Sibling Conspiracy Number Search

Jakub Pawlewicz
Institute of Informatics
University of Warsaw
pan@mimuw.edu.pl

Ryan Hayward
Department of Computer Science
University of Alberta
hayward@ualberta.ca

Abstract

For some two-player games (e.g. Go), no accurate and inexpensive heuristic is known for evaluating leaves of a search tree. For other games (e.g. chess), a heuristic is known (sum of piece values). For other games (e.g. Hex), only a local heuristic — one that compares children reliably, but non-siblings poorly — is known (cell voltage drop in the Shannon/Anshelevich electric circuit model). In this paper we introduce a search algorithm for a two-player perfect information game with a reasonable local heuristic.

Sibling Conspiracy Number Search (SCNS) is an anytime best-first version of Conspiracy Number Search based not on evaluation of leaf states of the search tree, but — for each node — on relative evaluation scores of all children of that node. SCNS refines CNS search value intervals, converging to Proof Number Search. SCNS is a good framework for a game player.

We tested SCNS in the domain of Hex, with promising results. We implemented an 11-by-11 SCNS Hex bot, DeepHex. We competed DeepHex against current Hex bot champion MoHex, a Monte-Carlo Tree Search player, and previous Hex bot champion Wolve, an Alpha-Beta Search player. DeepHex widely outperforms Wolve at all time levels, and narrowly outperforms MoHex once time reaches 4min/move.

1 Introduction

Consider a 2-player perfect information game with no known global heuristic, but with a reasonable *local* heuristic evaluation (good at relative scoring of children of a node, but bad at comparing non-sibling nodes). Suppose you want to build a bot for this game. What algorithm would you use?

The usual algorithms have drawbacks for a game with only a local heuristic. $\alpha\beta$ Search (Knuth and Moore 1975) needs a globally reliable heuristic. Monte-Carlo Tree Search[1] (Coulom 2007; Browne 2012), which uses random simulations, needs no heuristic but can be slow to converge. Proof-Number Search[2] (Allis, van der Meulen, and van den Herik 1994) performs well as a solver, particularly on search trees with non-uniform branching, but can be weak as a player, especially early in games with search trees with almost uniform branching.

In this paper we introduce Sibling Conspiracy Number Search, an algorithm for a two-player perfect information game with a reasonable local heuristic. SCNS is based on Conspiracy-Number Search (McAllester 1985; 1988), a generalization of PNS — where search tree has leaf scores only ± 1 — obtained by allowing leaf scores to have multiple possible values, e.g. any floating point value in the range from -1 to 1. For a node in a search tree and a target minimax value, the conspiracy number is the minumum number of leaves whose evaluations must change in order for the node's minimax score to reach the target. CNS expands leaves in an order based on conspiracy numbers. SCNS combines features of MCTS (anytime, best-first) and PNS (strong tactically, approaching perfect play near the end of a game).

Hex has a reliable local heuristic[3], so we pick 11×11 Hex as our test domain. We ran DeepHex, our SCNS Hex bot, against an MCTS player (current champion MoHex) and an $\alpha\beta$ player (previous champion Wolve) (Arneson, Hayward, and Henderson 2009; Hayward 2013). DeepHex outperforms Wolve at all time levels, and outperforms MoHex once time reaches 4min/move.

Note that many other games — e.g. Go — have evaluation functions that are weighted combinations of a variety of features, most of which are local. So SCNS might work in those domains as well.

2 Conspiracy Number Search

In 2-player game search, CNS has shown promise in chess (van der Meulen 1988; Schaeffer 1990; Klingbeil and Scha-

Copyright © 2015, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

[1]MCTS is a non-uniform best-first search that uses random simulations to evaluate leaves. Strong moves are exploited; weak moves are explored only if a visit threshold — based on an exploitation/exploration formula such as Upper Confidence Bound (Kocsis and Szepesvári 2006) — is crossed.

[2]PNS is used in *and/or* trees (i.e. each leaf has minimax value ± 1) and is guided by proof and disproof numbers (for each node, the smallest number of descendant leaves that need be 1, resp. -1, for the node to have value 1, resp. -1).

[3]Shannon built an analogue circuit to play the connection game Bridg-it, with moves scored by voltage drop (Gardner 1961). Adding links between virtual connected cells (Anshelevich 2002) improves the heuristic, which although erratic between non-sibling states is reliable among siblings (Henderson 2010). So we use this heuristic for our Hex SCNS bot.

effer 1990; Lorenz et al. 1995; Lorenz 2002; McAllester and Yuret 2002) and shogi (Iida, Sakuta, and Rollason 2002). CNS can be viewed as a generalization of PNS, which is how we will describe our implementation.

2.1 Proof Number Search

Definition 1. Each node n has a *proof number* (pn) p_n and *disproof number* (dn) d_n. A node's (dis)proof number is the smallest number of descendant leaves that, if all true (false), would make the node true (false).

Fact 1. *If a node n is a leaf then*

$$\begin{array}{lll} p_n = 1 & d_n = 1 & \textit{if } n \textit{ is non-terminal} \\ p_n = 0 & d_n = +\infty & \textit{if } n \textit{ is true} \\ p_n = +\infty & d_n = 0 & \textit{if } n \textit{ is false,} \end{array} \tag{1}$$

otherwise

$$\begin{array}{lll} p_n = \min_{s \in \text{child}(n)} p_s, & d_n = \sum_{s \in \text{child}(n)} d_s & \textit{if } n \textit{ is or-node} \\ p_n = \sum_{s \in \text{child}(n)} p_s, & d_n = \min_{s \in \text{child}(n)} d_s & \textit{if } n \textit{ is and-node.} \end{array} \tag{2}$$

Definition 2. *A* most proving node (mpn) is a leaf whose disproof reduces the root's disproof number, and whose proof reduces the the root's proof number.

