MAKING BEST USE OF AVAILABLE MEMORY WHEN SEARCHING GAME TREES

Subir Bhattacharya and Amitava Bagchi

Indian Institute of Management Calcutta
P.O. Box.16757, Calcutta-700027, INDIA

ABSTRACT

When searching game trees, Algorithm SSS* examines fewer terminal nodes than the alphabeta procedure, but has the disadvantage that the storage space required by it is much greater. ITERSSS* is a modified version of SSS* that does not suffer from this limitation. The memory M that is available for use by the OPEN list can be fed as a parameter to ITERSSS* at run time. For successful operation M must lie above a threshold value M_o . But M_o is small in magnitude and is of the same order as the memory requirement of the alphabeta procedure. The number of terminal nodes of the game tree examined by ITERSSS* is a function of M, but is never greater than the number of terminals examined by the alphabeta procedure. For large enough M, ITERSSS* is identical in operation to SSS*.

1. Introduction :

The alphabeta procedure is the best known of game tree search algorithms. Generally formulated as a recursive procedure, it is quite fast in execution and uses little memory. In the process of computing the minimax value at the root of the game tree it makes a left-to-right scan of the terminal nodes; it does not examine all the terminals, but looks only at those that to it appear capable of influencing the root value. Detailed expositions can be found in Knuth and Moore [2] and Pearl [3].

In 1979, Stockman [4] announced a new game tree search algorithm called SSS* quite different in nature from alphabeta. SSS* does not examine terminal nodes of the game tree in a left-to-right manner. It views a game tree as a union of its constituent solution trees, and at each step during the search, selects for inspection the most promising among the contending solution trees. The terminal nodes of this solution tree are examined in a left-to-right manner; if this not the best solution tree in the game tree, then a time comes when a more promising solution tree is found and that is then taken up for inspection. One node from each solution tree is kept in a list called OPEN. Nodes in OPEN have associated heuristic values, and OPEN is maintained as a priority queue with nodes having higher heuristic values at higher levels. At each iteration of the algorithm the node at the root of the priority queue is selected for examination. For a uniform game tree of depth d and branching factor b, SSS* requires $O(b^{d/2})$ cells of storage for the OPEN list. In contrast, the total storage required by alphabeta is O(d).

SSS* has an advantage over alphabeta in that it examines only a subset of the terminal nodes examined by alphabeta. Since the running time of a game-tree search algorithm is primarily determined by the number of terminal nodes it examines, SSS* should run faster than alphabeta. According to Pearl [3, p. 310], however, on the average alphabeta examines at most three times as many terminals as SSS*, and the much greater memory and bookkeeping requirements of SSS* tend to weigh the scale in favour of alphabeta.

In this paper we present a modified version of SSS* which we call ITERSSS*. The memory M that is available for the list OPEN is fed as a parameter to ITERSSS* at execution time, which then runs essentially like SSS* but not using more memory than M. For a uniform game tree of depth d and branching factor b, the minimum allowable value of M is $M_o = \lceil d/2 \rceil \cdot (b - 1) + 1$ when the root is a MAX node. So long as M is greater than this threshold value, ITERSSS* runs smoothly and outputs the minimax value at the root of the game tree. ITERSSS* examines more terminals than SSS* but fewer terminals than alphabeta, the number of terminals examined being a function of M. When $M = b^{\lceil d/2 \rceil}$, ITERSSS* is identical in operation to SSS*. When $M = M_o$, ITERSSS* examines no more terminals than alphabeta, and uses the same order of memory as alphabeta. From a programming point of view, ITERSSS* is of the same level of complexity as SSS*, and the flexibility it provides with regard to use of memory should give it an edge over both SSS* and alphabeta.

In section 2 of this paper we give formulations of SSS* and ITERSSS*, and in section 3 we describe some experimental results. Section 4 presents a few formal properties of game tree search algorithms, and Section 5 summarizes the paper and lists some open problems.

