Chapter 2

Proof-Number Search

2.1 Knowledge representation and search

Problem solving is one of the corner-stones of AI research. Within problem solving, we distinguish two subprocesses: choosing a knowledge representation and performing a search. We remark that the term knowledge representation is meant to include analysis, conceptualization and formalisation. A well-chosen representation may considerably reduce the amount of search needed to solve a problem, while a badly chosen representation may render solving a problem (virtually) impossible. As an example we present the mu-puzzle (Hofstadter, 1979).

> A production system consisting of four rewriting rules generates theorems consisting of the letters M, I and U. In each production, x and y denote any string of letters.
>
> 1. $x\mathrm{I} \rightarrow x\mathrm{IU}$
> 2. $\mathrm{M}x \rightarrow \mathrm{M}xx$
> 3. $x\mathrm{III}y \rightarrow x\mathrm{U}y$
> 4. $x\mathrm{UU}y \rightarrow xy$
>
> The goal of the mu-puzzle is to determine whether MU is a theorem in the above system, given that MI is the only axiom.
>
> In a first attempt to solve the puzzle, we represent a theorem simply by its string of letters. The rewriting rules are used to expand nodes of the search tree, where each node represents a theorem. We are now faced with a tree-search problem: to

> find a path of rewriting rules leading from the initial state MI to the goal state MU. A suitable tree-search algorithm is selected to perform the search, such as breadth-first search or depth-first search. To select a search algorithm, various criteria may be applied. For instance, breadth-first search guarantees that the first solution found is also the shortest solution (Nilsson, 1980). A disadvantage of breadth-first search is that it requires more working memory than an algorithm such as depth-first search (Nilsson, 1980). Generally, each of the applicable search algorithms has its own advantages and disadvantages. In case no solution exists, these algorithms have the disadvantage that the search will not terminate, as the set of theorems is infinite.
>
> Instead of concentrating on the selection of the best possible search algorithm, we may first try to optimize the chosen representation. For the mu-puzzle, a better representation involves an extra item of knowledge per theorem. This Boolean item, which we name IsTripleI, indicates whether the theorem's total number of Is is a multiple of three. We can now verify that each of the four rewriting rules creates new theorems with IsTripleI's value equal to that of the theorem it is created from. The observation that MI (false) and MU (true) have unequal IsTripleI values is sufficient to prove that MU is not a theorem.

In the mu-puzzle example, it was possible to eliminate all search by enhancing the representation of the puzzle. It illustrates that choosing a representation should have the highest priority when solving problems. Choosing a knowledge representation in problem solving is mostly domain-specific. Even though general techniques (such as abstraction, here applied to the mu-puzzle) exist, their successful application remains the fruit of a thorough understanding of the domain under investigation.

For problems more complex than the mu-puzzle, a good representation generally does not eliminate all search; it merely reduces the size of the state space to, hopefully, manageable proportions. It is then important to select a search algorithm which will find a solution, if it exists, in an efficient manner. The efficient manner is to be understood here in a broad sense, including programming time, calculation time and the required amount of working memory. The weighting of these resources depends on the circumstances in which the problem has to be solved.

Thus, the domain-specific task of finding a suitable knowledge representation is performed in combination with the selection of a search algorithm well-suited for the state space. In the course of a considerable number of years of research in AI, many different search algorithms have been developed. We distinguish between several *categories* of search problems, such as those represented by single-agent trees, AND/OR trees and game trees (Nilsson, 1971). While the category that a search problem belongs to restricts our choice of search algorithms, *within* each category several search algorithms exist, each with its own characteristics. These characteristics determine the scope of problems for which the algorithm may be preferred over the other algorithms within the same category. We remark that the division into search categories is not strict. An example relevant to this thesis is that two-valued game-tree searches can also be performed by search algorithms for AND/OR trees.

For the category of game trees, many different search algorithms have been developed. We name the best known algorithms and mention the type of search problems for which we believe they are best suited:

- By far the best-known game-tree search algorithm is α-β search (Knuth and Moore, 1975). It is a *directional* (also known as *depth-first*) algorithm, having working-memory requirements linear in the depth of the tree investigated. Knuth and Moore (1975) have shown that α-β search achieves optimal efficiency on perfectly-ordered uniform trees. Application of iterative deepening to α-β search ensures for many application domains that strongly-ordered trees are traversed, resulting in close-to-optimal efficiency on uniform trees (Campbell and Marsland, 1983).

- SSS* is a *best-first* search algorithm (Stockman, 1979). It will never investigate a node pruned by α-β search (Campbell and Marsland, 1983).

  The algorithm has two drawbacks. First, as with all best-first search algorithms, the working-memory requirements are linear in the number of nodes created, thus exponential in the depth of the tree. However, recently variants requiring less working memory have been developed (Reinefeld, 1994). Second, the reduction in the number of nodes searched compared with iterative-deepening α-β search does not outweigh the cost of maintaining the search tree (or open list) in working memory for most practical applications. However, if the cost of heuristic evaluation is large compared to the cost of traversing the tree, or if obtaining a good ordering through iterative deepening for α-β search is difficult for the domain under investigation, SSS* may be an alternative to be preferred.

- Another best-first search algorithm is B* (Berliner, 1979). It depends on the availability of reliable heuristic estimates for the upper and lower bounds on the value of internal nodes. For chess, the algorithm has been implemented in HITECH, but it remains unclear whether for this domain sufficiently accurate upper and lower bounds can be estimated to result in better move selection than by algorithms based on α-β search.

- Conspiracy-number search (cn-search) (McAllester, 1988; Schaeffer, 1989) is a best-first search algorithm which determines the cardinality of the smallest sets of (terminal) nodes which must change their value in order to change the value of the root. Once this cardinality grows beyond a pre-specified bound, it is considered unlikely that the root value will change, and the search is terminated. Cn-search has shown its merits in tactical chess positions (Schaeffer, 1989), but has failed in a comparison with α-β search in a tournament chess program (Van der Meulen, 1990). Cn-search has as disadvantages the large amount of bookkeeping necessary at each node, and the subsequent amount of working memory required to perform the search. One of the ideas underlying cn-search is that the distribution of the values over the leaf nodes of the tree, and the shape of the tree, should influence the selection of the next node to be investigated.

The last aspect of cn-search, using the shape of the tree to guide the search, has been singled out in *proof-number search* (pn-search), which can be seen as a successor to conspiracy-number search. In this chapter we present pn-search, which has the exploitation of non-uniformity as its main theme. Pn-search will be presented as an AND/OR tree search algorithm, even though all applications discussed in this thesis concern game trees.

We introduce in section 2.2 the pn-search algorithm for AND/OR trees. In section 2.3, several enhancements to the algorithm are presented. These include techniques to reduce execution time and usage of working memory, examples of the application of domain-specific knowledge, and a discussion regarding transpositions within pn-search. Results of applying pn-search to a practical domain, the game of awari, are presented in section 2.4, where its performance is compared with those of sophisticated implementations of

α-β search. Finally, section 2.5 contains a discussion of related algorithms, analyzing the similarities and differences between pn-search and conspiracy-number search, SSS*, B* and A*. (A* (Hart *et al.*, 1968), a single-agent search algorithm, has been included in this list because of its similarities with pn-search.)

2.2 Pn-search: the algorithm

In this section we introduce pn-search for AND/OR trees. First, in section 2.2.1 we define our tree model and a precise terminology for the remainder of the chapter. Then, the main assumptions of pn-search are described in section 2.2.2 and the notions of proof numbers and disproof numbers are introduced. Next, section 2.2.3 informally discusses the order in which the nodes of a pn-search tree should be created. Finally, an algorithm in pseudo-code for pn-search is presented in section 2.2.4.

2.2.1 The AND/OR-tree model

We define our tree model as follows. In the tree, there are two *types* of nodes: AND nodes and OR nodes. We assume that each node can be *evaluated*, leading to one of three values: false, true or unknown. Please note the difference between nodes which have not yet been evaluated (thus whose evaluation value is not yet known) and nodes which *have* been evaluated and obtained the value unknown.

Nodes with evaluation value unknown can be *expanded*. When a node J is expanded, a non-empty set of *child* nodes is created, each having J as *parent* node. A node which has been expanded is an *internal* node. There are three kinds of *leaf* nodes, i.e., nodes without children. First, a node evaluated to false or true is a *terminal* node. Second, a node which has evaluated to unknown is called a *frontier node*. Third, a node which has not yet been evaluated is also called a *frontier node*.

There are two *tree-creation procedures*, which we name *immediate evaluation* and *delayed evaluation*. When applying immediate evaluation each node in the tree is immediately evaluated upon creation. The tree is initialized by creating (and evaluating) the root. Then, as long as the tree has not been solved, at each step a frontier node is selected (which, since it has already been evaluated, must have value unknown), expanded and all its children are immediately evaluated. This process of expanding a node J and evaluating J's children is called *developing* node J. In case of delayed evaluation, each node is only evaluated when it is selected, instead of at creation. Thus, the tree is initialized by creating the root (without evaluation). Then, at each step a frontier node J is selected (which is guaranteed not to have been evaluated) and evaluated. If the evaluation value of J is unknown, J is expanded (without evaluating J's children). Here the process of evaluating a node, possibly followed by its expansion, is also called *developing* node J. We remark that the terms *frontier node* and *developing* each have a double meaning. However, once the tree-creating procedure has been specified, both terms are unique. This approach has been chosen so that pn-search can be explained independently of the tree-creation procedure.

The value of an expanded internal AND node A is determined as follows: if A has at least one child with value false, A also has value false; otherwise, if A has at least one child with value unknown, A has value unknown; otherwise A has value true. The value of an expanded internal OR node O is determined as follows: if O has at least one child with value true, O also has value true; otherwise, if O has at least one child with value unknown, O has value unknown; otherwise O has value false. A tree is *solved* if the value of its root has been established as either true or false. A solved tree with value true is called *proved*, while a solved tree with value false is called *disproved*.

Throughout this chapter, we depict AND nodes by circles and OR nodes by squares in each of the figures. Furthermore, AND nodes can be recognized by the arcs linking their children, in accordance with standard conventions for depicting AND/OR trees.

2.2.2 Main assumptions of pn-search

Best-first search algorithms select a best node (according to some criterion) in the search tree, develop the node and then update such information as is necessary for the algorithm to continue. The distinguishing factor of each best-first search algorithm is the manner in which a node is characterized as 'best'.

For pn-search we assume that we have no knowledge regarding *a priori* probable values of nodes, nor knowledge regarding correlations between node values, although this knowledge could be added to the program (see section 2.3.3). Instead, only the position of a node in the tree and its possible contribution to solving the tree is considered.

First, we formulate the assumptions of pn-search, implying the above. Second, we present some definitions to aid in the description of pn-search. Third, using an example, we illustrate that some nodes are better in their contribution to solving the tree than others. Finally, we summarize our findings.

Assumptions

While searching AND/OR trees, we make the following two assumptions.

1. The probability distribution of values (true, false, unknown) for a frontier node is unknown.

2. The probability distribution of values (true, false, unknown) for a frontier node is equal throughout the tree.

Even though these assumptions mean that we cannot distinguish between two nodes by looking at them independently of their context, nevertheless their position in the tree may influence their expected contribution to solving the tree.

Definitions

When searching AND/OR trees, developing a single frontier node is often insufficient to solve the tree. In most cases, several frontier nodes must obtain the value true to prove the tree or the value false to disprove it. This observation is reflected in definitions 2.1 and 2.2.

Definition 2.1 *For any* AND/OR *tree T a set of frontier nodes S is a* proof set *if proving all nodes within S proves T.*

Definition 2.2 *For any* AND/OR *tree T a set of frontier nodes S is a* disproof set *if disproving all nodes within S disproves T.*

Since it will turn out that we shall use the cardinality of proof and disproof sets, these are given names in definition 2.3 and 2.4.

