TR-99-20
Received Aug. 27, 1999

Proof-Set Search

Martin Müller

Electrotechnical Laboratory
Complex Games Lab
Umezono 1-1-4, Tsukuba
Ibaraki, JAPAN 305

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 handle transpositions very well. 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 the proof numbers for DAG's, they are too slow to be practical.

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

We study two modifications of PSS: *PSS with truncated node sets*, $PSS_{P,D}$, provides a well-controlled tradeoff between reduced memory requirements and solution quality. Both proof-number search and PSS are shown to be special cases of $PSS_{P,D}$. Another modification of PSS and $PSS_{P,D}$ makes use of heuristic estimates of leaf node costs, as have been proposed for proof-number search by Allis.

1 Proof-Number Search and Transpositions

Game tree search algorithms based on the minimax principle have been continuously enhanced over the last decades. One of the most significant recent developments has been Victor Allis' proof-number search [1]. This method does not compute a minimax value based on heuristic position evaluations as other methods do; rather its aim is to find a proof or disproof of a boolean predicate defined for game positions p, such as *IsWin(p)*.

Proof-number search (PNS) 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. There is a simple backup scheme for computing proof numbers, which is correct for trees. However, when the same backup method is used in directed acyclic graphs (DAG's), it fails to compute the correct proof and disproof numbers, since transpositions can cause the same node to be counted more that once. Therefore proof-number search can fail to identify a most-proving node, even though it is known that such a node always exists in a DAG [3, 1].

To overcome this problem, Schijf [3, 4] has developed exact methods for computing proof numbers in DAG's. 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.

The outline of the paper is as follows: the introduction continues with a short summary of proof-number search and an example that illustrates the problems of proof-number backup in DAG's. Section 2 describes the new proof-set search method. Section 3 contains several modifications and variants of the algorithm. Section 4 closes the paper with a short discussion of preliminary experiments on Tic Tac Toe and future work.

1.1 Brief Summary of Proof-Number Search

Proof-number search (PNS) computes proof and disproof numbers for each node in a game tree or DAG. These numbers are used to find a *most-proving node* to expand next in a tree search. There is a simple backup scheme for computing proof and disproof numbers, which is correct for trees.

Each search node n stores a proof number $pn(n)$ and a disproof number $dn(n)$. For a frontier (or leaf) node which is unproven, set $pn(n) = dn(n) = 1$. A proved frontier node is assigned $pn(n) = 0, dn(n) = \infty$, while a disproved 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 work the other way around, taking sums in OR nodes and minima in 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 is 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. Detailed explanations and algorithms can be found in [1].

After expanding a node, proof and disproof numbers must be recomputed. In DAG's, a change in a node must be propagated to all its parents.

1.2 Example 1: Tree-backup in a DAG Overestimates Proof Numbers

```
      A             AND-node
/  /    \  \
B  C    D   E      OR-nodes
\  \    /  / \
      F        G    AND-node
```

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 E's two children be F and G.

The proof number of F, $pn(F)$, is 1, and the proof numbers of $B \ldots E$ are also 1 since they are all OR nodes, which minimize the proof number of their children, and they all have a child with proof number 1. However, the proof number computed for A by the tree-backup scheme is inflated. Using the summation formula yields $pn(A) = pn(B) + pn(C) + pn(D) + pn(E) = 4$. The correct value of pn(A) is 1, since proving F proves A. In effect F is counted four times. If this 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 F, even though F is a very good candidate that could lead to a quick proof of A.

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, but backs up proof sets instead of proof numbers (which are an upper bound on the size of the minimal proof sets in a DAG). While the sets cannot be guaranteed to be minimal, they provide tighter bounds than is possible with proof numbers only. The prize for better approximation is that more memory is needed to store sets of nodes instead of numbers.

