parameter set generates trees that are randomly ordered. Equations 4.1 and 4.2 can be used to compute the probability curves for the five branch orderings; the resulting curves are plotted in Figure 4.6.

Chapter 5

Neural Network Based Prediction

5.1 Introduction

In a parallel alpha-beta search, a split-point must be chosen carefully if large increases in search overhead are to be avoided. If a node that will be cut-off is selected as a split-point, useless work will be performed when many processors collaborate on the node. On the other hand, a node that examines all of its successors makes a perfect split-point. In addition, for efficiency reasons, a node should be selected as a split-point only when it is reasonably certan that its score will not change further.

Given a large set of training data, a simple feed forward neural network [7] can be taught to approximate a wide variety of functions. For the purposes of a parallel alpha-beta searcher, a feed-forward neural network can be taught to predict when a node's score is going to stabilize and whether a node is going to cut-off. A neural network that is capable of performing these two tasks is presented here. Its application to parallel alpha-beta search is examined using both sequential and parallel experiments.

5.2 Neural Network Inputs and Outputs

The neural network is called as a new child node is being generated. Three inputs are required by the network:

- *pv*: The value of this input is 1 if the parent node is among the first nodes to be expanded by the search (i.e. a PV node). It is 0 otherwise.
- *type*: This input represents the parent node's type and it is a value in the range $[0, 100]$.
- *index*: The position of the child being expanded among the other children at this node.

The network produces two outputs:

- *type*: Represents the child node's type and it is a value in the range $[0, 100]$.
- *wait*: This output represents the number of moves that need to be searched before the score at the child node stabilizes.

Strong YBWC (Section 3.4) classifies a node into one of three types. Furthermore, in YBWC, given a node and its type, one can determine the type for the node's children as well. The network presented here performs a similar task; given the parent's type, the network generates the child's type. Note that a node's type may be one of 101 different types. This generality is necessary to allow the neural net to discover interesting patterns in the training data.

A parallel alpha-beta searcher can wait until the number of moves specified by the output, *wait*, is searched sequentially before trying parallel search. However, using the output, *type*, is slightly more involved. If the value for a node's *type* falls within a range of numbers that are known to represent cut-off nodes, extra care can be used before the node is selected as a split-point or the node can be avoided all together.

5.3 Generating Training Data

A single piece of training data (also referred to as a *pattern*) consists of a group of input values and the expected output values for the given inputs. Generating training data is a three step process. First, a basic sequential alpha-beta search is modified to generate the following data at every node:

- $(p_1, p_2, \ldots, p_{n-1})$: Specifies the "address" of the node in the tree. Here, n is equal to the height of the tree being searched. Each p value specifies the move that was selected at the tree level indicated by the subscript. For example, the node labeled B in Figure 5.1 has the address: $(1, 2, 2)$. When addressing a node, if a particular level has not been reached as yet, the corresponding p value is set to 0. For example, the node labeled A has the address: $(2, 1, 0)$.
- t: A type value associated with the node. If a cut-off occurs at the node, then its type value is 50. If a node updates its score but does not cut-off, then its type value is 0. If a node does not update its score and does not cut-off, then its type value is 100.
- u: The index of the move at which the final score update occurred or the index of the move that produced a cut-off.

Values for $(p_1, p_2, \ldots, p_{n-1})$, t and u are collected as the 200 trees from set tg_2 are searched to a depth of 4. Since the data is collected by traversing several different trees, there may be

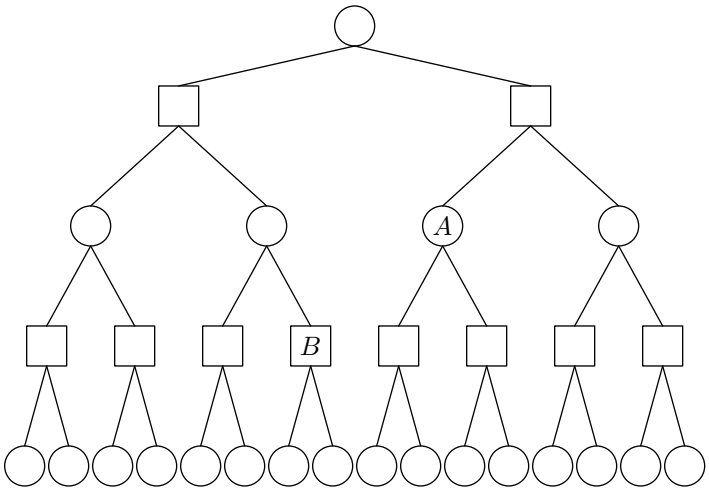

Figure 5.1: Node addressing.

several different values of t and u for a node address (i, j, k). This problem is rectified in the second step by computing the average value of t and u for each node address in the tree. The third and final step defines a mapping between adjacent levels in the tree. Let us consider how a mapping may be formed between the second and third levels in the tree. A node at the second level of the tree has an address of the form: $(i, 0, 0), i \neq 0$. The average values of t and u at this node are denoted $t_{avg}(i, 0, 0)$ and $u_{avg}(i, 0, 0)$, respectively. A node at the third level of the tree has an address of the form: $(i, j, 0), i \neq 0, j \neq 0$. A mapping between a parent at the second level and a child at the third level can be defined as follows:

$$(t_{avg}(i, 0, 0), j) \rightarrow (t_{avg}(i, j, 0), u_{avg}(i, j, 0)) \tag{5.1}$$

This mapping is computed for every possible value of i and j. The same process is applied to define a mapping between the nodes at the first and second levels and between the nodes at the third and fourth levels.

The data generated by the three-step process just described can be used as training data for a neural network. The task of the neural network is to discover the function that maps a pair of values at a parent into a pair of values at the child. For additional accuracy, a boolean value, pv, is also used as input to the function. This boolean value specifies whether the parent node is among the first nodes to be expanded by the search. Using the notation of Section 5.2, the task of the neural network is to discover the mapping that produces $(type, wait)$ as an output given $(pv, type, index)$ as an input.

Five sets of training data are generated, one for each of the five ordering types that are considered in this text (see Section 4.3.3).

5.4 Neural Network Structure and Back-Propagation

The structure of the neural network used is shown in Figure 5.2. It consists of three layers: an input layer (i_1, i_2, i_3, i_b), a hidden layer $(h_1, h_2, \ldots, h_{12}, h_b)$ and an output layer (o_1, o_2). The nodes, i_b and h_b, are *bias* neurons which always output 1. Except for the bias neurons and the neurons in the input layer, the output of any other neuron is given by the following equation:

$$f(\beta) = \frac{1}{1 + e^{-\beta}} \tag{5.2}$$

Figure 5.3 provides a plot of $f(\beta)$ for $-6 \leq \beta \leq 6$. If β is a large negative value, the function returns a value very close to 0. However, if β is a large positive value, the function returns a value close to 1. For the k-th neuron in the hidden layer, β is given by:

$$\beta = i_b wh_{k,b} + \sum_{j=1}^{3} i_j wh_{k,j} \tag{5.3}$$

Similarly, the value of β for the k-th neuron in the output layer is given by:

$$\beta = h_b wo_{k,b} + \sum_{j=1}^{12} h_j wo_{k,j} \tag{5.4}$$

A weight value, wh or wo, is associated with each input to a neuron and it is used to scale the input before it is summed. Note that each neuron has its own set of weights — one neuron's set of weights is not necessarily the same as another neuron's. The weights, $wh_{k,b}$ and $wo_{k,b}$, are used to scale the bias input.

The output of a neuron is a value between 0 and 1. In keeping with this convention, the inputs to the neural net are scaled and shifted to fit the range $[0.1, 0.9]$ before being placed in the input layer. Scaled and shifted versions of *pv*, *type* and *index* are placed in neurons i_1, i_2 and i_3 respectively. Similarly, the values in the output layer are scaled and shifted from the range $[0.1, 0.9]$ into the range required by the output variables.

To train the neural network, a set of input values is applied to the network. The network's outputs are compared to the expected outputs to determine the error associated with the outputs. The error associated with the k-th output neuron is given by:

$$e_k(p) = d_k(p) - o_k(p) \tag{5.5}$$

Here, $o_k(p)$ represents the output of the k-th output neuron for the input pattern p and $d_k(p)$ represents the desired output for the k-th output neuron. Error in the neural network output can be reduced by applying the back-propagation method [7] to adjust the weights in the hidden and output layers. The adjustment to the weights at the k-th neuron in the output layer is given by:

$$\Delta wo_{k,j} = \eta \gamma_k(p) h_j(p) \tag{5.6}$$

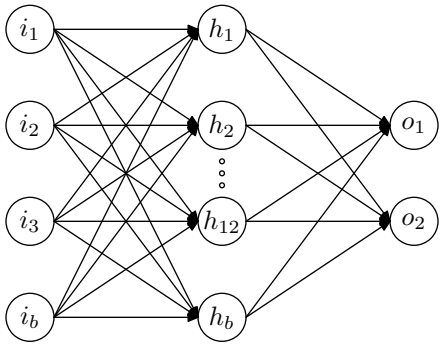

Figure 5.2: Neural network structure.

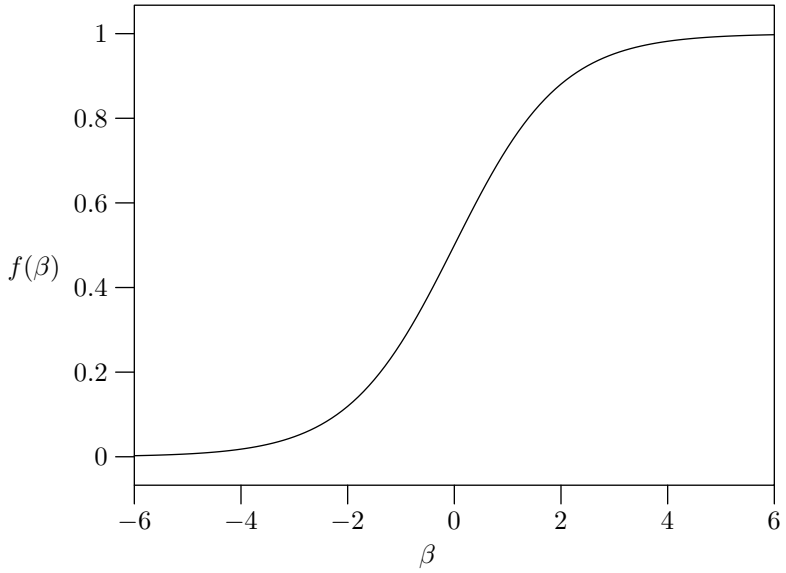

Figure 5.3: The threshold function used for each neuron in the neural network, $f(\beta) = \frac{1}{1+e^{-\beta}}$.

Here, η is the learning rate and $\gamma_k(p)$ is given by:

$$\gamma_k(p) = e_k(p)o_k(p)(1 - o_k(p)) \tag{5.7}$$

Similarly, the weights for the k-th neuron in the hidden layer are adjusted using:

$$\Delta wh_{k,j} = \eta\delta_k(p)i_j(p) \tag{5.8}$$

Although $\delta_k(p)$ and $\gamma_k(p)$ play similar roles, the computation of $\delta_k(p)$ is more involved:

$$\delta_k(p) = h_k(p)(1 - h_k(p)) \sum_{j=1}^{12} \gamma_j wo_{j,k} \tag{5.9}$$

Ordering (b, w_a, w_b)	E_{type}	E_{wait}
$(32, 79, 5)$	37.460133	0.633216
$(20, 69, 19)$	49.063758	0.309434
$(32, 66, 66)$	18.992261	0.073585
$(32, 33, 33)$	41.489635	0.297047
$(32, 0, 0)$	28.369211	4.095871

Table 5.1: Average squared error for the final neural network obtained after training.

