PHASED STATE SPACE SEARCH

T.A. MARSLAND and N. SRIMANI

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

ABSTRACT

PS*, a new sequential tree searching algorithm based on the State Space Search (SSS*), is presented. PS*(k) divides each MAX node of a game tree 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, and yet the corresponding increase in nodes visited is not so great even in the random tree case. The performance and requirements of PS* are compared on both theoretical and experimental grounds to the well known $\alpha\beta$ and SSS* algorithms. The basis of the comparison is the storage needs and the average count of the bottom positions visited.

To appear in the Procs. of the ACM/IEEE Fall Joint Computer Conference, Dallas, Nov. 1986, pp 514-518.

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

INTRODUCTION

Phased search is a new variation on a method for traversing minimax game trees. Although based on SSS*[1], phased search has a range of performance which represents a continuum of algorithms from SSS* to $\alpha\beta$[2]. The $\alpha\beta$ algorithm was the first minimax search method to incorporate pruning into game-playing programs, and modified versions of it still predominate, 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 larger storage requirement. Here better pruning implies fewer terminal node (bottom position) visits, although other measures of performance, such as execution time and storage needs, may be more important. Even so, the number of bottom positions (NBP) visited is particularly relevant, because in any game-playing program the evaluation function spends significant time in assessing these nodes. For this reason, SSS* has the potential to reduce the search time significantly by the virtue of its better pruning. However, for uniform trees with a constant width of w branches and a fixed depth of d ply, SSS* must maintain an ordered list (called OPEN) of $O(w^{d/2})$ entries. Because of this abnormally high memory demand and the considerable time spent in maintaining the OPEN list, SSS* is not widely used, despite its known pruning dominance over $\alpha\beta$.

In its general form, the phased search algorithm, denoted here by PS*, has lower storage requirements than SSS*, but at the same time consistently outperforms $\alpha\beta$ for trees of practical importance[3]. The Phased Search algorithm with k phases, PS*(k), partitions the set of all immediate successors of MAX nodes into k groups (each of maximum size $\lceil w/k \rceil$) 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 from left to right one at a time. Like SSS*, the search strategy within each phase of PS* is non-directional, but with a recursively sequential partitioning of the MAX nodes. Note that the storage requirement of PS*(k) is $O((\frac{w}{k})^{d/2})$, because PS*(k) searches only w/k successors at alternate levels of the game tree (i.e., at the MAX nodes).

GAME TREES

To provide a formal footing the following definitions are introduced. In a *uniform tree*, T(w,d), every interior node has exactly w immediate successors and all terminal nodes are at the same distance d from the root. The term *random tree* will be applied to those uniform trees whose terminal nodes are assigned *random* values from a uniform distribution. Such trees are commonly used for simulation as well as asymptotic studies of search algorithm performance, because they are regular in structure and are simple to analyze. In *ordered trees* the best branch at any node is 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. For random trees R = 1, while R = w corresponds to a *minimal tree*, that is, a tree in which the first successor is everywhere best. More useful are *probabilistically ordered trees* with parameter (p,R). Here it is only with probability p that the best subtree is among the first w/R successors. These definitions are useful since game tree searching algorithms have been compared on a basis of their effectiveness on random uniform trees[4, 5] and on probabilistically ordered trees[6, 7].

After generating the list of moves (a set of successor positions), most game playing programs use some knowledge of desirable features to sort the moves in order of merit. Often, the knowledge is quite accurate so the best successor will be found among the first few considered. Thus real game trees are not random, but have been approximated by *strongly ordered trees*[6]. These in turn are similar to probabilistically ordered trees with p=0.7 and R=w. The experimental results reported here have been obtained from searches of both ordered and random trees, so that the effectiveness of search algorithms can be observed under different conditions. More detailed results are to be found in Srimani's thesis[3].

PHASED SEARCH (PS*) ALGORITHM

Let PS* with k partitions be denoted by PS*(k). For simplicity, it is assumed that the partitions are of equal size. That is, the width w of the uniform search tree is a multiple of the number of partitions. This 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, n. 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 like 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,high), 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 *high* 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.

