Proof-Number Search and Transpositions

Martin Schijf
University of Leiden

August 1993

Contents

Preface

This thesis gives a description of proof-number search in combination with transpositions. I would like to thank a number of people who helped me with the many problems that arose during the research.

In the first place I'd like to thank my supervisors Ida Sprinkhuizen-Kuyper (University of Leiden), Victor Allis and Jos Uiterwijk (both of the University of Limburg, Maastricht) for their support during the research and the time they have spent in correcting this thesis.

Special thanks goes to Dennis Breuker who accommodated me during my visits to the University of Limburg.

Chapter 1

Introduction

Proof-number search (Allis *et al.*, 1991; Allis *et al.*, 1994), a search method derived from conspiracy-number search (McAllester, 1988) is a recent search method that has been developed while determining the game-theoretical value of Connect Four, Qubic and Go-Moku. It has also been applied to Awari and Othello. The present research initially dealt with the application of proof-number search to Nine Men's Morris. It then became clear that many equal positions were analysed more than once by proof-number search. This could be prevented by making use of transposition techniques. These techniques were not directly applicable to Nine Men's Morris, because in this game it is possible to repeat positions.

To understand and find a solution to this problem it is necessary to have a close look at the basic principles of proof-number search. It turned out that the problems existed not only in the 'repetition case', but also if proof-number search makes use of transpositions encountered via different move sequences (even for games without repetition of positions).

This thesis describes how proof-number search can be applied to three different search 'structures'. The first one is the search tree. In this structure no attention is paid to transpositions. Chapter 3 explains formally the way proof-number search works in trees. Thereafter transpositions are introduced converting a tree into a directed acyclic graph. The problems of proof-number search in this structure will be described in chapter 4. In chapter 5 proof-number search is described in a directed cyclic graph, a structure in which a transposition can also be a repetition. Some basic terms of game-tree search are first described in chapter 2. The last chapter contains the conclusions about the application of transpositions in proof-number search.

Chapter 2

Search trees

Zermelo (1912) proved that for each chess position (except for end positions) a best move exists. Naturally this holds not only for chess, but for all kinds of *two-person zero-sum games*. These are games in which a win for one player implies an equal loss for the other.

A best move in a position P can be obtained by examining all successors $p_1 \ldots p_n$ of P, the positions that are obtained by playing one move in P. If a p_i is won for the player to move in P then the move leading to p_i is a best move. If no such move exists the move that leads to a draw position is best. If all successors of P are lost then any move is a best move.

A *game tree* can be built to investigate whether a position P is a win, draw or loss for the *player to move* in P. This is done by generating all successors p_i of P and creating an edge from P to p_i. If a successor is an end position (check mate for instance), the result is determined by the rules of the game. This result can be represented by values -1, 0, 1 for a lost, drawn or won position, respectively. In some applications only two values, -1 and 1, are used, representing a non-win (loss or draw) and a win, respectively.

For each successor p_i that is not an end position, all successors of p_i are generated (by considering the *opponent*'s moves in p_i) and so on, until each position in the tree has been expanded (e.g the successors are already generated) or is an end position. The so created game tree contains three types of positions:

1. positions in which *player* is to move.

2. position in which *opponent* is to move.

3. end positions.

The *game-theoretical value* (the value that is obtained by optimal play of both *player* and *opponent*) of each internal position in the tree can now be computed with the aid of the values of its successors (termed children). Because all values will be computed with respect to the player to move in the position represented by the root of the tree (*player*), the value of any position of the first type is equal to the highest value (the maximum) of its children. If, on the other hand, the opponent is to move in a position (the second type), the value of that position is equal to the lowest value of its children (the minimum), because a low value with respect to *player* means a high value with respect to *opponent*. This is an informal description of the *Minimax* procedure (Neumann, 1928). Formally, the **g**ame **t**heoretical **v**alue of any

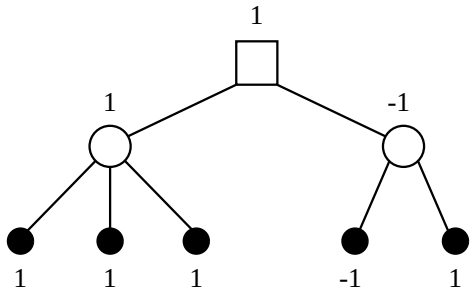

Figure 2.1: A game tree with game-theoretical values.

position P is given by:

$$gtv(P) = \begin{cases} Max_{p_i \in children(P)}(gtv(p_i)) & \text{, if } \textit{player} \text{ to move in } P \\ Min_{p_i \in children(P)}(gtv(p_i)) & \text{, if } \textit{opponent} \text{ to move in } P \\ -1 & \text{, if } P \text{ is an end position which is lost for } \textit{player} \\ 0 & \text{, if } P \text{ is an end position which is drawn for } \textit{player} \\ 1 & \text{, if } P \text{ is an end position which is won for } \textit{player} \end{cases}$$

Figure 2.1 gives an example of a game tree with game-theoretical values. In this tree positions in which *player* is to move are denoted by squares. Because their values are obtained by maximizing the values of their children, these nodes are called *max nodes*. The circles represent the positions in which *opponent* is to move. These nodes are similarly called *min nodes*. End positions are represented here by dots. Because of the type of nodes, player is also called *max player* and opponent *min player*.

A partial game tree is called a *search tree*. In a search tree, it is possible that not all nodes have been expanded. Such a not expanded node is called a terminal. The terminals are not expanded, because the search to the determination of the game-theoretical value is not finished yet and to obtain this value some nodes still have to be expanded (the values of these terminals are still unknown), or because it is not necessary for the determination of the game-theoretical value (the not expanded part of the game tree is then called 'pruned'). In figure 2.1, for instance, it suffices to determine three terminals (the left three) to prove that the value of the root is 1.

Many search techniques that try to minimize the number of nodes visited in the game tree have been developed, of which the α-β technique (Knuth and Moore, 1975) is the most widely-used. One alternative technique is proof-number search and will be explained in the next chapter.

Chapter 3

Trees and proof-number search

3.1 Informal description

Proof-number search or pn-search is a game-tree search method to determine the game-theoretical value of a given position. It is based on a two-valued evaluation, so each node in the tree can have one of the possible values a 'win' or a 'loss' (or still be 'unknown'). As long as the value of the root in the tree is unknown we have to get more information by expanding or evaluating terminals in the search tree. It may not be necessary to determine the values of all terminals, so the problem is which terminal to examine first to determine the game-theoretical value of the root as quickly as possible. Figure 3.1 shows a search tree. Each node in the tree has an unknown value. We want to determine the game-theoretical value of the root, node a. Because this game-theoretical value is a 'win' or a 'loss' we will seperately examine both possibilities.

To prove that the value of node a is a win, it is enough to prove that at least the value of one of its children is a win, because a is a max node. In chapter 2 was described that the player at a max node (the max player) always chooses the move that leads to a position with a win value (if there exists one), because this position has the highest value and the value in a max node is obtained by maximizing the values of its children. In other words if the value of a child of a max node is a win then the value of the max node is a win. So it can be concluded for figure 3.1 that the value of node a is a win if the value of node b or node c is. The player at a min node (the min player) will choose the move leading to a position that is

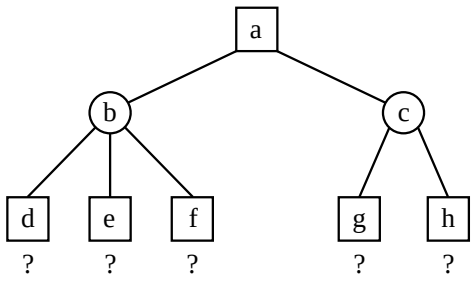

Figure 3.1: A search tree.

lost for the max player. If each possible move from this position leads to a won position, the min player cannot choose a move to a lost position. So to prove that the value of a min node is a win the value of all its children must be proven to be a win. To prove that the value of node b is a win, the value of all terminals d,e, and f must thus be proven to be a win. The value of node c is a win if both the values of terminals g and h are a win. It is clear now that to prove the value of node a to be a win, each terminal in $\{d,e,f\}$ or each terminal in $\{g,h\}$ must evaluate to a win.

Now let us look at what we can conclude if we want to prove that node a is a loss. To prove that node a is a loss the value of both children b and c must be a loss. The value of node b is a loss if at least the value of one of its children is. The same holds for node c. So the value of node a is a loss if at least one terminal in the set $\{d,e,f\}$ *and* at least one terminal in the set $\{g,h\}$ evaluate to a loss. Note that there are six combinations of terminals matching this condition, being $\{d,g\}$, $\{d,h\}$, $\{e,g\}$, $\{e,h\}$, $\{f,g\}$ and $\{f,h\}$.

What terminal do we have to choose for further examination to get closer to the proof of the game-theoretical value of the root?

To prove that the value of a is a win, each terminal in $\{d,e,f\}$ or in $\{g,h\}$ must evaluate to a win. Suppose that the chance of evaluating to a win is equal for each terminal. If each terminal would evaluate to a win then the least effort to prove the value of the root is a win is done by proving that the value of each terminal in the set with the *least* number of terminals is a win. So examining the terminals in the smallest set would lead quickest to the proof that the value of the root is a win, assuming that all terminals will evaluate to a win. Therefore examining one terminal in this smallest set (in the figure terminal g or h) would probably contribute most to a 'win proof'.

To prove that the value of a is a loss, each terminal in one of the six sets $\{d,g\}$ ... $\{f,h\}$ must evaluate to a loss. Suppose that each terminal has an equal chance of evaluating to a loss. The least effort to the 'loss proof' of the root is done by proving the loss value of the terminals in the smallest set of the six sets. Because each set contains two terminals no set has a preference above an other. Therefore examining any terminal in any of the six sets would contribute most to the 'loss proof'.

To prove as fast as possible the root's real value we are looking for a terminal that satisfies:

1. a quickest way to prove that the value of the root is a win, assuming that the chosen terminal would evaluate to a win (this is a terminal in a smallest set of terminals that are needed to prove the win value of the root).

2. a quickest way to prove the value of the root is a loss, assuming that the chosen terminal would evaluate to a loss (this is a terminal in a smallest set of terminals that are needed to prove the loss value of the root).

A terminal that satisfies both conditions is a terminal that is in a smallest set of terminals that proves the root's value is a win and in a smallest set of terminals that proves the opposite for the root's value. In the figure terminals g and h satisfy the first condition and all terminals satisfy the second condition. So terminal g or h is the terminal that satisfies both conditions (note that each of the six sets to the loss proof contains either g or h). A terminal that satisfies both conditions is called a *most-proving node*.

A most-proving node thus has to be examined to provide more information about the present search tree. This examination of a node consists of two actions. In the first place we evaluate the node. If by this the value of the node becomes known as a win or a loss, we

stop further examination; more useful information cannot be obtained for this node. If the evaluation does not definitely answer whether the node represents a won or lost position, it has to be expanded: we generate all children of this node and add them to the tree. If the time complexity of evaluating a node is low compared to expanding the node, it is better to expand the most proving node, followed by an evaluation of all generated children. If after evaluating and/or expanding the most-proving node the root's value is proven, the search is finished. Otherwise the search has to continue, selecting another terminal in the expanded search tree among the most-proving nodes.

A few questions still remain open. Does there always exist a node that is in both the smallest set to prove the root is a winning node, and in the smallest set to prove the opposite? In other words, does a most-proving node always exists? And, if so, how can this terminal be determined? The (positive) answer to the first question will be delayed to section 4.2. Section 3.2 contains some definitions that formally describe how the most-proving node can be determined and in section 3.3 the algorithms to determine this node are described.

3.2 Definitions

To explain formally how proof-number search works, we need some definitions.

A proof set of a node V is a set of terminals such, that if for each terminal in this set it is proven that its value is a win then the value of V is a win.

A minimal proof set of a node V is a proof set S such, that for any terminal $t \in S$: $(S \setminus t)$ ('S without t') is not a proof set for V.

The complete proof set of a node V is the set of *all* minimal proof sets of V.

A disproof set of a node V is a set of terminals such, that if for each terminal in this set it is proven that its value is a loss then the value of V is a loss (to prove that a value is a loss is the same as to *dis*prove that a value is a win, because only two values, win and loss are used).

A minimal disproof set of a node V is a disproof set S such, that for any terminal $t \in S$: $(S \setminus t)$ is not a disproof set for V.

The complete disproof set of a node V is the set of *all* minimal disproof sets of V.

To compute the complete (dis)proof sets a new set operation, the *Tensor Union*, denoted as $\uplus$ is needed:

$$\uplus : \text{set of sets} \rightarrow \text{set of sets}$$

$$A \uplus B = \bigcup_{a \in A, b \in B} \{a \cup b\}$$

For example $A = \{\{a, b\}, \{c\}\}$ and $B = \{\{b, c\}, \{d\}\}$.
Then $A \uplus B = \{\{a, b, c\}, \{a, b, d\}, \{b, c\}, \{c, d\}\}$. By definition if A or B is $\emptyset$ then $A \uplus B = \emptyset$, too.

Now suppose some node V is a max node. Each child of V has a number of minimal proof sets (the complete proof set). Because V is a max node to prove that it has a win value, at

least one of its children must be proven to have a win value. This means that any minimal proof set of a child of V is a minimal proof set of V. So to get all minimal proof sets of V the collection of all minimal proof sets of the children of V have to be taken.

