University of Alberta

Proof-Set Search

by

Martin Müller

Technical Report TR 01-09
May 2001

DEPARTMENT OF COMPUTING SCIENCE
University of Alberta
Edmonton, Alberta, Canada

Abstract

Victor Allis' proof-number search is a powerful best-first tree search method which can solve games by repeatedly expanding a most-proving node in the game tree. A well-known problem of proof-number search is that it does not account for the effect of transpositions. If the search builds a directed acyclic graph instead of a tree, the same node can be counted more than once, leading to incorrect proof and disproof numbers. While there are exact methods for computing proof numbers in DAG's, they are too slow to be practical.

Proof-set search (PSS) is a new search method which uses a similar value propagation scheme as proof-number search, but backs up proof and disproof *sets* instead of numbers. While the sets computed by proof-set search are not guaranteed to be of minimal size, they do provide provably tighter bounds than is possible with proof numbers.

The generalization *proof-set search with (P,D)-truncated node sets* or $PSS_{P,D}$ provides a well-controlled tradeoff between memory requirements and solution quality. Both proof-number search and proof-set search are shown to be special cases of $PSS_{P,D}$. Both PSS and $PSS_{P,D}$ can utilize heuristic initialization of leaf node costs, as has been proposed in the case of proof-number search by Allis.

1 Proof Sets and Proof Numbers

Victor Allis' *proof-number search (PNS)* [1] is a well-known game tree search algorithm, which has been successfully applied to games such as connect-four, qubic and gomoku. In contrast to many other methods, PNS does not compute a minimax value based on heuristic position evaluations; rather its aim is to find a proof or disproof of a partial boolean predicate P defined on a subset of game positions. The usual predicate is *CanWin(p)*, but predicates representing other goals such as the tactical capture of some playing piece can also be used with proof-number search.

Proof-number search is a best-first method for expanding a game tree. It computes proof and disproof numbers in order to find a *most-proving node*, which will be expanded next in the tree search. Search continues until the root is either proven or disproven.

There is a simple bottom-up backup scheme for computing proof numbers, which is correct for trees. However, many game-playing programs use a transposition table to detect identical positions reached by different move sequences. Such a table changes the search graph from a tree to a directed acyclic graph (DAG) or even a directed cyclic graph (DCG). If the same backup method for proof numbers is used on a DAG, it fails to compute the correct proof and disproof numbers, since the same node can be counted more that once along different paths. The new al-

gorithm of *proof-set search (PSS)* is designed to reduce this problem and thereby improve the search performance on game graphs containing many transpositions.

The outline of the paper is as follows: the introduction continues with a short description of proof-number search on game trees and on directed acyclic graphs, and with an example that illustrates the problems of proof-number computation in DAG's. Section 2 describes the new method of proof-set search, and characterizes it by a theorem establishing its dominance over PNS on the same DAG. On the other hand, counterexamples show that even PSS cannot always select a smallest proof set. Section 3 describes the algorithmic aspects of PSS in those areas where it differs from PNS. Section 4 introduces the data structure of a K*-truncated node set*, defines the generalization of PSS to *PSS with (P,D)-truncated node sets* or $PSS_{P,D}$, characterizes both PNS and PSS as special cases of $PSS_{P,D}$, and proves a generalized dominance theorem of $PSS_{P,D}$ over PNS. Section 5 describes how to use a heuristic initialization of leaf node costs in PSS and $PSS_{P,D}$. Section 6 closes with a discussion of future work, including the extension of PSS to cyclic game graphs and potential applications of PSS.

1.1 Proof-number Search in a Tree

This introductory section describes the basic procedure of proof-number search. For detailed explanations and algorithms, see [1]. Proof-number search (PNS) grows a game tree by incrementally expanding a *most-proving node* at the frontier. Nodes in a proof tree can have three possible states: *proven*, *disproven*, and *unproven*. Search continues as long as the status of the root is *unproven*. After each expansion, a leaf evaluation predicate P is applied to each new node, to see whether it is defined in the corresponding game position. If yes, the new node can be evaluated as *proven (P = true)* or *disproven (P = false)*, while if P does not apply, the node status becomes *unproven*. Proofs and disproofs are propagated to interior nodes by using proof numbers. Interior nodes are proven by finding a *proof tree* or disproven by finding a *disproof tree*. A proof tree s for a node r is a subtree of the game tree with following properties:

1. r is the root of s.
2. In all leaf nodes of s, the predicate P is well-defined and evaluates to true.
3. If n is an AND node in s, then *all* of its successor nodes in the game tree are also contained in s.

Analogous properties hold for a disproof tree s of r:

1. r is the root of s.

2. In all leaf nodes of s, the predicate P is well-defined and evaluates to false.

3. If n is an OR node in s, then *all* of its successor nodes in the game tree are also contained in s.

PNS maintains proof and disproof numbers for each node in a game tree, and updates them after each expansion. These numbers can be interpreted as the size of a minimal proof or disproof set: a smallest set of currently unproven terminal nodes of the tree with the property that (dis)proving all nodes in that set would create a (dis)proof tree for the root node. An impossible (dis)proof is represented by an infinite (dis)proof number.

Proof and disproof numbers are used to select a *most-proving node* to expand next. They are computed by a simple backup scheme. Each search node n stores a proof number $pn(n)$ and a disproof number $dn(n)$. For an *unproven* frontier (or leaf) node n, set $pn(n) = dn(n) = 1$. A *proven* frontier node is assigned $pn(n) = 0, dn(n) = \infty$, while a *disproven* node obtains $pn(n) = \infty, dn(n) = 0$. For non-frontier or interior nodes, let the children of a node n be $n_1, \ldots, n_k$. The backup rules for proof and disproof numbers are as follows: In an AND node, the proof number is computed as the sum of the proof numbers of the children. In an OR node, the proof number becomes the minimum among the proof numbers of all children.

$$\text{AND node: } pn(n) = pn(n_1) + pn(n_2) + \ldots + pn(n_k)$$

$$\text{OR node: } pn(n) = \min(pn(n_1), pn(n_2), \ldots, pn(n_k))$$

Disproof numbers are computed by taking sums at OR nodes and minima at AND nodes.

$$\text{OR node: } dn(n) = dn(n_1) + dn(n_2) + \ldots + dn(n_k)$$

$$\text{AND node: } dn(n) = \min(dn(n_1), dn(n_2), \ldots, dn(n_k))$$

The most-proving node to expand next can be found by traversing the graph from the root to a frontier node, selecting a child with identical proof number at OR nodes and a child with identical disproof number at AND nodes. After expanding a node, proof and disproof numbers of that node must be recomputed, and changes propagate upwards in the tree.