2.1 Backup Algorithm for Proof-Set Search

The algorithms for proof-set search are similar to the ones for proof number search [1]. Each search node n stores a proof set $pset(n)$ and a disproof set $dset(n)$. For unproven frontier nodes set $pset(n) = dset(n) = \{n\}$. A proved frontier node is assigned $pset(n) = \emptyset, dset(n) = \infty$, while a disproved node obtains $pset(n) = \infty, dset(n) = \emptyset$, where the overloaded symbol ∞ stands for an impossible (dis)proof, represented by a (dis)proof set of infinite size. For non-frontier (interior) nodes, Let the children of a node n be $n_1, \ldots, n_k$. 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.

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

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

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

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

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

Finding a minimal set among the sets of all children requires defining a total ordering on sets of nodes. A natural ordering is to always prefer a smaller-sized set to a larger one. Various tie-breaking methods for sets of the same size are discussed in section 3.2. Another possible ordering, using node evaluation as a heuristic measure of proof effort [1], is given in section 3.5.

In Example 1 above, if the nodes are ordered by $A < B < C < D < E < F < G$, and tie-breaks among sets are resolved by lexicographical ordering as in section 3.2, the computation proceeds as follows:

1. $pset(F) = dset(F) = \{F\}$, $pset(G) = dset(G) = \{G\}$.
2. $pset(B) = \min(pset(F)) = \{F\}$. $dset(B) = \bigcup(dset(F)) = \{F\}$.
3. In the same way, $pset(C) = pset(D) = \{F\}$ and $dset(C) = dset(D) = \{F\}$.
4. $pset(E) = \min(\{F\}, \{G\}) = \{F\}$ and $dset(E) = \{F\} \cup \{G\} = \{F, G\}$.
5. $pset(A) = \bigcup(pset(B), pset(C), pset(D), pset(E)) = \{F\}$.
6. $dset(A) = \min(dset(B), dset(C), dset(D), dset(E)) = \min(\{F\}, \{F, G\}) = \{F\}$.

2.2 Identifying a Most-Promising Node

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

Schijf [3] proves the existence of a most-proving node in DAG's and gives theoretically correct but impractical algorithms for computing it. Since PSS is a heuristic it can not always find a most-proving node. However, it computes a *most-promising node (mpn)* which lies in the intersection of the proof and disproof sets of the root node. The following theorem states that such a node always exists.

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

The induction proof is almost identical to the analogous theorem for proof numbers in [3]:

1. The hypothesis holds for all leaf nodes n since $pset(n) = dset(n) = \{n\}$.
2. Let n be an AND node with children $\{n_1, \ldots, n_k\}$. Assume the induction hypothesis holds for all 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 PSS over PNS

Proof-set search dominates proof number search in the following sense:

Theorem 2 *Given a DAG G, for each node $n \in G$ the size of the proof (disproof) sets computed by PSS is smaller-or-equal than the proof (disproof) number of n computed by proof number search on G.*

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

The proof is an easy consequence of the more general Theorem 6 in section 3.5, and is postponed until then.

Note that the result holds only for PSS and PNS operating on the same DAG. It does not hold for the different DAG's which are generated when PNS and PSS respectively are used to select nodes to expand next. Even though PNS's node selection is less informed than that of PSS, using the worse heuristic might occasionally lead to a smaller DAG being created.

2.4 Proof-Set Search Does not Always Find a Smallest Proof Set

As mentioned in section 2.2, PSS is a heuristic, because of its locally greedy minimum selection. It cannot always choose a set that will perform best when taking unions with other sets further up in the DAG.

2.4.1 Example 2: a Case where PSS Fails to Find a Smallest Proof Set

In the following example PSS fails to find a smallest proof set.

```
      A     AND-node
    /   \
   B      C     OR-nodes
    \    / \
      D     E     AND-nodes
     /|\    |\
    F G H   I J
```

It's easy to see that $\{F, G, H\}$ is the minimal proof set in this example. This set is necessary to prove D, which is needed to prove B, which is a required step to prove A. On the other hand, proving D also proves C, which completes the proof of A.

At node C, PSS makes 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 a proof set. Because of this choice, the proof set computed for A becomes $pset(A) = pset(B) \cup pset(C) = \{F, G, H, I, J\}$, which is almost twice as large as the optimum.

