Searching Game Trees Under Memory Constraints

Subir Bhattacharya and Amitava Bagchi
Indian Institute of Management Calcutta
Joka, Diamond Harbour Road
P. O. Box 16757, Calcutta - 700 027, INDIA
subir@iimcal.ernet.in

Abstract

The best-first game-tree search algorithm SSS* has greater pruning power than the depth-first algorithm Alpha-Beta. Yet it is seldom used in practice because it is slow in execution and requires substantial memory. Variants of SSS* have been proposed in recent years that overcome some, but not all, of its limitations. The recursive controlled-memory best-first search scheme MemSSS* described here is a new derivative of SSS* that compares favourably with Alpha-Beta in respect of all three major performance measures, namely, pruning power, running time and memory needs. MemSSS* improves upon an earlier controlled-memory algorithm IterSSS* which has most of the desired properties but is slow in execution.

1. Introduction

The best-first game-tree search algorithm SSS* [Stockman 1979] has greater pruning power than the depth-first algorithm Alpha-Beta. Yet SSS* is seldom used in practice. There are two main reasons for this. First, SSS* consumes a large amount of memory. Secondly, an excessive overhead is incurred during the execution of the algorithm. Modifications of SSS* have been proposed in recent years that overcome either the first or the second shortcoming. Consider the algorithms RecSSS* [Bhattacharya & Bagchi 1993] and RecDual* [Reinefeld & Ridinger 1994]. These have the same pruning power as SSS*, run about as fast as Alpha-Beta, but require even more memory than SSS*. Thus they succeed in reducing the overhead but not the memory needs. IterSSS* [Bhattacharya & Bagchi 1986], a memory-controlled version of SSS*, tries to make the best possible use of available memory. It prunes at least as many terminal nodes as Alpha-Beta, its pruning power being a function of the available memory. However, the overhead is comparable to that of SSS*. MGame [Bhattacharya & Bagchi 1994] was designed to surmount both limitations of SSS*, but it has proved only partially successful. It sometimes examines more terminal nodes than Alpha-Beta, and it is able to make use of only discrete slabs of memory. We are thus led to ask: Does there exist a best-first game-tree search method that compares favourably with Alpha-Beta in respect of all three performance measures: pruning power, running time and memory needs ?

In an effort to answer the question, this paper presents a new variant of SSS* called MemSSS* which combines the best features of RecSSS* and IterSSS*. MemSSS* has the following characteristics:

i) Its running time with full memory is comparable to that of RecSSS*, which is known to run faster than Alpha-Beta on random uniform trees [Reinefeld & Ridinger 1994].

ii) It puts to effective use whatever memory is available for OPEN, the minimum memory needed for OPEN being comparable to the stack space required by Alpha-Beta.

iii) It runs faster than Alpha-Beta on uniform strongly ordered trees of odd depth even when much less memory is available than that needed by RecSSS*.

iv) It never examines a terminal node pruned by Alpha-Beta, not even at minimum memory. Its pruning power increases fairly uniformly with available memory.

MemSSS* has been obtained by pulling IterSSS* apart and then recombining the parts in the manner of RecSSS*. It rivals Alpha-Beta in overall performance, establishing that best-first search in game trees can compete on equal terms with depth-first search. Its control over the use of memory might even give it an edge over Alpha-Beta.

We assume below, without loss of generality, that the root node s of the game tree T to be searched is a MAX node. The *depth* of a node p equals the length of the path from s to p; the depth of s is 0. The length of the longest path in T is denoted by d; b is the *branching degree*, and denotes the maximum number of sons of a non-terminal node in T. T is a *uniform (b,d) tree* if every non-terminal node has exactly b sons and every terminal node has depth d. Nodes are represented using a Dewey code [Pearl 1984]; the root node corresponds to the empty sequence, and the $w \leq b$ sons of a non-terminal node x are represented as $x.i$, $1 \leq i \leq w$. A *static evaluation function v(.)* scores each terminal node from MAX's point of view.