PNS iteratively selects a most-proving leaf and expands it. See Algorithms 1 and 2.

Algorithm 1 Proof number search

```
1: function PNS(root)
2:     while not root solved do
3:         n ← SELECTMPN(root)
4:         Expand n and initiate new children by (1)
5:         Update nodes along path to the root using (2)
```

Algorithm 2 Proof number search — Selection of mpn

```
1: function SELECTMPN(n)
2:     if n is leaf then
3:         return n
4:     else if n is or-node then
5:         return SELECTMPN( argmin_{s∈children(n)} p_s)
6:     else                                        ▷ n is and-node
7:         return SELECTMPN( argmin_{s∈children(n)} d_s)
```

2.2 Minimax value

PNS is used in two-player zero-sum games. One player is *us*, the other is *them* or *opponent*. Value *true* (*false*) is a win for us (them). We (they) move at an or-node (and-node).

PNS is hard to guide with an evaluation function, as leaves have only two possible values (Allis 1994; Breuker 1998; Nagai 2002; Winands 2004; Saito et al. 2007; Winands and Schadd 2011; Kishimoto et al. 2012). One can extend PNS by allowing a leaf to have any rational value, with $+(-)\infty$ for win(loss). If a leaf is terminal, its value is the actual game value; if not terminal, its value can be assigned heuristically. Internal node values are computed by minimax. We (they) want to maximize (minimize) value. A node from which we (they) move is a *max-node* (*min-node*). Value is computed in minimax fashion as follows, where $\text{EVAL}(n)$ is given by a heuristic (actual) value if n is non-terminal (terminal) (Korf and Chickering 1996):

$$\textsc{Minimax}(n) = \begin{cases} \text{EVAL}(n) & \text{if } n \text{ is leaf,} \\ \max_{s \in \text{children}(n)} \textsc{Minimax}(s) & \text{if } n \text{ is max-node} \\ \min_{s \in \text{children}(n)} \textsc{Minimax}(s) & \text{if } n \text{ is min-node.} \end{cases} \tag{3}$$

2.3 Replacing numbers by functions

PNS is computed using the two final values (true/false) and a temporary value (unknown) assigned to non-terminal leaves. The (dis)proof number measures how difficult it is to change from unknown to true (false). Rather than numbers, we introduce functions to represent the extended set of values denoted by $\mathbb{V} = \{-\infty\} \cup \mathbb{R} \cup \{+\infty\}$. $\mathbb{N}$ is the non-negative integers.

Definition 3. A *proof function* $p_n : \mathbb{V} \mapsto \mathbb{N}$ and *disproof function* $d_n : \mathbb{V} \mapsto \mathbb{N}$ are functions such that, for $v \in \mathbb{V}$,

- $p_n(v)$ is the minimum number of leaves in subtree at n that must change value so that $\textsc{Minimax}(n) \geq v$
- $d_n(v)$ is the minimum number of leaves in subtree at n that must change value so that $\textsc{Minimax}(n) \leq v$.

Rather than storing (dis)proof numbers at each node, we store (dis)proof functions, computed recursively.

Fact 2. *If n is a leaf and $x = \text{EVAL}(n)$ then*

$$\begin{aligned} p_n(v) &= \begin{cases} 0 & \textit{if } v \leq x \\ 1 & \textit{if } v > x \textit{ and } n \textit{ is non-terminal} \\ +\infty & \textit{if } v > x \textit{ and } n \textit{ is terminal,} \end{cases} \\ d_n(v) &= \begin{cases} 0 & \textit{if } v \geq x \\ 1 & \textit{if } v < x \textit{ and } n \textit{ is non-terminal} \\ +\infty & \textit{if } v < x \textit{ and } n \textit{ is terminal,} \end{cases} \end{aligned} \tag{4}$$

otherwise, for every $v \in \mathbb{V}$,

$$\left.\begin{aligned} p_n(v) &= \min_{s \in \text{children}(n)} p_s(v), \\ d_n(v) &= \sum_{s \in \text{children}(n)} d_s(v) \end{aligned}\right\} \textit{ if } n \textit{ is max-node}$$
$$\left.\begin{aligned} p_n(v) &= \sum_{s \in \text{children}(n)} p_s(v), \\ d_n(v) &= \min_{s \in \text{children}(n)} d_s(v) \end{aligned}\right\} \textit{ if } n \textit{ is min-node.} \tag{5}$$

Using Fact 2, (dis)proof functions can be propagated up from leaves. The set of possible game values is usually finite, so a simple way to store function f is as an array of possible results $f(v)$ for each value v.