1.1.1 Heuristic Initialization of Proof Numbers

Proof numbers can be viewed as a lower bound on the work required to prove a node. The standard algorithm assigns the same estimate of 1 to each unproven leaf

node. However, game-specific knowledge can be used to provide different initial estimates. Allis [1] proposes to use a heuristic initialization $h(n)$ for the proof and disproof numbers of new frontier nodes n. Such a heuristic initialization can be viewed as a heuristic lower bound estimate of the work required to prove n. Using the constant function $h(n) = 1$ yields the standard algorithm.

1.2 Proof-number Search in a Directed Acyclic Graph

The basic algorithm for proof-number search on a tree can also be used for directed acyclic game graphs (DAG) with some small modifications. However, proof numbers can overestimate the size of proof sets. Therefore the algorithm cannot always find a minimal proof set, and it can fail to identify a most-proving node, even though such a node always exists even in a DAG [6, 1]. The following example shows how tree backup in a DAG overestimates proof numbers.

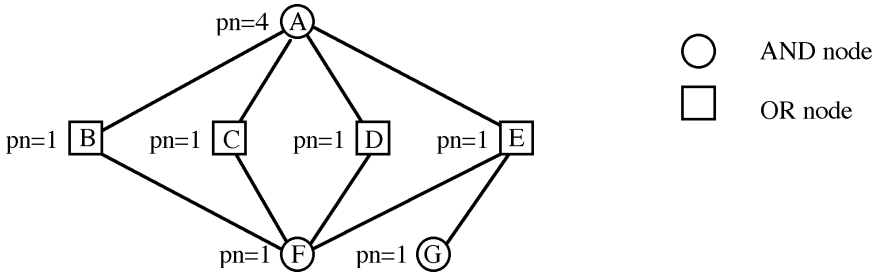

Figure 1: Overestimating proof numbers in a DAG

Example 1 *Consider a DAG with an AND node A at the top and 4 OR nodes B, C, D, E as children. Furthermore, let B, C and D have the same single child F, and let E's two children be F and G.*

The proof numbers of the leaves F and G are initialized to 1. The proof numbers of $B \ldots E$ are also all equal to 1 since these are OR nodes, which obtain the minimum proof number of their children, and they all have a child with proof number 1 but no child with proof number 0. The proof number of A is greatly overestimated by the tree-backup scheme as $pn(A) = pn(B)+pn(C)+pn(D)+pn(E) = 4$. The true value of $pn(A)$ is 1, since proving the single frontier node F proves A, but F is counted four times by the algorithm. The example can easily be extended, to make the difference between true and computed values arbitrarily large, by adding further nodes at the OR level which are only connected to A above and to F below. If such a DAG occurs as part of a larger problem, the overestimate of A's proof number can be very costly. It can greatly delay the expansion of the

sub-DAG below A, and lead the search into different parts of the DAG for a long time, even though F is a very good candidate node. Expanding F could lead to a quick proof of F and thereby A. This may be by far the fastest - or even the only - way to solve the overall problem.

To overcome the problem of overestimation, Schijf [6, 7] has developed exact methods for computing proof numbers in DAG's and identifying a most-proving node. Unfortunately, these methods seem to have a huge computational overhead and have turned out to be impractical even in tests on small Tic-Tac-Toe game DAG's. The new method of *proof-set search* reported here lies in between proof-number search and Schijf's exact method, both in terms of complexity and solution quality. Furthermore, by using *truncated node sets*, the tradeoff between informedness and memory overhead of proof-set search can be controlled precisely.

2 Proof-Set Search

Proof-set search, or *PSS*, is a new search method which uses the simple children-to-parents propagation scheme for DAG's. Instead of using proof numbers, which are an upper bound on the size of the minimal proof sets in a DAG, PSS backs up *proof sets* directly. While the sets selected by PSS cannot be guaranteed to be minimal, they provide provably tighter bounds than is possible with proof numbers only. This can lead to a better node selection and thereby to a smaller (dis)proof DAG being generated. The prize for better approximation is that more memory and time per expansion step is needed to store and propagate sets of nodes instead of numbers.

2.1 Backup Algorithm for Proof Sets

The algorithms for proof-set search are similar to the ones for proof-number search [1]. Each search node n stores both a proof set $pset(n)$ and a disproof set $dset(n)$. For unproven frontier nodes, set $pset(n) = dset(n) = \{n\}$. An impossible (dis)proof is indicated by a set of infinite size, represented by the symbol $\{\}_\infty$. A proved frontier node is assigned $pset(n) = \emptyset, dset(n) = \{\}_\infty$, while a disproved node obtains $pset(n) = \{\}_\infty, dset(n) = \emptyset$. For interior nodes n, with children $n_1, \ldots, n_k$, the backup rules for proof and disproof sets are as follows: In an AND node, the proof set is defined to be the *union* of the proof sets of all children. In an OR node, the proof set is the minimal set among the proof sets of all children, computed by a function set-min according to some total ordering of node sets.

$$\text{AND node: } pset(n) = pset(n_1) \cup pset(n_2) \cup \ldots \cup pset(n_k)$$

OR node: $pset(n) =$ set-min$(pset(n_1), pset(n_2), \ldots, pset(n_k))$

Disproof sets are backed up analogously, taking minima in AND nodes and unions in OR nodes.

OR node: $dset(n) = dset(n_1) \cup dset(n_2) \cup \ldots \cup dset(n_k)$

AND node: $dset(n) =$ set-min$(dset(n_1), dset(n_2), \ldots, dset(n_k))$

The following rules define a simple total order on node sets, which is close to the spirit of the original PNS:

1. A smaller set is always preferred to a larger one.
2. To break ties between sets of the same size, first define a total ordering of single nodes. For example, the ordering given by a depth first traversal of the DAG, the order of node expansion, or even the memory address of a node can be chosen.