The next section takes a quick look at IterSSS*. Section 3 presents the new algorithm MemSSS*. Section 4 contains test results comparing MemSSS* with Alpha-Beta, IterSSS* and MGame. Section 5 concludes the paper.

2. A Quick Look at IterSSS*

The best-first algorithm SSS* stores the frontier (or tip) nodes of the currently explored portion of the game tree T in a global list called OPEN. Each such node serves as a representative of the set of solution trees that contain it. The representatives of all solution trees in T contend with each other in OPEN, and the node that is most promising among them gets chosen for further exploration. The size of OPEN for a uniform *(b,d)* tree is $b^{\lceil d/2 \rceil}$. IterSSS* differs from SSS* in that the size of OPEN is limited to the available memory M. At any given instant the represent-

ative nodes of only some of the solution trees can be accommodated in M, and contention gets restricted to these nodes. Other nodes of the explicit tree are kept inactivated. Each inactive node represents a set of solution trees currently debarred from contention with other solution trees. The best among the active nodes is found, and if this node happens to be solved, nodes representing provably sub-optimal solution trees are purged from OPEN. A node that is inactive is then activated, the space just released in OPEN being allocated for exploring the game tree below this node. The search proceeds in this way until the entire tree is traversed and the optimal solution tree is found.

The memory M available for use by OPEN is supplied to IterSSS* as input. A node x in OPEN has three fields: h and *status* as in SSS*, and an additional field *type* which takes the value ACTIVE or INACTIVE. At start, IterSSS* puts the root node s in OPEN with h(s) = ∞ (infinity), status(s) = LIVE and type(s) = ACTIVE. The algorithm is similar to SSS*; an additional check ensures that the total memory deployed for OPEN never exceeds M. Let a LIVE ACTIVE MAX node x be selected from OPEN. If not enough memory is currently available for exploration of the tree below x, then x is not expanded and is labelled INACTIVE. In other words, solution trees containing x are held back from contention for the time being. An inactive node in OPEN can be viewed as a node in 'suspended animation', exploration below which has been deferred owing to lack of memory. When a solved MIN node p with h-value h(p) is selected from OPEN, the parent node x of p can not be immediately declared solved as in SSS*, since there might be an inactive node below x in OPEN with h-value > h(p). If there is no such node, x can be declared solved; else, the lexicographically smallest such inactive descendant y of x is activated and p remains solved in OPEN. This brings y into contention; the subtree below y is searched using the memory released by the solved node p. Since any inactive successor z of x must be at a depth $\geq$ depth(p), the space released below p is adequate for exploration below z. To run IterSSS* on a uniform *(b,d)* tree, OPEN must provide room for at least $(b-1)*\lceil d/2 \rceil + 1$ nodes, namely *(b-1)* inactive nodes at non-terminal MAX levels and b terminal nodes.

An inactive node in the game tree has a Dewey code that is lexicographically larger than that of any active node. The lexicographically smallest inactive node gets activated when memory becomes available. Thus IterSSS* induces a partitioning along the horizontal axis, and falls in between SSS*, which is non-directional, and Alpha-Beta, which is fully directional. The degree of directionality of IterSSS* decreases as more memory is made available. In contrast, MGame partitions a game tree along the vertical axis into trees of smaller height and runs RecSSS* on each of them.

3. The Proposed Scheme

SSS* and IterSSS* both need to perform the following two tasks quite frequently: i) At each iteration, the node with the highest h-value must be selected from OPEN, which is typically a large list. ii) Whenever a non-terminal MAX node p gets solved, terminal nodes that are descendants of p must be purged from OPEN; these nodes, if allowed to remain in OPEN, may later cause errors in the selection of the highest h-valued node. This need for purging makes it inconvenient to implement OPEN as a priority queue, and as a result the above two tasks can not be achieved efficiently together. Recursion helps RecSSS* to overcome this problem. In SSS*, OPEN contains the nodes at the current search frontier. In RecSSS*, OPEN has a slightly different structure; it also contains the non-terminal MAX predecessors of the frontier nodes. It is maintained as an array containing only MAX nodes and terminal MIN nodes (if any). At any instant, for each non-terminal MAX node p, OPEN contains one grandson of p below each son of p. An entry for a node p in OPEN has four fields: the Dewey code of p, h(p), status(p) and expanded(p). The first three have the same meaning as in SSS*. The *expanded* field is set to YES if p has been expanded and to NO otherwise; its use nullifies the need to purge nodes from OPEN.