To prove that the value of a max node V is a loss, the value of all its children must be proven to be a loss. So the union of a (*one*) minimal disproof set of each child of V is a minimal disproof set of V. To get all minimal disproof sets of V, we have to make all combinations of minimal disproof sets of the children of V or, more formally, we have to take the Tensor-union of the complete disproof sets of its children. If V is a min node the complete (dis)proof sets are analogously computed.

If node V is an unproven terminal, the complete (dis)proof set has only one set of terminals that can prove or disprove a value. This is the set with terminal V itself.

If a terminal V represents a won end position, the value of the terminal *is* a win. To *prove* it has a win value, no more terminals need to be proven to have a win value. So the minimal proof set of this terminal contains no terminals and is equal to the empty set, $\emptyset$, which is the only element in the complete proof set.

To prove this terminal V has a loss value, is impossible. So for this terminal there exists no minimal disproof set. Therefore the complete disproof set has no elements. The computation

if V is an internal node **then**
  if V is a Maxnode **then**
    V.complete-proof-set = $\bigcup_{s \in children(V)} s$.complete-proof-set;
    V.complete-disproof-set = $\biguplus_{s \in children(V)} s$.complete-disproof-set;
  else
    V.complete-proof-set = $\biguplus_{s \in children(V)} s$.complete-proof-set;
    V.complete-disproof-set = $\bigcup_{s \in children(V)} s$.complete-disproof-set;
  fi
else
  if V represents a won position **then**
    V.complete-proof-set = $\{\emptyset\}$;
    V.complete-disproof-set = $\emptyset$;
  else
    if V represents a lost position **then**
      V.complete-proof-set = $\emptyset$;
      V.complete-disproof-set = $\{\emptyset\}$;
    else
      V.complete-proof-set = $\{\{V\}\}$;
      V.complete-disproof-set = $\{\{V\}\}$;
    fi
  fi
fi

Figure 3.2: The definition of complete (dis)proof sets.

of the complete proof sets is given in figure 3.2.

In the previous section it was described that the terminal that contributes most to the proof or disproof of the root's value (most-proving node) is a terminal that is both in a smallest minimal proof set and a smallest minimal disproof set of the root. Because the

complete (dis)proof set of a node V contains all minimal (dis)proof sets of V, it also contains the *smallest* minimal (dis)proof set (the (dis)proof set with the least number of terminals). So the smallest minimal proof set in the complete proof set of the root and the smallest minimal disproof set in the complete disproof set of the root are the only two important sets to determine the most-proving node. Formally for any node V in a search tree these sets are defined as:

$$V.\text{smallest-(dis)proof-set} \in \bigcup\{s^*\} \quad \text{, where} \quad s^* \in V.\text{complete-(dis)proof-set}$$
$$\text{and} \qquad s^* = Min_{s \in V.\text{complete-(dis)proof-set}}|s|$$

If V is the root of a tree then

$$\textit{most-proving node} \in (V.\text{smallest-proof-set} \cap V.\text{smallest-disproof-set})$$

Note that if for a node V the complete proof set $= \emptyset$ that the smallest proof set does not exist. To compute these two important sets for the root, the complete proof set and complete disproof set must be computed first. The complete (dis)proof sets are computed with the aid of the complete (dis)proof sets of the children.
A lot of unnecessarily computation can be avoided if we make use of an obvious (but unproven) tree property:

> Each minimal proof set of a node V is *disjunct* with each minimal proof set of any sibling of V (same holds for minimal disproof sets).

So if for each child of some max node V a smallest proof set and disproof set is known, the smallest proof set and smallest disproof set of V can be computed. The first one is equal to the smallest of the smallest proof sets of its children and the second is equal to the union of the smallest disproof sets of each child (for a min node V the computation goes analogously).

After each expansion of the most-proving node the smallest proof and disproof sets should be computed again to determine a new most-proving node. This computation is time expensive if the smallest (dis)proof sets for each node is not available in memory. On the other hand if for each node both sets would be available, the process would be memory expensive. By only storing the *number* of elements for both the smallest proof and disproof set of each node, the most-proving node still can be determined.

The most proving node t of a tree is in both sets of the root of that tree. The smallest proof set of the root is equal to a smallest proof set of a child (assuming that the root is a max node). So if t is in the smallest proof set of the root it is also in the smallest proof set of that particular child. This child is a root of a subtree and for any tree a most-proving node exists (which will be proven in section 4.2). This most-proving node is, again, in both sets of the root of that subtree. This time the smallest disproof set is equal to a smallest disproof set of its children. This means that the most-proving node is also in the smallest disproof set of that child. Continuing this procedure we finally end up in a terminal, the most-proving node. If only the number of elements are stored, the same path can be followed from the root to the terminal. In a max node V the most-proving node is also the most-proving node of a child that has the same number of elements of the smallest proof set as V. For a min node the same holds with respect to the number of elements in the smallest disproof set.
These numbers are described as:

The proof number of a node is the minimal number of terminals for which the value must be shown to be a win to prove that the node has a win value.

The disproof number of a node is the minimal number of terminals for which the value must be shown to be a loss to prove that the node has a loss value.

Or formally described in terms of complete (dis)proof sets:

$$V.\text{(dis)proof-number} = Min_{s \in V.\text{complete-(dis)proof-set}} |s|$$

By definition the (dis)proof number of a node is ∞ if the complete (dis)proof set of that node is $\emptyset$.

As for any two disjunct sets A and B, $|A \cup B| = |A| + |B|$ and because any minimal proof set of a node V is disjunct with all minimal proof sets of each of its siblings, the proof number of a min node is equal to the summation of the proof numbers of all its children. The same holds for disproof numbers and max nodes. The disproof number of a min node is equal to the smallest disproof number of its children. The proof number of a max node is computed analogously. In figure 3.3 the formal description of the (dis)proof numbers is given, independent of the complete (dis)proof sets.

```
if V is an internal node then
    if V is a Maxnode then
        V.proofnumber = Minimum_{s∈children(V)}(s.proofnumber);
        V.disproofnumber = Σ_{s∈children(V)} s.disproofnumber;
    else
        V.proofnumber = Σ_{s∈children(V)} s.proofnumber;
        V.disproofnumber = Minimum_{s∈children(V)}(s.disproofnumber);
    fi
else
    if V represents a won position then
        V.proofnumber = 0;
        V.disproofnumber = ∞;
    else
        if V represents a lost position then
            V.proofnumber = ∞;
            V.disproofnumber = 0;
        else
            V.proofnumber = 1;
            V.disproofnumber = 1;
        fi
    fi
fi
```

Figure 3.3: The determination of (dis)proof numbers.

Figure 3.4 shows the tree of figure 3.1 with proof- and disproof numbers added, denoted as (proof number, disproof number) at each node. The bold edges represent the path to a most-proving node starting at the root.

The definitions of (dis)proof numbers in this section leads to simple and clear algorithms, wich will be described in the next section.

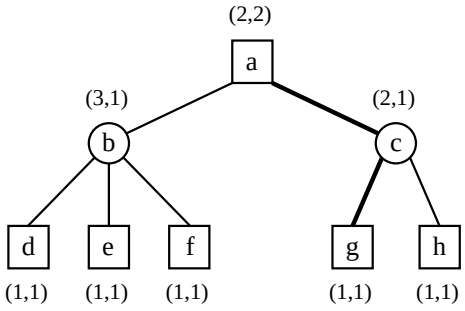

Figure 3.4: A search tree with (dis)proof numbers.

3.3 Algorithms

In this section the pseudo code of the algorithms that will prove the game-theoretical value, e.g. a 'win' or a 'loss', of a given position are described. In the algorithms an abbreviation for proof number, disproof number and most proving node will be used. They will be denoted by *pn, dpn* and *mpn*, respectively.

First of all we have to determine the most-proving node in a search tree. In the previous section it was described that a most-proving node of a tree can be determined by constructing a path from a max node to a child with equal proof number to that of the max node and from a min node to a child with equal disproof number. Continuing this proces until a terminal is reached yields a most-proving node. Sometimes there may be more children with an equal proof or disproof number to that of their parent (if this is the case, the intersection of the smallest proof set with the smallest disproof set of this parent contains more than one terminal, or more than one smallest (dis)proof set exists for the parent). There is no theoretical preference for any one of these terminals. In the algorithm described in this section, the left-most terminal in the current search tree that is *a* most-proving node is chosen as *the* most-proving node. The algorithm described in figure 3.5, illustrates how to construct the correct path starting from a root of a (sub)tree to the most-proving node of that (sub)tree.

If we have determined the most-proving node, this node should be examined closer. In section 3.1 it is described that this can be done in two ways. The first method is to evaluate the terminal and, if it does not evaluate to a win or loss, to expand the node. The other method expands the terminal, knowing that it is not evaluated to a win or loss, followed by evaluating all the generated children. The algorithm described in figure 3.6 shows the pseudo code for the second method.

After the expansion of the most-proving node the search tree has changed, so the (dis)proof numbers of the nodes have to be recomputed. The only nodes that can get other (dis)proof numbers than they had before the expansion, are the nodes on the 'expansion path', the path from the root to the most-proving node. So these nodes are also the only nodes for which the (dis)proof numbers have to be recomputed. In figure 3.7 a recursive procedure that handles this recomputation is described.

Now that the tree has its updated values, we need a new most-proving node, which we have to expand, etcetera. The main *pn-search* procedure is described in figure 3.8.

```
function determine-most-proving-node (V : node);
begin
    while V is an internal-node do
        if Maxnode then
            V := leftmost-child-with-equal-proofnumber (V);
        else
            V := leftmost-child-with-equal-disproofnumber (V);
        fi
    od
    return V;
end
```

Figure 3.5: Determining most-proving node.

```
procedure expand (V : node);
begin
    Children(V) := Generate-Children(V);
    for each s ∈ Children(V) do
        Create-Node (s);
        Make-Edge (V, s);
        value := Evalution(s);
        if value is a 'win' then
            s.pn = 0;
            s.dpn = ∞;
        else
            if value is a 'not win' then
                s.pn = ∞;
                s.dpn = 0;
            else
                s.pn = 1;
                s.dpn = 1;
            fi
        fi
    od
end
```

Figure 3.6: Expanding a node.

```
procedure update (V : node);
begin
    if V is a Maxnode then
        V.pn := Minimum_{s∈Children(V)}(s.pn);
        V.dpn := Σ_{s∈Children(V)}(s.dpn);
    else
        V.pn := Σ_{s∈Children(V)}(s.pn);
        V.dpn := Minimum_{s∈Children(V)}(s.dpn);
    fi
    if V has a parent then
        update(parent(V));
    fi
end
```

Figure 3.7: Updating the (dis)proof numbers.

```
function pn-search (position);
begin
    root := Create-Root (position);
    while (root.pn ≠ 0) and (root.dpn ≠ 0) do
        mpn := determine-most-proving-node (root);
        expand (mpn);
        update (mpn);
    od
    if root.pn = 0 then
        return 'win'
    else
        return 'not win'
    fi
end
```

Figure 3.8: pn-search.

3.4 Enhancements, (dis)advantages and speed-up

This last section of this chapter is partitioned in five paragraphs. The first paragraph mentions some areas in which pn-search has been applied. The second one explains some advantages and disadvantages of this search method and thereafter the memory problem as well as some techniques to deal with it are described. Enhancements on pn-search are given in the fourth paragraph. This section will end with a nice property that can be used to speed up the execution time.

Applications Pn-search has been developed to determine the game-theoretical value of initial positions (solving games) and other game positions (post-mortem analysis). It has also its applications in tournament programs.

Starting with initial game positions, pn-search was succesful for the games Connect Four (Uiterwijk *et al.*, 1990) and Qubic (Allis and Schoo, 1992). To solve these games, powerful evaluation functions were used. Go-Moku has been solved with the aid of *threat-space search* as a very strong evaluation function for pn-search (Allis and Van den Herik, 1992).

In determining the game-theoretical value of other game positions, pn-search outperforms the well known α-β variants on sufficiently non-uniform trees which has been shown for the game of Awari (Allis *et al.*, 1991; Allis *et al.*, 1994)).

For tournament play, a multivalued version of pn-search can be used. Suppose a game has an evaluation function that produces values in the range $0 \ldots 31$. First we prove whether the value is smaller than 16 or not. Now this is a two-valued process (with values *smaller-than-16* resp. *greater-than-or-equal-to-16*), so pn-search can be used. If the answer is *greater-than-or-equal-to-16* then we use pn-search again to prove whether the value is smaller than 24 or not. If the answer is *smaller-than-16* pn-search is used to prove the value is smaller than 8 or not. Continuing this binary proces, the real value will be proven in at most five steps.

Advantages and disadvantages The question whether pn-search outperforms α-β algorithms in solving positions, depends on the complete form of the game tree. If it is a uniform tree, like Tic-tac-toe starting with the empty board, pn-search does worse than α-β search. In the worst case it even simulates *Minimax* (in fact the first five levels of the Tic-tac-toe tree built by pn-search are equal to the tree generated by Minimax!). But if the tree is not uniform and especially if there are forced moves to be played, pn-search shows its strength. A nice example of this has been found in an implemenation of *Give-away-chess*[1](Allis, 1994). After white has played **1.d2-d4, 1.d2-d3 or 1.e2-e4** in the initial position, pn-search showed that the position was a loss for white in less than 10 seconds. So a winning variant for black with at least 32 plies had been found.

