COMPLEXITY DRIVEN ALGORITHM FOR COMPUTER GAMES

HASAN KRAD
Computer Science Department
Dillard University
2601 Gentilly Blvd.
New Orleans, LA 70122
UNITED STATES OF AMERICA

KOSTAS PETRAKOS
Mathematics and Computer Science Department
Adelphi University
Garden City, N.Y. 11530
UNITED STATES OF AMERICA

Abstract: We present an algorithm for computer games, which generates partial game trees in accordance with the theme, that when comparing the values of several alternative moves, the criterion should depend not only on their apparent scores, but also on the amount of time already expended in their examination, and remaining game time. We show that such algorithm admits an interpretation as a generalized minmax search.

Key Words: Search Algorithms, Minmax Algorithm, Escape Function, Time bounded Algorithm.

1 Introduction

Computer game playing programs typically choose their next move by searching a tree to some arbitrary depth, assigning estimates of the utility values to the nodes at that depth, then minmaxing these values to assign estimated utility values of all ancestor nodes [3], [5], and [6].

What distinguishes these algorithms is their choice of rules or heuristics used in guiding the search together with the heuristics encoded in their evaluation function. For some examples, see [1], [2], [4], and [7].

What is often not addressed by these algorithms is that a good move can be discovered only after a few positions have been evaluated, allowing for a player to use his remaining decision (game) time in evaluating future moves.

Furthermore, we suggest that often, whether the largest score among a set of children thus far evaluated corresponds to a good position or not, is a function of its value, and the amount of time that player has expanded examining those children.

Suppose, for instance, that a player, emulating a node P after having evaluated 4 of $P's$ children, has obtained a best score of 0.6 from Q, the second child examined. Such a score might appear appealing to that player had he examined 9 or 10 of $P's$ children with no improvement over

the score provided by Q. In the latter case that player might be more inclined to abandon his search among the remaining children of P, and instead concentrate his efforts elsewhere

In this paper we introduce a new algorithm whose distinguishing feature is that the search is guided by both the amount of a time a player has spend examining the descendants of a given position and their apparent score, and his remaining decision time.

2 Theory Development

To incorporate the above ideas algorithmically we introduce a number of preliminaries starting with the introduction of an e*scape function* defined as a monotone non decreasing function on the closed unit interval with $f(0)=1$ and $f(1)=-1$.

The use of escape functions as the next example illustrates permits for a natural interpretation of what it means to examine a fraction of a position's children as a function of time and be consistent with the theme. Before proceeding with our example, a number of preliminaries are in order.

Henceforth, when discussing game instances we will assume the apriority existence of a game tree together with assigned values at each node corresponding to utility estimates. The assigned scores will be from the player's perspective whose turn is to move from there. Furthermore, we will speak of a node as have been visited to mean that its assigned value has been accessed.

Definition: Let $q_1,\ldots,q_n$, $s_1,\ldots,s_n$ denote a set of nodes with common parent, and their respectively assigned scores. Furthermore, if q_j is such that $s_j=\max(-s_1,\ldots,s_n)$, then q_j is called the *minmax node*, and s_j is called the *minmax score*.

Definition: The complexity assigned value of a node is the amount of time that elapsed for that node to acquire its present score. For example the complexity assigned to a non expanded visited node is merely that node's lookup time that has been assigned a priori, while the complexity assigned to a node Q with visited children is the sum of the complexities of all its visited children plus the time associated with the expansion of node Q. In light of these definitions, we will refer to time dependent algorithm as *Complexity Driven Algorithm*

Example: Let us assume that a player has just expanded a node Q corresponding to his current position, and this action results in the creation of children $q_1,q_2\ldots,q_5$, Fig. 1.

Let us also suppose that the player has adopted the strategy of visiting one child at a time until he finds one that corresponds to a good position, but is not willing to visit more than $3/4$ of these children, although his remaining decision time permits otherwise. Let us also suppose that the total number of time units in the game is 120.