As notation, let $n_1 < n_2$ if n_1 precedes n_2 in the chosen total order on single nodes. Sort each set $s = \{n_1, \ldots, n_k\}$ such that $n_1 < n_2 < \ldots < n_k$. Then a total ordering of sets of nodes can be defined as follows:

1. $s_1 < s_2$ if $|s_1| < |s_2|$.
2. $s_1 < s_2$ if $|s_1| = |s_2|$ and s_1 precedes s_2 in lexicographical ordering. In other words, given two sorted sets of equal size $s_1 = \{n_1, \ldots, n_k\}$ and $s_2 = \{m_1, \ldots, m_k\}$, $s_1 < s_2$ iff there is an $i, 1 \leq i \leq k$, such that $n_j = m_j$ for all $j, 1 \leq j < i$, and $n_i < m_i$.

The function set-min is obtained from this ordering by setting set-min$(a, b) = a \Leftrightarrow a \leq b$. Another possible ordering, using node evaluation as a heuristic measure of proof effort [1], is given in Section 5. Set operations are extended to $\{\}_\infty$ in the natural way, by $s \cup \{\}_\infty = \{\}_\infty$ and set-min$(s, \{\}_\infty) = s$ for all finite sets s, and by $\{\}_\infty \cup \{\}_\infty = \{\}_\infty$ and set-min$(\{\}_\infty, \{\}_\infty) = \{\}_\infty$.

Example 2 *See Figure 2. Assume that the nodes of Example 1 are ordered $A < B < C < D < E < F < G$, and that tie-breaks among sets of the same size are resolved by a lexicographical ordering of the sorted lists of elements, so that for example $\{F, E, D\} = \{D, E, F\} < \{D, E, G\}$. Then the computation of proof and disproof sets proceeds as follows:*

1. $pset(F) = dset(F) = \{F\}$, $pset(G) = dset(G) = \{G\}$.

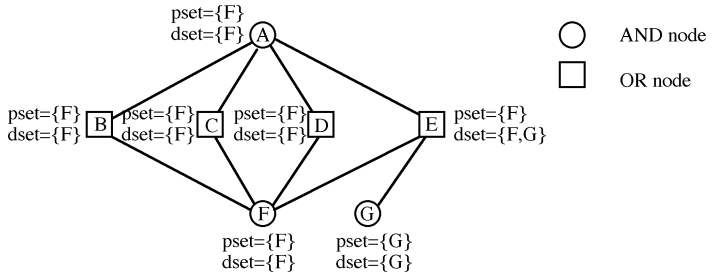

Figure 2: Computing proof and disproof sets in a DAG

2. $pset(B) =$ *set-min*$(pset(F)) = \{F\}$. $dset(B) = \bigcup(dset(F)) = \{F\}$.

3. *In the same way,* $pset(C) = pset(D) = \{F\}$ *and* $dset(C) = dset(D) = \{F\}$.

4. $pset(E) =$ *set-min*$(\{F\}, \{G\}) = \{F\}$ *and* $dset(E) = \{F\} \cup \{G\} = \{F, G\}$.

5. $pset(A) = \bigcup(pset(B), pset(C), pset(D), pset(E)) = \{F\}$.

6. $dset(A) =$ *set-min*$(dset(B) \ldots dset(E)) =$ *set-min*$(\{F\}, \{F, G\}) = \{F\}$.

By using proof sets instead of proof numbers, the root's proof set has cardinality 1, whereas the proof number computed by PNS was 4. So with proof sets, the promising node F *is likely to be expanded much earlier.*

2.2 Identifying a Most-Promising Node

In proof-number search on trees, a *most-proving node* always exists, and it can easily be identified. (Dis)proving a most-proving node reduces (dis)proof number of the root by at least one.

Schijf [6] proves the existence of a most-proving node in a DAG. However, it is not easy to identify such a node. Schijf develops theoretically correct but impractically slow algorithms to identify a most-proving node. On the other hand, since PSS uses a locally greedy heuristic to select a minimal set, it can not always identify a most-proving node. However, PSS computes a *most-promising node (mpn)* which lies in the intersection of the proof and disproof sets computed for the root node by an efficient PNS-like backup scheme. The following theorem states that such a most-promising node always exists.

Theorem 1 *Given a DAG G, compute proof and disproof sets using PSS. Then for each unproven node n, the intersection of proof and disproof set, $pset(n) \cap dset(n)$, is nonempty.*

The induction proof follows the lines of the analogous theorem for proof numbers in [6]:

1. The theorem holds for all unproven leaf nodes n since $pset(n) = dset(n) = \{n\}$.

2. Let n be an unproven AND node with unproven children $\{n_1, \ldots, n_k\}, k > 0$. Assume the induction hypothesis holds for all these children. Let n_i be a node such that $dset(n) = dset(n_i)$. Since $pset(n_i) \cap dset(n_i)$ is nonempty by the induction assumption, let x be a node in the intersection. Then $x \in dset(n)$ and since $pset(n) \supseteq pset(n_i)$, it follows that $x \in pset(n)$.

3. The proof for OR nodes is obtained by interchanging the roles of $pset$ and $dset$ in step 2.

2.3 Dominance of Proof-Set Search over Proof-Number Search

As a more informed search method, proof-set search dominates proof-number search in the following sense:

Theorem 2 *For each node n in a DAG G, the size of the proof (disproof) set computed by PSS is less than, or equal to the proof (disproof) number of n computed by proof-number search on G.*

$$|pset(n)| \leq pn(n)$$
$$|dset(n)| \leq dn(n)$$

We omit a proof here, since Theorem 2 is a special case of the more general Theorem 6, which will be proven in Section 5.

Note that the result holds only for PSS and PNS operating on the same DAG. The theorem cannot compare nodes in the two different DAG's which are generated when PNS and PSS respectively are used as the algorithm to select most-promising nodes for expansion. Since these two DAG's are generated by different mechanisms, their shape and their node sets may be completely different, and are therefore not comparable.

2.4 Proof-Set Search Does Not Always Select a Smallest Proof Set

As mentioned in Section 2.2, PSS uses a locally greedy heuristic to select a minimum set among all unproven children. A local method cannot always choose a set that will perform best when taking unions with other sets further up in the DAG. In the worst case, PSS can do no better than PNS. In the following two examples, PSS fails to find a smallest proof set.