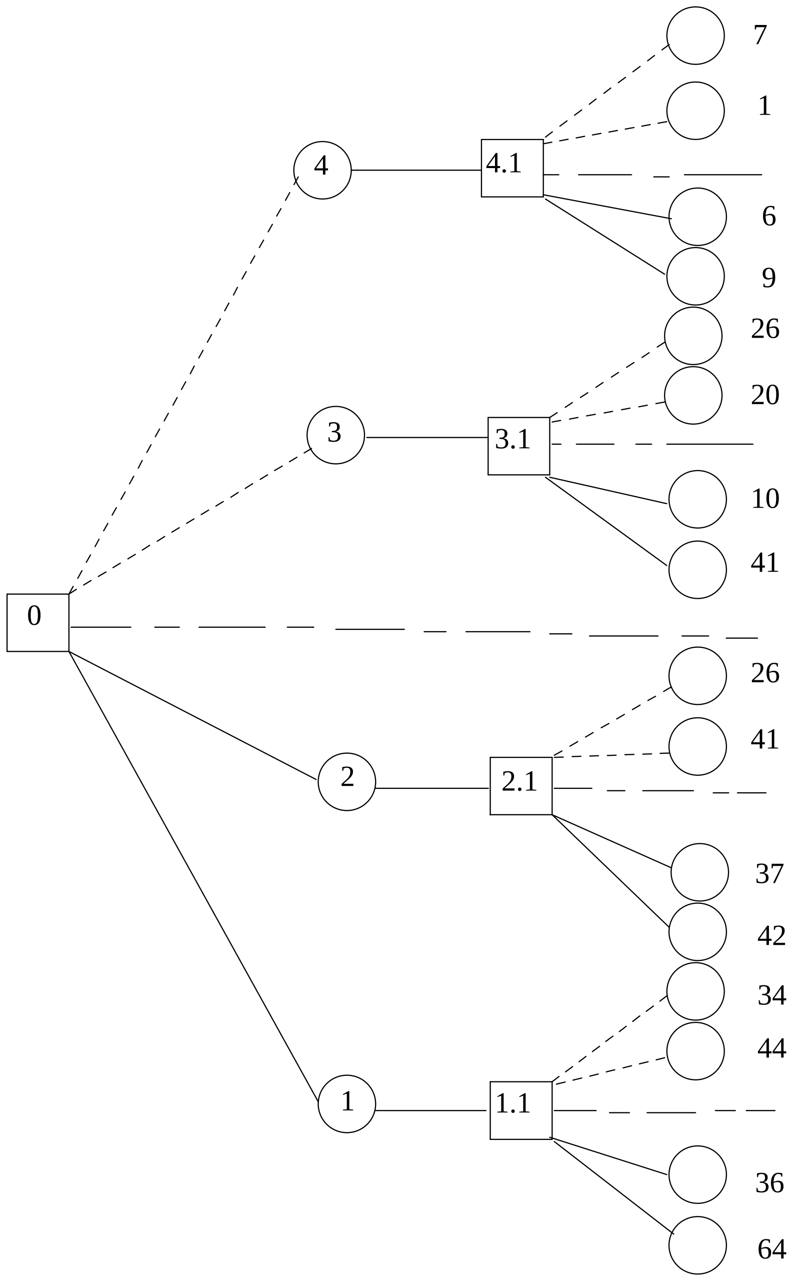

Figure 1: How PS*(2) Partitions a Tree.

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 broken lines 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 example in Figure 1, at any instant no more than four terminal nodes are present in the OPEN list for

PS*(2), while in contrast SSS* would have sixteen nodes present in OPEN simultaneously at some points in the search. Finally, note that PS*(2) will always work well if the best successor occurs in the first half of the subtrees at nodes which must be fully expanded (i.e., to use Knuth and Moore's terminology[2], at the type 1 and type 3 nodes).

Description of the Algorithm

Following the lines of Stockman's SSS* algorithm, and using his Γ operator terminology[1], PS*(k) is formed as follows:

(1) For simplicity, assume that w is a multiple of k and set PSIZE = w/k.

(2) Place the initial state (n=root, s=LIVE, hi=$+\infty$) on the OPEN list.

(3) Repeatedly retrieve the next state (n,s,hi) from OPEN (this node has the currently highest merit, hi) and invoke the Γ operator, described in Table 1, until the termination condition is reached.