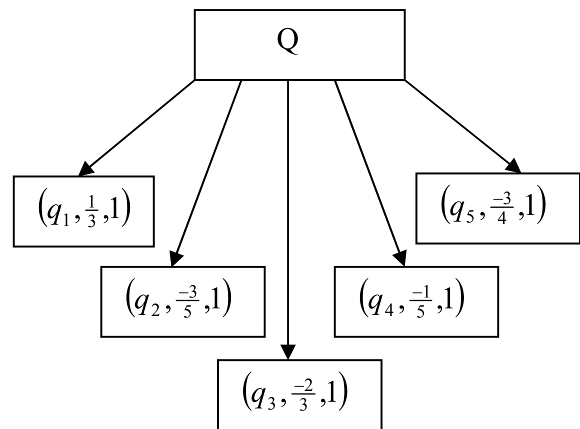

Fig. 1

We can capture this decision process a bit more algorithmically as follows:

Begin visiting $Q's$ children from left to right starting at q_1. After each visit compare the best score (i.e. minmax score) of the visited children against the value of the local escape function $l(x) = \begin{cases} 1 & 0 \le x \le 0.75 \\ -1 & otherwise \end{cases}$ at $\frac{C_k}{C_n}$, where $C_k = n c_\delta + \sum_{i=1}^{k} c_i$, $1 \le k \le n$, $C_n = n c_\delta + \sum_{i=1}^{n} c_i\ c_\delta$. The quantity c_δ shall denote the number of time units utilized in expanding a node, while c_i corresponds to the number of time units associated with evaluating a node q_i, and k the number of visited nodes. Once $m \ge e\left(\frac{C_k}{C_n}\right)$ then the child of Q with assigned score $-m$ is the next candidate for expansion. We shall refer to such a node as an *escape candidate*.

Furthermore, let C_k where $1 \le k \le n$ be the sum of the assigned complexities of the children of Q already visited plus $n\ c_\delta$, and let C_n denote the sum of the assigned complexities of all the children of Q plus $n\ c_\delta$, then the node q_i where $1 \le i \le k$ is said to be a *local escape candidate* if the following is true:

1: It has been visited
2: Its assigned score s_{q_i} is the minmax value of all the visited children, that is $s_{q_i} = \max(-s_{q_1}, \cdots, -s_{q_k})$
3: it escapes (i.e. $s_{q_i} \ge e\left(C_k / C_n\right)$).

If at any point the algorithm's remaining decision time is not sufficient to visit any more nodes, the algorithm will choose, as the next escape candidate, the minmax child of the already visited children of Q regardless of whether condition 3 is satisfied.

Getting back to our example after visiting q_1, the player evaluates $\frac{-1}{3} \ge e\left(\frac{1}{5}\right) = 1$ given that the expression is false and there is ample remaining decision time, he next visits q_2 and evaluates $\max\left(\frac{-1}{3}, \frac{3}{5}\right) \ge e\left(\frac{2}{5}\right)$. Not having satisfied the escape candidate constraint (and having time to visit more nodes), he visits q_3 and evaluates $\max\left(\frac{-1}{3}, \frac{3}{5}, \frac{2}{3}\right) \ge e\left(\frac{3}{5}\right) = 1$. Once again not having settled on an escape candidate he visits q_4 and after evaluating $\max\left(\frac{-1}{3}, \frac{3}{5}, \frac{2}{3}, \frac{1}{5}\right) \ge e\left(\frac{4}{5}\right) = -1$, he selects q_2 as the next escape candidate.

The fact that a node has been elected as an escape candidate does not guarantee that it will be the next one to be expanded.

Before stating what the criterion is, we need to define few terms.

A node is said to be *terminal* or *leaf,* if it has no descendants. A node is said to be *solved,* if it is assigned score is the minmax value of all its descendants, hence any visited leaf node is solved. A node p is said to be a *confirmation node* or to have been confirmed if the following is true:

(a) m_p the minmax value of the scores of p's visited non-solved children is greater than or equal to the second best visited child of the root.
(b) Let $p_1, \ldots, p_k$ denote the visited non-solved siblings of p, and let s_j, $1 \le j \le k$ denote the minmax value of the scores of p_j's

visited non-solved children, then $\max\left(-s_p,-s_1,\ldots,-s_k\right)=m_p$. Note that initial root node is always confirmed.

