Chapter 5

The graph-history-interaction problem

This chapter is an updated and abridged version of

1. Breuker D.M., Herik H.J. van den, Allis L.V., and Uiterwijk J.W.H.M. (1997a). A Solution to the GHI Problem for Best-First Search. *Proceedings of the Ninth Dutch Conference on Artificial Intelligence* (eds. K. van Marcke and W. Daelemans), pp. 457–468. University of Antwerp, Antwerp, Belgium, and
2. Breuker D.M., Herik H.J. van den, Allis L.V., and Uiterwijk J.W.H.M. (1998a). A Solution to the GHI Problem for Best-First Search. Submitted as journal publication. Also published (1997) as Technical Report CS 97-02, Universiteit Maastricht, Maastricht, The Netherlands.

5.1 The history of a position

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. For details, see Section 2.3. 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 usually 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.

These common ways of dealing with transpositions contain an important flaw: determining whether *nodes* are identical is not the same as determining whether the *search states* represented by the nodes are identical (cf. Section 2.1). 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 (Kažić *et al.*, 1985).

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. Palay, 1985; Campbell, 1985).

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 done so far, e.g., the cost of preparing a machine for performing some process depends on the state left after the previous process.

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. A disadvantage of such a solution is that too many properties may be relevant, resulting in the need for 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 is necessary to determine whether a draw by three-fold repetition has been encountered. Unfortunately, it 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 (cf. Section 2.2). Campbell (1985) mentioned that, contrary to best-first search (which he calls selective search), in depth-first search the GHI problem occurs less frequently.

In this chapter we deal with the third problem statement: is it possible to give a solution for the GHI problem for best-first search? A solution to the GHI problem for best-first search is presented with only a few relevant properties included in a node. In Section 5.2 an example of the GHI problem is given. Previous work on the GHI problem is discussed in Section 5.3. In Section 5.4 the general solution to the GHI problem for best-first search is described. A formalized description and the pseudo-code for the implementation in pn search is given in Section 5.5. Section 5.6 lists experiments with the new algorithm. It is compared to three other pn-search variants. The results are presented in Section 5.7. Finally, Section 5.8 provides conclusions.

5.2 An example of the GHI problem

Figure 5.1 shows a pawn endgame position, taken from Campbell (1985), where the GHI problem can occur. White (to move) has achieved a potentially won position. However, we show that it is possible to evaluate this position incorrectly as a draw. In this chapter we assume that a single repetition of positions evaluates to a draw, in contrast with the FIDE ruling which stipulates that the same position must occur three times.

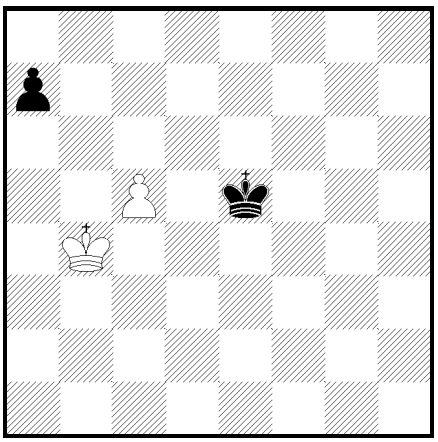

Figure 5.1: A pawn endgame (WTM).

In Figure 5.2 a relevant part of the search tree is depicted. 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) is evaluated as a draw. Backing up this draw leads to the incorrect conclusion that node A evaluates to 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 to a win.

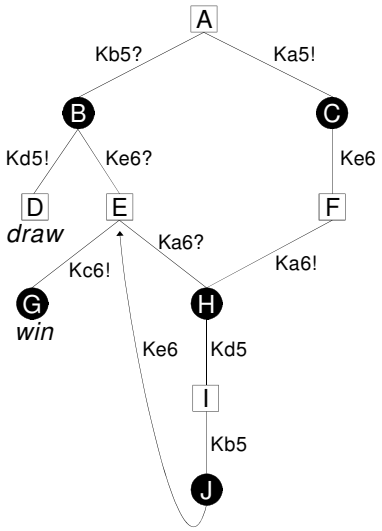

Figure 5.2: The GHI problem in the pawn endgame.