2. Algorithms SSS* and ITERSSS* :

We assume the root s of the game tree T to be a MAX (i.e. OR) node. The sons of s are then MIN (i.e. AND) nodes. The game tree T is also assumed to have a finite minimax value. A Dewey radix-b code is used for representing the nodes in T. We suppose that T is a uniform tree of depth d and branching factor b. Then

i) the root s is represented by the empty sequence ;
ii) the sons of nonterminal node x are represented as x.j, $1 \leq j \leq b$.

The maximum possible length of the Dewey code is d digits, and terminal nodes in T have d digit codes. The definition can be readily generalized to non-uniform trees. The nodes of T get linearly ordered by the lexicographic ordering of their Dewey codes. We also note that each terminal node x in T has a static evaluation score v(x).

A standard formulation of the alphabeta procedure using the minimax approach can be found in Knuth [2, p.300], and this is the version of the algorithm used by us in our experimental investigations described in the next section. We now present SSS*. This algorithm maintains a list called OPEN that is initially empty. Each node x in T has an associated heuristic value h(x), which gives the current value of node x; the node x also has another field called STATUS (x), which is either LIVE or SOLVED. The function first (OPEN) returns a node x having the current maximum h-value in OPEN; ties are always resolved in favour of lexicographically smaller nodes. The procedure is invoked by calling SSS*(s).

Procedure SSS* (s)

```
begin
  OPEN := { s } ; h(s) := ∞ ; STATUS (s) := LIVE;
  repeat
     x := first (OPEN);
     case : x is terminal and STATUS(x) = LIVE :
              h(x) := min (h(x), v(x));STATUS (x)
                          := SOLVED ;
          : x is a nonterminal MIN node and
                   STATUS(x) = LIVE :
              remove x from OPEN;
              insert x.1 in OPEN with h(x.1):=h(x),
                      STATUS (x.1) := LIVE;
          : x is a nonterminal MAX node and
                      STATUS(x) = LIVE :
              remove x from OPEN;
              insert x.j in OPEN with h(x.j):=h(x),
                      STATUS(x.j):=LIVE for 1≤j≤b;
          : x = x'.j is a MIN node and STATUS(x)
                      = SOLVED :
              remove all successors of x' from OPEN;
              insert x' in OPEN with h(x'):=h(x),
                      STATUS(x') := SOLVED;
          : x = x'.j is a MAX node and x ≠ s and
                      STATUS(x)  = SOLVED :
              remove x from OPEN ;
              if j = b then
                 insert x' in OPEN with h(x'):=h(x),
                      STATUS(x') := SOLVED
              else (* 1 ≤ j ≤ b *)
                 insert x'.j+1 in OPEN with
                      h(x'.j+1) := h(x),
                      STATUS (x'.j+1) := LIVE ;
  until  x = s and STATUS (x) = SOLVED ;
  output h(x) ;
end ;
```

Example 2.1 : The game tree T shown in Fig. 1 has b = 3, d = 4, and 81 terminal nodes. A terminal node x is said to be examined only when its assigned value v(x) is computed. Alphabeta needs to do this for 41 of the terminal nodes, while SSS* needs to do this for only 28 of them.

Algorithm ITERSSS* is very similar to SSS*. Here too a list OPEN is maintained, but the size of OPEN is constrained by the availability of storage. A node in OPEN, in addition to h and STATUS fields, also has a TYPE field. The TYPE of a node can be either ACTIVE or INACTIVE. The procedure is invoked by calling ITERSSS* (s,M) where s is the root of the game tree and M the amount of storage that OPEN can use. It is assumed that $M \geq M_0$. As in SSS*, OPEN is initially empty, and ties for selection from OPEN are resolved in favour of lexicographically smaller nodes.