Definition 2.3 *For any* AND/OR *tree T, the* proof number *of T is defined as the cardinality of the smallest proof set of T.*

Definition 2.4 *For any* AND/OR *tree T, the* disproof number *of T is defined as the cardinality of the smallest disproof set of T.*

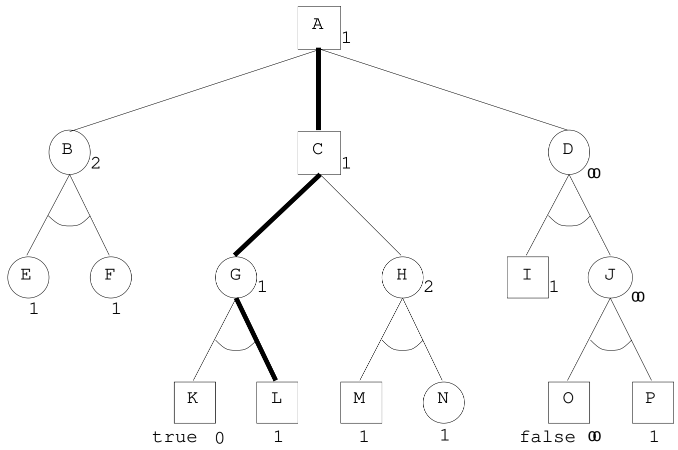

Figure 2.1: AND/OR tree with proof numbers.

Examples

To illustrate how the context can be used to distinguish between nodes, we have depicted an AND/OR tree in figure 2.1.

With each node, we have associated the *proof number* of the subtree with that node as its root, as defined in definition 2.3.

All frontier nodes (E, F, I, L, M, N and P in figure 2.1) have proof number 1. This follows from the fact that only the node itself needs to obtain the value true to prove the whole subtree (consisting of only the node itself). A terminal node with value true (node K in figure 2.1) has proof number 0, since its value has already been proved. Terminal nodes with value false (node O in figure 2.1), have proof number ∞, since there is no smallest finite set of nodes which can undo the fact that the node is disproved. Internal AND nodes obtain the value true only if all their children are proved. Thus, internal AND nodes (B, D, G, H and J in figure 2.1) have proof numbers equal to the *sum* of the proof numbers of their children. For internal OR nodes it suffices to prove one of their children, in order to have the parent obtain the value true. Thus, for internal OR nodes (A and C in figure 2.1) we establish the proof number by taking the *minimum* of the proof numbers of their children.

Root A of the tree in figure 2.1 has proof number 1. This indicates that somewhere in the tree a frontier node exists, which, by obtaining the value true, would complete the proof of the tree. The path from the root to this frontier node can be found by examining the proof numbers. To prove the

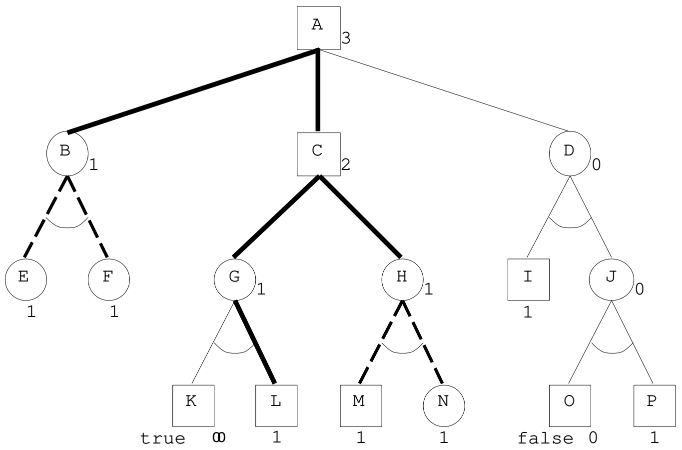

Figure 2.2: AND/OR tree with disproof numbers.

root (an OR node), it is sufficient to prove one of its children. Child C has the smallest proof number among the three children of A. The frontier node we are looking for thus lies within subtree C. In the same way, node G is preferred over node H, since G's proof number is equal to 1, while H's proof number equals 2. To prove node G (an AND node), it is necessary to prove all its children. Child K has already been proved, thus only a proof of node L is needed, which is the frontier node we have been looking for.

We could now proceed and develop node L, in an attempt to prove the tree. Instead, we will first determine which nodes may contribute to a potential disproof. In figure 2.2 we have depicted the same tree as in figure 2.1. With each node, we have associated the *disproof number* of the subtree with that node as root, as defined in definition 2.4.

The disproof numbers behave analogously to proof numbers, interchanging the roles of AND nodes and OR nodes, and the cardinalities 0 and ∞. Thus, frontier nodes (E, F, I, L, M, N and P in figure 2.2) have disproof number 1. A terminal node with value false (node O in figure 2.2) has disproof number 0, since it is already disproved. Terminal nodes with value true (node K in figure 2.2) have disproof number ∞. Internal AND nodes (B, D, G, H and J in figure 2.2) have disproof numbers equal to the *minimum* of the disproof numbers of their children. Internal OR nodes (A and C in figure 2.2) have disproof numbers equal to the *sum* of the disproof numbers of their children.

Root A of the tree in figure 2.2 has disproof number 3. This means that at least 3 nodes must obtain the value false to disprove the tree. Analysis of the tree shows that it involves one of the nodes E and F, node L and one of the nodes M and N.

Summary

The previous paragraphs illustrate that proof numbers and disproof numbers can be used to find nodes within the smallest subset of frontier nodes in the tree which, by all obtaining the same value, solve the tree.

From the assumptions underlying pn-search it follows that the probability that all nodes in a proof set obtain the value true increases with decreasing cardinality of the proof set (except in the trivial cases that the probability of evaluation to true equals either 0 or 1). As a result the total number of node developments needed to solve a tree is (on the average) reduced by first focusing on potential solutions involving a small number of nodes (i.e. subtrees with small proof and/or disproof numbers), before trying to find solutions known to require a larger number of nodes. This expectation is the basis for the pn-search algorithm as described in the following sections.

2.2.3 Informal description of pn-search

Pn-search continuously tries to solve the tree by focusing on the potentially shortest solution, i.e., consisting of the least number of nodes. At each step of the search, a node which is part of the potentially shortest solution available is selected and developed. After the development of a node, its proof number and disproof number are established anew. Then, the proof and disproof numbers of its ancestors are updated. This process of selection, development and ancestor updating is repeated until either the tree is solved or we have run out of resources (time or working memory).

The main issue yet to be resolved is to decide (1) to select a node in the smallest *proof* set, or (2) to select a node in the smallest *disproof* set. We will show in the following paragraphs that, maybe surprisingly, we can always do both at the same time. This results in the definition of a most-proving node as in definition 2.5.

Definition 2.5 *For any* AND/OR *tree T, a* most-proving node *of T is a frontier node of T, which by obtaining the value* true *reduces T's proof number by 1, while by obtaining the value* false *reduces T's disproof number by 1.*

Definition 2.5 assumes that within each unsolved tree T a frontier node exists, which is an element of the intersection of a smallest proof set and of a smallest disproof set of T. A stronger claim is that *each* pair consisting of a smallest proof set and a smallest disproof set has a non-empty intersection. We prove this stronger claim by induction.

Proof

- *Basis*
  For each frontier node J the singleton set containing J is both the only proof set, and the only disproof set. The intersection of these two sets contains node J and thus is not empty.

- *Induction step*
  Suppose that the assumption has been proved for all children J_1, .., J_n of an internal AND node J. To disprove J, only one child needs to be disproved. Let disp(J_x) be any disproof set of J_x which has minimal cardinality among all disproof sets of children of J. Then disp(J_x) is also a minimal disproof set of J. To prove J, all children must be proved. Let prove(J_i) ($1 \leq i \leq n$) be arbitrary minimal proof sets for each of the children J_i. Then $\bigcup_{i=1}^{n}$ prove(J_i) is a minimal proof set of J, which we name prove(J). Thus disp(J_x) is a minimal disproof set of J, and prove(J_x) is contained in a minimal proof set of J. As disp(J_x) and prove(J_x) are minimal disproof and proof sets of J_x, they have a non-empty intersection according to the induction assumption. Thus disp(J) and prove(J) have a non-empty intersection.

  The proof for internal OR nodes proceeds analogously.

□

We conclude that there is no conflict of strategies between trying to prove or to disprove the tree: by repeatedly selecting a most-proving node, both strategies are executed simultaneously, without one strategy delaying the other. How to select the most-proving node using proof and disproof numbers is illustrated with an example tree.

Below each node of the tree depicted in figure 2.3, we have depicted its proof number and disproof number (in that order). Thus, the least number of nodes which must be developed to prove the tree is 3. The same number of nodes is needed to disprove the tree.

First, let us analyze the effort necessary to disprove the tree. As node A is an OR node, it will only obtain value false if both children obtain value false. In other words, both children must be solved (with value false) to disprove the tree. Thus, in both subtrees frontier nodes exist which are part of the

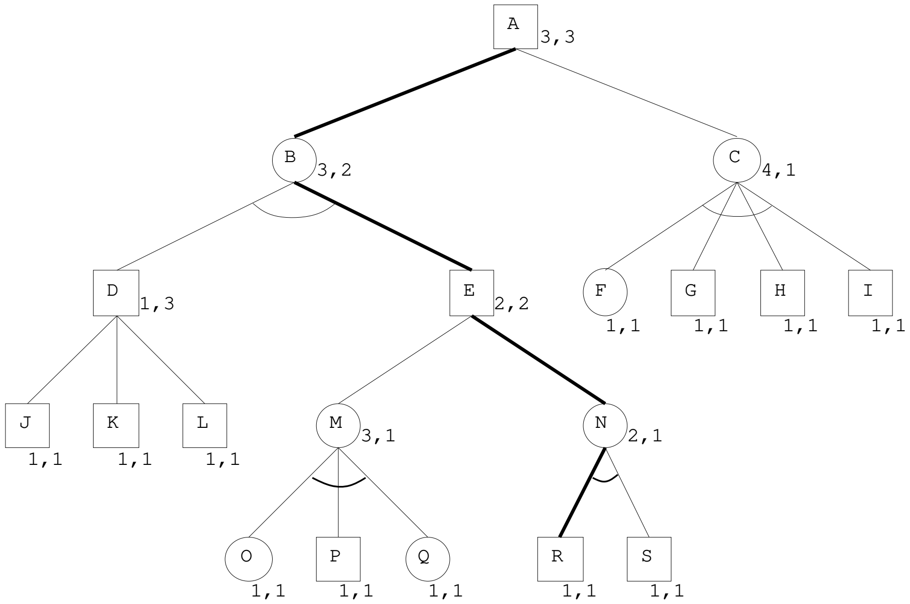

Figure 2.3: AND/OR tree with most-proving node R.

smallest disproof set of A. Second, let us look at the least number of node developments needed to prove the tree. For an OR node it is sufficient to have one child with value `true` to prove the OR node. In other words, only one child needs to be solved (with value `true`) to prove the tree. The proof number of child B (3) is one less than the proof number of child C (4). Thus, all frontier nodes of a smallest proof set lie within subtree B. We conclude that *all* most-proving nodes lie within subtree B.

With respect to subtree B an analogous analysis applies. However, since node B is an AND node, the roles of proof number and disproof number are interchanged. Thus, to prove B, both its children must be proved. Therefore, in both subtrees D and E, frontier nodes exist which are part of the smallest proof set of B. To disprove B, it is sufficient to disprove one child. Node E has disproof number 2, one less than disproof number 3 of node D. Thus, all frontier nodes of a smallest disproof set lie within subtree E. We conclude that *all* most-proving nodes lie within subtree E.

The selection within OR node E is based on the disproof numbers, as it was for node A, and thus subtree N is selected. Within AND node N no preference exists on the basis of the disproof numbers and both R and S

```
procedure ProofNumberSearch(root);
   Evaluate(root);
   SetProofAndDisproofNumbers(root);
   while root.proof ≠ 0 and root.disproof ≠ 0 and
          ResourcesAvailable() do
      mostProvingNode := SelectMostProving(root);
      DevelopNode(mostProvingNode);
      UpdateAncestors(mostProvingNode)
   od ;
   if root.proof = 0 then root.value := true
   elseif root.disproof = 0 then root.value := false
   else root.value := unknown
   fi
end
```

Table 2.1: Pn-search algorithm.

are most-proving nodes according to definition 2.5. In such cases we will, somewhat arbitrarily, always select the leftmost child. Thus, R is selected to be developed.

