ELSEVIER

Theoretical Computer Science 252 (2001) 121–149

Theoretical Computer Science

www.elsevier.com/locate/tcs

A solution to the GHI problem for best-first search

Dennis M. Breuker[a], H. Jaap van den Herik[a], Jos W.H.M. Uiterwijk[a,*], L. Victor Allis[b]

[a] *Department of Computer Science, Universiteit Maastricht, P.O. Box 616, 6200 MD Maastricht, Netherlands*

[b] *Quintiq B.V., Van Grobbendoncklaan 83, 5213 AV 's Hertogenbosch, Netherlands*

Abstract

Best-first search algorithms usually amalgamate identical nodes for optimization reasons, meanwhile transforming the search tree into a search graph. However, identical nodes may represent different search states, e.g., due to a difference of history. So, in a search graph a node's value may be dependent on the path leading to it. This implies that different paths may result in different values. Therefore, it is difficult to determine the value of any node unambiguously. The problem is known as the *graph–history-interaction* (GHI) problem. This paper provides a solution for best-first search. First, we give a precise formulation of the problem. Then, for best-first search and for other searches, we review earlier proposals to overcome the problem. Next, our solution is given in detail. Here we introduce the notion of *twin nodes*, enabling a distinction of nodes according to their history. The implementation, called *base-twin algorithm* (BTA), is performed for pn search, a best-first search algorithm. It is generally applicable to other best-first search algorithms. Experimental results in the field of computer chess confirm the claim that the GHI problem has been solved for best-first search.

Keywords: Graph–history interaction (GHI) problem; Best-first search; Base-twin algorithm (BTA)

1. The GHI problem

Search algorithms are used in many domains, ranging from theorem proving to computer games. The algorithms are searching in a state space containing problem

* Corresponding author.

E-mail addresses: breuker@cs.unimaas.nl (D.M. Breuker), herik@cs.unimaas.nl (H.J. van den Herik), uiterwijk@cs.unimaas.nl (J.W.H.M. Uiterwijk), victor@quintiq.nl (L.V. Allis)

0304-3975/01/$ - see front matter
PII: S0304-3975(00)00079-7

states (positions), often represented as nodes. A move which transforms a position into a new position is represented as an edge connecting the two nodes.

In a search tree, it may happen that identical nodes are encountered at different places. If these so-called *transpositions* are not recognized, the search algorithm unnecessarily expands identical subtrees. Therefore, it is profitable to recognize transpositions and to ensure that for each set of identical nodes, only one subtree is expanded.

In computer-chess programs using a *depth-first* search algorithm, this idea is realized by storing the result of a node's investigation in a transposition table (e.g., [9, 12]). If an identical node is encountered in the search process, the result is retrieved from the transposition table and used without further investigation.

If a (selective) *best-first* search algorithm (which stores the whole search tree in memory) is used, the search tree is converted into a search graph, by joining identical nodes into one node, thereby merging the subtrees.

In some domains, these common ways of dealing with transpositions contain an important flaw, since determining whether *nodes* are identical is not the same as determining whether the *search states* represented by the nodes are identical. For two reasons, the path leading to a node cannot be ignored. First, the history of a node may partly determine the *legitimacy of a move*. For instance, in chess, castling rights are not only determined by the position of the pieces on the board, but also by the knowledge that in the position under investigation the King and Rook have not moved previously. Second, the history of a node may play a role in determining the *value of a node*. For instance, a position may be declared a draw by its three-fold repetition or by the so-called k-move rule [10].

We refer to the first problem as the *move-generation problem*, and to the second problem as the *evaluation problem*. The combination of these two problems is referred to as the *graph–history-interaction* (GHI) problem (cf. [8, 13]).

The GHI problem is a noteworthy problem not only in chess but in the field of game playing in general. Its applicability extends though to all domains where the history of states is important. To mention just one example: in job-shop scheduling problems the costs of a task may be dependent on the tasks performed so far, e.g., the cost of preparing a machine for performing some process depends on the state in which the machine has been left after the previous process. Different sequences may lead to different costs.

A possible solution to the GHI problem is to include in all nodes the status of the relevant properties of the history of the node, i.e., the properties which may influence either the move generation or the evaluation of the node. For instance, in job-shop scheduling problems it is common usage to include the state of the machine as part of the search state. A disadvantage of such a solution is that sometimes too many properties may be relevant, resulting in the need of storing large amounts of extra information in each node. For chess, we can distinguish four relevant properties of the history of a position (the first two being relevant for the move-generation problem, and the last two for the evaluation problem):

(1) the castling rights (Kingside and Queenside for both players),
(2) the *en-passant* capturing rights,

(3) the number of moves played without a capture or a pawn move, and
(4) the set of all positions played on the path leading to this node.

The first two properties can be included in each node, without much overhead. The third property can be included in each node, but will reduce the frequency of transpositions drastically. The inclusion of the fourth property, necessary to determine whether a draw by three-fold repetition has been encountered, would require too much overhead. As a result, in most chess programs, the first two properties are included in a node, while the last two are not.

Depending on which properties are included in a node, the probability of two nodes being identical will be reduced. If not all relevant properties are included and transpositions are used, it is possible that incorrect conclusions are drawn from the transpositions. Campbell mentioned that, contrary to best-first search (which he calls selective search), in depth-first search the GHI problem occurs relatively infrequently [8].

In this paper we give a solution to the GHI problem for best-first search with only a few relevant properties included in a node. In Section 2 an example of the GHI problem is given. Previous work on the GHI problem is discussed in Section 3. In Section 4 the general solution to the GHI problem for best-first search is described. A formalized description and the pseudo-code for the implementation in proof-number (pn) search is given in Section 5. Section 6 lists experiments with the new algorithm. It is compared to three other pn-search variants. The results are presented in Section 7. Finally, Section 8 provides conclusions.

2. An example of the GHI problem

Fig. 1 shows a pawn endgame position, taken from [8], where the GHI problem can occur. White (to move) has achieved a winning position. However, we show that it is possible to evaluate this position incorrectly as a draw. In this paper we assume that a single repetition of positions evaluates as a draw, in contrast with the FIDE ruling which stipulates that the same position must occur three times.

In Fig. 2 the relevant part of the search tree is pictured. In this article we follow the notation of [2], i.e., for all AND/OR trees (or graphs) white squares represent OR nodes (positions with the first player to move), and black circles represent AND nodes (positions with the second player to move).

After the move sequence **1**. ♔**b5** ♔**e6** **2**. ♔**a6** ♔**d5** **3**. ♔**b5** ♔**e6** the position after move 1 is repeated (node E), and evaluated as a draw. Since White does not have any better alternative on the third move, the position after **2**. ♔**a6** (node H) should be evaluated as a draw. Backing up this draw leads to the incorrect conclusion that node A evaluates as a draw. However, after the winning move sequence **1**. ♔**a5** ♔**e6** **2**. ♔**a6** the same position (node H) is reached, which is (now) evaluated as a win after **2**. … ♔**d5** **3**. ♔**b5** ♔**e6** **4**. ♔**c6** (node G). Backing up this win leads to the correct conclusion that node A evaluates as a win.

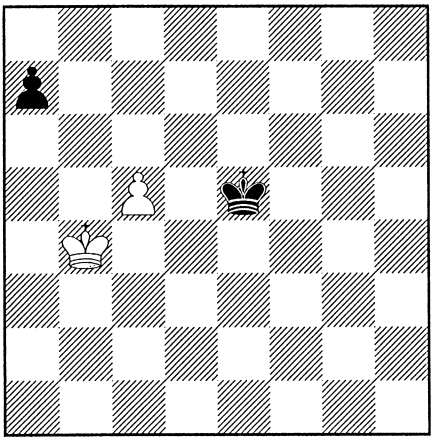

Fig. 1. A pawn endgame (WTM).

