Dealing with Infinite Loops, Underestimation, and Overestimation of Depth-First Proof-Number Search

Akihiro Kishimoto
Department of Mathematical and Computing Sciences
Graduate School of Information Science and Engineering
Tokyo Institute of Technology
PRESTO, Japan Science and Technology Agency, Japan
Email:kishimoto@is.titech.ac.jp

Abstract

Depth-first proof-number search (df-pn) is powerful AND/OR tree search to solve positions in games. However, df-pn has a notorious problem of infinite loops when applied to domains with repetitions. Df-pn(r) cures it by ignoring proof and disproof numbers that may lead to infinite loops.

This paper points out that df-pn(r) has a serious issue of underestimating proof and disproof numbers, while it also suffers from the overestimation problem occurring in directed acyclic graph. It then presents two practical solutions to these problems. While bypassing infinite loops, the threshold controlling algorithm (TCA) solves the underestimation problem by increasing the thresholds of df-pn. The source node detection algorithm (SNDA) detects the cause of overestimation and modifies the computation of proof and disproof numbers.

Both TCA and SNDA are implemented on top of df-pn to solve tsume-shogi (checkmating problem in Japanese chess). Results show that df-pn with TCA and SNDA is far superior to df-pn(r). Our tsume-shogi solver is able to solve several difficult positions previously unsolved by any other solvers.

Introduction

Developing efficient AND/OR tree search has been a fundamental topic in AI, because solving complex AND/OR trees is required for problem-solving procedures. Such an example is to find a winning way from a given position in games.

Research in solving tsume-shogi (checkmating problem in Japanese chess) has produced a variety of powerful domain-independent AND/OR tree search techniques (Seo, Iida, and Uiterwijk 2001; Nagai 2002). In particular, successful ideas behind depth-first proof-number search (df-pn) (Nagai 2002) have been adapted to other games, including checkers (Schaeffer et al. 2007), and the life and death problem in the game of Go (Kishimoto and Müller 2005).

Df-pn is an enhanced reformulation of best-first proof-number search (PNS) (Allis, van der Meulen, and van den Herik 1994) in a depth-first manner. A leaf node to expand next is selected by Allis' proof and disproof numbers, an estimated difficulty to find a win or loss. The df-pn search is controlled by the thresholds of proof and disproof numbers.

If the search space is a tree, df-pn is equivalent to PNS with preserving good properties of reducing memory requirement and interior node re-expansions. However, the search space of many games is directed acyclic graph (DAG), or even directed cyclic graph (DCG). While df-pn is still a pragmatic choice, several issues must be addressed.

The infinite loop problem is an essential problem causing df-pn never to solve a position even in infinite time. Although df-pn(r) (Kishimoto 2005) cures this issue, we address that df-pn(r) suffers from underestimating proof and disproof numbers. The overestimation problem is caused by counting proof and disproof numbers of the same node many times, and has been addressed previously in (Schijf 1993; Müller 2003; Seo, Iida, and Uiterwijk 2001). Both overestimation and underestimation delay searching promising nodes and increase the search effort by a large margin.

This paper presents techniques to handle the three aforementioned problems. Its contributions are synthesis of df-pn with two novel methods and a demonstration of the promise of our approach in solving difficult tsume-shogi instances.

The rest of this paper is organized as: Starting with a description of tsume-shogi, we review related work. Our new algorithms are then described, followed by experimental results, and conclusions with outlines of further work.

Tsume-Shogi as a Testbed of AI Research

Shogi (Japanese chess) is a game with 15 million players and the history of over 400 years. Not only there are a few hundred human professional players but also an annual world computer shogi championship has been held for 20 years to determine the best of over 40 participating programs[1].

As in chess the goal of shogi is to capture the king. However, unlike in chess, captured pieces can be later reused by dropping one of them on the board (Hosking 1996). Because of this rule, shogi has a larger average branching factor than chess (80 - 100 versus 35). Developing strong shogi programs has been an important subject of game-playing (Schaeffer 2001; Iida, Sakuta, and Rollason 2002).

Tsume-shogi is the checkmating problem where the attacker checks the king and the defender escapes. The attacker is the first player. To solve tsume-shogi, the attacker must find at least one winning move, while all the moves responded by the defender must be proven to be check mate.

[1]`http://www.computer-shogi.org/index_e.html`

Tsume-shogi is an ideal domain to investigate ideas on AND/OR tree search, because there are many difficult instances composed by tsume-shogi creators for hundreds of years. Searching over a thousand of depth (*ply*) is often required to find solutions. Besides, specialized tsume-shogi search engines are incorporated into most shogi programs, because checkmating attacks plays a crucial role in shogi.

