PS*, a new Algorithm for Searching Game Trees

N. Srimani
T.A. Marsland

Computing Science Department
University of Alberta
Edmonton, T6G 2H1
Canada

ABSTRACT

A new sequential tree searching algorithm (PS*) is presented. Based on SSS*, Phased Search (PS*) divides each MAX node into k partitions, which are then searched in sequence. By this means two major disadvantages of SSS*, storage demand and maintenance overhead, are significantly reduced, although the corresponding increase in nodes visited is less apparent even in the random tree case. The performance of PS* is compared theoretically as well as experimentally to the well known $\alpha\beta$ and SSS* algorithms, on a basis of the storage needs and the number of bottom positions visited.

Acknowledgement

Financial support was provided by Canadian Natural Sciences and Engineering Research Council Grant A7902.

Technical Report TR 86-2

N. Srimani is now at: Computer Science Department, Southern Illinois University, Carbondale, IL 62901

June 18, 2001

PS*, a new Algorithm for Searching Game Trees

N. Srimani
T.A. Marsland

Computing Science Department
University of Alberta
Edmonton, T6G 2H1
Canada

1. Introduction

Phased search is a new method for traversing minimax game trees. Although based on SSS* [Sto79], phased search has a range of performance which represents a continuum of algorithms from SSS* to $\alpha\beta$ [KnM75]. $\alpha\beta$ was the first minimax search algorithm to incorporate pruning into game-playing programs, and various modified versions of $\alpha\beta$ are still predominant, even though more efficient pruning methods exist. For example, SSS* never visits more terminal nodes than $\alpha\beta$, achieving better pruning at the expense of a large storage requirement. Here better pruning implies fewer terminal node (bottom position) visits, although other measures of performance, such as execution time and storage needs are also important. Number of bottom positions (NBP) visited is particularly relevant, because in any game-playing program the evaluation function spends significant time assessing these nodes. For this reason, SSS* has the potential to reduce the search time significantly by the virtue of its better pruning, but must maintain an ordered list (called OPEN) of $O(w^{\frac{d}{2}})$ entries. Because of this abnormally high requirement, and the considerable time spent maintaining the OPEN list, SSS* is not customarily used, in spite of its known dominance over $\alpha\beta$.

In its general form, Phased Search (PS*) has lower storage needs than SSS*, but at the same time maintains dominance over $\alpha\beta$ for trees of practical importance [Sri85]. The Phased Search algorithm with parameter k, PS*(k), partitions the set of all successors of MAX nodes into k groups (each of maximum size w/k) and limits its search to one partition per phase. It does not generate all the solution trees simultaneously as does SSS*; generating instead only a subset of them. The algorithm searches the

partitions serially from left to right one at a time. Like SSS*, the search strategy within each phase of PS* is non-directional, but with a recursive partitioning of the MAX nodes. Note that the storage requirement of PS*(k) is $O((\frac{w}{k})^{\frac{d}{2}})$, because PS*(k) searches only w/k successors at alternate levels of the game tree, (i.e., at the MAX nodes).

2. Game trees:

To put our study on a formal footing we introduce the following definitions. A tree T(w,d), of constant width w and fixed depth d, is said to be a uniform tree if every interior node has w immediate successors and all terminal nodes are at a distance d from the root. In a random uniform tree the terminal nodes are assigned random values from a uniform distribution. Random uniform trees are commonly used for simulation as well as asymptotic studies of search algorithm performance, because they are regular in structure and are simpler to analyze, though at first sight they are not good models of typical game trees. In ordered trees the best branch at any node is to one of the first w/R successors. Such a tree is said to be of order R. The higher the value of R the stronger the order, so that the case R = w corresponds to a minimal tree. More useful are probabilistically ordered trees with parameter (p,R) where, with probability p, the best move at every node is among the first w/R successors. Game tree searching algorithms are often compared on a basis of their effectiveness on random uniform trees [MuS85, RoP83] or probabilistically ordered trees [MaC82, RSM85].

After generating the list of moves (a set of successor positions), most game playing programs sort them in order of merit, using some knowledge of their strength. Often, the knowledge is nearly perfect so the best successor is expected to be among the first few considered. Thus real game trees are not random, but are modelled by strongly ordered trees [MaC82], e.g., probabilistically ordered trees with p=0.6 and R=w. The experimental results reported here have been obtained from searches of both ordered and random trees, thus the effectiveness of search algorithm on different tree types may be observed. More detailed results are to be found in Srimani's thesis [Sri85].

3. Phased Search (PS*) Algorithm:

Let PS* with k partitions be denoted by PS*(k). For simplicity, it is assumed that partitions are of equal size. That is, the width w of the uniform search tree is a multiple of the number of partitions. That is not a restriction, since PS*(k) generalizes easily to encompass arbitrary partition sizes.