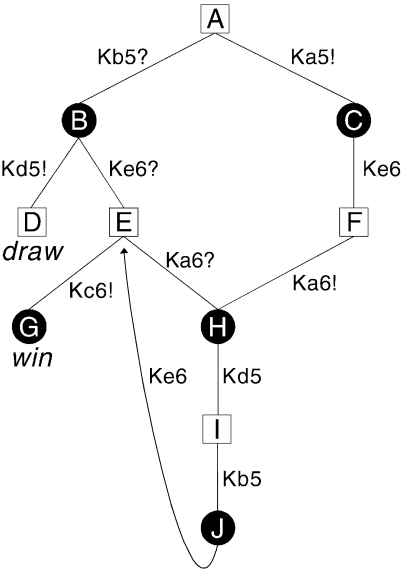

Fig. 2. The GHI problem in the pawn endgame.

An example of the general case is given in Fig. 3. It shows an AND/OR search tree with identical positions.[1] The values of the leaves (given in italics) are seen from the OR player's point of view. The values given next to the nodes are back-up values. We note that the GHI problem can occur in any type of AND/ OR tree. However, to keep

[1] In games such as chess, a repetition of positions is impossible after only two ply (node C in the left subtree of node B and node F in the subtree of node D). Our example disregards this characteristic for simplicity's sake.

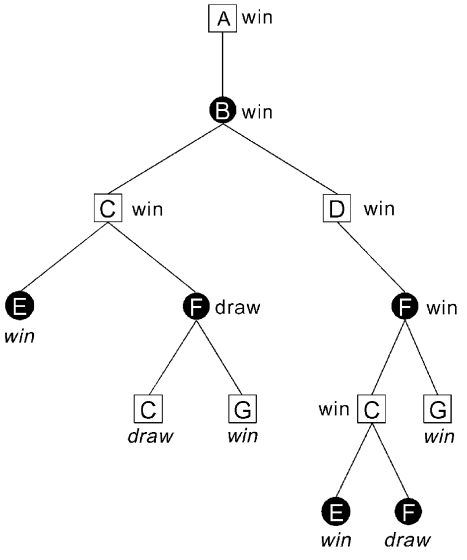

Fig. 3. A search tree with repetitions.

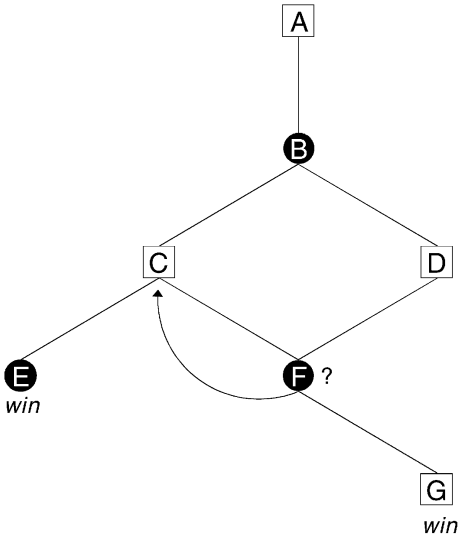

Fig. 4. The DCG corresponding with the tree of Fig. 3.

the example as clear as possible we have chosen to show the example for a minimax game tree.

The terminal nodes E and G are a win for the OR player, and the terminal nodes C and F are evaluated as a draw because of repetition of positions. Propagating the evaluation values of the terminal nodes through the search tree results in a win

at the root. When making use of transpositions, every node should occur only once in the tree. Assume that a parent generates its children and that one of its children already exists in the tree. Then a connecting edge from the parent to the existing node is made. This transforms the search tree into a directed cyclic graph (DCG) (Fig. 4).

In this DCG it is difficult to determine unambiguously the value of node F due to the GHI problem. The value of this node is dependent on the path leading to it. Following the path A–B–C–F, child C of node F is a repetition and hence F is evaluated as a draw, but following the path A–B–D–F, child C is not a repetition and is not evaluated as a draw. Thus, in the DCG, node F has two different values. Hence, in this example it is not possible to determine the value of root A, since in the first case it is a draw, and in the second case it is a win, due to the values of E and G.

3. A review of previous work

Although several authors have mentioned the GHI problem, so far no solution to this problem has been described. Only provisional ideas have been given. Below, we review the five most important ideas.[2]

Palay first identified the GHI problem [13]. He suggested two "solutions": (1) refrain from using graphs, and (2) recognize when the GHI problem occurs and handle accordingly. The first "solution" (apart from not being a real solution, it merely ignores the problem) had as a drawback that large portions of the graph would be duplicated every time a duplicate node occurred, wasting a large amount of time and memory. The second solution worked as follows. When the positions suffering from the GHI problem were recognized, the path from the repetition node upwards to the ancestor with multiple parents was split into separate paths. He did not implement this strategy, since he conjectured that such positions only occurred occasionally (the GHI problem occurred in three out of 300 test positions). A disadvantage of this solution is that the recognition of positions suffering from the GHI problem is not straightforward.

Another idea for a solution originates from Thompson [8]. While building a tactical analyzer, Thompson used a directed cyclic graph (DCG) representation. He saw it suffering from the GHI problem [19]. He cured the problem by taking into account the history of the node to be expanded. The value of this node was then, if necessary, corrected for its history. The newly-generated children were evaluated by doing $\alpha\beta$ searches, yet neglecting their history. As a consequence, the only history errors could occur at the leaves. These errors were corrected as soon as such a leaf was expanded, but it could happen that the expansion of a node was suppressed due to the error.

Campbell discussed the GHI problem thoroughly, applying it to depth-first search only [8]. The key in avoiding most occurrences of the GHI problem appears to be

[2] Berliner and McConnell suggested the use of conditional values as an idea to solve the GHI problem [4]. They promised details in a forthcoming paper.

iterative deepening. Some problems (called "draw-first") can be overcome.[3] However, other problems, which he called "draw-last" could not be solved by his approach.[4] Finally, he remarked that "the GHI problems occur much more frequently in selective search programs, and require some solution in order to achieve reasonably general performance. Both Palay's and Thompson's approaches seem to be acceptable." We conclude that Campbell gave a partial solution for depth-first search, and no solution for best-first search.

Baum and Smith stumbled on the GHI problem, when implementing their best-first search algorithm best play for imperfect players (BPIP) [3]. Baum and Smith completely store the DCG in memory and grow it by using "gulps". In each gulp a fraction of the most interesting leaves is expanded. For each parent-child edge e a subset $S(e)$ was defined as the intersection of *all* ancestor nodes and *all* descendant nodes of edge e. A DCG was claimed to be legitimate (i.e., no nodes have to be split) if and only if, for all children C with more than one parent P, $S(e_{PC})$ is independent of P. Their solution was as follows. Each time a new leaf was created three possibilities were distinguished: (1) if the leaf was a repetition it was evaluated as a draw, else (2) if a duplicate node existed in the graph, these two nodes were merged on the condition that the resultant DCG was legitimate, else (3) the node was evaluated normally. After leaf expansion it was exhaustively investigated whether every node C with multiple parents passed the $S(e)$ test. If not, such a node C was split into several nodes C', C'',..., with distinct subsets $S(e_{PC})$. Then, the subtrees of the newly-created nodes had to be rebuilt and re-evaluated. Baum and Smith gave this idea as a solution to the GHI problem without the support of an implementation. Moreover, they remarked that "Implementation in a low storage algorithm would probably be too costly". We believe that the overhead introduced by our idea, described in the next section, is much less than the overhead introduced by Baum and Smith's idea.

