Partition Search

Matthew L. Ginsberg
CIRL
1269 University of Oregon
Eugene, OR 97403
ginsberg@cirl.uoregon.edu

Abstract

We introduce a new form of game search called *partition search* that incorporates dependency analysis, allowing substantial reductions in the portion of the tree that needs to be expanded. Both theoretical results and experimental data are presented. For the game of bridge, partition search provides approximately as much of an improvement over existing methods as α-β pruning provides over minimax.

Introduction

Computers are effective game players to the extent that brute-force search can overcome innate stupidity; most of their time spent searching is spent examining moves that a human player would discard as obviously without merit.

As an example, suppose that White has a forced win in a particular chess position, perhaps beginning with an attack on Black's queen. A human analyzing the position will see that if Black doesn't respond to the attack, he will lose his queen; the analysis considers places to which the queen could move and appropriate responses to each.

A machine considers responses to the queen moves as well, of course. But it must also analyze in detail every *other* Black move, carefully demonstrating that each of these other moves can be refuted by capturing the Black queen. A six-ply search will have to analyze every one of these moves five further ply, even if the refutations are identical in all cases. Conventional pruning techniques cannot help here; using α-β pruning, for example, the entire "main line" (White's winning choices and all of Black's losing responses) must be analyzed even though there is a great deal of apparent redundancy in this analysis.[1]

In other search problems, techniques based on the ideas of dependency maintenance (Stallman & Sussman 1977) can potentially be used to overcome this sort of difficulty. As an example, consider chronological backtracking applied to a map coloring problem. When a dead end is reached and the search backs up, no information is cached and the effect is to eliminate only the specific dead end that was encountered. Recording information giving the reason for the failure can make the search substantially more efficient.

In attempting to color a map with only three colors, for example, thirty countries may have been colored while the detected contradiction involves only five. By recording the contradiction for those five countries, dead ends that fail for the same reason can be avoided.

Dependency-based methods have been of limited use in practice because of the overhead involved in constructing and using the collection of accumulated reasons. There is substantial promise for overcoming this difficulty in game search, however, since most algorithms already include similar information in the form of a transposition table.

A transposition table stores a single game position and the backed up value that has been associated with it. The name reflects the fact that many games "transpose" in that identical positions can be reached by swapping the order in which moves are made. The transposition table eliminates the need to recompute values for positions that have already been analyzed.

These collected observations lead naturally to the idea that transposition tables should store not single positions and their values, but *sets* of positions and their values. Continuing the dependency-maintenance analogy, a transposition table storing sets of positions can prune the subsequent search far more efficiently than a table that stores only singletons.

There are two reasons that this approach works. The first, which we have already mentioned, is that most game-playing programs already maintain transposition tables, thereby incurring the bulk of the computational expense involved in storing such tables in a more general form. The second and more fundamental reason is that when a game ends with one player the winner, the reason for the victory is generally a local one. A chess game can be thought of as ending when one side has its king captured (a completely local phenomenon); a

[1] An informal solution to this is Adelson-Velskiy et.al.'s *method of analogies* (Adelson-Velskiy, Arlazarov, & Donskoy 1975). This approach appears to have been of little use in practice because it is restricted to a specific class of situations arising in chess games.

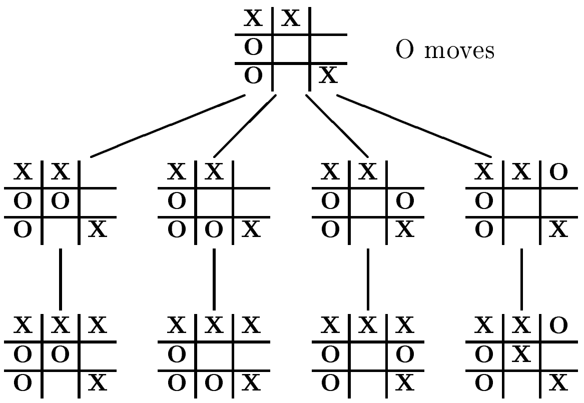


Figure 1: A portion of the game tree for tic-tac-toe

checkers game, when one side runs out of pieces. Even if an internal search node is evaluated before the game ends, the reason for assigning it any specific value is likely to be independent of some global features (e.g., is the Black pawn on $a5$ or $a6$?). Partition search exploits both the existence of transposition tables and the locality of evaluation for realistic games.