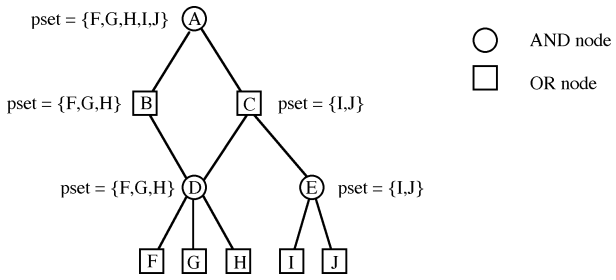

Figure 3: PSS fails to find a smallest proof set

Example 3 *In Figure 3, it is easy to see that $\{F, G, H\}$ is the minimal proof set for the root A. This set is necessary to prove node D, which in turn is needed to prove B, which is required to prove A. On the other hand, proving D also proves the only other child of A, node C, so a complete proof of A is obtained by proving $\{F, G, H\}$.*

At node C, PSS' locally greedy selection is the wrong choice: given the proof sets of C's children, $pset(D) = \{F, G, H\}$ and $pset(E) = \{I, J\}$, PSS selects the smaller set $\{I, J\}$ as the minimal proof set for C. Because of this choice, the proof set of A becomes $pset(A) = pset(B) \cup pset(C) = \{F, G, H, I, J\}$, which is almost twice as large as the optimum.

Example 4 *The ratio between the real optimum and the set computed by PSS can be made arbitrarily large by repeating a similar construction. In Figure 4, the DAG of Figure 3 has been extended to the top and right by nodes $S \ldots Z$. $\{F, G, H\}$ is still a smallest proof set for node A. Applying the same argument as above for nodes $S - T - U - A$, it can be seen that $\{F, G, H\}$ is also a smallest proof set for node S. However, as in Example 3 PSS computes a proof set of size five for node A, and therefore selects the right branch at node U, with a four-element proof set $\{W, X, Y, Z\}$. The proof set backed up to the root contains all nine leaf nodes, whereas the smallest set has only three nodes. Repeating this construction n times yields a DAG for which PSS computes a proof set of size $2^{n+1} + 1$ containing all leaf nodes, while the size of the smallest proof set is 3.*

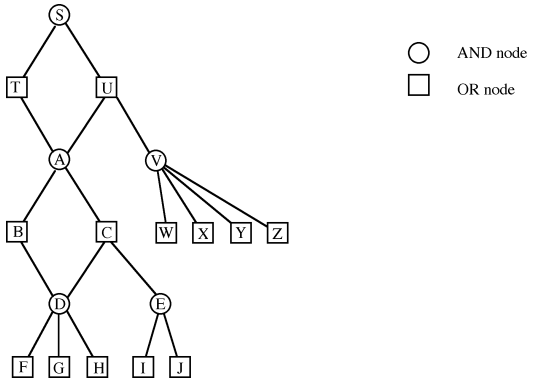

Figure 4: Example 4: extending Example 3

When computing proof sets incrementally, it is possible to take the proof sets of other siblings into account for the minimum selection. Section 5.2 discusses such a variation of the algorithm, which tries to improve the likelihood of selecting a set that works together well with sets from other siblings.

3 Algorithmic Aspects of Proof-Set Search

This section describes some algorithmic aspects of proof-set search, especially in areas where it differs from proof-number search, such as the selection of a most-promising node, different resource requirements and the representation of sets of nodes. It also discusses the problem of multiple updates of the same node. This problem already exists in PNS, but is more severe for PSS since the update cost per node is larger.

3.1 Selecting a Most-Promising Node

Selecting a most-promising node in proof-set search is extremely simple. Since PSS represents proof and disproof sets directly, it is not necessary to traverse the DAG as in PNS. Theorem 1 guarantees that the intersection of proof and disproof sets of the root is nonempty, and any node in the intersection is a suitable most-promising node.

3.2 Ancestor Updating Algorithm

This subsection discusses some problems of value propagation in DAG's. It applies to proof-number search as well as PSS. An ancestor updating algorithm must update all children of a node before the node itself can be computed. In a tree, a simple bottom-up computation suffices. On the other extreme, in a directed *cyclic* graph (DCG) a 'right' ordering of nodes does not exist because of cyclic dependencies. Special methods have been developed for proof-number search in DCG's [2], but it is unclear how they relate to PSS. This is a topic for future research, see the discussion in Section 6.1.

In a DAG, the right order of updates can be assured by a topological sorting of nodes. However, dynamically maintaining such a sorted order may be expensive, and in practice it may be preferable to accept multiple updates of some nodes instead.

The simplest updating algorithm for DAG's, modeled after that for trees, starts with the just developed node *mpn*, then adds its predecessors to a queue [6]. For node sets, such an algorithm can be written as follows:

```
UpdateAncestorSets(mpn)
{  ListOf<Node> updateQ = [mpn]; // start with just expanded mpn
   while (updateQ.NonEmpty())
   {
      node = updateQ.Pop(); // extract first node from queue
      NodeSet oldPS = node->PS(), oldDS = node->DS();
      node->SetProofAndDisproofSets(); // backup from children
      if ((oldPS == node->PS()) && (oldDS == node->DS()))
      {} // unchanged, no need to propagate
      else // update parents
         updateQ.Union(node->Parents());
         // append new parents to the end of the queue
   }
} // UpdateAncestorSets
```

In the general case, this method does not guarantee a perfect order of updates. Some nodes might be updated more than once, as the following example shows.

Example 5 *In the DAG of Figure 5, there is a direct move from $A - D$, but there is also a longer path $A - B - C - D$ between the same nodes. Assume that parents of a node are added to the queue in left-to-right order. Then after updating D, A is appended to the queue before C, so A will be updated first. However, this is a useless update, since A is also an ancestor of C via B. After updating C and B, A is re-added to the queue and updated once more.*

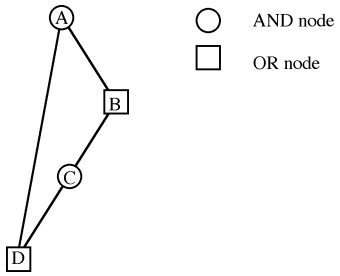

Figure 5: Queue update algorithm causes multiple updates of the same node

As the example shows, the queue backup algorithm can cause multiple updates of nodes in a DAG, which slows down the computation. There are two choices: either accept the inefficiency caused by multiple updates of the same node, or keep the nodes sorted in a priority queue, using a topological sorting of the DAG. This can be accomplished by an algorithm such as *Topological_Order* in [5, p.137], which enumerates nodes from the root downwards. We have not yet implemented such an ordering method. It is unclear whether an efficient incremental version of the algorithm exists, which can update the ordering after each node expansion. The problem is at least as hard as cycle detection in directed graphs, since computing an ordering is possible exactly for cycle-free graphs, and the algorithm must be able to complain if no ordering is possible because of a cycle.