An example of the general case is given in Figure 5.3. 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 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 the 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) (Figure 5.4).

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

[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.

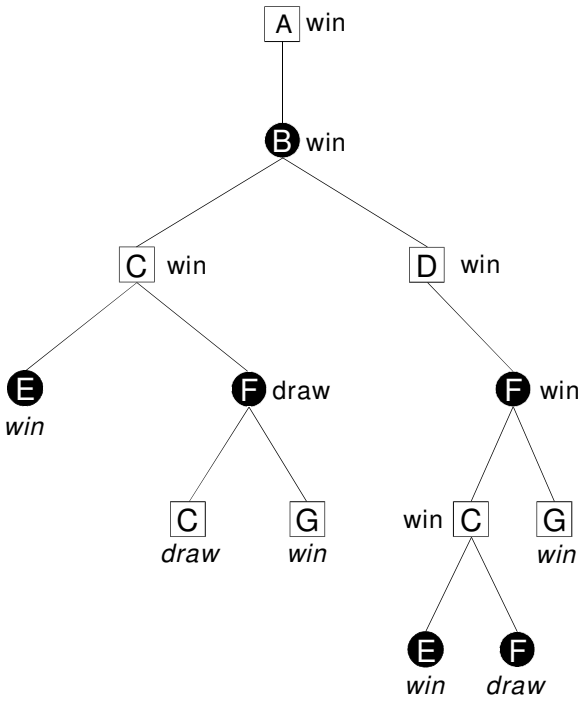

Figure 5.3: A search tree with repetitions.

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 mentioned case it is a draw, and in the second case it is a win, due to the values of E and G.

5.3 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 (1985) first identified the GHI problem. 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 now

[2] Berliner and McConnell (1996) suggested the use of conditional values as an idea to solve the GHI problem. They promised details in a forthcoming paper.

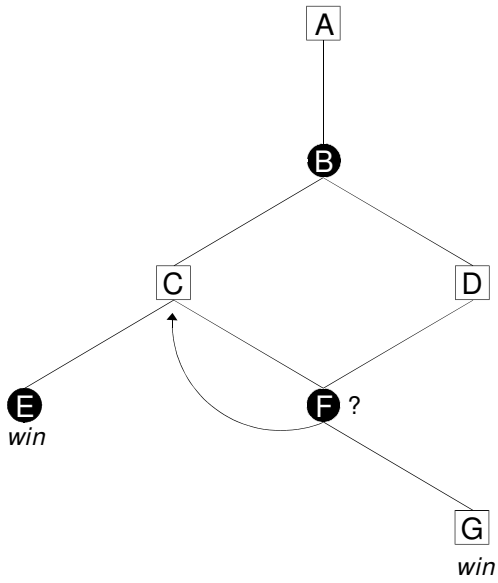

Figure 5.4: The DCG corresponding with the tree of Figure 5.3.

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 (Campbell, 1985). While building a tactical analyzer, Thompson (1995) used a DCG representation. He saw it suffering from the GHI problem. 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 (1985) discussed the GHI problem thoroughly, applying it to depth-first search only. The key in avoiding most occurrences of the GHI problem appears to be iterative deepening. Some problems (called "draw-first") can be overcome[3]. However,

[3] In the draw-first case node F in Figure 5.4 is first reached through path A–B–C–F (and the

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 (1995) stumbled on the GHI problem when implementing their best-first search algorithm BPIP (Best Play for Imperfect Players). 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 the idea of Baum and Smith.