In order to continue the search along the subtree rooted at the most recent escape candidate q, we require that q be non-solved and its parent p be confirmed. The requirement that q be non-solved stems from the fact that no further information can be obtained by searching along the descendants of a solved node. If condition (a) is not satisfied, it signifies the existence of a node whose score is better than that of the current escape candidate, in which case we require the algorithm to seek a next escape candidate elsewhere. Condition (b) is required to insure that p is being confirmed based on the score of the most recent escape candidate.

Next the score and complexity values of the parent of q_2 are updated as described above, followed by either its confirmation or refutation. Since Q, the parent of q_2, is the initial root node, it is confirmed by default. There are three instances where root can be refuted. Two of those instances include the case where time has run out, or when all of $Q's$ descendants have been visited, in which case the algorithm halts returning the minmax child of Q. The remaining instance occurs when a player after having updated a node, contrasts the minmax score of initial root against his global escape function so as to determine whether he should continue searching along the current subtree or simply make his move. Hence, in our example $\frac{2}{3}$ is contrasted against $g(x)=\begin{cases}1 & 0\le x\le 30\\ -1 & x\ge 30\end{cases}$ at $x=C_k=4$.

Since $\frac{2}{3}<1$, we continue searching along our current subtree.

Next, q_3 is expanded creating the game tree depicted in Fig. 2. Let c_δ and $l(x)$ be as described above, and assume that nodes are visited from left to right.

Although we omit the details, one can see that nodes $q_{31},\cdots,q_{35}$, will be visited before appointing q_{32} as the next escape candidate. The information obtained from visiting children of q_3 is now used to update the complexity and score of q_3, resulting in Fig. 3.

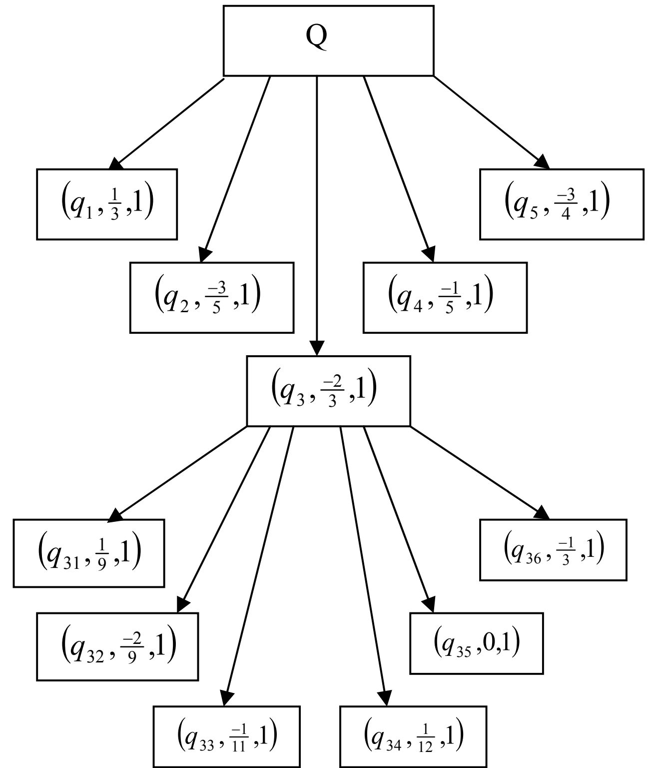

Fig. 2

Once q_3 is updated, it no longer possesses the best score among the visited children of Q, hence refuted.

Once again, since the updated minmax value of the root does not exceed $g(x)$, we continue our search with the current subtree.