Procedure ITERSSS* (s, M)

```
begin
  SPACE := M ;
  OPEN := { s }; STATUS(s):= LIVE; h(s):=∞ ;
                 TYPE(s) := INACTIVE ;
  SPACE := SPACE - 1 ;  FLAG := INACTIVE ;

  repeat
    x := first (OPEN, FLAG) ;
    case : x is a terminal node and STATUS(x) =
                                            LIVE :
             h(x) := min(h(x), v(x)); STATUS(x):=
                     SOLVED; TYPE(x) := ACTIVE ;
         : x is a nonterminal MIN node and
                     STATUS(x) = LIVE :
             remove x from OPEN ;
             insert x.1 in OPEN with h(x.1):= h(x),
                     STATUS (x.1) := LIVE,
                     TYPE (x.1) := FLAG;
         : x is a nonterminal MAX node and
                     STATUS (x) = LIVE :
             if SPACE ≥ b - 1 then
               begin
                 remove x from OPEN ;
                 insert x.j in OPEN with h(x.j):=
                     h(x), STATUS(x.j):= LIVE,
                     TYPE(x.j):=FLAG for 1 ≤ j ≤ b;
                 SPACE := SPACE - b + 1 ;
               end
             else
               begin
                 TYPE(x)  := INACTIVE ;
                 FLAG :=  ACTIVE ;
               end;
         : x = x'.j is  a MAX node and x ≠ s and
                     STATUS (x) = SOLVED :
             remove x from OPEN ;
             if j = b then
                insert x' in OPEN with h(x'):=h(x),
                     STATUS(x') := SOLVED,
                     TYPE(x') := ACTIVE
             else (* 1≤ j ≤ b*)
               insert x'.j+1 in OPEN with h(x'.j+1)
                     :=  h(x),
```

```
      STATUS(x'.j + 1) := LIVE,
      TYPE(x'.j + 1) := ACTIVE;
  : x = x'.j is a MIN node and STATUS (x) =
      SOLVED :
    for each node y ≠ x in OPEN such that y is
      a successor of x' do
        if h(y) ⩽ h(x) then
          begin
            remove y from OPEN ;
            SPACE := SPACE + 1 ;
          end;
    y := inactivesucc (OPEN, x) ;
    if y = null then
       begin
         remove x from OPEN ;
         insert x' OPEN with h(x'):=h(x),
           STATUS (x') := SOLVED,
           TYPE(x') := ACTIVE ;
       end
     else  TYPE(y) := ACTIVE ;
   : x = null :
     FLAG := ACTIVE ;
  until x = s and STATUS (x) = SOLVED ;
  output h(x) ;
end ;
```

FLAG is an indicator that takes one of two values : INACTIVE or ACTIVE. Initially FLAG is INACTIVE, and once it becomes ACTIVE it remains ACTIVE. A LIVE node in OPEN is either INACTIVE or ACTIVE, while a SOLVED node in OPEN is always ACTIVE. An INACTIVE node can be thought of as a node that cannot be expanded because of lack of memory space, and it is our intention to confine selections from OPEN to ACTIVE nodes only. An exception occurs at the beginning of the execution of the algorithm when there are no ACTIVE nodes at all, and we must expand INACTIVE nodes and fill up the available memory. Thereafter only ACTIVE nodes get selected from OPEN. The function first (OPEN, FLAG) returns that node x from OPEN whose current h-value is highest among all nodes (if any) in OPEN with TYPE = FLAG, ties begin resolved in favour of lexicographically smaller nodes as usual. If there is no node in OPEN with TYPE = FLAG then null is returned. The function inactivesucc (OPEN, x) returns an INACTIVE successor z of x' (where x=x'.j) such that z is at the greatest depth among all INACTIVE successors of x' in OPEN; if no such z can be found it returns null.

ITERSSS* begins with FLAG set to INACTIVE. So long as FLAG = INACTIVE only INACTIVE nodes get expanded and ACTIVE nodes, if any, in OPEN are all terminal nodes. Since $M \geq M_o$, the algorithm ensures that at least b terminal nodes are brought to the ACTIVE condition before storage runs out. Once FLAG = ACTIVE, INACTIVE nodes in OPEN do not participate in selection and remain in "suspended animation" until some nodes get purged from OPEN and storage is released. When storage becomes available the TYPE of only one node is changed from INACTIVE to ACTIVE; this ensures that the algorithm never gets "stuck" because of insufficient storage.