2.4 Proof and disproof function properties

Fact 3. *For each node n*

(i) *p_n is a non-decreasing staircase function, and d_n is a non-increasing staircase function.*

(ii) *$\textsc{Minimax}(n)$ is the meet point of p_n and d_n, i.e.:*

$$\begin{aligned} p_n(v) = 0, \quad d_n(v) > 0 \quad & \textit{for } v < \textsc{Minimax}(n), \\ p_n(v) = 0, \quad d_n(v) = 0 \quad & \textit{for } v = \textsc{Minimax}(n), \\ p_n(v) > 0, \quad d_n(v) = 0 \quad & \textit{for } v > \textsc{Minimax}(n). \end{aligned}$$

See Figure 1. Following McAllester, the *conspiracy number* $CN_n(v) = p_n(v) + d_n(v)$ is the smallest number of leaves (called conspirators) whose values must change for the minimax value of n to reach v. $CN_n(v) = 0$ iff $v = \textsc{Minimax}(n)$.

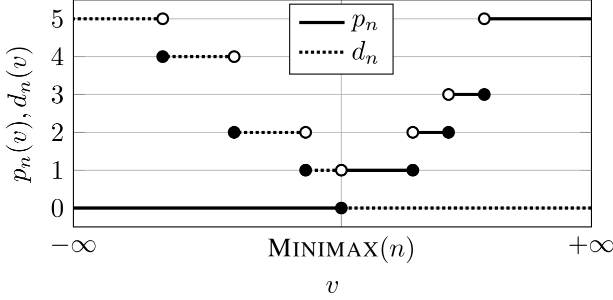

Figure 1: Each proof function p_n (solid segments) and disproof function d_n (dashed segments) is monotonic staircase. Each black dot belongs to its segment (i.e. closed endpoint), each white dot does not (i.e. open endpoint). The intersection of p_n and d_n is the single point $(\textsc{Minimax}(n), 0)$.

2.5 Node expansion

Our implementation of CNS follows the outline of PNS (Algorithm 1): iteratively select and expand a mpn and then update (dis)proof functions on the path to the root. So we define a CNS mpn.

Let $v_{root} = \textsc{Minimax}(root)$. Choose target values $v_{\max}$ for Max (the max player) and $v_{\min}$ for Min so that $v_{\min} < v_{root} < v_{\max}$. We call $[v_{\min}, v_{\max}]$ *the search value interval*, or search interval.[4] We discuss ways to choose $v_{\max}$ and $v_{\min}$ in §2.6. For fixed $v_{\max}$ and $v_{\min}$, we say that Max (Min) wins if value $v_{\max}$ ($v_{\min}$) is reached. To find a mpn we use $\textsc{SelectMPN}(root)$, with pn $p_n(v_{\max})$ and dn $d_n(v_{\min})$ for every node n. Algorithm 3 is our implementation, which differs from that of McAllester: we alter both sides of the search interval at once.

2.6 Choosing the search interval

Let $v_{root} = \textsc{Minimax}(root)$ denote the minimax value of the root. A standard way to set the search interval is to set $v_{\max}$ and $v_{\min}$ close enough to v_{root} so that the number of leaf expansions required to reach $v_{\max}$ or $v_{\min}$ is within some threshold, i.e.

$$\begin{aligned} v_{\max} &= \max_{v \in \mathbb{V}} \{v : p_{root}(v) \le P_{\max}\} \\ v_{\min} &= \min_{v \in \mathbb{V}} \{v : d_{root}(v) \le D_{\max}\}, \end{aligned} \tag{6}$$

where thresholds $P_{\max}$ and $D_{\max}$ are (possibly equal) constants. Search proceeds until either $v_{\max} - v_{\min} \le \Delta$, where Δ is an acceptable error tolerance (McAllester 1988; Klingbeil and Schaeffer 1990), or a time limit is reached.

[4]This is the current likely range of final minimax value of the root. This is analogous to the aspiration window of $\alpha\beta$ search.

Algorithm 3 Conspiracy number search

```
 1: function CNS(root)
 2:     while not reached time limit do
 3:         SetInterval                          ▷ Set v_max and v_min
 4:         n ← SelectMPN(root)
 5:         Expand n and initiate new children by (4)
 6:         Update nodes along path to the root using (5)
 7: function SelectMPN(n)
 8:     if n is leaf then
 9:         return n
10:     else if n is max-node then
11:         return SelectMPN( argmin_{s∈children(n)} p_s(v_max))
12:     else                                     ▷ n is min-node
13:         return SelectMPN( argmin_{s∈children(n)} d_s(v_min))
```

2.7 Choosing the best move

To use CNS as a player, we run the search until error tolerance is reached or time runs out and then pick the best move. But which move is best? In MCTS, there are two usual candidates for move selection: 1) the move with the best win rate, or 2) the move on which most time — leaf expansions in its subtree — has been spent. Depending on the MCTS domain, either criterion can be preferred. Here, we cannot use 1), since there is no known good heuristic.[5] So we have 2) or 3): some form of minimax search. We experimented with forms of 3), including that of (Lorenz 2002), but all performed much worse than 2).

One problem with 2) is when the effective search depth is insufficient to reveal the strength of the best move. Here CNS can choose a move before adequately exploring others. We remedy this problem in SCNS (§3).

2.8 Efficient storage of proof function

If the granularity of an evaluation function is high then an array indexed by all possible function values takes much space. One fix is to bucket function values, but this worked poorly for us. Instead, we exploit the staircase nature of the (dis)proof functions, storing only the stair steps. Each step is represented by a number pair: a rational — step width (minimax value range), and an integer — step height (conspiracy number range). To store a leaf's proof function leaf we need

[5]Experiments showed the circuit resistance heuristic to be weak.

only one pair. The size (in steps) of an internal node's proof function is at most the sum of the sizes of its children's proof functions. So proof functions for nodes near the tree bottom are small, and total proof function storage is proportional to tree size.