Schijf et al. investigated the problem [17] in the context of proof-number search (pn search) [1]. They examined the problem in directed acyclic graphs (DAGs) and DCGs separately. They noted that, when the pn-search algorithm for trees is used in DAGs, the proof and disproof numbers are not necessarily correctly computed, and the most-proving node is not always found. Schijf proved that the most-proving node always exists in a DAG [16]. Furthermore, he formulated an algorithm for DAGs that correctly determines the most-proving node. However, this algorithm is only of theoretical importance, since it has an unfavourable time-and-memory complexity. Therefore, a practical algorithm was developed. Surprisingly, only two minor modifications to the pn-search algorithm for trees are needed for a practical algorithm for DAGs. The first modification is that instead of updating only *one* parent, *all* parents of a node have to

[3] In the draw-first case node F in Fig. 4 is first reached through path A–B–C–F (and the value of node F is based on child C being a repetition) and later in the search node F is reached through path A–B–D–F and the previous value of node F is used.

[4] In the draw-last case node F in Fig. 4 is first reached through path A–B–D–F (and the value of node F is based on child C being *no* repetition) and later in the search node F is reached through path A–B–C–F and the previous value of node F is used.

be updated. The second modification is that when a child is generated, it has to be checked whether this node is a transposition (i.e., if it was generated earlier). If this is the case, the parent has to be connected to this node that has already been generated. Schijf et al. note that this algorithm contains two flaws [17]. First, the proof and disproof numbers do not represent the cardinality in the smallest proof and disproof set, but these numbers are upper bounds to the real proof and disproof numbers. Second, the node selected by the function SelectMostProvingNode is not always equal to a most-proving node. However, it still holds that if the node chosen is proved, the proof number of the root decreases, whereas if this node is disproved, the disproof number of the root decreases. In either case the proof or disproof number may decrease by more than unity, as a result of the transpositions present. This algorithm has been tested on tic-tac-toe [16]. For the problem of applying pn search to a DCG Schijf et al. give a time-and-memory-efficient algorithm, which, however, sometimes inaccurately evaluates nodes as a draw by repetition [17]. They remark that, as a consequence, their algorithm is sometimes unable to find the goal, even though it should have found it.

4. BTA: an enhanced DCG algorithm

In this section we describe a new and correct algorithm (denoted BTA: base-twin algorithm) for solving the GHI problem for best-first search. The BTA algorithm is based on the distinction of two types of node, termed *base nodes* and *twin nodes*. The purpose of these types is to distinguish between equal positions with different history. Although it was known in the DCG algorithm described by Schijf et al. [17] that nodes sometimes *may* be incorrectly evaluated as a draw, their algorithm was unable to note *when* this occurs. We have devised an alternative in which a sufficient set of relevant properties for correct evaluation is recorded. We have chosen to include in a node only a small number of relevant properties. The reasons for not including *all* relevant properties are:

- some properties are only relevant for a *small* number of nodes,
- the more properties are included, the lower the frequency of transpositions, and
- some properties require too much overhead and/or take up too much space when included in a node.

The move-generation problem (cf. Section 1) can easily be solved by including the relevant properties (in chess these are the castling rights and the *en-passant*-capturing rights) into each node. Hence, only the evaluation problem (cf. Section 1) needs to be solved. We have chosen to describe the solution of repetition of positions, since repetition of positions occurs in many search problems, and the k-move rule is a special rule which seldomly shows up in practice. As mentioned before, we assume that a single repetition of positions results in a draw.

We further distinguish between terminal nodes and leaves. A *terminal node* represents a terminal position, i.e., a position where the rules of the game determine whether

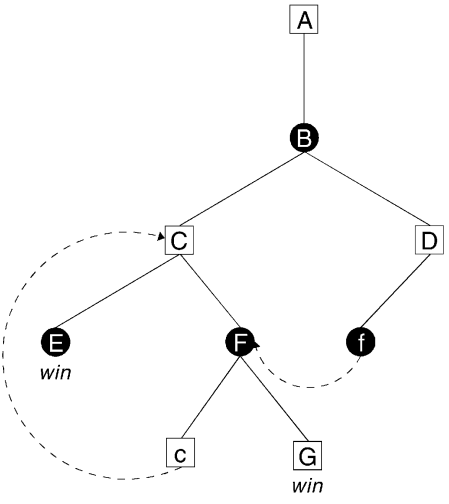

Fig. 5. Our DCG with base nodes and twin nodes corresponding with the DCG of Fig. 4.

the result is a win, a draw, or a loss. *Leaves* are nodes which do not have children (yet). Leaves include terminal nodes and nodes which are not yet expanded.

4.1. Our representation of a DCG

Basically the GHI problem occurs because the search tree is transformed into a DCG by merging nodes representing the same position, but having a different history. To avoid such an undesired coalescence, we propose an enhanced representation of a DCG. In the graph we distinguish two types of node: *base* nodes and *twin* nodes. After a node is generated, it is looked up in the graph by using a pointer-based table. If it does not exist, it is marked as a *base node*. If it exists, it is marked as a *twin node*, and a pointer to its base node is created. Thus, any twin node points to its base node, but a base node does not point to any of its twin nodes. Only base nodes can be expanded. The difference with the "standard implementation" of a DCG is that if two or more nodes are representing the same position (ignoring history) they are not merged into one node. However, their subtree is generated only once. In general, a twin node may have a value different from its base node, although they represent the same position.

Fig. 5 exhibits our implementation of the DCG given in Fig. 4 (assuming that the position corresponding with node F is first generated as child of node C and only later as child of node D). Nodes in upper-case are base nodes, nodes in lower-case are twin nodes. The dashed arrows are pointers from twin nodes to base nodes. The problem mentioned in Fig. 4 can now be handled by assigning separate values to nodes F and f, and to C and c, depending on the paths leading to the corresponding positions.

4.2. The BTA algorithm as solution

As stated before, encountering a repetition of positions in node p does not mean that the repetition signals a *real* draw (defined as the inevitability of a repetition of positions under optimal play). To handle the distinction, we introduce the following two concepts.

Definition 1 (*Possible-draw*). A node p is characterized as a *possible-draw* if the node is a repetition of a node P in the search path.

When during a search iteration a node p is recognized as a *possible-draw*, an accordant marking is stored in node p. Whether a *possible-draw* also is a real draw depends on the history.

Definition 2 (*Possible-draw depth*). The *possible-draw depth* of a node p marked as a possible draw is the depth of node P in the search path.

The possible-draw depth is also stored in node p.

The BTA algorithm for best-first search consists of three phases. Phase 1 deals with the selection of a node. Phase 2 evaluates the selected node. Phase 3 backs up the new information through the search path. The three phases are repeatedly executed until the search process is terminated. At the termination of an iteration all possible-draw markings are removed.

4.2.1. Phase 1: select the best node

In phase 1 a node is selected for evaluation.[5] First, the root is selected (for further selection, see below). Then, for each selected node, two cases exist:

(1) if a child of the selected node is marked as a *possible-draw*, and the remaining children are either real draws, or marked as *possible-draw*s, then the selected node is marked as a *possible-draw* and the corresponding possible-draw depth is set to the minimum of the possible-draw depths of the children. Subsequently, all possible-draw markings from the children are removed and the parent of the selected node is re-selected for investigation;
(2) otherwise, the best child is selected for investigation, ignoring the children which are either real draws, or marked as *possible-draw*s.

Lemma 3. *Assume that a node at depth d in the search path is marked as a possible-draw and the corresponding possible-draw depth is equal to d. Then the node is a real draw by repetition, independent of the history of the node.*

Proof. If a node P is marked as a *possible-draw*, its possible-draw depth is defined as the minimum of the possible-draw depths of all its children. Since this is applied

[5] We assume that the selection of a node proceeds in a top-down fashion.