This paper explains these ideas via an example and then describes them formally. Experimental results for the game of bridge are also presented.

An example

Our illustrative examples will be taken from the game of tic-tac-toe. A portion of the game tree for this game appears in Figure 1, where we are analyzing a position that is a win for X. We show O's four possible moves, and a winning response for X in each case. Although X frequently wins by making a row across the top of the diagram, α-β pruning cannot reduce the size of this tree because O's losing options must all be analyzed separately.

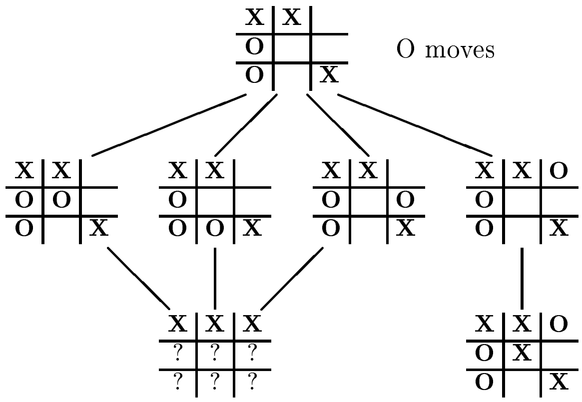
Consider now the position at the lower left in the diagram, where X has won:

$$\begin{array}{c|c|c} \mathbf{X} & \mathbf{X} & \mathbf{X} \\ \hline \mathbf{O} & \mathbf{O} & \\ \hline \mathbf{O} & & \mathbf{X} \end{array} \tag{1}$$

The reason that X has won is local. If we are retaining a list of positions with known outcomes, the entry we can make because of this position is:

$$\begin{array}{c|c|c} \mathbf{X} & \mathbf{X} & \mathbf{X} \\ \hline ? & ? & ? \\ \hline ? & ? & ? \end{array} \tag{2}$$

where the ? means that it is irrelevant whether the associated square is marked with an X, an O, or unmarked. This table entry corresponds not to a single position, but to approximately 3^6 because the unassigned squares can contain X's, O's, or be blank. We can reduce the game tree in Figure 1 to:



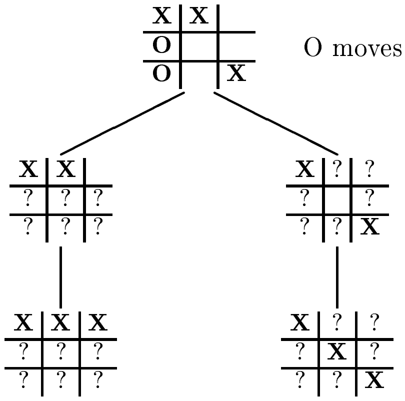
Continuing the analysis, it is clear that the position

$$\begin{array}{c|c|c} \mathbf{X} & \mathbf{X} & \\ \hline ? & ? & ? \\ \hline ? & ? & ? \end{array} \tag{3}$$

is a win for X if X is on play.[2] So is

$$\begin{array}{c|c|c} \mathbf{X} & ? & ? \\ \hline ? & & ? \\ \hline ? & ? & \mathbf{X} \end{array}$$

and the tree can be reduced to:



Finally, consider the position

$$\begin{array}{c|c|c} \mathbf{X} & \mathbf{X} & \\ \hline ? & & \\ \hline ? & ? & \mathbf{X} \end{array} \tag{4}$$

where it is O's turn as opposed to X's. Since every one of O's moves leads to a position that is known to be a win for X, we can conclude that the above position is a win for X as well. The root node in the reduced tree can therefore be replaced with the position of (4).

These positions capture the essence of the algorithm we will propose: If player x can move to a position that is a member of a set known to be a win for x, the given position is a win as well. If every move is to a position that is a loss, the original position is also.

[2]We assume that O has not already won the game here, since X would not be "on play" if the game were over.

Existing methods

In this section, we present a summary of existing methods for evaluating positions in game trees. There is nothing new here; our aim is simply to develop a precise framework in which our new results can be presented.