Let P(n) be the Dewey-decimal identifier of the parent of a node n, let PSIZE be the size of each partition and let V(n) be the static evaluation at a terminal node. We will show that PS*(1) has identical performance to SSS*, and PS*(w) is equivalent to $\alpha\beta$. PS* is based on SSS*, but maintains two lists, one is similar to the OPEN list in SSS* and the other is a BACKUP list, to keep track of partially expanded MAX nodes. OPEN consists of triples (n,s,hi), where n is the node identifier, s is the status (an element in the set {LIVE, SOLVED}), and hi is a bound on the merit of that state (a real number in [-∞,+∞]). As in SSS*, the OPEN list is maintained as an ordered list of triples with non-increasing value of hi. The BACKUP list consists of vectors of the form (n,last,low,hi), where n is the identifier of a MAX node, last is the node identifier of the last son of n included in OPEN, and low and hi are the current lower and upper bounds on the value of node n. Whenever a MAX node in the OPEN list is solved or pruned, the corresponding vector is deleted from BACKUP.

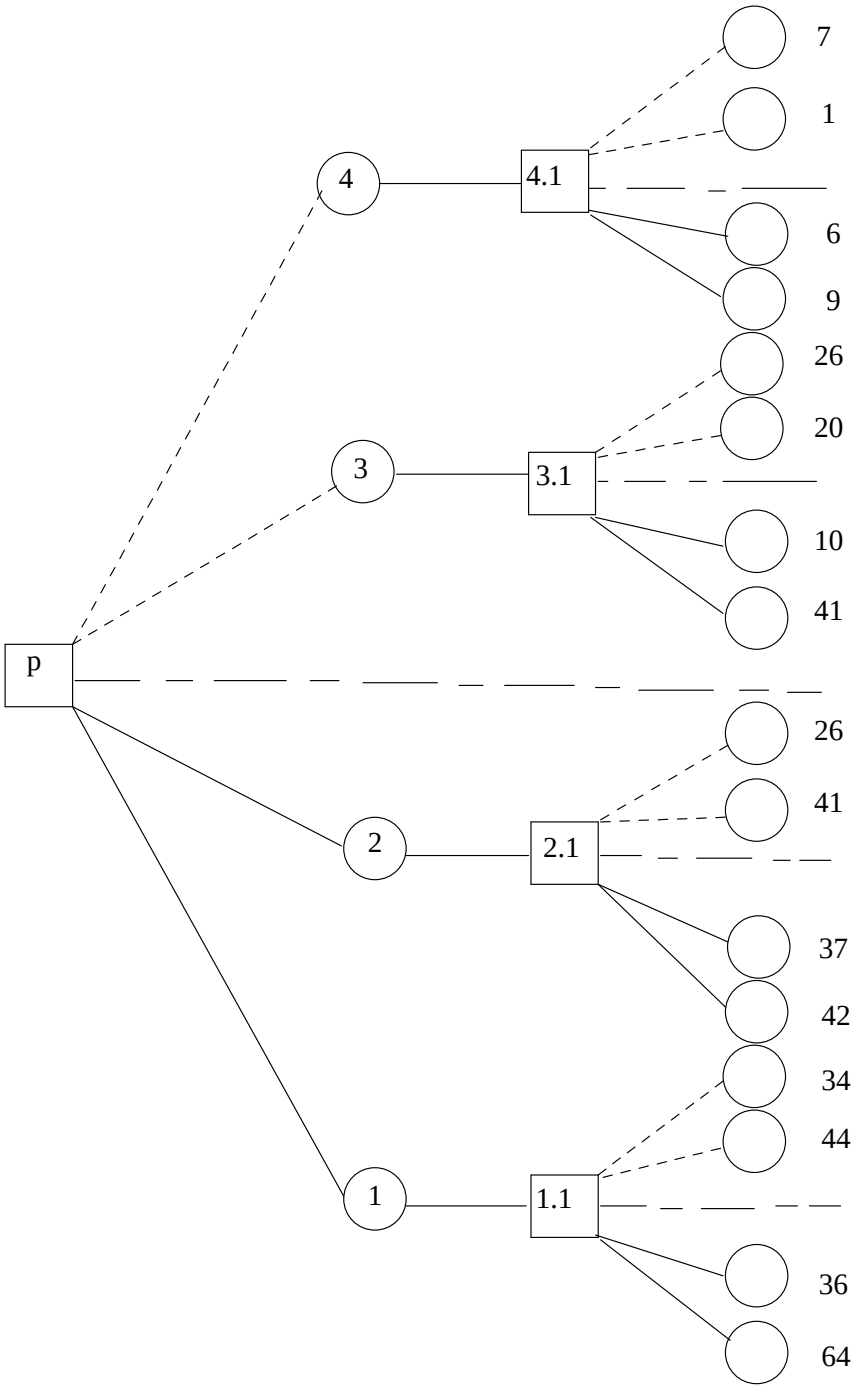

Figure 1: An Example of a Tree Searched by PS*(2)

The operation of phased search is seen most easily by an example. Figure 1 shows the search of a tree T(4,3) by PS*(2). Note that the successors of MAX nodes are divided into two partitions of equal size, as shown by vertical bars in Figure 1. This partitioning is done recursively at each MAX node in the tree, so that the successors of a MAX node in two different partitions are never added to the OPEN list in

the same phase of the PS* algorithm. Note also that, as with SSS*, at every MIN node only one successor at a time is included in the search tree. Thus, for the Figure 1 example, at any instant no more than four terminal nodes are present in the OPEN list for PS*(2), while in contrast SSS* would have all 16 nodes present in OPEN simultaneously at various points of the search.