To summarize, the selection of a most-proving node is based on proof numbers among the children of OR nodes and on disproof numbers among the children of AND nodes.

2.2.4 The pn-search algorithm

In this section the algorithmic details of pn-search are presented in pseudocode, except for three domain-specific procedures and functions. In each of these three cases, the code for the implementation depends on the domain of investigation. The *goal* of each of these three, however, is domain-independent and has been specified below.

1. *Evaluate(node).* Assigns to node.value one of the values true, false and unknown.

2. *GenerateAllChildren(node).* Assigns to node.numberOfChildren the number of children of the node, and to node.children[1..node.numberOfChildren] (pointers to) the children themselves.

```
function SelectMostProving(node);
    while node.expanded do
        case node.type of
            or :
               i := 1;
                while node.children[i].proof ≠ node.proof do
                   i := i+1
                od
            and :
               i := 1;
                while node.children[i].disproof ≠ node.disproof do
                   i := i+1
                od
        esac ;
       node := node.children[i]
    od ;
    return node
end
```

Table 2.2: Most-proving node selection algorithm.

3. *ResourcesAvailable().* Returns a Boolean value which indicates whether sufficient resources are available to continue the search. This function will typically test the availability of working memory.

The algorithm of table 2.1 encodes the main loop of pn-search. The root of the tree is created and evaluated. Then, at each iteration, a most-proving node is selected and developed, followed by updating the proof and disproof numbers of the most-proving node and its ancestors.

The algorithm terminates when the tree is solved, or the program runs out of resources. We remark that there is a choice between implementing *immediate evaluation* and *delayed evaluation.* The main difference between these two methods is the amount of information available within trees of the same size: with immediate evaluation, all nodes in the tree have been evaluated, while with delayed evaluation the frontier nodes have not been evaluated. Due to the extra information, under the same working-memory limitations, immediate evaluation is more often able to solve a tree than delayed evaluation. In rare circumstances, however, delayed evaluation may be preferable. Examples of these circumstances include trees with a large

```
procedure SetProofAndDisproofNumbers(node);
    if node.expanded then
        case node.type of
            and :
                node.proof := Σ_{N∈Children(node)} N.proof;
                node.disproof := Min_{N∈Children(node)} N.disproof
            or :
                node.proof := Min_{N∈Children(node)} N.proof;
                node.disproof := Σ_{N∈Children(node)} N.disproof
        esac
    elseif node.evaluated then
        case node.value of
            false : node.proof := ∞; node.disproof := 0
            true : node.proof := 0; node.disproof := ∞
            unknown : node.proof := 1; node.disproof := 1
        esac
    else node.proof := 1; node.disproof := 1
    fi
end
```

Table 2.3: Proof and disproof numbers calculation algorithm.

variance in the branching factor, and slow evaluation. We have chosen to implement immediate evaluation, as it is used in all our applications of pn-search to games. Thus, all frontier nodes in the tree have been evaluated.

The algorithm of table 2.2 encodes the selection of a most-proving node, in accordance with the description in section 2.2.3. Thus, at OR nodes the child with lowest proof number is selected, while at AND nodes the child with lowest disproof number is selected. In case of a tie between children, the leftmost child is selected. Selecting the child with minimal proof number (in an OR node) or disproof number (in an AND node) is equivalent to selecting a child with proof number or disproof number equal to its father's. We remark that in most applications children will not be ordered by their proof or disproof number, as the cost of updating the ordering may be prohibitive. If the children are unordered, selecting the leftmost child with equal proof or disproof number on the average reduces the selection time of the most-proving node by at least a factor two, compared with determining the minimum over all children. A detailed discussion of enhancements to the algorithm can be

```
procedure DevelopNode(node);
   GenerateAllChildren(node);
   for i := 1 to node.numberOfChildren do
      Evaluate(node.children[i]);
      SetProofAndDisproofNumbers(node.children[i])
   od
end
```

Table 2.4: Node-development algorithm.

```
procedure UpdateAncestors(node);
   while node ≠ nil do
      SetProofAndDisproofNumbers(node);
      node := node.parent
   od
end
```

Table 2.5: Ancestor-updating algorithm.

found in section 2.3.

The algorithm of table 2.3 encodes the calculation of proof and disproof numbers for a given node. It is a direct translation into pseudo-code of the case-by-case observations made in section 2.2.2. We remark that "Σ" in the algorithm indicates that the sum is calculated over all children, while "Min" indicates that the minimum over all children is calculated.

The algorithm of table 2.4 encodes the development of a node. As stated before, we have implemented immediate evaluation.

The algorithm of table 2.5 updates the proof and disproof numbers of the most-proving node and its ancestors. This is necessary to ensure that all nodes in the tree correctly reflect the new situation after the development of the most-proving node. Starting from the most-proving node, the tree is traversed in the direction of the root, updating the proof and disproof numbers of each ancestor. After the proof and disproof numbers of the root have been updated, the algorithm is terminated (indicated by the fact that the root has no parent).

This concludes our formal description of pn-search.

2.3 Enhancements

In the previous section we have presented the pn-search algorithm. Several enhancements exist. Some of these should be applied in most practical circumstances, since the added performance outweighs the additional implementation effort. The advantage associated with the other enhancements depends on the domain of application. In section 2.3.1 we focus on enhancements reducing the amount of working memory needed to execute a search. Section 2.3.2 deals with reducing the execution time necessary to select the most-proving node and to update the proof and disproof numbers of the ancestors. The role of domain-specific knowledge when enhancing the algorithm is examined in section 2.3.3. Finally, transpositions are discussed in section 2.3.4.

2.3.1 Reducing memory requirements

Pn-search has working-memory requirements linear in the size (number of nodes) of the search tree. Depth-first search algorithms, such as α-β search, only require working memory linear in the depth of the search. As a result, working memory is a possible bottleneck when applying pn-search. We discuss two techniques to reduce memory requirements. The first technique is concerned with the removal of solved subtrees, while the second technique performs pn-search at two levels.

Deleting solved subtrees

A node in a pn-search tree may influence the search process in two ways:

1. it is (on the path to) the most-proving node;
2. its proof and disproof numbers influence the proof and disproof numbers of its parent.

Below, we show that solved subtrees do not influence the search process in either way, except that they may solve their parent immediately after they were solved themselves.

First, we show that a solved node is never on the path to the most-proving node. As long as the search is in progress the root is not solved. We thus start the selection of the most-proving node from an unsolved node. All unsolved nodes have finite proof and disproof numbers unequal to zero. Since at each step of the selection, a child is chosen with a proof or disproof number equal to that of its parent, each subsequent node must also be unsolved. We conclude that a solved node cannot be on the path to the most-proving node.

Second, we show that the proof and disproof numbers of a solved node either solves its parent, or does not influence its parent's values. A solved node with value `true` has proof number 0 and disproof number infinity. A parent OR node is solved by this child, and immediately obtains the value `true`. A parent AND node sums its children's proof numbers, to which the 0 does not contribute, while it minimizes its children's disproof numbers, to which infinity does not contribute. Only if this child were the last unsolved child is the AND node solved and obtains the value `true`. To a solved child with value `false` an analogous reasoning applies, with `false` and `true`, proof number and disproof number, and AND node and OR node interchanged.

We conclude that a solved subtree, once its parent has been updated, no longer influences the search, and thus may be removed. An efficient way to implement this enhancement in the *SetProofAndDisproofNumbers()* algorithm of table 2.3 is by deleting solved children when calculating the sum and minimum of the childrens' proof and disproof numbers.

For a discussion of the expected gain of this technique, we refer to section 2.4.

Pn^2-search

As a second technique to reduce memory requirements, we present a short description of a recent, so far unpublished, development in pn-search, named pn^2-search. The algorithm has been developed in collaboration with Stef Keetman.

Pn^2-search consists of two levels of pn-search. The first level consists of a pn-search (pn_1), which calls as evaluation of any node J a pn-search at the second level (pn_2), with a bound N on the maximum tree size. In pn^2-search N is chosen to be the current size of the pn_1 search tree. The second level of pn-search is a standard pn-search, with a normal (either standard or domain-specific) evaluation. The result of pn_2 on node J is the value `true` or `false` in case pn_2 solved J, or the proof and disproof numbers of J, if J has not been solved. In the latter case, the proof and disproof numbers are used to initialize J in pn_1. After termination of pn_2, its tree is removed from memory. We remark that several enhancements to pn^2-search have been suggested to reduce the overhead associated with recreating deleted parts of the tree. One example of such an enhancement involves storing the M last pn_2 trees in a cache, instead of deleting them, as suggested by Schaeffer (1994). The gain achieved by such enhancements is a topic of future research. Pn^2-search has the following properties.

1. A search resulting in a pn_1 tree of size N has searched approximately $\frac{1}{2} \cdot N^2$ nodes.

2. The memory requirements during the creation of a pn_1 tree of size N are approximately $2N$ nodes.

3. Implementing pn^2-search requires only minor changes to an implementation of standard pn-search

It has been established that the memory requirements of pn^2-search are on the order of the square root of the number of nodes investigated. Comparisons on awari and draughts have shown experimentally that pn^2-search investigates on the average three times as many nodes as standard pn-search to solve the same problem. This factor of three is independent of problem size within the range investigated.

Given the approximate constancy of this factor, it follows that in cases where pn-search is bounded by trees of 10^6 nodes, pn^2-search, with the same resources of memory may usefully investigate 10^{12} nodes. This conclusion can be extrapolated to even larger problems only when the factor of three suggested by the experiments remains constant. Whether it does and whether the extrapolation therefore remains valid, is a topic for future research.

2.3.2 Reducing execution time

The main difference in execution time between a best-first search algorithm, such as pn-search, and a depth-first search algorithm, such as α-β search, is the number of node traversals necessary to select the most-proving node. The overhead specific to pn-search is the calculation of proof and disproof numbers at internal nodes, being linear in the number of node traversals. The enhancement presented in this section reduces the number of node traversals per selection of the most-proving node. We remark that the same enhancement can be and has been applied to conspiracy-number search (Klingbeil, 1989).

At each iteration of pn-search we traverse the tree starting at the root and ending at the most-proving node. After developing the most-proving node, we follow the same path backwards until we are at the root. The basis of the enhancement consists of two observations.

- If the proof and disproof numbers of an ancestor do not change, the updating process can be terminated.
- If a node J is on the path from the root to the most-proving node, and J's proof and disproof numbers are not changed by the updating process, J also lies on the path from the root to the next most-proving node.

From these two observations it follows that at each iteration a node exists where we can terminate the updating process, and start the next most-proving node selection. Such a node is called the *current node*, which is defined as follows.

Definition 2.6 *For any* AND/OR *tree T, at any time during the execution of pn-search, the* current node *of T is defined as the ancestor of the previous most-proving node J, closest to J, which had no changes to its proof and disproof numbers caused by the development of J. Initially, the current node equals the root.*

Enhancing the pn-search algorithm to use the notion of current node changes the algorithms for *ProofNumberSearch* and *UpdateAncestors*. The new algorithms are shown in the tables 2.6 and 2.7.

The current-node enhancement reduces the number of node traversals per iteration from linear in the depth of the search tree to close to constant and should therefore be included in most practical implementations of pn-search.

We remark that at the cost of storing a most-proving node per subtree, the selection process can be changed into an instant most-proving node selection. Then, the most-proving nodes of the subtrees are updated during the updating of the proof and disproof numbers within the tree. Since the working memory is the main bottleneck in most applications, we feel that small gains in terms of processing speed do not warrant the extra space requirements.

2.3.3 Applying domain-specific knowledge

Two assumptions underly the formulation of the pn-search algorithm. First, the probability distribution of expected values of frontier nodes is equal throughout the tree. Second, the distribution of probabilities over the three evaluation values (true, false, unknown) is unknown. These two assumptions describe a situation in which no domain-specific knowledge can be applied to guide the search through the tree. In many practical domains, however, at

```
procedure ProofNumberSearch(root);
   Evaluate(root);
   SetProofAndDisproofNumbers(root);
   currentNode := root;
   while root.proof ≠ 0 and root.disproof ≠ 0 and
          ResourcesAvailable() do
      mostProvingNode := SelectMostProving(currentNode);
      ExpandNode(mostProvingNode);
      currentNode := UpdateAncestors(mostProvingNode)
   od ;
   if root.proof = 0 then root.value := true
   elseif root.disproof = 0 then root.value := false
   else root.value := unknown
   fi
end
```

Table 2.6: Pn-search algorithm (with current node).

least some knowledge is available. In this section we show how such knowledge can be applied to pn-search by altering the initialization of the proof and/or disproof numbers of frontier nodes.