A training iteration consists of feeding every training input to the network. Any error in the output is used to adjust the network weights. As the iterations progress, the average squared error in each output is measured. The average squared error for output k is given by:

$$E_k = \frac{1}{|P|} \sum_{p \in P} \frac{(d_k(p) - o_k(p))^2}{2} \tag{5.10}$$

Here, P is the set of all training patterns. One neural network was created for every type of ordering considered. Each network was trained over 10000 iterations and the learning rate, η, was set to 0.1 during the training. Table 5.1 summarizes the average squared error for the final network obtained after training.

5.5 Performance of Node Classification Schemes

Consider three different schemes that predict whether a node will cut-off or will examine all its successors. The first scheme, which will be referred to as scheme A, is taken from strong YBWC. A node in this scheme is classified as follows (Section 3.4):

- The root node is of type Y-PV. The first successor at a Y-PV node is of type Y-PV, while the rest of the successors are Y-CUT.
- The first successor to a Y-CUT node is a Y-ALL node, while the rest are Y-CUT nodes.
- All successors at Y-ALL nodes are Y-CUT nodes.

The second scheme, scheme B, is derived from the description of the DTS algorithm (Section 3.5):

- A node that has the same *alpha* and *beta* values as the *Root* is classified as D-PV.
- A minimizing node with the same *beta* as the *Root* or a maximizing node with the same *alpha* as the *Root* is classified as D-CUT.

- Any node that doesn't fit the first two categories is classified as a D-ALL node.
- If more than some number of moves, n, have been searched at a D-CUT node without an occurrence of a cut-off, then the type of the node is changed to D-ALL.
- Two D-ALL nodes are allowed at subsequent levels in the tree exactly once. Following the two D-ALL nodes, if they exist, nodes at subsequent levels are forced into an alternating sequence of D-CUT and D-ALL nodes.

Both scheme A and scheme B have types that correspond to the ALL and CUT types in a perfectly ordered game tree. Although PV nodes are named differently in each scheme, a node that would be classified as Y-PV in scheme A would be classified as D-PV in scheme B. Each scheme considers its PV type to be the first sequence of nodes expanded.

In scheme B a node may change from being a D-CUT node to a D-ALL node. In this case, only the initial prediction is used in determining performance. Note that when the D-CUT node changes, it affects the classification of nodes that are below it in the tree.

The final scheme, scheme C, is an application of the neural networks that were described in the preceding section. In this scheme, a node may be classified as being one of 101 types. The neural network generates a type value for a child node given the parent's type value and the location of the child among the parent's other children. If the type value for a node falls within a range $[50-m, 50+m]$ centered around 50, the node is defined to be an N-CUT node. If the type falls outside this range, the node is said to be an N-ALL node. If a node is among the first nodes to be expanded by the search, the neural network output is not used and the node is defined to be an N-PV node.

To compare the performance of schemes A, B and C, each scheme was used to categorize the internal nodes of 100 artificially generated trees. In each tree, the first sequence of nodes expanded (PV nodes) are not included in the performance measure because all three schemes generate the same type for these nodes. The trees were generated using set tg_1 and were searched to a depth of 6. Tables 5.2, 5.3 and 5.4 summarize the results of the experiments. In each table, the $True_{CUT}$ column indicates how often the prediction was correct for a CUT type node (Y-CUT, D-CUT or N-CUT). In other words, this column indicates how often a CUT type node was found to cut-off without having searched all of its branches. Similarly, the $True_{ALL}$ column indicates how often an ALL type node (Y-ALL, D-ALL or N-ALL) searched all of its branches. On the other hand, the $False_{CUT}$ column indicates the number of CUT nodes that did not cut-off and the $False_{ALL}$ column indicates the number of ALL nodes that did not search all of their branches. The *Error* column is obtained by summing the number of mispredicted CUT nodes and the number of mispredicted ALL nodes. When using scheme B, a value must be specified for parameter n. Recall that n is the number of moves after which a

Ordering (b, w_a, w_b)	$True_{CUT}$	$False_{CUT}$	$True_{ALL}$	$False_{ALL}$	*Error*
$32, 79, 5$	5224782	96205	384361	38844	135049
$20, 69, 19$	1466634	51794	153707	21592	73386
$32, 66, 66$	5263594	104381	333611	45230	149611
$32, 33, 33$	8066731	374809	487466	190667	565476
$32, 0, 0$	21170883	2346414	971304	1256668	3603082

Table 5.2: Performance of Scheme A.

D-CUT node changes into a D-ALL node. The second column in Table 5.3 specifies the value of n that was found to give the lowest prediction error for a given ordering. As in scheme B, scheme C requires a value for parameter m. Recall that this parameter defines a range of type values $[50-m, 50+m]$ that are used to check if a node is of the CUT type. The second column in Table 5.4 specifies the value of m that was found to give the lowest prediction error.

The lowest error, regardless of branch ordering, is exhibited by scheme C. Scheme A, which is based on strong YBWC, has the next highest error. The worst performance is exhibited by scheme B. Scheme B obtains its best results with $n = 1$. When $n = 1$, scheme B employs a rule that is quite similar to a rule used by scheme A. With $n = 1$, when one successor has been examined at a D-CUT node without an occurrence of a cut-off, the node's type is changed to D-ALL. A successor to this new D-ALL node will be of the D-CUT type if the rule regarding two consecutive D-ALL nodes is used. Note the similarity between this and scheme A's rule for Y-CUT nodes. In scheme A, the first successor to a Y-CUT node is of type Y-ALL while the rest are of type Y-CUT.

In Table 5.4, note that the last row has two entries that are 0. This means that for ordering $(32, 0, 0)$ scheme C does not use the N-ALL classification at all! Consider what happens in scheme A for ordering $(32, 0, 0)$. Scheme A places 2227972 nodes into the Y-ALL category. However, only 971304 were found to be true Y-ALL nodes. Scheme C takes the approach of eliminating the N-ALL category all together, thereby eliminating the problem of inaccurate classification. Although this may seem simplistic, this is a safer approach for a parallel alpha-beta searcher since a CUT node is subject to more stringent rules before it is picked as a site for parallel search.

Figure 5.4 presents 5 bar graphs comparing the performance of schemes A, B and C. Scheme B has a significantly higher error than either scheme A or C for all orderings considered. The performances of schemes A and C are relatively close to each other except for those orderings where w_b is of reasonable size. For example, when the ordering $(32, 66, 66)$ is considered, scheme

Ordering (b, w_a, w_b)	n	$True_{CUT}$	$False_{CUT}$	$True_{ALL}$	$False_{ALL}$	*Error*
$32, 79, 5$	1	4520268	172402	308164	743358	915760
$20, 69, 19$	1	1233943	73866	131635	254283	328149
$32, 66, 66$	1	4567304	132753	305239	741520	874273
$32, 33, 33$	1	6464164	393138	472117	1812923	2206061
$32, 0, 0$	1	16010177	2084414	1233304	6417374	8501788

Table 5.3: Performance of Scheme B.

Ordering (b, w_a, w_b)	m	$True_{CUT}$	$False_{CUT}$	$True_{ALL}$	$False_{ALL}$	*Error*
$32, 79, 5$	10	5196278	66924	413642	67348	134272
$20, 69, 19$	10	1453897	33895	171606	34329	68224
$32, 66, 66$	20	5270913	43527	394465	37911	81438
$32, 33, 33$	15	8011267	199923	665332	265820	465743
$32, 0, 0$	20	22427551	3317718	0	0	3317718

Table 5.4: Performance of Scheme C.

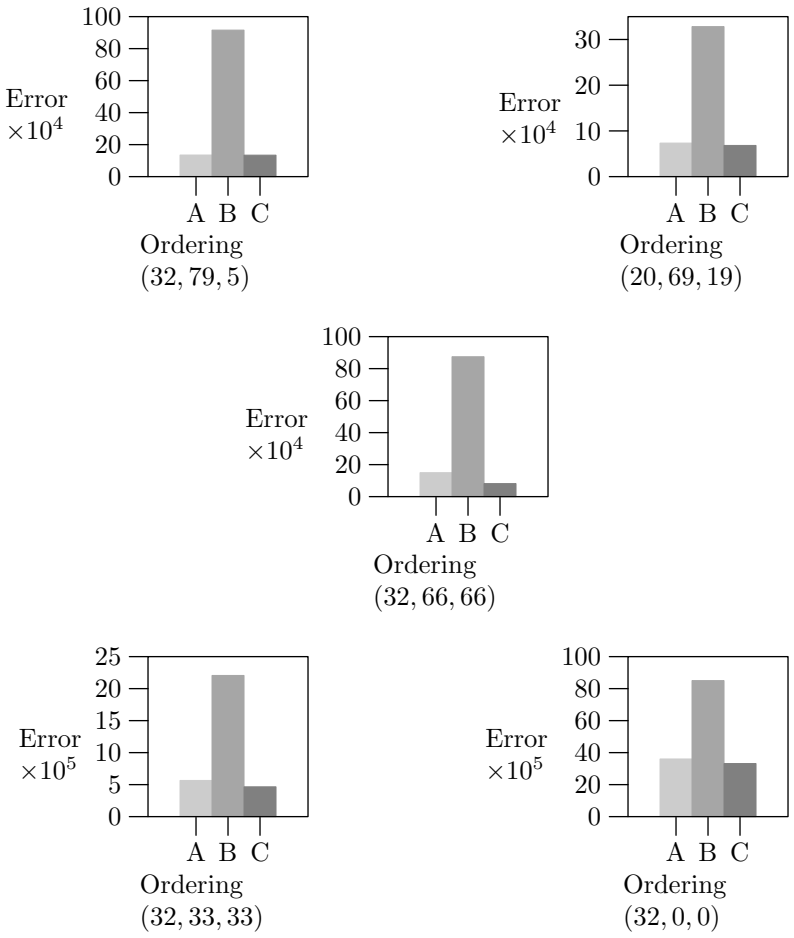

Figure 5.4: Comparing the performance of schemes A, B and C.

A produces an error value of 149611 while scheme C produces an error value of 81438. In this case, scheme C produced an error value that is over 40 percent lower than that produced by scheme A. In Figure 5.5, the improvement of C over A is plotted against the value of w_b that produced it. For higher values of w_b, scheme C dominates scheme A by a significant margin.

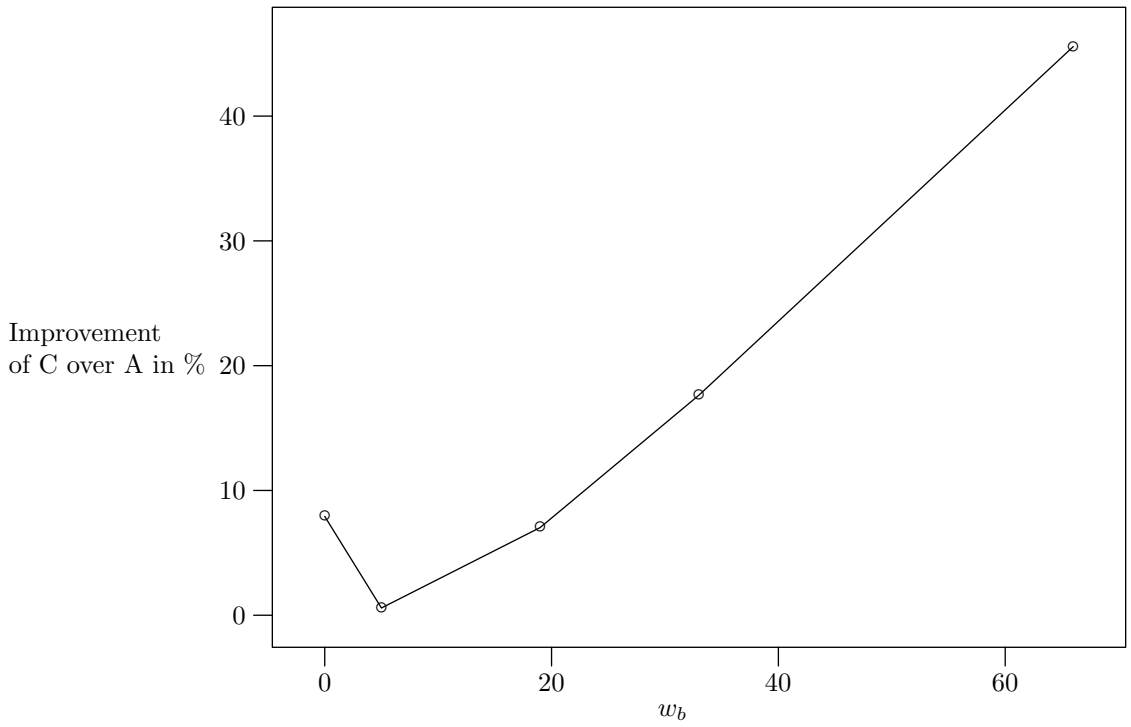

Figure 5.5: Comparing the performance improvement of scheme C over scheme A for various values of w_b.