Related Work and Problem Descriptions

Depth-First Proof-Number Search

Let a proof be a win for the attacker (OR node) and a disproof be a win for the defender (AND node). Df-pn (Nagai 2002) leverages proof and disproof numbers $\mathbf{pn}(n)$ and $\mathbf{dn}(n)$ (Allis, van der Meulen, and van den Herik 1994) to estimate a difficulty of finding a proof or disproof of node n.

$\mathbf{pn}(n)$ is the minimum number of leaf nodes that must be *proven* to find a proof for n in a currently generated tree. $\mathbf{dn}(n)$ is the minimum number of leaf nodes to be *disproven* to find a disproof for n. $\mathbf{pn}(n) = 0$ and $\mathbf{dn}(n) = \infty$ are set for proven terminal node n, while $\mathbf{pn}(n) = \infty$ and $\mathbf{dn}(n) = 0$ for disproven terminal node. $\mathbf{pn}(n) = \mathbf{dn}(n) = 1$ is set for unproven leaf. Let $n_1, n_2, \cdots, n_k$ be children of interior node n. Since only one proven child suffices to prove an OR node, while all children must be proven to prove an AND node (and vice versa for disproof). $\mathbf{pn}(n)$ and $\mathbf{dn}(n)$ are:

- For OR node n, $\mathbf{pn}(n) = \min_{i=1,\cdots,k} \mathbf{pn}(n_i)$ and $\mathbf{dn}(n) = \sum_{i=1}^{k} \mathbf{dn}(n_i)$.
- For AND node n, $\mathbf{pn}(n) = \sum_{i=1}^{k} \mathbf{pn}(n_i)$ and $\mathbf{dn}(n) = \min_{i=1,\cdots,k} \mathbf{dn}(n_i)$.

As in PNS, df-pn expands a leaf node (called a *most-promising node*) reached from the root by selecting a child with the smallest proof number at each OR node, and a child with the smallest disproof number at each AND node. Unlike PNS, df-pn does so by using two thresholds: $\mathbf{th_{pn}}(n)$ for proof numbers and $\mathbf{th_{dn}}(n)$ for disproof numbers.

At the root r, $\mathbf{th_{pn}}(r)$ and $\mathbf{th_{dn}}(r)$ are initialized to ∞. Whenever df-pn reaches n from its parent p, or backtracks to n after expanding n's descendants, it computes $\mathbf{pn}(n)$ and $\mathbf{dn}(n)$ by using proof and disproof numbers of n's children. $\mathbf{pn}(n)$ and $\mathbf{dn}(n)$ are cached in the transposition table for reuse. If $\mathbf{pn}(n) \geq \mathbf{th_{pn}}(n)$ or $\mathbf{dn}(n) \geq \mathbf{th_{dn}}(n)$ holds, df-pn backtracks to p. Otherwise, it selects a child n_1 with the smallest (dis)proof number at OR (AND) node n. Let n_2 be a child with the second smallest (dis)proof number at OR (AND) node n. Df-pn searches n_1 with the following thresholds indicating the condition in which either n_1 or n is no longer on the path to reach a most-promising node:

- For OR node n, $\mathbf{th_{pn}}(n_1) = \min(\mathbf{th_{pn}}(n), \mathbf{pn}(n_2) + 1)$ and $\mathbf{th_{dn}}(n_1) = \mathbf{th_{dn}}(n) - \mathbf{dn}(n) + \mathbf{dn}(n_1)$.
- For AND node n, $\mathbf{th_{pn}}(n_1) = \mathbf{th_{pn}}(n) - \mathbf{pn}(n) + \mathbf{pn}(n_1)$ and $\mathbf{th_{dn}}(n_1) = \min(\mathbf{th_{dn}}(n), \mathbf{dn}(n_2) + 1)$.

Df-pn continues these procedures until finding a solution.

The Infinite Loop Problem and Df-pn(r)

When standard df-pn is applied to DCG, it has a fundamental problem of causing infinite loops. Although (Nagai 2002) does not mention the existence of this problem[2], (Kishimoto 2005) confirms it occurs in many domains including Go, checkers, and shogi. Besides, (Kishimoto and Müller 2008) prove that df-pn loops forever when searching a DCG in Figure 1. Df-pn must expand P to prove A. However, this never occurs, because df-pn keeps selecting two paths (a) $A \to C \to F \to H \to L \to N$ and (b) $A \to C \to G \to J \to O$, and always satisfies $\mathbf{th_{dn}}(N) \leq \mathbf{dn}(N) = \mathbf{dn}(O)$ via (a) and $\mathbf{th_{dn}}(O) \leq \mathbf{dn}(O) = \mathbf{dn}(H) + \mathbf{dn}(P)$ via (b).