Section 3.7 discusses a variation of the algorithm that tries to improve the likelihood of selecting a set that works together well with sets from other siblings.

3 Algorithm Details and Modifications

In this section we work out some details of PSS, especially in areas where it differs from PNS, such as the selection of a most-promising node, different resource requirements and the representation of sets of nodes.

We also describe a `Truncated NodeSet` type with fixed memory requirements per set, describe $PSS_{P,D}$, a PSS version that uses such truncated node sets, and prove theorems that characterize both PNS and PSS as extreme cases of $PSS_{P,D}$.

In section 3.5 we describe another variant of PSS which uses heuristic evaluation to initialize new frontier nodes, as proposed by Allis [1] for the case of PNS.

3.1 Ancestor Updating Algorithm

The ancestor updating algorithm must take care that all children of a node are updated before the node itself. 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 [3]. Here is such an algorithm for node sets:

```
UpdateAncestorSets(mpn)
{  updateQ = ListOf(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();
      if ((oldPS == node->PS()) && (oldDS == node->DS()))
      {} // unchanged, do nothing
      else // update parents
```

```
        updateQ.Union(node->Parents());
        // append parents to the queue if not yet included
  }
} // 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.

3.1.1 Example 3: a Case where the Queue Update Algorithm Causes Multiple Updates of the Same Node

```
      A
    /   \
   /     B
  /     /
  \    C
   \ /
    D
```

In the example, there is a direct move from A to D, but there is also a longer path $A - B - C - D$. After updating D, first A and then C are appended to the queue, so A is updated before C. However, this not the final result for A, since A precedes B and C. After updating C and B, A is re-added to the queue and updated once more.

As the example shows, in the general case, the queue backup algorithm can cause multiple updates, which slows down the computation. There are two choices: either accept the inefficiency caused by multiple updates of the same node, or compute a topological ordering of the DAG by an algorithm such as [2, p.137], which marks nodes from the root downwards. We have not yet implemented such a method, and it is unclear if an efficient incremental version exists, which can update the ordering after each node expansion.

3.1.2 A Sufficient Condition for the Optimality of the Queue Backup Algorithm

The ancestor relation defines a partial ordering of the nodes in the DAG. The structure of this partial order is related to the optimality of the queue backup algorithm in the following way:

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 expands each node at most once.*

A *rank function* r [5, 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. In our case, it means that $r(c) = r(n) + 1$ for all children c of a node n.

Proof of Lemma 1: We prove that the queue backup algorithm processes nodes in order of their rank: if x is inserted into the queue before y, then $r(x) \geq r(y)$. We prove an even stronger statement: At each stage of the algorithm, the ranks of nodes in the queue are ordered monotonically decreasing and have at most two distinct values v and $v-1$. In other words, if the queue contains k elements, $q = [n_1, \ldots, n_k]$, 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 is true: Initially, the queue contains only a single element. Each Pop() operation maintains the condition, removing the element n_1 of rank $r(n_1)$. The parents of this element, some of which may be appended to the end of the queue, have rank $r(n_1) - 1$.