Chapter 6

Experiments on a Parallel Alpha-Beta Simulator

6.1 Introduction

In Section 5.5, the performance of three different node classification schemes was considered within the framework of a sequential alpha-beta searcher. Here, the three schemes from that section are extended to create three split-point selection schemes whose performance is studied using a parallel alpha-beta simulator. The parallel alpha-beta simulator (PABSim) is a partial implementation of the DTS method described in Section 3.5 on a simplified shared memory multiprocessor system. The implementation is partial in the sense that there is no split-point selection mechanism in PABSim. It is the responsibility of the user to specify the details of the split-point selection mechanism. This facility will be used to compare the performance of the three split-point selection schemes.

6.2 A Simplified Shared Memory Multiprocessor

The simplified multiprocessor that is considered here has 48 processors. An illustration of the system is presented in Figure 6.1. Each processor has a portion of the total memory in the system. A cache is attached to each processor primarily to provide fast access to both local and remote memory. Each cache line is 128 bytes wide. Memory in the system is cache coherent and coherency is ensured using the 3-state MSI protocol in the caches along with a directory protocol [2] in the memory units. Details of how the processors are interconnected and the nature of the interconnection medium does not have a significant impact on the parallel alpha-beta searcher that is going to be described thus this issue will not be considered further. However, the timing of the various types of memory accesses is of great significance and this topic is treated next.

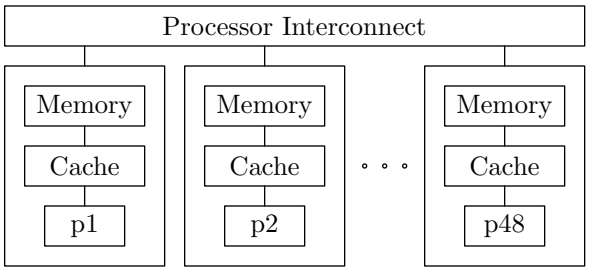

Figure 6.1: The simplified shared memory multiprocessor system.

Timing in the simplified multiprocessor system is expressed in *ticks*. The value of a tick is not important as long as all timing in the system is expressed in ticks. A tick is defined to be the length of time it takes for a block of memory (cache line) to return from the local memory unit.

A block in the cache may be in one of three states: modified (M), shared (S) or invalid (I). Furthermore, to ensure cache coherency, the home node has a flag indicating whether a block is clean or dirty along with a list of nodes that have a copy of the block. The delays associated with reading a block that is in one of several possible states is given in Table 6.1. For example, when reading a block that is either in state M or state S, the read can be performed directly from the cache and there is virtually no delay. If the block is in state I and is flagged as being clean at the home node, it can be read in 1 or 2 ticks. The read takes 1 tick if the block is local and 2 ticks if the block is from a remote node. However, if the block is flagged as being dirty, the block has to be read from the node that modified the block and extra delays are incurred. A set of delays for block writes is presented in Table 6.2. If a block is in state M in the cache, it can be modified without any delay. However if a block is in state S or state I, invalidations have to be sent to all processors that have a cached copy of the block.

The cache in the simplified multiprocessor is infinite in size. This simplification means that capacity and conflict misses do not occur in the caches. However, this assumption produces results that are drastically different from reality if the program being studied has a large working set. Fortunately, in the case of an alpha-beta search routine, the working set is quite small. A single block is usually enough to store all the data at a node. For example, a chess move is represented using 2 bytes of memory: 6 bits are used to represent the square that the piece is moving from, another 6 bits are used to represent the square that the piece is moving to and the remaining 4 bits are used as flags. On average, a chess position has 32 moves available, thus 64 bytes are enough to represent the move list at a node. A few extra bytes are required to hold the value of *alpha*, *beta* and any other temporary variables. If the height of the tree is 6 and the data at any node can be stored in a single block of memory, 6 blocks are enough to

Cache State (M/S/I)	*Home State (Dirty/Clean)*	*Home Node (Local/Remote)*	*Delay (ticks)*
M/S	Dirty/Clean	Local/Remote	0
I	Clean	Local	1
I	Clean	Remote	2
I	Dirty	Local	2
I	Dirty	Remote	3

Table 6.1: Delay for a memory read under various conditions in the simple multiprocessor.

Cache State (M/S/I)	*Home State (Dirty/Clean)*	*Home Node (Local/Remote)*	*Delay (ticks)*
M	Dirty/Clean	Local/Remote	0
S/I	Clean	Local	1
S	Clean	Remote	2
I	Clean	Remote	3
I	Dirty	Local	2
I	Dirty	Remote	4

Table 6.2: Delay for a memory write under various conditions in the simple multiprocessor.

store the data for the nodes along the path from the root to the leaf of the tree. Therefore, the working set is no more than 6 blocks.

6.3 Multiprocessor Simulation

Normally, a multiprocessor simulator and the program being run on the simulator are two separate entities. However, the approach that is taken in PABSim is to embed the multiprocessor simulation within the code that performs the parallel alpha-beta search. Before detailing the simulation process itself, a description of *fibers* [19] is given. Fibers are of primary importance to the simulation process because they provide the mechanism by which several processors are simulated on a uniprocessor system.

6.3.1 Win32 Fibers

Most operating systems today provide support for threads. A process running on such an operating system may create threads that are then scheduled by the operating system. All

threads have access to the data within the process. The Win32 subsystem that is found in Microsoft's Windows 98 and Windows NT operating systems provides support for threads. Starting with version 4.0 of the subsystem, support is also provided for an entity referred to as a fiber. Fibers are similar to threads except that they are scheduled by the user rather than the operating system. In a multiprocessor simulation, a fiber can be used to simulate the activity of a processor in the system. Fibers are more natural than threads for a multiprocessor simulation because precise control is required over the process of selecting which processor to simulate next.

Only three Win32 functions are needed to perform the basic fiber operations. A fiber can be created with a call to CreateFiber. The function takes three arguments: a value for the size of the stack used for the fiber, a pointer to a function where the fiber begins execution and a parameter to be passed to the function where execution begins. The function returns a handle to the newly created fiber. During cleanup, a fiber can be deleted with a call to DeleteFiber. A fiber that is executing can suspend itself and can schedule another fiber for execution with a call to SwitchToFiber. The function takes the handle of the fiber that is to be scheduled as an argument.

6.3.2 Simulation Support Routines

Five routines are defined to help create multiprocessor simulations. The first function, Yield, advances the simulation time for the fiber that calls the function and allows the execution of other fibers that are waiting to run. Scheduling of the other fibers is done in a round robin fashion. That is, if fiber 1 calls the Yield function, fiber 1 is suspended and fiber 2 is executed. In order to model accesses to memory blocks, two functions are provided: CRead and CWrite. A read request to a memory block can be simulated using a call to CRead and a write request to a memory block can be simulated using a call to CWrite. Both functions properly model the state transitions for the block in the cache and the home node. The functions also account for the delays associated with reading and writing a block as indicated in Tables 6.1 and 6.2. For example, if a fiber executes the CRead function to access a block that is set to state I in the cache, there will be delays in accessing the block. When such delays occur inside CRead or CWrite, a call is made to Yield to advance the simulation time for the fiber. Furthermore, the call to Yield suspends the fiber that initiated the read and another fiber is allowed to run. Both CRead and CWrite account for memory block contention. If one processor requests access to a memory block that is busy serving another processor's request, the request is rejected and the processor that initiated the request will have to try again. Although CRead and CWrite account for delays due to network traversal, they do not account for delays due to network contention. Recall that the network connecting the processors was left unspecified

in Section 6.2. There may be delays due to network contention depending on the network structure used. Both CRead and CWrite assume that they can access the network as soon as a memory request requires communication with a remote node. The last two support routines allow the creation of regions that are accessible to one fiber at a time. A lock is essentially a block of memory that is modified atomically. A test-and-set operation is used to create the lock entry function, LockEnter. The converse of the lock entry function is the LockLeave function that is used to release a lock.

To illustrate how these simulation support routines are used, a simple matrix multiplication example is examined next. The routine in Figure 6.2 multiplies two 64×64 matrices, a and b, to produce a third matrix, c. It is assumed that four processors are executing the routine illustrated with each processor's id (0, 1, 2 or 3) stored in *tid*. The rows in matrix c are divided evenly among the four processors. Each processor is responsible for determining the values to be placed in the rows assigned to it. All three matrices are shared by the four processors performing the computation. Now, consider using the simulation support routines to simulate the execution of the matrix multiplication routine on the simple multiprocessor system. In the simulation, the matrix multiplication routine is executed by four fibers. Each fiber executes the routine illustrated in Figure 6.3. The variables, r, c, i and t, fit into the same block of memory and are local to each processor. The array, *BlockT*, contains the blocks where these variables are stored. There is an entry in the array corresponding to each processor in the system. Since each of these blocks is accessed by a single processor, a call *CRead* or *CWrite* is not required each time r, c, i or t is accessed. However, the initial write request to r needs to be modeled since the block may not be within the cache of the processor requesting the write. After the initial write, the block will remain in the cache in the correct state and all accesses to it will not incur extra delays. Recall that the size of the cache is assumed to be infinite, thus the block will never be replaced. If each value in the matrix is a four byte integer, 32 entries fit into a single block of memory (recall that a block is 128 bytes wide). Assuming that the matrices are aligned to block boundaries, the row and column counters can be used to determine the block containing a matrix entry. The blocks corresponding to matrix entries in a, b and c are stored in *BlockA*, *BlockB* and *BlockC*, respectively. A read or write request is sent to the appropriate block as required (lines 7, 8 and 14). Note that a call to Yield is made when *time* reaches 10 (line 12). The assumption is that time proportional to a tick has elapsed when *time* reaches a count of 10. Consider what happens as four fibers execute the routine in Figure 6.3. If a fiber executes any read or write request that has an associated delay, execution of the fiber is suspended and another fiber is run. Similarly, if a fiber has executed a tick worth of operations, it is suspended and another fiber takes over the simulation system for a tick. In this manner the four fibers, each of which represents a processor in the multiprocessor system, execute on a

```
1:  for r ← 16 · tid to 16 · (tid + 1)
2:      for c ← 0 to 63
3:          t ← 0
4:          for i ← 0 to 63
5:              t ← t + a[r][i] · b[i][r]
6:          c[r][c] ← t
```

Figure 6.2: Parallel matrix multiplication.

uniprocessor system.

6.4 The Parallel Alpha-Beta Simulator (PABSim)

In the following description of PABSim, calls to the first three simulation support routines, Yield, CWrite and CRead, are omitted for clarity. These calls would never be present in any parallel program — they are only present in PABSim for simulation purposes. Furthermore, code that is used to generate the artificial tree is also omitted. Note that the artificial tree generation code in PABSim is identical to the one described in Section 4.2. The resulting description details the DTS-like parallel alpha-beta search procedure that forms the core of PABSim without the added clutter of simulation support and artificial tree generation code.