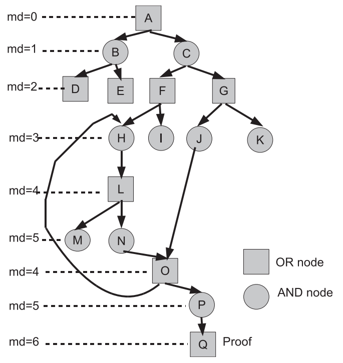

Figure 1: An example showing that df-pn loops forever, adapted from (Kishimoto and Müller 2008)

Df-pn(r) is a practical method to handle infinite loops (Kishimoto and Müller 2003; Kishimoto 2005). $\mathbf{pn}(n)$ for OR node and $\mathbf{dn}(n)$ for AND node are computed in a standard way. Because adding (dis)proof numbers from an ancestor to a node is intuitively bad, df-pn(r) detects this case by computing *minimal distance* $\mathbf{md}(n)$, the length of the shortest path from the root to node n (see $\mathbf{md}$ in Figure 1). Df-pn(r) classifies n's children c into two types: c is *normal* if $\mathbf{md}(n) < \mathbf{md}(c)$. Otherwise c is *old*. If at least one unproven normal child exists, all old children are ignored to compute $\mathbf{dn}(n)$ for OR node n. Otherwise, $\mathbf{dn}(n)$ is set to the maximum of disproof numbers of all old children. $\mathbf{pn}(n)$ for AND node n is analogously treated.

In Figure 1, df-pn(r) computes $\mathbf{dn}(O) = \mathbf{dn}(P)$ for unproven P and H, because H is old and P is normal at O.

The Underestimation and Overestimation Problems

Although (Kishimoto 2005) shows that df-pn(r) is effective in Go and checkers, we point out that it often underestimates (dis)proof numbers, as shown in the left example in Figure 2. Df-pn(r) computes $\mathbf{dn}(D) = \mathbf{dn}(E)$ for unproven E and F, because F is D's old child. However, more reasonable is $\mathbf{dn}(D) = \mathbf{dn}(E) + \mathbf{dn}(F)$, because both E and F must be disproven to disprove D.

Moreover, both df-pn and df-pn(r) overestimate proof and disproof numbers. In the right example in Figure 2, the

[2]Despite our best effort, it is not possible to compare our solution with Nagai's "solution". Nagai also writes several Japanese articles on tsume-shogi. However, the infinite loop problem is never described, although it would be impossible to develop a strong solver without handling it.

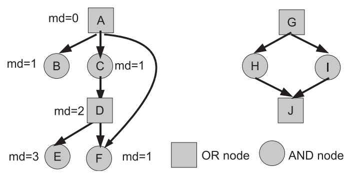

Figure 2: An example of the problems of underestimation (left) and overestimation (right)

true value of $\mathbf{dn}(G)$ should be equal to $\mathbf{dn}(J)$. However, computing $\mathbf{dn}(G)$ counts $\mathbf{dn}(J)$ twice, derived by $\mathbf{dn}(G) = \mathbf{dn}(H) + \mathbf{dn}(I) = \mathbf{dn}(J) + \mathbf{dn}(J) = 2 \times \mathbf{dn}(J)$.

Previous Work on the Overestimation Problem

The overestimation problem occurs only when computing $\mathbf{pn}$ for AND node and $\mathbf{dn}$ for OR node. In the other cases, $\mathbf{pn}(n)$ and $\mathbf{dn}(n)$ are computed in a standard way.

Since overestimation also occurs in PNS, a few solutions are presented but are specific to PNS (Allis 1994; Schijf, Allis, and Uiterwijk 1994; Müller 2003). An exact method in (Schijf 1993) is unfortunately impractical even for tic-tac-toe, due to high computational overhead.

In Figure 2, let us call J a *destination node*, which has more than one parent, and G a *source node* of destination node n_d, which is n_d's first ancestor merged into by tracing back different paths starting at n_d. (Nagai 2002) presents a method to detect a source node n_s and takes the maximum of (dis)proof numbers of n_s's children to compute $\mathbf{pn}(n_s)$ (or $\mathbf{dn}(n_s)$), instead of summing up them. In his method, each transposition table (TT) entry of node n has one pointer to one parent p. When df-pn reaches n via p, p is saved in n's TT entry. Then, if df-pn reaches n via another parent q, it indicates that n is a destination node and there may exist a source node n_s that counts $\mathbf{pn}(n)$ or $\mathbf{dn}(n)$ more than once. n_s is detected by checking if one of the ancestors obtained by recursively traversing pointers from n in the TT is merged into n_s, which is also on the path of the current df-pn search.