RecSSS* has the following interesting properties:

i) It evaluates the same terminal nodes as SSS* in the same order when ties in h-values are resolved identically.

ii) It is faster than SSS* at spotting the next frontier node to examine. Since OPEN contains the non-terminal MAX nodes that lie at intermediate levels, it can make a series of local searches below the MAX nodes, each local search involving a scan of only b grandsons, instead of a global search of the entire OPEN list.

iii) *RecSSS* does not need to purge nodes from OPEN.* A significant amount of execution time is thereby saved. Suppose a MAX node p has been solved, and p has a right brother q. Then p is replaced by q in OPEN, and expanded(q) is reset to NO. Entries in OPEN that contain nodes belonging to the subtree below p can henceforth be treated as unoccupied. These positions get overwritten by descendants of q if and when RecSSS* gets called at q at a later instant of time.

iv) For a uniform *(b,d)* tree the number of nodes in the OPEN list of RecSSS* jumps by a factor of b per MAX level; so the number of entries in OPEN is greater than in SSS* and is given by the expression:

$$1 + b + b^2 + \ldots + b^{\lceil d/2 \rceil} \simeq b^{\lceil d/2 \rceil} * b/(b-1).$$

It would seem that the only way to speed up IterSSS* is through a similar structural transformation. This involves the following changes. The *status* field of a node in OPEN is made tri-valued: LIVE, SOLVED and INACTIVE. This field is the union of the status and type fields of IterSSS*. The notion of an active node is not directly used by the algorithm; an active node can be viewed as one that is either LIVE or SOLVED. (Note that the status field here is not the same as the tri-valued status field used in [Reinefeld & Ridinger 1994].) In IterSSS*, the free memory that is available for allocation is held in common; whenever memory is needed, allocation is made from this common pool. When a node is added to (or deleted from) OPEN, there is a corresponding decrease (or increase) in the free memory. MemSSS* is a recursive algorithm and it adopts

a different strategy. It allocates memory to non-terminal MAX nodes individually; the memory is used for searching the subtree below the node. Let p be a non-terminal MAX node in a uniform *(b,d)* game tree that enters OPEN at some time during the execution of MemSSS*. When p gets generated it is allotted an amount *mem(p)* of memory; this memory is used for searching the subtree rooted at p. If and when p is expanded at a later instant, b units of memory store the grandsons $p.i.1$ of p. The remaining mem(p)-b units are distributed among the grandsons in the following manner. Let the node $p.i.1$ be at a depth t, $0 < t < d$. Define the two parameters *minmem(t)* and *maxmem(t)* as follows:

$$\text{minmem}(t) = b^{*}\lceil (d-t)\rceil/2;$$
$$\text{maxmem}(t) = (b^{\lceil (d-t)\rceil/2+1}-1)/(b-1)-1.$$

Here minmem(t) is the minimum memory needed for running MemSSS* on the subtree rooted at $p.i.1$, and maxmem(t) is the memory needed for running RecSSS* on the same subtree. (The memory used for storing $p.i.1$ is not included in these expressions.) The memory mem(p)-b is distributed among the grandsons $p.i.1$ in left-to-right order starting from $p.1.1$. When a node $p.i.1$ is reached during the course of memory allocation, if the remaining memory is found to be < minmem(t) then $p.i.1$ and the grandsons of p to the right of $p.i.1$ in OPEN are inactivated and no memory is allocated to them. On the other hand, if the available memory m is $\geq$ minmem(t) but < maxmem(t) when $p.i.1$ is reached, then all of m is allocated to $p.i.1$, and the grandsons to the right of $p.i.1$ in OPEN are made inactive. The scheme ensures that an inactive node has a Dewey code that is lexicographically greater than the codes of all active nodes.