We can view proof and disproof numbers as lower bounds on the effort necessary to solve a tree. So far, the effort has been measured in node developments. We consider three methods to use alternative measures of effort. First, we use the number of node *evaluations* as a measure of effort. Second, a domain-specific measure of effort is applied. Third, a function of the tree depth is used to influence the shape of the tree searched. Each method is illustrated using a particular game, being give-away chess, awari and go-moku, respectively. Finally, we review the three methods applied.

Evaluations as a measure of effort

A node development, when using immediate evaluation, consists of expanding the node and evaluating each of its children. Thus, the amount of effort involved in a node development depends on the number of children. We will use as J's proof number the least number of node evaluations necessary to prove node J and as its disproof number the least number of node evaluations necessary to disprove J. Let us assume that J will have n children when

```
function UpdateAncestors(node);
   changed := true ;
   while node ≠ nil and changed do
      oldProof := node.proof;
      oldDisproof := node.disproof;
      SetProofAndDisproofNumbers(node);
      changed := (oldProof ≠ node.proof) or
                 (oldDisproof ≠ node.disproof);
      previousNode := node;
      node := node.parent
   od
   return previousNode
end
```

Table 2.7: Ancestor updating algorithm (enhanced).

expanded. J's proof and disproof numbers can be initialized using that knowledge, even before J is expanded. If J is an OR node, only one child needs to evaluate to true to prove J, thus J's proof number equals 1. To disprove J, all n children must evaluate to false. J's disproof number is therefore initialized to n. For an AND node, the proof number is initialized to n, while the disproof number becomes 1.

The advantage of using the number of evaluations as a measure of effort is that a distinction between frontier nodes can be made which is not present in standard pn-search. It allows pn-search to focus on frontier nodes with fewer children before developing frontier nodes with more children. It is expected that in this way pn-search will find solutions more quickly. Below we present results from applying this method to give-away chess.

Give-away chess is a variant of chess where a player wins as soon as she cannot make a legal move (i.e., she has no pieces left or her remaining pieces are blocked). The pieces move as in chess, with two exceptions:

1. the king has no special status and can be captured like any other piece;
2. a player is forced to make a capture move if she can (like in checkers and draughts).

Castling and *en-passant* capturing are extremely rare in give-away chess. To simplify our implementation task, we have omitted these types of moves, thus rendering them illegal. In collaboration with Barney Pell we created the give-away chess program *Prove-away*, solely based on pn-search. A node evaluates to true, if white is to move and has no legal moves, while it evaluates to false if black is to move and has no legal moves. All other nodes evaluate to unknown. Pn-search was implemented in two variants, one variant using the standard initialization, and the other one using node evaluations as measures of effort.

To enable *Prove-away* to play games against opponents, it selects its moves by performing pn-search with a predetermined bound on the number of nodes to be created. If the tree is not solved within that limit, the 1-ply nodes are inspected and the move leading to a node with the minimal ratio of proof and disproof numbers is selected. If the tree is proved within the limit, the move proving the tree is selected, ensuring a win for *Prove-away*. If the tree is disproved, the 1-ply node with the largest subtree is selected, speculating on the opponent not seeing her winning line. Although we have no clear indication of the strength of *Prove-away*, it has beaten its human opponents in all but three of its games (out of several dozen). Most games are decided by *Prove-away* finding a winning line in which the opponent is forced at each move to capture one of the program's pieces, until the program runs out of moves and wins. The maximum depth of such lines in give-away chess is 32 ply (16 moves by the program and 16 captures by the opponent).

We conducted an experiment to compare the two variants of pn-search described above. During the experiment, *Prove-away* plays random games against itself. At each move in the game, both variants of pn-search (standard initialization and using evaluations as measure of effort) create a tree, with the current game position as root. As soon as one or both variants solve the tree, the game is terminated. If in a position neither variant solves the tree within 25,000 nodes, *Prove-away* plays a random legal move to continue the game. A total of 30 games were played, which lasted on the average 5.6 ply (i.e., a little less than three moves by white and three moves by black). Three games where duplicates of other games, due to the fact that the program quickly proved that black wins after the opening moves 1. d2-d4, 1. d2-d3 or 1. e2-e4, and each of these moves was selected twice as opening move during the 30 games. In the following we disregard the three duplicate games.

The conditions of the experiment ensure that the final position of each random game has been proved a win for one of the players by at least one of the pn-search variants. In some games, both variants proved the win, while in others only the pn-search variant with the number of evaluations as the measure of effort succeeded. In none of the games did only standard

	standard initialization	*initialization by no. of moves*	*improvement factor*
developments	5928	2661	2.2
nodes visited	62323	7838	8.0
branching factor	10.5	2.9	3.6
max tree size	48935	5988	8.2
nodes per sec.	169	132	0.8

Table 2.8: Give-away chess results.

pn-search solve the tree of the final position. To compare the performances of both algorithms, we reran the standard algorithm with unlimited working memory on the positions where that variant had not found the win within the limit of 25,000 nodes. The results of the experiment are presented in table 2.8.

Measured in number of node developments, the enhanced algorithm (using evaluations as measure of effort) gains a factor of a little over 2, while in number of nodes the improvement factor is almost 8. These numbers indicate that the enhanced algorithm develops nodes with, on average, a 4 times smaller branching factor (2.9 vs. 10.5). This clearly indicates that the selection of most-proving nodes is strongly influenced by the non-standard initialization. The average amount of working memory necessary to complete the search is specified in table 2.8 as the maximal tree in memory per search. It is directly related to the total size of the tree created, resulting in an improvement by a factor 8. The extra time spent on counting the number of moves per terminal node slows the algorithm down approximately 20% per node, compared to the standard initialization. Thus, the overall gain in CPU time amounts to a factor of more than 6.

We conclude that using the number of node evaluations as a measure of effort to initialize the proof and disproof numbers may yield a significant reduction in node evaluations, node developments and CPU time.

Domain-specific measures of effort

In many domains, domain-specific properties exist which give an indication of the amount of effort involved in solving a position (i.e., in solving the AND/OR tree with the position as root).

For instance, in othello solving a position with only a few empty squares is easier than solving a position with more empty squares. In draughts, it is simpler to solve a position if both players have only four men than if both players have ten men. In these cases, we could select as domain-specific measures of effort the number of moves to the end of the game (othello) or the number of men of the opponent to be captured (draughts and checkers).

We illustrate the idea on the game awari. In the initial awari position, there are 48 stones on the board. Both players move stones around, with the goal of capturing stones. The goal of awari is to capture more stones than your opponent. It follows that a player who has captured at least 25 out of the total of 48 stones, wins the game (for a definition of the rules of awari, see section 2.4.2). We use the number of stones a player needs to capture as the measure of effort. Let us assume that we would like to determine whether north can win, or whether south can obtain at least a draw. Let us furthermore assume that south has so far captured 11 stones, while north has collected 8 stones. We build the tree from the perspective of south, thus proving the tree means showing that south can reach at least a draw. In the given position, south must capture at least another 13 stones to reach a draw, while north needs another 17 stones to obtain the 25 stones necessary for a win. These values, 13 and 17, are then used as proof and disproof numbers of the position.

In section 2.4.7 we present test results of applying pn-search to awari for both the standard initialization and the stone-based initialization as suggested here.

Depth-related measures of effort

By inspecting trees created by pn-search, we have found some occasions in which the shape of the tree indicated that much effort was spent on variations which were less likely to succeed quickly than some others. For instance, in mating problems in chess, where the weaker side was restricted to moving one piece between two squares, most variations had proof number one. As a result, variations where the attacker moved a single piece aimlessly over the board were searched very deeply. On one occasion, this resulted in a mate in 114 moves being found, while a mate in 4 moves existed. Instead of putting a hard limit on the depth of the search, examining deep variations can be somewhat discouraged by initializing the proof and disproof numbers of a node using a function of the depth of the node.

By assigning higher proof and disproof numbers to nodes deeper in the tree, it is expected that pn-search will create a somewhat shallower and broader tree. Analogously, by assigning smaller proof and disproof numbers to nodes deeper in the tree, pn-search is expected to create deeper and narrower trees. Inspection of trees created by pn-search with such alternative proof-and-disproof-numbers initializations shows that the average node depth is indeed influenced in accordance with these expectations.

Experiments on go-moku (see chapter 5), with each node's proof and disproof numbers initialized to the depth of the node measured in full moves, show that a somewhat broader, shallower tree is created, without losing pn-search's ability to find narrow, deep variations leading to a win. Comparisons on go-moku showed that this initialization was an improvement over the standard initialization. The depth-related initialization was used in the search which led to solving go-moku.

Despite this example, we do not have much ground for the assumption that such an initialization is an enhancement to pn-search for domains with behavior similar to go-moku. Furthermore, the evaluation function we developed for go-moku also influenced the success of the non-standard initialization. Although a linear function in the depth of the node worked well in go-moku, more complicated functions may be necessary for other domains. The strongest conclusion we are prepared to draw is that by using a function of the depth of the node, the shape of the tree can be somewhat influenced (either made broader and shallower, or narrower and deeper).

Reviewing the application of domain-specific knowledge

We have presented three ways in which domain-specific knowledge can be used to change the initialization of the proof and disproof numbers at frontier nodes. Although each of the three methods has been successful in improving the performance in a practical domain, some caution is in order, particularly with the second and third methods. While the use of non-standard proof-and-disproof-numbers initializations may seem useful to guide the search process, the underlying principles of pn-search are violated. Two examples of violated principles are: (1) the assumption that all frontier nodes are indistinguishable and (2) the assumption that the proof and disproof numbers are lower bounds on the effort required to solve the tree. The positive influence of different initializations may at the same time result in negative effects. We have found that for some domains, such as othello, it is necessary to perform a large number of experiments to fine-tune the initialization process, akin to the process of fine-tuning evaluation functions in game-playing programs

(Gnodde, 1993). We conclude that as yet we lack a proper understanding of the precise effects associated with knowledge-driven proof-and-disproof-numbers initializations.

2.3.4 Transpositions

The definition of pn-search depends on the graph searched being a tree. When determining the proof and disproof numbers of an internal node J, the cardinality of the smallest proof set and disproof set must be determined. In a tree, the subtrees rooted at the children of J are disjoint, ensuring that the cardinality of the smallest proof set and disproof set of J can be calculated from the cardinality of the smallest proof sets and disproof sets of the children.

In many domains, however, the same subtree may be encountered several times during the search, at different places in the tree. The standard pn-search algorithm will in such cases obtain an upper bound on the cardinality of the smallest proof and disproof sets, instead of the true proof and disproof numbers. Problems and solutions related to the problem of the common subtree (transpositions) in combination with pn-search have been investigated by Schijf (1993) and Schijf *et al.* (1994).

In the following, we shortly describe problems and practical solutions for transpositions in pn-search. We distinguish between *directed acyclic graphs*, abbreviated as DAGs and *directed cyclic graphs*, abbreviated as DCGs. We remark that practical techniques for handling transpositions in game-playing programs using α-β search have been extensively described in the literature (Greenblatt *et al.*, 1967).

Transpositions in DAGs

Transpositions resulting in DAGs necessarily occur in games where each move is a *conversion*, i.e. an irreversible alteration of the state of the game. In chess, captures and pawn moves are examples of conversions, while non-capture moves by a piece (except for castling, and castling-forbidding moves) are non-conversions. In connect-four, qubic and go-moku, each move is a conversion, as in all three games the number of stones on the board strictly increases.