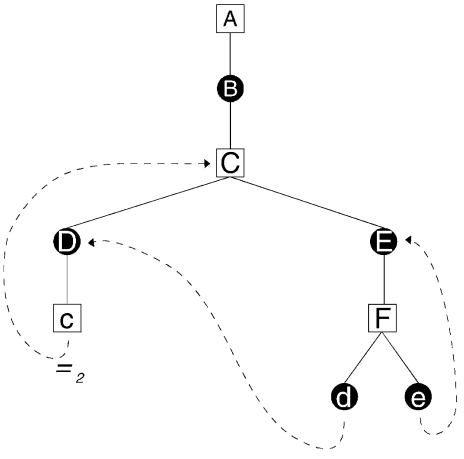

Fig. 6. Encountering the first repetition *c*.

recursively, the possible-draw depth of a node is determined to be the minimum depth of all nodes being repetitions of some node (not necessarily of P), which are reachable from P. If this depth equals the depth of P it means that no ancestors of P are reachable from P. This implies that the possible-draw marking of P is based solely on repetitions of positions *in the subtree of* P and on real draws. P therefore is a real draw. □

The selection of a node is repeated until (1) a real draw by repetition has been encountered, or (2) (a twin node of) a base node with known game-theoretic value has been found,[6] or (3) a leaf has been found.

The selection of a node in the BTA algorithm is illustrated below. In Fig. 6 part of a search graph is depicted. The selection starts at the root (node A). Assume the traversal is in a left-to-right order. Then, at a certain point, node c is selected, and marked as a *possible-draw* because it is a repetition of node C at depth two in the search path (see Fig. 6; the equal sign represents the possible-draw marking and the subscript two represents the possible-draw depth).

After marking node c as a *possible-draw*, the parent of this node (node D) is re-selected and marked as a *possible-draw*, with the same possible-draw depth as node c. Further, the possible-draw marking of node c is removed. After marking node D as a *possible-draw*, its parent C is re-selected. The next best child (not marked as a *possible-draw*) E is selected. Continuing this procedure, at a certain point child d of node F is selected. The child c of twin node d is found by directing the search to the base node D of node d. Node c is (again) marked as a *possible-draw* because it is a repetition of node C at depth two in the search path; see Fig. 7.

[6] This is possible, because a base node does not point to its twin nodes. If the game-theoretic value of a twin node becomes known, its corresponding base node is evaluated accordingly, but other twin nodes remain unchanged.

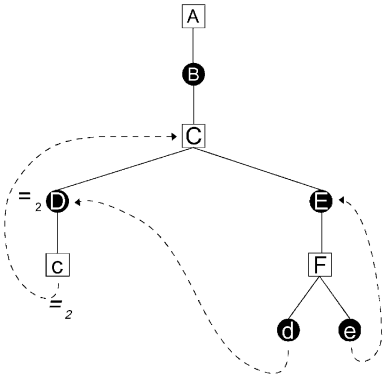

Fig. 7. Encountering the second repetition *c*.

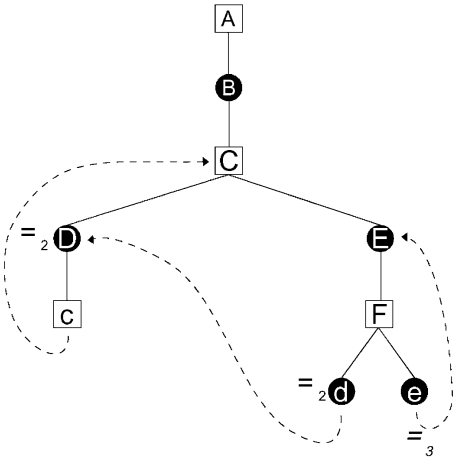

Fig. 8. Encountering the repetition *e*.

After the re-marking of node *c* as a *possible-draw*, the parent of this node (twin node *d*) is re-selected and marked as a *possible-draw*, with the same possible-draw depth as node *c*. Thereafter, the possible-draw marking of node *c* is removed (for the second time). After marking node *d* as a *possible-draw*, its parent *F* is re-selected. The next best child (not marked as a *possible-draw*) *e* is selected. This node is a repetition of node *E* at depth three in the search path, and is now marked as a *possible-draw*; see Fig. 8.

After marking node *e* as a *possible-draw*, the parent of this node (node *F*) is re-selected. All its children are marked as a *possible-draw*. Therefore, node *F* is also marked as a *possible-draw*, with a possible-draw depth of two (the minimum of the possible-draw depths of the children). Further, the possible-draw markings of all children are removed; see Fig. 9.

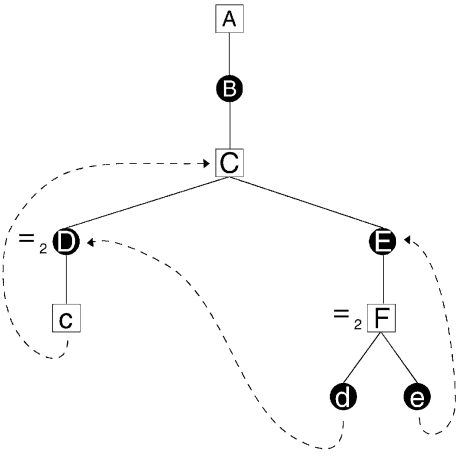

Fig. 9. Marking node F as a *possible-draw*.

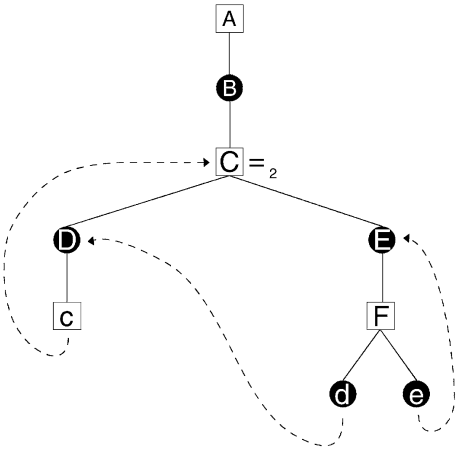

Fig. 10. Marking node C as a *possible-draw*.

After marking node F as a *possible-draw*, the parent of this node (node E) is re-selected and marked as a *possible-draw*, with the same possible-draw depth as node F. Subsequently, the possible-draw marking of node F is removed. After marking node E as a *possible-draw*, its parent (node C) is re-selected. However, all its children are marked as a *possible-draw*. Therefore, node C is also marked as a *possible-draw*, with a possible-draw depth of two (the minimum of the possible-draw depths of the children). Again, the possible-draw markings of all children are removed; see Fig. 10.

Now the selection process finishes, since node C is marked as a *possible-draw and* its corresponding possible-draw depth is equal to the depth of the node in the search path. This means that *all* continuations from C lead, in one or another way, to repetitions

occurring in the subtree of node C. Therefore, node C is evaluated as a real draw by repetition, independent of the history of the node, but on the basis of its potential continuations.

4.2.2. Phase 2: evaluate the best node

In phase 2 the selected node (say P) is evaluated. Three cases are distinguished:

(1) If P is a real draw by repetition, it is evaluated as a draw. The corresponding base node (if existing) is also evaluated as a draw.
(2) If P is a twin node and its corresponding base node is a terminal node, P becomes a terminal node as well and is evaluated as such.
(3) If P is a leaf, it is expanded, the children are evaluated and P is evaluated using the evaluation values of the children.

4.2.3. Phase 3: back up the new information

In phase 3 the value of the selected node is updated to the root[7] and all possible-draw markings are removed. In contrast to the tree algorithm, in the BTA updating process nodes marked as a *possible-draw* may occur. The back-up value of a node is determined by using only the evaluation values of children not marked as a *possible-draw*. Thus, the children marked as a *possible-draw* are ignored, because in the next iteration the search could be mistakenly directed to one of these children, whereas this child was a repetition in the current path, not giving any new information. After establishing the back-up value of a node, the possible-draw markings of the children are removed.