3.2.1 A Sufficient Condition for Optimality of the Queue Backup Algorithm

The ancestor relation defines a partial ordering of the nodes in the DAG. If this partial order has a special structure, the optimality of the queue backup algorithm is assured. A *rank function* r [8, p.99] on a partially ordered set is a function mapping elements to integers such that $r(y) = r(x) + 1$ whenever y *covers* x (immediately follows x) in the partial order. In our case, that means that $r(c) = r(n) + 1$ for all children c of a node n. For example, for all the DAG's in Figures 1 - 4 a rank function exists, while for the DAG in Figure 5 there is none.

Lemma 1 *Consider the partially ordered set $P(G, Anc)$ given by the nodes in a DAG G and the ancestor relation Anc on G. If a rank function r exists for $P(G, Anc)$, then the queue backup algorithm is optimal: it computes each node value at most once.*

Proof of Lemma 1: The queue backup algorithm processes nodes in order of their rank: if x is inserted into the queue before y, then $r(x) \geq r(y)$. Actually, the

following stronger statement will be proven: At each stage of the algorithm, the ranks of nodes in the queue are in monotonically decreasing order and assume at most two distinct values v and $v-1$. In other words, if the queue $q = [n_1, \ldots, n_k]$ contains $k > 0$ elements, then there exists j, $1 \leq j \leq k$, such that $r(n_i) = r(n_1)$ for $1 \leq i \leq j$ and $r(n_i) = r(n_1) - 1$ for $j+1 \leq i \leq k$. It is easy to see that this property is an invariant maintained by the algorithm: Initially, the queue contains only a single element. Removing the element n_1 of rank $r(n_1)$ from the head of the queue maintains the invariant. The only nodes appended to the end of the queue by the algorithm are parents of n_1, which all have rank $r(n_1) - 1$ by definition of r.

Examples of games for which a rank function exists are all those where a move adds exactly one stone to the game state and does not remove anything, such as Othello (except for forced pass moves), connect-4, qubic, gomoku or Tic-Tac-Toe. The number of stones on the board is a rank function for positions in such games. In contrast, games with loops such as chess, shogi or Go do not have a rank function, unless the whole move history is taken into account for defining a position.

3.2.2 Comparing the Ancestor Updating Algorithms of PNS and PSS

Both PNS and PSS can stop propagating values to ancestors as soon as a node's value does not change. However, updates in PSS are certain to propagate all the way to the root, since the root's proof and disproof sets contain the just expanded *mpn*, which is no longer a leaf node. Usually, PSS will have to update more ancestors than PNS, since it distinguishes between sets of the same size with different elements. However, because of transpositions, the opposite case can also happen. If a node which is already contained in a proof set is re-added along a new path, the proof number increases but the proof set remains unchanged.

4 Truncated Node Sets for Proof-Set Search

This section describes a *truncated node set* data type with bounded memory requirements per set, and uses it in the algorithm $PSS_{P,D}$, PSS with (P,D)-truncated node sets. We prove theorems that characterize both PNS and PSS as extreme cases of $PSS_{P,D}$.

Set union and assignment operations on large node sets are expensive, both in terms of memory and computation time. The new data structure of a *K-truncated node set* provides a compromise between the two extremes of using a number and using a node set of unbounded size. A truncated node set stores at most K nodes explicitly. In addition, it stores an upper bound on the overall set size, in the same sense that proof numbers represent an upper bound on the size of proof sets. In this

sense, proof numbers can be regarded as 0-truncated node sets, which store only a bound but no elements.

Definition 1 *Let K be a nonnegative integer, and let set-min be a function computing the minimum of two sets based on a total order of nodes. A* K-truncated node set s *is a pair* $(rep(s), bound(s))$*, where* $rep(s)$ *is a set of nodes of cardinality at most* K *and* $bound(s)$ *is a nonnegative integer. In addition, a node set has the following properties:*

- $bound(s) < K \Leftrightarrow |rep(s)| = bound(s)$
- $bound(s) \geq K \Leftrightarrow |rep(s)| = K$

Definition 2 *The operations of minimum selection and set union for truncated node sets are defined as follows:*

- $bound(a \cup b) = bound(a) + bound(b) - |rep(a) \cap rep(b)|$
- $rep(a \cup b) = x$*, where x is the smallest* K*-element subset of* $rep(a) \cup rep(b)$ *according to set-min.*
- $min(a, b) = a \Leftrightarrow bound(a) < bound(b) \vee (bound(a) = bound(b) \wedge set - min(rep(a), rep(b)) = rep(a))$

In other words:

- *Minimum selection: If the bounds of two sets are different, the set with the smaller bound is the minimum. Otherwise, the explicitly represented parts of the sets are compared as if they were unbounded sets.*
- *Set union: The truncated set union computes the union of the explicitly represented sets, and stores the first* K *elements according to some total ordering of nodes, plus the best-possible bound for the size of the union.*
- *Initialization by a single element: A truncated node set* s *is initialized to store a single node* n *as follows:* $s = (\{n\}, 1)$ *if* $K > 0$*,* $s = (\{\}, 1)$ *if* $K = 0$*.*

Example 6 *Let* $K = 8$*, let nodes be represented by letters ordered alphabetically, let* $s_1 = \{A, D, E, H, K, L, M, Q\}_{16}$*,* $s_2 = \{C, D, E, F, H, M, P, Q\}_{13}$*. For each truncated set,* K *elements are given explicitly, and the subscript represents the bound on the set size.* s_1 *represents a set of at most 16 elements, including the eight listed. The truncated set union* $s_1 \cup s_2 = \{A, C, D, E, F, H, K, L\}_{24}$ *is computed by truncating* $rep(s_1) \cup rep(s_2) = \{A, C, D, E, F, H, K, L, M, P, Q\}$

to the smallest $K = 8$ elements. The bound on the set union size is computed by adding the bounds, then subtracting the double-counted elements in $rep(s_1) \cap rep(s_2)$, $16 + 13 - 5 = 24$.