Examples of games where a rank function exists are those where each move adds exactly one stone to the game state, such as Connect-4, Qubic, Gomoku or Tic Tac Toe. The number of stones on the board is a rank function for a position in such a game.

3.1.3 Comparing the Ancestor Updating Algorithms of PNS and PSS

Both PNS and PSS can stop updating 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 disprove 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, it can also happen that a node's proof number changes but its proof set remains the same. This happens when a node which is already in the set is re-added along a new path. This will increase the proof number but not affect the set.

3.2 A Total Ordering for Sets of Nodes

The following rules define a simple total order on sets of nodes, which is close to the spirit of the original PNS: A smaller set is always preferred to a larger one. 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, or the order of node expansion can be chosen. As notation, let $n_1 < n_2$ if n_1 precedes n_2 in the chosen order. 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 $n_i < m_i$.

3.3 Truncated Node Sets

Set union and assignment operations on large node sets are expensive in terms of both memory and computation time. Therefore we propose a new data structure: a *K-truncated NodeSet* stores at most K nodes explicitly. For larger sets, 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 the PSS algorithm, the only time a node set can overflow a size bound K is when computing the union of two sets. Therefore we concentrate on detecting and handling this case. (We assume that the union of more than two sets is computed by successively taking unions of a two sets S and s_i, where S is the union of all sets so far, $S = s_1 \cup s_2 \cup \ldots \cup s_{i-1}$. Computing the union in one sweep over all s_i might be more efficient, but we did not try this.)

The truncated method attempts to compute the union in any case, and stores the first K elements of each overflowing set, plus the bound. Example: let $K = 8$, let nodes be represented by numbers, and let $s_1 = \{1, 3, 4, 6, 7, 8, 9, 11\}_{16}$ and $s_2 = \{2, 3, 4, 5, 6, 9, 10, 11\}_{13}$. For each truncated set, the first K elements are given explicitly, and the subscript represents the bound on the set size. So s_1 represents a set of at most 16 elements, including the 8 listed. The truncated set union is $s_1 \cup s_2 = \{1, 2, 3, 4, 5, 6, 7, 8\}_{24}$. It is computed as follows: the union of the known elements of $s_1 \cup s_2$ is $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$, which is truncated to the first $K = 8$ elements. The bound on the set union size is the size of the untruncated union plus the remaining possible elements of s_1 and s_2, $11 + (16 - 8) + (13 - 8) = 24$.

The next Lemma formalizes the intuitively clear fact that larger truncation thresholds result in tighter bounds.

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 $s_1 \cup s_2 \cup \ldots \cup s_n$ by the truncated set method and by 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 explicit part of the $L-$truncated union is a subset of the explicit part of the $K-$truncated union.*

The (easy) proof is left to the reader.

3.4 Proof-Set Search with Truncated Node Sets

Given two integers P and D, the algorithm *proof-set search with 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.

3.4.1 Selecting a Most Promising Node

With truncated node sets, Theorem 1, which guarantees that $pset(n) \cap dset(n) \neq \emptyset$ for each node, does not guarantee that a most-promising node can be found immediately. The problem is that none of the common nodes might be represented explicitly. The algorithm for selecting a most promising node is an intermediate form of the respective PNS and PSS algorithms.