5. The pseudo-code of the algorithm

In this section an implementation of the BTA algorithm in pn search [1] is given. An explanation following the three phases of Section 4 provides details on the seven relevant pn-search procedures and functions. For chess, the goal of pn search is finding a mate. A loss and a real draw are in this respect equivalent (i.e., they are no win). Hence, two types of node with a known game-theoretic value exist: proved nodes (*win*) and disproved nodes (*no win possible*). A proved or disproved node is called a solved node.

5.1. Phase 1: select the most-proving node

Phase 1 of the algorithm deals with the selection of a (best) node for evaluation. This node is termed the most-proving node. In Fig. 11 the main BTA pn-search algorithm is shown. The only parameter of the procedure is root, being the root of the search

[7] In a DCG there can exist more than one path from a node to the root. However, only the path along which the node was selected is taken into account. Other paths, if any, may be updated after other selection processes.

```
procedure BTAProofNumberSearch( root )
 Evaluate( root )
 SetProofAndDisproofNumbers( root )
 root.expanded := false
 root.depth := 0

 while root.proof ≠ 0 and root.disproof ≠ 0 and
   ResourcesAvailable() do begin
  mostProvingNode := SelectMostProvingNode( root )
  ExpandNode( mostProvingNode )
  UpdateAncestors( mostProvingNode.parent, root )
 end

 if root.proof=0 then root.value := true
 elseif root.disproof=0 then root.value := false
 else root.value := unknown /* resources exhausted */
end /* BTAProofNumberSearch */
```

Fig. 11. The BTA pn-search algorithm for DCGs.

tree. The BTA algorithm resembles the tree algorithm described in [1], a difference being that procedure UpdateAncestors is called with the *parent* of the most-proving node as parameter instead of the most-proving node itself, since the most-proving node already has been evaluated in procedure ExpandNode.

The procedures Evaluate and SetProofAndDisproofNumbers and the function ResourcesAvailable are identical to the same procedures and function in the standard tree algorithm (see [7]), and not detailed here. The function SelectMostProvingNode finds a most-proving node according to certain conditions. The function is given in Fig. 12. The only parameter of the function is node, being the root of the (sub)tree where the most-proving node is located.

The function starts to examine whether the node under investigation (say P) is a twin node. If so, then the investigation proceeds with the associated base node.

If P has been solved (case 1), P is returned, because the graph has to be backed up using this new information.

If P has not been solved, it is examined whether P is a repetition in the current path (case 2). If so, it is marked as a *possible-draw*. Its ancestor transposition node in the current path is looked up, and the pdDepth (possible-draw depth) of the node becomes equal to the depth in the search path of the ancestor node.[8] Since it is not

[8] The variable pdDepth will act as an indicator of the lowest level in the tree at which there are nodes having repetition nodes in their subtrees.

```
function SelectMostProvingNode( node )
 if NodeHasBaseNode( node )
 then baseNode := BaseNode( node )
 else baseNode := node
 /* 1: Base node has been solved */
 if baseNode.proof=0 or baseNode.disproof=0
 then return node
 elseif Repetition( node )
 then begin /* 2: Repetition of position */
  MarkAsPossibleDraw( node )
  ancestorNode := FindEqualAncestorNode( node )
  node.pdDepth := ancestorNode.depth
  return SelectMostProvingNode ( node.parent )
 end elseif not baseNode.expanded then /* 3: Leaf */
  return node
 else begin /* 4: Internal node; look for child */
  bestChild := SelectBestChild( node, baseNode, pdPresent )
  if bestChild = NULL then begin
   if pdPresent then begin
    MarkAsPossibleDraw( node )
    node.pdDepth := ∞
    for i:=1 to baseNode.numberOfChildren do begin
     if PossibleDrawSet( baseNode.children[ i ] ) then
      if baseNode.children[ i ].pdDepth < node.pdDepth then
       node.pdDepth := baseNode.children[ i ].pdDepth
      UnMarkAsPossibleDraw( baseNode.children[ i ] )
    end
    if node.depth = node.pdDepth then return node
    else return SelectMostProvingNode( node.parent )
   end else begin
    /* All children are solved, so choose any one */
    baseNode.proof := baseNode.child[ 1 ].proof
    baseNode.disproof := baseNode.child[ 1 ].disproof
    return node
   end
  end else begin
   bestChild.depth := node.depth+1
   return SelectMostProvingNode( bestChild )
  end
 end
end /* SelectMostProvingNode */
```

Fig. 12. The function SelectMostProvingNode.

useful to examine a repetition node further, the selection of the most-proving node is directed to the parent of P.

If P has not been solved and is not a repetition in the current path, it is examined whether P is a leaf (case 3). If so, P is the most-proving node which has to be expanded, and P is returned.

Otherwise (case 4), the best child is selected by the function SelectBestChild, to be discussed later. If no best child was found, it means that every child is either solved (proved in case of an AND node, and disproved in case of an OR node) or is marked as a *possible-draw*. If any of the children is marked as a *possible-draw*, P is marked as a *possible-draw* as well. The pdDepth of the node is set to the minimum of the children's pdDepths and the markings of *all* children are removed, *etc*. See Section 4.

In Fig. 13 the function SelectBestChild is listed. The function has three parameters. The first parameter (node) is the parent from which the best child will be selected. The second parameter (baseNode) is the base node of that parent.[9] Finally, the third parameter (pdPresent, meaning *possible-draw* present) indicates whether one of the children is marked as a *possible-draw*. The parameter pdPresent is initialized by the function SelectBestChild. If the node is an OR node, a child marked as a *possible-draw* will not be selected as best child, since it gains nothing and the goal (win) cannot be reached. A best child (of an OR node) is a child with the lowest proof number. If the node is an AND node, a child marked as a *possible-draw* is a best child, since the player to move in the AND node is satisfied with a repetition (thereby making it impossible for the opponent to reach the goal). Otherwise, a best child (of an AND node) is a child with the lowest disproof number. This best child is returned. If the best child is either solved or marked as a *possible-draw*, NULL is returned.

5.2. Phase 2: evaluate the most-proving node

After the most-proving node has been found, it has to be expanded and evaluated. Phase 2 of the algorithm performs this task. Fig. 14 provides the procedure ExpandNode. The only parameter is node, being the node to be expanded.

The procedure starts establishing the base node of the node.[10] If the base node is solved (case 1), the node is evaluated accordingly.

Otherwise, if the node is marked as a *possible-draw* (case 2) (and since it was chosen by function SelectMostProvingNode), it is evaluated as a real draw.

In case 3 the node has to be expanded. All children are generated, and evaluated. If a generated child has no corresponding base node, the attribute expanded is initialized to false; if it has a corresponding base node, the attribute expanded has been initialized before. Then the node itself is initialized by procedure SetProofAndDisproofNumbers.

[9] We note that if the parent is a base node itself, then the base node is equal to the parent.
[10] We note that if the node is a base node itself, then the base node is equal to the node.

```
function SelectBestChild( node, baseNode, pdPresent )
 bestChild := NULL
 bestValue := ∞
 pdPresent := false
 if node.type=OR then begin /* OR node */
  for i := 1 to baseNode.numberOfChildren do begin
   if PossibleDrawSet( baseNode.children[ i ] ) then
    pdPresent := true
   elseif baseNode.children[ i ].proof < bestValue
   then begin
    bestChild := baseNode.children[ i ]
    bestValue := bestChild.proof
   end
  end
 end else begin /* AND node */
  for i := 1 to baseNode.numberOfChildren do begin
   if PossibleDrawSet( baseNode.children[ i ] ) then begin
    pdPresent := true
    break
   end
   if baseNode.children[ i ].disproof < bestValue
   then begin
    bestChild := baseNode.children[ i ]
    bestValue := bestChild.disproof
   end
  end
 end

 return bestChild
end /* SelectBestChild */
```

Fig. 13. The function SelectBestChild.