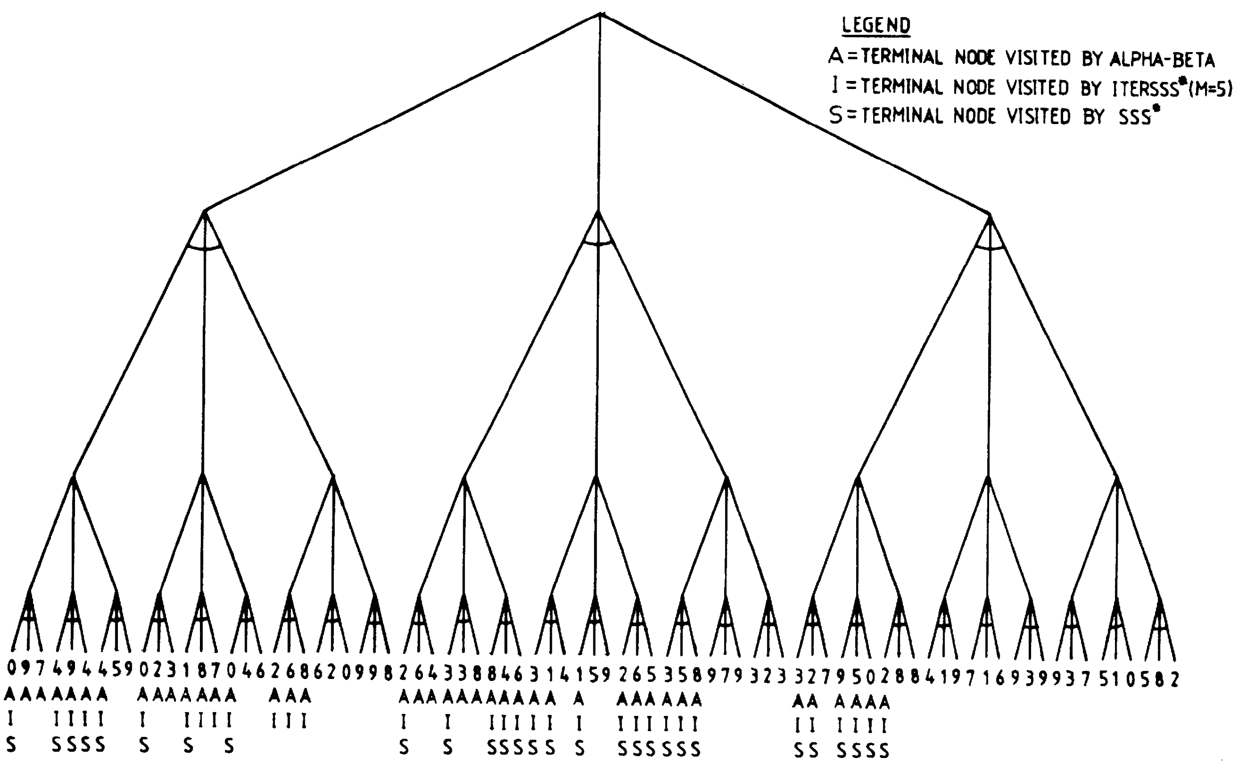

Fig. 1

The *case* statement in ITERSSS* differs from that in SSS* in two ways. The expansion of a LIVE MAX node x can get held up because of lack of storage; if this happens x is made INACTIVE. If a SOLVED MIN node x = x'.j is selected from OPEN, we cannot immediately throw out x from OPEN and assign a SOLVED STATUS to x' since x' can have INACTIVE successors in OPEN; one of these successors of x' in OPEN must then have its TYPE changed to ACTIVE, and x remains in OPEN until x' has no INACTIVE successors in OPEN.

Example 2.2 : For the uniform game tree T of Fig.1, M_o= 5. When M is 5 or 6, ITERSSS* examines 33 terminals, which is much less than that seen by alphabeta. When M $\geqslant$ 7, ITERSSS* examines 28 terminals, exactly as many as are examined by SSS*.