As stated above, in a DAG, addition of proof numbers or disproof numbers possibly overestimates the cardinality of the minimal set of nodes needed to solve the tree. Theoretically correct algorithms exist to establish the correct proof and disproof numbers at each node, but these are slow or use

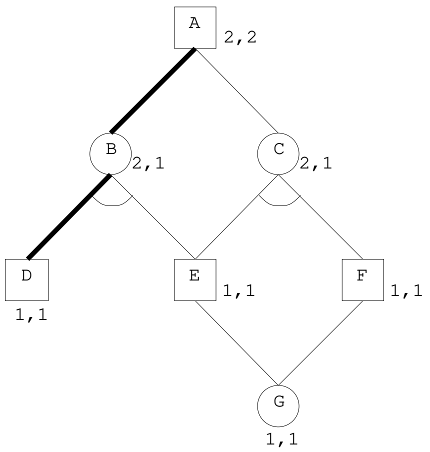

Figure 2.4: AND/OR DAG with practical solution.

an inordinate amount of working memory, or both, thus barring practical application (Schijf, 1993).

A practical solution to this problem is to treat the DAG as if it were a tree, thus calculating (incorrect) proof and disproof numbers of a node directly from its children. The main difference in the algorithm is that while updating ancestors, all parents of a node must be updated recursively. In figure 2.4, a DAG is depicted where proof and disproof numbers are calculated directly from their children. It can easily be shown that if node G is solved, root A obtains the same value as G. Thus, the proof and disproof numbers of A should equal 1. Furthermore, G should be the most-proving node. Thus, both numbers in the root are too high, and the selection mechanism incorrectly selects node D as the most-proving node. This example clearly indicates that the practical solution is no longer in accordance with the definitions of section 2.2.2. Still, our experience with connect-four, qubic and go-moku shows that this practical pn-search algorithm for DAGs has advantages similar to those of standard pn-search.

Transpositions in DCGs

Transpositions resulting in DCGs appear in games where a series of non-conversion moves leads to a position which has occurred before. Special rules govern the continuation of games after repetitions, leading by complex regulations to game-specific outcomes. There is fascination in the diversity

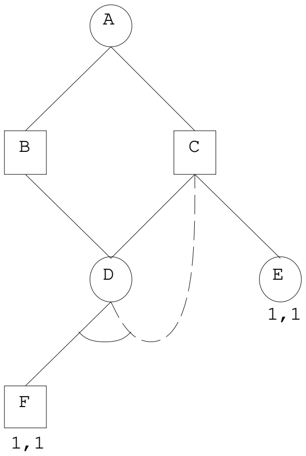

Figure 2.5: Cyclic AND/OR graph.

of these rules: in Chinese chess, some repetitions are illegal by the operation of complex rules; in go, any repetition is outlawed by the *ko* rule; in chess, finally, a repeated position can give rise to a claim of a draw from its third occurrence onwards.

Figure 2.5 depicts a DCG; in figure 2.6 we have converted that graph into a tree. Each path in the tree terminates at a frontier node of the graph, or at a repetition of positions in the path. In this example we assume that a repetition evaluates to false. The tree contains three duplicates of node D. Among these three, two have the value false, while one has proof number 2 and disproof number 1. The fact that the same node may have different proof and disproof numbers depending on the path it lies on forms the basis of the complexity of performing pn-search on DCGs. Node C has properties similar to node D. Moreover, we note that A's proof number (2) is less than the sum of the proof numbers of its children, as subtrees B and C have node E in common. The proof number at the root indicates that to prove the tree, both E and F must be proved. The disproof number 1 of A indicates that disproving either E or F disproves the tree.

The dependence of the proof and disproof numbers of a node on the path to that node forms the basis of the difficulties of cyclic transpositions. In Schijf (1993), a theoretically correct algorithm for pn-search on DCGs is described. Unfortunately, its time and working-memory requirements are too costly to warrant practical application.

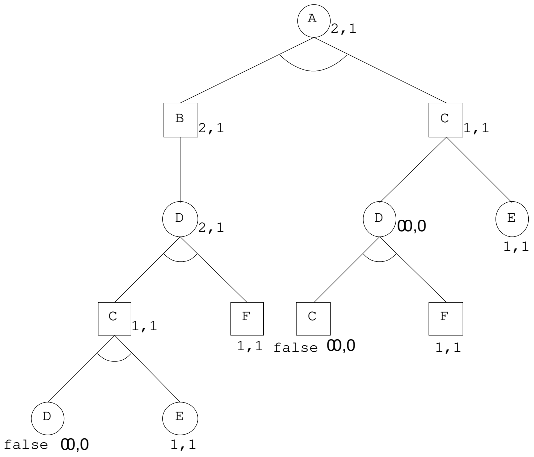

Figure 2.6: Tree version of the graph of figure 2.5.

Practical methods to apply pn-search to DCGs also exist. First, the practical algorithm for DAGs may be applied with one modification: only positions created after a conversion move are eligible to have more than one parent. As a result, some transpositions are investigated only once, while for others duplicates are created in the graph. Second, for each set of equivalent positions, at most two nodes are created: one for all paths in which the node occurs for the first time, and the second node when the node is its own ancestor. The second node is initialized to the value associated with a repetition of positions in the game under investigation. In this case, if a node is its own ancestor through at least one path, the repetition of positions is used to update the ancestors on all paths leading to the node, including those in which the node is not a repetition. Therefore, the search may incorrectly deduce that a node must have the value of a repetition of positions. Thus, if the value of the root is proved to equal the value assigned to repetitions of positions, the proof is not fully reliable. If the opposite value is proved, however, the proof is bound to be correct. For a detailed description of these two practical algorithms for pn-search in DCGs, we refer to Schijf (1993).

We believe that pn-search on directed cyclic graphs requires further investigation.

2.4 Results

2.4.1 Introduction

In this section we compare pn-search's performance with that of a sophisticated implementation of α-β search, by far the most commonly applied game-tree search algorithm in tournament programs for strategic games. As a test domain, we have selected the game of awari, one of the games on the Olympic List. We have chosen awari for two main reasons. First, awari search trees contain non-uniformity, which make them suitable for the application of pn-search. Second, all strong tournament programs competing in the Computer Olympiads selected their moves using sophisticated implementations of α-β search, establishing that awari search trees are suitable for application of α-β search.

It will be shown that, for the purpose of *proving* the game-theoretic value of a position in awari, pn-search outperforms α-β search by a wide margin. It proves that a category of search trees exists for which pn-search outperforms α-β. Further indications of pn-search's strengths can be found in chapters 4 and 5, where pn-search's contribution to solving qubic and go-moku is described.

This section is organized as follows. First, we present the rules of awari. Second, we give a description of the strongest existing awari programs, which presents evidence that our implementation of α-β search is competitive with α-β search implementations of other authors. Third, we describe in detail the implementations of pn-search and α-β search and their performances are compared. Fourth, it is explained how the nodes visited by both algorithms are counted, which is important due to the different nature of the algorithms. Fifth, we describe the set of awari positions to which the algorithms were applied. Finally, we present and analyze the empirical data.

2.4.2 The rules of awari

Awari is a two-player (south and north) zero-sum game with perfect information. It is one instance of a large family of games named mancala, of which some 1200 variants are known. The mancala games originate from Africa. Awari is mainly played in its western regions, such as Nigeria. For the game described here, the names wari or awele are also used (Deledicq and Popova, 1977).

Awari is played on a wooden board containing two rows of six pits. Each player controls the row on her side of the board. South's pits (from left to

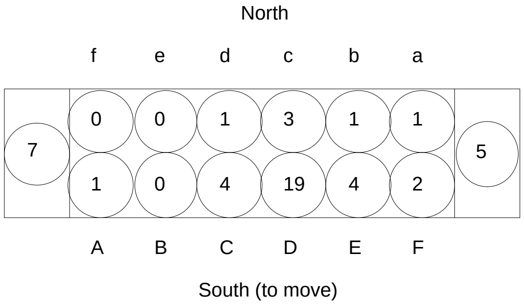

Figure 2.7: A position with legal moves $A1$, $C4 \times 2$, $D19 \times 7$, $E4$ and $F2 \times 4$.

right, as seen by south) are named A through F, while north's pits (from left to right, as seen by north) are named a through f. At the right-hand side of each row, an auxiliary pit is used to contain a player's captured stones. At the start of the game each pit (except the auxiliary pits) contains four stones, for a total of 48 stones on the board.

At each move, a player selects a non-empty pit X from her row. Starting with X's neighbor, she then sows all stones from X, one at the time, counter-clockwise over the board (omitting the two auxiliary pits). If X contains sufficient stones to go around the board (12 stones or more), pit X is skipped and sowing continues. Thus, after the move, X will always be empty. Finally, captured stones, if any, are removed and stored in the auxiliary pit. Stones are captured if the last stone sown lands in an enemy pit which after landing contains 2 or 3 stones. If such a capture is made, and the preceding pit contains 2 or 3 stones and the pit is an enemy pit, those stones are also captured. This procedure is successively repeated for the pits preceding and ends as soon as a pit is encountered containing a number of stones other than 2 or 3, or the end of the opposing row is reached.

A move is described by the name of the pit, followed by the number of stones sown (the name of the pit by itself defines the move, but such a notation is prone to error). The number of stones captured, if any, is indicated by the amount preceded by a "$\times$". In figure 2.7 an example position is shown with south to move. Legal moves for south are: $A1$, $C4 \times 2$, $D19 \times 7$, $E4$ and $F2 \times 4$.

The goal of awari is to capture more stones than the opponent. The game ends as soon as one of the players has collected 25 or more stones. Two other conditions exist which terminate the game. First, if a player is unable

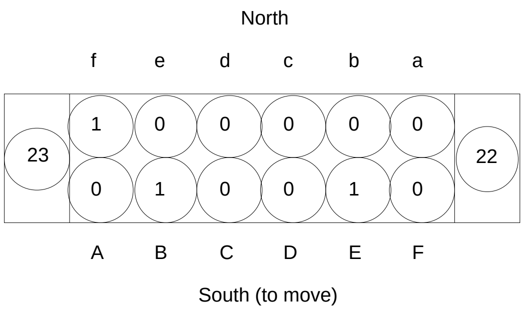

Figure 2.8: 1. $B1$ $f1$ wins. After 1. $E1$? $f1$ south must play 2. $F1$.

to move (i.e., all her pits are empty), the remaining stones are captured by her opponent. Second, if the same position is encountered for the third time, with the same player to move, the remaining stones on the board are evenly divided among the players. In all cases, after the end of the game, the winner is the player who captured the most stones. If both players capture 24 stones, the game is drawn.

A last rule exists to prevent players from running out of moves early in the game. Whenever possible, a player is forced to choose a move such that her opponent is able to make a reply move. It is, however, not compulsory to look several moves ahead to ensure that the opponent will continue to be able to reply. For instance, figure 2.8 shows a position in which south by playing *1. B1* can deliberately create a position in which she is unable to offer north any stones on her next move. By doing so, south captures all three stones remaining on the board and wins the game. However, would she have played 1. $E1$, then after 1. … $f1$ she is forced to play 2. $F1$, leaving the game for the moment undecided (although after 2. … $a1$ 3. $B1$ $b1$ 4. $C1$ $c1$ 5. $D1$ $d1$ 6. $A1$ $e1$ we are back at the initial position, giving south a second chance to play the winning move).

2.4.3 Tournament programs

Lithidion

In 1990 Maarten van der Meulen and the author constructed an awari-playing tournament program, named *Lithidion* (Greek for 'little stone'). *Lithidion* at the time consisted of an α-β search algorithm, and an endgame database containing the game-theoretic value of each awari position with 13 stones or fewer left on the board (Allis *et al.*, 1991c).

In 1991 *Lithidion* was enhanced with pn-search and a larger database (all positions with 17 stones or fewer). In 1992 *Lithidion* was further enhanced with an opening book. In describing *Lithidion*, we will concentrate on this last version of *Lithidion*.

The basis for *Lithidion* is its α-β search algorithm. Any position not in the opening book or the endgame database is searched with iterative-deepening α-β search. The evaluation function for leaf nodes is trivial: at each leaf node it is assumed that the players divide the remaining stones evenly. If, in the search tree, a position is encountered having 17 stones or fewer on the board, its exact value is retrieved from the endgame database. Thus, the value of the α-β search is based on a combination of crude guesses for some leaf nodes, and exact values for others (Beal, 1984). We remark that in awari the difference in the number of stones by which one wins is irrelevant. Therefore, the value retrieved from the endgame database is converted into $-\infty$ for losses, 0 for draws, and ∞ for wins. Once the game has progressed to a position contained in the endgame database, no search is needed, and at each turn the best move is played instantly.