5.3. Phase 3: back up the new information

Phase 3 of the algorithm has as task to back up the evaluation value of the most-proving node. The procedure to update the values of the nodes in the path is listed in Fig. 15. The procedure has two parameters. The first parameter (node) is the node to be updated, while the second parameter (root) is the root of the search tree. Depending on the node type, UpdateOrNode (Fig. 16) or UpdateAndNode (Fig. 17) is performed.

The parameters of UpdateOrNode are node and baseNode. The algorithm basically is the same as the OR part of procedure SetProofAndDisproofNumbers. It only differs

```
procedure ExpandNode( node )
 if NodeHasBaseNode( node )
 then baseNode := BaseNode( node )
 else baseNode := node

 if baseNode.proof=0 or baseNode.disproof=0 then begin
  /* 1: base node already solved */
  node.proof := baseNode.proof
  node.disproof := baseNode.disproof
 end elseif PossibleDrawSet( node ) then begin
  /* 2: node has become a real draw */
  node.proof := ∞
  node.disproof := 0
  baseNode.proof := ∞
  baseNode.disproof := 0
 end else begin
  /* 3: node has to be expanded */
  GenerateAllChildren( baseNode )
  for i:=1 to baseNode.numberOfChildren do begin
   Evaluate( baseNode.children[ i ] )
   SetProofAndDisproofNumbers( baseNode.children[ i ] )
   if not NodeHasBaseNode( baseNode.children[ i ] ) then
    baseNode.children[ i ].expanded := false
  end
  SetProofAndDisproofNumbers( baseNode )
  baseNode.expanded := true
  node.proof := baseNode.proof
  node.disproof := baseNode.disproof
 end
end /* ExpandNode */
```

Fig. 14. The procedure ExpandNode.

when a child is marked as a *possible-draw*. In that case, the child is discarded so its value is not used when calculating the back-up value of the node. Then, the possible-draw marking of the child is removed. If the node appears to be disproved (since all children are either disproved or marked as a *possible-draw*) and a repetition child exists, the value of the node is calculated by procedure SetProofAndDisproofNumbers. Otherwise, the value has been calculated correctly. If the node has been solved, its base node is evaluated accordingly.

The two parameters of UpdateAndNode are equal to the parameters of procedure UpdateOrNode. The procedure differs from the AND part of the procedure

```
procedure UpdateAncestors( node, root )
 while node ≠ nil do begin
  if NodeHasBaseNode( node )
  then baseNode := BaseNode( node )
  else baseNode := node

  if node.type=OR
  then UpdateOrNode( node, baseNode )
  else UpdateAndNode( node, baseNode )

  node := node.parent /* parent in current path */
 end
 if PossibleDrawSet( root ) then
  UnMarkAsPossibleDraw( root )
end /* UpdateAncestors */
```

Fig. 15. The procedure UpdateAncestors.

SetProofAndDisproofNumbers when the node is solved, and hence the value of its base node is evaluated accordingly. [11]

6. Experimental

6.1. The proof-number search engine

The proof-number search engine has been implemented in a straightforward chess program. The only goal of the pn-search algorithm is searching for mate. We distinguish between the attacker and the defender. A position is proved if the attacker can mate, while draws (stalemate, repetition of positions, and the 50-move rule) and mates by the defender are defined to be disproved positions for the attacker.

6.2. The test set

Since proof-number search operates best when searching for mates in chess [5], we used a set of mating problems [11, 15]. Krabbé's 35 positions are indicated by kx, in which x refers to the diagram number in the source. The diagrams are 8, 35, 37, 38, 40, 44, 60, 61, 78, 192, 194, 195, 196, 197, 198, 199, 206, 207, 208, 209, 210, 211, 212, 214, 215, 216, 217, 218, 219, 220, 261, 284, 317, 333 and 334. Reinfeld's 82 positions are indicated by rx, x again referring to the problem number in the source, this time running over 1, 4, 5, 6, 9, 12, 14, 27, 35, 49, 50, 51, 54, 55, 57, 60, 61, 64,

[11] We note that it is impossible for a child of an AND node to be marked as a *possible-draw*, since in that case the search for a most-proving node would have been terminated in an earlier phase, and the parent already would have been marked as a *possible-draw*.

```
procedure UpdateO2Node( node, baseNode )
min := ∞
sum := 0
pdPresent := false
for i:=1 to baseNode.numberOfChildren do begin
 if PossibleDrawSet( baseNode.child[ i ] ) then begin
  pdPresent := true
  proof := ∞
  disproof := 0
  UnMarkAsPossibleDraw ( baseNode.child[ i ] )
 end else begin
  proof := baseNode.child[ i ].proof
  disproof := baseNode.child[ i ].disproof
 end
 if proof < min then min := proof
 sum := sum + disproof
end
if min=∞ and pdPresent then
 SetProofAndDisproofNumbers( node )
else begin
 node.proof := min
 node.disproof := sum
end
if node.proof=0 or node.disproof=0
then begin /* node solved */
 baseNode.proof := node.proof
 baseNode.disproof := node.disproof
end
end /* UpdateO2Node */
```

Fig. 16. The procedure UpdateO$_2$Node.

79, 84, 88, 96, 97, 99, 102, 103, 104, 105, 132, 134, 136, 138, 139, 143, 154, 156, 158, 159, 160, 161, 167, 168, 172, 173, 177, 179, 182, 184, 186, 188, 191, 197, 201, 203, 211, 212, 215, 217, 218, 219, 222, 225, 241, 244, 246, 250, 251, 252, 253, 260, 263, 266, 267, 278, 281, 282, 283, 285, 290, 293, 295 and 298. This results in a test set of 117 positions.

6.3. The setting

Our BTA algorithm, denoted by *BTA*, is compared with the following three pn-search variants:

(1) the standard tree algorithm, denoted by *Tree*,

```
procedure UpdateAndNode( node, baseNode )
  min := ∞
  sum := 0
  for i:=1 to baseNode.numberOfChildren do begin
    proof := baseNode.child[ i ].proof
    disproof := baseNode.child[ i ].disproof
    sum := sum + proof
    if disproof < min then min := disproof
  end

  node.proof := min
  node.disproof := sum
  if node.proof=0 or node.disproof=0
  then begin /* node solved */
    baseNode.proof := node.proof
    baseNode.disproof := node.disproof
  end
end /* UpdateAndNode */
```

Fig. 17. The procedure UpdateAndNode.

(2) a DAG algorithm, developed by Schijf [16], denoted by *DAG*, and
(3) an (incorrect) DCG algorithm, developed by Schijf et al. [17], denoted by *DCG*.

The results for the DAG and DCG algorithm will be taken from the literature [16, 17]. In all implementations, the move ordering is identical. All four algorithms searched for a maximum of 500,000 nodes per test position. After 500,000 nodes the search was terminated and the problem was marked as not solved. Under these conditions 10 positions (k8, k40, k78, k195, k209, k210, k220, r96, r105, r201) turned out to be not solvable by any of the four algorithms. Therefore they are not taken into account in the next section.

7. Results

To verify our solution we have first tested the position given in Fig. 1.[12] *Tree* finds a solution within 482,306 nodes. *DCG*, ignoring the history of a position, incorrectly states that White cannot win (due to the GHI problem). Our *BTA* does find a solution within 10,694 nodes. This provides evidence that this occurrence of the GHI problem has been correctly handled. *BTA* shows the benefit of being a DCG algorithm, as

[12] We note that for this problem the goal for White was set to promotion to Queen (without Black being able to capture it on the next ply) instead of mate. Further, the search was restricted to the 5×5 a4–e8 board. This helps to find the solution faster, but does not influence the occurrence of the GHI problem.