As before, infinite size sets $\{\}_\infty$ are added along with rules for computing minima and unions involving such sets.

4.1 Some Properties of Truncated Node Sets

The next lemma formalizes the intuitively clear fact that larger truncation thresholds result in tighter bounds for set unions.

Lemma 2 *Given two integers $K > L$, and sets of nodes $s_1 \ldots s_n$, compute both the $K-$truncated and the $L-$truncated union of $s_1 \cup s_2 \cup \ldots \cup s_n$, using the truncated set method with the same node ordering and the same sequence of two-set union operations. Then the bound on the $K-$truncated union is smaller-or-equal than that on the $L-$truncated union. Furthermore, the explicitly represented set of the $L-$truncated union is a subset of the explicitly represented set of the $K-$truncated union.*

The proof is straightforward from the definition of truncated set union and is omitted here to save space. It is worth noting that since truncating sets loses information, some properties of the usual set union operations are lost. For example, the absorption law $a \cup a = a$ does no longer hold, since two sets that look identical may contain different nonrepresented nodes.

Example 7 *In the extreme case of 0-truncated sets, computing the truncated set union is equivalent to adding the bounds for both sets, as in $\{\}_5 + \{\}_5 = \{\}_{10}$.*

Example 8 *Let $a = a_1 = a_2 = \{A, B\}_5$ be 2-truncated sets. Then it would be wrong to set $a_1 \cup a_2 = a$, since the two sets might contain up to three different elements. The correct result is $a_1 \cup a_2 = \{A, B\}_{5+5-2} = \{A, B\}_8$.*

4.2 Proof-Set Search with Truncated Node Sets

Given two integers P and D, the algorithm *proof-set search with (P,D)-truncated node sets*, $PSS_{P,D}$, is obtained from standard PSS by replacing all proof sets with P-truncated node sets, and all disproof sets with D-truncated node sets.

4.2.1 Selecting a Most-Promising Node

Selection of a most-promising node in PSS relies on Theorem 1 of Section 2.2, which guarantees that for each unproven node n, $pset(n) \cap dset(n) \neq \emptyset$. With truncated node sets, a most-promising node may not always be found immediately, since all the elements in the intersection might have been cut off by truncation. The algorithm for selecting a most-promising node with truncated node sets is an intermediate form of the respective algorithms in PNS and PSS.

```
SelectMPN() // find most-promising node with truncated node sets
{
   node = root; mpn = NIL;
   while (IsInteriorNode(node))
   {
      // CommonNode(a,b) returns NIL if no common node
      // is found in rep(a) and rep(b).
      mpn = CommonNode(pset(node), dset(node));
      if (mpn != NIL) // found a node in intersection, done
         return mpn;
      SetType t = 'disproof' if 'node' is AND node,
                  'proof' if 'node' is OR node;
      node = child with same set of type t as node;
   }
   return node; // reached a leaf node, done.
} // SelectMPN
```

4.2.2 Characterizing PSS and PNS as Special Cases of $PSS_{P,D}$

Theorem 3 *$PSS_{\infty,\infty}$ is the same algorithm as standard PSS.*

Theorem 4 *$PSS_{0,0}$ is the same algorithm as proof-number search.*

The proofs follow immediately from the facts that an ∞-truncated node set is an ordinary untruncated node set and a 0-truncated node set is equivalent to a proof number. Two other special cases are interesting, and may be appropriate parameter choices if search behavior is highly biased towards only proofs or only disproofs: $PSS_{\infty,0}$ combines proof sets with disproof numbers, while $PSS_{0,\infty}$ uses proof numbers together with disproof sets.

A nice property of $PSS_{P,D}$ is that if $P < \infty$ and $D < \infty$, then the required memory remains bounded by a constant factor of what PNS would use on the same DAG.

4.2.3 Dominance Theorem of PSS with Truncated Node Sets

Theorem 5 *Let G be a DAG and let $P \geq 0$ and $D \geq 0$ be integers. Then at each node $n \in G$, the size bounds for proof and disproof sets computed by $PSS_{P,D}$ are smaller-or-equal to the proof and disproof numbers computed by PNS.*

$$bound(pset(n)) \leq pn(n)$$
$$bound(dset(n)) \leq dn(n)$$

The proof is by induction: the theorem holds for single nodes, and remains true when taking truncated set unions or selecting minima in interior nodes.

1. The theorem holds for all proven, disproven and unproven leaf nodes by definition of $PSS_{P,D}$ and PNS.
2. Assume $bound(pset(n_i)) \leq pn(n_i)$ for all children n_i of node n.
3. Set union: By Definition 2,

$$bound(pset(n)) = bound(pset(n_1) \cup \ldots \cup pset(n_k))$$
$$\leq bound(pset(n_1)) + \ldots + bound(pset(n_k))$$
$$\leq pn(n_1) + \ldots + pn(n_k) = pn(n).$$

4. Minimum selection: By definition,

$$pn(n) = min(pn(n_1), \ldots, pn(n_k)) \text{ and}$$
$$bound(pset(n)) = min(bound(pset(n_1)), \ldots, bound(pset(n_k))).$$

   Assume the minimal proof number is achieved in node n_i. Then

$$pn(n) = pn(n_i) \geq bound(pset(n_i)$$
$$\geq min(bound(pset(n_1)), \ldots, bound(pset(n_k))) = bound(pset(n)).$$

5. The proof for disproof sets is the same as for proof sets.

Theorem 5 generalizes Theorem 2 of Section 2.3, which dealt with the extreme case $PSS_{\infty,\infty}$. As in Theorem 2, the dominance holds only when comparing the computation of the algorithms on the same DAG G. It does not hold, and is not even meaningful, for the different DAG's generated by using $PSS_{P,D}$ and PNS respectively to generate the DAG.

5 Using PSS with a Heuristic Leaf Evaluation Function

In analogy to the refinement of PNS described in Section 1.1.1, PSS can utilize a heuristic node initialization function h for new frontier nodes. For simplicity, we will use the same function symbol h for both the proof and the disproof case. In practice, two different initialization functions will be used, since proofs and disproofs are opposite goals.

For a node set $s = \{n_1, \ldots, n_k\}$, define the heuristic weight by $h(s) = \sum h(n_i)$, and set $h(\{\}_\infty) = \infty$. In PSS, modify the minimum selection among sets as follows: set-min$(s_1, s_2) = s_1$ if $h(s_1) < h(s_2)$. If $h(s_1) = h(s_2)$, break the tie according to a secondary criterion such as lexicographical ordering. Combining heuristic initialization with truncated node sets is also relatively straightforward and will be described in Section 5.1.