In Table 1, case 1 corresponds to the retrieval of a LIVE interior MAX or MIN node. If a MAX node is found the first partition is added to OPEN, otherwise (for a MIN node) only the first successor is taken. In the case of a MAX node, an entry is also added to the BACKUP list. For a LIVE terminal node, Γ either inserts n into OPEN with SOLVED status, or inserts the parent of n, P(n), into OPEN with SOLVED status. The choice is made in alternatives 2a & 2b and depends on how V(n), the evaluation of node n, compares to the low bound. For a SOLVED MAX node, n, Γ purges the successors of P(n) from the BACKUP list, and either adds the next successor of P(n) onto OPEN or prunes by pushing the solved parent node onto the OPEN list, cases 3b & 3c respectively. Similarly for SOLVED MIN nodes, Γ either adds another partition to OPEN or purges the pruned successor partitions. Here case 4(b) is especially complex, since it must deal with the situation when the parent MAX node has another partition to process. More than any other, it is the actions in case 4 which distinguish PS* from SSS*.

Correctness of PS*

To make it clear that the PS* algorithm always returns the minimax value, the following theorem is provided.

Theorem.

PS*(k), with its state operator Γ, computes the minimax value of the root for all trees.

Proof: It is necessary to show that

(1) PS* always terminates, and

(2) PS* does not terminate with an inferior solution.

The aim of a game tree search is to find the best solution tree. Each solution tree is a unique subtree of the game tree and is made up of all successors of each MIN node, but only one successor of each MAX node it contains. The minimax value of a game tree is the value of the best solution tree. Hence following the notation of Stockman[1], g(root) $\geq$ f(T_{root}), where f(T_{root}) is the value of a solution tree and g(root) is the minimax value. Also, if $T0_{root}$ is the best solution tree then g(root) = f($T0_{root}$). It follows that the algorithm always terminates after a finite number of steps, since there are only a finite number of solution trees, and any subtree once solved or discarded is not searched again.

PS* manipulates its search among different solution trees, in order to find the best one as easily as possible. Within a solution tree it carries out a minimax search, and uses the upper bound stored in the BACKUP list to help prune the search of redundant subtrees. For any solution tree, T_{root}, f(T_{root}) also represents the minimax value returned by PS*, provided PS* has searched that solution tree completely. All that remains is to show that PS* finds the best solution tree $T0_{root}$. 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 if $T0$ and $T1$ occur in the same partition, since PS* will select the best for the same reason that SSS* does. If $T0$ is in a previous partition, then $T0$ would be SOLVED before $T1$ is encountered, and there would be a triple (n,s,hi_0) for the solution tree $T0$ such that, hi_0 = f($T0_{root}$) ≥ f($T1_{root}$). The value hi_0 is held as a lower bound in the BACKUP list, and so prevents $T1$ from being fully expanded and selected. Otherwise, if $T1$ is SOLVED and $T0$ occurs 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 as the best solution tree, $T0$, cannot be pruned.

From the theorem it follows naturally that if the number of phases in PS* is k, then

for k=1, PS*(k) is equivalent to SSS* and
for k=w, PS*(k) is equivalent to $\alpha\beta$,

as far as nodes visited is concerned, since PS*(w) reduces to a depth-first left to right (directional) search. Also, for k > 1, the space requirement for OPEN is less than the $w^{d/2}$ needed for SSS*, since

the maximum size of OPEN for PS* with k partitions is at most $(\frac{w}{k})^{d/2}$ entries.

Finally the BACKUP list, for partially expanded MAX nodes, requires

$\sum_{j=0}^{\lfloor\frac{d-1}{2}\rfloor} (\frac{w}{k})^j$ entries, which is about $(\frac{w}{k})^{\lfloor\frac{d-1}{2}\rfloor}$ entries.

Thus for PS*(k) the size of BACKUP is about k/w of the size of OPEN, and the two lists together occupy significantly less space than the single OPEN list used by SSS*. Consequently, if S(A) denotes the space needed by an algorithm A, then S(PS*(k)) ≤ S(SSS*) for any k > 1 AND FOR any depth and width of the search tree.

Comparison with other Methods

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[4], let I(A) represent the number of bottom positions visited by algorithm A.

(1) For minimal trees (optimally ordered game trees), I(SSS*) = I(PS*(k)) = I($\alpha\beta$), because all algorithms traverse the best branch

first and so achieve maximal cut-offs.

(2) For ordered trees, when p = 1 and R ≥ k, I(PS*(k)) ≤ I(SSS*) ≤ I(AB), since the best solution is always among the first w/R branches at every node in the solution tree. 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 provides 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 the value of node 2.2 has an upper bound of 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 visited by SSS* but not by PS*(2).