6.4.1 Node Structure

The node structure is fundamental to all routines in PABSim. Figure 6.4 illustrates the fields within the *NodeT* structure that is used to hold all node data. The first field, *lock*, is a lock variable that controls access to the node. A node may be accessed by several processors at the same time, therefore a lock is necessary to ensure that only one processor is allowed to modify the node at any time. The second field, *threads*, is an integer that indicates the number of threads working at the node. Recall that in DTS node ownership may transfer from one processor to another. A pointer to the parent node is maintained in *parent* so that the node's current owner knows where to return the score determined at the node when it is closed. When returning the score determined at a node to its parent, certain data structures have to be cleared at the parent so that the node isn't visited again. The integer field, *parentidx*, contains an index that specifies the position of the child among the other children at the parent. For example, if the current node is the fifth node expanded at the parent, *parentidx* would contain 4 (indexing starts at 0). A count of the moves that remain unexamined at a node is held in

```
1:  time ← 0
2:  CWrite(tid, BlockT[tid])
3:  for r ← 16 · tid to 16 · (tid + 1)
4:      for c ← 0 to 63
5:          t ← 0
6:          for i ← 0 to 63
7:              CRead(tid, BlockA[(64 · r + i)/32])
8:              CRead(tid, BlockB[(64 · i + r)/32])
9:              t ← t + a[r][i] · b[i][r]
10:             time ← time + 1
11:             if time = 10 then
12:                 Yield(tid, 1)
13:                 time ← 0
14:         CWrite(tid, BlockC[(64 · r + c)/32])
15:         c[r][c] ← t
```

Figure 6.3: Simulating matrix multiplication.

```
1:  record NodeT
2:      lock
3:      threads
4:      parent
5:      parentidx
6:      mvsleft
7:      mvsfini
8:      type
9:      alpha
10:     beta
11:     score
12:     branch[32]
13:     child[32]
```

Figure 6.4: The *NodeT* structure.

mvsleft. Note that a move that is currently being searched by a processor is not included in this count. A similar count for moves that have been completely searched is held in *mvsfini*. PABSim uses the minimax formulation as opposed to the negamax formulation. Recall that in negamax every node is of the maximizing type. In minimax, a distinction is made between a node that is of the minimizing type and a node that is of the maximizing type. A flag that specifies the node type is stored in *type*. The lower and upper bound at a node are stored in *alpha* and *beta*, respectively. At a minimizing node, the lowest value found so far is stored in *score*. The same field is used at a maximizing node to store the highest value found. A list of moves at a node is stored in the array, *branch*. As branches are expanded, the newly allocated node data structures are stored in *child*. Note that the highest branching factor at a node is assumed to be 32.

6.4.2 Node Allocation

Nodes are allocated and deallocated as the search of the tree progresses. The allocation and deallocation routines use two heaps of nodes: a global heap and a local heap. The global heap is shared by all processors and it initially contains $p \cdot d \cdot 2$ nodes. Here, p is the number of processors involved in the search and d is the depth of the tree being searched. Each processor also has a local heap of nodes. Although the local heaps are initially empty, the local heaps

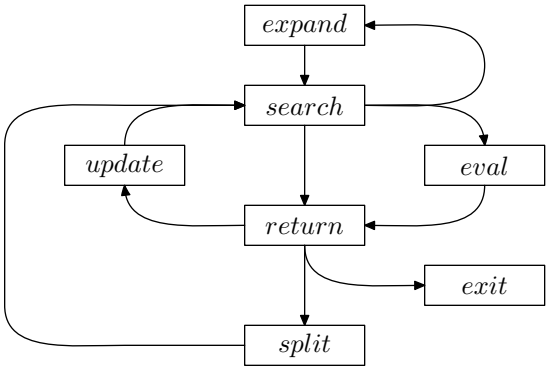

Figure 6.5: All possible state transitions for a processor performing parallel alpha-beta search.

can contain up to $d \cdot 2$ nodes. When a processor allocates a node, the local heap is checked for a free node before the global heap is used. The global heap is shared by all processors and access to it is controlled by a lock, thus it is faster to satisfy node allocation requests locally whenever possible. When a node is deallocated, it is returned to the local heap if possible. If the local heap has reached its maximum size, the node is returned to the global heap. At the start of the search, all allocations will use the global heap. However, as the search progresses, all node allocation requests will be satisfied locally.

6.4.3 Search Procedure

A processor that is participating in the parallel alpha-beta search procedure defined by PABSim may be in one of seven possible states: *expand*, *search*, *eval*, *return*, *split*, *update* and *exit*. Figure 6.5 illustrates all possible state transitions that can occur. Except for the last state, *exit*, there is a routine in PABSim corresponding to each of the six other states. These routines are referred to as *state routines*. Each processor also maintains a pointer to the "current" node. The target of all work performed by a processor is its current node. A processor may be viewed as a state machine whose behavior is governed by its state and its current node. Parallel search is then a result of several interacting state machines.

Figure 6.6 illustrates the *TDataT* data structure that is used to hold search data local to a processor. A pointer to the current node is stored in *node* and the processor's current state is stored in *state*. An additional field called *value* is used as a temporary data storage area when a processor switches states. A variable of type, *TDataT*, is passed into each state routine.

State: expand

The routine corresponding to the *expand* state is illustrated in Figure 6.7. When a node is initially created, its move list is empty. In the *expand* state, a call is made to a move generation

```
1: record TDataT
2:     node
3:     state
4:     value
```

Figure 6.6: The *TDataT* data structure.

```
Expand(tdata)

1: BranchGenerate(tdata.node)
2: tdata.state ← search
```

Figure 6.7: The *expand* state routine.

routine to create a valid move list. Since PABSim searches artificial trees, the move generation routine that it uses creates a set of seeds that are to be passed on to the child nodes (see Section 4.2). Once the move generation is complete, a processor that is in the *expand* state changes to the *search* state.

The node being expanded is already locked before the *expand* routine is executed. It is locked when it is initially created by *search* state. When the state changes from *expand* to *search*, the node remains locked.

State: search

The *search* state routine is given in Figure 6.8. At the start of the routine, two checks are performed to ensure that the node is searchable. The first check verifies that the node score is within the bounds at the node (lines 1–10) and the second check verifies that there are unassigned moves at the node (lines 11–13). If either of the two checks fail, a transition is made to the *return* state. If both checks pass, an unassigned move is expanded to create a new node. A node created by NodeAlloc is initially locked. This is to prevent other processors from entering the newly created node before the node's move list is populated. Note that when the call to NodeAlloc returns, the processor executing the *search* routine holds two locks: one lock controls access to the current node and the other lock controls access to the newly created node. Search now moves to the new node; the lock corresponding to the current node is released and the newly created node becomes the current node. If the new node is a leaf, the

processor's state changes from *search* to *eval*. However, if the new node is an internal node, the state changes to *expand*.

State: eval

The *eval* state routine is illustrated in Figure 6.9. In the *eval* state, a score is determined for the current node. Normally this would involve the execution of an application specific routine that computes a score for the node. This routine is usually a significant percentage of the total search time. However, since PABSim searches artificial trees, the score is available instantaneously. In PABSim, Evaluate does not return the score immediately but delays the processor by a specified amount of time before returning. Once evaluation is complete, the processor's state is changed to *return*.

State: return

In the *return* state, the current node has been completely searched and the score determined at the node is ready to be returned to its parent. The *return* state routine is presented in Figure 6.10. Recall that in DTS, a node is owned by the last processor that is left searching the node. The responsibility of returning the node's score to its parent falls on the node's owner. At the start of the *return* routine, a check is performed to see if the current processor is the last one at the node. If the current processor is not the last, the thread count is simply decremented and the processor enters the *split* state where it searches for another node to work on. However, if the current processor is the last one at the node, it begins the process of deallocating the node (lines 6–12). If the node being deallocated is the *root* node itself, then the search is complete and the processor enters the *exit* state. In addition, a global flag called *searchend* is set to indicate that the search of the tree is complete. For any other node, the node's score is placed within the processor's *value* field. The *value* field is used as temporary storage for a node's score before it is actually used in the *update* state. The current node is then deallocated, the parent node becomes the current node and the processor's state is changed to update.

State: update

In the *update* state, the search has just returned from the search of a child node. The child node's score is found in the processor's *value* field. Figure 6.11 shows the *update* state routine. If the score returned by the child is better than the score at the node, the score is updated. When an update occurs, the new score is actually a bound for the nodes lower in the tree. If the new score is a lower bound, TransLB is used to transmit the new bound to the subtree rooted at the current node. TransLB performs a depth first traversal on the subtree. At each

```
SEARCH(tdata)

1:  if tdata.node.type = max then
2:      if tdata.node.score ≥ tdata.node.beta then
3:          tdata.node.score ← tdata.node.beta
4:          tdata.state ← return
5:          return
6:  else
7:      if tdata.node.score ≤ tdata.node.alpha then
8:          tdata.node.score ← tdata.node.alpha
9:          tdata.state ← return
10:         return
11: if tdata.node.mvsleft = 0 then
12:         tdata.state ← return
13:         return
14: temp ← NODEALLOC(tdata.node)
15: tdata.node.mvsleft ← tdata.node.mvsleft − 1
16: LOCKLEAVE(tdata.node.lock)
17: tdata.node ← temp
18: if tdata.node.depth = 0 then
19:     tdata.state ← eval
20: else
21:     tdata.state ← expand
```

Figure 6.8: The *search* state routine.

```
EVAL(tdata)

1:  tdata.node.score ← EVALUATE(tdata.node)
2:  tdata.state ← return
```

Figure 6.9: The *eval* state routine.

```
Return(tdata)

1:  if tdata.node.threads > 1 then
2:      tdata.node.threads ← tdata.node.threads − 1
3:      LockLeave(tdata.node.lock)
4:      tdata.state ← split
5:  else
6:      if tdata.node = root then
7:          LockLeave(tdata.node.lock)
8:          tdata.state ← exit
9:          searchend ← true
10:     else
11:         tdata.value ← tdata.node.score
12:         tdata.node ← NodeDealloc(tdata.node)
13:         tdata.state ← update
```

Figure 6.10: The *return* state routine.

```
UPDATE(tdata)

1:  tdata.node.mvsfini ← tdata.node.mvsfini − 1
2:  if tdata.node.type = max then
3:     if tdata.value > tdata.node.score then
4:        tdata.node.score ← tdata.value
5:        if tdata.node.threads > 1 then
6:           TRANSLB(tdata.node)
7:  else
8:     if tdata.value < tdata.node.score then
9:        tdata.node.score ← tdata.value
10:       if tdata.node.threads > 1 then
11:          TRANSUB(tdata.node)
12: tdata.state ← search
```

Figure 6.11: The *update* state routine.

node, the new lower bound is compared to the existing bound. If the new bound is better, the node's bound is changed. A similar routine exists for transmitting upper bounds and it is called TRANSUB. Once the current node's score is updated, the processor enters the *search* state.

State: split

The *split* state is a state in which idle processors try to find a node where work is available. Figure 6.12 illustrates the *split* state routine. At the start of the routine, the global variable, *searchend* is checked to determine if the tree search is complete. If search is complete, the processor in the *split* state enters the *exit* state. However, if the search is not complete, the processor tries to find a split-point where the search effort can be shared. The SPLITPNTSEARCH routine can be used to search through the entire tree to find a split-point. However, SPLITPNTSEARCH is too expensive to execute each time a split-point is needed. The previous results of executing SPLITPNTSEARCH can be found in *splitpnt*. When a split-point is needed, the node in *splitpnt* is checked for validity before SPLITPNTSEARCH is invoked. A valid split-point is one that has some unassigned moves and one where the node score is within the node bounds. Note that even after executing *SplitPntSearch*, a valid split-point may not be found. In this case, the processor delays for some time before retrying.