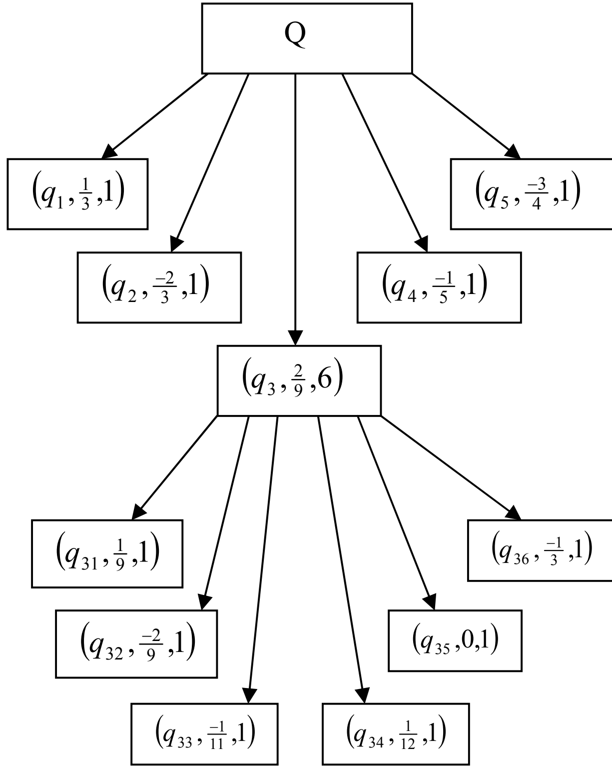

Fig. 3

At this stage, the next escape candidate can be selected among either the unvisited children of Q or the visited siblings of q_3. We have opted for the algorithm to first seek an escape candidate among the visited children before searching among the non-visited children as follows:

Let $\pi_1 = q_1, \ldots, q_k$ denotes a set of visited nodes with common parent and let $\pi_2 = q_{k+1}, \ldots q_{k+j}$ denotes their non-visited siblings. Furthermore let $s_1, \ldots, s_{k+j}$ denotes the assigned scores to nodes $q_1, \ldots, q_{k+j}$, respectively. If at least one of the nodes in π_1 is non-solved, we let s_j equal to the minmax value of the set consisting of all the scores of all visited non-solved nodes in π_1, if $s_j \geq e\left(\frac{C_k}{C_{k+j}}\right)$ then $next_sib \leftarrow q_j$ else $next_sib \leftarrow q_{k+i}$ where $s_{k+i} \geq e\left(\frac{C_{k+i}}{C_{k+j}}\right)$, $1 \leq i \leq j$. We shall refer to the above procedure as $next_sib$.

The $next_sib$ procedure is invoked after a node has been revoked, and the algorithm first seeks a next escape candidate among its siblings. There will however arise instances where the algorithm will seek an escape candidate along a previously expanded and visited node, which in turn has visited children. In such case the algorithm will invoke the routine $trace(p)$ that behaves as follows:

$trace(p)$: Returns a value of 0, if p has yet to be expanded else it seeks a non visited node among $p's$ children, visits it and return that node's id, otherwise it returns the minmax child of p among the $p's$ visited children.

3 Complexity Bounded Algorithm

Next we present our algorithm in pseudo code form followed by a description and function of its main routines:

```
Integer  next_move(integer p)
{
   Global Variable: boolean g_status;
   boolean status ;
   integer q;
   q←trace(p)
   if(q=0)
     {
       expand(p)
       q←local_esc_child(p)
       update(p)
```

```
    update g_status (p)
     status←refuted(p)
    }
  else
     status←true
  while(not status)
    {
     n←next_move(q)
     q←next_sib(q)
     update(p)
     update g_status(p)
     status←refute(p)
    }
  next_move←minmax_node(q)
}
```

$trace(p)$: Returns a value of 0, if p has yet to be expanded else it seeks a non visited node among $p's$ children, visits it and return that node's id, otherwise it returns the minmax child of p among $p's$ visited non solved children.

$expand(p)$: Generates all the children of p if not previously generated, otherwise exit.

$esc_child(p)$: Returns an escape candidate from the children of p.

$update(p)$: Updates $p's$ score, complexity value and solved status as follows:

* p is assigned the updated score of the minmax child of p.
* Complexity becomes the sum of its initially assigned values and the updated complexities of all its visited descendants
* Solved status of p is assigned the conjunction of the solved status of all its children.