3.1. Description of the Algorithm

Following the lines of Stockman's SSS* algorithm, and using his GAMMA operator terminology [Sto79], PS* is formed as follows:

(1) Note the value of k and w, and initialize PSIZE = w/k.

(2) Place the initial state (n=root, s=LIVE, hi=+∞) in OPEN and (n=root, last=NIL, low=-∞, hi=+∞) in BACKUP.

(3) Remove the first state f = (n,s,hi) from OPEN (this node has the highest merit).

(4) If n=root and s=SOLVED, terminate the algorithm with hi as the minimax value.

(5) Otherwise expand the state f by using the GAMMA operator, defined in Table 1, to modify both OPEN and BACKUP lists.

(6) Go to step (3).

PSIZE = w/k.
Let n be the m-th successor of its parent node i.
Thus i = P(n) and n = i.m, where n is not a root node.

Case	Condition of the input state (n,s,hi)	Action of GAMMA
1.	s=LIVE, Type(n) = MAX, n is interior.	Add states (n.j,s,hi) for all j=1,...,PSIZE to the front of OPEN in increasing order of j. (n,PSIZE,low,hi) is added to BACKUP, where low is the lower bound of n and hi is the upper bound. Note that, either low=-∞, if n=root, or low = low of P(i) stored in BACKUP.
2.	s=LIVE, Type(n) = MIN, n is interior.	Place (n.1,s,hi) in front of the OPEN list.
3.	s=LIVE, n is terminal.	Score = Min(V(n),hi). V(n) is the value returned by the evaluation function.
3a	Type(n) = MIN, or Score > low of P(i).	Place (n,SOLVED,Score) in OPEN in front of all states of lesser merit. Ties are resolved in favor of nodes which are leftmost in the tree.
3b	r mod PSIZE = 0, where i=P(i).r	Score = low of P(i)
3c	Type(n) = MAX, Score ≤ low of P(i)	Place (i,SOLVED,Score) in OPEN, maintaining the order of the list.
4.	s = SOLVED, m < w, Type(n) = MAX.	Purge all states corresponding to the successors of i from BACKUP.
4a	hi > low of P(i).	Place (i.m+1,LIVE,hi) in front of OPEN.
4b	hi ≤ low of P(i).	Place (i,SOLVED,hi) in front of OPEN.
5.	s = SOLVED, m = w, Type(n) = MAX.	Purge all successors of i from BACKUP. Place (i,SOLVED,hi) to the front of OPEN.
6.	s = SOLVED, Type(n) = MIN,	Obtain values of low(i) and hi(i) from the BACKUP list. Update low(i) = Max(low(i),hi).
6a	If low(i) ≥ hi(i)	Purge all successors of i from BACKUP and OPEN. Place (i,SOLVED,hi(i)) in front of OPEN.
6b	If low(i) < hi(i)	If there are some incompletely searched MAX successors (immediate or non-immediate) of node i present in BACKUP, then add the next partition of the first such node found in BACKUP to the front of OPEN; Else add the next partition of successors of i to the front of OPEN.

Table 1: State Space Operator (GAMMA) for PS*(k).

Table 1 describes the GAMMA operator for PS*(k) by its action states. In each iteration, the first state vector is removed from OPEN and GAMMA modifies both OPEN and BACKUP lists as necessary, depending on the status and type of the current node. For interior LIVE nodes, GAMMA either adds its first partition of successors or only the first successor, for node type MAX or MIN respectively, as described under cases (1) and (2) in Table 1. For a LIVE terminal node, n, V(n) is the value returned by the evaluation function; GAMMA either inserts n into OPEN with SOLVED status, or inserts i = P(n) into OPEN with SOLVED status, depending on how the value V(n) compares to the low and hi bounds, as described in steps 3(a) - 3(c) of Table 1. For a SOLVED MAX node, n, GAMMA purges the successors

of n from both OPEN and BACKUP, and either adds the next successor of P(n) to OPEN or prunes them according to the input conditions given in cases 4 and 5. Similarly for SOLVED MIN nodes, GAMMA either adds the next partition to OPEN or prunes the rest of the partitions, as described in case 6 of Table 1.

3.2. Proof of correctness of PS* algorithm:

Before discussing PS* any further, let us prove that the algorithm always returns the minimax value.

Theorem 1: PS*(k), with its state operator GAMMA, computes the minimax value of the root for all trees, for any k which is a factor of w.

To prove the correctness of PS*(k), it is necessary to show:

(1) PS* does not terminate with an inferior solution (i.e., PS* is admissible).

(2) PS* always terminates.