We implement CNS in depth-first fashion[6] using recursion and a transposition table, so over time unimportant states are dropped from memory and most remaining nodes have multi-step proof functions. So we need to further reduce proof function storage.

For proof function f with $|f(v_1) - f(v_2)|$ small, merging the steps for v_1 and v_2 has little impact on performance. So we approximate step height:

Definition 4. The upper bound approximation $\hat{f}$ of f with parameter $\eta \in (0, 1)$ is defined as

$$\hat{f}(v) = \max_{v' \in \mathbb{V}} \{f(v') \, : \, f(v)(1 + \eta) \geq f(v')\}. \tag{7}$$

$\hat{f}$ approximates f via a sequence of steps, each differing slightly from its successor. E.g. for $\eta = 0.01$, steps of height 100 and 101 are merged as a step of height 101. This introduces little error, especially for games with transpositions: in such games, the true game DAG is often approximated by a game tree, so high proof numbers are already less accurate than low ones. So it is often not necessary to distinguish between large but close proof numbers.

Lemma 1. *For two consecutive steps of $\hat{f}$ with heights $p_1 < p_2$, $p_1(1 + \eta) < p_2$.*

Theorem 1. *The number of steps of $\hat{f}$ is at most $O(\log p)$, where p is the maximum proof number, i.e. $p = f(+\infty)$.*

Proof. For any base b, the number of steps is at most

$$\log_{1+\eta} p = \frac{\log_b p}{\log_b(1 + \eta)} \approx \frac{1}{\eta} \log_b p = O(\log p).$$

□

This approximation works well with η as large as .25: e.g., if proof numbers oscillate around 1000, the number of steps is then (at most) around 25.

3 Sibling CNS

Here we describe how we convert a local heuristic — one that reliably scores relative strengths of siblings — into a global heuristic useful for our CNS player.

Definition 5. Let n be a node. For every child s of n, let $E(n \to s)$ be the score — positive and rational — of the move from n to s. Let s_0 be the best child score, i.e.

$$s_0 = \underset{s \in \text{children}(n)}{\text{argmax}} \; E(n \to s). \tag{8}$$

Define the *relative error* $e(n \to s)$ of s as

$$e(n \to s) = \log \frac{E(n \to s_0)}{E(n \to s)}. \tag{9}$$

So $e(n \to s) = 0$ if s is as strong as the best move, otherwise $e(n \to s) > 0$. This relative error measures divergence from optimal play. Now we have our evaluation function.

Definition 6. Let n be the game tree node found by descending from the root by the path

$$\textit{root} = p_0 \to p_1 \to \cdots \to p_k = n. \tag{10}$$

Also, let

$$\sigma(p_i) = \begin{cases} -1 & \text{if we are to move in } p_i \\ 1 & \text{if the opponent is to move in } p_i. \end{cases} \tag{11}$$

Then the evaluation of a non-terminal node n is defined as

$$\text{EVAL}(n) = \sum_{i=1}^{k} \sigma(p_{i-1}) \cdot e(p_{i-1} \to p_i) \tag{12}$$

We call this *siblings comparison evaluation function* (SCEF).

Consider SCEF when applied to CNS with a small search interval. Set $P_{\max} = D_{\max} = 1$ and use (6) to set the search interval $(v_{\min}, v_{\max})$. Then CNS works as follows. First, CNS follows the path, say π_0, from root to a terminal

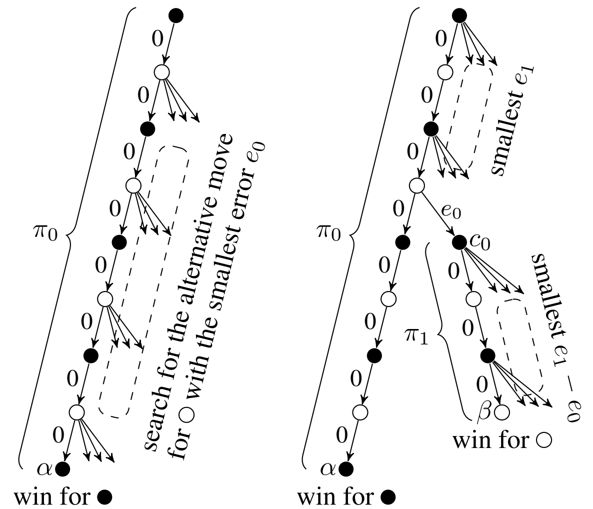

Figure 2: The first few steps of SCNS.

state via moves with error zero (best possible). Along the way, it expands all siblings of nodes on π_0. Assume that the terminal node, say α, wins for us. Then CNS searches for a child c_0 of an opponent node from π_0 with smallest possible error e_0, and the search branches from c_0. From c_0 it follows a path π_1 using moves with error zero until it reaches a terminal node; again siblings of all encountered nodes are expanded. Now assume that this terminal node, say β, is a loss for us. Then CNS tries to diverge from the current terminal path, either before c_0 on π_0, or at or after c_0 on π_1. CNS tries to find a child of one of our nodes with the smallest error e_1 or, if this is on π_1, with smallest error $e_1 - e_0$. See Figure 2.