```
SelectMPN() // find most promising node with truncated node sets
{
   node = root; mpn = 0;
   while (IsInteriorNode(node))
   {
      // JointNode() returns 0 if no joint node found.
      mpn = JointNode(pset(node), dset(node));
```

```
        if (mpn != 0)
            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;
    }
    // reached leaf node, this must be it.
    return node;
} // SelectMPN
```

3.4.2 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 (easy) proofs are left to the reader. Two other special cases are interesting: $PSS_{\infty,0}$ 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 $D < \infty$, the required memory remains bounded by a constant factor of what PNS would use on the same DAG.

3.4.3 Monotonicity Theorem of PSS with Truncated Node Sets

Theorem 5 *Given a DAG G and nonnegative integers P, P', D, D'. If $P \geq P'$ and $D \geq D'$, 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 those computed by $PSS_{P',D'}$.*

The theorem follows directly from Lemma 2. This theorem generalizes Theorem 2, which compared the two extreme cases $PSS_{0,0}$ and $PSS_{\infty,\infty}$. As in Theorem 2, the dominance holds only when comparing the computation of the algorithms on an identical DAG G. It does not hold, and is not even well-defined, for the different DAG's generated by actually running $PSS_{P,D}$ and $PSS_{P',D'}$.

Of course, in any given problem instance $PSS_{P',D'}$ might get lucky and grow a smaller DAG than $PSS_{P,D}$, but generally the DAG size should decrease with increasing values of P and D.

3.5 Using PSS with a Heuristic Leaf Evaluation Function

Proof numbers can be viewed as a lower bound on the work required to prove a node. The standard algorithm gives each unproven leaf node the same weight 1. However, game-specific knowledge can be used to provide different initial weights. Allis [1] proposes to use a heuristic evaluation to initialize the proof numbers of new frontier nodes. PSS can also be modified to use such a function. Define a heuristic evaluation $h(n)$ for each frontier node n as a heuristic lower bound estimate of the work required to prove n. As in proof-number search, the constant function $h(n) = 1$ results in the standard algorithm.

For a set $s = \{n_1, \ldots, n_k\}$, define $h(s) = \sum h(n_i)$. Select a minimum among sets as follows: $min(s_1, s_2) = s_1$ if $h(s_1) < h(s_2)$. If $h(s_1) = h(s_2)$, break the tie by a lexicographical ordering as in section 3.2, sorting nodes by their $h()$ value as the primary criterion and a suitable total order as a secondary criterion. An adaptation to truncated node sets will be described in Section 3.6.

Theorem 6 *Given a DAG G, and a positive heuristic evaluation function $h()$ which is defined for each position represented by a node in G. Extend $h()$ to sets of nodes by $h(s) = \sum_{n \in s} h(n)$. Then for each node $n \in G$, the evaluation of the proof (disproof) sets computed by PSS is smaller-or-equal than the proof (disproof) number of n computed by PNS with the children-to-parents backup algorithm and leaf node initialization given by $h()$.*

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

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 AND node with children $\{n_1, \ldots, n_k\}$. Assume the induction hypothesis holds for all children n_i: $h(pset(n_i)) \leq pn(n_i)$.

$$h(pset(n)) = h(pset(n_1) \cup pset(n_2), \ldots, \cup pset(n_k))$$
$$\leq h(pset(n_1)) + h(pset(n_2)) + \ldots + h(pset(n_k))$$
$$\leq pn(n_1) + pn(n_2) + \ldots + pn(n_k) = pn(n).$$

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 a proof of Theorem 2 in section 2.3, since for every set s, $h(s) = \sum_{n \in s} 1 = |s|$.

3.6 Combining Truncated Node Sets with Heuristic Leaf Evaluation Functions

The two generalizations of PSS, truncated node sets and heuristic leaf initialization, can be combined as follows: The heuristic evaluation of a truncated set consists of an exact sum of the explicit nodes' evaluation plus an upper bound on the evaluation of the truncated part. Evaluation of a union of two sets is computed in the obvious way by adding the exact valuation of the explicit nodes in the union with the two bounds of the truncated parts.

Example: 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. Then $s_1 = A_{10} \cup B_{15} \cup C_7 = \{C_7, A_{10}, B_{15}\}_{32}$ and $s_2 = A_{10} \cup D_{16} \cup E_{18} = \{A_{10}, D_{16}, E_{18}\}_{44}$ are examples of exactly representable sets. However, $s_1 \cup s_2$ overflows the truncation size 4, $s_1 \cup s_2 = \{C_7, A_{10}, B_{15}, D_{16}\}_{66}$. 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.

Lemma 3 *Let $K > L$ be integers, s be a node set computed by a fixed series of set unions, $h()$ a heuristic node evaluation function and $h_K(s), h_L(s)$ be the bounds on the evaluation of S computed using truncated node sets of size K and L respectively. Then $h_K(s) \leq h_L(s)$.*

Theorem 7 *Let $K > L$ be integers and $h_K(), h_L()$ be defined as in Lemma 3. Let G be a DAG and compute proof sets for each node $n \in G$ using $PSS_{K,K}$ and $PSS_{L,L}$. Then for each node n in a DAG G*

$$h_K(pset(n)) \leq h_L(pset(n))$$

$$h_K(dset(n)) \leq h_L(dset(n))$$

3.7 Favoring Nodes From a Given Set

Assume an OR node n is expanded, and its parents' proof sets already contain other nodes $P = \{p_1, \ldots, p_k\}$. When computing minima at n, it is probably better to choose nodes which are already contained in P, since they will not increase the sets further up in the DAG. The node evaluation can be modified to discount such nodes: $h'(x) = 0$ 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, and it may be inefficient or difficult to compute the necessary sets in the right order. Also, care should be taken to always prefer a proved node x with $h(x) = 0$ over an unproven but completely discounted node y with $h(y) \neq 0$ but $h'(y) = 0$.

3.7.1 Example 3

Consider evaluating the DAG of example 2 by depth-first traversing it from left to right.