Recall that we are trying to compare the performance of three different split-point selection

```
Split(tdata)

1:  for ever
2:      if searchend = true then
3:          tdata.state ← exit
4:          return
5:      LockEnter(splitpntlock)
6:      if SplitPntValid(splitpnt) = true then
7:          splitpnt.threads ← splitpnt.threads + 1
8:          tdata.node ← splitpnt
9:          tdata.state ← search
10:         LockLeave(splitpntlock)
11:         return
12:     splitpnt ← SplitPntSearch()
13:     if SplitPntValid(splitpnt) = true then
14:         splitpnt.threads ← splitpnt.threads + 1
15:         tdata.node ← splitpnt
16:         tdata.state ← search
17:         LockLeave(splitpntlock)
18:         return
19:     LockLeave(splitpntlock)
20:     Delay(rechecktime)
```

Figure 6.12: The *split* state routine.

schemes. In Figure 6.12, SplitPntSearch is actually a function pointer. The function that is executed changes depending on the split-point selection scheme being used.

State Timing

Apart from the delays due to memory access, lock entry and lock release, there are finite delays associated with the computation that takes place within each state. The *search* and *return* state each consume a tick of processor time. The *split* state is one tick in duration if a valid split-point is found immediately. However, if a split-point search routine is started, an extra tick of processor time is consumed for every node that is examined during the search. If the search does not find a split-point, another SplitPntSearch is not started until 36 ticks have

elapsed. The *update* state is one tick in duration if no score updates are performed. However, if an update has to be performed, an extra tick is spent for every child that has to be informed of the score change. The *expand* and *eval* state are 3 and 8 ticks in duration, respectively.

6.5 Split-Point Selection Schemes

Three different split-point selection schemes are considered. The first scheme, A, is based on the YBWC search procedure. Node classification is identical to the one described in Section 3.4. In scheme A, a node can be selected as a split-point only if it meets the following criteria:

- At least one branch at the node must have been examined.
- If the node is of the Y-CUT type then at least m_A branches must have been examined. Here, m_A is a value that is varied during the performance experiments.

At any point in the search, there will be several nodes in the tree that meet this criteria. Three guidelines are used to determine which node gets selected as the split-point. In order of decreasing priority, the guidelines are as follows:

1. A node that is higher up in the tree is preferable as it represents more work.
2. A node that has many branches already examined is a good split-point since the score at the node may have stabilized. This guideline is not applicable once more than a quarter of the moves at a node have been examined. For example, on a tree with a branching factor of 32, a node that has 4 branches examined is preferable to one that has 3 branches examined. However, a node that has 10 branches examined is considered equally good as a node that has 11 branches examined.
3. A node that has several unexamined branches makes a good split-point as a lot of parallel work is available at the node.

Scheme B is based on the DTS search procedure. Nodes in the tree are classified according to the scheme described in Section 3.5. In selecting a split-point, scheme B uses guidelines 1 and 3 from scheme A. However, guideline 2 is replaced by a new guideline that favors nodes with higher confidence. At a D-PV or D-ALL node, confidence is measured by the number of moves examined. As the number of moves examined increases, the confidence that the node is a D-PV or D-ALL node increases. The number of moves examined is also used at a D-CUT node as a measure of confidence. However, at a D-CUT node, the confidence decreases as more moves are examined. Once more than m_B moves have been examined at a D-CUT node, the node's type is changed from D-CUT to D-ALL and the confidence that it is a D-ALL node

begins increasing. A node can be selected as a split-point only if it is a D-PV or a D-ALL node with a confidence of at least 1. Furthermore, in a manner similar to guideline 2 in scheme A, the confidence measure is not applicable if the nodes being compared have a confidence value that exceeds $b/4$ on a tree with a branching factor of b.

The third scheme, C, uses the neural network described in Section 5.4 to classify the nodes in the tree. A node's type is determined using the procedure described in Section 5.5. The range used to determine if a node is of the N-CUT type is identical to that used in Section 5.5. In addition to the value used for node classification, the neural network also provides a constraint, w, on the number of branches that must be searched sequentially before the node can be selected as a split-point. For greater flexibility, two variables, $m_{C,a}$ and $m_{C,c}$, are used to adjust the constraint that the neural network produces. At all N-ALL and N-PV nodes, $w+m_{C,a}$ branches have to be searched sequentially before the node may become a split-point. At a N-CUT node, the constraint is increased even higher — $w+m_{C,a}+m_{C,c}$ nodes have to be searched sequentially before parallel search is possible. When a few processors are used for the parallel search, $m_{C,a}$ and $m_{C,c}$ are usually large; greater accuracy in split-point selection is obtained at the cost of reduced parallelism. However, when a large number of processors are participating in the search, $m_{C,a}$ and $m_{C,c}$ are set to a small value; split-point selection accuracy is reduced but the parallelism that is available is increased. From the set of nodes that pass these constraints, the node that is finally selected as the split-point is based on the same set of guidelines as in scheme A.

6.6 Performance of Split-Point Selection Schemes

The performance of the three split-point selection schemes, A, B and C, was compared using the trees generated by set tg_1. The results reported here are totals for the 100 trees generated by the set. During the performance experiments, variables, m_A, m_B, $m_{C,a}$ and $m_{C,c}$, were varied to find the values at which the highest speed-up was obtained. The figures reported here correspond to the highest speed-up obtained for each scheme.

6.6.1 Performance Measures

Before introducing the measures used to compare the performance of the split-point selection schemes, a few definitions are introduced. On a p processor system, the time taken to search the 100 trees generated by tg_1 is denoted t_p. On the same set of trees, the number of nodes examined by a p processor search is denoted n_p.

Three measures are used to compare performance. The first measure, *speed-up*, is the

traditional metric used to judge the efficiency of a parallel program relative to a sequential one:

$$SU_p = \frac{t_1}{t_p} \tag{6.1}$$

A parallel search usually examines a larger tree compared to a sequential search. The *search overhead* associated with a p processor search is given by:

$$SO_p = \frac{n_p}{n_1} - 1 \tag{6.2}$$

Ideally, each processor participating in a parallel search spends all of its time searching the tree. However, there are several obstacles that prevent the realization of this ideal behavior. Communication is necessary in PABSim to ensure that each processor has the latest bounds. There are synchronization overheads associated with lock entry and release. Furthermore, the process of selecting a split-point for an idle processor adds extra overheads. These three overheads are grouped into a single measurement referred to as *communication-synchronization-split* (*CSS*) overhead. On a shared memory multiprocessor, communication and synchronization costs are usually small. Much of the *CSS* overhead will be due to the time spent searching for a split-point. The *CSS* overhead can be approximated as the extra effort required to evaluate each node in a parallel search:

$$CSS_p = \frac{t_p p/n_p}{t_1/n_1} - 1 = \frac{p}{SU_p(SO_p - 1)} - 1 \tag{6.3}$$

Note that the *CSS* overhead should not be used as a figure of merit without taking *SU* and *SO* into account. For example, a parallel search that spends most of its time evaluating subtrees that are usually cut-off will have a small *CSS* overhead. However, this particular search will have a small speed-up and a large search overhead.

6.6.2 Uniprocessor Search

Table 6.3 presents the results of a uniprocessor search on the trees generated by tg_1. These results will be used in evaluating SU_p, SO_p and CSS_p as p takes on values in the set $\{12, 24, 48\}$.

6.6.3 Trees With Ordering $(32, 79, 5)$

The performance of schemes A, B and C under ordering $(32, 79, 5)$ is presented in Table 6.4. Figure 6.13 compares the performance of the three schemes on the basis of speed-up, search overhead and *CSS* overhead. Schemes A and B perform similarly on all three measures. When using 24 and 48 processors, Scheme C's behavior is remarkably different from that exhibited by schemes A and B. With 24 processors, scheme C has a lower speed-up. Furthermore, it has a higher search overhead as well as a higher *CSS* overhead. The higher search overhead can

Ordering	t_1 (*ticks*)	n_1 (*nodes*)
$32, 79, 5$	220380112	22123280
$20, 69, 19$	59604402	6034410
$32, 66, 66$	202468590	20495914
$32, 33, 33$	411967993	40776051
$32, 0, 0$	1855606486	178053342

Table 6.3: Performance of alpha-beta search on a single processor.

be interpreted as a reduction in split-point selection accuracy and the higher *CSS* overhead can be interpreted as an increase in the time spent performing communication and split-point selection. With 48 processors, scheme C has a much higher speed-up and an even higher search overhead. However, its *CSS* overhead is significantly lower than the *CSS* overhead calculated for schemes A and B. Here, scheme C keeps all processors busy by reducing the split-point selection accuracy. Although there is increased inaccuracy in the split-point selection, there is also an increase in the useful work performed by the processors participating in the search.

6.6.4 Trees With Ordering $(20, 69, 19)$

Table 6.5 presents the performance of the three split-point selection schemes under the ordering $(20, 69, 19)$. Figure 6.14 compares the speed-up, search overhead and *CSS* overhead observed for the three schemes. Although schemes A and B perform similarly on the speed-up measurement, there is a difference in the way in which they achieve their individual speed-up values. Scheme A spends less time selecting a split-point and searches more nodes, whereas scheme B spends more time in split-point selection and examines fewer nodes. For both schemes, the speed-up with 48 processors is less than the speed-up obtained with 24 processors. With 48 processors, both schemes exhibit a large *CSS* overhead. As described in Section 6.6.1, *CSS* overhead is primarily due to the time spent searching for a split-point. Using a modified version of PABSim that provides extra timing information, the *split* state is found to account for over 60 percent of the total search time using scheme A. When using scheme B, the *split* state is found to account for over 70 percent of the total search time. The *split* state has effectively become the limiting factor in the parallel search for the following reasons:

- More processors are actively searching for a split-point.
- There are fewer split-points as a result of the smaller tree size and the constraints imposed by the schemes A and B.

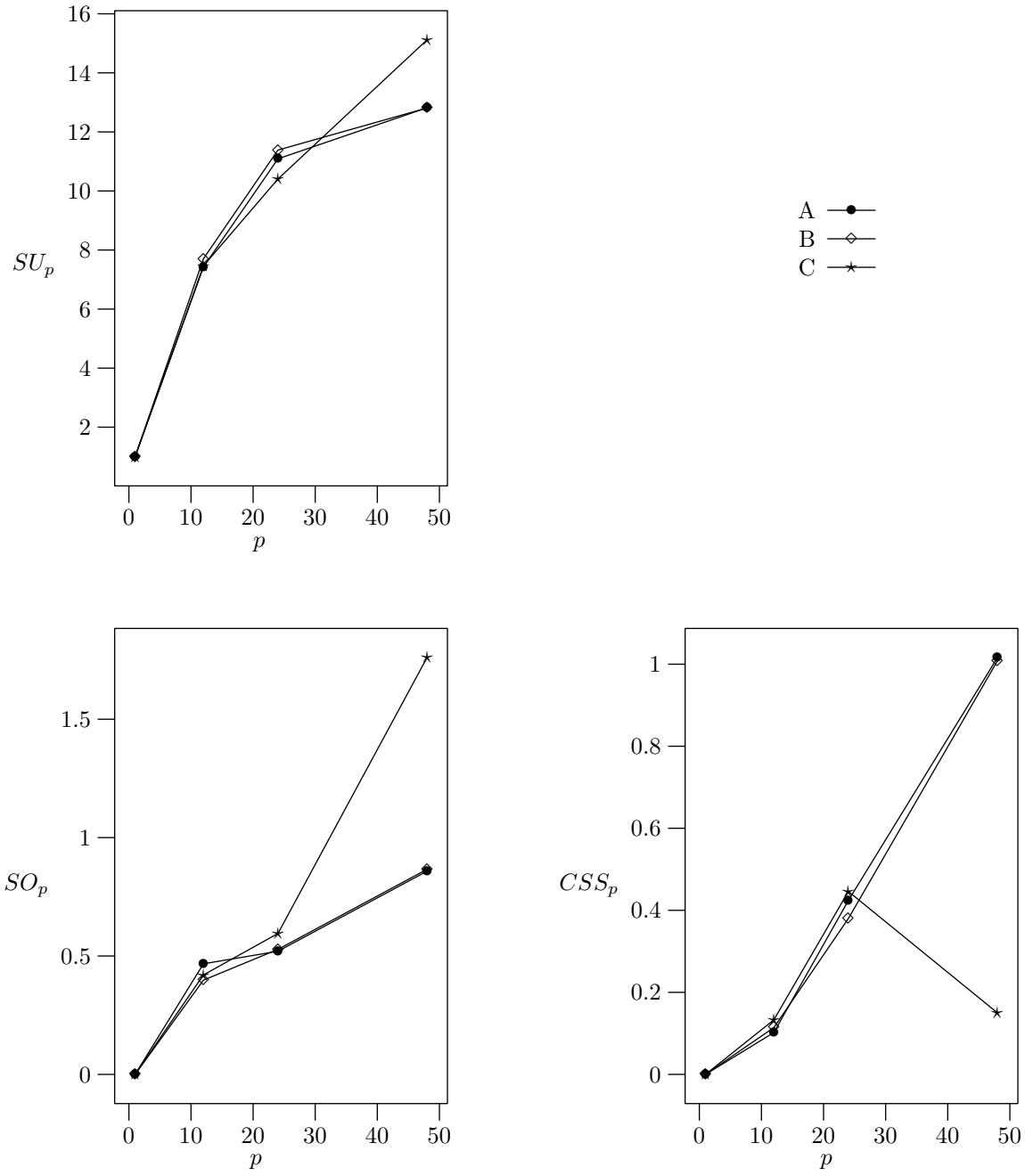

Figure 6.13: Performance of schemes A, B and C under ordering $(32, 79, 5)$.