Generally, CNS constructs paths to terminal nodes, and then branches so that the player for whom the terminal node

[6]Our CNS implementation is similar to that of DFPNS, a depth-first version of PNS (Nagai 2002; Pawlewicz and Lew 2007).

was losing tries to find another response in a subtree minimizing the cumulative error. So, the player tries to fall back on another most promising move of the entire tree.

This behaviour seems close to that of humans, who often follow the best line until finding it bad for one player, at which point they seek a deviation helping that player.

So CNS with SCEF is SCNS — Sibling Conspiracy Number Search. Experiments show that SCNS works well. It often explores deep lines of play, dealing well with long forcing sequences (ladders), while still widening the game tree by the most promising moves.

3.1 Example

Here we illustrate an example run of SCNS. We assume that each terminal node evalutes either as $-\infty$ (loss) or $+\infty$ (win). Each non-terminal leaf is evaluated by SCEF. We introduce a compact notation for (dis)proof functions.

Definition 7. Let n be a non-terminal node and let k, l be non-negative integers. An increasing sequence $\langle v_{-k} \cdots v_{-1}\, \boldsymbol{v_0}\, v_1 \cdots v_l \rangle$, where element v_0 is marked, represents a pair of functions p_n and d_n such that

$$d_n(v) = \begin{cases} 0 & \text{for } v \in [v_0, +\infty), \\ i+1 & \text{for } 0 \le i \le k+1 \text{ and } v \in [v_{-i-1}, v_{-i}), \end{cases}$$
$$p_n(v) = \begin{cases} 0 & \text{for } v \in (-\infty, v_0], \\ i+1 & \text{for } 0 \le i \le l+1 \text{ and } v \in (v_i, v_{i+1}], \end{cases} \tag{13}$$

where we assume that $v_{-k-1} = -\infty$ and $v_{l+1} = +\infty$. Thus $v_0 = \text{MINIMAX}(n)$. If the desired minimax value we hope to achieve by search is at most v_0, we need expand no leaves; if it is greater than v_0 we must expand at least one leaf. Starting from v_1 we need to expand at least two leaves, and so on. For the opponent we read the sequence similarly starting from v_0, but in the opposite direction.

For each tree leaf n, we have $k = l = 0$. E.g., if a leaf n has value 5, then p_n and d_n are each represented by $\langle \mathbf{5} \rangle$. Terminal leaves can evaluate only to $-\infty$ and $+\infty$, so we overload the meaning of $\langle -\infty \rangle$ and $\langle +\infty \rangle$:

Definition 8. By $\langle -\infty \rangle$ ($\langle +\infty \rangle$) we denote a pair of functions p_n and d_n for losing (winning) state n.

Now we can start our example.

Initialization. We expand the root node r, revealing children a and b as in Figure 3. Edge labels denote relative move error, so $e(r \to a) = 0$ and $e(r \to b) = 1$. (Dis)proof functions are shown beside the corresponding node. A square (circle) is a state where we (the opponent) are to move.

The SCEF evaluation of b is -1, since on the path from the root SCEF alternately multiplies move errors by -1 and 1 depending on whether the error is ours or the opponent's.

Iteration 1. We find the search interval by (6) with $P_{\max} = D_{\max} = 1$: $v_{\max} = +\infty$, $v_{\min} = -1$. We descend from the root to a mpn. We compare values of proof functions at $v_{\max}$ for all children of the root: $p_a(+\infty) =$

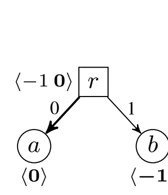

Figure 3: Initial tree.

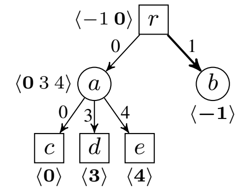

Figure 4: After iteration 1.

$p_b(+\infty) = 1$. These values are equal, so we can choose either node. We select a, since it has the smaller relative move error. Node a is a leaf so it is a mpn. The path from the root to this leaf is marked by thick edges in Figure 3.

We expand a and update (dis)proof functions along the path to the root, yielding the tree in Figure 4. Leaves d and e each have positive SCEF score, as each is reached with no player relative error plus positive opponent relative error.

Iteration 2. The search interval $(v_{\min}, v_{\max})$ is again $(-1, +\infty)$. Now $p_a(+\infty) = 3 > p_b(+\infty) = 1$ so we descend to b, and it is a mpn. We expand b and update (dis)proof functions, yielding the tree in Figure 5.

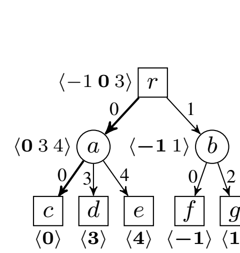

Figure 5: After iteration 2.

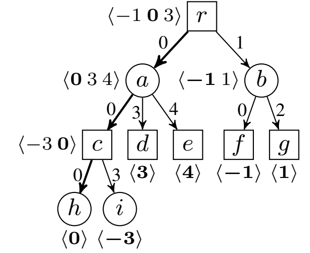

Figure 6: After iteration 3.

Iteration 3. Now $(v_{\min}, v_{\max}) = (-1, 3)$, and $p_a(3) = 1 < p_b(3) = 2$, so we select a. For a's children we apply disproof functions at $v_{\min}$: $d_c(-1) = d_d(-1) = d_e(-1) = 1$. We select c, the child with the smallest relative move error. We expand it and update (dis)proof functions, resulting in the tree shown in Figure 6.