Theorem 6 *Let G be a DAG, and let a positive heuristic leaf node initialization function h be defined for each game position represented by a node in G. Furthermore, extend h to sets of nodes by $h(s) = \sum_{n \in s} h(n)$. Then for each unproven node $n \in G$, the evaluation of the proof (disproof) sets computed by PSS is less-or-equal the proof (disproof) number of n computed by PNS with the same leaf node initialization function h.*

$$h(pset(n)) \leq pn(n)$$
$$h(dset(n)) \leq dn(n)$$

Proof: by induction.

1. The theorem holds for all leaf nodes n since $pset(n) = dset(n) = \{n\}$ and $h(pset(n)) = h(dset(n)) = h(n) = pn(n) = dn(n)$.

2. Let n be an unproven AND node with (unproven) children $\{n_1, \ldots, n_k\}$. Assume that the induction hypothesis holds for all children n_i, $h(pset(n_i)) \leq pn(n_i)$.

$$\begin{aligned} h(pset(n)) &= h(pset(n_1) \cup pset(n_2) \cup \ldots \cup pset(n_k)) \\ &\leq h(pset(n_1)) + h(pset(n_2)) + \ldots + h(pset(n_k)) \\ &\leq pn(n_1) + pn(n_2) + \ldots + pn(n_k) = pn(n). \end{aligned}$$

3. Let n be an OR node with children $\{n_1, \ldots, n_k\}$, and again assume the induction hypothesis holds for the children.

$$h(pset(n)) = \min(h(pset(n_1)), h(pset(n_2)), \ldots, h(pset(n_k)))$$

$$\leq \min(pn(n_1), pn(n_2), \ldots, pn(n_k)) = pn(n).$$

4. The claim for disproof sets is proved by swapping the AND with the OR case in steps 2 and 3.

Setting $h(n) = 1$ for all n results in another proof of Theorem 2 in Section 2.3, since for every set s, $h(s) = \sum_{n \in s} 1 = |s|$.

5.1 Combining Truncated Node Sets with Heuristic Leaf Evaluation Functions

The two generalizations of PSS by truncated node sets and heuristic leaf initialization can be combined by reinterpreting the bound on the set size as a bound on the set evaluation: The bound of a union of two sets is then defined by $bound(a \cup b) = bound(a) + bound(b) - \sum_{x \in I} h(x)$, $I = rep(a) \cap rep(b)$.

Example 9 *Given the following nodes, with their heuristic evaluation written as a subscript:* $A_{10}, B_{15}, C_7, D_{16}, E_{18}, F_{39}$*. Consider a 4-truncated node set representation, with the subscript of the whole set showing the evaluation bound for the whole set. Examples of exactly representable sets are* $s_1 = A_{10} \cup B_{15} \cup C_7 = \{C_7, A_{10}, B_{15}\}_{32}$ *and* $s_2 = A_{10} \cup D_{16} \cup E_{18} = \{A_{10}, D_{16}, E_{18}\}_{44}$*. Taking the union leads to set truncation,* $s_1 \cup s_2 = \{C_7, A_{10}, B_{15}, D_{16}\}_{66}$*. Different sets may share the same truncated set but have different bounds. For example,* $s_1 \cup \{B_{15}, D_{16}, F_{39}\}_{70} = \{C_7, A_{10}, B_{15}, D_{16}\}_{87}$.

The following lemma and theorem are easy generalizations of Lemma 2 and Theorem 5 respectively. Proofs are omitted for lack of space.

Lemma 3 *Let* $K > L \geq 0$ *be integers, let* s_i *be (untruncated) node sets, and let* h *be a heuristic node evaluation function. Compute the* K*-truncated set union* u_K *and the* L*-truncated set union* u_L *using a set ordering based on* h*, the same secondary sorting criterion, and the same sequence of two-set union operations. Then* $bound(u_K) \leq bound(u_L)$.

Theorem 7 *Let* K *be an integer, let* h *be a heuristic node evaluation function, and let* G *be a DAG. For each node* $n \in G$*, compute proof and disproof sets using* $PSS_{K,K}$*, and compute proof and disproof numbers using PNS with node initialization by* h*. Then for each unproven node* n *in* G*,*

$$bound(pset(n)) \leq pn(n)$$

$$bound(dset(n)) \leq dn(n).$$

5.2 Favoring Nodes From a Given Set

Assume a most-promising OR node n is expanded by PSS, and that before the expansion, the proof set of the root already contains other nodes $P = \{p_1, \ldots, p_k\}$. When computing the new minimal proof sets after expanding n, it is probably better to choose nodes which are already contained in P, since they will not increase the size of sets further up in the DAG. The node evaluation can be modified to discount such nodes by a factor ϵ, $0 \leq \epsilon < 1$: $h'(x) = \epsilon h(x)$ if $x \in P, h'(x) = h(x)$ if $x \notin P$.

A problem with this approach is that the evaluation of node sets becomes context-dependent. Also, care should be taken to always prefer a proven node x with $h(x) = 0$ over an unproven but completely discounted node y with $h(y) \neq 0$ but $h'(y) = 0$.

Example 10 *In the DAG on the left side of Figure 6, node C is the only most-promising node and is expanded next. The figure on the right shows the situation at the time of recomputing the proof set of C after the expansion. (Proof sets of node D and below are not affected by expanding C.)*

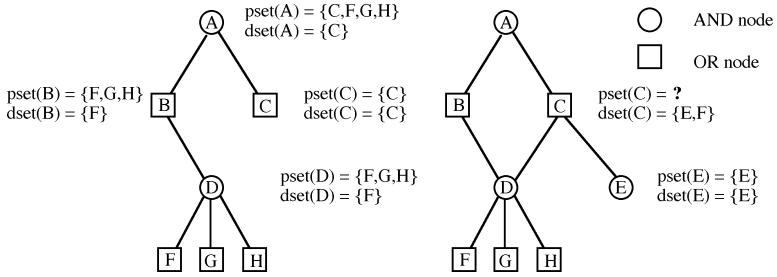

Figure 6: Discounting existing nodes

At node C, standard PSS would select the small proof set $\{E\}$ as its minimal proof set. However, the modified algorithm would discount the values of F, G and H, since they are contained in the proof set of the root A, and therefore select D over E. This way, the proof set of A shrinks because of the expansion of C, $pset(A) = \{F, G, H\}$.