Scheme A						
p	m_A	t_p (*ticks*)	SU_p	n_p (*nodes*)	SO_p	CSS_p
12	4	29687828	7.423	32457819	0.467	0.102
24	5	19873555	11.089	33626988	0.520	0.424
48	2	17194575	12.817	41103307	0.858	1.016
Scheme B						
p	m_B	t_p (*ticks*)	SU_p	n_p (*nodes*)	SO_p	CSS_p
12	6	28601791	7.705	30927242	0.398	0.114
24	5	19356414	11.385	33797226	0.528	0.380
48	1	17202647	12.811	41280623	0.866	1.008
Scheme C						
p	$m_{C,a}, m_{C,c}$	t_p (*ticks*)	SU_p	n_p (*nodes*)	SO_p	CSS_p
12	$-1, 3$	29539208	7.461	31415747	0.420	0.133
24	$-1, 3$	21180441	10.405	35290701	0.595	0.446
48	$-5, 3$	14579205	15.116	61063962	1.760	0.150

Table 6.4: Performance of schemes A, B and C under ordering $(32, 79, 5)$.

With 12 and 24 processors, scheme C has similar speed-up values as schemes A and B. However, when searching with 48 processors, scheme C performs significantly better than schemes A and B. To achieve this speed-up, scheme C trades split-point selection accuracy in favor of keeping its processors busy.

6.6.5 Trees With Ordering $(32, 66, 66)$

The performance of schemes A, B and C under ordering $(32, 66, 66)$ is presented in Table 6.6. A comparison of the speed-up, search overhead and *CSS* overhead exhibited by each scheme is presented in Figure 6.15. The performance of schemes A and B on all three measures is similar. Scheme C outperforms both scheme A and scheme B on the speed-up measurement. When searching with 12 processors, scheme C achieves a small search overhead at the cost of a small increase in *CSS* overhead. With 24 processors, the search overhead is still quite small but the increase in *CSS* overhead is much larger. When searching with 48 processors, scheme C reduces split-point selection accuracy to keep all its processors busy.

Scheme A						
p	m_A	t_p (*ticks*)	SU_p	n_p (*nodes*)	SO_p	CSS_p
12	3	9126291	6.531	9164428	0.519	0.210
24	2	6785067	8.785	10052455	0.666	0.640
48	1	8459246	7.046	13424461	1.225	2.062
Scheme B						
p	m_B	t_p (*ticks*)	SU_p	n_p (*nodes*)	SO_p	CSS_p
12	4	8828227	6.752	8710874	0.444	0.231
24	2	6751780	8.828	9168454	0.519	0.789
48	1	8695562	6.855	10482967	0.737	3.031
Scheme C						
p	$m_{C,a}, m_{C,c}$	t_p (*ticks*)	SU_p	n_p (*nodes*)	SO_p	CSS_p
12	$-1, 3$	9298183	6.410	8803379	0.459	0.283
24	$-3, 3$	6699466	8.897	14298996	1.370	0.138
48	$-4, 3$	5331472	11.180	17930200	1.971	0.445

Table 6.5: Performance of schemes A, B and C under ordering $(20, 69, 19)$.

Scheme A						
p	m_A	t_p (*ticks*)	SU_p	n_p (*nodes*)	SO_p	CSS_p
12	4	31659621	6.395	35071150	0.711	0.097
24	3	18647768	10.858	35806782	0.747	0.265
48	3	14933821	13.558	29842937	0.456	1.432
Scheme B						
p	m_B	t_p (*ticks*)	SU_p	n_p (*nodes*)	SO_p	CSS_p
12	5	31281023	6.473	35130206	0.714	0.082
24	3	18527219	10.928	35449416	0.730	0.270
48	2	14879228	13.607	29993951	0.463	1.410
Scheme C						
p	$m_{C,a}, m_{C,c}$	t_p (*ticks*)	SU_p	n_p (*nodes*)	SO_p	CSS_p
12	$0, 3$	23623744	8.571	24429533	0.192	0.175
24	$0, 3$	16897659	11.982	25056267	0.223	0.638
48	$-3, 3$	12389182	16.342	51026145	1.490	0.180

Table 6.6: Performance of schemes A, B and C under ordering $(32, 66, 66)$.

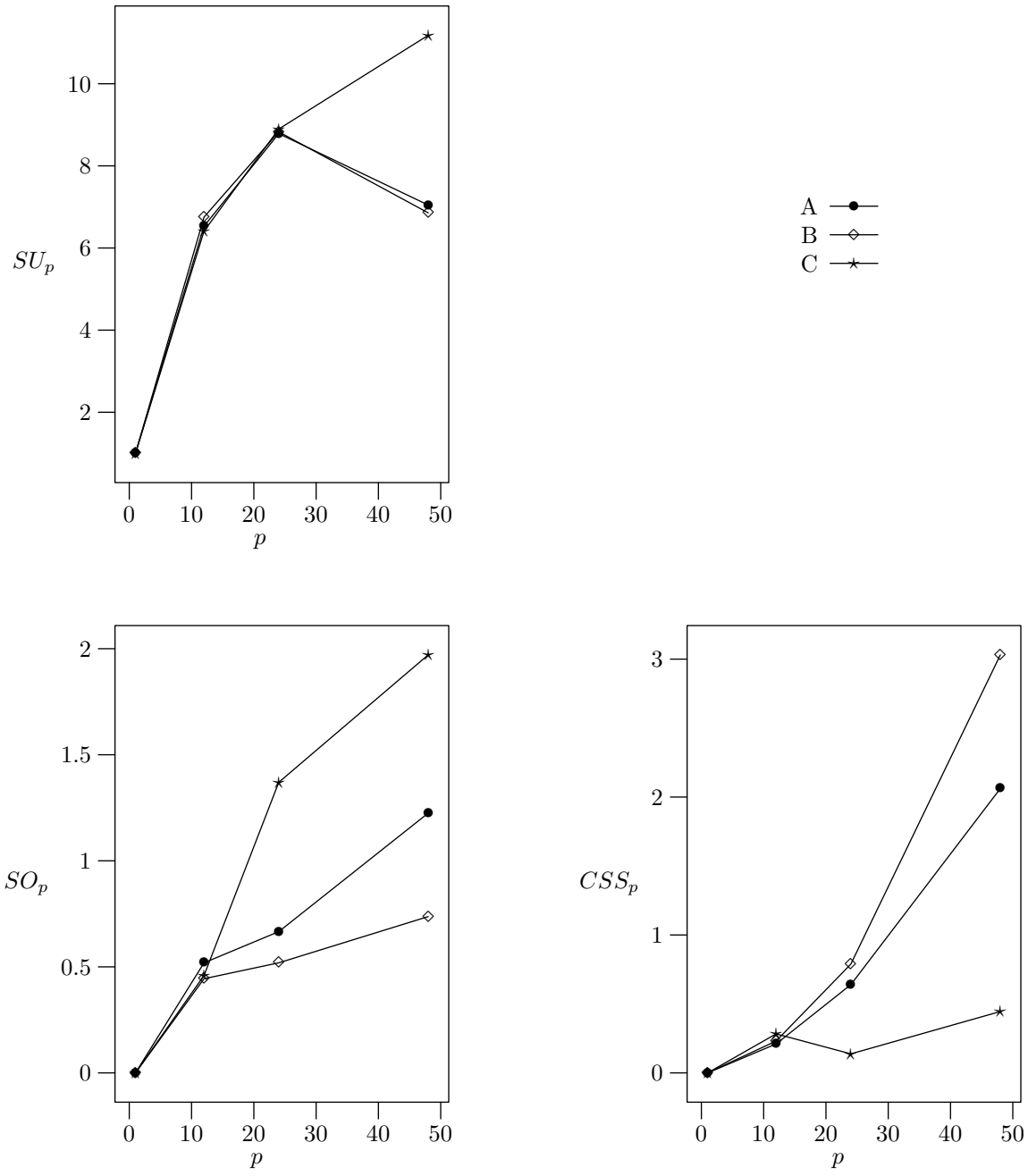

Figure 6.14: Performance of schemes A, B and C under ordering $(20, 69, 19)$.