After a move has been selected by α-β search (typically based on an 18-to-20 ply search), pn-search is called to check the move. If a proof can be found that the selected move loses, the move is rejected, α-β is asked to select a new move, and the procedure is repeated. If all moves are proved losses, the first move selected is played, hoping for an error by the opponent. While the opponent is pondering on the position, *Lithidion* performs pn-searches on her potential moves looking for wins. In case the opponent selects a losing move, *Lithidion* uses the proof by pn-search to select its winning move. The pn-search algorithm regards positions within the endgame database as terminal nodes, just as it treats positions where a player has no legal moves. All other positions are internal nodes. In summary: pn-search is only used to prevent *Lithidion* from playing losing moves and to detect winning lines after erroneous moves by the opponent. All other moves are based on α-β search.

Opponents

Lithidion has played in three tournaments: the awari tournaments of the 2nd, 3rd and 4th Computer Olympiads (London 1990, Maastricht 1991 and London 1992). *Lithidion* won the gold medal each time. The tournaments of the 2nd and 3rd Olympiads have been described in Levy and Beal (1991) and Van den Herik and Allis (1992).

In 1990, *Lithidion*'s only opponent, *Marco*, written by Remi Nierat, winner of the gold medal at the awari tournament of the 1st computer Olympiad, lost all its games. *Marco* is based on human-expert knowledge of awari, shallow α-β searches (averaging fewer than 10 ply) and no endgame databases. In most games, both *Marco* and *Lithidion* had prospects of winning, until *Lithidion*'s endgame database was reached. At that point *Marco* made one or more erroneous moves, leaving *Lithidion* with an easy win. In 1990, the main deciding factor was the endgame database (at that time, all positions of 13 stones or fewer).

In 1991, a new opponent appeared: *MyProgram* written by Eric van Riet Paap. *MyProgram* had been created using the published description of *Lithidion* (Allis *et al.*, 1991c). It contained a large endgame database (all positions of 16 stones or fewer), a fast implementation of α-β search including the singular-extension enhancement (Anantharaman *et al.*, 1989) and the same evaluation function as *Lithidion* (see above). *Lithidion* defeated *MyProgram* by the smallest possible margin, with three wins, two losses and one draw. In at least one of the games, pn-search played a decisive role, finding a deep winning line in a position unclear to α-β search. Given the small differences between the programs (a 17-stone database versus a 16-stone database, pn-search versus singular extensions, and *MyProgram* searching twice as many nodes per second), it is unclear what the exact impact of pn-search on the match has been.

In 1992, two new opponents appeared: *Marvin* and *Juju*. *Juju* turned out to be no competition for its two strong opponents and lost all its games. *Marvin* was created by Ralph Gasser with *Lithidion* as its example. The α-β search algorithms of *Marvin* and *Lithidion* performed almost equally well. *Marvin*'s endgame database (20 stones), however, was much larger than *Lithidion*'s (17 stones). A disadvantage to *Marvin* was that its database did not fit in RAM memory. Each entry retrieved from the hard disc slowed down the α-β search. Two further disadvantages to *Marvin* were its lack of a pn-search implementation and of an opening book. As a later test indicated, the opening book was the decisive factor in this match, which *Lithidion* won by a score of 4-2. The test consisted of replaying the first game from the position where *Lithidion* had exited its opening book, with *Marvin* and *Lithidion* changing places. *Marvin* easily won the game, similarly to the way *Lithidion* had won the game during the tournament. Clearly, the opening book had provided *Lithidion* with a winning advantage.

Conclusion

We have given a description of the architecture of *Lithidion*, the role of α-β search in it, and the competition it faced. From this description we conclude that *Lithidion*'s α-β-search implementation has been thoroughly tested and has performed well in competition with strong opponents. We stress this point, since *Lithidion*'s α-β search has been selected as the sparring partner for pn-search in our comparison tests on awari. Such a comparison is only valid if made against a sophisticated implementation, and we believe that practical evidence suggests that *Lithidion*'s α-β search meets those requirements.

2.4.4 The algorithms compared

For our experiments, we have compared two variants of α-β search, and two variants of pn-search. We will use the following abbreviations for the four algorithms:

α-β — α-β iterative-deepening search *without* transposition tables.

transposition — α-β iterative-deepening search *with* transposition tables.

basic pn — pn-search with standard initialization.

stones pn — pn-search with initialization based on the number of stones to be captured.

The **α-β** algorithm has the following characteristics. At each node, moves are pre-ordered by capture size. The largest captures are evaluated first, since the resultant positions are most likely to hit the database. Another reason for processing captures first is that they are often good moves. An iterative-deepening search is performed with a depth increase of 1 per iteration. The result of each iteration is a value and a move ordering of the full principal variation. The search terminates as soon as the value of the position has reached $-\infty$ or $+\infty$, indicating that the value of the position has been determined.

The **transposition** algorithm is the same as **α-β**, except that it is extended with a transposition table of a quarter of a million entries. The transposition table is implemented as a hash table, with one entry per hash code. At each node in the search tree, we first examine whether the position is present in the transposition table. Then we investigate whether the depth to which it had previously been searched is at least as large as the current depth. If both conditions are met, the range of possible values stored in the entry is used to narrow the α-β window. If after updating α exceeds or equals β, the search returns to the node's parent. Otherwise, the search is continued with the narrowed window.

After a node's value has been established, the results are stored in the transposition table. If the value of the node is equal to the initial α or β, we only know that the node's value is less or equal to α, or greater or equal to β, respectively. Only if the value lies between α and β proper, is the value reliable and can be stored as the true outcome of the search to the given depth. Values $-\infty$ and ∞ are treated separately, since these values are always indisputable. For those values, the searched depth is set to ∞ as well, since deeper searches cannot change a reliable value, making the information applicable to each following iteration. Collisions in the hash table are resolved in favor of the position which has been searched most deeply. We remark that unlike tournament chess programs, we store a full Gödel code per entry in the transposition table, ensuring that two different positions will not mistakenly be regarded as equal.

The transposition table is expected to be useful in awari in the middle and end games, when empty pits and pits containing single stones are common. A confirmation of this assumption will transpire from the results of our experiments presented in section 2.4.7.

Basic pn is the standard pn-search algorithm, enhanced with the technique which removes solved subtrees. Each frontier node is initialized to proof number 1 and disproof number 1.

Stones pn is equal to **basic pn**, except for the initialization of frontier nodes. Instead of proof and disproof numbers being initialized to 1, the number of stones still to be captured by a player to achieve her goal is used as the initialization, as explained in section 2.3.3. We remark that neither variant of pn-search uses transposition tables.

The **α-β** algorithm calculates approximately 10,000 nodes per second on a SUN SPARCstation 1+. The other three algorithms are roughly a factor two slower. For **transposition**, storing and retrieving information from the transposition tables is responsible for the slowed-down performance, while the pn-search variants have as extra overhead the creation and deletion of nodes, as well as the calculation of the proof and disproof numbers.

2.4.5 Comparing the performances

When selecting a search algorithm for an application, the elapsed CPU time is an important selection criterion. However, experimental results on tree searches when measured in CPU time are difficult to generalize, due to implementation details. Instead, it is customary to compare the number of nodes visited.

In this case, a careful analysis is needed to determine the fairest way to compare the number of nodes visited by α-β search and pn-search.

Let us consider the number of nodes visited by α-β iterative-deepening search. On the one hand, we could sum the number of nodes visited in each iteration. However, this would be unfair to α-β search, since a smaller number of iterations (e.g., by searching to even ply depths only) may result in almost the same ordering and thus reducing the number of nodes visited. On the other hand, we could just take the number of nodes visited in the last iteration. That would be unfair towards pn-search, as the last iteration does use the move ordering of previous iterations, and these searches should be included in the total node count somehow. Moreover, α-β search with transposition tables obtains many early cut-offs during the last iteration due to the solved subtrees stored in the transposition table.

Instead, we have chosen to count at iteration i only the nodes at depth i. Then the extra iterations are an asset to α-β search, without costing anything in terms of the number of nodes visited. Re-ordering of the moves may result in terminal nodes in a new iteration, which are not at the deepest level. These nodes are not counted at all. This slight bias in favor of α-β iterative-deepening search does not significantly influence the results.

For pn-search, we simply count the total number of nodes created during the search.

2.4.6 Test positions

As mentioned in section 2.4.3, *Lithidion* has taken part in three awari tournaments of Computer Olympiads. In total, she played 23 games (5 against *Marco* and 6 each against *MyProgram*, *Juju* and *Marvin*), of which two games were identical, which can be explained as follows. Each of the five programs described in section 2.4.3 plays deterministically. Therefore, before the next game against the same opponent, a change should be made in the opening choice of the program to avoid losing in exactly the same way. *Juju* forgot to do so once, and lost two games in identical fashion.

In the 22 different games a total of 1707 positions have occurred (from the initial position to the position after the last move had been played). Of these there were 1599 unique positions, which have been selected as the *initial* test positions.

For each of the initial test positions, a search with all four algorithms was performed. Since an awari game has three possible outcomes: win, draw and loss, and pn-search is a two-valued search algorithm, the three outcomes must be divided into two sets. We arbitrarily chose to treat a draw as equivalent to a loss for the player to move. Each of the searches has one of three possible outcomes:

- The player to move has a proved win.
- The opponent has at least a draw.
- The search ran out of resources.

Not all test positions can be used to compare the performance of the four algorithms. First, positions with 17 stones or fewer are solved immediately by all four algorithms through a single database lookup. Second, positions too early in the game are likely to be unsolvable by all four algorithms. Therefore, we have selected the relevant positions from the 1599 initial positions as follows. Each position has been investigated by all four algorithms with a resource limit of 500,000 nodes per position. If after 500,000 nodes the search had not succeeded, it was terminated. Using the outcome of the searches, the following selection was made. First, the 2 positions in which the game had just ended were discarded since all four algorithms solved the positions visiting only a single node. The reason why only 2 such positions were found out of 22 different games is that most games ended by resignation. Second, all positions with 17 stones or fewer (496 in total) were excluded. Third, all positions which were not solved by any of the algorithms (764 in total) were labeled unsolvable. The remaining 337 positions are named the *final* test positions.

We remark that in this way positions which are well suited for α-β search will be selected for the final test positions, as well as those positions well suited for pn-search. Thus, in our selection method of test positions there is no bias towards either of the algorithms.

Each of the algorithms which failed to solve one of the final test positions within the 500,000 nodes limit, was given virtually unlimited resources to try again. In practice this meant a limit of a quarter billion nodes per position for

α-β search, while for pn-search no final test position took more than about one and a half million nodes to solve.

2.4.7 Results

In this section we present the results of the comparison of the four algorithms described in section 2.4.4 on the 337 final test positions of section 2.4.6.

Each of the 337 final test positions was solved by **basic pn** and **stones pn**. Two positions were not solved by α-β within a quarter billion nodes, while there were two more positions not solved by both α-β and **transposition**. In this section we have set the solution size of unsolved positions at a quarter billion, which is a lower bound on the number of nodes necessary to solve them. Although this results in a bias in favor of α-β search, it does not influence our conclusions and it allows us to include the positions in the test results. Removing the positions from the final test set would be particularly unfair towards pn-search, as it would ignore its finest results.

First, we present figures indicating how often one algorithm outperformed another, without paying attention to the exact difference in node counts. Second, we tabulate the total number of nodes visited by each of the four algorithms, and calculate averages. Third, we group test positions by size of solution, and graphically depict the average difference in performance of the search algorithms per group.

Outperforming the other algorithms

In this section, we are interested in whether one algorithm performed better on a specific test position than another algorithm, but ignore the size of the difference. In our results we have divided the set of positions into two halves: the *easy* and the *hard* positions. To this end, we have sorted the positions according to the minimum number of nodes in which a position was solved. As a result, the 169 positions which were solved by at least one algorithm in fewer than 3200 nodes, were classified as easy positions, while the 168 positions with smallest solution larger than 3200 nodes were named the hard positions.