Memory problem Because pn-search is a best-first-search technique the complete game tree that has been built must be kept in main memory. This sometimes may cause memory-exceed problems. To reduce the size of the tree, several techniques are known. A subtree can be deleted if the root of this subtree has been proven (e.g. proof or disproof number of this node is equal to zero). This reduction helps, but does not always solve the problem.

[1]In this chess variant a player *must* capture an opponent's piece if possible and has a free choice otherwise. The first player that cannot make a move wins the game.

If the main memory is 'full of nodes' but the search is not finished yet, some parts of the present search tree must be deleted to free some memory and continue the search. In (Allis *et al.*, 1991; Allis *et al.*, 1994) a technique called *delete-least-proving-nodes* is described. It is unlikely that such a least proving node will be needed soon in the further search. By deleting a couple of these nodes, some memory becomes available and the search can continue.

Some experiments with deleting complete subtrees are done by Gnodde for the Othello game (Gnodde, 1993).

Enhancements Gnodde also described the improvements that are made by initializing the (dis)proof number in the nodes generated after an expansion other than the standard *proof number = disproof number = 1*. This different initialization is based on a heuristic guess of the number of nodes that probably will be needed to (dis)prove the value of the node generated. These heuristic proof and disproof numbers are based on specific game characteristics. For example, if a position seems to be almost won then its proof number (the number of terminals to prove that it is a win or, informally, the amount of work that has to be done) is initialized with a lower number than in positions where the win seems further away. In section 3.2 pn-search was described assuming that the chance for each terminal to be a win or loss is equal for each terminal. With these heuristic initializations, terminals are attributed a bigger or smaller chance to be a win or a loss. In his thesis, Gnodde compares pn-search using heuristic initialization and pn-search with the standard initialization. He shows that the number of nodes needed to solve some Othello positions is about a factor 20 less in favour of the heuristic technique.

Another form of initializing the proof or disproof number, is using a value that it would have had if the node had been expanded. So if the node is a max node, the disproof number is equal to the number of moves the max player can make and the proof number is still equal to 1. If the node is a min node the proof number will be initialized with the number of moves the min player can make and the disproof number is set to 1. Using this (non-heuristic) initialization the (dis)proof number of each terminal is determined one ply ahead, so the information that is now available in the tree is much more than the information there would be if the (dis)proof numbers of the terminals were initialized with (1,1). This technique is used in finding check-mate variants of given chess positions (Breuker *et al.*, 1994).

Speed-up The execution time of the *update procedure* described in section 3.3 can be sped up by using a property of pn-search and trees. This property yields that (dis)proof numbers of a node are based on the (dis)proof numbers of their children. Knowing this the update procedure will be more efficient if updating the (dis)proof numbers of the nodes on the expansion path is quitted as soon as both proof and disproof number of a node have not been changed. Because the values of the siblings of this node have not changed either, the values of the parent remain the same as well, as will the values of all other predecessors.

This property however is a specific *tree* property of proof-number search. When pn-search is applied to trees with *transpositions*, which are no longer trees, but graphs, this property does not hold. The next chapters will discuss this problem.

Chapter 4

DAGs and proof-number search

Transpositions are used to prevent doing the same work twice in a search tree. In α-β search transpositions are implemented by making use of *hash tables*, also called *transposition tables*. If for a position its value has been computed, this position is stored in the hash table, along with some additional information, like the depth of the position in the tree and whether the value has been determined with α-β cutoffs. When at another place in the α-β tree this position is encountered again, the value in the hash table can be used instead of recalculating the value.

If transpositions are used in best-first-search techniques, in which the total search tree is available in memory, the structure is no longer a tree, but a *graph*. There are two types of these search graphs created by two types of transpositions.

The first type, known as **D***irected* **A***cyclic* **G***raph* (DAG), is created by games that have strictly converting positions. This means that from some position after each move it is impossible to reach that position again by any legal sequence of moves. Tic-tac-toe, Connect Four, Go-Moku and Qubic are games having this property.

The second type of game graphs are the **D***irected* **C***yclic* **G***raph*s (DCGs). Games that create DCGs are games that can have repetition of positions, like Chess and Nine Men's Morris. The DCGs will be discussed in chapter 5.

This chapter is partitioned in six sections. First the problem of DAGs in combination with proof-number search will be explained. Thereafter will be proven that the most-proving node as defined in section 3.2 exists in DAGs. The third and fourth sections contain algorithms that determine a most-proving node, distinguished into a theoretically correct algorithm (section 4.3) and a practically usable algorithm (section 4.4). In the fifth section results of these algorithms are compared with a tree algorithm that do not make use of transpositions. Finally, some additional problems and enhancements specifically for DAGs are discussed.

4.1 The problem

A transposition is a position that can be reached from another position via different paths. For example, position T in figure 4.1 can be reached from the initial Tic-tac-toe position A via paths $A-B-C-T$ and $A-E-F-T$. If no notice is given to this, two equal subtrees could be generated in a tree with both position T as the root of the subtrees. If the nodes are linked together a lot of effort will be avoided.

When the proof and disproof numbers of pn-search are applied to graphs with transposi-

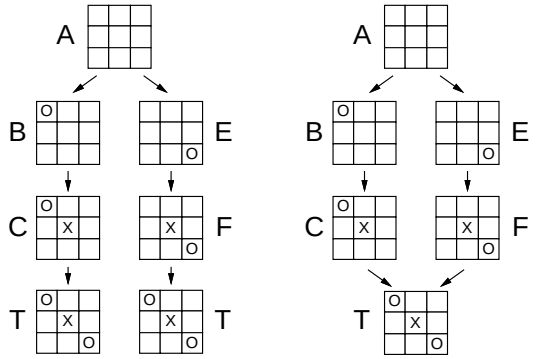

Figure 4.1: Example of a transposition.

tions some problems arise. Suppose node c is the last expanded node in the graph in figure 4.2.

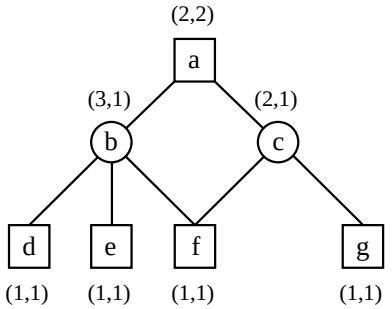

Figure 4.2: Double counted transpositions.

It generated two children f and g, of which f was generated earlier in the search as a child of node b. So node f is reachable via two paths, $a-b-f$ and $a-c-f$, and therefore is a transposition node. After the expansion of node c the graph has to be updated. Doing this with the standard tree algorithms described in section 3.3 give the (dis)proof numbers next to the nodes in figure 4.2. Let's have a closer look at these numbers. The (3,1) at node b is without problem. These numbers were already computed in an earlier iteration before node c was expanded. The (dis)proof numbers at node c are also without problem. To prove the value of node c is a win, at least *two* terminals, f and g, must be proven to have a win value, and only *one*, f or g, needs to evaluate to a loss to prove the value of c is a loss. But the numbers at the root are not so obvious. At the first look they seem to fit: the proof number of a is the minimum of the proof numbers of b and c and the disproof number is equal to the summation of the disproof numbers of its children. The proof number is indeed correct, but what about its disproof number.

Suppose transposition node f evaluates to a loss. Then the value of both b and c will be a loss because they are both min nodes and the player to move in the positions represented by min nodes always chooses a variant that will lead to a lost position, if available. But if *both* values of b and c are losses then, because all children of a now have a loss value, the value

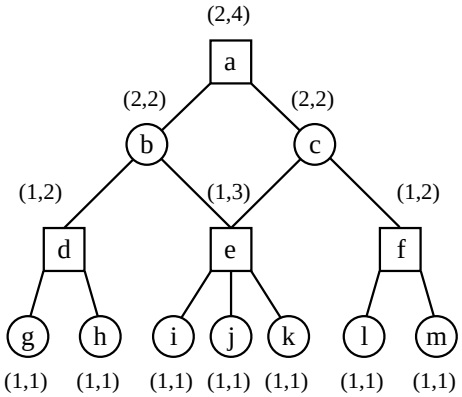

Figure 4.3: Ignoring transpositions.

of node a is a loss. So only the value of *one* terminal (node f) needs to evaluate to a loss to prove that the value of node a is a loss, instead of the 'at least *two*' that the computed disproof number indicated.

To explain this, let's look at the smallest disproof sets of nodes b and c, $\{d, e, f\}$ and $\{f, g\}$, respectively. The smallest disproof set of node a is determined by all possible combinations between the disproof set of its children, here $\{d, f\}$, $\{d, g\}$, $\{e, f\}$, $\{e, g\}$, $\{f, g\}$ and $\{f, f\} = \{f\}$. All combinations, except the last one have two elements. It looks like the last one has also two elements, but because both items are equal it is reduced to one (which is usual for sets) and this was something that was not known when the disproof *number* of a was computed. So the numbers in node a have to be (2,1) instead of (2,2).

Because node f was counted twice the disproof number of a was wrong, but *summing* (dis)proof numbers is not the only problem when the tree algorithms are applied to DAGs. In figure 4.3 node e is a transposition node. The (dis)proof numbers are computed as if it were a tree with node e (including its subtree) as a child of nodes b and c. In this example the disproof number of node e is not counted twice, but it is simply ignored. The disproof number at the root stands for the terminals $\{g, h, l, m\}$. But if terminals i, j and k evaluate to a loss then the value of e is a loss, so will the values of b and c and therefore the value of node a is also a loss. So the least number of nodes needed to prove node a will be a loss is *three* instead of *four*, so node a should have (dis)proof numbers (2,3). This example shows that when (dis)proof numbers are computed by simply minimizing at (min) max nodes a lot of information can disappear.

Computing the correct (dis)proof number at each node in a DAG is one problem. Another problem even appears when the (dis)proof numbers are correct. Suppose node a in figure 4.3 has the correct values (2,3) instead of (2,4). To determine the most-proving node, a path could be followed, according to the algorithm in section 3.3, from a max node to the left-most child with a proof number equal to its parent and from a min node to the one with equal disproof number. In figure 4.3 this path leads from node a via nodes b and d to terminal g. But earlier it was shown that the smallest disproof set is $\{i, j, k\}$ and the most-proving node must be in the intersection of the smallest proof set with the smallest disproof set of the root. Because $g \notin \{i, j, k\}$ it is also not in the intersection of the two smallest sets.

The definitions given in section 3.2 are still correct for DAGs, but the examples in this section showed that the algorithms as described in section 3.3 to determine the most-proving node in a DAG are incorrect. Section 4.3 describes algorithms that correctly determines the most-proving node in a DAG. But first it will be proven that this most-proving node really does exist in a DAG.

4.2 Most-proving node exists

The examples in the previous section showed that the computation of the smallest (dis)proof sets is not as easy for DAGs as it is for trees. To compute the smallest (dis)proof sets in trees we used the *tree* property that all smallest (dis)proof sets of any two sibling nodes are disjunct. So only one of all possible smallest (dis)proof sets for each child was needed to compute the smallest (dis)proof sets of the parent, because the union of two disjunct smallest (dis)proof sets always will contain the same number of terminals as the union of any other two smallest (dis)proof sets. Because it is possible that in DAGs the minimal (dis)proof sets of some nodes are no longer disjunct with some minimal (dis)proof sets of their siblings (for example node b and its sibling c in figure 4.2 both have $\{f\}$ as a minimal disproof set) this property does not hold anymore.

The definitions in section 3.2 showed that there are two methods to compute the smallest (dis)proof sets for each node. The first method computed the complete (dis)proof set for a node and determined the smallest (dis)proof set by comparing all minimal (dis)proof sets in it. The second one computed the smallest (dis)proof set of a node by looking only at the smallest (dis)proof sets of the children of that node. The last method is only correct for trees. But the first is still correct for DAGs, because *all* minimal (dis)proof sets are computed for each node without making use of the *tree* property that two (dis)proof sets are disjunct for two siblings.

The most-proving node is a terminal that is both in the smallest disproof set and smallest proof set. Theorem 1 says that there always exists such a terminal in an unproven DAG (note that by the term *unproven* DAG is meant a DAG of which the game-theoretical value of the root is not established as yet).

Theorem 1 *Any unproven search DAG has a most-proving node.*

To prove theorem 1 we have to prove that the intersection of the smallest proof set with the smallest disproof set is not empty. This will be done by proving that the intersection of *any* minimal proof set in the complete proof set with *any* minimal disproof set in the complete disproof set for each unproven node V in the unproven search DAG is not empty.