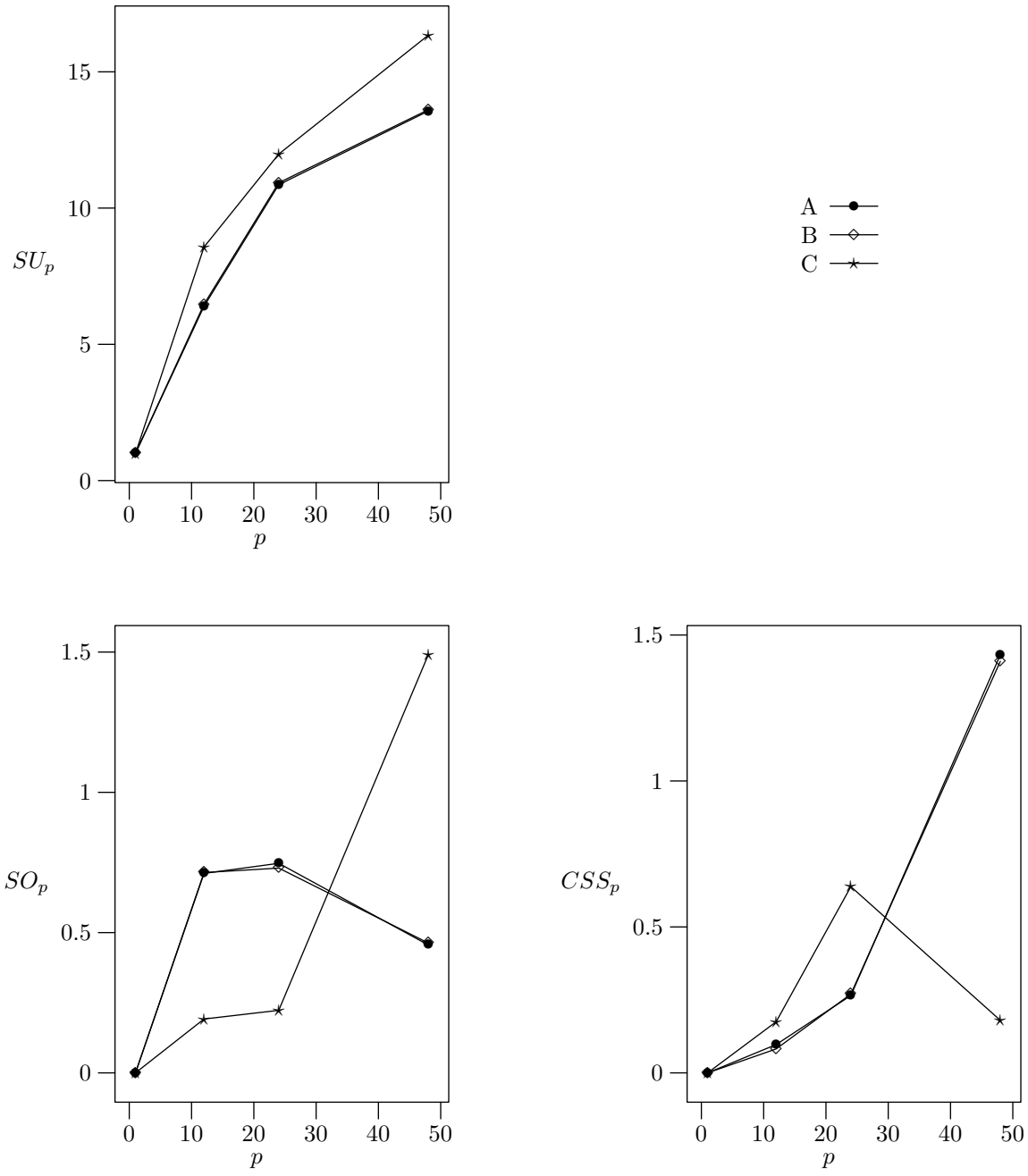

Figure 6.15: Performance of schemes A, B and C under ordering $(32, 66, 66)$.

Scheme A						
p	m_A	t_p (*ticks*)	SU_p	n_p (*nodes*)	SO_p	CSS_p
12	6	65197824	6.319	72018499	0.766	0.075
24	6	37510255	10.983	76283377	0.871	0.168
48	3	26194285	15.727	77506026	0.901	0.606
Scheme B						
p	m_B	t_p (*ticks*)	SU_p	n_p (*nodes*)	SO_p	CSS_p
12	7	64122450	6.425	72502627	0.778	0.050
24	5	37225749	11.067	77679512	0.905	0.138
48	2	26066328	15.805	77601935	0.903	0.596
Scheme C						
p	$m_{C,a}, m_{C,c}$	t_p (*ticks*)	SU_p	n_p (*nodes*)	SO_p	CSS_p
12	$0, 1$	55947868	7.363	57645755	0.414	0.153
24	$-1, 1$	37256277	11.058	73518488	0.803	0.204
48	$-3, 1$	23038327	17.882	99488108	1.440	0.100

Table 6.7: Performance of schemes A, B and C under ordering $(32, 33, 33)$.

6.6.6 Trees With Ordering $(32, 33, 33)$

Table 6.7 presents the performance of schemes A, B and C under the ordering $(32, 33, 33)$. Figure 6.16 compares the performance of the three schemes on the basis of speed-up, search overhead and *CSS* overhead. Once again, schemes A and B exhibit a similar performance on all three measures. Scheme C has a higher speed-up relative to schemes A and B when searching with 12 and 48 processors. With 24 processors, all three schemes exhibit a similar speed-up. When searching with 12 and 24 processors, scheme C favors a higher split-point selection accuracy at the cost of a higher *CSS* overhead. When searching with 48 processors, it favors a lower *CSS* overhead so that all the processors can be kept busy.

6.6.7 Trees With Ordering $(32, 0, 0)$

The performance of the schemes A, B and C under the ordering $(32, 0, 0)$ is presented in Table 6.8. Figure 6.17 compares the performance of the three schemes on the basis of speed-up, search overhead and *CSS* overhead. All three schemes exhibit similar speed-ups. Compared to schemes A and B, there is a significant difference in the way scheme C achieves its speed-up results. With 12 and 24 processors, scheme C favors an increased split-point selection accuracy at the cost of increased *CSS* overhead. With 48 processors, the scenario is reversed. Split-point

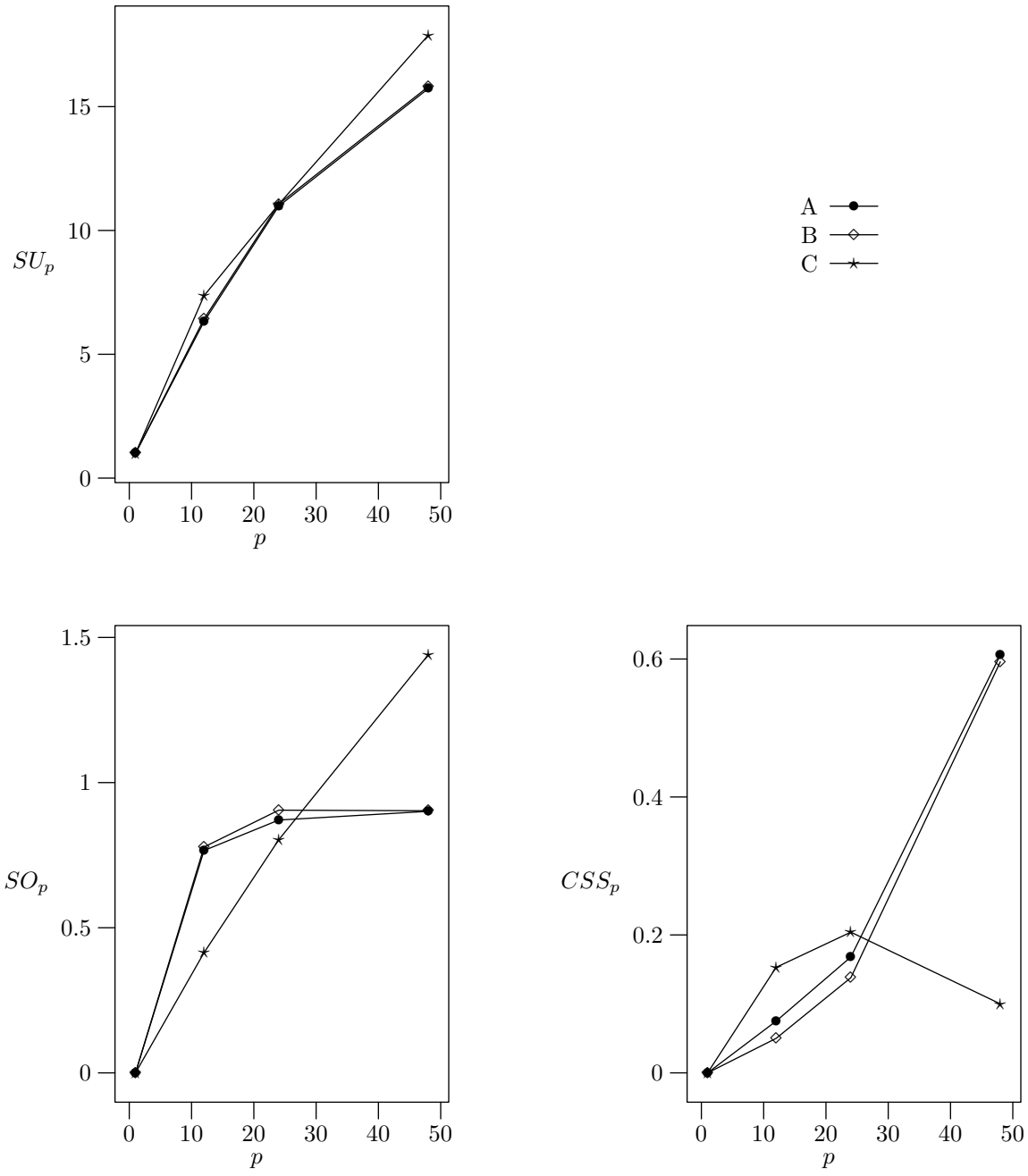

Figure 6.16: Performance of schemes A, B and C under ordering $(32, 33, 33)$.

Scheme A						
p	m_A	t_p (*ticks*)	SU_p	n_p (*nodes*)	SO_p	CSS_p
12	6	260414822	7.126	292745638	0.644	0.024
24	4	161551036	11.486	359652466	1.020	0.034
48	5	94042385	19.732	371461151	1.086	0.166
Scheme B						
p	m_B	t_p (*ticks*)	SU_p	n_p (*nodes*)	SO_p	CSS_p
12	8	258220759	7.186	291397171	0.637	0.020
24	6	158327672	11.720	349871306	0.965	0.042
48	4	93253513	19.899	372052246	1.090	0.154
Scheme C						
p	$m_{C,a}, m_{C,c}$	t_p (*ticks*)	SU_p	n_p (*nodes*)	SO_p	CSS_p
12	$-5, 0$	246839068	7.517	272691007	0.532	0.042
24	$-8, 0$	152396535	12.176	334553742	0.879	0.049
48	$-12, 0$	89229804	20.796	398466668	1.238	0.031

Table 6.8: Performance of schemes A, B and C under ordering $(32, 0, 0)$.

accuracy is allowed to decrease in order to keep all processors busy.

6.6.8 The Significance of the Neural Network Approach

In the experiments, the neural network approach, as embodied by scheme C, exhibits behavior that is quite unlike the other two schemes. In particular, for non-random trees (an exponential ordering where both w_a and w_b are non-zero), scheme C has a higher speed-up than schemes A and B when a large number of processors are used. Scheme C achieves this by reducing the split-point selection accuracy to keep all of its processors busy. In scheme A, when $m_A = 1$, a node may be selected as a split-point if at least one branch has been examined. Since scheme C achieves a higher degree of parallelism, it must have less stringent criteria for split-point selection. Specifically, scheme C may select a node as a split-point even before a branch has been fully examined. When searching a tree with a large w_b using a few processors, scheme C obtains a higher speed-up than schemes A and B by increasing the split-point selection accuracy at the cost of an increase in *CSS* overhead. The performance that the neural network approach obtains over the other two schemes is primarily due to its flexibility. It is flexible enough to increase parallelism when a large number of processors are being used and to increase split-point selection accuracy when a few processors are being used.