Proof:

Following the argument of Stockman [Sto79], let g(root) be the minimax value of the tree being searched, and f(T_{root}) be the value returned by PS*(k) for the solution tree T. It follows that g(root) ≥ f(T_{root}) for any solution tree T and that there exists a solution tree $T0$ such that g(root) = f($T0_{root}$) [Sto79]. To show (1) by contradiction, suppose that, for some k ≥ 1, PS*(k) terminates with a solution tree $T1$ which is inferior to $T0$, that is, f($T1_{root}$) < f($T0_{root}$). This cannot happen, since if $T0$ is in the same partition with $T1$ (or in one of the previous partitions), then $T0$ would be solved first and there would be a triple (n,s,hi_0) for the solution tree $T0$ such that, hi_0 ≥ f($T0_{root}$) ≥ f($T1_{root}$) and $T0$ would be solved before $T1$. Otherwise, if $T1$ is fully solved and $T0$ is in one of the later partitions, the corresponding state (n,s,hi_0) would appear at the front of OPEN before the root node can be declared SOLVED. When $T0$ appears, the corresponding solution tree would be evaluated fully and found to be better than $T1$, since it cannot be pruned. The BACKUP list keeps track of partially expanded nodes and is the mechanism which protects the algorithm against termination with an inferior solution.

Part (2) is true, since there are only a finite number of solution trees, and any subtree once solved or discarded is not searched again. So the algorithm terminates after a finite number of steps.

3.3. Comparisons of PS* with other Algorithms

Let R be the order of the tree being searched, and let PS*(k) denote the Phased Search algorithm with k phases. Using the notation of Roizen and Pearl [RoP83], where I(A) represents the number of bottom positions visited by algorithm A,

(1) For minimal trees, I(SSS*) = I(PS*(k)) = I($\alpha\beta$), because all algorithms traverse the best branch first and so achieve maximal cut-offs.

(2) When R ≥ k, I(PS*(k)) ≤ I(SSS*) ≤ I(AB), since the best solution is one of the first w/R branches at every node. Although there may not be many cases where strict inequality holds, PS*(k) is at least as good as SSS* as long as R ≥ k, because the best solution is always found in the first partition.

Figure 2 is an example where I(PS*(2)) < I(SSS*), for a tree of depth=5 and width=4. Only that part of the tree which is enough to demonstrate the point has been presented. Assume that node 2.1 is solved with value 64, so node 2.2 has upper bound 64. Consequently, 2.2.1.1.1 and 2.2.1.1.2 are solved with values 18 and 21 respectively. Then 2.2.2.1, 2.2.2.2, 2.2.2.3 and 2.2.2.4 are included in OPEN and solved with values ≥ 64, hence node 2.2.2 is solved. Note that nodes crossed in Figure 2 are not visited by PS*(2) but are by SSS*.