The proof proceeds by induction:

Proof :

base

The unproven DAG contains only one node V. Node V is a terminal. By definition, the complete (dis)proof set contains only one set, $\{V\}$. So the intersection of 'any' minimal proof set with 'any' minimal disproof set of V is:
$\{V\} \cap \{V\} \neq \emptyset$.

induction hypothesis

Assume that
The intersection of minimal proof set with minimal disproof set for each unproven node in the unproven DAGs $d_1 \ldots d_n$ $(n \geq 1)$ *is not empty.*

induction step

Let V be a node such that V is not in d_i $(1 \leq i \leq n)$ and that each d_i contains at least one unproven child of V. Suppose V is a max node. To compute the complete disproof set of V the Tensor union has to be taken of all the children's complete disproof sets. Due to the Tensor union two important relations between parent V and each child, say s_i, exists:
Tensor properties:

- *For each* $ds \in s_i$*.complete-disproof-set a* $vds \in V$*.complete-disproof-set exists, such that* $ds \subseteq vds$.
- *For each* $vds \in V$*.complete-disproof-set a* $ds \in s_i$*.complete-disproof-set exists, such that* $ds \subseteq vds$.

The intersection of any $ps \in s_i$.complete-proof-set with any $ds \in s_i$.complete-disproof-set is not empty (due to the induction hypothesis).
So the intersection of any $ps \in s_i$.complete-proof-set with any $vds \in V$.complete-disproof-set is not empty (due to the Tensor properties).
This is true for each child s_i of V and because V.complete-proof-set is the union of the complete proof sets of all children of V, the intersection of any $vps \in V$.complete-proof-set with any $vds \in V$.complete-disproof-set is not empty.
(For a min node, the proof goes analogously)

□

So the most-proving node exists in any search DAG (since each tree is a special DAG this also means that a most-proving node exists in any search tree). The next section shows some problems as well as some algorithms to determine this most-proving node.

4.3 Theoretical algorithm

Only (dis)proof numbers are used to determine the most-proving nodes in trees. Because for any two siblings in a tree the smallest (dis)proof sets are mutually disjunct, the number of elements in these sets (the (dis)proof numbers) gives enough information to determine

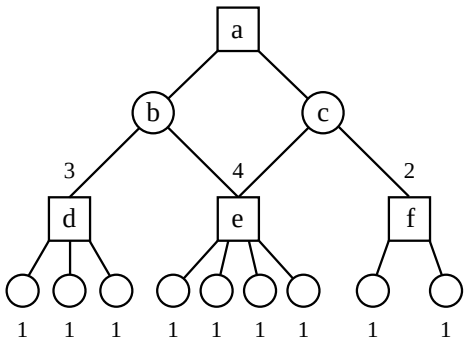

Figure 4.4: A DAG with some disproof values.

the most-proving node. Sections 4.1 and 4.2 showed that this property does not hold for DAGs. The proof of theorem 1 showed that all combinations of terminals have to be stored to determine the (dis)proof number and find the most-proving node. This may require too much memory, so some less memory-intensive algorithm is needed.

In figure 4.4 we only determine the disproof numbers (the proof numbers are determined analogously). Nodes d, e and f have straightforward disproof numbers. The disproof number of node c is simply equal to the minimum disproof number of its children (here the disproof number of f), but because node e is a transposition node the value of e can exert its influence through more than one path to the root. All these 'transposition effective paths' will meet each other in a so-called *join node*. At this node the influence of the transposition comes evident (in figure 4.4 node a is a join node: from transposition e two paths e-b-a and e-c-a join in node a). So besides the minimal disproof number of node f some additional information about transposition node e has to be stored in c. Note that the value of node c is a loss if and only if the value of node e *or* node f is a loss, so the disproof value of node c is $<$**2** $\vee$ **value-of(e)**$>$. Node b derives analogously $<$**3** $\vee$ **value-of(e)**$>$. Node a is proven to be a loss if the values of node b *and* node c both are losses. So the disproof value of node a is :

$$<\textbf{3} \vee \textbf{value-of(e)}> \wedge <\textbf{2} \vee \textbf{value-of(e)}>$$

This is equal to the $2 * 2$ combinations :

$$<\textbf{2+3} \vee \textbf{2+value-of(e)} \vee \textbf{3+value-of(e)} \vee \textbf{value-of(e) + value-of(e)}>.$$

The first three combinations of the disproof value are summations of disjunct terminals but the last one is not. Value-of(e) + value-of(e) implies a double count of the disproof number of node e. This can now be prevented by removing all double 'value-of(x)'s in a term of the disproof information. So the last term has to be replaced by **0 + value-of(e)**. Now the last three terms can be simplified to one term without loosing transposition information because $(0 + x) < (2 + x) < (3 + x)$ and we are only interested in the smallest value. The disproof value of node a thus is :

$$<\textbf{5 ór value-of(e)}>.$$

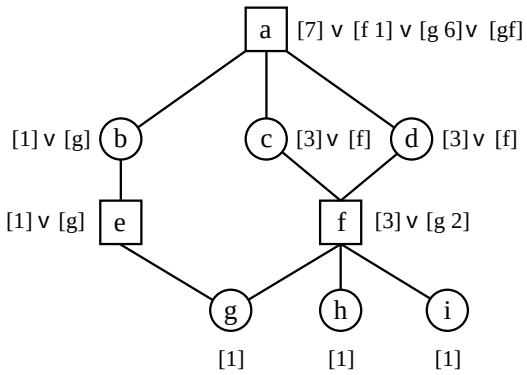

Figure 4.5: A DAG with two transpositions on one path.

The disproof *number* is equal to the smallest value of the disproof value. Since value-of(e) is 4 (stored in node e) the disproof number of a is equal to minimum(4,5) = 4.

Generally, to determine the (dis)proof values of a node all combinations have to be stored in which transpositions can have their effect. This is done by computing a *disjunctive normal form* (dnf) for both the proof and the disproof of a node. This *(dis)proof-dnf* has the form :

$$\text{(dis)proof-dnf} = \text{clause}_1 \vee \text{clause}_2 \vee \ldots \vee \text{clause}_n.$$

in which each clause has two parts: a (possibly empty) transposition set and an integer representing a number of terminals. $[a \wedge b \wedge c\ 13]$ abbreviated to $[abc\ 13]$ means that the value of this clause is equal to the summed values of transposition nodes a, b and c plus 13 terminals. In figure 4.4 the disproof-dnf of node a is $([\emptyset\ 5] \vee [e\ 0])$ or short $([5] \vee [e])$.

To compute the proof-dnf of a max node all the proof-dnfs of its non-transposition children have to be 'or'ed and the result simplified (by removing unnecessary clauses). If a child s is a transposition, not the child's dnf but only the clause $[s]$ is 'or'ed.

The disproof-dnf of a max node is equal to the *'and'* of the disproof-dnfs of his children's (with the same exception if a child s is a transposition).

By definition the first clause of each (dis)proof-dnf has an empty transposition set and the integer represents the correct (dis)proof number. In figure 4.4 the disproof number of node a is equal to 4. This means that the disproof-dnf of node a becomes: $([4] \vee [5] \vee [e]) = ([4] \vee [e])$.

To determine the correct (dis)proof number of node a in the first example we simply filled in the disproof number of node e. Figure 4.5 showed that it is not always that easy. Nodes g, h and i are terminals and each have disproof-dnf [1] (again only the disproof will be determined). Because g is a transposition the disproof-dnf of node e is $([1] \vee [g])$. The disproof-dnf of node f is $([3] \vee [g\ 2])$. Because f is also a transposition the disproof-dnf of nodes c and d is $([3] \vee [f])$. The disproof-dnf of b is equal to the disproof-dnf of its only child e. The disproof-dnf of node a is the 'and' of the disproof-dnfs of all its children :

$$\begin{aligned}
&([1] \vee [g]) \wedge ([3] \vee [f]) \wedge ([3] \vee [f]) = \\
&([7] \vee [f\ 4] \vee [f\ 4] \vee [ff\ 1] \vee [g\ 6] \vee [gf\ 3] \vee [gf\ 3] \vee [gff]) = \\
&([7] \vee [f\ 4] \vee [f\ 1] \vee [g\ 6] \vee [gf\ 3] \vee [gf]) = \\
&([7] \vee [f\ 1] \vee [g\ 6] \vee [gf])
\end{aligned}$$

This is not the correct disproof-dnf yet, because the disproof number must be calculated and stored in the first clause. The correct disproof number is the minimum value of the values represented by each clause, so for each clause its disproof number must be calculated. The computation of the last clause is the most interesting. After substitution of f we get:

$$[gf] = [g \wedge ([3] \vee [g\ 2])] = [g\ 3] \vee [gg\ 2] = [g\ 2].$$

and after substitution of g the value of clause $[gf]$ becomes $1 + 2 = 3$.

Note that the order of substitution is important. If g had substituted earlier than f then the value of this clause would be :

$$[gf] = [[1] \wedge f] = [f\ 1] = [([3] \vee [g\ 2]) \wedge 1] = [4] \vee [g\ 3] = [4] \vee [[1] \wedge 3] = [4]$$

which is incorrect. Generally, if t_1 and t_2 are two transpositions in a clause and the disproof-dnf of t_1 depends on the disproof-dnf of t_2 (t_1 is an ancestor of t_2), then t_1 must be substituted *before* t_2 is substituded. The substitution order can be determined with the aid of *dependency numbers* (dep-nr). Define for each node V :

$$V.\text{dep-nr} = \begin{cases} Max_{c \in children(V)}(c.\text{dep-nr}) + 1 & \text{, if } V \text{ is internal and transposition node} \\ Max_{c \in children(V)}(c.\text{dep-nr}) & \text{, if } V \text{ is internal and not transposition node} \\ 0 & \text{, otherwise} \end{cases}$$

If a disproof-dnf of a transposition node t_1 depends on the disproof-dnf of t_2 then the dependency number of t_1 is greater than the dependency number of t_2. The disproof number of a clause can now be computed by substituting first the transposition with the highest dependency number. The computation of the real proof number of a proof-dnf is described in figures 4.6. The computation of the real disproof number of a disproof-dnf is done analogously. In the description, the transposition set of a clause c is denoted as 'c.transpositionset' and the integer that represents the number of (extra) terminals in a clause c is denoted as 'c.teminalnumber'.

In a max node, the proofDNF is determined by 'or'-ing the children's proof-dnfs. Therefore, the real proof number of a max node is equal to the minimum of the real proof numbers of its children. The same holds for a min node with respect to the disproof number. Figure 4.7 describes the algorithm that updates the dnfs in a DAG after the expansion of a node. In *Normalized-dnf*, the computed temp-(dis)proofDNF is reset to a dnf, without equal clauses and the real (dis)proof number in the first clause. This update procedure must be executed for *all* parents V of the generated children. The initialization of the (dis)proof-dnfs is described in the expand algorithm in figure 4.8.

With the aid of only the correct (dis)proof numbers stored in every node, the most-proving node cannot be determined. In figure 4.4 the disproof number of node a has been determined by the influence of transposition node e. So for node a the most-proving node *could be* in the successors of node e. But a is a max node and in a max node the most-proving node is (as for trees) one of the successors of the child with equal proof number. So the first step of the path to the most-proving node (the most-proving path) is moving to node b. From node b the next step is normally moving to the child with equal *disproof* number, but this time the *disproof* number of *an ancestor on the most-proving path* has been determined by using a transposition node. So the most-proving path has to lead to that direction if possible. This means that the next step of the most-proving path is moving to node e. Finally the leftmost child of e is the most-proving node.

```
function ComputeProofMinimum (dnf : DNF-TYPE,
                               out used-transposet : SET-TYPE);
begin
    w := ∞;
    used-transposet := ∅;
    for each clause c of dnf do
        if c.transpositionset ≠ ∅ then
            t := TranspositionWithHighestDependencyNumber (c.transpositionset);
            new-dnf := Substitute (t.proofDNF for t in c);
            v := ComputeProofMinimum (new-dnf, temp-set);
            if v < w then
                w := v;
                used-transposet := temp-set ∪ {t};
            fi
        else
            w := minimum (c.terminalnumber, w);
        fi
    od
    return w
end
```

Figure 4.6: Computation of the minimum proof number.

Generally, the next step of a most-proving path from a max node V is moving to the child that leads to a transposition that has been used to compute the proof number of a min node earlier on the most-proving path. From a min node the next step moves analogously with respect to the disproof number. So when selecting the next step of the most-proving path, two things have to be known for each node:

1. what transpositions are used to compute the (dis)proof numbers of this node ?
2. is this node on a path to a transposition ?

The first question is answered during the computation of the minimum (dis)proof value of a (dis)proof-dnf (see algorithm in figure 4.6), the 'used transpositions' are stored in the *used-transposet* of the node.

To answer the second question, suppose it is known that the proof number of an ancestor V' of a max node V on the most-proving path has been determined with the use of some transpositions $t_1 \ldots t_n$ (the used-transposet of node V'). Note that the proof-dnf has been computed with the aid of the proof-dnfs of the children. So if a transposition t_i is in a proof-dnf, it must be in the proof-dnf of one of its children, or a child must be t_i itself. This means that if a t_i is in the proof-dnf of a child of V or t_i is a child of V then that child is on a path leading to a transposition that has been used by determining the proof number of V'. If no child on a path leads to a transposition t_i in the used-transposet of V' then the left-most child with equal proof number will be the next step on the most-proving path. The selecting algorithm is described in figure 4.9.