In table 2.9 we have listed for each algorithm how often it outperformed all other algorithms, separated for easy and hard positions. If two algorithms shared first place on a position, they were each awarded half a point.

As can be seen from table 2.9, at the easy positions there is hardly any difference between the α-β search algorithms (84 times best algorithm) and the pn-search algorithms (85 times best algorithm). For the hard positions

	α-β	transposition	basic pn	stones pn
easy	23	61	41	44
hard	0	22	$31\frac{1}{2}$	$114\frac{1}{2}$

Table 2.9: Number of times an algorithm performed best of all.

	α-β	transposition	basic pn	stones pn
α-β	-	35	79	$91\frac{1}{2}$
transposition	134	-	89	100
basic pn	90	80	-	82
stones pn	$77\frac{1}{2}$	69	87	-

Table 2.10: Comparing pairs of algorithms on easy positions.

the picture is entirely different: the pn-search variants are 146 times best, against just 22 times for the α-β search variants.

Table 2.10 shows per pair of algorithms, how often one algorithm outperformed the other, on the easy positions. Each entry at row R and column C in the table indicates how often the algorithm heading row R found a solution more quickly than the algorithm heading column C. The same information for the hard positions is displayed in table 2.11.

Table 2.10 indicates that **transposition** wins against the other three variants, albeit with a small margin compared with the two pn-search variants (89 against 80 and 100 against 69).

Table 2.11 clearly indicates that **α-β** has the worst performance of all four algorithms. It loses in all cases against **transposition**, and only 6 times outperforms the pn-search variants. **Transposition** occasionally does better than the pn-search variants, but is outperformed in more than 85% of all hard positions. Between the pn-search variants, the initialization based on the stones to be captured seems to pay off, given the 126 against 42 win compared with the standard initialization.

	α-β	transposition	basic pn	stones pn
α-β	-	0	6	6
transposition	168	-	25	24
basic pn	162	143	-	42
stones pn	162	144	126	-

Table 2.11: Comparing pairs of algorithms on hard positions.

	total nodes	average nodes	factor	tree size
α-β	2,437,035,522	7,231,559	128.8	-
transposition	1,285,839,816	3,815,548	68.0	-
basic pn	28,214,875	83,723	1.5	42,767
stones pn	18,918,032	56,136	1.0	25,505

Table 2.12: Test figures per algorithm.

Nodes visited

In this section we concentrate on the number of nodes visited by each algorithm.

In table 2.12 the first column of results lists the total number of nodes visited on the 337 test positions, per algorithm, while the second column contains the average per position. In the third column, the factor difference between each algorithm's average and the best average is presented. For both pn-search variants we have also determined the maximum number of nodes present in memory during each search. The average of these maxima have been listed in the last column of the table.

From table 2.12 a pattern similar to that seen in tables 2.10 and 2.11 becomes apparent: the pn-search variants perform best, with **stones pn** doing somewhat better than **basic pn**. With factors 68.0 and 128.8, both **α-β** and **transposition** are clearly outperformed.

The average maximum tree size in memory during the pn-searches, compared to the average solution size, indicates that removing solved subtrees during the search results in somewhat smaller memory requirements. Here approximately a factor 2 is gained. We remark that these figures only relate

	10^1	10^2	10^3	10^4	10^5	10^6	10^7	10^8	10^9
α-β	18	47	40	45	57	61	36	26	7
transposition	18	47	44	59	67	53	23	23	3
basic pn	14	43	52	77	92	57	2		
stones pn	14	37	57	77	101	51			

Table 2.13: Positions per group, per grouping algorithm.

to *solved* positions. In searches which are not successful, the number of solved subtrees is smaller, rendering the technique less effective.

Performance by size

Table 2.12 shows that pn-search is capable of outperforming α-β search by a large factor. The table does not indicate, however, to what extend the gain factor is related to the size of the search problems. Furthermore, we must realize that in the table the hard problems dominate the results.

Measuring the size of the search problems is not a straightforward task, since a position which is difficult to solve with α-β search may be rather simple for pn-search or vice versa. Therefore, we have grouped the test positions in four different ways, each time according to one of the algorithms applied in our experiments. We describe the grouping process based on α-β.

We have created groups for each power of 10. Thus, group i consists of all positions which were solved by α-β in more than 10^{i-1} nodes, and less than or equal to 10^i nodes. Within each group, the average number of nodes necessary to solve all positions in the group is calculated, for each of the four algorithms. These averages are then compared to see which algorithm performs best on positions of the size represented by the group.

In table 2.13 we have listed for each algorithm the number of positions per group, depending on the algorithm used as grouping criterion. These numbers indicate the size of each of the groups on which figures 2.9, 2.10, 2.11 and 2.12 are based.

Figures 2.9, 2.10, 2.11 and 2.12 contain the results per group, where the groups are created according to the solutions of α-β, **transposition**, **basic pn** and **stones pn**, respectively. For each figure, the numbers on the horizontal axis indicate the log_{10} of the size of the groups. The numbers on the vertical axis indicate the log_2 of the factor difference between the

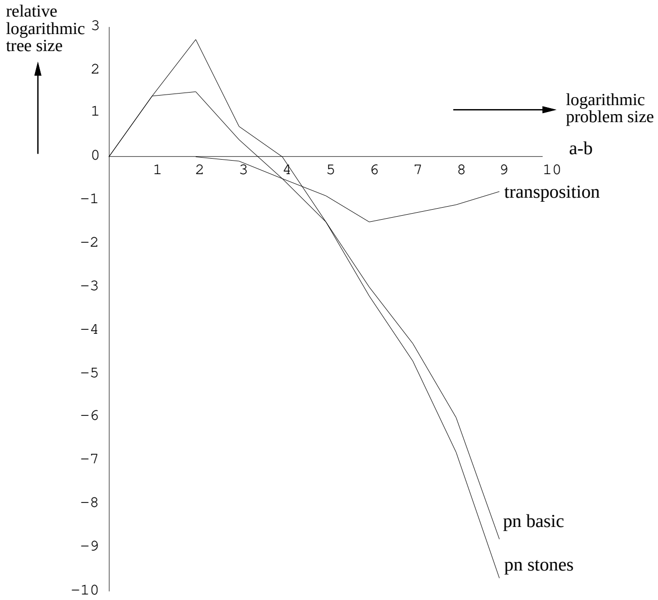

Figure 2.9: Comparison based on grouping by α-β

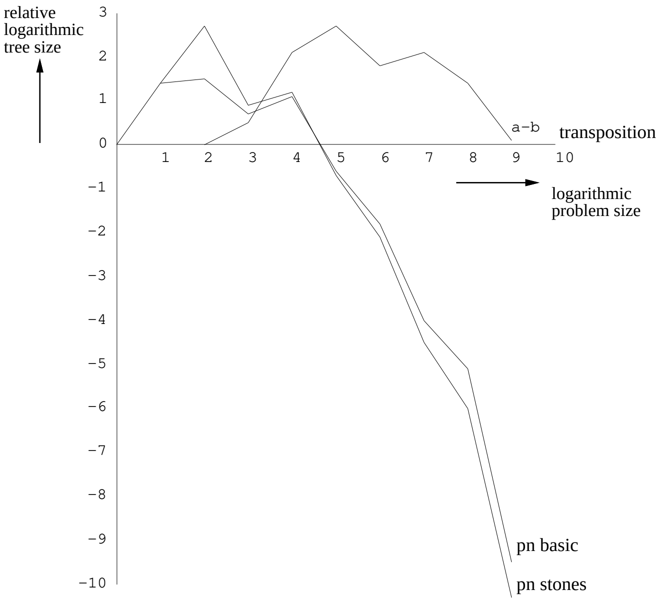

Figure 2.10: Comparison based on grouping by **transpositions**

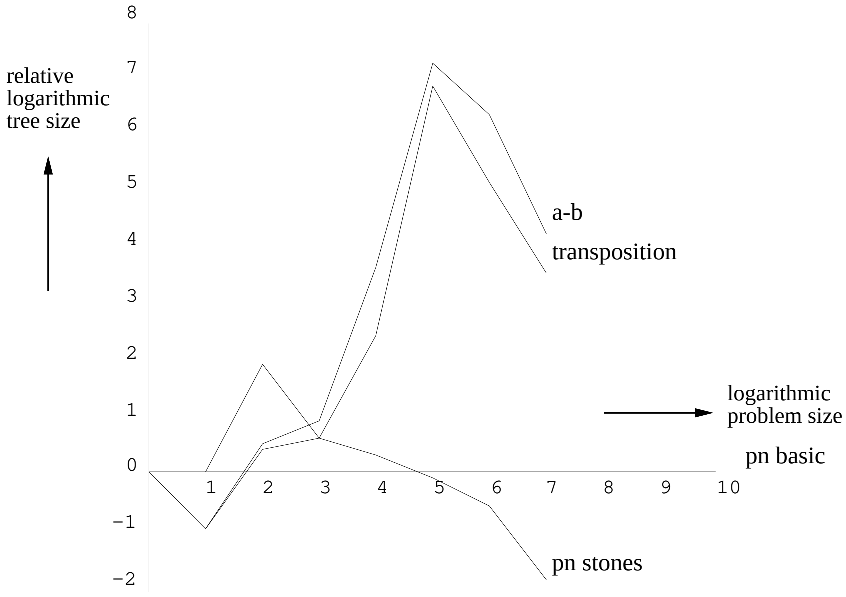

Figure 2.11: Comparison based on grouping by **basic pn**

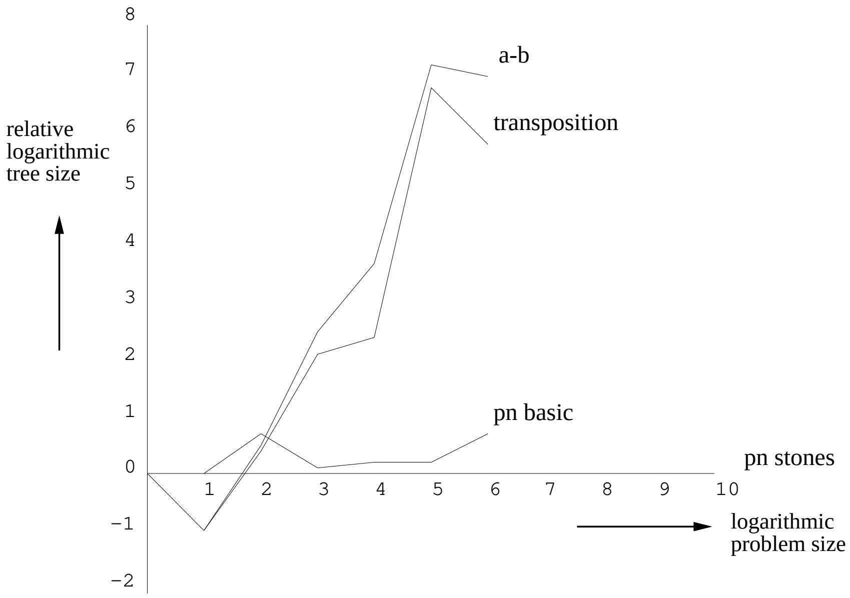

Figure 2.12: Comparison based on grouping by **stones pn**

10^1	10^2	10^3	10^4	10^5	10^6	10^7	10^8
15	43	46	65	75	57	35	1

Table 2.14: Positions per group, grouped by all four algorithms.

algorithms.

In figures 2.9 and 2.10 we see that on small problems α-β search does somewhat better, while with increasing problem size, pn-search does better and better. For the largest problems, the gain factor is around 500.

In figures 2.11 and 2.12, again pn-search does worse on the smallest problems and quickly starts doing better on increasing problem size. It is remarkable that the gain factor reduces when the problem size further increases. The cause of this phenomenon is described below.

In each figure the algorithm used as grouping criterion plays an important role. In the first few groups we find positions which were suitable for that type of algorithm, while in the last few groups the positions found were difficult to solve for the algorithm. It is thus to be expected that in the graphs the other algorithms will do somewhat worse in the first groups, while they do somewhat better on the last groups.