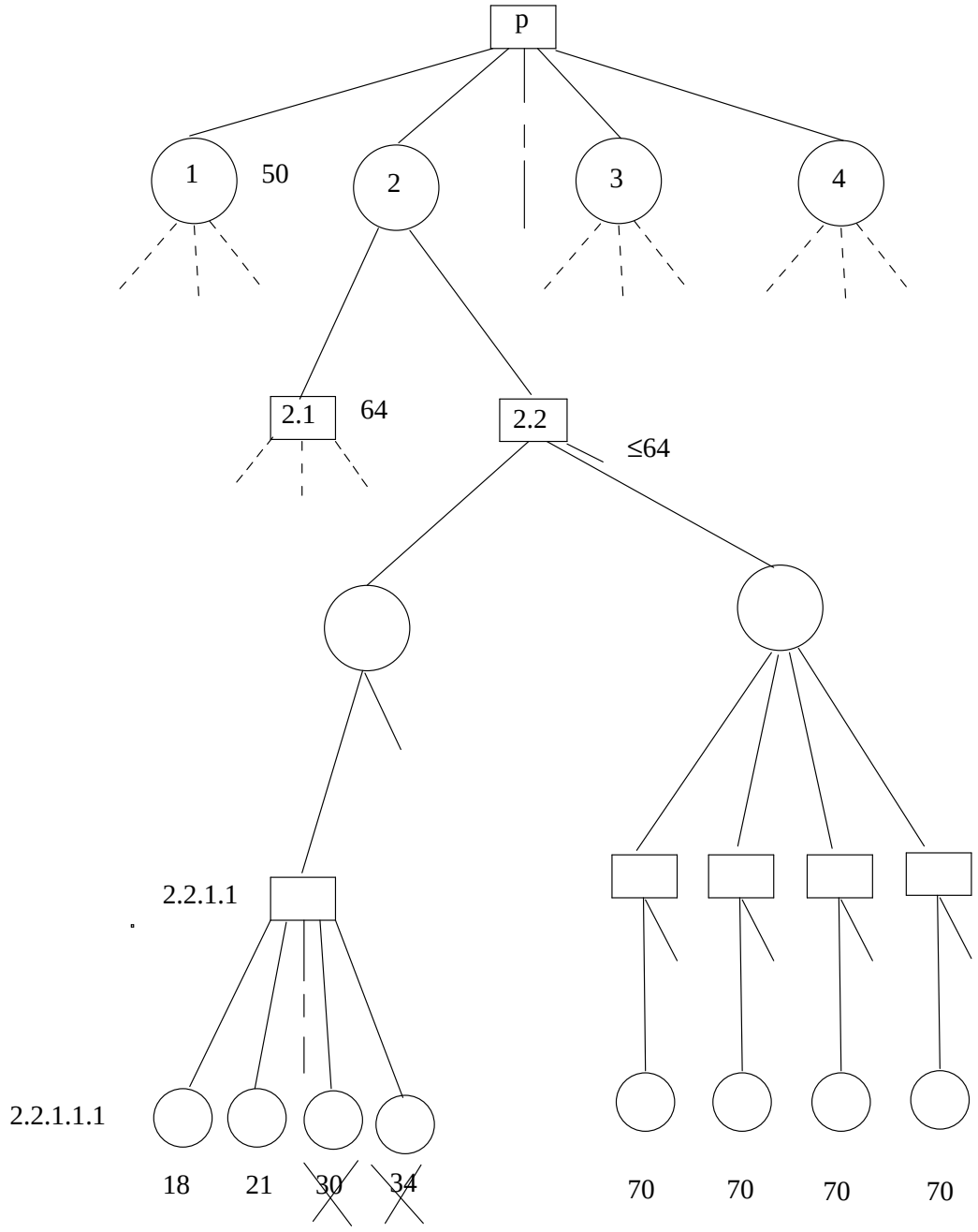

Figure 2: Tree T(4,5) in which PS*(2) is better than SSS*.

(3) If R < k, for some trees I(PS*) can be greater than I(SSS*). Similarly, if the tree is random, then PS*(k) will occasionally evaluate some extra nodes. However, our experimental results show that even when R < k, in most of the cases (including random trees) PS*(k) is still better than $\alpha\beta$.

As shown in Figure 3 PS*(2) would evaluate the nodes underlined, but SSS* will not. On the other hand, in the example of Figure 3, the boxed terminal nodes are evaluated by $\alpha\beta$, but not by SSS*

and PS*(2).

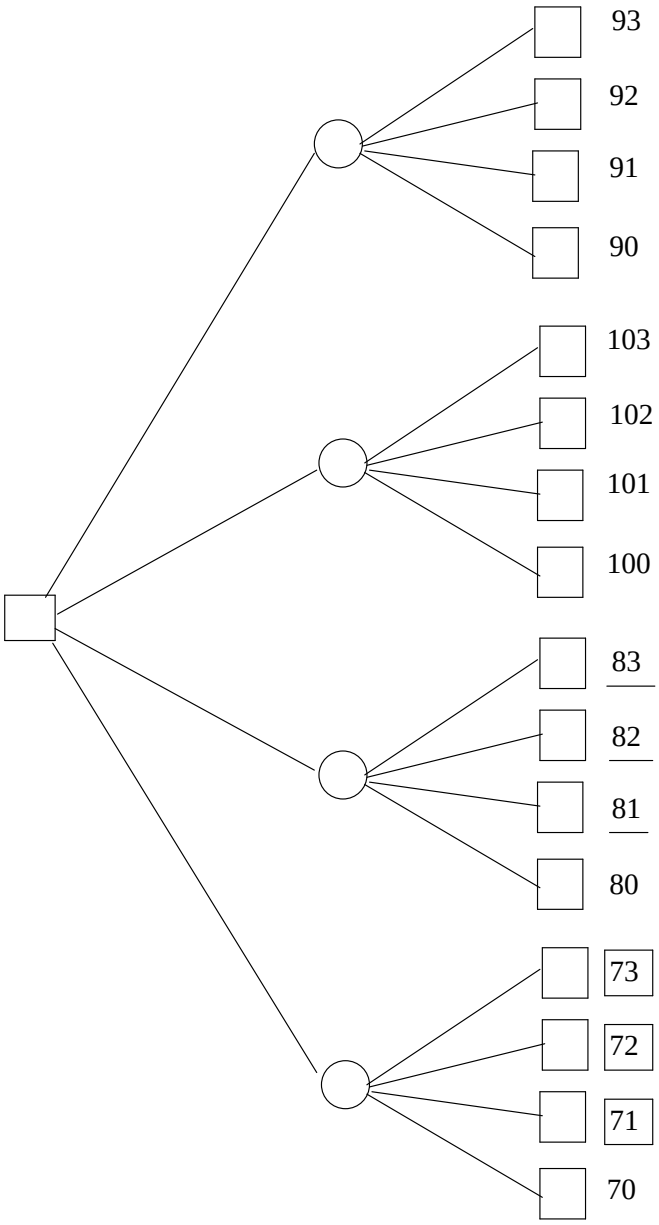

Figure 3: Tree T(4,2) in which I(SSS*) > I(PS*(2)) > I($\alpha\beta$).

(4) There are trees which are unfavorable for PS*, so that I(PS*(k)) > I($\alpha\beta$). Figure 4 illustrates the case where $\alpha\beta$ ignores the nodes in circles, but PS*(2) evaluates them. Such trees are statistically insignificant, and are uncommon in typical applications.