3. Experimental Observations :

We conducted some experiments on a VAX 11/750 to find out the average number of terminal nodes examined by alphabeta, SSS* and ITERSSS*. Four (b, d) pairs were chosen. For each pair, ten sets of terminal node values were obtained with the help of a random number generator. Both alphabeta and SSS* were run ten times, once with each set of terminal values, and the average number of terminals examined was computed, the average being expressed as a percentage of the total number of terminal nodes in the game tree. For each set of terminal values, ITERSSS* was run five times for five different values of M. The average percentage of terminals visited was computed for each value of M. The programs were written in PASCAL.Table 1 gives the results. It can be seen that the average number of terminals examined by ITERSSS* decreases steadily as M increases; for small M, ITERSSS* examines a slightly smaller number of terminals than alphabeta, while for large M it examines just as many terminals as SSS*. More extensive experimental investigations are needed with much larger numbers of values of M and of sets of terminal values, but we do not expect any departures from the trend shown in Table 1.

4. Theoretical Analysis

How can we characterize the terminal nodes of a game tree that get examined by the alphabeta procedure or SSS* or ITERSSS* ? We do a theoretical analysis to obtain the exact pruning conditions. We begin with some definitions.

Definition 4.1 : Let a game tree T be given.

i) A subtree T' of T is called a *solution tree* if
   a) the root s of T is in T' ;
   b) for every nonterminal MAX node x in T', exactly one son of x in T is in T' ;
   c) for every nonterminal MIN node x in T', all sons of x in T are in T'.

ii) The *value* $v_{T'}$ of the solution tree T' is defined as
$v_{T'} = \min\{v(x) \mid x \text{ is a terminal node in } T'\}$

iii) A solution tree T'_o in T is said to be *optimal* if $v_{T'_o} \geqslant v_{T'}$ for every solution tree T' in T.

iv) For any nonterminal node x in T, let t_x be the minimax value of the subtree rooted at x. When x is a terminal node, let $t_x = v(x)$.

v) Let v_T denote the minimax value of the game tree T, i.e.
$v_T = \max\{v_{T'} \mid T' \text{ is a solution tree in } T\}$

TABLE 1 : Number of terminals examined by alphabeta, SSS* and ITERSSS*

Serial No.	b	d	Total number of terminal nodes = b^d	average number of terminals examined (expressed as a percentage of the total number of terminals)			
				alphabeta	SSS*	Available storage M	ITERSSS*
1	2	15	32768	12.47	8.50	9	12.40
						64	10.36
						128	9.49
						192	9.37
						256	8.50
2	3	10	59049	10.44	6.44	11	10.11
						61	8.31
						122	7.75
						183	7.65
						243	6.44
3	5	6	15625	16.51	11.48	13	15.49
						32	13.46
						63	12.68
						95	12.60
						125	11.48
4	9	5	59049	13.67	10.24	25	13.67
						183	12.34
						365	11.87
						548	11.27
						729	10.24

Remark : We note that $v_T = t_s = v_{T'_o}$.

Definition 4.2 : Let a game tree T be given.

i) Let x and y be two nodes in T. We write $x \lessdot y$ if the Dewey code for x is strictly smaller lexicographically then the Dewey code for y.

ii) Let x be any node in T.

Let $L(x) = \{ z \mid z$ is a terminal node in T, $z \lessdot x$, and there is no solution tree in T to which both x and z belong $\}$.

$R(x) = \{ z \mid z$ is a terminal node in T, $x \lessdot z$, and there is no solution tree in T to which both x and z belong $\}$.

iii) Let T' be a solution tree in T, and let node x be in T'. Let left (x, T') =

$$= \begin{cases} \infty & \text{if there is no terminal node } z \in T' \text{ such that } z \lessdot x \\ \min\{ v(z) \mid z \text{ is a terminal node in } T',\ z \lessdot x \} & \text{otherwise} \end{cases}$$

Now define $B(x) = \max_{x \in T'} \{ \text{left}\ (x, T') \}$.

Note that $B(s) = \infty$.

iv) Let T' be a solution tree in T, and let x be a node in T that does not belong to T'.

$$\text{Let Lin}\ (x,T') = \begin{cases} -\infty & \text{if } L(x) \cap T' = \Phi \\ \min\{ v(z) \mid z \text{ is a terminal node and } z \in L(x) \cap T' \} & \text{otherwise} \end{cases}$$

$$\text{Rin}\ (x, T') = \begin{cases} -\infty & \text{if } R(x) \cap T' = \Phi \\ \min\{ v(z) \mid z \text{ is a terminal node and } z \in R(x) \cap T' \} & \text{otherwise} \end{cases}$$

v) We now define

$$A_L(x) = \max_{x \notin T'} \{ \text{Lin}\ (x, T') \},$$

$$A_R(x) = \max_{x \notin T'} \{ \text{Rin}\ (x, T') \},$$

Again note that $A_L(s) = A_R(s) = -\infty$.