Definition 1 *A* game *is a quadruple $(G, p_I, s, \mathtt{ev})$, where G is a finite set of legal positions, $p_I \in G$ is the initial position, $s : G \to 2^G$ gives the successors of a given position, and* ev *is an evaluation function*

$$\mathtt{ev} : G \to \{\mathtt{max}, \mathtt{min}\} \cup [0, 1]$$

Informally, $p' \in s(p)$ means that position p' can be reached from p. The structures G, p_I, s and ev *are required to satisfy the following conditions:*

1. *There is no sequence of positions $p_0, \ldots, p_n$ with $n > 0$, $p_i \in s(p_{i-1})$ for each i and $p_n = p_0$. In other words, there are no "loops" that return to an identical position.*
2. $\mathtt{ev}(p) \in [0, 1]$ *if and only if $s(p) = \emptyset$. In other words,* ev *assigns a numerical value to p if and only if the game is over. Informally, $\mathtt{ev}(p) = \mathtt{max}$ means that the maximizer is to play and $\mathtt{ev}(p) = \mathtt{min}$ means that the minimizer is to play.*

We use 2^G to denote the power set of G, the set of subsets of G. There are two further things to note about this definition.

First, the requirement that the game have no "loops" is consistent with all modern games. In chess, for example, positions can repeat but there is a concealed counter that draws the game if either a single position repeats three times or a certain number of moves pass without a capture or a pawn move. In fact, dealing with the hidden counter is more natural in a partition search setting than a conventional one, since the evaluation function is in general (although not always) independent of the value of the counter.

Second, the range of ev includes the entire unit interval $[0, 1]$. The value 0 represents a win for the minimizer, and 1 a win for the maximizer. The intermediate values might correspond to intermediate results (e.g., a draw) or, more importantly, allow us to deal with internal search nodes that are being treated as terminal and assigned approximate values because no time remains for additional search.

The evaluation function ev can be used to assign numerical values to the entire set G of positions:

Definition 2 *Given a game $(G, p_I, s, \mathtt{ev})$, we introduce a function $\mathtt{ev}_c : G \to [0, 1]$ defined recursively by*

$$\mathtt{ev}_c(p) = \begin{cases} \mathtt{ev}(p), & \text{if } \mathtt{ev}(p) \in [0, 1]; \\ \max_{p' \in s(p)} \mathtt{ev}_c(p'), & \text{if } \mathtt{ev}(p) = \mathtt{max}; \\ \min_{p' \in s(p)} \mathtt{ev}_c(p'), & \text{if } \mathtt{ev}(p) = \mathtt{min}. \end{cases}$$

The value *of $(G, p_I, s, \mathtt{ev})$ is defined to be $\mathtt{ev}_c(p_I)$.*

To evaluate a position in a game, we can use the well-known *minimax* procedure:

Algorithm 3 (Minimax) For a game $(G, p_I, s, \mathtt{ev})$ and a position $p \in G$, to compute $\mathtt{minimax}(p)$:

if $\mathtt{ev}(p) \in [0, 1]$ **return** $\mathtt{ev}(p)$
if $\mathtt{ev}(p) = \mathtt{max}$ **return** $\max_{p' \in s(p)} \mathtt{minimax}(p')$
if $\mathtt{ev}(p) = \mathtt{min}$ **return** $\min_{p' \in s(p)} \mathtt{minimax}(p')$

There are two ways in which the above algorithm is typically extended. The first involves the introduction of transposition tables; we will assume that a new entry is added to the transposition table T whenever one is computed. (A modification to cache only selected results is straightforward.) The second involves the introduction of α-β pruning. Incorporating these ideas gives us:

Algorithm 4 (α-β pruning with transposition tables) Given a game $(G, p_I, s, \mathtt{ev})$, a position $p \in G$, cutoffs $[x, y] \subseteq [0, 1]$ and a transposition table T consisting of triples $(p, [a, b], v)$ with $p \in G$ and $a, b, v \in [0, 1]$, to compute $\alpha\beta(p, [x, y])$:

if there is an entry $(p, [x, y], z)$ in T **return** z
if $\mathtt{ev}(p) \in [0, 1]$ **then** $v_{\text{ans}} = \mathtt{ev}(p)$
if $\mathtt{ev}(p) = \mathtt{max}$ **then**
  $v_{\text{ans}} := 0$
  for each $p' \in s(p)$ **do**
    $v_{\text{new}} = \alpha\beta(p', [\max(v_{\text{ans}}, x), y])$
    if $v_{\text{new}} \geq y$ **then**
      $T := T \cup (p, [x, y], v_{\text{new}})$
      return v_{new}
    if $v_{\text{new}} > v_{\text{ans}}$ **then** $v_{\text{ans}} = v_{\text{new}}$
if $\mathtt{ev}(p) = \mathtt{min}$ **then**
  $v_{\text{ans}} := 1$
  for each $p' \in s(p)$ **do**
    $v_{\text{new}} = \alpha\beta(p', [x, \min(v_{\text{ans}}, y)])$
    if $v_{\text{new}} \leq x$ **then**
      $T := T \cup (p, [x, y], v_{\text{new}})$
      return v_{new}
    if $v_{\text{new}} < v_{\text{ans}}$ **then** $v_{\text{ans}} = v_{\text{new}}$
$T := T \cup (p, [x, y], v_{\text{ans}})$
return v_{ans}