Although these algorithms correctly determine the most-proving node they are not practical. The memory complexity of the (dis)proof dnfs and the time complexity of the *Com-*

```
procedure updateDNF (V : node);
begin
    if V is a Maxnode then
        temp-proofDNF :=
                    ⋁_{s∈children(V)} and ¬transposition(s)(s.proofDNF) ∨
                    ⋁_{s∈children(V)} and transposition(s)([s 0]);
        temp-disproofDNF :=
                    ⋀_{s∈children(V)} and ¬transposition(s)(s.disproofDNF) ∧
                    ⋀_{s∈children(V)} and transposition(s)([s 0]);
        proofnumber := Minimum_{s∈children(V)}terminal-nr-of-first-clause(s.proofDNF);
        disproofnumber := ComputeDisproofMinimum (temp-disproofDNF,used-transpos);
    else
        temp-proofDNF :=
                    ⋀_{s∈children(V)} and ¬transposition(s)(s.proofDNF) ∧
                    ⋀_{s∈children(V)} and transposition(s)([s 0]);
        temp-disproofDNF :=
                    ⋁_{s∈children(V)} and ¬transposition(s)(s.disproofDNF) ∨
                    ⋁_{s∈children(V)} and transposition(s)([s 0]);
        proofnumber := ComputeProofMinimum (temp-proofDNF,used-transpos);
        disproofnumber := Minimum_{s∈children(V)}terminal-nr-of-first-clause(s.disproofDNF);
    fi
    V.proofDNF := Normalized-dnf (temp-proofDNF, proofnumber);
    V.disproofDNF := Normalized-dnf (temp-disproofDNF, disproofnumber);
    V.used-transposet = used-transpos;
    for each f ∈ parents(V) do
        updateDNF (f);
    od
end
```

Figure 4.7: Updating the (dis)proofDNFs.

```
procedure DagExpand (V : node);
begin
    Children(V) := GenerateChildren (V);
    for each s ∈ Children(V) do
        if s already in DAG then
            Make-Edge (V, s);
        else
            Create-Node (s);
            Make-Edge (V, s);
            value := Evaluation (s);
            if value is a 'win' then
                s.proofDNF := [∅ 0];
                s.disproofDNF := [∅ ∞];
            else
                if value is a 'not win' then
                    s.proofDNF := [∅ ∞];
                    s.disproofDNF := [∅ 0];
                else
                    s.proofDNF := [∅ 1];
                    s.disproofDNF := [∅ 1];
                fi
            fi
            s.used-transposet := ∅;
        fi
    od
end
```

Figure 4.8: Expanding a DAG node.

```
function DetermineMpnInDags (V : node);
begin
    proofset := ∅;
    disproofset := ∅;
    while V is an internal node do
        if V is a Maxnode then
            if (disproofset = ∅) then
                disproofset = V.used-transposet;
            fi
            if ChildOnPathToUsedTranpositionExists (V, proofset) then
                V := LeftMostChildToUsedTranposition (V, proofset);
            else
                proofset := ∅;
                V := LeftMostChildWithEqualProofnumber (V);
            fi
        else
            if (proofset = ∅) then
                proofset = V.used-transposet;
            fi
            if ChildOnPathToUsedTranpositionExists (V, disproofset) then
                V := LeftMostChildToUsedTranposition (V, disproofset);
            else
                disproofset := ∅;
                V := LeftMostChildWithEqualDisproofnumber(V);
            fi
        fi
    od
    return V;
end
```

Figure 4.9: Determining the most-proving node in a DAG.

pute(Dis)ProofMinimum procedures are too large. The next section describes some practically usable algorithms that do not have these complexity problems but also do not always find the theoretical correct most-proving node.

4.4 Practical algorithm

The (dis)proof number, by definition the number of terminals in the smallest (dis)proof set, have a nice property for trees: for a node V the (dis)proof numbers are directly obtained by summing or minimizing the (dis)proof numbers of V's children. This property leads to an efficient algorithm, pn-search.

The previous sections showed that this property does not hold for DAGs. This means that the algorithm cannot be used for computing the correct (dis)proof numbers and finding the most-proving node in DAGs. In section 4.3 an algorithm is described that correctly determines the most-proving node, but this algorithm has an unfavourable time and memory complexity. Hence an efficient algorithm is needed that can still handle the transpositions.

Although the tree algorithm is theoretically incorrect for DAGs, it still can be applied to DAGs. The so computed (incorrect) proof and disproof numbers do not represent the number of terminals in the smallest (dis)proof set, but they contain a number that is an upper bound to the real (dis)proof number. Suppose all nodes in a tree are labeled. Equal labels stand for equal positions in the nodes. The difference between a DAG application and a tree application of the algorithm is that after finding the most-proving node the tree application only expands *one* node and updates the path to the root. The DAG application expands *all* nodes with a label equal to the label of the most-proving node. To prevent building big trees all nodes with equal labels are joined into one node thus creating a DAG.

In trees the most-proving node has the property that if it would evaluate to a loss, the disproof number in the root would decrease with one and the same holds for the proof number if the most-proving node would evaluate to a win. If the tree algorithms are used for DAGs this property still holds. In some cases the (dis)proof number would decrease even more. This happens when the (dis)proof numbers of transposition nodes are counted more than once. Figure 4.2 gives an example in which the disproof number would decrease by two if the most-proving node, f, would evaluate to a loss .

So with some small extensions all algorithms in section 3.3 can be used to perform proof-number search in DAGs. The first extension is in the *update* procedure. Instead of recursively updating only *one* parent (last line in the procedure) *all* parents have to be recursively updated. The other extension is in the expand procedure. For each generated child it must be checked whether the position it represents has been generated earlier in the tree. If so, this child is a transposition and instead of creating a new node, only the edge between node V and the already existing transposition node is made. This DAG algorithm has successfully been applied for the games Connect Four, Qubic and Go-Moku by Uiterwijk *et al.* (1990), Allis and Schoo (1992) and Allis and Van den Herik (1992), respectively.

4.5 Results

In this section results of the three algorithms that are described in the previous sections and chapters will be compared. Because one of the three algorithms (the theoretically correct algorithm for DAGs) is slow and uses much memory to execute, a small domain, Tic-tac-toe,

has been selected as test domain. Tic-tac-toe is a game in which no repetitions can occur, so if transpositions occur during the search, the search graph will be a DAG.

Despite the small domain, the theoretical DAG algorithm could not solve the initial Tic-tac-toe position in reasonable time. After more than forty hours of executing the program was terminated abnormally, without determining the value of the root. The algorithm could solve positions with seven empty squares instead of nine, but these results do not give a fair comparison with the other two algorithms, because the number of transposition nodes created during the determination of the value of the root is rather small. Therefore this section does not contain results of the theoretical DAG algorithm, but only those of the tree algorithm and the practical DAG algorithm.

It is well known that the game-theoretical value of the initial position of Tic-tac-toe (the empty board) is a draw. Because pn-search is a two-valued search technique, two phases are needed to prove this by pn-search. In the first phase the drawn end positions (positions which have no empty fields and neither cross nor noughts has *three in a row*) will be interpreted as lost. With this interpretation the value of the root will be proven to be lost, meaning that the player to move in the root cannot win. In the second phase the draw positions will be interpreted as won positions and with this interpretation the value of the root will be proven to be a win, meaning similarly that the player to move in the root cannot lose. Using the proven values of both interpretations it can be concluded that the player to move in the root can neither win nor loose, so the game-theoretical value is a draw. In this section the results of only the first phase will be described, thus drawn positions are interpreted as lost positions and the game-theoretical value is a loss.

It is possible that in a tree or a DAG more than one most-proving node exists. Because the algorithms choose the left-most child with equal proof (or disproof) number of a max node (or min node, respectively) on a path to a most-proving node, the order in which the children are generated for each node is important in proving a value. To obtain a fair comparison of the three algorithms, a number of different move orders is chosen. If a node is expanded during a search the generated children are randomly ordered and added to the tree or DAG.

Table 4.1 contains a selection of the experimental results of the two algorithms applied to Tic-tac-toe. For a number of different random orders of generating the children the number of nodes that are visited by each algorithm is given. The first column contains the start seed of the random generator. The second column, labeled with *tree algorithm*, contains the number of nodes visited by the standard pn-search algorithm, that is without making use of transpositions. The last column, *practical DAG algorithm*, contains the number of nodes visited by the practical DAG algorithm (section 4.4). The last row contains the average number of nodes visited by each algorithm. The average is taken over 100 experiments, of which the first ten are given in table 4.1.

The results in table 4.1 show that the use of transpositions in combination with pn-search is favourable. Compared to the tree algorithm the practical DAG algorithm has used a factor 5.23 less nodes to prove the game-theoretical value of Tic-tac-toe. Section 5.5 shows that the profit of transpositions becomes much larger when applied to large game trees.

The number of nodes are not the only results that have been compared. Table 4.2 describes the average number of updates (measured in the test cases of table 4.1) that are needed for a complete search. One 'update' in the practical DAG and tree algorithms denotes the computation of proof and disproof number of a node.

In a tree the nodes on only *one* path (from the most-proving node to the root) need to be updated. In a DAG it is possible that there exist more than one path from the most-proving

	number of nodes	
random seed	tree algorithm	practical DAG algorithm
1	13,376	3,261
2	16,797	3,248
3	15,813	3,091
4	15,325	3,108
5	16,766	3,250
6	15,093	3,367
7	16,682	3,131
8	17,278	3,216
9	16,618	3,169
10	17,747	3,331
average	17,086	3,265

Table 4.1: A typical sample of test results on Tic-tac-toe.

average number of updates	
tree algorithm	practical DAG algorithm
25,137	32,922

Table 4.2: Updates in Tic-tac-toe.

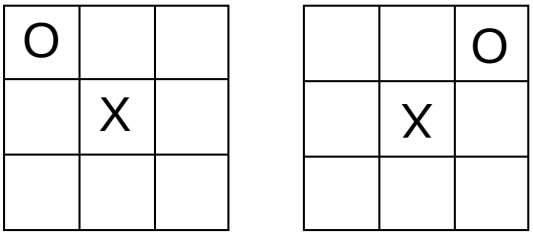

Figure 4.10: Two symmetrical positions.

node to the root. So compared with a tree the number of updates after an expansion in a DAG will be much higher. Table 4.2 shows that the total number of updates in a DAG indeed is much higher for the complete search than the number of updates in a tree. However, this increase in the number of updates is more than compensated by the gain in the number of nodes, because the creation of a node (generation of a child) is, in an implementation, a more expensive operation than the computation of the (dis)proof numbers of a node. Moreover, the number of updates can be reduced, which will be discussed in the next section.

The results of Tic-tac-toe show that using transpositions in combination with pn-search results in a rather large gain. It also showed that the practical DAG algorithm is a usable and good alternative for the theoretical DAG algorithm.

4.6 Enhancements, additional work and speed-up

In this section some additional work of the practical pn-search algorithm for DAGs will be discussed. After that some enhancements for this algorithm as well as some techniques to speed up the execution time of the algorithm are described. Finally some problems with memory constraints are mentioned.

Additional work Compared with the tree algorithm the DAG algorithm has to do some additional work. Each generated position has to be checked to see if the position is already in the graph, in other words, whether the position is a transposition. If all (unique!) positions in the graph are sorted, this checking costs $^2\log N$ comparisons, where N is the total number of positions in the graph. The positions can be kept sorted by creating a binairy search tree which contains each position that is in the graph. With the use of AVL techniques this binary tree can be kept balanced, so the lower boundary of $^2\log N$ comparisons can be approached. Nevertheless, the additional comparisons can for most domains be neglected with respect to the gain on the total number of nodes visited during pn-search.

Enhancements Of course, all enhancements described in section 3.4 can also be applied to DAG applications of pn-search. Moreover in a DAG structure another improvement can be made.

A DAG arises when transpositions are used. If *symmetries* of positions are interpreted as transpositions, a larger gain can be made. A symmetry of a position P is a position P' so that P and P' are isomorf with respect to certain permutations, like rotation and reflection. For example the Tic-tac-toe positions in figure 4.10 are isomorf with respect to a rotation of

90 degrees, so these positions are symmetrical to each other. Using symmetries decreases the number of nodes visited in a search. Appendix A shows some results that are obtained by interpreting symmetries as transpositions in Tic-tac-toe.

Speed-up The speed-up techniques described in section 3.4 (quit updating as soon as the proof and disproof number of a node have not changed) can also be applied to the practical DAG algorithm, because in this algorithm the (dis)proof number of each node is directly obtained from the (dis)proof numbers of its children. This means that if the proof and disproof number of a node have not changed, all nodes on the paths from this node to the root do not need to be updated anymore. Table 4.3 shows the number of updates that is made by this technique applied to trees and DAGs. Note that with this simple technique, the number of updates is this time much smaller for DAGs than for trees.

average number of updates	
tree algorithm	practical DAG algorithm
18,894	9,816

Table 4.3: Reduced updates in Tic-tac-toe.

Because there are more paths from the most-proving node to the root, it is possible that some nodes are part of more than one of these paths, such as the root of a DAG, that is part of each path from the most-proving node to the root. The proof and disproof number of these nodes are updated more than once, because of the recursive update procedure. This can be prevented by using a *queue* in stead of a *stack* (recursion). If the (dis)proof numbers of a node are updated, only the parents of this node that are not already in the queue are inserted in the queue. The queue implementation can only be applied to search DAGs, for which the length of all paths from each node to the root are equal (Tic-tac-toe, Connect Four and Go-moku are examples of games that have such search DAGs). Only for these search DAGs the use of queues guarantee that all children of a node V have been updated before a node V will be updated. A stack application does not guarantee this.