In Figure 2, assume that J points to H and H points to G in the TT, because df-pn previously reaches J via $G \rightarrow H \rightarrow J$. Then, if df-pn reaches J via $G \rightarrow I \rightarrow J$, Nagai's method detects that J has a different parent I, which is not stored in J's TT entry (i.e., H). G is detected as a source node by traversing $J \rightarrow H$ and then $H \rightarrow G$ in the TT, and G is on path $G \rightarrow I \rightarrow J$. Nagai's method then sets $\mathbf{pn}(G) = \max(\mathbf{dn}(H), \mathbf{dn}(I)) = \mathbf{dn}(J)$ accurately. However, this approach underestimates $\mathbf{dn}(A)$, because it considers A to be a source and computes $\mathbf{dn}(A) = \max(\mathbf{dn}(B), \mathbf{dn}(C), \mathbf{dn}(F)) = \max(\mathbf{dn}(B), \mathbf{dn}(E) + \mathbf{dn}(F), \mathbf{dn}(F)) = \max(\mathbf{dn}(B), \mathbf{dn}(E) + \mathbf{dn}(F))$. The true value of $\mathbf{dn}(A)$ should be $\mathbf{dn}(B) + \mathbf{dn}(E) + \mathbf{dn}(F)$.

Let b be the number of unproven children and $n_1, \cdots n_k$ be children of node n. Weak proof-number search (WPNS) (Ueda et al. 2008), which is a refinement to (Okabe 2005), modifies $\mathbf{pn}(n)$ and $\mathbf{dn}(n)$ in the following way:

- For OR node n, $\mathbf{dn}(n) = \max_{i=1,\cdots,k} \mathbf{dn}(n_i) + b - 1$.
- For AND node n, $\mathbf{pn}(n) = \max_{i=1,\cdots,k} \mathbf{pn}(n_i) + b - 1$.

WPNS avoids counting $\mathbf{dn}(J)$ twice to compute $\mathbf{dn}(G)$ in Figure 2, at the price of slightly overestimating $\mathbf{dn}(G)$ to $\mathbf{dn}(J)+1$ and reusing the proof and disproof numbers in the TT less frequently than df-pn. Also, WPNS underestimates proof and disproof numbers, because most of the (dis)proof numbers at interior nodes are assumed to be 1.

Our Solutions

This section describes two practical algorithms to overcome the issues of previous approaches.

The Threshold Controlling Algorithm

Df-pn(r) tries to break the termination condition at node n that has an unproven old child by decreasing $\mathbf{pn}(n)$ or $\mathbf{dn}(n)$. This way, it reaches a leaf, avoiding infinite loops. In contrast, the Threshold Controlling Algorithm (TCA), breaks the termination condition by increasing the thresholds at n. This way, TCA tries to reach a leaf to expand.

The underestimation problem of df-pn(r) is caused by ignoring old children determined by $\mathbf{md}(n)$, although $\mathbf{md}(n)$ does not always indicate the existence of infinite loops. TCA computes $\mathbf{pn}(n)$ and $\mathbf{dn}(n)$ in a standard way. TCA therefore cures underestimation in Figure 2. $\mathbf{dn}(D) = \mathbf{dn}(E) + \mathbf{dn}(F)$ in TCA while $\mathbf{dn}(D) = \mathbf{dn}(E)$ in df-pn(r).

To overcome infinite loops, TCA leverages the minimal distance as a criterion of increasing thresholds. When TCA enters n and computes $\mathbf{pn}(n)$ and $\mathbf{dn}(n)$ by retrieving the TT entries of n's children, it checks if any unproven child of n is old. If n has no unproven old child, the standard df-pn procedure is performed (this will be modified later). Otherwise, TCA resets the thresholds to $\max(\mathbf{th_{pn}}(n), \mathbf{pn}(n) + 1)$ and $\max(\mathbf{th_{dn}}(n), \mathbf{dn}(n) + 1)$. With the increased thresholds, $\mathbf{pn}(n) < \mathbf{th_{pn}}(n)$ and $\mathbf{dn}(n) < \mathbf{th_{dn}}(n)$ hold. TCA therefore expands n and selects the best child n_1 - i.e., one with the smallest proof number at OR node and one with smallest disproof number at AND node. TCA adjusts $\mathbf{th_{pn}}(n_1)$ and $\mathbf{th_{dn}}(n_1)$ in a standard way except that they are based on the increased thresholds.

Assume that TCA increases $\mathbf{th_{pn}}(n)$ and $\mathbf{th_{dn}}(n)$ at node n that has an unproven old child, selects the best child c at n, and enters c. $\mathbf{th_{pn}}(c)$ and $\mathbf{th_{dn}}(c)$ are also increased even if c has only unproven normal children. This is to avoid infinite loops caused by the case where TCA immediately backtracks to n from c by satisfying the termination condition at c. TCA continues the procedure of increasing thresholds until either expanding a leaf, or detecting a new cyclic path. Expanding a leaf reduces the unexplored search space. If a new cycle is detected, (Kishimoto and Müller 2004) correctly saves a (dis)proof in the TT, and the (dis)proof is propagated back. This also reduces the unexplored space. When TCA backtracks to n, it makes progress in finding a solution. Therefore, after TCA recomputes $\mathbf{pn}(n)$ and $\mathbf{dn}(n)$, the previously increased thresholds are reused to determine if TCA still needs to re-search n.