Each entry in the transposition table consists of a position p, the current cutoffs $[x, y]$, and the computed value v. We need to include the cutoff information because it is only for these cutoffs that the value returned by Algorithm 4 is only guaranteed to be correct.

The upper cutoff y is the currently smallest value assigned to a minimizing node; the minimizer can do at least this well in that he can force a value of y or lower. Similarly, x is the currently greatest value assigned to a maximizing node.

Proposition 5 *Suppose that $v = \alpha\beta(p, [x, y])$ for each entry $(p, [x, y], v)$ in T. Then if $\mathtt{ev}_c(p) \in [x, y]$, the value returned by Algorithm 4 is $\mathtt{ev}_c(p)$.*

Partition search

We are now in a position to present our new ideas. We begin by formalizing the idea of a position that can reach a known winning position or one that can reach only known losing ones.

Definition 6 *Given a game* $(G, p_I, s, \mathtt{ev})$ *and a set of positions* $S \subseteq G$*, we will say that the* set of positions that can **reach** S *is the set of all* p *for which* $s(p) \cap S \neq \emptyset$*. This set will be denoted* $R_0(S)$*.*

The set of positions **constrained** to reach S *is the set of all* p *for which* $s(p) \subseteq S$*, and is denoted* $C_0(S)$*.*

In practice, of course, it may not be feasible to construct the R_0 and C_0 operators exactly; the data structures being used to describe a set S of situations may not conveniently describe the set of all situations from which S can be reached. Somewhat more specifically, we may be analyzing a particular position p and know that it is a win for the maximizer because the maximizer can move from p to the winning set S; in other words, p is a win because it is in $R_0(S)$. We would like to record at this point the fact that the set $R_0(S)$ is a win for the maximizer, but may not be able to construct or represent this set conveniently. We will therefore assume that we have some computationally effective way to approximate the R_0 and C_0 functions. We will also assume that when our evaluation function returns a specific value, it provides us with information about a set of positions that would evaluate similarly:

Definition 7 *Let* $(G, p_I, S, \mathtt{ev})$ *be a game. Let* f *be any function with range* 2^G*, so that* f *selects a set of positions based on its arguments. We will say that* f respects *the evaluation function* $\mathtt{ev}$ *if whenever* $p, p' \in F$ *for any* F *in the range of* f*,* $\mathtt{ev}(p) = \mathtt{ev}(p')$*.*

A partition system *for the game is a triple* (P, R, C) *of functions that respect* $\mathtt{ev}$ *such that:*

1. $P : G \to 2^G$ *maps positions into sets of positions such that for any position* p*,* $p \in P(p)$*.*
2. $R : G \times 2^G \to 2^G$ *accepts as arguments a position* p *and a set of positions* S*. If* $p \in R_0(S)$*, so that* p *can reach* S*, then* $p \in R(p, S)$ *and* $R(p, S) \subseteq R_0(S)$*.*
3. $C : G \times 2^G \to 2^G$ *accepts as arguments a position* p *and a set of positions* S*. If* $p \in C_0(S)$*, so that* p *is constrained to reach* S*, then* $p \in C(p, S)$ *and* $C(p, S) \subseteq C_0(S)$*.*

From a commonsense point of view, the function P tells us which positions are sufficiently "like" p that they evaluate to the same value. In tic-tac-toe, for example, the position (1) where X has won with a row across the top might be generalized by P to the set of positions

$$\begin{array}{c|c|c} \mathbf{X} & \mathbf{X} & \mathbf{X} \\ \hline ? & ? & ? \\ \hline ? & ? & ? \end{array} \tag{5}$$

as in (2).