Memory problems In a tree the memory problem could partially be solved by deleting subtrees of nodes for which the value already had been proven. This is not possible for DAGs. In figure 4.11 the value of node b is proven to be a loss and the value of node c is not proven yet. If b was a node of a tree both the subtrees of d and e could be deleted. But in this example only the subtree of node d can be deleted, because the subtree of node e is also a (part of a) subtree of the yet unproven node c.

This is, however, not the only problem. In the subtree of node d some nodes can occur that later in the search will become a transposition. By deleting this subtree a lot of work is deleted. Therefore it is possible that some work is done for the second (or third, or fourth, etcetera) time, because this work is deleted with the subtree. This additional work was supposed to be prevented by the use of transpositions.

Nevertheless some nodes can be deleted. A node that is proven to be a win or loss, just by evaluating and not by pn-search (an end node), can be deleted. For each internal proven node, the children that are terminals and *not* transpositions can be deleted. With these deletions, no additional work has to be done.

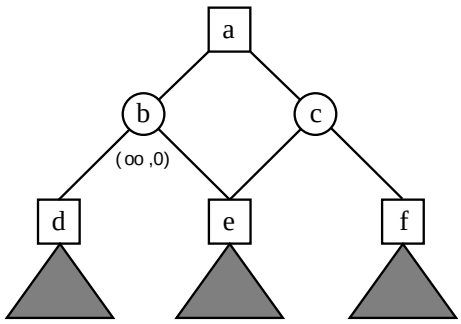

Figure 4.11: A DAG with a proven node.

Chapter 5

DCGs and proof-number search

If we are dealing with two-person games that can have repetitions of positions[1] , two major issues have to be considered.

The first is the evaluation of a repeated position. This, of course, depends on the rules of the game. In Nine Men's Morris the game is a draw as soon as a repetition occurs. In Awari and Chess the game is also a draw, but in these games only after the second repetition of the same position. The number of repetitions for which a game terminates is not important for a search technique. As soon as the second occurrence of a position has been found, this position is marked as an end position. If the computed value of the first occurrence is a draw with the aid of the second occurrence, the second occurrence would also be a draw if it had a subtree (including a third occurrence which is an end position). The form of this subtree would be equal to the form of the subtree with the first occurrence as the root and thus the second occurrence would evaluate to a draw. So the assumption that the second occurrence is a drawn end position is correct if the first occurrence would finally evaluate to a draw. If it does not evaluate to a draw there is a variant that leads to a win or loss, which does *not* contain the second occurrence. So during search it is correct to mark the first repetition of a position as an end node.

The second issue is about the structure of the search graph. Because a repetition of a position is also a transposition (the position can be reached via different paths) the created structure is no longer a tree. Because of the repetition, the structure is not acyclic anymore (and thus also no DAG), but it is a DCG (**D***irected* **C***yclic* **G***raph*).

This chapter discusses the DCGs in combination with pn-search. The first section characterizes the problem that arises with DCGs and in the second section it is proved that the most-proving node also exists in DCGs. The algorithm to determine such a most-proving node is described in the third section. After that a practical algorithm is presented and the chapter concludes with some results on DCGs.

5.1 The problem

Repetition of position in combination with transpositions (or hash tables) during the search has been a known problem for many years. A 'solution' to this problem was given by Frey (1977) in a chess implementation with use of hash tables. In his implementation he simply ignored the problem. For Chess and Awari most of the time this will do. The number of

[1] A repetition of a position is a position that occurs as ancestor of itself in the game tree.

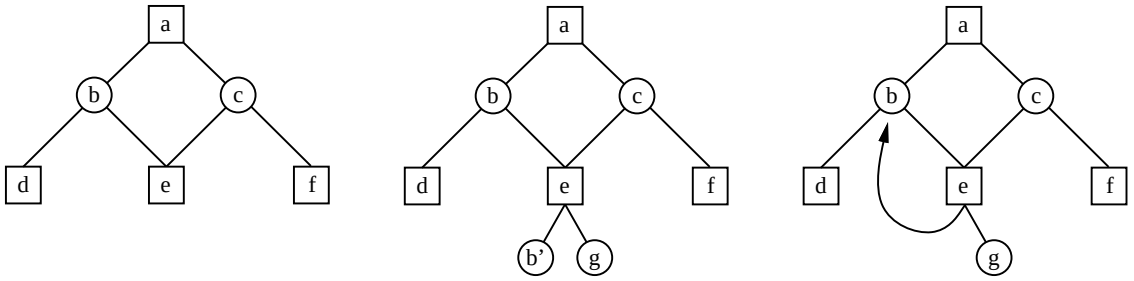

Figure 5.1: Creating a cycle by expanding.

repetitions is small with respect to the total number of nodes examined during search, so the possible problems caused by the repetitions can be neglected. On the other hand Gasser (1991) showed, with the aid of databases, that the number of repetitions in Nine Men's Morris is relatively large. So in this application neglecting may turn out to cause serious problems.

The problems caused by repetition of position in combination with transpositions will be clear if the goal of transposition use will be specified again:

> Transposition use prevents the recomputation of subtrees, thus reducing computation time and memory usage.

In figure 5.1 three example graphs are denoted. In the left-most graph node e is a DAG-like transposition node. This will not cause any problem. In the second graph node e has been expanded and it generated two children. Child g is a 'normal' terminal but child b' represents the same position as node b. Because we want to prevent equal positions in a graph, node b' has to be removed and node b becomes a transposition node. This generates the third graph which is, because of the cycle b-e-b a DCG.

A nice property of a tree or a DAG node is that its value is *unique* for the node and only depends on the values of the subgraphs of the successors. In DCGs this property does not hold. The value of, for example, node e in the third graph of figure 5.1 depends on the *path* leading to e. If e is entered via node b (path a-b-e), then the left child of e (node b) is a repetition and therefore has to be interpreted as an end node. If, on the other hand, node e is entered via node c, node b is not a repetition and therefore not an end node. So for node e two different situations can occur and it is possible that node e does not have a unique value. The same holds for node b. Its value depends of the ancestors on the paths a-b and a-c-e-b.

It is not only impossible to assign some DCG nodes a unique value, it is also not possible to compute unique proof and disproof numbers for each node. The next section will prove this.

5.2 Most-proving node exists

Despite the problems a DCG causes to prove a value of some given position with pn-search, the basic assumptions for pn-search are still applicable: the proof and disproof set can be defined, using which a most-proving node can be determined. So among the terminals in the DCG, a terminal may exist that contributes to both proof and disproof. Such a most-proving

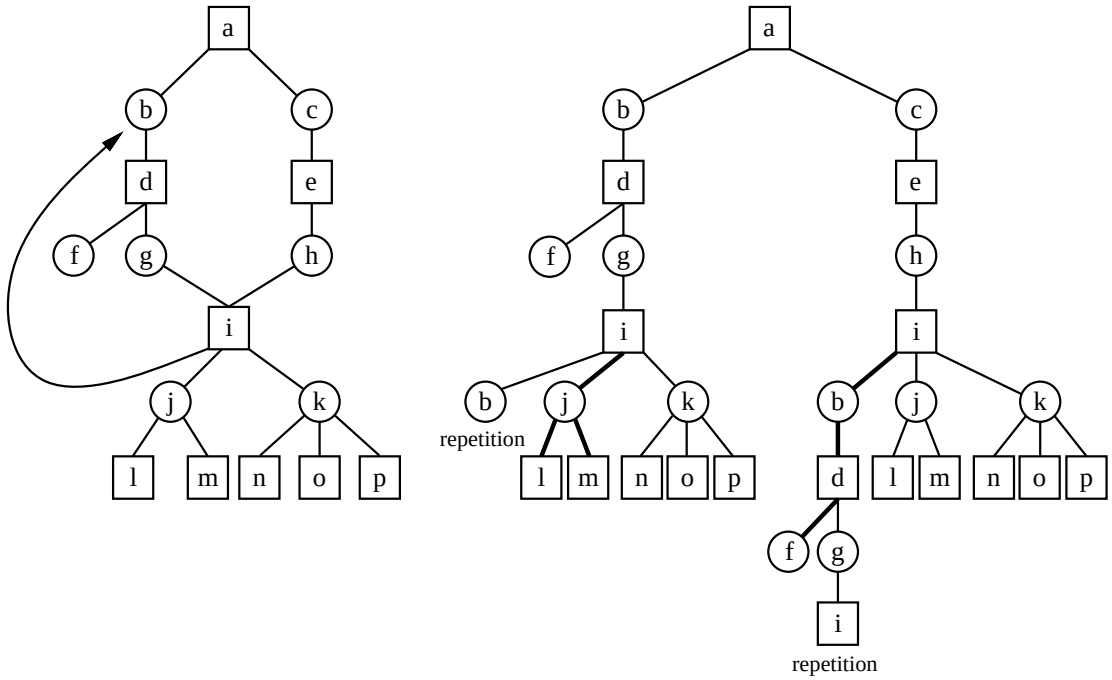

Figure 5.2: A DCG converts to a tree.

node must have the property that it is both in a smallest proof set and a smallest disproof set of the root. For DAGs the existence of a most-proving node was proven with the aid of induction: for each unproven node in the DAG the intersection of the smallest proofset with the smallest disproof set is not empty. With the aid of figure 5.2 it will be proven that not each node in a DCG has a unique most-proving node. In this figure a DCG (at the left) is converted to a tree (at the right) by creating a *unique* path in the tree for each *possible* path in the DCG. Nodes with equal labels represent equal positions. If for each node in a DCG a unique set of most-proving nodes exists then all nodes with equal labels in the tree of the converted DCG must have the same sets of most-proving nodes. The tree in figure 5.2 proves the opposite. Node i in the left subtree of root a has a set of two most-proving nodes l and m (most-proving paths are denoted with bold edges). On the other hand node i in the right subtree has only node f as most-proving node. So despite both nodes having equal labels the sets of most-proving nodes are not equal. So not every node has a unique set of most-proving nodes.

The most interesting node in trees and DAGs is the root. A terminal in the intersection of the smallest proof and smallest disproof set of the root contributes most to the proof of the game-theoretical value of the position in which we are interested (this position is represented by the root). The root in a tree or a DAG has a special property: it is the only node that has no incoming edges, in other words, it is not a successor of any node in the tree (or DAG). It is not necessary that such a node exists in a DCG, because it is possible to create a DCG such that each node is a node in a cycle; each node in that DCG is a successor of some other node. To force that a DCG does have a root with no incoming edges, the following definitions

are needed:

DCG$^+$ is a DCG in which one node V has been added, so that V has no incoming edges and only has one outgoing edge. The outgoing edge points to the node in the DCG that contains the position of which the game-theoretical value has to be proven.

DCG-root is the unique node in a DCG$^+$ without incoming edges.

If it is possible to determine the game-theoretical value of the DCG-root, then this value must be equal to the value of the position under examination. Starting in the DCG-root only one move is possible. So if the DCG-root has 'a win' (or 'a not win') value, then that only move must be a won (resp. lost) move to another node with a won (lost) position. This is the node that contains the 'to be proven' position. This also means that if the most-proving node of the DCG-root exists that it is a terminal that contributes most to the proof of the value of the 'to be proven' position.

Theorem 2 *Any unproven search DCG$^+$ has a most-proving node.*

Proof:

Each terminal in the DCG (and DCG$^+$) has no successors. So the terminal that contributes most to the proof of the value of the terminal is the terminal itself. Each terminal is unique in a DCG and therefore the complete proofset is unique. So if a DCG$^+$ is converted to a tree (like in figure 5.2), any pair of terminals with equal labels has equal (labeled) most-proving nodes. So these terminals can be joined together without loosing information. This joining creates a special type of a DAG. In figure 5.3 a formal construction is described that converts a DCG$^+$ into a DAG such that each terminal has initial complete (dis)proof sets. With the terminals initialized, the complete (dis)proof sets of each internal node of the created DAG can now be determined (see figure 3.2). According to theorem 1 each node in a DAG has a most-proving node. There is only one path in the DCG$^+$ from the root to a node that has a unique label in the constructed DAG. So the complete (dis)proof sets of these nodes are also unique. This means that for each of these nodes the intersection of the smallest proof set and the smallest disproof set can be determined and (according to theoreom 1) is not empty. The DAG's root is such a node with a unique label, because it is 'created' by the DCG$^+$'s root in the convert construction. The root of the DCG$^+$ (DCG-root) has, by definition, no incoming edges. So during the construction to a DAG it can only be pushed one time on the stack (the first time) and therefore can not generate more occurrences in the DAG.
So the DCG-root has a unique complete (dis)proof set and therefore a most-proving node.