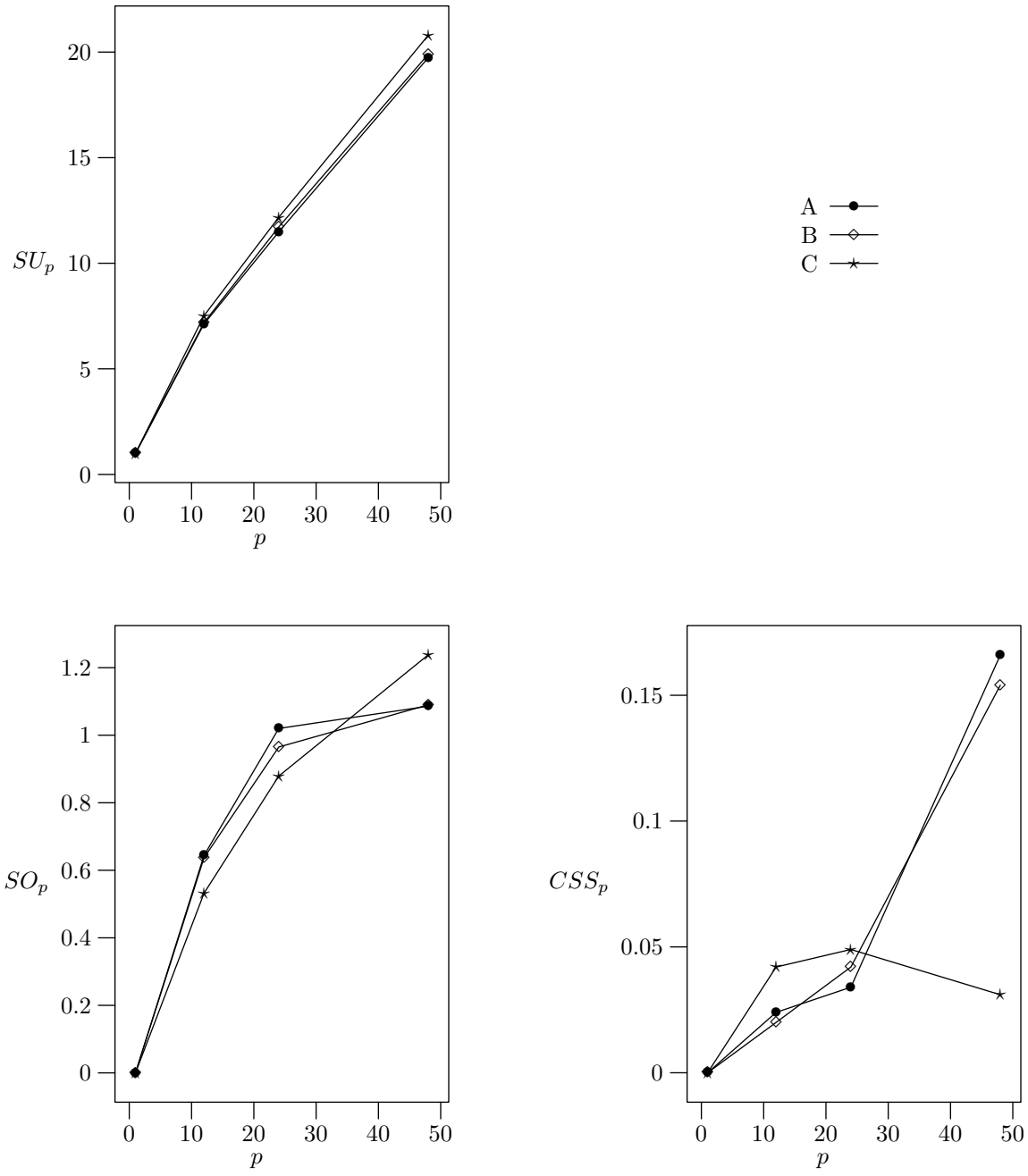

Figure 6.17: Performance of schemes A, B and C under ordering $(32, 0, 0)$.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

This work offers three main contributions. First, the concept of an exponentially ordered game tree was introduced as a model for the game trees encountered in practice. Various statistics arising from the search of exponentially ordered trees were compared to the statistics arising from the search of chess trees. With a certain set of parameters, exponentially ordered trees were found to produce trees that were quite similar to chess trees. Second, neural networks were introduced as a method of solving the node classification problem. The neural network approach was found to outperform all other node classification techniques. The prediction accuracy of the neural network approach increased as the tree ordering parameter, w_b, was increased. Third, the neural network approach was used as a split-point selection scheme for parallel search. Here, the neural network approach was found to perform better than techniques based on YBWC and DTS when a large number of processors was used. Furthermore, the neural network approach performed well on a few processors if the tree ordering parameter w_b was relatively large.

7.2 Future Work

There is a lot of room for experimentation with the neural network approach. In this work, we explored the performance of the neural network approach on exponentially ordered game trees. While exponentially ordered trees mimic most of the behavior of real game trees, there is no substitute for experimentation on real game trees. Chess has long served researchers as the main source of game trees and chess can be used once again as a testbed for the neural network approach. The parallel search results described here are based on simulations. This can be extended by performing the same experiments on a real shared memory multiprocessor with real game trees.

Appendix A

Seeds Used For Artificial Tree Generation

The artificial tree generation method presented in Section 4.2 generates a tree based on the value of the *seed* parameter. The experiments conducted in this text use two sets of seeds: tg_1 and tg_2. The first set consists of 100 seed values while the second set consists of 200 seed values.

A.1 Set 1, tg_1

A simple linear congruential random number generator produces the seed values for the first set, tg_1. The random number generator is based on the equation: $r_n = (r_{n-1} \times 16807) \bmod 2147483647$. Note that this is one of the two linear congruential generators used in Section 4.2. To start the computation of r_n, r_0 is initially set to 12345678. The values of r_n where $1 \leq n \leq 100$ make up set tg_1. Table A.1 presents the seeds in the set.

A.2 Set 2, tg_2

The second set, tg_2, is generated using the digits of π. A seed is formed using 9 sequential digits of π. For example, the first seed in this set is 314159265. Tables A.2 and A.3 present the seeds in this set.

$1 \leq n \leq 25$	$26 \leq n \leq 50$	$51 \leq n \leq 75$	$76 \leq n \leq 100$
1335380035	869731965	2022461825	374445651
380653449	1811434274	1144728060	1180970648
288732931	2047663247	138510948	1529815363
1568812745	1650749155	81229689	1932584058
231587350	799812493	1575267179	250101932
1048223087	1348423279	1337077238	837673946
1677066869	555123363	988256859	2030704338
740000409	1289399374	994503316	90206996
1109074287	677796942	752007412	2133010638
53485650	1479940507	1047310890	1565273496
1285155105	1204501596	1378157419	876971523
211328210	1877467351	2080608239	1080117701
2002756980	1616542887	1306448773	858932617
677879783	1420683613	1611720884	695418786
724765547	1706296346	1981657778	1297529329
607302646	226065185	398393524	2026480866
2093280779	578992753	2093430170	2120757090
1692947500	882795115	2056278390	1778322372
1357793398	172980683	416569560	1734190906
1272407165	1740964791	487905701	898500059
705065330	930551963	1138552462	2132969557
218237165	1810924688	1571934065	874825129
9963080	2072985933	1152005062	1512895742
2093244742	2047370651	36515683	1032355315
1087273641	1028055477	1686244787	1275395093

Table A.1: The seed values in set tg_1

$1 \leq n \leq 25$	$26 \leq n \leq 50$	$51 \leq n \leq 75$	$76 \leq n \leq 100$
314159265	564823378	807446237	958537105
358979323	678316527	996274956	079227968
846264338	120190914	735188575	925892354
327950288	564856692	272489122	201995611
419716939	346034861	793818301	212902196
937510582	045432664	194912983	086403441
097494459	821339360	367336244	815981362
230781640	726024914	065664308	977477130
628620899	127372458	602139494	996051870
862803482	700660631	639522473	721134999
534211706	558817488	719070217	999837297
798214808	152092096	986094370	804995105
651328230	282925409	277053921	973173281
664709384	171536436	717629317	609631859
460955058	789259036	675238467	502445945
223172535	001133053	481846766	534690830
940812848	054882046	940513200	264252230
111745028	652138414	056812714	825334468
410270193	695194151	526356082	503526193
852110555	160943305	778577134	118817101
964462294	727036575	275778960	000313783
895493038	959195309	917363717	875288658
196442881	218611738	872146844	753320838
097566593	193261179	090122495	142061717
344612847	310511854	343014654	766914730

Table A.2: The first hundred seed values in set tg_2

$101 \leq n \leq 125$	$126 \leq n \leq 150$	$151 \leq n \leq 175$	$176 \leq n \leq 200$
359825349	953311686	125338243	841757467
042875546	172785588	003558764	289097777
873115956	907509838	024749647	279380008
286388235	175463746	326391419	164706001
378759375	493931925	927260426	614524919
195778185	506040092	992279678	217321721
778053217	770167113	235478163	477235014
122680661	900984882	600934172	144197356
300192787	401285836	164121992	854816136
661119590	160356370	458631503	115735255
921642019	766010471	028618297	213347574
893809525	018194295	455570674	184946843
720106548	559619894	983850549	852332390
586327886	676783744	458858692	739414333
593615338	944825537	699569092	454776241
182796823	977472684	721079750	686251898
030195203	710404753	930295532	356948556
530185296	464620804	116534498	209921922
899577362	668425906	720275596	218427255
259941389	949129331	023648066	025425688
124972177	367702898	549911988	767179049
528347913	915210475	183479775	460165346
151557485	216205696	356636980	680498862
724245415	602405803	742654252	723279178
069595082	815019351	786255181	608578438

Table A.3: The second hundred seed values in set tg_2

Bibliography

[1] Brockington, M.G. (1996). A Taxonomy of Parallel Game-Tree Searching Algorithms. *ICCA Journal*, Vol. 19, No. 3, pp. 162 – 174.

[2] Culler, D.E., Singh, J.P. with Gupta, A. (1999). *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann.

[3] Donninger, C. (1993). Null Move and Deep Search: Selective-Search Heuristics for Obtuse Chess Programs. *ICCA Journal*, Vol. 9, No. 1. pp. 3 – 19.

[4] Foster, I. (1995). *Designing and Programming Parallel Programs*. Addison-Wesley.

[5] Feldmann, R. (1993). *Game Tree Search on Massively Parallel Systems*. Ph.D. Thesis, University of Paderborn, Paderborn, Germany.

[6] Feldmann, R., Monien, B., Mysliwietz, P. and Vornberger, O. (1989). Distributed Game Tree Search. *ICCA Journal*, Vol. 12, No. 2, pp. 65 – 73.

[7] Haykin, S. (1999). *Neural Networks: A Comprehensive Foundation*. 2nd ed., Macmillan/IEEE Press.

[8] Hyatt, R.M. (1988). *A High-Performance Parallel Algorithm to Search Depth-First Game Trees*. Ph.D. Thesis, University of Alabama, Birmingham.

[9] Hyatt, R.M. (1997). The Dynamic Tree-Splitting Parallel Search Algorithm. *ICCA Journal*, Vol. 20, No. 1, pp. 3 – 19.

[10] Kaindl, H. (1988). Useful Statistics from Tournament Programs. *ICCA Journal*, Vol. 11, No. 4. pp. 156 – 159.

[11] Knuth, D. E. (1998). *The Art of Computer Programming*, Vol. 2: *Seminumerical Algorithms*. 3rd ed., Addison-Wesley, Ontario.

[12] Knuth, D.E. and Moore, R.W. (1975). An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, Vol. 6, No. 4, pp. 293 – 326.

[13] Kopec, D. and Bratko, I. (1982). The Bratko-Kopec Experiment: A Comparison of Human and Computer Performance in Chess. In M.R.B. Clarke, editor, *Advances in Computer Chess 3*, pp. 57 – 72. Permagon Press, Oxford.

[14] Manohararajah, V. (1997). Rajah: The design of a chess program. *ICCA Journal*, Vol. 20, No. 2, pp. 87 – 91.

[15] Marsland, T.A. (1986). A Review of Game-Tree Pruning. *ICCA Journal*, Vol. 9, No. 1, pp. 3 – 19.

[16] Marsland, T.A. and Campbell, M.S. (1982). Parallel Search of Strongly Ordered Game Trees. *ACM Computing Surveys*, Vol. 14, No. 4, pp. 533 – 551.

[17] Marsland, T.A. and Popowich, F. (1985). Parallel Game-Tree Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. PAMI-7, No. 4, pp. 442 – 452.

[18] Marsland, T.A., Reinefeld, A. and Schaeffer, J. (1987). Low Overhead Alternatives to SSS*. *Artificial Intelligence*, Vol. 31, No. 2, pp. 185 – 199.

[19] Microsoft Platform SDK Documentation.

[20] Reinefeld, A. (1983). An Improvement of the Scout Tree-Search Algorithm. *ICCA Journal*, Vol. 6, No. 4, pp. 4 – 14.

[21] Schaeffer, J. (1989). Distributed Game-Tree Searching. *Journal of Parallel and Distributed Computing*, Vol. 6, No. 2, pp. 90 – 114.

[22] Shannon, C. E. (1950). Programming a Computer for Playing Chess. *Philosophical Magazine*, Vol. 41, No. 7, pp. 256 – 275.

[23] Slate, D.J. (1987). A Chess Program that uses its Transposition Table to Learn from Experience. *ICCA Journal*, Vol. 10, No. 2. pp. 59 – 71.