Iteration 4. The (dis)proof functions of r and a did not change, so we have the same search interval as in iteration 3, and we first select a. For a's children we again have $d_c(-1) = d_d(-1) = d_e(-1) = 1$, so we select c, and from c we select h and expand. Assume that h has only one child j, and that j loses. This triggers updates of all functions along path to the root, shown in Figure 7.

Iteration 5. Until now SCNS has explored the best line of play, and also some siblings. But after reaching this losing state SCNS must now diverge and find for us a best response. The new search interval is $(-3, 3)$. We still have $p_a(3) =$

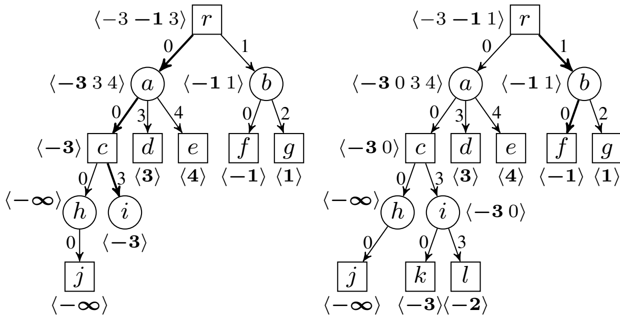

Figure 7: After iteration 4. Figure 8: After iteration 5.

$1 < p_b(3) = 2$, and $d_c(-3) = 1$, so we select i and expand. See Figure 8.

In the next iteration $v_{\max} = 1$ and $p_a(1) = 2 > p_b(1) = 1$, so the search will switch to branch b and develop another line of play. We could set $P_{\max}$ and $D_{\max}$ to values greater than 1, which would broaden the search and develop further lines of play, possibly strengthening performance.

3.2 SCEF and minimax

To better understand SCEF, consider how minimax search would behave using SCEF to evaluate leaf nodes. Consider a minimax search that fails to reach any terminal position, e.g. any search early in the game. Consider the principle variation, i.e. the path that starts at the root and follows moves with relative error 0. If one player deviates from the principle variation by a move with positive relative error, then from that point the opponent can always follow moves with relative error 0, giving the player a negative score. So for this search, the minimax value is 0 and a best move is any root move with relative error 0.

So it might not be useful to use SCEF inside any variation of minimax, e.g. $\alpha\beta$ search, as this would result in simply picking depth-0 best moves until perhaps the middle of the game, by which point any reasonable opponent would presumably have a crushing advantage.

3.3 Avoiding unpromising sibling expansion

Although SCNS explores good lines of play, the version we have described so far is wasteful, as it expands all siblings whenever a new child is expanded. Let us explain why. In Algorithm 3, when function SELECTMPN arrives at a max-node n whose children are all leaves, then $p_n(v_{\max}) = 1$ and the same holds for all n's children. This is because leaf (dis)proof functions are initialized by (4). See Figure 9. Now SELECTMPN can call any child. The best option is to call the child with smallest move error. Here s_1 is best if error $e_1 = 0$. Now, even if s_1 is expanded, $p_n(v_{\max})$ will not change because of the other children, so remains 1. Thus the next leaf to be expanded will be one of n's remaining children. Notice that at this moment SCNS does not distinguish among children s_i, $i = 1, 2, 3$, even if their evaluations $\text{EVAL}(s_i)$ vary. This is a drawback of CNS in general.

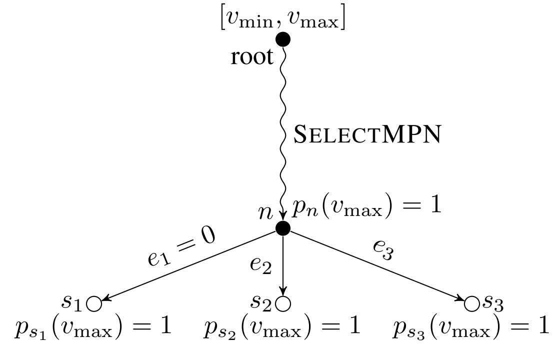

Figure 9: Illustration of the effect of siblings expansion.

To avoid this unnecessary expansion, especially for unpromising children with relatively high move error, we encode extra information in the (dis)proof function when creating a leaf. If a move has high error compared to its best sibling, then to increase the minimax value of this move by this error will likely require many expansions. So, rather than initializing (dis)proof functions in two steps (4), we use a more complicated initialization process whose number of steps is logarithmic in the difference of a value from the minimax value. Hence

$$p_n(v) = \begin{cases} 0 & \text{if } v \leq x \\ i & \text{if } i^\delta < 2^{(v-x)} \leq (i+1)^\delta \end{cases}$$
$$d_n(v) = \begin{cases} 0 & \text{if } v \geq x \\ i & \text{if } i^\delta < 2^{(x-v)} \leq (i+1)^\delta \end{cases} \tag{14}$$

where $x = \text{MINIMAX}(v)$, i is a positive integer and δ a positive rational. See Figure 10. Using (14) to initialize non-

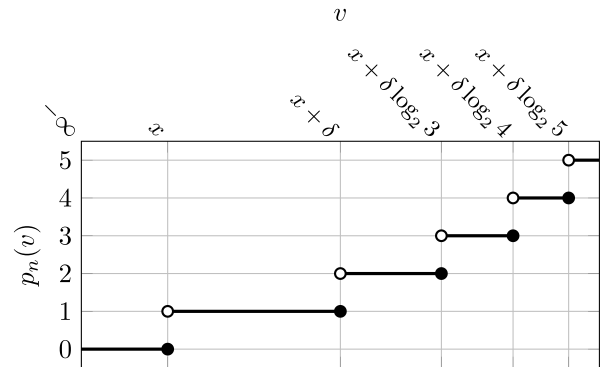

Figure 10: Proof function p_n for leaf.

terminal leafs, SCNS expands only siblings whose score diverges from that of the best sibling by at most δ. Depending on how values shift during search, other (weaker) siblings might be expanded if the minimax value changes by more than δ. With this modification, SCNS's search behaviour is now closer to that of the human-like behaviour described above.