Algorithm MemSSS* given below consists of a main routine MemSSSroot and a recursive procedure MemSSS. An amount M of memory is made available to MemSSSroot for the global OPEN list. MemSSSroot inserts the root node s of the game tree in OPEN, sets h(s) to ∞ (infinity), and calls MemSSS repeatedly until s gets SOLVED. The procedure *insert* that enters a new node into OPEN has five arguments, namely, the Dewey code of the node, h-value, status, expanded and memory allocated. The root node s is stored in one unit of memory and so M-1 units are allocated to s for searching the tree below it.

MemSSS is similar to the RecSSS procedure of RecSSS* up to the end of the **while** loop, except that only active (i.e., LIVE or SOLVED) nodes are under consideration and memory is allocated to individual nodes that enter OPEN. Let x be the current highest-valued active grandson of p in OPEN right after exit from the **while** loop. If x is LIVE, h(x) defines a tighter upper bound on p, so we update h(p) and return. If x is solved, p can be declared solved only if there is no inactive descendant of p in OPEN with h-value > h(x). Otherwise, the lexicographically smallest such descendant z of p is activated and allocated memory.

The algorithm allocates to node z the memory released below the SOLVED node x. Suppose $z = y.i.j$. Then y is a descendant of p. In practice, it is simpler and more efficient

```
MemSSSroot (s: node; M: integer);
begin
 insert(s, ∞, LIVE, NO, M-1);
 while (s is not SOLVED) do MemSSS(s);
 return h(s);
end;
MemSSS (p: node);
var x, z: node; m, i, j: integer;
begin
 if p is a terminal node then begin
   status(p) := SOLVED; h(p) := minimum(h(p), v(p)); return;
 end;
 if expanded(p) = NO then begin
   expanded(p) := YES; m := mem(p)-b;
   for i = 1 to b do begin
     if p.i is a terminal node then
       insert(p.i, h(p), LIVE, NO, 0)
     else if p.i.1 is a terminal node then
       insert(p.i.1, h(p), LIVE, NO, 0)
     else if m < minmem(t) then      { where t = depth(p.i.1) }
       insert(p.i.1, h(p), INACTIVE, NO, 0)
     else insert(p.i.1, h(p), LIVE, NO, minimum(m, maxmem(t)));
     update m;
   end;
 end;
 x := immediate LIVE successor of p in OPEN with highest h-
   value;
 while h(x) = h(p) and x is LIVE do begin
   MemSSS(x);                                  { x = p.i.j, say }
   if x is SOLVED and has a MAX right brother then
     replace x in OPEN with (p.i.j+1, h(x), LIVE, NO, mem(x));
   x := immediate LIVE or SOLVED successor of p in OPEN
     with highest h-value;
 end;                                          { end while }
 if x is LIVE then h(p) := h(x)
 else if p has no INACTIVE descendant z with h(z) > h(x) then
   begin h(p) := h(x); status(p) := SOLVED; end
 else begin                                    { see explanation }
   z := lexicographically smallest INACTIVE descendant of p in
     OPEN with h(z) > h(x);
   status(z) := LIVE; mem(z) := mem(x);
   for each node q in OPEN lying on the path from p to z do
     h(q) := h(z);
 end;
end;
```

to allocate to z the memory that is currently allocated to the leftmost grandson $y.1.k$ of y in OPEN. This is a valid step because: (i) the depth of y is greater than that of p; (ii) $y.1.k$ had been allocated the largest amount of memory among the grandsons of y; (iii) node x, which is a grandson of node p, was selected from OPEN in the solved state, so the subtree rooted at $y.1.k$ must be suboptimal.