Schijf *et al.* (1994) investigated the problem in the context of pn search (Allis *et al.*, 1994). 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 (1993) proved that the most-proving node always exists in a DAG. 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 be updated. The second modification is that when a child is generated, it has to be checked whether this node is a

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 Figure 5.4 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.

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.* (1994) note that this algorithm contains two flaws. 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 (Schijf, 1993). The DAG algorithm uses considerably fewer nodes (viz. a factor of five) to prove the game-theoretic value of tic-tac-toe. For the problem of applying pn search to a DCG, Schijf *et al.* (1994) give a time-and-memory-efficient algorithm, which, however, sometimes inaccurately evaluates nodes as a draw by repetition. They remark that, as a consequence, their algorithm is sometimes unable to find the goal, even though it should have found it.

5.4 BTA: an enhanced DCG algorithm

In this section we describe a new algorithm (denoted BTA: Base-Twin Algorithm) for solving the GHI problem for best-first search. The algorithm had been developed in a joint effort with Victor Allis. Its correctness has been proven experimentally. A formal proof is beyond the scope of this research. The description given below provides a clarity of reasoning, which in our opinion, is sufficiently convincing in its own.

The BTA algorithm is based on the distinction of two types of nodes, termed *base nodes* and *twin nodes*. The purpose of these types is to distinguish between identical positions with different history. Although it was known in the DCG algorithm described by Schijf *et al.* (1994) 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 that 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 5.1) 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 5.1)

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.

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 nodes: *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 represented by 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.

Figure 5.5 exhibits our implementation of the DCG given in Figure 5.4 (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 Figure 5.4 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.

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 new concept of *possible-draw*. Node p is marked as a *possible-draw* if a node is a repetition of a node P in the search path. (Whether a possible draw also is a real draw depends on the history.) Then the depth of node P in the search path (termed the *possible-draw depth*) is 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.

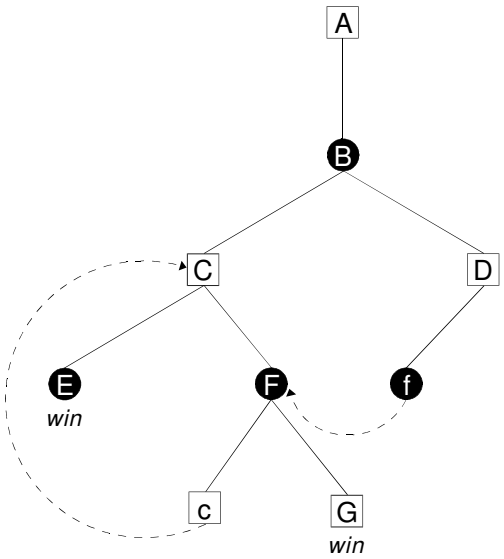

Figure 5.5: Our DCG with base nodes and twin nodes corresponding with the DCG of Figure 5.4.

5.4.1 Phase 1: select the best node

In phase 1 a node is selected for evaluation[5]. This is accomplished in a way similar to the best-first tree algorithm (see Section 2.1). For comparison, a short outline of the tree algorithm is given. First, the root is selected. Next, a best child from the selected node is selected according to the best-first-search criteria. The last step is repeated until (1) a repetition has been encountered (evaluating to a draw), or (2) a leaf has been found.

The selection of a node in the BTA algorithm is as follows. 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, a best child is selected for investigation, ignoring the children which are either real draws, or marked as a *possible-draw*.

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

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. This implies that the possible-draw marking of this node is based solely on repetitions of positions *in the subtree of the node* and on real draws. Therefore, the node is a real draw by repetition, independent of the history of the node. Hence, the node is evaluated accordingly.

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 Figure 5.6 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 Figure 5.6 (the equal sign represents the possible-draw marking and the subscript two represents the possible-draw depth).