$next_sib(q)$: Returns a zero if g_status is true else returns a node from the set consisting of q and its siblings, using the above described selection process.

update g_status(p): Returns a value of true if minmax of current root exceeds $g(x)$ at $\frac{C}{Remaining\ Gametime}$, where C corresponds to roots acquired complexity else returns False.

refuted(p): Rreturns false if p is not confirmed or *g_status(p)* is true , otherwise returns True.

Theorem: Let G be a finite game graph where each node has an initial score and complexity value, let t correspond to the amount of time a player has in deciding his next move on G using the Complexity Control Algorithm. Furthermore, let q denote the next move id returned by the Complexity Control Algorithm s_q its associated score, and let $g(x)=1$. For sufficiently large t (i.e. more than or equal to expand and visit all of G) then $-s_q = M$, where M is the minmax value of G.

Proof: Since the Complexity Control Algorithm updates the score of a node by assigning it the minmax value of its visited descendants. It suffices to show that the Complexity Control Algorithm, for sufficiently large t, will visit all the nodes of G. Given that the algorithm is at some arbitrary non-terminal state, we can show that it will subsequently visit a new node. Without loss of generality, we assume that

$next_move(p)$ has just been invoked with some arbitrary node p. When our algorithm invokes $trace(p)$, it will branch out with one of the following states:

State A: $trace(p)$ returns a value of 0.
State B: $trace(p)$ returns q_1, the id of a previously non-visited node.
State C: $trace(p)$ returns q_2, the id of a previously visited non-solved node.

If state A is realized, the algorithm subsequently expands p and visits at least one of $p's$ children. If the algorithm branches out to state B, then q_1 is visited. If the algorithm branches out to State C, the algorithm will recursively invoke $trace(p)$, and eventually landing on state A or state B. The claim that either State A or State B will eventually be realized is assured by the fact that $trace(p)$ always returns either 0 or the id of a non solved node, guarantying that at least one of $p's$ descendants has yet to be visited. After a previously non-visited node q becomes visited, the algorithm will update its parent p and determine whether p is confirmed or refuted. If confirmed, the algorithm will subsequently invoke $trace(p)$, thus assuring that eventually State A or State B is realized. The only way for the algorithm to fail branching out to either State A or State B is when p and its parents are constantly refuted. This, however, cannot go on indefinitely, for $p = root$ cannot be refuted unless it has been solved (i.e. all of its descendants have been solved in which case the algorithm rightfully terminates). Thus the algorithm will invoke $trace(p)$, where p is a non-solved node, and subsequently either State A or State B will be realized. This completes the proof.

4 Conclusion

We have presented a domain independent game tree search algorithm for choosing next moves in a two person zero sum game. Its distinguishing features over other such algorithms is that it is driven by both the best position examined and the amount of time expanded in attaining such a position, and allows savings of time on a given move to be used for future moves. Furthermore for finite game trees, it allows the interpretation of being a generalized minmax algorithm in the asymptotic sense as a function of time.

References:

[1] Allis, L. Victor, Maatren van der Meulen and H. Jaap van den Herik, "Proof number search", Artificial Intelligence, Vol. 66, 1994, pp.91-124

[2] Berliner Hans, "The B* Tree Search Algorithm: A Best-First Proof Procedure", Artificial Intelligence, Vol.12, 1979, pp.23-40

[3] Kunth, Donald E. and Ronald W. Moore, "An Anallysis of Alpha-Beta Prunning", Artificial Intelligence, Vol. 6, 1975, pp.239-326

[4] McAllester, David Allen, "Conspiracy Numbers for Min-Max Search", Artificial Intelligence, Vol. 35, 1988, pp.287-310.

[5] Nau, Dana S., "Pathology on Game Trees Revisited, an Alternative to Minimaxing", *Artificial Intelligence*, Vol. 21, 1983, pp. 221-244.

[6] Pearl, Judea, "Asymptotic Properties of Minmax Trees and Game-Searching Procedures", *Artificial Intelligence*, Vol. 14, 1980, pp. 113-138.

[7] Schaeffer, Jonathan, "Conspiracy Numbers", Artificial Intelligence, Vol. 43, 1990, pp.67-84.