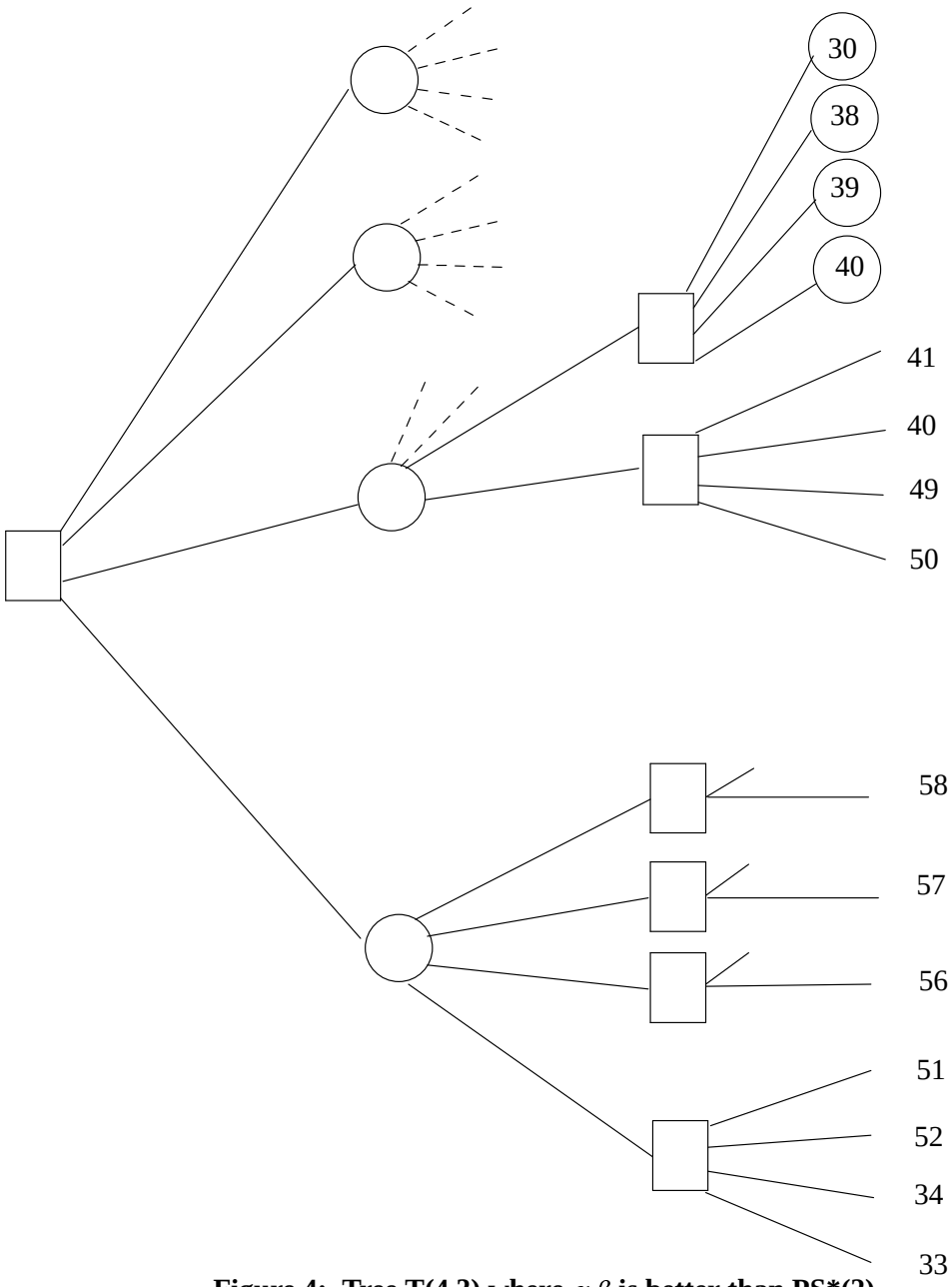

Figure 4: Tree T(4,3) where $\alpha\beta$ is better than PS*(2).

3.4. Space Requirement Analysis:

Lemma 1: Maximum size of OPEN for PS* with k partitions is $O((\frac{w}{k})^{\frac{d}{2}})$.

For k > 1 this is less than the space requirement of SSS*, which is $O(w^{\frac{d}{2}})$.

Proof: Each phase of PS*(k) is equivalent to an SSS* search on a tree of width w/k, since at each MAX node only w/k of its successors are included in OPEN. Hence the lower storage requirement follows by

analogy.

Lemma 2: The BACKUP list, needed to store partially expanded MAX nodes, is of size $O((\frac{w}{k})^{\lfloor \frac{d-1}{2} \rfloor})$.

Proof: The maximum number of interior MAX nodes at level i to be kept in the BACKUP list is $((\frac{w}{k})^{\frac{i}{2}})$. For a tree of depth d the MAX nodes are at levels i = 0, 2, ..., d-1. Hence for trees of depth d, the BACKUP size would be

$$= \sum_{j=0}^{\lfloor \frac{d-1}{2} \rfloor} (\frac{w}{k})^j$$

which is a simple geometric progression with approximate sum of

$$= O((\frac{w}{k})^{\lfloor \frac{d-1}{2} \rfloor})$$

Corollary: Let S(A) denote the space needed by an algorithm A, then

$$S(PS^*(k)) \leq S(SSS^*)$$

for any k > 1 and for any depth and width of the search tree.