4 Experimental Results

Using SCNS with the Shannon/Anshelevich local heuristic (energy drop at each cell) described in §1, we implemented a Hex bot DeepHex on the open-source Benzene framework (Arneson, Henderson, and Hayward 2009).

We used two bots as opponents: Wolve and MoHex, each also implemented on Benzene. Wolve uses $\alpha\beta$ Search, using circuit resistance (with pruning to improve performance) as evaluation function. MoHex — the strongest Hex bot since 2009 — uses MCTS with RAVE, patterns, prior knowledge estimation, progressive bias, and CLOP tuning of parameters (Huang et al. 2014). Both Wolve and MoHex also compute virtual connections that prune moves and help solve positions long before the game ends.

To compare the strength of these three bots, we ran a round-robin tournament. We used 36 relatively balanced 1-move openings: a2 to k2, a10 to k10, b1 to j1, and b11 to j11. For each bot, we ran 5 versions, one each with time limit 30s, 1m, 2m, 4m, and 8m per move. Thus there were 15 bot competitors in the tournament. Each version played each other version two times on each opening, once as black (1st-player) and once as white (2nd-player). So each version played 1008 of the 7560 tournament games. The results are scored by BayesElo (Coulom 2010) in Figure 11.

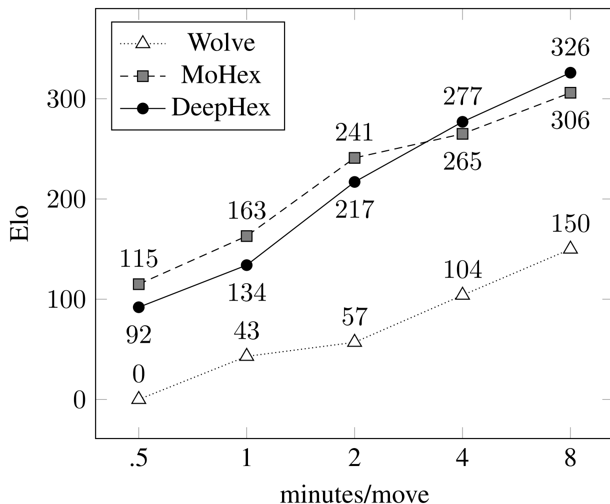

Figure 11: Tournament results. Each point has error within 14 Elo, with 80% confidence.

Overall, DeepHex is similar in strength to MoHex. With short time per move, DeepHex is weaker. But this strength gap decreases with time, with DeepHex 12 Elo ahead at 4m/move (although error is up to 14 Elo with 80% confidence) and 20 Elo ahead at 8m/move. This perhaps shows SCNS adapting more quickly than MCTS to new lines of play. Also, MoHex use its knowledge computations (virtual connections) to shape the growth of its tree, while SCNS does not. Without this optimization MoHex's strength deteriorates more quickly (Arneson, Hayward, and Henderson 2010).

Figure 12 shows an 8m/move win of DeepHex over MoHex. MoHex — with steady early play — reaches a winning position. But DeepHex recognizes the situation before MoHex, and quickly takes advantage once MoHex blunders.

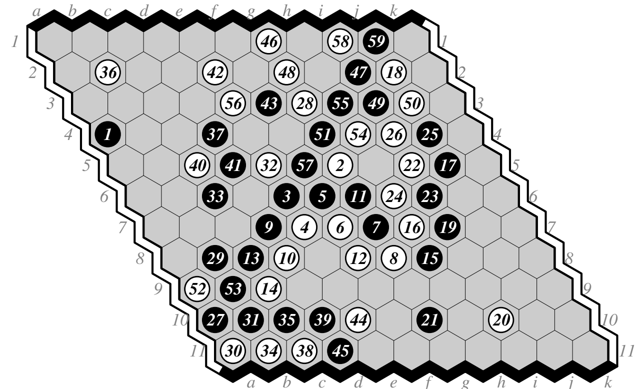

Figure 12: An 8min/move DeepHex (Black) win over MoHex. With **26**.i4 MoHex has winrate .51 and PV a10 g3 g4 f4 a8 c7 d5 d4 a6 a5 b5 c5 e4 f2 i3 i2 j3 a11 f5 h3 h4 h9 g9 d10 e9 e10. With **27**.a10 DeepHex finds a loss – unproven due to pruning – with PV f4 h4 h5 g4 f5 b8 d5 a6 a5 b5 a11 b10 b11 c10 c11 d10 c3 b4 b2 i3 j3 e4 e5 c2 b3. But MoHex blunders with **28**.g3: winrate is .52 but DeepHex sees a huge advantage. With **33**.c6 DeepHex finds a proven win, PV d1 d4 e3 d2 e10 b11 c2 d3 e1 e2 f1 f2 g1 g2 h1 h2 i1 b2 b3 a3 b4 a5 b6 b5 c5 c4 d5 f5. By **38**.c11 MoHex finds a proven loss.