The functions R and C approximate R_0 and C_0. Once again turning to our tic-tac-toe example, suppose that we take S to be the set of positions appearing in (5) and that p is given by

$$\begin{array}{c|c|c} \mathbf{X} & \mathbf{X} & \\ \hline \mathbf{O} & \mathbf{O} & \\ \hline \mathbf{O} & & \mathbf{X} \end{array}$$

so that S can be reached from p. $R(p, S)$ might be

$$\begin{array}{c|c|c} \mathbf{X} & \mathbf{X} & \\ \hline ? & ? & ? \\ \hline ? & ? & ? \end{array} \tag{6}$$

as in (3), although we could also take $R(p, S) = \{p\}$ or $R(p, S)$ to be

$$\begin{array}{c|c|c} \mathbf{X} & \mathbf{X} & \\ \hline \mathbf{O} & \mathbf{O} & \\ \hline \mathbf{O} & & \mathbf{X} \end{array} \cup \begin{array}{c|c|c} & \mathbf{X} & \mathbf{X} \\ \hline ? & ? & ? \\ \hline ? & ? & ? \end{array} \cup \begin{array}{c|c|c} \mathbf{X} & & \mathbf{X} \\ \hline ? & ? & ? \\ \hline ? & ? & ? \end{array}$$

although this last union might be awkward to represent. Note that R and C are functions of p as well as S; the set returned must include the given position p but can otherwise be expected to vary as p does.

We will now modify Algorithm 4 so that the transposition table, instead of caching results for single positions, caches results for sets of positions. As discussed in the introduction, this corresponds to the introduction of truth maintenance techniques into adversary search. The modified algorithm appears in Figure 2 and returns a pair of values – the value for the given position, and a set of positions that will take the same value.

Theorem 8 *Suppose that* $v = \alpha\beta(p, [x, y])$ *for every* $(S, [x, y], v)$ *in* T *and* $p \in S$*. Then if* $\mathtt{ev}_c(p) \in [x, y]$*, the value returned by Algorithm 9 is* $\mathtt{ev}_c(p)$*.*

Zero-window search

The effectiveness of partition search depends crucially on the size of the sets maintained in the transposition table. If the sets are large, many positions will be evaluated by lookup. If the sets are small, partition search collapses to conventional α-β pruning.

An examination of Algorithm 9 suggests that the points in the algorithm at which the sets are reduced the most are those marked with a dagger in the description, where an intersection is required because we need to ensure both that the player can make a move equivalent to his best one *and* that there are no other options. The effectiveness of the method would be improved if this possibility were removed,

To see how to do this, suppose for a moment that the evaluation function always returned 0 or 1, as opposed to intermediate values. Now if the maximizer is on play and the value $v_{\text{new}} = 1$, a prune will be generated because there can be no better value found for the maximizer. If all of the v_{new} are 0, then $v_{\text{ans}} = 0$ and we can avoid the troublesome intersection. The

Algorithm 9 (Partition search) Given a game $(G, p_I, s, \texttt{ev})$ and (P, R, C) a partition system for it, a position $p \in G$, cutoffs $[x, y] \subseteq [0, 1]$ and a transposition table T consisting of triples $(S, [a, b], v)$ with $S \subseteq G$ and $a, b, v \in [0, 1]$, to compute $\alpha\beta(p, [x, y])$:

if there is an entry $(S, [x, y], z)$ with $p \in S$ **return** $\langle z, S\rangle$
if $\texttt{ev}(p) \in [0, 1]$ **then** $\langle v_{\text{ans}}, S_{\text{ans}}\rangle = \langle \texttt{ev}(p), P(p)\rangle$
if $\texttt{ev}(p) = \texttt{max}$ **then**
  $v_{\text{ans}} := 0$
  $S_{\text{all}} := \emptyset$
  for each $p' \in s(p)$ **do**
    $\langle v_{\text{new}}, S_{\text{new}}\rangle = \alpha\beta(p', [\max(v_{\text{ans}}, x), y])$
    if $v_{\text{new}} \geq y$ **then**
      $T := T \cup (S_{\text{new}}, [x, y], v_{\text{new}})$
      return $\langle v_{\text{new}}, S_{\text{new}}\rangle$
    if $v_{\text{new}} > v_{\text{ans}}$ **then** $\langle v_{\text{ans}}, S_{\text{ans}}\rangle = \langle v_{\text{new}}, S_{\text{new}}\rangle$
    $S_{\text{all}} := S_{\text{all}} \cup S_{\text{new}}$
  if $v_{\text{ans}} = 0$ **then** $S_{\text{ans}} = C(p, S_{\text{all}})$
  else $S_{\text{ans}} = R(p, S_{\text{ans}}) \cap C(p, S_{\text{all}})$ †
if $\texttt{ev}(p) = \texttt{min}$ **then**
  $v_{\text{ans}} := 1$
  $S_{\text{all}} := \emptyset$
  for each $p' \in s(p)$ **do**
    $\langle v_{\text{new}}, S_{\text{new}}\rangle = \alpha\beta(p', [x, \min(v_{\text{ans}}, y)])$
    if $v_{\text{new}} \leq x$ **then**
      $T := T \cup (S_{\text{new}}, [x, y], v_{\text{new}})$
      return $\langle v_{\text{new}}, S_{\text{new}}\rangle$
    if $v_{\text{new}} < v_{\text{ans}}$ **then** $\langle v_{\text{ans}}, S_{\text{ans}}\rangle = \langle v_{\text{new}}, S_{\text{new}}\rangle$
    $S_{\text{all}} := S_{\text{all}} \cup S_{\text{new}}$
  if $v_{\text{ans}} = 1$ **then** $S_{\text{ans}} = C(p, S_{\text{all}})$
  else $S_{\text{ans}} = R(p, S_{\text{ans}}) \cap C(p, S_{\text{all}})$ †
$T := T \cup (S_{\text{ans}}, [x, y], v_{\text{ans}})$
return $\langle v_{\text{ans}}, S_{\text{ans}}\rangle$

Figure 2: The partition search algorithm

maximizer loses and there is no "best" move that we have to worry about making.

In reality, the restriction to values of 0 or 1 is unrealistic. Some games, such as bridge, allow more than two outcomes, while others cannot be analyzed to termination and need to rely on evaluation functions that return approximate values for internal nodes. We can deal with these situations using a technique known as *zero-window search* (originally called *scout* search (Pearl 1980)). To evaluate a specific position, one first estimates the value to be e and then determines whether the actual value is above or below e by treating any value $v > e$ as a win for the maximizer and any value $v \leq e$ as a win for the minimizer. The results of this calculation can then be used to refine the guess, and the process is repeated. If no initial estimate is available, a binary search can be used to find the value to within any desired tolerance.

Zero-window search is effective because little time is wasted on iterations where the estimate is wildly inaccurate; there will typically be many lines showing that a new estimate is needed. Most of the time is spent on the last iteration or two, developing tight bounds on the position being considered. There is an analog in conventional α-β pruning, where the bounds typically get tight quickly and the bulk of the analysis deals with a situation where the value of the original position is known to lie in a fairly narrow range.

In zero-window search, a node always evaluates to 0 or 1, since either $v > e$ or $v \leq e$. This allows a straightforward modification to Algorithm 9 that avoids the troublesome cases mentioned earlier.

Experimental results for bridge

Partition search was tested by analyzing 1000 randomly generated bridge hands and comparing the number of nodes expanded using partition search and conventional methods.

Bridge was used as the test domain because it is a game for which partition search can be expected to be useful. In order for this to happen, two criteria must be met: First, the functions R_0 and C_0 must support a partition-like analysis: It must be the case that an analysis of one situation will apply equally well to a variety of similar ones. Second, it must be possible to build approximating functions R and C that are reasonably accurate representatives of R_0 and C_0.

Bridge satisfies both of these properties. Expert discussion of a particular hand often will refer to small cards as x's, indicating that it is indeed the case that the exact ranks of these cards are irrelevant. Second, it is possible to "back up" x's from one position to its predecessors. If, for example, one player plays a club with no chance of having it impact the rest of the game, and by doing so reaches a position in which subsequent analysis shows him to have two small clubs, then he clearly must have had *three* small clubs originally. Finally, the fact that cards are simply being replaced by x's means that it is possible to construct data structures for which the time per node expanded is virtually unchanged from that using conventional methods.