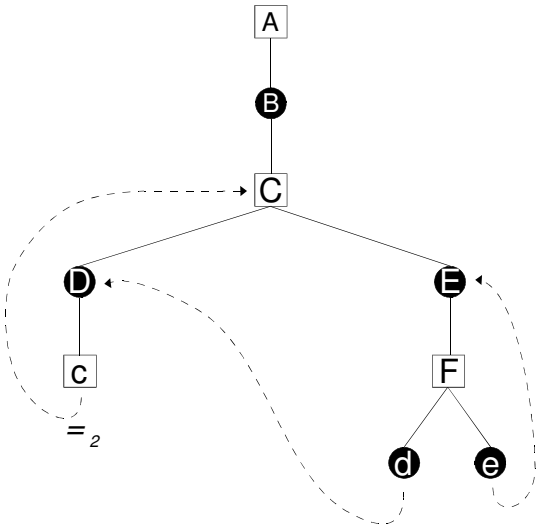

Figure 5.6: Encountering the first repetition c.

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

[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.

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 Figure 5.7.

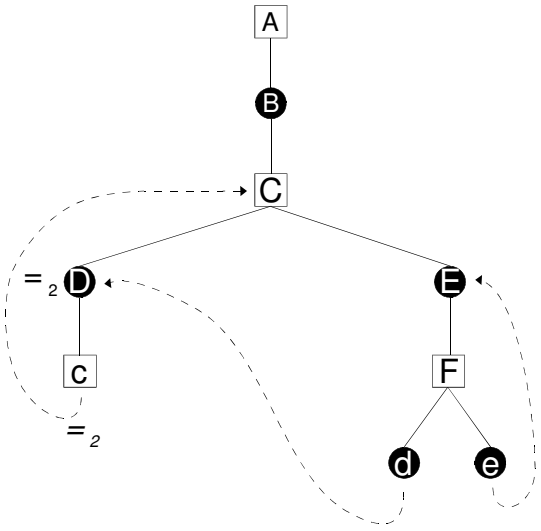

Figure 5.7: Encountering the second repetition c.

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 marked as a *possible-draw*. See Figure 5.8.

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 Figure 5.9.

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 Figure 5.10.

Now the selection process finishes, since node C at depth two in the search path

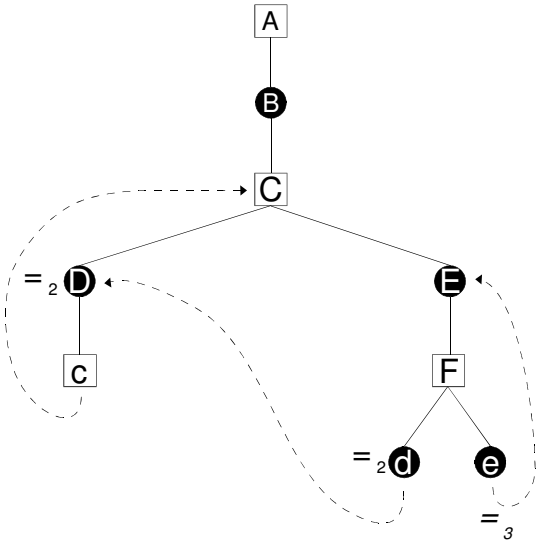

Figure 5.8: Encountering the repetition e.

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.

5.4.2 Phase 2: evaluate the best node

In phase 2 the selected node (say node P) is evaluated. For comparison, again a short outline of the tree algorithm is given. The evaluation of node P is dependent on the condition under which phase 1 has terminated.

1. If node P is a repetition, it is evaluated as a draw.

2. If node P is a leaf, it is expanded, the children are evaluated and node P is evaluated using the evaluation values of the children.

For the evaluation of node P in the BTA algorithm three cases are distinguished.

1. If node 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 node P is a twin node and its corresponding base node is a terminal node, node P becomes a terminal node as well and is evaluated as such.