5 Conclusions and further research

We introduce Sibling Conspiracy Number Search, a version of Conspiracy Number Search designed to work with a local heuristic (i.e. one that reliably estimates move strength when compared to its siblings).

We implemented SCNS in Hex, creating the bot DeepHex, which we compared to the champion bot MoHex, an MCTS player, and previous champion Wolve, an $\alpha\beta$ player. DeepHex outperforms Wolve at all time levels, and outperforms MoHex once time reaches 4min/move.

Directions for future work include testing SCNS in other domains. We suspect that SCNS would work well in any domain with a reliable local heuristic.

Acknowledgments.

This research was supported by an NSERC Discovery Grant.

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, Maastricht, Netherlands.

Anshelevich, V. V. 2002. A hierarchical approach to computer Hex. *Artificial Intelligence* 134(1–2):101–120.

Arneson, B.; Hayward, R. B.; and Henderson, P. 2009. Wolve 2008 wins Hex tournament. *ICGA* 32(1):49–53.

Arneson, B.; Hayward, R. B.; and Henderson, P. 2010. Monte Carlo Tree Search in Hex. *IEEE Transactions on Computational Intelligence and AI in Games* 2(4):251 –258.

Arneson, B.; Henderson, P.; and Hayward, R. B. 2009. Benzene. http://benzene.sourceforge.net/.

Breuker, D. M. 1998. *Memory versus Search in Games*. Ph.D. Dissertation, Maastricht University, Maastricht, Netherlands.

Browne, C. 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games* 4(1):1–49.

Coulom, R. 2007. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In *Computers and Games*, Springer LNCS 4630, 72–83.

Coulom, R. 2010. Bayesian elo rating. http://remi.coulom.free.fr/Bayesian-Elo.

Gardner, M. 1961. *The 2nd Scientific American Book of Mathematical Puzzles and Diversions*. New York: Simon and Schuster. chapter 7, 78–88.

Hayward, R. B. 2013. Mohex wins Hex tournament. *ICGA Journal* 36(3):180–183.

Henderson, P. 2010. *Playing and solving Hex*. Ph.D. Dissertation, UAlberta. http://webdocs.cs.ualberta.ca/~hayward/theses/ph.pdf.

Huang, S.-C.; Arneson, B.; Hayward, R. B.; Müller, M.; and Pawlewicz, J. 2014. Mohex 2.0: A pattern-based mcts hex player. In *Computers and Games*, LNCS 8427. Springer. 60–71.

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

Kishimoto, A.; Winands, M.; Müller, M.; and Saito, J.-T. 2012. Game-tree searching with proof numbers: the first twenty years. *ICGA Journal* 35(3):131–156.

Klingbeil, N., and Schaeffer, J. 1990. Empirical results with conspiracy numbers. *Computational Intelligence* 6:1–11.

Knuth, D. E., and Moore, R. W. 1975. An analysis of alpha-beta pruning. *Artificial Intelligence* 6(4):293–326.

Kocsis, L., and Szepesvári, C. 2006. Bandit based monte-carlo planning. In *ECML*, Springer LNCS 4212, 282–293.

Korf, R., and Chickering, D. 1996. Best-first minimax search. *Artif. Intell.* 84(1-2):299–337.

Lorenz, U.; Rottmann, V.; Feldman, R.; and Mysliwietz, P. 1995. Controlled conspiracy number search. *ICCA Journal* 18(3):135–147.

Lorenz, U. 2002. Parallel controlled conspiracy number search. In *Euro-Par 2002*, LNCS 2400. Springer. 420–430.

McAllester, D., and Yuret, D. 2002. Alpha-beta conspiracy search. *ICGA* 25(1):16–35.

McAllester, D. 1985. A new procedure for growing min-max trees. Technical report, Artificial Intelligence Laboratory, MIT, Cambridge, MA, USA.

McAllester, D. 1988. Conspiracy numbers for min-max search. *Artif. Intell.* 35(3):287–310.

Nagai, A. 2002. *Df-pn Algorithm for Searching AND/OR Trees and Its Applications*. Ph.d. thesis, Dept. Info. Science, University Tokyo, Tokyo, Japan.

Pawlewicz, J., and Lew, L. 2007. Improving depth-first pn-search: $1+\varepsilon$ trick. In *Computers and Games 2006*, LNCS 4630. Springer. 160–170.

Saito, J.-T.; Chaslot, G.; Uiterwijk, J.; and van den Herik, H. 2007. Monte-carlo proof-number search for computer go. In *Computers and Games*, LNCS 4630. Springer. 50–61.

Schaeffer, J. 1990. Conspiracy numbers. *Artificial Intelligence* 43(1):67–84.

van der Meulen, M. 1988. Parallel conspiracy-number search. Master's thesis, Vrije Universiteit Amsterdam, the Netherlands.

Winands, M., and Schadd, M. 2011. Evaluation-function based proof-number search. In *Computers and Games 2010*, LNCS 6515. Springer. 23–35.

Winands, M. 2004. *Informed Search in Complex Games*. Ph.D. Dissertation, Universiteit Maastricht, Netherlands.