□

Although the most-proving node exists, the proof shows that determining it may be a hard problem. The next section deals with this problem.

```
InitStack (stack);
CreateDAG-Root (DCG-root);
Push (stack, DCG-root);
while not Empty (stack) do
    Pop (stack, V);
    for each child s of V do
        if s is internal node in DCG then
            CreateNode (s);
            CreateEdge (V, s);
            if s on path from root(DAG) to V in DAG then
                s.complete-proof-set = ∅;
                s.complete-disproof-set = {∅};
            else
                Push (stack, s);
            fi
        else
            if s represents a won end position then
                CreateNode (s);
                CreateEdge (V, s);
                s.complete-proof-set = {∅}
                s.complete-disproof-set = ∅
            elseif s represents a lost end position then
                CreateNode (s);
                CreateEdge (V, s);
                s.complete-proof-set = ∅
                s.complete-disproof-set = {∅}
            elseif s not already in DAG then
                CreateNode (s);
                CreateEdge (V, s);
                s.complete-proof-set = {{s}}
                s.complete-disproof-set = {{s}}
            else
                CreateEdge (V, s);
            fi
        fi
    od
od
```

Figure 5.3: Converting a DCG$^+$ to a DAG.

5.3 Theoretical algorithms

The proof of theorem 2 shows a way to determine the most-proving node in a DCG. This proof converts a DCG to a DAG and in a DAG the most-proving node can be determined. This converting can also be done implicitly.

In a DCG each possible path will be examined depth-first, building implicitly a (labeled) tree. Each unproven terminal of the DCG is an unproven terminal in the tree and in a tree the most-proving node can be recursively determined. During the depth-first search no information about a most-proving node may be lost. Because the implicit tree can have equal labeled terminals (these terminals are joined together in the DCG to DAG conversion in the proof of theorem 2) the tree can be interpreted as a DAG. In a DAG information about the effect of a transposition must be stored. This information can be stored in (dis)proofDNFs (see section 4.3). In the interpreted DAG (the labeled tree) a node can only be a transposition if it is a terminal, but it is hard to find out whether a terminal in the DCG will be a transposition in the converted tree. To avoid this problem the complete (dis)proof set of each node in the tree can be computed in stead of the (dis)proofDNF. The depth-first algorithm that computes the complete (dis)proof sets for the root of the DCG is described in figure 5.4. If the complete (dis)proof sets are computed it is easy to determine the most-proving node.

This algorithm is, of course, not practically executable. After each expansion each node in the DCG has to be examined again at least one time to compute the new complete (dis)proof sets of it in the converted tree. Besides the expensive execution time, the storage of the complete (dis)proof sets is very expensive. Although the number of stored complete (dis)proof sets is relatively low (a maximum of d complete (dis)proof sets, with d the *depth of the converted tree*) the number of elements in the complete (dis)proof sets grows exponential. For a uniform tree with *depth* 4 and *branching factor* 3 (so the tree has $3^4 = 81$ terminals) and the label of each terminal is unique (no transpositions) the complete disproof set of the root has 3^{12} minimal disproof sets each containing 9 terminals. This exponential growth cannot be prevented by using the (dis)proofDNFs instead of the complete (dis)proof sets. Although the number of elements of the DNFs will be much lower it still grows exponentially.

The algorithm described above is not an incremental algorithm. This means that after each expansion each node has to be examined again. Despite the fact that it is impossible to compute a correct unique value for each node it is possible to compute some unique value for each node and with the aid of these values a kind of incremental update can be made in the DCG. The 'unique' complete (dis)proof set of a node V in the DCG is the complete (dis)proof set that V has if a converted tree would be built with V as the root of the DCG. Suppose that each node in the DCG in figure 5.5 has this 'unique' complete (dis)proof sets. Because node e has complete (dis)proof sets assuming that each successor node of e is not an ancestor node, the complete (dis)proof sets of c are directly derived from its children e and f, because each ancestor of c, and c itself, are no successors of e. So the assumption for which the complete (dis)proof sets of e were computed is still a correct assumption for node c. On the other hand, the complete (dis)proof sets of b cannot be directly derived from his children, because the sets in e are computed assuming that b as successor of e is a first occurrence. But looking from the perspective of node b itself, this is no longer a correct assumption (the first occurrence of b in the converted tree with e as root, is the second occurrence in the converted tree with b as root. So the complete (dis)proof sets of b must be computed independent of the sets of e. If the sets have been computed for b the sets of a can be derived directly from its children.

```
procedure ComputeCompleteSets (V : node, path : PATH-TYPE,
                               out compl-proof-set, compl-disproof-set);
begin
    if V is Terminal then
        if V represents a won position then
            compl-proof-set = {∅};
            compl-disproof-set = ∅;
        elseif V represents a lost position then
            compl-proof-set = ∅;
            compl-disproof-set = {∅};
        else
            compl-proof-set = {{V}};
            compl-disproof-set = {{V}};
    else
        if V on path then /* repetition of position ! */
            compl-proof-set = ∅;
            compl-disproof-set = {∅};
        else
            AddToPath (path, V);
            if V is a Maxnode then
                compl-proof-set = {∅};
                compl-disproof-set = ∅;
                for each child s of V do
                    ComputeCompleteSets (s, path, tmp-proof-set,
                                                  tmp-disproof-set);
                    compl-proof-set = compl-proof-set ⊎ tmp-proof-set;
                    compl-disproof-set = compl-disproof-set ∪ tmp-disproof-set;
                od
            else
                compl-proof-set = ∅;
                compl-disproof-set = {∅};
                for each child s of V do
                    ComputeCompleteSets (s, path, tmp-proof-set,
                                                  tmp-disproof-set);
                    compl-proof-set = compl-proof-set ∪ tmp-proof-set;
                    compl-disproof-set = compl-disproof-set ⊎ tmp-disproof-set;
                od
            fi
            RemoveFromPath (path, V);
    fi
end
```

Figure 5.4: Computing the DCG (dis)proof sets.

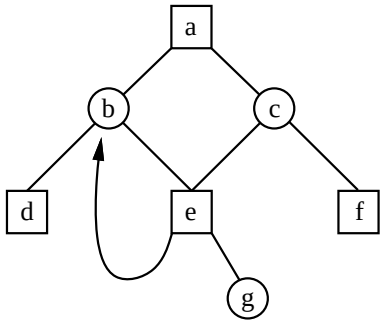

Figure 5.5: A DCG with a cycle and a transposition.

Generally, the complete (dis)proof sets of a node V in the DCG that is both a transposition and on a cycle in the DCG, cannot be derived from the sets of its children. These sets must be computed separately. For each other node in the DCG the sets can be derived from the children. This has a big advantage. Suppose that in figure 5.5 node f will be expanded and the generated children cause no new cycles in the DCG. Then the only nodes that have new complete (dis)proof sets are f, c and a, so these are the only nodes for which new complete (dis)proof sets must be computed. If on the other hand node d or g will be expanded, then the sets of all nodes (except f) must be recomputed.

In DAGs the algorithm to find the most-proving node makes use of (dis)proofDNFs but in this incremental DCG algorithm this is impossible, because with a substitution a correct unique (dis)proofDNF of the transposition to be substituted is needed. But the sets stored in the nodes of the DCG are not always correct to substitute, so the DNF method cannot be applied here. Therefore the complete (dis)proof sets must be computed to ensure the determination of a correct most-proving node.

Although this algorithm has the advantage of an incremental update its disadvantages are at a par with the algorithm first described algorithm. The storage of the complete (dis)proof sets is still an exponential problem. Although at first sight the incremental algorithm is likely to be faster than the first algorithm, this is not always the case. If for a transposition node on a cycle the sets must be (re)computed, a lot of nodes must be examined again. This has to be done for each of the nodes that are visited during the update.

It is clear that both algorithms cannot be used for practical applications. In other words, the aim of correctly determining a most-proving node is too difficult to achieve.

5.4 Practical algorithms

The largest problem in DCGs is the fact that it is impossible to derive a unique proof and disproof number for each node in the DCG. Still, this is necessary to develop an efficient and practically usable pn-search algorithm. In the previous chapters it was shown that algorithms are practically applicable if the (dis)proof numbers of each node can be directly derived from the (dis)proof numbers of his children, because an incremental update is possible in such a case.

For an incremental update, however, a DCG has two difficulties. First, it is impossible

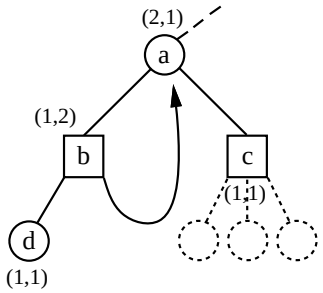

Figure 5.6: A DCG with proof and disproof numbers derived from the children.

to find unique values for each node. In section 5.3 a method is described in which for each node in the DCG certain unique values are defined, but the computation of these values is very time expensive. The memory problem with the complete (dis)proof sets can be avoided by using only a proof and a disproof number. The same has been done in the practical DAG algorithm.

The second difficulty is caused by the cycles in the DCG. Suppose that each node has some unique (dis)proof numbers, directly derivable from its children, like in figure 5.6. It is clear that the (dis)proof numbers of each internal node are derived from the (dis)proof numbers of its children. Suppose node c is the node to be expanded and it generates three children (denoted by the dotted nodes). Each child generated is initialized with (dis)proof numbers (1,1). After c's expansion, the (dis)proof numbers in the DCG have to be updated: node c is assigned (1,3) and node a (2,2). Because the (dis)proof numbers of a are changed, the (dis)proof numbers of its parents can change, too. Therefore node b has to be updated: it receives (1,3). Thereafter the values of node a have to be updated again: (2,3), and after that the values of node b updated again: (1,4) and a again: (2,3). The (dis)proof numbers of a have been updated three times in this example and if c had generated more than three children, the deriving of stable values for node a would require even more iterations. In the worst case, if node c evaluates to a loss ((dis)proof numbers: (∞,0)) for example, an infinite number of iterations is, theoretically, needed to derive the stable values of node a (and b).

The opposite, forcing that each node will be updated at most one time, after an expansion, does not work either. Suppose that in figure 5.6 the proof and disproof number of each node are computed after expansion of node b (node c is not expanded). If terminal d now evaluates to a loss, its values will be (∞,0). We have already seen that with these values the value of a can be proven to be a loss. But if each node will be at most updated once, the values of a are not (∞,0), but (3,1). Because the value of a is not proven yet, a new most-proving node needs to be found. Because node c is the only unproven terminal left, it must be the most-proving node and therefore will be expanded. Again it will take a long time before a value is proven.

The two methods described above make clear that the troubles are caused by cycles. So the solution may be to prevent the appearance of cycles in the search structure. Three methods come to mind.

The first method is to neglect all kind of transpositions, but in that case the created structure is a tree, and a lot of additional (unnecessary) work will be done in proving a value.

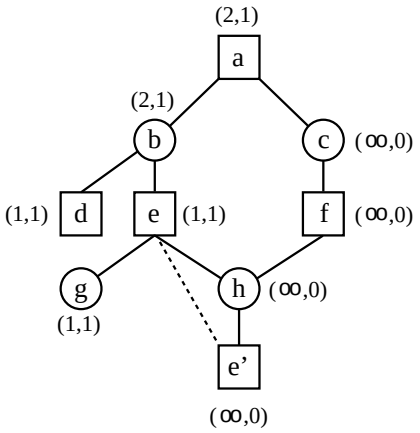

Figure 5.7: Missing a won variant by incorrect assumption.

This additional work was aimed to be avoided by the use of transpositions.

The second method is to consider only the positions after a *converting move* as a (possible) transposition (for example in chess all positions after capture moves, pawn moves or castling). Although the number of visited nodes decreases with respect to the number of vistited nodes in a tree (thus without using transpositions) still a lot of equal positions remain in the DAG.

The third method in avoiding cycles reduces the number of equal positions in a DAG to two, of which one always is an end node. Each new generated position p is checked for two things. First it is checked whether p is generated earlier in the search DAG. If so, then this position is a transposition. It will be treated as a transposition *unless* this position is on at least one path from the new generated position to the root. If this is the case then p is *considered* to be an *end* position and not as a transposition, because that would create a cycle. The (dis)proof numbers of the new node, node V', that represents this end position, are initialized with the interpretation of a draw position. If, for example, a draw is interpreted as a loss, then the (dis)proof numbers of node V' become (∞,0). End node V' keeps proof and disproof number (∞,0) as long as the value of the other node, say V, representing the same position as V', does not prove the opposite. If, however, the value of V is proven to be a win (the opposite), then this is proved with the (for node V correct) assumption that the value of V' is a loss. But if the value of node V is a win, then V represents a won position, and therefore the equal position represented by V' is also won. So the proof and disproof number of V' must be reset to (0,∞) and the (dis)proof numbers of each ancestor of V' must be updated.

If with these *assumed* values the value of the root is proven to be a win, then a won variant exists that leads from the root to some end node that represents a won position. This variant does *not* contain the (possibly wrong) assumed end positions. So despite of the fact that the assumptions might be wrong, the proof of the value is correct.

If, however, the value of the root is proven to be a loss, it is possible that a won variant is missed. In figure 5.7 node e' represents the same position as node e. Now suppose that node d is going to be expanded and that it will evaluate to a loss. Then the (dis)proof numbers of nodes b and a will be updated to (∞,0) and therefore it is proven (with the assumption that

node e' is a lost end position) to be a loss. If, however, node g would evaluate to a win, then node e would be a win, too. So in that case the assumption that node e' represents a lost position is wrong. In fact if it was known that the value of node e' would be a win, then h, f, c and a would evaluate to a win, too. So a won variant is missed if e' is assumed to represent a lost end position.