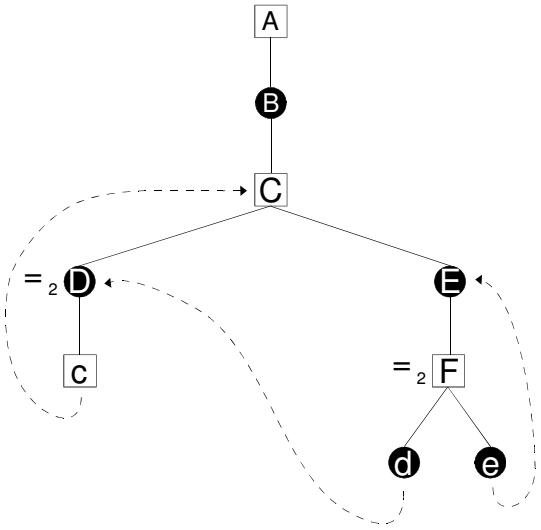

Figure 5.9: Marking node F as a *possible-draw*.

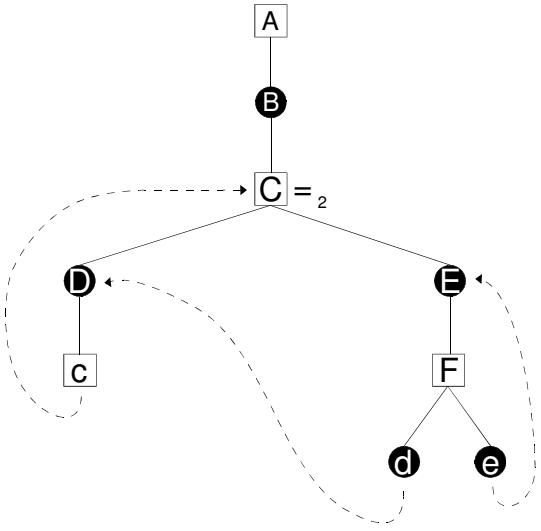

Figure 5.10: Marking node C as a *possible-draw*.

3. If node P is a leaf, it is expanded, the children are evaluated, and node P is evaluated using the evaluation values of the children.

5.4.3 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.

5.5 The pseudo-code of the BTA algorithm

In this section an implementation of the BTA algorithm in pn search (see Chapter 3) is given. An explanation following the three phases of Section 5.4 provides details on the seven relevant pn-search procedures and functions. We will make use of several properties of pn search, in order to simplify and accelerate the general BTA algorithm. 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 nodes 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.

5.5.1 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 Figure 5.11 the main BTA pn-search algorithm is shown. The only parameter of the procedure is root, being the root of the search tree. The BTA algorithm resembles the tree algorithm described in Section 3.2, a difference being that procedure UpdateAncestors is called with the *parent* of the most-proving node as the 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 Figure 3.2), and not detailed here. The function SelectMostProvingNode finds a most-proving node, according to certain conditions. The function is given in Figure 5.12. 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 node P) is a twin node. If so, then the investigation proceeds with the associated base node.

[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 */
```

Figure 5.11: The BTA pn-search algorithm for DCGs.

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

If node P has not been solved, it is examined whether node 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 useful to examine a repetition node further, the selection of the most-proving node is directed to the parent of node P.

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

Otherwise (case 4), a 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*, the node 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 5.4.

In Figure 5.13 the function SelectBestChild is listed. The function has three parameters. The first parameter (node) is the parent from which a best child will be

[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.child[ i ] ) then
                        if baseNode.child[ i ].pdDepth<node.pdDepth then
                            node.pdDepth := baseNode.child[ i ].pdDepth
                        UnMarkAsPossibleDraw( baseNode.child[ 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 */
```

Figure 5.12: The function SelectMostProvingNode.

```
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.child[ i ] ) then
                pdPresent := true
            elseif baseNode.child[ i ].proof<bestValue then begin
                bestChild := baseNode.child[ i ]
                bestValue := bestChild.proof
            end
        end
    end else begin /* AND node */
        for i:=1 to baseNode.numberOfChildren do begin
            if PossibleDrawSet( baseNode.child[ i ] ) then begin
                pdPresent := true
                break
            end
            if baseNode.child[ i ].disproof<bestValue then begin
                bestChild := baseNode.child[ i ]
                bestValue := bestChild.disproof
            end
        end
    end

    return bestChild
end /* SelectBestChild */
```

Figure 5.13: The function SelectBestChild.