6 Future Work and Summary

Future work on PSS includes extending it to cyclic game graphs and testing it in a variety of applications.

6.1 PSS for Directed Cyclic Graphs (DCG)

Directed cyclic graphs, or DCG's, cause problems for proof-based search procedures because of cyclical dependencies, often called *graph history interaction* in this context. Breuker et al. develop one solution to this problem with their *base-twin algorithm* [2].

It is presently unknown whether PSS is well-defined and how it works on general DCG's. It seems necessary to adapt the update and propagation rules, since now the same node can appear both as leaf and as interior node in the DCG. A promising sign is that PSS has no trouble solving the following example, taken from Figure 5 of [7]:

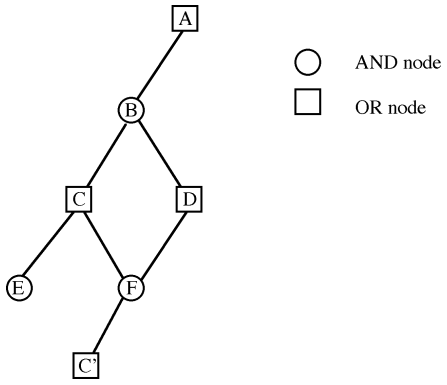

Figure 7: Directed cyclic graph with cycle $C - F - C$

Example 11 *C' is a transposition of C, leading to a cycle $C - F - C$. Let's treat the graph as a DAG and update proof sets, assuming nodes are ordered alphabetically for minimum selection. Initially, $pset(C') = \{C'\} = \{C\}$, $pset(E) = \{E\}$, $pset(F) = \{C\}$, $pset(C) = \min(pset(E), pset(C)) = \{C\}$, $pset(D) = \{C\}$, $pset(B) = pset(C) \cup pset(D) = \{C\}$, $pset(A) = \{C\}$. After proving E, $pset(E) = \emptyset$, $pset(C) = \min(pset(E), pset(C)) = \emptyset$. Now C is proved, and can be propagated through the DCG, leading successively to proofs of F, D, B and A.*

6.2 Applications of PSS

Applications should lead to a better understanding of PSS in both theory and practice. A first test on Tic-Tac-Toe resulted in PSS growing a 10% smaller DAG than PNS, with 1114 nodes against PNS' 1237 to disprove that Tic-Tac-Toe is a win for the first player. However, Tic-Tac-Toe is hardly a challenging test case, and large gains cannot be expected here.

The original motivation to develop PSS came from a report [4] that some *tsume shogi* (shogi mating) problems are hard for proof-number based algorithms because of transpositions. In first experiments with PSS in this domain, the method proved viable on moderately large DAG's with up to several hundred thousand nodes when using truncated node sets with $P = D = 20$. Preliminary results seem to indicate a correlation between the frequency of transpositions and the performance of PSS. However, a proper study remains as future work.

Another promising application area are subproblems in the game of Go, such as life and death puzzles or tactical capturing problems. Both shogi and Go provide a rich and very challenging set of test cases.

6.3 More Research Topics

Investigate the performance of the queue backup method What is the average and worst-case performance of the queue backup method on different types of DAG's? Is it sufficient for the DAG's encountered in practice? Are there DAG's for which the performance is unacceptably bad?

Find necessary conditions for optimality of queue backup Lemma 1 in Section 3.2.1 proves that the existence of a rank function for the ancestor relation is sufficient to ensure the optimality of the queue backup algorithm. Are there other, more general sufficient conditions? What are necessary conditions for optimality? Such a criterion might be based on characterizing forbidden subgraphs, such as the one in Figure 5.

Efficient data structures for large node sets A straightforward implementation of node sets by sorted lists is easy to program, but slow for large sets. Are there applications where it is essential to deal with large sets, and if so, are there more efficient data structures?

Choice of truncation values Which values of P and D provide a good tradeoff between memory and accuracy for $PSS_{P,D}$? How are the values related to the problem type? Is it beneficial to dynamically adapt P and D during the search, or use different values in different regions of the DAG?

Multi-level schemes Are schemes such as pn^2-search [3] effective for PSS?

6.4 Summary

Proof-set search, or *PSS*, is a new search method which addresses the problems caused by overestimating the size of the minimal proof set in DAG's. Like PNS, PSS uses the simple children-to-parents propagation scheme for DAG's, but unlike

PNS, it backs up proof sets instead of proof numbers. The sets computed by PSS provide better approximations than is possible with only proof numbers on the same DAG. However, more memory and more computation is needed to store and manipulate node sets instead of numbers. The trade-off between the advantages of a more focused search and the disadvantages of using more memory per node and a more expensive backup procedure need further investigation. Since overestimating proof numbers in a DAG can lead search into a completely wrong direction for a long time, any improvement in the node expansion strategy can potentially achieve large savings in search efficiency, especially on hard problems.

References

[1] L.V. Allis. *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis, University of Limburg, Maastricht, 1994.

[2] D. M. Breuker, H. J. Van den Herik, J. W. H. M. Uiterwijk, and L. V. Allis. A solution to the GHI problem for best-first search. *Lecture Notes in Computer Science*, 1558:25–49, 1999.

[3] D.M. Breuker, J.W.H.M. Uiterwijk, and H.J. van den Herik. The pn^2-search algorithm. Technical Report CS 99-04, Department of Computer Science, Universiteit Maastricht, Maastricht, 1999. To appear in Advances in Computer Chess 9 (eds. H.J. van den Herik and B. Monien).

[4] A. Kishimoto. Seminar presentation. ETL, 1999.

[5] J.A. McHugh. *Algorithmic Graph Theory*. Prentice-Hall, 1990.

[6] M. Schijf. Proof-number search and transpositions. Master's thesis, University of Leiden, 1993.

[7] M. Schijf, L.V. Allis, and J.W.H.M. Uiterwijk. Proof-number search and transpositions. *ICCA Journal*, 17(2):63–74, 1994.

[8] R. Stanley. *Enumerative Combinatorics Vol. 1*. Number 49 in Cambridge Studies in Advanced Mathematics. Cambridge University Press, 1997.