```
      A     AND-node                    A     AND-node
    /   \                             /   \
   B      C     OR-nodes             B      C     OR-nodes
    \   /                             \   / \
      D          AND-nodes              D     E     AND-nodes
     /|\                               /|\    |\
    F G H                             F G H   I J
```

After backing up F, G, and H, the proof sets are as follows: $pset(D) = \{F, G, H\}$, $pset(B) = pset(C) = \{F, G, H\}$, $pset(A) = \{F, G, H\}$.

Now consider the stage when the proof set from E is propagated up to C. Standard PSS would select the small proof set $\{I, J\}$ of E as its minimal proof set, and therefore choose E as the most promising child of C. The modified algorithm would discount the values of F, G and H, since they are needed in a parent of C anyway, and therefore select D. This way, the proof set of A remains $pset(A) = \{F, G, H\}$.

4 Discussion and Future Work

Proof-set search, or *PSS*, is a new search method which uses the simple children-to-parents propagation scheme for DAG's, but backs up proof sets instead of proof numbers (which are an upper bound on the size of the minimal proof sets in a DAG). While the sets cannot be guaranteed to be minimal, they provide tighter bounds than is possible with proof numbers only. The prize for better approximation is that more memory is needed to store sets of nodes instead of numbers. This trade-off between the advantages of a more focused search and the disadvantage of using more memory per node is very much dependent on the specific search problem, and needs to be investigated carefully for each new type of problem.

4.1 Applications

A first test on Tic Tac Toe resulted in a 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. It now seems urgent to try applications where PNS is known to have trouble. The original motivation to develop PSS came from reports that some *tsume shogi* (shogi mating problems) are hard for proof-number based algorithms because of transpositions. Another promising area are *tsume go* (life and death problems in Go). These applications should provide a rich set of test cases, including those that are much more complex than the Tic Tac Toe example.

4.2 PSS for Directed Cyclic Graphs (DCG)

How to make PSS work with directed *cyclic* graphs (DCG)? The following example is from Figure 5 of [4]

```
         A       OR-node
        /
       B       AND-node
     /   \
    C     D     OR-nodes
  /  \   /
 E     F       AND-nodes
      /
     C'    OR-node
```

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.

PSS has no trouble solving this example. However, it is presently unknown how PSS works on general DCG's. It seems necessary to adapt the update and propagation rules, since now the same node can be both leaf and interior node in the DCG.

4.3 More Questions

Theoretical Properties of PSS Is there a worst-case bound on the ratio of the size of PSS sets versus minimal proof sets? Does the ratio depend on depth of the DAG? Idea: iterated version of example in section 2.4.1. It is easy to see that there is no such bound for PNS: use Example 1 with n nodes at the second level.

Find Necessary Conditions for Optimality of Queue Backup Lemma 1 in Section 3.1.2 proves that a ranked ancestor relation is sufficient to ensure optimality of the queue backup algorithm. What are necessary conditions?

Do More Efficient NodeSet Data Structures Exist? Sorted lists are simple to implement, but slow in performing most operations on large sets. Are there applications where it is essential to deal with large sets, and if so, are there more efficient data structures for implementing NodeSet?

Adaptive Threshold Truncation Change D, P during the search, use different values in different nodes.

Choices for NodeSet Truncation Values P, D Which values provide a good memory-accuracy tradeoff for $PSS_{P,D}$?

References

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

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

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

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

[5] R. Stanley. *Enumerative Combinatorics*, volume 1. Cambridge University Press, 1997.