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

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

is returned. If the best child is either solved or marked as a *possible-draw*, NULL is returned.

5.5.2 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. Figure 5.14 provides the procedure ExpandNode. The only parameter is node, being the node to be expanded.

```
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.child[ i ] )
            SetProofAndDisproofNumbers( baseNode.child[ i ] )
            if not NodeHasBaseNode( baseNode.child[ i ] ) then
                baseNode.child[ i ].expanded := false
        end
        SetProofAndDisproofNumbers( baseNode )
        baseNode.expanded := true
        node.proof := baseNode.proof
        node.disproof := baseNode.disproof
    end
end /* ExpandNode */
```

Figure 5.14: The procedure ExpandNode.

The procedure starts establishing the base node of the node[10]. If the base node

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

5.5.3 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 for updating the values of the nodes in the path is listed in Figure 5.15. 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 (Figure 5.16) or UpdateAndNode (Figure 5.17) is performed.

```
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 */
```

Figure 5.15: The procedure UpdateAncestors.

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

```
procedure UpdateOrNode( 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 /* UpdateOrNode */
```

Figure 5.16: The procedure UpdateOrNode.

The two parameters of UpdateAndNode are equal to the parameters of procedure UpdateOrNode. The procedure differs from the AND part of the procedure SetProofAndDisproofNumbers when the node is solved, and hence the value of its base node is evaluated accordingly[11].

```
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 */
```

Figure 5.17: The procedure UpdateAndNode.

5.6 Experimental set-up

In this section give the experimental set-up for evaluating the BTA pn-search algorithm presented in Section 5.5. The game of chess is used as the test domain. Our BTA algorithm, denoted by *BTA*, is compared with the following three pn-search variants:

1. the standard tree algorithm (see Section 3.2), denoted by *Tree*,
2. a DAG algorithm, developed by Schijf (1993), denoted by *DAG*, and
3. an (incorrect) DCG algorithm, developed by Schijf *et al.* (1994), denoted by *DCG*.

[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*.

In all implementations, the move ordering is identical. The test set of 117 positions is given in Section 3.4 (see Appendix D). All four algorithms searched for a maximum of 500,000 nodes per test position. After 500,000 nodes the search was terminated and if no solution had been found the problem was marked as not solved[12].

5.7 Results

To verify our solution we have first tested the position given in Figure 5.1[13]. *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 the occurrence of the GHI problem has been correctly handled. *BTA* shows the benefit of being a DCG algorithm, as evidenced by the decrease in number of nodes investigated by a factor of roughly 40 as compared to *Tree*.

	# of pos. solved (out of 117)	Total nodes (96 positions)
Tree	99	4,903,374
DAG	102	3,222,234
DCG	103	2,482,829
BTA	107	2,844,024

Table 5.1: Comparing four pn-search variants.

Thereafter, we have performed the experiments as described in Section 5.6. The outcomes are summarized in Table 5.1. The complete results are listed in Appendix G. The first column 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. *BTA* solves each position which was solved by at least one of the other three 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.

Tree: K208, K215, R281.

DAG: K208, K215, K216, R168, R182, R281.

DCG: K44, K60, K217, K284, R168, R182, R252.

[12] The maximum number of nodes in these pn-search experiments is lower than the corresponding number given in Chapter 3 due to implementation details.

[13] 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.

BTA: K44, K60, K208, K215, K216, K217, K284, R168, R182, R252, R281.