Table 1
Comparing four pn-search algorithms

	No. of positions solved (out of 117)	Total nodes (96 positions)
Tree	99	4,903,374
DAG	104	3,222,234
DCG	103	2,482,829
BTA	107	2,844,024

evidenced by the decrease in number of nodes investigated by a factor of roughly 40 as compared to *Tree*.

Thereafter, we have performed the experiments with the test set described in Section 6.2. The outcomes are summarized in Table 1; the complete results are given in the appendix. The first column of Table 1 shows the four pn-search variants. The number of positions solved by each algorithm is given in the second column. Exactly 96 positions were solved by all four algorithms. In the third column the total number of nodes evaluated for the 96 positions are listed. The additional positions solved per algorithm are as follows:

- for *Tree*: k208, k215, r281;
- for *DAG*: k60, k208, k215, k216, k284, r168, r182, r281;
- for *DCG*: k44, k60, k217, k284, r168, r182, r252;
- for *BTA*: k44, k60, k208, k215, k216, k217, k284, r168, r182, r252, r281.

Obviously, *Tree* investigates the largest number of nodes, the easy explanation being that this algorithm does not recognize transpositions. Further, *DCG* examines the smallest number of nodes: this algorithm sometimes prematurely disproves positions; hence, on the average less nodes have to be examined. However, if such a prematurely disproved position does lead to a win and the node is important to the principal variation of the tree, the win can be missed, as happens in the positions k208, k215, k216 and r281. This is already mentioned by Schijf et al. [17].

From Table 1 it further follows that *BTA* performs best. It solves each position which was solved by at least one of the other three algorithms. Furthermore, the four positions which were incorrectly disproved by *DCG* were proved by *BTA*. Compared to the tree algorithm, *BTA* solves eight additional positions and uses only 58% of the number of nodes: a clear improvement. The reduction in nodes compared to *DAG* is still 11.7%. The increase in nodes searched relative to *DCG* (12.7%) is already explained by the unreliability of the latter. We feel that the advantage of the larger number of solutions found heavily outweighs the disadvantage of the increase in nodes searched. We note that the selection of the most-proving node in *BTA* can be costly in positions with many possible transpositions. However, in these types of position the reduction in the number of nodes searched is even larger than in "normal" positions.

Below we illustrate the results by two examples. The position of Fig. 18 corresponds with Diagram 44 in Krabbé [11]. White has a multitude of moves at his disposal; Black only has a few moves. For the standard pn-search algorithm (*Tree*), all white moves

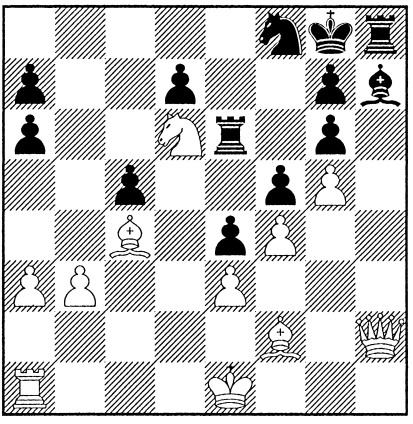

Fig. 18. Mate in 6 (WTM); (N. Macleod).

are equally good. Therefore, *Tree* does not solve the position (within 500,000 nodes). Although *DAG* recognizes transpositions, no solution is found (within 500,000 nodes). The DCG algorithm and the BTA algorithm recognize transpositions and are able to search cyclic graphs adequately. Consequently, they find a solution. *DCG* solves the position in 274,211 nodes, and *BTA* solves it in 146,938 nodes. For the chess-playing reader, the intended solution is **1. 0–0–0**... **2.** ♗**e1**... **3.** ♗**c3**... **4.** ♗**a1**... **5.** ♕**b2**... **6.** ♕**xg7 mate**. Black's moves are not mentioned, since Black cannot prevent this mate.

The position of Fig. 19 corresponds with Diagram 208 in Krabbé [11]. Many transpositions (and many repetitions of positions) exist, since White has a so-called *zwickmühle* and can position the Rook anywhere along the d-file for free. For instance, after **1.** ♖**d2** + ♔**e1 2.** ♖**d3** + ♔**e2** almost the same position with the same player to move has been reached: the Rook has moved from d7 to d3. At any time White can choose such a manoeuvre. *Tree*, *DAG*, and *BTA* solve the position in 72,468 nodes, 65,279 nodes, and 31,648 nodes, respectively. *DCG* fails to find a solution because one or more crucial winning positions were inaccurately evaluated as a draw by repetition. Popandopulo's intended solution was **1.** ♖**d2**+ ♔**e1 2.** ♖**d3**+ ♔**e2 3. e4 f4 4.** ♖**d2**+ ♔**e1 5.** ♖**d4**+ ♔**e2 6. e5 f5 7.** ♖**d2**+ ♔**e1 8.** ♖**d5**+ ♔**e2 9. e6 f6 10.**♖**d2**+ ♔**e1 11.** ♖**d6**+ (**11.** ♖**d8**+ ♔**e2 12. e7** ♘**c7 13. e8** = ♕ + ♘**xe8 14.** ♖**d2**+ ♔**e1 15.** ♖**d6**+ ♔**e2 16.** ♖ **e6** mate is two moves faster than the intended solution.) **11.** ... ♔**e2 12. e7** ♘**c7 13.** ♖**d2**+♔**e1 14.** ♖**d8**+♔**e2 15. e8** = ♕ +♔**xe8 16.**♖**d2**+♔**e1 17.** ♖**d6**+♔**e2 18.** ♖**e6 mate**.

8. Conclusions

In this article we have given a solution to the GHI problem for best-first search, resulting in an improved DCG algorithm for pn search, denoted BTA (Base-Twin

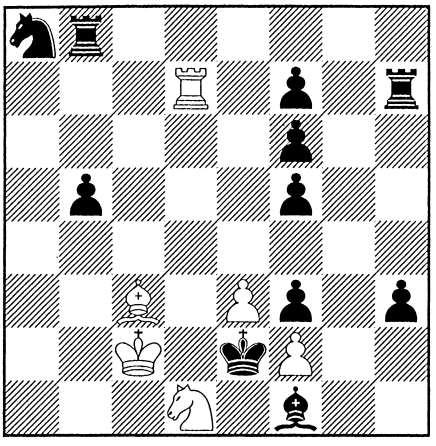

Fig. 19. Mate in 18 (WTM); (A. Popandopulo).

Algorithm). It is shown that in a well-known position, in which the GHI problem occurs when a naïve DCG algorithm is used, our BTA algorithm finds the correct solution. The results on a test set of 117 selected positions support our claim. Despite the additional overhead to recognize positions suffering from the GHI problem, our BTA algorithm is hardly less efficient than other, non-reliable, DCG algorithms, and finds more solutions.

We note that, though our algorithms are confined to pn search, the strategy used is generally applicable to any best-first search algorithm. The only important criterion for application is that a DCG is being built according to the best-first principle (choose some leaf node, expand that node, evaluate the children, and back up the result). We consider the GHI problem in best-first search as solved. The importance of this statement is that the increasing availability of computer memory stimulates a growing tendency to use best-first search algorithms, such as SSS* [18] and variants thereof, or best-first fixed-depth algorithms [14], especially since they no longer suffer from the GHI problem. What remains is solving the GHI problem for depth-first search. This will need a different approach, storing additional information in transposition tables rather than in the search tree/graph in memory. Since Campbell already noted that in depth-first search the frequency of GHI problems is considerably smaller than in best-first search [8], the solution of the GHI problem for depth-first search remains a nearly theoretical exercise.