Despite this incorrectness, the algorithm derived from this method is useful, which is shown by the results in the next section. The pseudo code for this algorithm is equal to the pseudo code for the DAG algorithm (it is, in fact, a DAG algorithm) with a small extension to the *expand* procedure (see figure 4.8 in section 4.4). In this procedure it must be checked whether a new generated position p, which is a transposition, is on a path to the root. If this is the case, a new (end) node, say node V', representing p, is created with (dis)proof number equal to the draw interpretation, unless the value of the other node that represents p, say node V, has already been proven to be the opposite of the draw interpretation.

The update procedure is a little more difficult, because if the value of a node V is proven to be a win and an end node V' exists, representing the same position as V, with the assumed 'loss' value, then the proof and disproof number of V' must be reset. So the algorithm needs a (directed) link from node V to node V'. This link can be created in the expand procedure.

It is important *not* to update the (dis)proof number of the parent(s) of a node, if both proof and disproof number of that node are not changed. Suppose the proof and disproof number of a node V' are just reset, because the value of a node V, which represents the same position as V', is proven to be a win. Then all ancestors of V' must be updated. One of the ancestors of V' is V. After updating V, the value is, of course, again proven to be a win. So V' must be reset and now an updating cycle will occur, if again all ancestors are updated. If, however, the ancestors of an 'unchanged' node are not updated then the endless loop will be avoided.

5.5 Results

This section contains the performances of the latter practical DCG algorithm, described in the previous section. The practical DCG algorithm has been applied to chess positions, which are all described by Krabbé (1985) and Reinfeld (1958). All these positions are won for white. The performances of the DCG-pn algorithm will be compared with the performances of the tree-pn algorithm. The results of the tree algorithm are obtained by (Breuker *et al.*, 1994). For both tree and DCG algorithm a special initialization for proof and disproof number in the terminals not representing end positions, has been used. The proof number of a min node is initialised with the number of moves the min player can do. The proof number is initialized with one. For a max node, the disproof number is initialized with the number of moves the max player can do and the disproof number with one.

The results of the algorithms on 35 positions from *Chess Curiosities* have been listed in table 5.1. In appendix B the results on the other test positions have been listed. In table 5.1 and the tables in appendix B the first column contains a number representing the test position. The number corresponds to the problem number. The second column, labeled with *tree algorithm*, contains the number of nodes visited by the tree algorithm. The column labeled with *DCG algorithm* contains the number of nodes visited by the practical DCG algorithm. The last column contains the gain factor. If both algorithms did not prove a win for a position then the gain factor is denoted as '-'. The tree search was terminated after 1,750,000 nodes

were held up in memory. The DCG search was terminated after 800,000 nodes. Because the tree algorithm removed the solved subtrees, sometimes more than 1,750,000 nodes could be visited. The DCG algorithm did not remove any nodes during the search. In this implementation, transposition nodes are stored more than once. This explains the fact that the DCG algorithm seems to terminate before the 800,000 nodes are reached.

The results in table 5.1 and in appendix B show that the DCG algorithm performs in most of the tested positions better than the tree algorithm. More important is the fact that in only 3 of the 82 tested positions it is sure that a won variant is missed (positions 206, 208 and 219 in table 5.1). One other position, 215, is solved but the DCG algorithm visited more nodes than the tree algorithm.

As expected, the difference in the number of visited nodes of both algorithms grows with the number of visited nodes in the tree. If the number of nodes is low in a tree, the chance that a transposition occurs is rather small. Therefore the DCG algorithm will not perform better than the tree algorithm. On the other hand, if the number of nodes in the tree is large, it contains probably many equal positions. This explains the much better performance of the DCG algorithm in, for example, position 284.

position nr.	tree algorithm	DCG algorithm	factor
008	1,110,701	590,492	1.9
035	296	276	1.1
037	43,221	21,865	2.0
038	273	272	1.0
040	> 2,162,847	> 724,408	-
044	> 2,112,804	271,219	> 7.8
060	1,724,491	365,063	4.7
061	42,228	34,333	1.2
078	> 2,501,127	> 769,362	-
192	23,290	14,302	1.6
194	229,423	28,400	8.1
195	> 1,795,039	> 652,568	-
196	298,428	78,741	3.8
197	323	318	1.0
198	247,435	151,150	1.6
199	370,016	151,804	2.4
206	15,978	> 647,160	< 0.025
207	95,418	31,646	3.0
208	62,791	> 626,160	< 0.1
209	> 2,187,888	> 691,329	-
210	> 1,854,764	> 683,914	-
211	957	831	1.2
212	81,842	50,753	1.6
214	685	669	1.0
215	114,060	336,392	0.3
216	592,890	156,825	3.8
217	> 1,876,572	340,643	> 5.5
218	118,361	127,359	0.9
219	310,447	> 685,302	< 0.5
220	> 1,979,542	> 676,491	-
261	482	420	1.1
284	> 1,795,414	2,658	> 675.5
317	173,480	114,346	1.5
333	145,922	104,991	1.4
334	217,516	116,952	1.9

Table 5.1: Test results on chess positions from *Chess Curiosities*.

Chapter 6

Conclusions

After sketching the use of pn-search in common tree search, we tackled the problem of transpositions in game-tree search. It turned out that determining the theoretically correct most-proving node is not an attainable goal, when pn-search is applied to DAGs. On the other hand, the algorithms described for trees can also be applied to DAGs. Though the most-proving node that is determined by this algorithm need not be theoretically correct, the algorithm is efficient and seems always be better with respect to the number of nodes visited compared to pn-search on trees.

The theoretically correct most-proving node in DCGs is even more difficult to achieve than it is in DAGs. In fact, creating an algorithm that can run efficiently in a DCG is a hard problem that is not solved during the research described in this thesis. Nevertheless the algorithm described in section 5.4 seems to fulfil the goal of the research: using transpositions in combination with pn-search. In most of the test cases it reduces the number of nodes, works efficiently and in most positions a won variant was not missed. Therefore the DCG algorithm is a strong alternative if tree algorithms cannot solve a position, especially for post-mortem analysis.

The technique described in section 5.4 is not only a solution for transpositions and repetitions in combination with pn-search, but it is applicable to all kinds of best-first search techniques that uses transpositions.

Still a lot of improvements have to be done. The biggest problem for pn-search with transpositions is reducing the memory size of the search DAG. Deleting solved subtrees or subgraphs is only applicable for terminals in the DAG. Improvements on deleting (unimportant) nodes are necessary, to break through the memory constraint.

Appendix A

DAG results on Tic-tac-toe

Below are the test results of the practical DAG algorithm applied to Tic-tac-toe. In these experiments the DAG is not only created by transpositions, but also by interpreting symmetrical positions as transpositions. Two Tic-tac-toe positions are symmetrical if they are isomorph with respect to rotation and reflection. The results obtained by the standard tree algorithm are the same as those of table 4.1 in section 4.5 and are given for comparison purposes only. The average (last row) has been determined over hundred experiments, of which the first 10 are listed.

	number of nodes	
random seed	no transpositions	symmetrical positions
1	13,376	561
2	16,797	676
3	15,813	564
4	15,325	577
5	16,766	587
6	15,093	634
7	16,682	606
8	17,278	659
9	16,618	672
10	17,747	662
average	17,086	617

Table A.1: A typical sample of test results on Tic-tac-toe.

Appendix B

DCG results on Chess

Below are the results on test positions obtained from *Win at Chess*.

position nr.	tree algorithm	DCG algorithm	factor
001	7,640	5,782	1.3
004	82	82	1.0
005	57	57	1.0
006	71,966	9,563	7.5
009	207	198	1.0
012	175	173	1.0
014	324,542	155,761	2.1
027	77	77	1.0
035	527	339	1.6
049	16,546	13,920	1.2
050	183	183	1.0
051	227,361	129,293	1.8
054	85	85	1.0
055	31,456	25,446	1.2
057	113	113	1.0
060	69	69	1.0
061	78	78	1.0
064	137	137	1.0
079	152	152	1.0
084	93	93	1.0
088	759	687	1.1
096	1,640,786	595,457	2.8
097	107	107	1.0
099	75,411	60,718	1.2

Table B.1: Test results on chess positions from *Win at Chess*.

position nr.	tree algorithm	DCG algorithm	factor
102	279	279	1.0
103	2,150	2,031	1.1
104	5,047	4,566	1.1
105	> 2,224,022	> 733,704	-
132	2,301	2,084	1.1
134	854	768	1.1
136	185	185	1.0
138	211,466	137,848	1.5
139	274	274	1.0
143	900	816	1.1
154	117	117	1.0
156	82	82	1.0
158	526	525	1.0
159	385,487	221,318	1.7
160	110	110	1.0
161	2,045	1,191	1.7
167	896	761	1.2
168	596,956	236,008	2.5
172	99	99	1.0
173	419	404	1.0
177	527	520	1.0
179	184	184	1.0
182	807,709	216,039	3.7
184	82	82	1.0
186	108	108	1.0
188	117	117	1.0
191	22,466	17,287	1.3
197	95	95	1.0

Table B.2: Test results on chess positions from *Win at Chess*.

position nr.	tree algorithm	DCG algorithm	factor
201	1,019,679	418,655	2.4
203	19,917	17,068	1.2
211	278	231	1.2
212	458	457	1.0
215	164	164	1.0
217	271	271	1.0
218	277,639	177,847	1.6
219	157	157	1.0
222	59,591	25,194	2.4
225	342	342	1.0
241	360,983	191,588	1.9
244	458	458	1.0
246	120	120	1.0
250	1,147	1,121	1.0
251	136,479	84,844	1.6
252	537,628	353,999	1.5
253	2,355	1,066	2.2
260	807	730	1.1
263	887	848	1.0
266	716	711	1.0
267	1,206	1,137	1.1
278	636	636	1.0
281	317,214	36,705	8.6
282	749	725	1.0
283	30,778	28,008	1.1
285	218	218	1.0
290	523	523	1.0
293	121,720	93,647	1.3
295	81	81	1.0
298	150	150	1.0

Table B.3: Test results on chess positions from *Win at Chess*.

Bibliography

The numbers between parentheses after each bibliographic entry refer to the pages on which a reference to the entry in question occurred.

Allis L.V. (1994). *Games and Artificial Intelligence*. PhD thesis, University of Limburg, Maastricht, The Netherlands. To appear. (14)

Allis L.V. and Schoo P.N.A. (1992). Qubic Solved Again. *Heuristic Programming in Artificial Intelligence 3: the third computer olympiad* (eds. H.J. van den Herik and L.V. Allis), pp. 192–204. Ellis Horwood, Chichester, England. (14, 29)

Allis L.V. and Van den Herik H.J. (1992). Go-Moku opgelost met Nieuwe Zoektechnieken. *Proceedings of the N.A.I.C. '92* (eds. De Swaan Arons, Koppelaar, and Kerckhoffs). Delftse Universitaire Pers, Delft. (14, 29)

Allis L.V., Van der Meulen M., and Van den Herik H.J. (1991). *Proof-Number Search*. Report CS 91-01, Dept. of Computer Science, University of Limburg. (1, 14, 15)

Allis L.V., Van der Meulen M., and Van den Herik H.J. (1994). Proof-Number Search. *Artificial Intelligence*. To appear. (1, 14, 15)

Breuker D.M., Allis L.V., and Van den Herik H.J. (1994). Mate in 38: Applying Proof-Number Search to Chess. *Advances in Computer Chess 7*. To appear. (15, 45)

Frey P.W. (1977). *Chess Skill in Man and Machine*. Springer-Verlag, New-York. (35)

Gasser R. (1991). Applying Retrogade Analysis to Nine Men's Morris. *Heuristic Programming in Artificial Intelligence 2: the second computer olympiad*. (eds. D.N.L. Levy and D.F. Beal), pp. 161–173. Ellis Horwood Limited, Chichester, England. (36)

Gnodde J. *Aïda, New Search Techniques Applied to Othello*. Master's thesis, University of Leiden, The Netherlands, (1993). (15)

Knuth D.E. and Moore R.W. (1975). An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, Vol. 6, No. 4, pp. 293–326. (4)

Krabbé T. (1985). *Chess Curiosities*. George Allen and Unwin, Ltd, London. (45)

McAllester D.A. (1988). Conspiracy Numbers for Min-Max Search. *Artificial Intelligence*, Vol. 35, pp. 287–310. (1)

Neumann J. von (1928). Zur Theorie der Gesellschaftsspiele. *Math. Ann*, Vol. 100, pp. 295–320. Reprinted (1963) in John von Neumann Collected Works (ed. A.H. Taub), Vol. VI, pp 1–26. Pergamon Press, Oxford–London–New-York–Paris. (3)

Reinfeld F. (1958). *Win at Chess*. Dover Publications, Inc., New-York. (45)

Uiterwijk J.W.H.M., Van den Herik H.J., and Allis L.V. (1990). A Knowledge-Based Approach to Connect-Four: The Game is Over White to Move Wins. *Heuristic Programming in Artificial Intelligence: The first computer olympiad.* (eds. D.N.L. Levy and D.F. Beal), pp. 113–133. Ellis Horwood Limited, Chichester, England. (14, 29)

Zermelo E. (1912). Uber eine Anwendung der Mengenlehre auf die Theorie des Schachspiels. *Proceedings of the fifth International Congress of Mathematics, Vol. 2*, pp. 501–504. Cambridge, England. (3)