This is exactly what can be seen in all four graphs. In figures 2.9 and 2.10 pn-search outperform α-β search starting from group 4, while in figures 2.11 and 2.12 pn-search is the better algorithm from group 2 onwards. Furthermore, in the first two graphs pn-search's gain factor towards the last few groups grows remarkably fast, while in the second two graphs, with pn-search as the grouping criterion, pn-search's advantage reduces in the last two groups.

Thus, when looking at the groups for the hard problems, figures 2.9 and 2.10 are too flattering towards pn-search while figures 2.11 and 2.12 do not give pn-search full credit.

As a solution to this problem, we present one final graph. This time we have determined the size of a problem in a more elaborate way. For each solution by an algorithm, we determine the log_{10} of the number of nodes visited. For the four algorithms we then determine the average of these exponents and use it as group number. The number of positions per group has been tabulated in table 2.14. We average the logs since node counts tend to grow exponentially instead of linearly.

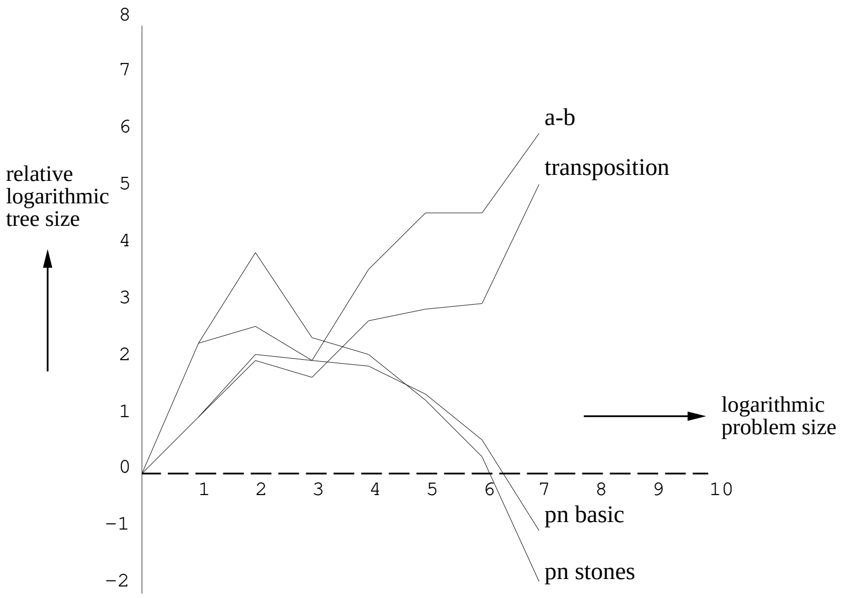

Figure 2.13: Comparison based on grouping using all four algorithms.

The singleton last group has been deleted, and its position has been added to the second last group, making a total of 36 entries in that group. The graph produced by this grouping criterion is pictured in figure 2.13. The numbers on the axes have the same meaning as in figures 2.9, 2.10, 2.11 and 2.12. In it, the bias towards a single algorithm no longer exists. The figure confirms the suggestion from the previous four figures, that pn-search's gain factor, compared with α-β search, grows with increasing problem size.

2.4.8 Conclusions

In this section we have compared the behavior of two pn-search variants with two variants of α-β search. The comparisons lead to clear conclusions: pn-search significantly outperforms both variants of α-β search (cf. table 2.12) in proving game-theoretic values in awari. The gain factor difference between pn-search and the α-β variants tends to increase with increasing problem size (cf. figure 2.13).

We further conclude from table 2.12 that a domain-dependent initialization can be beneficial on awari, with the enhancement yielding a profit of about a factor 2. Moreover, the removal of solved subtrees in pn-search decreases the working memory requirements by a factor of about 2, in problems which are ultimately solved.

We believe that the success of pn-search on awari is due to the non-uniformity of the tree. Allis *et al.* (1991b) have attempted to measure the degree of non-uniformity necessary for pn-search to outperform alternative algorithms. The results of this section show that awari's non-uniformity warrants the selection of pn-search for proving game-theoretic values instead of α-β search variants.

We tentatively conclude from these results that pn-search has contributed significantly to proving the game-theoretic values of other non-uniform trees, such as those of connect-four, qubic (see chapter 4) and go-moku (see chapter 5).

2.5 Related algorithms

In this chapter we have presented pn-search as an AND/OR tree search algorithm. Its roots, however, lie within the game-tree search algorithms. So far we have applied pn-search only to game trees, including awari, chess (Breuker *et al.*, 1994), connect-four (Allis, 1988), give-away chess, go-moku (see chapter 5), othello (Gnodde, 1993) and qubic (see chapter 4). In our discussion of related algorithms we will therefore focus mainly on game-tree search algorithms. In this section, we discuss the relationships with conspiracy-number search, SSS*, B* and A*, the latter being the only single-agent search algorithm in the list.

2.5.1 Conspiracy-number search

Conspiracy-number search (cn-search) is pn-search's direct ancestor. Cn-search was developed in the middle of the 1980s by McAllester, and has received attention of many researchers since then (McAllester, 1988; Klingbeil and Schaeffer, 1988; Klingbeil, 1989; Schaeffer, 1989; Van der Meulen, 1990; Allis *et al.*, 1991b; Lister and Schaeffer, 1994).

While pn-search focuses on the minimum number of nodes which must conspire to *prove* the value of a position, cn-search determines the minimum number of nodes which must conspire to *change* the value of a position. This main difference is more apparent when looking at the search tree: pn-search does not use a heuristic evaluation function to evaluate non-terminal nodes, while cn-search does.

Subtle differences between cn-search and pn-search can be identified by creating an instantiation of cn-search which resembles pn-search as much as possible. To do so, we define a three-valued evaluation function for cn-search, which returns -1 for a disproved node, 0 for a node with value unknown, and 1 for a proved node. In such a tree, the conspiracy numbers for -1 and 1 of a node correspond to the proof and disproof numbers of that node. These algorithms only differ in the manner in which the next node to be developed is selected, for which unpublished experiments on connect-four have shown that the selection mechanism of pn-search performs better than the selection mechanism of cn-search.

In cn-search, for any potential value v of the evaluation function it is determined for each subtree how many nodes, say N_v, within the subtree must change their evaluation value to v, to change the value of the subtree to v. If N_v for the root exceeds a certain limit, for all v unequal to the current root value, cn-search assumes that the current root value is reliable and terminates the search. Schaeffer's implementation showed that cn-search could achieve good results in tactical chess positions (Schaeffer, 1989). Unfortunately, experiments with tournament chess programs (Van der Meulen, 1990) have not been successful.

We remark that, despite pn-search's success in analyzing awari positions, we do not claim that pn-search is better suited than cn-search to perform well in a tournament chess program. Instead, we claim that the ideas behind cn-search, such as applied in pn-search, are better suited for *proving* values, than for determining the reliability of heuristic root values. Pn-search capitalizes on this suitability, concentrating on proving only. We do envision applications in tournament programs, as we have in our awari program. For instance, Breuker *et al.* (1994) have shown that pn-search may be an asset to chess programs, to prove quickly whether a mating sequence exists in a given chess position.

We conclude that cn-search and pn-search are closely related, with pn-search focusing on a different goal and being successful at it.

2.5.2 SSS*

With the availability of large internal memories, algorithms which store the search in working memory have become of practical interest. One of the earliest game-tree search algorithms which uses a stored tree is SSS* (Stockman, 1979; Campbell and Marsland, 1983).

SSS* and pn-search are both best-first search algorithms. At each step in the algorithm a node is selected according to a certain criterion and then developed. This process is repeated until the tree has been solved, or resources have run out. An important similarity between SSS* and pn-search is that neither algorithm uses a heuristic evaluation function for internal nodes. Only leaf nodes are assigned a value, either by a heuristic evaluation function or by reliable game knowledge.

The main difference between the algorithms is the criterion which determines the selection of the next node. SSS* selects a node purely based on the upper bound still achievable. At any point during the search the node which has the highest possible upper bound is selected, while among equals the leftmost node in the tree is preferred. Pn-search does not use a range of terminal-node values. Instead, the set of possible terminal-node values is split into two. Solving the tree means determining in which of the two sets the true value lies. If the exact value from a larger range must be determined, pn-search should be called repeatedly, for instance by having pn-search be the discriminating function in a binary search. While pn-search does not use a range of values, it bases its selection on the proof and disproof numbers implying that a node is tried which may be part of a solution with minimal effort.

A predecessor of pn-search, viz. $\alpha\beta$-cn search (Allis *et al.*, 1991b), can be seen as a hybrid form of pn-search and SSS*. It uses both a range of values *and* proof and disproof numbers (although these were named differently) to determine the next node to be developed. The main criterion is the range of possible values, like in SSS*, while in case of a tie the proof and disproof numbers are used. It can be shown, however, that trees exist with solutions of only a few nodes, in which $\alpha\beta$-cn search could spend a long time in irrelevant subtrees (Allis *et al.*, 1994). The solution to this problem consisted of reducing the impact of the range of values, while enlarging the role of the proof and disproof numbers. The result of this change has been the development of pn-search.

For a comparison of SSS* and $\alpha\beta$-cn search on random trees, see Allis *et al.* (1991b).

2.5.3 B*

B* is a best-first game-tree search algorithm introduced by Berliner (1979). It assumes that at each frontier node a special evaluation function returns a reliable lower and upper bound on the true value of the node. After a node is expanded, the lower and upper bounds of a node are calculated by maximizing (or minimizing, depending on the node type) the lower and upper bounds of its children. Let us assume that the root of the tree is a max node. Let us further assume that the root has two children A and B, with values in the intervals $[0, 2]$ and $[1, 3]$. B*'s main goal is to determine the best move, without necessarily knowing the exact *value* of such a move. In our example, B is the most-promising child of the root. Before we can terminate the search, however, we should either prove that the upper bound (2) on A's interval can be reduced to a value below the lower bound of B, which currently equals 1, or we should prove that the lower bound of B can be raised to at least the value of A's upper bound. These two different strategies are called *prove* and *disprove*.

Focusing both on proving and disproving is a similarity with pn-search. However, a difference with pn-search is that there is no way to simultaneously work on both strategies. Thus, in B*, at each step first a choice must be made for one of the strategies, followed by the selection of a node. Of course, after each node expansion, a change of strategies may take place. Since B* does not assume that some nodes may change their bounds more easily than others, we suggest that the concept of proof and disproof numbers could be a useful addition to B*.

An important prerequisite of B* is the *reliable* evaluation function which determines the lower and upper bound per node. Such an evaluation function heavily depends on domain-specific knowledge, and may be a serious obstacle in many domains. If, however, the knowledge to create such a function is readily available, B* provides a sound mechanism to incorporate it to guide the search process. An alternative way to obtain these bounds through a small search has been described by (Palay, 1982). For pn-search such a clear mechanism has not yet been formulated. In this respect B* has advantages above pn-search.

2.5.4 A*

A*, a single-agent search algorithm, has links with pn-search. A* is a best-first search algorithm, which uses an admissible evaluation function at each frontier node. Such a function calculates a lower bound on the total costs of the path from the root to a solution through that node. At each step a node with minimal lower bound on the solution costs is developed. A* thus guarantees finding an optimal solution (Hart *et al.*, 1968; Hart *et al.*, 1972; Nilsson, 1980).

Where A* concentrates on the cheapest *overall* solution, including the effort already spent (i.e., the cost of the path from root to frontier node), pn-search selects a node on the basis of the cheapest *remaining* solution, thus ignoring the contribution of already solved nodes and the path length from the root to the most-proving node. As a result, pn-search is not guaranteed to find the solution tree of minimal size.

Surprisingly, a small change to pn-search is sufficient to let it find the minimal solution tree. If, at each internal node, we add one to the proof number and disproof number as calculated from its children's proof and disproof numbers, then the proof number and disproof number at each node are a lower bound on the size of a solution tree for the node. We remark that proof and disproof numbers now can only increase, making some changes to the algorithm necessary. This algorithm, originating from discussions with Ingo Althöfer, has been named MST*, short for Minimal Solution-Tree search.

MST*, as variant of pn-search, will be subject of future research.