Figure 3 presents the pseudo-code of df-pn with TCA. In particular, see lines marked by (*) to clarify the difference between standard df-pn and TCA. Note that inc_flag is a flag to determine if TCA increases $thpn$ and $thdn$.

```
     DFPNwithTCA($n$, thpn, thdn, inc_flag) {
       if ($n$ is a terminal node) { handle $n$ and return; }
       first_time = true;
       while (1)  {
(*)      /* determine whether thpn and thdn are increased. */
(*)      if ($n$ is a leaf)  inc_flag = false;
(*)      if ($n$ has an unproven old child) inc_flag = true;
(*)      expand and compute $\mathbf{pn}(n)$ and $\mathbf{dn}(n)$;
(*)      if (first_time  &&  inc_flag)  {
(*)        /* increase thresholds */
(*)        thpn = max(thpn, $\mathbf{pn}(n)$ + 1);
(*)        thdn = max(thdn, $\mathbf{dn}(n)$ + 1);
(*)      }
         if ($\mathbf{pn}(n) \geq$ thpn || $\mathbf{dn}(n) \geq$ thdn)
           break; // termination condition is satisfied
(*)      first_time= false;
         find the best child $n_1$ and second best child $n_2$;
         if ($n$ is an OR node)  { /* set new thresholds */
           thpn_child = min(thpn, $\mathbf{pn}(n_2)$ + 1);
           thdn_child = thdn - $\mathbf{dn}(n)$ + $\mathbf{dn}(n_1)$;
         else {
           thpn_child = thpn - $\mathbf{pn}(n)$ + $\mathbf{pn}(n_1)$;
           thdn_child = min(thdn, $\mathbf{dn}(n_2)$ + 1);
         }
         DFPNwithTCA($n_1$, thpn_child, thdn_child, inc_flag);
       }
       save $\mathbf{pn}(n)$ and $\mathbf{dn}(n)$ in TT.
     }
```

Figure 3: Pseudo code of df-pn with TCA

One might argue that increasing thresholds may result in exploring the larger search space that may not efficiently help prove or disprove the root. However, the threshold control scheme of TCA is based on the observation that most nodes have only normal children. As we will see in the next section, TCA empirically performs better than df-pn(r).

The Source Node Detection Algorithm

Because TCA does not ignore old children, the overestimation problem tends to occur more frequently than in df-pn(r). The Source Node Detection Algorithm (SNDA) overcomes this problem by generalizing Nagai's method, while avoiding underestimation that appears in his method, and computing (dis)proof numbers more accurately than WPNS.

As in (Nagai 2002), SNDA contains one pointer to one parent in each TT entry. SNDA detects a source node n_s in the same way. However, unlike in Nagai's method, each TT entry contains the fixed number N of moves.

Assume that c_1 and c_2 are n_s's children created by respectively making moves m_1 and m_2 at n_s. SNDA detects that n_s is a source node leading to a destination node n_d, if n_s and c_1 are on the path currently taken by df-pn search, c_2 is reached by a recursive procedure of traversing parent pointers starting from n_d, and the pointer in c_2's TT entry points to n_s. m_1 and m_2 are then saved in the TT entry of n_s, because they eventually lead to n_d. If another move m_3 is detected at n_s, m_3 is additionally saved in n_s's TT entry, as long as less than N moves are retained.

Let $n_1, \cdots, n_l$ be n's children obtained by making moves in the TT entry of n, and $n_{l+1}, \cdots n_k$ be other children of n. SNDA computes $\mathbf{pn}(n)$ and $\mathbf{dn}(n)$ as follows:

$$\begin{aligned}\mathbf{dn}(n) &= \max_{i=1,\cdots,l} \mathbf{dn}(n_i) + \sum_{j=l+1}^{k} \mathbf{dn}(n_j) \text{ (OR node)}\\ \mathbf{pn}(n) &= \max_{i=1,\cdots,l} \mathbf{pn}(n_i) + \sum_{j=l+1}^{k} \mathbf{pn}(n_j) \text{ (AND node)}\end{aligned}$$

In Figure 2, SNDA saves moves $A \to C$ and $A \to F$ in A's TT entry when detecting that A is a source node of F. $\mathbf{dn}(A)$ is now accurately computed because:

$$\begin{aligned}\mathbf{dn}(A) &= \max(\mathbf{dn}(C), \mathbf{dn}(F)) + \mathbf{dn}(B)\\ &= \max(\mathbf{dn}(D), \mathbf{dn}(F)) + \mathbf{dn}(B)\\ &= \max(\mathbf{dn}(E) + \mathbf{dn}(F), \mathbf{dn}(F)) + \mathbf{dn}(B)\\ &= \mathbf{dn}(E) + \mathbf{dn}(F) + \mathbf{dn}(B)\end{aligned}$$

Also, $\mathbf{dn}(G) = \mathbf{dn}(J)$ is derived by SNDA.