Although the details of the approximation functions and data structures are dependent on the rules of bridge and a discussion of their implementation is outside the scope of this paper, let me include at least a single example. Consider the following partial bridge hand in which there are no trumps:

	♠ — ♡ — ♢ AK ♣ —	
♠ 10 ♡ A ♢ — ♣ —		♠ AQ ♡ — ♢ — ♣ —
	♠ KJ ♡ — ♢ — ♣ —	

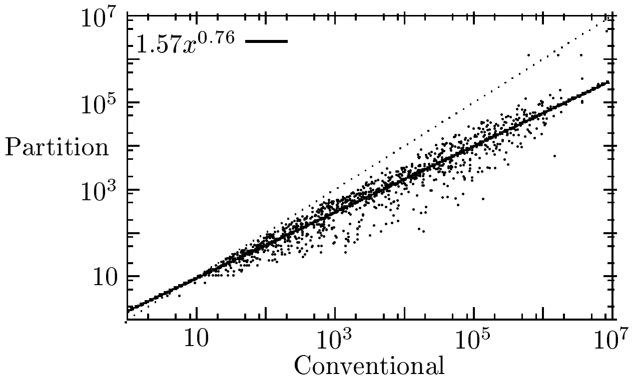


Figure 3: Nodes expanded as a function of method

An analysis of this situation shows that in the main line, the only cards that win tricks by virtue of their ranks are the spade Ace, King and Queen. This sanctions the replacement of the above figure by the following more general one:

♠ —
♡ —
♢ xx
♣ —

♠ x
♡ x
♢ —
♣ —

♠ AQ
♡ —
♢ —
♣ —

♠ Kx
♡ —
♢ —
♣ —

Because of the suitability of the domain, the results we are about to describe are stronger than the results that would be obtained by applying partition search to other games. The results for bridge are striking, however, leading to performance improvements of an order of magnitude or more on fairly small search spaces (perhaps 10^6 nodes). The hands we tested involved between 12 and 48 cards and were analyzed to termination, so that the depth of the search varied from 12 to 48. The branching factor for minimax without transposition tables appeared to be approximately 4. The results appear in Figure 3.

Each point in the graph corresponds to a single hand. The position of the point on the x-axis indicates the number of nodes expanded using α-β pruning and transposition tables, and the position on the y-axis the number expanded using partition search. Both axes are plotted logarithmically.

In both the partition and conventional cases, a binary zero-window search was used to determine the exact value to be assigned to the hand, which the rules of bridge constrain to range from 0 to $\frac{1}{4}$ times the number of cards in play. Hands generated using a full deck of 52 cards were not considered because the conventional method was in general incapable of solving them. The program expands approximately 15K nodes/second on a Sparc 5 or PowerMac 6100. The transposition table uses approximately 6 bytes/node.

The dotted line in the figure is $y = x$ and corresponds to the break-even point relative to the best current methods. The solid line is the least-squares best fit to the logarithmic data, and is given by $y = 1.57x^{0.76}$. This suggests that partition search is leading to an effective reduction in branching factor of $b \rightarrow b^{0.76}$. This improvement, above and beyond that provided by α-β pruning, can be contrasted with α-β pruning itself, which gives a reduction when compared to pure minimax of $b \rightarrow b^{0.75}$ if the moves are ordered randomly (Pearl 1982) and $b \rightarrow b^{0.5}$ if the ordering is optimal.

Conclusion

Partition search brings dependency maintenance techniques to bear on problems in adversary search. The principal difficulty that has arisen in the application of dependency techniques generally is that there is no convenient way to store the conclusions drawn as the search proceeds; this is frequently not an issue in adversary search because transposition tables are constructed and maintained in any event.

Partition search does require that one find effective computational representations for the set of game positions that can reach a particular position p, or the set of positions from which one player is constrained to reach a fixed set of positions S. If these computational representations can be found, however, the method leads to substantial reductions in the number of nodes expanded when evaluating game trees.

Acknowledgement

This work has been supported by AFOSR under contract 92-0693, by ARPA/Rome Labs under contracts F30602-91-C-0036 and F30602-93-C-00031, and by the NSF under grant number STI-9413532. I would like to thank Murray Campbell, Jimi Crawford, Ari Jónsson, Rich Korf, David McAllester, Bart Massey, Bart Selman, and the members of CIRL for discussing these ideas with me.

References

Adelson-Velskiy, G.; Arlazarov, V.; and Donskoy, M. 1975. Some methods of controlling the tree search in chess programs. *Artificial Intelligence* 6:361–371.

Pearl, J. 1980. Asymptotic properties of minimax trees and game-searching procedures. *Artificial Intelligence* 14(2):113–138.

Pearl, J. 1982. A solution for the branching factor of the alpha-beta pruning algorithm and its optimality. *Comm. ACM* 25(8):559–564.

Stallman, R. M., and Sussman, G. J. 1977. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence* 9:135–196.