The h-values of nodes in OPEN that lie on the path from p to z are updated by MemSSS*. Why is this necessary ? At any instant during execution, the h-value of a node reflects the highest h-value of its active descendants. When node z is activated, h-values of nodes along the path from

p to *z* should increase to h(*z*). To facilitate this task, a backward link from a node to its grandfather is provided in OPEN. The h-values of the ancestors of *p* get updated automatically because the algorithm is recursive.

To identify the leftmost inactive successor below a node quickly and easily, a list of inactive nodes is maintained sorted on depth. This list is small since there can be at most $(b-1)*\lceil (d-2)/2 \rceil$ inactive nodes at any time. Note that nodes become inactive in non-decreasing order of depth.

Figure 1 shows the terminal nodes that are examined by MemSSS* when run on a uniform (2, 6) game tree for various values of available memory *M* between 7 and 15. The tree has 64 terminal nodes with the static evaluation scores shown. A hyphen (-) indicates a *don't care* value and a cross (x) indicates that the corresponding terminal node has been examined by the algorithm. MemSSS* never examines a terminal not examined by Alpha-Beta, not even when $M = 7$, which is the minimum memory required to run the algorithm. RecSSS* needs 15 units of memory to run on this game tree; when $M = 15$, MemSSS* becomes equivalent to RecSSS*. The number of terminal nodes examined by MemSSS* tends to decrease as *M* increases. However, for two given values m_1 and m_2 of *M*, where $m_1 < m_2$, it is possible that a terminal node examined when $M = m_2$ is not examined when $M = m_1$. In Fig. 1 for example, terminal nodes having Dewey codes 211211, 211221 and 211222 with values 3, 4 and 5 respectively are examined when $M = 13$ but not when $M = 9$. When $M = 13$, there is only one inactive node, namely 2121, which becomes live when node 2112 is selected in solved state, causing the above terminals to be evaluated. But when $M = 9$, the only inactive node, 21, is activated when node 12 is selected in solved state. With the released memory, the entire subtree below 21 can be searched as in the manner of RecSSS*. Since the subtree below 2121 dominates the subtree below 2111, terminal nodes below 2112 do not get examined.

Partial pruning conditions for IterSSS* were derived in [Bhattacharya & Bagchi 1986]. These apply to MemSSS* as well. Since MemSSS*, unlike IterSSS*, allocates memory to individual nodes, complete pruning conditions for the two algorithms, when found, would not be identical. The fact that MemSSS* never examines a terminal node pruned by Alpha-Beta has a simple intuitive explanation. Since inactive nodes lie to the right of all active nodes, MemSSS* always has complete information about the tree to the left of the active node currently being visited; it is therefore at least as well informed as Alpha-Beta, and prunes all terminal nodes pruned by Alpha-Beta.

4. Experimental Results

MemSSS* was compared experimentally with Alpha-Beta and with other memory-controlled variants of SSS* like MGame and IterSSS*. All the algorithms were programmed in C and run on a UNIX-based 64-bit DEC Alphastation 250-4/266 with 128 MB of memory. Two sets of experiments were conducted. In the first set, MemSSS* was compared with Alpha-Beta and MGame. In the second set, it was compared with Alpha-Beta and IterSSS*. The method of generation of terminal node scores was different for the two sets as explained below.