Implementation Details

As in (Kishimoto and Müller 2003), TCA updates the minimal distance of node n if n has only unproven old children.

There are several choices when implementing SNDA. We choose the best possible one, based on the results obtained by our preliminary implementations.

In the worst case, the complexity of operations to traverse pointers is equal to the maximum search depth of all previous search. Moreover, this checking process is performed whenever SNDA detects a destination node n_d reached via a parent that is not in n_d's TT entry. This is because many paths to n_d exist as well as whether a source node n_s exists or not is determined only by comparing a node on the *current* path of depth-first search with a set of nodes obtained by traversing the pointers in the TT. Since detecting n_s incurs intensive computations, we incorporate a strategy in (Nagai 2002). This strategy checks if $\mathbf{pn}(n_d)$ or $\mathbf{dn}(n_d)$ is involved in computing $\mathbf{pn}(n_s)$ or $\mathbf{dn}(n_s)$. If n_d is not involved, the procedure of detecting n_s is immediately terminated.

Assume that move m is saved in n_s's TT entry. While m was a cause of overestimating $\mathbf{pn}(n_s)$ or $\mathbf{dn}(n_s)$ in previous iterations, it may be unrelated to overestimating $\mathbf{pn}(n_s)$ or $\mathbf{dn}(n_s)$ in the next iteration. One choice is to remove m from the TT entry in this case. However, m is preserved once m is saved in the TT entry in our implementation, because removing m returned larger (dis)proof numbers, and degraded the performance. Similarly, the implementation retains the parent pointer once it is saved in the TT entry.

Each TT entry contains at most 10 moves within a total of 64 bit integer, which is possible due to a small average branching factor (about 5) in tsume-shogi. The move indexes obtained by sorting moves are saved in the TT entry to have consistent results for the identical node via different paths. If more than 10 moves exist causing overestimation, 10 best moves are selected by shogi-specific knowledge.

Experimental Results

Setup of Experiments

Many state-of-the-art techniques are integrated with all of our implementations including (Kawano 1996; Seo 1999; Nagai 2002; Kaneko et al. 2005; Kishimoto and Müller 2004; 2005). Overestimation is sometimes easily detected, such as the case where both players keep capturing pieces at a certain square on the board. These are handled by shogi-specific techniques. The remaining overestimation problem is hard to detect by shogi-specific knowledge.

Unlike (Ueda et al. 2008), our WPNS implementation includes an evaluation function $h(n)$ based on the material balance to heuristically initialize proof and disproof numbers at leaf node. Therefore, we slightly change the computation of WPNS to add $(b-1) \times h(n)$, instead of adding $(b-1)$ to $\mathbf{pn}(n)$ or $\mathbf{dn}(n)$. This modification improves WPNS by a large margin. Additionally, $h(n)$ is incorporated into all the versions.

78 notoriously difficult tsume-shogi instances composed by tsume-shogi creators are selected from (Kato 2009), including ones that turned out to have no mating sequence or shorter unexpected solutions, because many of them are still non-trivial. The solution lengths range between 300 and 1525 ply. The instances involve complicated DAG and cycles that occur after tsume-shogi solvers search over tens of ply. Moreover, the test suite includes several instances that have not been solved by any other tsume-shogi solver [3].

Our experiments were run on a 2.66GHz Xeon L5410 with 6MB L2 cache with the time limit of 50,000 seconds (about 13.9 hours) per instance with a 2GB transposition table. We prepare several versions to compare performance.

Results

Table 1 summarizes the number of unsolved instances. The table confirms that df-pn with TCA solves more instances than df-pn(r). TCA solves 40 instances more quickly than df-pn(r) of 63 instances solved by both versions.

A typical observed example showing that df-pn(r) fails in solving instances due to underestimation is explained with the help of Figure 4. Assume that B must be proven to prove the root, C has two unproven children D and E, which are typically created by moving kings in two directions, and E is old because of a shorter path via F, which is typically created by moving a promoted bishop or rook. Because df-pn(r) underestimates $\mathbf{pn}(C) = \mathbf{pn}(D)$ by ignoring E, it selects C at A and delays expanding B. In contrast, TCA selects B because $\mathbf{pn}(C) = \mathbf{pn}(D) + \mathbf{pn}(E) > \mathbf{pn}(D)$.