Neither algorithm: K8, K40, K78, K195, K209, K210, K220, R96, R105, R201.

Obviously, *Tree* investigates the largest number of nodes. The explanation is easy: the algorithm does not recognize transpositions. Further, *DCG* examines the smallest number of nodes: this algorithm sometimes prematurely disproves positions; hence, on the average fewer 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 remarked by Schijf *et al.* (1994).

From Table 5.1 it further follows that *BTA* performs best. 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 drawback 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 positions the reduction in the number of nodes searched is even larger than in "normal" positions.

As a case in point we present Figure 5.18 corresponding with Diagram 216 in Krabbé (1985). It is solved by our BTA algorithm (in 247,686 nodes) and by the DAG algorithm (in 366,336 nodes) and not by the two other algorithms (within 500,000 nodes). Many transpositions (and many repetitions of positions) exist, since after **1. ♖a5+ ♔b8** White has a so-called *zwickmühle* and can position the Bishop anywhere along the a7–g1 diagonal for free. For instance, after **2. ♗a7+ ♔a8 3. ♗b6+ ♔b8** almost the same position with the same player to move has been reached: the Bishop has moved from d4 to b6. At any time White can choose such a manoeuvre. For the chess-playing reader, the solution is **1. ♖a5+! ♔b8 2. ♗a7+ ♔a8 3. ♗c5+! ♔b8 4. ♖b5+ ♔a8 5. ♖e7! ♗f7 6. ♖a5+ ♔b8 7. ♗a7+ ♔a8 8. ♗d4+! ♔b8 9. ♖b5+ ♔a8 10. ♖d7! ♕g5 11. ♖a5+ ♔b8 12. ♗a7+ ♔a8 13. ♗b6+ ♔b8 14. ♗xc7 mate**.

5.8 Chapter conclusions

In this chapter we have given a solution to the GHI problem, resulting in an improved DCG algorithm for pn search, called BTA (Base-Twin Algorithm). It is shown that the restricted version (a4–e8 board) of 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 positions do not falsify our claim. Despite the additional overhead for recognizing positions suffering from the GHI problem, our BTA algorithm is hardly less efficient than other, not entirely correct DCG algorithms, and finds more solutions.

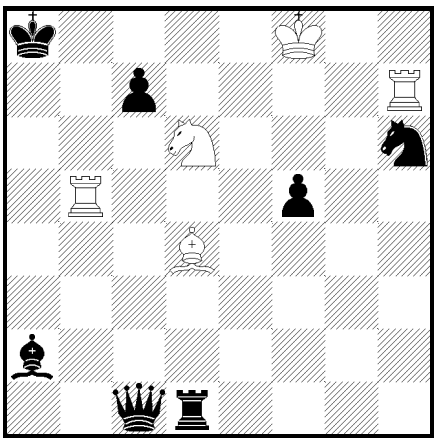

Figure 5.18: Mate in 14 (WTM); (J. Kriheli).

Although 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 to be solved. The importance of this statement is that with the increasing availability of computer memory a growing tendency exists to use best-first search algorithms and variants thereof, or best-first fixed-depth algorithms (Plaat *et al.*, 1996), which no longer suffer from the GHI problem.

Our solution to the GHI problem gives an affirmative answer to the third problem statement: is it possible to give a solution for the GHI problem for best-first search? By transforming the search tree into our DCG representation, less memory is needed, since only the roots of equal subtrees are duplicated. Moreover, less search is needed, since the DCG contains fewer nodes than the tree. One disadvantage is the cost of finding a most-proving node. If many transpositions exist in the tree, many *possible draw*s will occur, prolonging the search for a most-proving node. We are convinced that the advantage of solving the GHI problem outweighs this disadvantage. 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. However, Campbell (1985) already noted that in depth-first search the frequency of GHI problems is considerably smaller than in best-first search. The solution of the GHI problem for depth-first search therefore seems to be of minor importance for practical use.