In the first set, for each of four *(b,d)* pairs, 100 randomly ordered uniform game trees and 100 strongly ordered uniform game trees were generated. For each pair and each type of ordering, the running time and the number of terminal nodes examined were averaged over the 100 runs. MemSSS* was run for six different values of total available memory *M*, namely, the minimum, the maximum, and four intermediate values. The intermediate values of *M* were so chosen that the ratio of any two successive values was roughly the same. At the maximum value of *M*, MemSSS* became identical to RecSSS*. MGame was run for *searchd* = 2 and *searchd* = *d*. MGame with *searchd* = 2 is QuickGame, which needs as much memory as the minimum memory of MemSSS* [Bhattacharya 1995]; MGame with *searchd* = *d* is RecSSS*, which needs as much memory as the maximum memory of MemSSS*. Since *searchd* takes only discrete integer values, for some values of *M* no exact corresponding value of *searchd* can be found; in view of this, MGame was not run at intermediate values of *searchd*. When MemSSS* was run on trees with odd depth, terminal nodes were *not* stored in OPEN. Instead, a simple search as in Alpha-Beta was conducted at the lowest MAX level; as pointed out in [Reinefeld & Ridinger 1994], this tends to speed up execution. Random numbers were generated using the random() and srandom() functions; it is known that random numbers obtained with these functions have far superior randomness properties than those obtained with rand() and srand(). The srandom() function was initialized with proper seeds at appropriate moments during execution to ensure that the three algorithms Alpha-Beta, MGame and MemSSS* ran on identical trees. Strongly ordered trees were formed as suggested in [Marsland *et al* 1987]; at an internal node, the optimal choice was the leftmost son at least 70% of the time, and lay among the leftmost 25% of the sons 90% of the time. Thus, if b = 8, the probability that a specific son was the optimal choice was given by (0.7, 0.2, 0.0166, 0.0166,..., 0.017).

```
172543-- 3645 2-35 621- 46-- 53-- 1-54 642- 3-45 752- 56-- 57-- 3-65 ---- ----

xxxxxx   xxxx x xx xxx  xx   xx   x xx xxx  x xx xxx  xx   xx   x xx  (39 Alpha-Beta)
x xxxx   x xx x xx xxx  xx   xx   x xx xxx  x xx xxx  xx   xx   x xx  (37 MemSSS*  M = 7)
x x      x xx x xx xxx       xx   x xx xxx       xxx  xx   xx   x xx  (29 MemSSS*  M = 9)
x x      x xx x x                      xxx  x xx xxx  xx   xx   x xx  (23 MemSSS*  M = 13)
x x      x x                           xxx       xxx  xx   xx   x xx  (17 MemSSS*  M = 15)
```

Figure 1: Terminals examined by Alpha-Beta and MemSSS* in a (2,6) uniform tree

Table 1 : Comparison of Alpha-Beta, MemSSS* and MGame

	Alpha-Beta				MemSSS*					MGame				
	Random		Strong			Random		Strong			Random		Strong	
(b, d)	terminal count	time secs	terminal count	time secs	memory	terminal count	time secs	terminal count	time secs	searchd	terminal count	time secs	terminal count	time secs
(5,10)	114,525	10.80	8,992	1.02	26	84,482	12.63	8,433	1.49	2	76,048	10.04	18,963	3.06
					70	70,138	10.98	8,192	1.49					
					190	62,871	9.93	7,856	1.43					
					517	50,984	8.17	7,660	1.39					
					1,408	41,157	6.63	7,431	1.35					
					3,906	31,758	5.11	7,219	1.30	10	31,758	5.10	7,219	1.30
(6,9)	116,270	10.14	16,960	1.60	25	89,186	9.62	15,322	1.78	2	78,314	7.86	21,965	2.56
					57	74,338	8.10	14,801	1.72					
					130	67,178	7.28	14,209	1.61					
					296	61,906	6.68	14,031	1.58					
					676	52,700	5.63	13,402	1.47					
					1,555	43,059	4.52	13,290	1.45	8	43,059	4.52	13,290	1.45
(9,8)	250,305	20.47	29,279	2.74	37	181,696	25.89	25,114	4.23	2	148,467	18.52	43,692	6.27
					106	148,601	21.99	23,529	4.03					
					305	130,106	19.38	21,971	3.76					
					879	117,005	17.42	21,465	3.66					
					2,534	98,500	14.70	19,310	3.27					
					7,381	74,948	11.16	18,470	3.12	8	74,948	11.13	18,470	3.11
(10,7)	106,636	8.27	18,701	1.52	31	81,026	7.56	17,325	1.66	2	68,679	5.97	21,261	2.00
					63	71,915	6.67	16,799	1.58					
					128	63,881	5.90	16,681	1.57					
					261	61,376	5.63	16,291	1.50					
					533	54,268	4.91	15,869	1.44					
					1,111	47,069	4.19	15,645	1.41	6	47,069	4.19	15,645	1.41

Table 1 compares Alpha-Beta, MemSSS* and MGame. In these runs, the time spent to evaluate a terminal is negligible. The running times include the time to generate the tree in a specified order as well as the overhead of the respective algorithms. The following observations can be made on the experimental results.