Due to the overestimation problem caused by DAG, df-pn with TCA occasionally suffers from a phenomenon in which the (dis)proof number at the root exceeds a large integer value(=200,000,000), which is defined as ∞ in our implementation to represent proof and disproof numbers within 32 bit integer. Of 8 unsolved instances, TCA is unable to solve 5 instances because of the phenomenon. This demonstrates that it is necessary to handle overestimation.

[3](Nagai 2002) claims that his solver solved all instances with solution lengths of over 300 ply, which were available at the year of 2002. Not only quite a few difficult instances are later composed by tsume-shogi creators, but also Nagai's remarkable results are unfortunately known to be unreproducible, partly due to a lack of descriptions on the infinite loop problem. For example, although (Okabe 2005) develops one of the best solvers, his solver cannot still solve at least 6 instances in our test suite. Moreover, neither the source nor executable code of Nagai's solver is publicly available.

Table 1: The number of problems unsolved by each method

Algorithm	Num. unsolved
df-pn(r)	14
df-pn(r) + SNDA	20
df-pn + TCA	8
df-pn + TCA + NAGAI	4
df-pn + TCA + WPNS	1
df-pn + TCA + SNDA	1

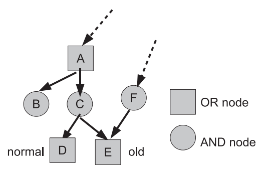

Figure 4: A typical case of underestimation on df-pn(r)

Interestingly, while including a method to detect overestimation (i.e., NAGAI, WPNS, or SNDA) improves the solving ability of df-pn with TCA, df-pn(r) with SNDA solves much fewer instances than standard df-pn(r). We observed that df-pn(r) with SNDA occasionally suffers from severer underestimation and results in poor performance. For example, df-pn(r) happens to accurately compute $\mathbf{dn}(A) = \mathbf{dn}(B) + \mathbf{dn}(E) + \mathbf{dn}(F)$ for unproven B, E, and F in Figure 2, because $\mathbf{dn}(D) = \mathbf{dn}(E)$. With SNDA, df-pn(r) underestimates $\mathbf{dn}(A) = \mathbf{dn}(B) + \max(\mathbf{dn}(C), \mathbf{dn}(F)) = \mathbf{dn}(B) + \max(\mathbf{dn}(E), \mathbf{dn}(F))$, because SNDA detects the existence of source node A from destination F.

In Table 1, it is not surprising to confirm that SNDA is better than NAGAI to be integrated with df-pn + TCA, because SNDA extends NAGAI. However, WPNS with df-pn + TCA solves the same set of instances solved by SNDA. In most cases, WPNS achieves comparable performance to SNDA. Of 77 problems solved by both, SNDA solves 38 problems more quickly, while WPNS solves 39 problems more quickly. On average, the node expansion rate with WPNS is about 37% larger than that with SNDA.

Table 2: Execution time (seconds) on the hardest problems for df-pn + TCA + WPNS to solve

Instance	TCA + WPNS	TCA + SNDA	Solution Length
Meta-Shinsekai	45,356	7,298	941
Sekitoba	18,545	7,241	525
Megalopolis	48,907	13,590	515

However, when instances become extremely hard to solve, SNDA is more important. Tables 2 and 3 show the

Table 3: Node expansions on the hardest problems for df-pn + TCA + WPNS to solve.

Instance	TCA + WPNS	TCA + SNDA
Meta-Shinsekai	7,775,332,911	1,045,623,765
Sekitoba	3,260,072,126	1,048,715,194
Megalopolis	7,479,994,586	853,059,521

hardest instances for df-pn + TCA + WPNS. SNDA solves these instances 2.5-6.2 times more quickly than WPNS, because WPNS expands 3.1-8.8 times more nodes than SNDA. In solving Meta-Shinsekai, while WPNS is expected to underestimate proof numbers, SNDA returns smaller proof numbers than WPNS on several positions located on the solution sequence in our analysis. One hypothesis is that slight overestimation of proof numbers in WPNS is accumulated when proof numbers are backed up to the root with very long sequence, and eventually becomes non-negligible[4].

One important note is that both SNDA and WPNS must be integrated with TCA. While SNDA and WPNS often avoid the infinite loop problem by making proof and disproof numbers smaller, they do not always handle infinite loops. If we switch off TCA, the number of unsolved instances is increased to 7 for both df-pn + SNDA and df-pn + WPNS. Of 7 unsolved instances, each version seems to be unable to solve 5 instances because of the infinite loop problem.