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

(3) For ordered trees, if p = 1 and R < k, 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 the $\alpha\beta$ algorithm[3].

(4) There are trees which are unfavorable for PS*, so that I(PS*(k)) > I($\alpha\beta$). Such trees are statistically insignificant, and are uncommon in typical applications, because they represent a worst first ordering within every partition.

PERFORMANCE COMPARISON

The search algorithms PS*(k), SSS*, and $\alpha\beta$ have been implemented on a VAX 11/780 using the C language. Experimental investigations were carried out with both ordered and random trees, using different combinations of depth, width and tree ordering. Some of the results on minimal, random, and ordered versions of the 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 R = 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[8], and the average NBP visited by each algorithm are presented in the tables. The maximum amount of space needed is also given in terms of list entries.

Based on the search of 100 different trees, the following observations about the average performance of PS*(k) are possible:

(1) Data in Tables 2 through 6 show that on random trees (R = 1), the average NBP for PS*(2) is much less than for $\alpha\beta$, but more than for SSS*. For trees of order R = 2 and higher, PS*(2) and SSS* have the same performance, but it is clear that PS*(2) needs much less space.

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

perfectly ordered trees, each algorithm visits minimum bottom positions.

(3) Table 4 shows the results on both ordered and probabilistic trees of depth=4, width=24 and of orders R=2 and 4. In the probabilistic case, the average NBP are slightly greater, as we would expect, because at every MAX node there is some nonzero probability that the best branch is not found in the first partition searched by PS*.

(4) For most of the trees, $I(PS^*(i)) < I(PS^*(j))$, for $1 \le i < j \le w$. That is, PS*(k) visits terminal nodes in increasing number with increasing k. There are some trees for which this is not true[3]. It is also known that the above relation marginally fails to hold for ordered trees (probability p=1), see for example Tables 5 and 6 where PS*(2) and PS*(4) often have statistically insignificant better performance than SSS* (i.e., PS*(1)) on order R = 4 trees.

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

Search method	R = 1 (random)	R = 2	R = 4	R = 8 (minimal)	Space needs
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 3. Average NBP on Trees with depth=4 and width=16.

Search method	R = 1 (random)	R = 2	R = 4	R = 16 (minimal)	Space needs
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 4. Average NBP on Trees with depth=4 and width=24.

For minimal trees of depth 4 and width 24 NBP=1151.

		prob=1.00		prob=0.90		
Search method	R = 1 (random)	R = 2	R = 4	R = 2	R = 4	Space needs
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 5. Average NBP on Trees with depth=4 and width=32.

Search method	R = 1 (random)	R = 2	R = 4	R = 8	R = 32 (minimal)	Space needs
SSS*	10816	8493	6424	4633	2047	1024
PS*(2)	13989	8478	6422	4632	2047	273
PS*(4)	16464	11089	6420	4632	2047	73
PS*(8)	18512	12782	8313	4631	2047	21
PS*(16)	20145	13966	9330	6209	2047	7
$\alpha\beta$	20836	14665	10046	6974	2047	4

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

Search method	R = 1 (random)	R = 2	R = 4	R = 8 (minimal)	Space needs
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

Choice of Partition Count

From the previous discussions, it is clear that selection of the partition count, k, is important if PS*(k) 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 the NBP would be low but space needed would be as much as for SSS*. Thus PS* forms a continuum of alternatives 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. Different ordering schemes must be considered, since ordered trees often are more typical of those appearing in applications than are random trees.

Storage needs are also significant. For example, SSS* needs 1024 entries in the OPEN list to search a tree of depth=4 and width=32, whereas PS*(4) requires 64+9 = 73, and PS*(8) needs only 16+5 = 21 for both the OPEN and BACKUP lists. Note that although PS* maintains two ordered lists, the total size of the two lists is much less than that of the single list of SSS*. Also, an ordered list of size 64 or 16 is much cheaper to maintain than a list of 1024 elements. Hence, the time spent by PS* manipulating these overhead lists may be less than that needed by SSS*.

CONCLUSION

The new algorithm PS*(k) can be viewed as a continuum between SSS* and $\alpha\beta$, as it attempts to make use of the best characteristics of both. The $\alpha\beta$ algorithm processes nodes in 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 update process. The phased search algorithm PS* also does some bookkeeping and achieves much better pruning than $\alpha\beta$ in a statistical sense. Since PS* concentrates only on a subset of the solution trees in each phase, it consequently needs smaller storage and may even require less execution time than SSS*. Thus PS*(k) can be comparable to SSS* in performance, especially on *bushy* trees (i.e., trees with w > 20), and yet at the same time has significantly lower storage overhead than SSS*. Because of the built-in flexibility provided by phasing and the possibility for choosing the partition size parameter (PSIZE), PS* is expected to be useful in practice. PS* becomes most efficient if parameter selection can be done using some a priori knowledge of the expected location of the solution.