Strongly ordered trees: i) MemSSS* examined fewer terminal nodes than Alpha-Beta *and ran faster than Alpha-Beta on strongly ordered trees of odd depth*, even when the available memory was less than that required by RecSSS*. For example, when run on (6,9) uniform trees, MemSSS* dominated Alpha-Beta in terms of both terminal count and running time when the memory supplied was 296, which is less than 20% of the memory needed to run RecSSS*.

ii) On strong trees of even depth, MemSSS* ran slower than Alpha-Beta. But, on the basis of the known properties of RecDual* [Reinefeld & Ridinger 1994], it is reasonable to expect that MemDual*, the counterpart of MemSSS* that can be obtained by modifying RecDual*, would run faster than Alpha-Beta on such trees. iii) The performance of QuickGame was weakest on strongly ordered trees.

Randomly ordered trees: i) QuickGame (*i.e.*, MGame with *searchd* = 2) was superior to MemSSS* at minimum M in terms of running time and the number of terminal nodes examined. QuickGame was also superior to Alpha-Beta. Thus on random trees when the available memory M is small, QuickGame seems to be preferable to both MemSSS* and Alpha-Beta. ii) MemSSS* examined far fewer terminal nodes than Alpha-Beta. When full memory was made available, the number of terminal nodes examined by MemSSS* was around a third of the number examined by Alpha-Beta, and MemSSS* ran about twice as fast. This agrees with the reports of Reinefeld and Ridinger [1994]. As expected, MemSSS* and MGame at maximum M are very close in performance, both being equivalent to RecSSS*. A direct coding of RecSSS* would have lower overhead and would run slightly faster than either MemSSS* or MGame. iii) When run on odd depth trees, MemSSS* ran faster than Alpha-Beta even at minimum memory. iv) The number of terminal nodes examined by MemSSS* decreased substantially with increase of memory.

In this set of experiments no time was taken to compute the static evaluation scores of terminal nodes. In actual game playing situations the computation time would be large, and Alpha-Beta would run slower than MemSSS* because it would examine many more terminal nodes.

Table 2: Comparison of MemSSS* and IterSSS*

	Alpha-Beta			MemSSS*		IterSSS*	
	terminal	time		terminal	time	terminal	time
(b,d)	count	secs	mem	count	secs	count	secs
(4,10)	51,135	3.15	21	50,251	3.15	49,218	3.24
			48	48,347	3.03	47,376	3.25
			110	46,027	2.88	45,162	3.34
			253	42,670	2.67	40,909	3.64
			583	39,864	2.50	38,153	4.44
			1,365	34,643	2.17	34.651	6.06
(6,8)	77,604	4.76	25	76,255	4.78	76,122	4.92
			57	71,975	4.50	72,451	4.84
			130	69,910	4.37	69,071	4.92
			296	65,288	4.09	64,571	5.28
			676	61,971	3.89	59,781	6.41
			1,555	52,949	3.32	53,112	8.86

Recommendations: On strongly ordered trees use MemSSS* with available memory as an input parameter. On random trees, QuickGame or MGame with appropriate *searchd* would be the right choice.

How much faster is MemSSS* than IterSSS*? It is known that IterSSS* runs quite fast at minimum memory. But as the available memory increases, its running time increases quite dramatically in spite of a marked decrease in the number of terminal nodes examined. This is due to the overhead of searching OPEN. IterSSS* stores in OPEN only the nodes at the search frontier, so it is not possible to generate randomly ordered and strongly ordered game trees dynamically. Randomly ordered game trees were generated for two *(b,d)* pairs using a different scheme. The score of a terminal node was computed directly from the pair (Dewey code, run number) using srandom() and random(). No calls were made to the random number generators at non-terminal nodes. Runtimes were averaged over 100 runs. No runs were made on strongly ordered trees as such trees can not be generated in the manner described above.