Corollary: If the number of phases in PS* is k, then as far as storage is concerned

PS*(k) is equivalent to SSS* for k=1 and

PS*(k) is equivalent to $\alpha\beta$ for k=w.

The first case is clear, and the second too, since PS*(w) reduces to a depth-first left to right (directional) search.

3.5. Choice of Partition Count

From the previous discussions, it is clear that selection of k, the partition count, is important if the algorithm is to achieve its maximum benefit. If from some previous knowledge we know that the tree is of order R, we can choose k = R. Then I(PS*(k)) would be the same as I(SSS*), but the storage requirement of PS*(k) would be about $1/(k^{d/2})$ of that of SSS*.

Clearly, there is a trade-off between space and bottom positions visited. If k=w, minimum space is required, but NBP will increase to that of an $\alpha\beta$ search. On the other hand, if k=1, NBP would be low but space needed would be as much as for SSS*. Thus PS* forms a continuum between SSS* and $\alpha\beta$. PS* can be made effective by using information about the ordering properties of game trees, since one can choose the parameter k both on the basis of the tree ordering and on the memory space available.

It is necessary to consider different ordering schemes, since ordered trees are better than random trees at approximating typical games. For example, for a tree of depth=4 and width=32 SSS* needs 1024 storage areas whereas PS*(4) requires 64+9 = 73, and PS*(8) needs only 16+5 = 21 for both the OPEN and BACKUP lists. Also, maintaining an ordered list of size 64 or 16 is much cheaper than for a list of 1024 elements. Note also that, unlike SSS*, although PS* maintains two ordered lists, OPEN and BACKUP, the total size of the two lists in PS* is much less than that of the single list of SSS*. Hence, the time spent by PS* manipulating these overhead lists is less than that needed by SSS*.

4. Experimental Performance Comparison:

The search algorithms PS*(k), SSS*, and $\alpha\beta$ have been implemented on a VAX/780 using C; extensive experimental investigations have been carried out with ordered as well as random trees. Uniform trees with different combinations of depth, width and tree ordering were tried in the experiments, some of which are reported here.

Experimental results on minimal, random and ordered versions of uniform trees T(8,4), T(16,4), T(24,4), T(32,4) and T(8,6) are presented. For the trees of width = 8, 16 and 24, orders 2 and 4 were searched and for trees of width = 32, order 8 was also studied. For each combination, 100 different trees were generated using a modified version of the scheme developed by Campbell[CaM83], and the average NBP's visited by each algorithm are presented in the tables. The maximum amount of space needed is also given. Some of the results in Tables 2 through 6 are also shown graphically in Figures 5 through 7.

The following observations about the experimental results can be made:

(1) For most of the trees, PS*(i) < PS*(j), for $1 \leq i < j \leq w$. That is, PS*(k) visits terminal nodes in increasing number with increasing k. There are some trees, like the ones given in Figures 2 and 3,

for which this is not true. Also in the tables it is shown that the above relation marginally fails to hold for ordered trees with p=1, where PS*(2) and PS*(4) often have statistically insignificant better performance than SSS* (i.e., PS*(1)).

(2) SSS* is always better than $\alpha\beta$ and is statistically better than PS* for both random and probabilistically ordered trees, Table 4.

(3) Data in Tables 2 through 6 show that on random trees, the NBP for PS*(2) is much less than for $\alpha\beta$, but more than for SSS*. For trees of order = 2 and higher, PS*(2) and SSS* have the same performance, but it is clear that PS*(2) needs much less space.

(4) For perfectly ordered trees, each of the algorithms visits minimum bottom positions. Table 4 shows the results on both ordered and probabilistic trees of depth=4 and width=24 of order=2 and 4. In the probabilistic case, NBPs are slightly greater, as we would expect, because at every MAX node there is a 10% probability that the best branch is not found in the first partition searched by PS*.

Search algorithm	ord=1 (random)	ord=2	ord=4	ord=8 (minimal)	size
SSS*	439	287	190	127	64
PS*(2)	571	286	190	127	21
PS*(4)	634	375	190	127	7
$\alpha\beta$	689	415	248	127	4

Table 2: NBP on Trees with depth=4 and width=8.

Search algorithm	ord=1 (random)	ord=2	ord=4	ord=16 (minimal)	size
SSS*	2250	1637	1146	511	256
PS*(2)	2829	1637	1146	511	73
PS*(4)	3363	2114	1146	511	21
PS*(8)	3743	2388	1496	511	7
$\alpha\beta$	3952	2981	1664	511	4

Table 3: NBP on Trees with depth=4 and width=16.