Appendix. Detailed results

Table 2 contains the results of the 117 test positions for four distinct pn-search algorithms using the same move ordering. The numbers in the columns two to five indicate the number of nodes searched. A dash signifies that no solution was found due to the

Table 2
The results for four distinct pn-search algorithms

	Tree	*DAG*	*DCG*	*BTA*
k8	—	—	—	—
k35	296	296	276	276
k37	35,724	25,737	17,981	19,886
k38	273	273	272	272
k40	—	—	—	—
k44	—	—	274,211	146,938
k60	—	310,251	372,634	487,969
k61	43,911	41,997	35,770	38,446
k78	—	—	—	—
k192	22,525	15,429	14,252	15,767
k194	238,085	51,427	30,699	102,336
k195	—	—	—	—
k196	318,276	97,717	88,069	93,447
k197	429	429	417	413
k198	333,165	262,255	171,720	177,929
k199	369,555	290,903	151,043	202,903
k206	11,931	11,543	9483	9191
k207	236,568	88,024	41,348	50,870
k208	72,468	65,279	—	31,648
k209	—	—	—	—
k210	—	—	—	—
k211	1059	1059	939	937
k212	83,413	59,988	52,946	55,290
k214	645	645	629	624
k215	124,984	94,108	—	74,967
k216	—	366,336	—	247,686
k217	—	—	311,027	407,633
k218	122,058	109,308	124,868	107,215
k219	277,250	129,232	63,297	83,329
k220	—	—	—	—
k261	414	388	388	424
k284	—	2337	2337	2851
k317	157,424	120,358	103,033	94,043
k333	165,725	134,339	123,599	139,184
k334	145,291	88,430	74,375	82,889
r1	4275	4095	3996	4270
r4	82	82	82	82
r5	57	57	57	57
r6	96,059	32,953	11,703	13,179
r9	173	173	169	168
r12	99	99	99	99
r14	335,936	213,098	157,269	185,918
r27	77	77	77	77
r35	597	559	371	522
r49	16,935	14,767	13,797	15,625
r50	399	383	369	408
r51	270,495	191,822	173,922	204,299
r54	256	256	256	256
r55	15,245	13,293	12,749	14,552

Table 2 (*Continued*)

r57	287	287	287	287
r60	69	69	69	69
r61	78	78	78	78
r64	153	153	153	153
r79	152	152	152	152
r84	93	93	93	93
r88	595	547	542	583
r96	—	—	—	—
r97	107	107	107	107
r99	31,767	31,302	27,264	27,290
r102	199	199	199	199
r103	1837	1742	1731	1780
r104	5042	4660	4658	4870
r105	—	—	—	—
r132	2291	2105	2077	2135
r134	804	798	758	760
r136	230	230	230	230
r138	192,886	164,106	118,729	137,030
r139	182	182	182	182
r143	521	520	520	519
r154	197	197	196	196
r156	82	82	82	82
r158	495	495	494	494
r159	403,797	253,108	274,275	263,275
r160	110	110	110	110
r161	1790	1209	1209	1332
r167	923	901	813	810
r168	—	317,557	209,725	301,527
r172	99	99	99	99
r173	419	418	404	402
r177	349	349	349	349
r179	156	156	156	156
r182	—	230,648	212,087	372,096
r184	82	82	82	82
r186	108	108	108	108
r188	117	117	117	117
r191	22,830	20,480	17,858	17,046
r197	95	95	95	95
r201	—	—	—	—
r203	20,980	18,429	17,265	17,397
r211	278	272	231	230
r212	545	545	543	543
r215	164	164	164	164
r217	199	199	199	199
r218	270,277	225,638	160,720	210,311
r219	140	140	140	140
r222	60,855	48,209	22,827	49,934
r225	263	263	263	263
r241	365,495	254,998	195,577	231,746
r244	323	323	323	323
r246	61	61	61	61

Table 2 (*Continued*)

r250	1102	1101	1076	1071
r251	88,547	70,104	53,285	57,798
r252	—	—	352,315	386,046
r253	2709	1189	1176	2477
r260	841	794	729	804
r263	654	621	621	651
r266	716	716	711	711
r267	1136	1001	1001	1089
r278	333	333	333	333
r281	316,252	93,578	—	46,033
r282	749	729	725	742
r283	14,787	14,530	14,070	14,165
r285	218	218	218	218
r290	408	408	408	408
r293	97,666	94,138	75,283	75,509
r295	134	134	134	134
r298	150	150	150	150

memory constraints (500,000 nodes). The first column lists the test positions. Columns two to five show the results for the *Tree* algorithm, the *DAG* algorithm, the *DCG* algorithm, and the *BTA* algorithm, respectively. For the sources of the experiments we refer to Section 6, and for details on the results to Section 7.

References

[1] L.V. Allis, M. van der Meulen, H.J. van den Herik, Proof-number search, Artificial Intelligence 66 (1) (1994) 91–124.
[2] L.V. Allis, Searching for solutions in games and artificial intelligence, Ph.D. Thesis, University of Limburg, Maastricht, The Netherlands, 1994.
[3] E.B. Baum, W.D. Smith, Best play for imperfect players and game tree search, Technical Report, NEC, Princeton, NJ, USA.
[4] H.J. Berliner, C. McConnell, B* probability based search, Artificial Intelligence 86 (1) (1996) 97–156.
[5] D.M. Breuker, L.V. Allis, H.J. van den Herik, How to mate: applying proof-number search, in: H.J. van den Herik, I.S. Herschberg, J.W.H.M. Uiterwijk (Eds.), Advances in Computer Chess 7, University of Limburg, Maastricht, The Netherlands, 1994, pp. 251–272.
[6] D.M. Breuker, H.J. van den Herik, L.V. Allis, J.W.H.M. Uiterwijk, A solution to the GHI problem for best-first search, Technical Report No. CS 97-02, Computer Science Department, Universiteit Maastricht, Maastricht, The Netherlands, 1997.
[7] D.M. Breuker, Memory versus search in games, Ph.D. Thesis, Universiteit Maastricht, Maastricht, The Netherlands, 1998.
[8] M. Campbell, The graph-history interaction: on ignoring position history, in: 1985 Association for Computing Machinery Annual Conf., ACM, New York, 1985, pp. 278–280.
[9] R.M. Hyatt, A.E. Gower, H.L. Nelson, Cray Blitz, in: D.F. Beal (Ed.), Advances in Computer Chess 4, Pergamon Press, Oxford, UK, 1984, pp. 8–18.
[10] B. Kažić, R. Keene, K.A. Lim, The Official Laws of Chess and Other FIDE Regulations, B.T. Batsford Ltd., London, UK, 1985.
[11] T. Krabbé, Chess Curiosities, George Allen and Unwin, London, UK, 1985.
[12] T.A. Marsland, A review of game-tree pruning, ICCA J. 9 (1) (1986) 3–19.
[13] A.J. Palay, Searching with probabilities, Ph.D. Thesis, Boston University, Boston, MA, USA, 1985.

[14] A. Plaat, J. Schaeffer, W. Pijls, A. de Bruin, Best-first fixed-depth minimax algorithms, Artificial Intelligence 87 (2) (1996) 255–293.
[15] F. Reinfeld, Win at Chess, Dover Publications, New York, NY, USA, 1958, originally published as Chess Quiz, David McKay Company, New York, NY, USA, 1945.
[16] M. Schijf, Proof-number search and transpositions, M.Sc. Thesis, University of Leiden, Leiden, The Netherlands, 1993.
[17] M. Schijf, L.V. Allis, J.W.H.M. Uiterwijk, Proof-number search and transpositions, ICCA J. 17 (2) (1994) 63–74.
[18] G. Stockman, A minimax algorithm better than alpha-beta?, Artificial Intelligence 12 (1979) 179–196.
[19] K. Thompson, personal communication, 1995.