Experimental results reported here are based on a game tree model, and the algorithm remains to be tested with a typical game-playing program. However, experience with other alternatives to $\alpha\beta$[7] shows that performance on probabilistic uniform trees is a good indicator of performance in a typical application[9]. 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 further work is necessary to determine if unequal partition sizes offer a performance advantage in practice. Certainly for probabilistically ordered trees increasing partition sizes could be useful.

Acknowledgements

Financial support in the form of Canadian Natural Sciences and Engineering Research Council Grant A7902 made the experimental work possible. Discussions with Liwu Li on theoretical points and the independent implementation of PS* by Erik Altmann helped and were appreciated.

References

1. G.C. Stockman, A minimax algorithm better than alpha-beta?, *Artificial Intelligence 12(2)*, (1979), 179-196.
2. D. Knuth and R. Moore, An analysis of alpha-beta pruning, *Artificial Intelligence 6(4)*, (1975), 293-326.
3. N. Srimani, A new algorithm (PS*) for searching game trees, M.Sc. thesis, Computing Science Dept., University of Alberta, Edmonton, July 1985.
4. I. Roizen and J. Pearl, A minimax algorithm better than alpha-beta? Yes and No., *Artificial Intelligence 21(2)*, (1983), 199-220.
5. 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.
6. T.A. Marsland and M. Campbell, Parallel search of strongly ordered game trees, *Computing Surveys 14(4)*, (1982), 533-551.
7. A. Reinefeld, J. Schaeffer and T.A. Marsland, Information acquisition in minimal window search, *Procs. 9th Int. Joint Conf. on Art. Intell.*, Los Angeles, 1985, 1040-1043.
8. M.S. Campbell and T.A. Marsland, A comparison of minimax tree search algorithms, *Artificial Intelligence 20(4)*, (1983), 347-367.
9. 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.

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

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

Case	Condition of the input state (n,s,hi)	Action of Γ
1.	s=LIVE, n is interior	
1a	Type(n) = MAX	Push states (n.j,s,hi) for all j=1,...,PSIZE onto the OPEN stack in reverse order. Push (n,PSIZE,low,hi) onto BACKUP, where low is the lower bound of n and hi is the upper bound. Note that, if n = root, then low=-∞ else low = low of P(i) stored in BACKUP.
1b	Type(n) = MIN.	Push (n.1,s,hi) onto the front of the OPEN list.
2.	s=LIVE, n is terminal	Set Score = Min(V(n),hi), where V(n) is the value returned by the evaluation function.
2a	Type(n) = MIN, or Score > low of P(i).	Insert (n,SOLVED,Score) into OPEN in front of all states of lesser merit. Ties are resolved in favor of nodes which are leftmost in the tree.
2b	Type(n) = MAX, Score ≤ low of P(i)	If i is the last node in the current partition at P(i), then Score is changed to low of P(i). Insert (i,SOLVED,Score) into OPEN, maintaining the order of the list.
3.	s = SOLVED, Type(n) = MAX.	Purge all successors of i = P(n) from BACKUP.
3a	m = w, n = root	Terminate: hi is the minimax value of the tree.
3b	hi > low of P(i), m < w.	Expand: push (i.m+1,LIVE,hi) onto the front of OPEN.
3c	Otherwise	Prune: push (i,SOLVED,hi) onto the front of OPEN.
4.	s = SOLVED, Type(n) = MIN,	Obtain values of low(i) and high(i) from BACKUP. Set low(i) = Max(low(i),hi) and update low for all descendants of i on BACKUP.
4a	If low(i) ≥ high(i)	Purge all successors of i from OPEN and BACKUP. Push (i,SOLVED,high(i)) onto the front of OPEN.
4b	If low(i) < high(i)	If there are incompletely searched MAX successors (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 Purge all successors of i from OPEN and BACKUP, and *either* push the next partition of successors of i onto OPEN *or*, if there are no more partitions, Push (i,SOLVED,low(i)) onto the front of OPEN.