Table 2 compares MemSSS* with IterSSS*. It can be seen that MemSSS* ran almost as fast as Alpha-Beta when the available memory M was minimum; IterSSS* was marginally slower. But as M increased MemSSS* speeded up while IterSSS* slowed down. At maximum memory, MemSSS* was somewhat faster than Alpha-Beta and was 2.5 to 3 times as fast as IterSSS*. An available memory of M has different meanings for IterSSS* and MemSSS*; for the same value of M, IterSSS* can store many more terminal nodes in OPEN than MemSSS*, since it does not store any non-terminal MAX nodes. This is why for a given M, IterSSS* typically examines fewer terminals than MemSSS*. The terminal count for maximum memory is not the same for the two algorithms since ties are resolved differently. On uniform trees of odd depth MemSSS* is much faster than IterSSS* because MemSSS* does not store terminal nodes in OPEN for such trees. This feature can not be incorporated in IterSSS* in a natural way, so we do not give running times for odd-depth trees.

5. Conclusion

This paper describes a new memory-controlled game-tree search algorithm that improves upon the existing methods IterSSS* and MGame. IterSSS* has a number of attractive properties, but it tends to slow down as the size of OPEN increases. RecSSS*, a recursive version of SSS*, runs much faster than SSS* and is about as fast as Alpha-Beta. This suggests that if IterSSS* is converted to a RecSSS*-like format, an improvement in speed might be achieved. The resulting algorithm is MemSSS*. It has all the desirable properties of IterSSS*, and runs much faster than IterSSS* when the memory available for OPEN is large. It is also superior to MGame in that it never examines more terminal nodes than Alpha-Beta; moreover, it is capable of putting any amount of available memory to fruitful use, unlike MGame which can utilize memory only in discrete slabs. It examines far fewer terminal nodes than Alpha-Beta on randomly ordered trees. *Most importantly*, MemSSS* rivals Alpha-Beta even on strongly ordered trees in both running time and pruning power.

It remains to determine the exact pruning conditions for MemSSS*. Partial pruning conditions for IterSSS* were derived in [Bhattacharya & Bagchi 1986], and similar conditions will hold for MemSSS* as well. But the exact pruning conditions for the two algorithms would differ because of the different memory allocation schemes.

References

[Bhattacharya 1995] Subir Bhattacharya, Experimenting with Revisits in Game Tree Search, *Proc IJCAI-95*, 14th International Joint Conference on Artificial Intelligence, Montreal, August 1995, vol I, pp 243-249.

[Bhattacharya & Bagchi 1994] Subir Bhattacharya and A. Bagchi, A General Framework for Minimax Search in Game Trees, *Info Proc Letters*, vol 52, 1994, pp 295-301.

[Bhattacharya & Bagchi 1993] Subir Bhattacharya and A. Bagchi, A Faster Alternative to SSS* with Extension to Variable Memory, *Info Proc Letters*, vol 47, 1993, pp 209-214.

[Bhattacharya & Bagchi 1986] Subir Bhattacharya and A. Bagchi, Making Best Use of Available Memory When Searching Game Trees, *Proc AAAI-86*, American Association for Artificial Intelligence, Philadelphia, August 1986, pp 163-167.

[Marsland *et al* 1987] T. A. Marsland, A. Reinefeld and J. Schaeffer, Low Overhead Alternatives to SSS*, *ArtifIntel*, vol 31, 1987, pp 185-199.

[Pearl 1984] J. Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley, Reading Mass, 1984

[Reinefeld & Ridinger 1994] A. Reinefeld and P. Ridinger, Time-Efficient State Space Search, *ArtifIntel*, vol 71, 1994, pp 397-408

[Stockman 1979] G. C. Stockman, A Minimax Algorithm Better Than Alpha-Beta ?, *ArtifIntel*, vol 12, 1979, pp 179-196.