With these definitions we are in a position to state some lemmas and theorems. Proofs are omitted. Related analyses can be found in Baudet [1] and Pearl [3].

Lemma 4.1 : Let node x be a successor of node z in a game tree T. Then

$$A_L(x) \geq A_L(z) \text{ and } B(x) \leq B(z).$$

Definition 4.3 : Let the alphabeta procedure be run on a game tree T. Let x be a node in T. We say that the node x is pruned by alphabeta if no call is made to alphabeta with x as an argument, i.e. if none of the terminal nodes in the subtree rooted at x are examined by alphabeta.

Theorem 4.1 : When the alphabeta procedure is run on a game tree T, a node x in T is pruned iff $A_L(x) \geq B(x)$.

This is the standard pruning condition for alphabeta (see [3]). We have just reformulated the basic definitions in terms of solution trees. Lemma 4.1 would be needed in the proof of Theorem 4.1.

Definition 4.4 : Let T be a game tree, and let SSS* (or ITERSSS*) be run on T.

i) Each call to the function "first" is regarded as a distinct instant of execution. By the k^{th} instant we mean the moment of time immediately following the k^{th} time "first" returns a value.

ii) A node x in T is examined if at some instant during the execution of SSS* (or ITERSSS*), x is returned by the function "first" and x is LIVE.

Lemma 4.2 : At each instant during the execution of SSS* on a game tree T, OPEN contains exactly one node from each solution tree in T.

Theorem 4.2 : Algorithm SSS* when run on any game tree T terminates successfully, i.e. it finally selects s from OPEN in the SOLVED state. At termination, $h(s) = v_T$.

Theorem 4.3 : Let T be a game tree. When SSS* is run on T, a node x in T is not examined iff $[A_L(x) \geq B(x)$ or $A_R(x) > \overline{B(x)}]$.

Remark : Lemma 4.2 clearly does not hold for ITERSSS* if we consider only the ACTIVE nodes in OPEN. However, Theorem 4.2 is still true. The following modified form of Theorem 4.3 also holds.

Theorem 4.4 : Let T be a game tree. When ITERSSS* is run on T, a node x in T is not examined if $A_L(x) \geq B(x)$.

If follows that terminal nodes examined by ITERSSS* are also examined by alphabeta.

5. Conclusion :

SSS* examines fewer terminals in a game tree than alphabeta but takes an inordinate amount of storage. An additional overhead is incurred in maintaining OPEN as a priority queue. These limitations of SSS* were noticed by Stockman [4], who suggested the use of a hybrid alphabeta-SSS* procedure when storage was in limited supply. ITERSSS* is not such a hybrid procedure, however; it is not related to alphabeta at all and can be viewed as a modification of SSS*. Its most desirable feature is that the amount of storage M available for OPEN can be supplied to it as a parameter at run time. Experiments indicate that it performs as per expectations. What would be of great interest is an average case analysis of the dependence on M of the number of terminal nodes examined. More extensive experimental studies are also needed to find out whether ITERSSS* outperforms alphabeta in practical situations.

REFERENCES

[1] Baudet, G.M. "On the Branching Factor of the Alpha-Beta Pruning Algorithm." Artificial Intelligence 10(2) (1978) 173-199.

[2] Knuth, D. E. and Moore, R.W. "An Analysis of Alpha-Beta Prunning," Artificial Intelligence 6(4) (1975) 293-326.

[3] Pearl, J. "Heuristics: Intelligent Search Strategies for Computer Problem Solving" Addison-Wesley 1984.

[4] Stockman, G. "A Minimax Algorithm Better than Alpha-Beta?" Artificial Intelligence 12(2) (1979) 179-196.