To the best of our knowledge, three instances ("Megalopolis" (modified version, solution length of 515 ply), "Journey to Jupiter" (411 ply), and "Atlantis" (951 ply)) are newly solved only by our solver (13,590 seconds, and 5,481 seconds, and 210 seconds by df-pn + TCA + SNDA). Also, our solver solves all the problems that are solved by Nagai's solver but that remain unsolved by any other solvers. Such examples include Meta-Shinsekai and Sekitoba in Table 2.

Conclusions and Future Work

This paper presents practical solutions to handle infinite loops, underestimation, and overestimation occurring in depth-first proof-number search. Promising results are obtained in solving notoriously difficult tsume-shogi problems.

One obvious research direction is to apply TCA and SNDA to other domains. Tsume-Go is an ideal domain, because the best solver is based on df-pn(r) (Kishimoto and Müller 2005). Also, because our TCA also improves WPNS, which has an advantage of the faster node expansion rate than SNDA, a hybrid approach of SNDA and WPNS is another research possibility. Finally, as an open question, (Kishimoto and Müller 2008) describes that df-pn(r) is a strong candidate for guaranteeing the completeness on solving any finite DCG. Our remedy for infinite loops is also a candidate to investigate theoretical completeness properties.

[4]Solving Meta-Shinsekai and Megalopolis builds a DAG as in the right example in Figure 2 with many layers. They are typically created by a few moves on defender's promoted pawns and king.

Acknowledgments

Finacial support was provided by the JST PRESTO program. We would like to thank Tomoyuki Kaneko and Martin Müller for their beneficial comments.

References

Allis, L. V.; van der Meulen, M.; and van den Herik, H. J. 1994. Proof-number search. *Artificial Intelligence* 66(1):91–124.

Allis, L. V. 1994. *Searching for Solutions in Games and Artificial Intelligence*. Ph.D. Dissertation, University of Limburg.

Hosking, T. 1996. *The Art of Shogi*. The Shogi Foundation.

Iida, H.; Sakuta, M.; and Rollason, J. 2002. Computer shogi. *Artificial Intelligence* 134(1-2):121–144.

Kaneko, T.; Tanaka, T.; Yamaguchi, K.; and Kawai, S. 2005. Df-pn with fixed-depth search at frontier nodes (in Japanese). In *10th Game Programming Workshop*, 1–8.

Kato, T. 2009. Tsume-shogi toy box: List of tsume-shogi problems with very long solution sequences (in Japanese). http://www.ne.jp/asahi/tetsu/toybox/kenkyu/cholist.htm.

Kawano, Y. 1996. Using similar positions to search game trees. In *Games of No Chance*, volume 29 of *MSRI Publications*, 193–202.

Kishimoto, A., and Müller, M. 2003. Df-pn in Go: Application to the one-eye problem. In *Advances in Computer Games Many Games, Many Challenges*, 125–141.

Kishimoto, A., and Müller, M. 2004. A general solution to the graph history interaction problem. In *AAAI'04*, 644–649.

Kishimoto, A., and Müller, M. 2005. Search versus knowledge for solving life and death problems in Go. In *AAAI-05*, 1374–1379.

Kishimoto, A., and Müller, M. 2008. About the completeness of depth-first proof-number search. In *Computers and Games 2008*, volume 5131 of *Lecture Notes in Computer Science*, 146–156.

Kishimoto, A. 2005. *Correct and Efficient Search Algorithms in the Presence of Repetitions*. Ph.D. Dissertation, University of Alberta.

Müller, M. 2003. Proof-set search. In *Computers and Games*, volume 2883 of *Lecture Notes in Computer Science*, 88–107.

Nagai, A. 2002. *Df-pn Algorithm for Searching AND/OR Trees and Its Applications*. Ph.D. Dissertation, University of Tokyo.

Okabe, F. 2005. Application of the route branch number for solving tsume shogi problems (in Japanese). In *10th Game Programming Workshop*, 9–16.

Schaeffer, J.; Burch, N.; Björnsson, Y.; Kishimoto, A.; Müller, M.; Lake, R.; Lu, P.; and Sutphen, S. 2007. Checkers is solved. *Science* 317(5844):1518–1522.

Schaeffer, J. 2001. A gamut of games. *AI Magazine* 2(3):29–46.

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

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

Seo, M.; Iida, H.; and Uiterwijk, J. W. H. M. 2001. The PN*-search algorithm: Application to tsume-shogi. *Artificial Intelligence* 129(1-2):253–277.

Seo, M. 1999. On effective utilization of dominance relations in tsume-shogi solving algorithms (in Japanese). In *8th Game Programming Workshop*, 137–144.

Ueda, T.; Hashimoto, T.; Hashimoto, J.; and Iida, H. 2008. Weak proof-number search. In *Computers and Games 2008*, volume 5131 of *Lecture Notes in Computer Science*, 157–168.