For minimal trees NBP=1151 for each algorithm.						
		prob=1.00		prob=0.90		
Search algorithm	ord=1 (random)	ord=2	ord=4	ord=2	ord=4	size
SSS*	5805	4423	3206	4702	3513	576
PS*(2)	7345	4423	3203	4956	3690	157
PS*(4)	8650	5718	3201	6460	3940	43
PS*(6)	9207	6222	3950	7126	4649	21
PS*(8)	9753	6652	4300	7517	4938	13
$\alpha\beta$	10602	7437	5031	8364	5660	4

Table 4: NBP on Trees with depth=4 and width=24.

Search algorithm	ord=1 (random)	ord=2	ord=4	ord=8	ord=32 (minimal)	size
SSS*	10816	8493	6424	4633	2045	1024
PS*(2)	13989	8478	6422	4632	2045	273
PS*(4)	16464	11089	6420	4632	2045	73
PS*(8)	18512	12782	8313	4631	2045	21
PS*(16)	20145	13966	9330	6209	2045	7
$\alpha\beta$	20836	14665	10046	6974	2045	4

Table 5: NBP on Trees with depth=4 and width=32.

Search algorithm	ord=1 (random)	ord=2	ord=4	ord=8 (minimal)	size
SSS*	6044	3475	1932	1023	512
PS*(2)	9984	3437	1921	1023	85
PS*(4)	11283	5213	1915	1023	15
$\alpha\beta$	11565	5555	2659	1023	6

Table 6: NBP on Trees with depth=6 and width=8.

Figure 5: NBP on trees with depth = 4 and width = 16.
Figure 6: NBP on trees with depth = 4 and width = 24.
Figure 7: NBP on trees with depth = 4 and width = 32.

5. Conclusion

The new algorithm PS*(k) may be viewed as a continuum between SSS* and $\alpha\beta$, and it attempts to make use of the best characteristics of both. $\alpha\beta$ searches a game tree much faster than SSS*, but SSS*, making more use of the knowledge gained at earlier steps, prunes better than $\alpha\beta$ and as a result visits fewer bottom positions. SSS* achieves this better pruning at the expense of extra bookkeeping which needs more storage and considerable time for the retrieval process. The phased search algorithm PS* also

does some bookkeeping and achieves much better pruning than $\alpha\beta$ in a statistical sense. Also, since PS* concentrates only on a subset of the solution trees in each phase, it consequently needs smaller storage and less execution time than SSS*. Thus PS*(k) can be comparable to SSS* in performance, especially on bushy trees (w > 20), and yet at the same time has significantly lower storage overhead than SSS*. Also, because of the built-in flexibility provided by phasing and the freedom of choosing the partition size parameter (PSIZE), PS* is expected to be useful in practice. PS* becomes most efficient if the parameter selection is done carefully using some a priori knowledge of the expected location of the solution.

Experimental results reported here are based on simulated game trees, and the algorithm remains to be tested with a typical game playing program. However experience with other alternatives to $\alpha\beta$ [RSM85] show that the performance on probabilistic uniform trees is a good indicator of performance in a typical application [Mar83]. In the work reported here, the successors of a MAX node in the PS*(k) algorithm are divided into partitions of equal sizes. This is not a restriction, but experiments with unequal size partitioning constitutes future work. We speculate that a progressively increasing partition size for each phase will be a good compromise, since a bound will be obtained in the early phases which will be good enough to reduce the space needs for the remaining phases. It also appears that PS*'s, partitioning scheme can be easily tailored to parallel game tree search.

6. BIBLIOGRAPHY

References

[CaM83] M.S. Campbell and T.A. Marsland, A comparison of minimax tree search algorithms, *Artificial Intelligence 20(4)*, (1983), 347-367.

[KnM75] D. Knuth and R. Moore, An analysis of Alpha-Beta pruning, *Artificial Intelligence 6(4)*, (1975), 293-326.

[MaC82] T.A. Marsland and M. Campbell, Parallel Search of strongly ordered game trees, *Computing Surveys 14(4)*, (1982), 533-551.

[Mar83] T.A. Marsland, Relative Efficiency of Alpha-beta Implementations, *Procs. 8th Int. Joint Conf. on Art. Intell.*, (Los Altos: Kaufmann), Karlsruhe, West Germany, Aug. 1983, 763-766.

[MuS85] A. Musczycka and R. Shinghal, An Empirical Comparison of Pruning Strategies in Game Trees, *IEEE Trans. on Systems, Man and Cybernetics SMC-15*, 3 (1985), 389-399.

[RSM85] A. Reinefeld, J. Schaeffer and T.A. Marsland, Information acquisition in Minimal Window Search, *9th IJCAI Conf. Procs.*, Los Angeles, 1985, 1040-1043.

[RoP83] I. Roizen and J. Pearl, A minimax algorithm better than Alpha-Beta? Yes and No., *Artificial Intelligence 21(2)*, (1983), 199-220.

[Sri85] N. Srimani, A new algorithm (PS*) for searching game trees, M.Sc. thesis, Computing Science Dept., University of Alberta, Edmonton, July 1985.

[Sto79] G.C. Stockman, A minimax algorithm better than Alpha-Beta?, *Artificial Intelligence 12(2)*, (1979), 179-196.