Jakub Pawlewicz and Ryan B. Hayward, Conspiracy number search with relative sibling scores, Theoretical Computer Science, Vol 644, 2016 Sep 6, pp 127-142

Conspiracy Number Search with Relative Sibling Scores

Jakub Pawlewicz[1] and Ryan B. Hayward[2]

[1] Institute of Informatics, University of Warsaw, pan@mimuw.edu.pl
[2] Computing Science, University of Alberta, hayward@ualberta.ca.
Research supported by Natural Sciences and Engineering Research Council of Canada.

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.
We tested the strength of SCNS features: most critical is to initialize leaves via a multi-step process. Also, we show a simple parallel version of SCNS: it scales well for 2 threads but less efficiently for 4 or 8 threads.

Keywords: conspiracy number search, search algorithm, Hex

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 [20] needs a globally reliable heuristic. Monte-Carlo Tree Search[3] [9,8], which

[3] 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 [21] — is crossed.

uses random simulations, needs no heuristic but can be slow to converge. Proof-Number Search[4] [2] 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 [25,26], which generalizes PNS: in PNS, each search tree leaf score is -1 or 1, while in CNS a leaf score can have any value — e.g. any floating point value in the range from -1 to 1 — that indicates an associated final game score. For a node in a search tree and a target minimax value, the conspiracy number is the minimum 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 that is 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). We will explain CNS and SCNS in further detail later.

Hex has a reliable local heuristic[5], 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) [4,14]. DeepHex outperforms Wolve at all time levels, and outperforms MoHex once time reaches 4min/move.

Next, we measure the relative contribution of the feature enhancements of our SCNS Hex bot, and measure the performance of a parallel implementation.

2 Conspiracy Number Search

In 2-player game search, CNS has shown promise in chess [35,34,19,24,23,27] and shogi [17]. 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)[6].

[4] 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).

[5] Shannon built an analogue circuit to play the connection game Bridg-it, with moves scored by voltage drop [12]. Adding links between virtual connected cells [3] improves the heuristic, which although erratic between non-sibling states is reliable among siblings [15]. So we use this heuristic for our Hex SCNS bot.

[6] In PNS, a leaf node with value true indicates that the search goal is reached. Usually the search goal is to determine the game win/loss value, but it could be any desired search goal, e.g. in chess indicating the capture of a queen, or that the game ends in a win or draw but not a loss.

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

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

Definition 2. *A* most proving node (mpn) is a leaf whose proof will reduce the root's proof number and whose disproof will reduce the root's disproof 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:         $n \leftarrow$ 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($\operatorname{argmin}_{s \in \text{children}(n)} p_s$)
6:     else                                                    ▷ $n$ is and-node
7:         return SELECTMPN($\operatorname{argmin}_{s \in \text{children}(n)} d_s$)
```

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 game values (i.e. minimax outcomes) [1,7,28,36,33,37,18]. 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. We (they) want to maximize (minimize) value. A node from which we (they) move is a *max-node* (*min-node*). Internal node values are computed in minimax fashion. $\textsc{Minimax}(n)$ denotes the minimax value of node n. Value is computed in minimax fashion as follows, where $\textsc{eval}(n)$ is given by a heuristic (actual) value if n is non-terminal (terminal) [22]:

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

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 is the number of nodes whose values need to change from unknown to true (false). Rather than numbers, we use functions to represent the extended set of values denoted by $\mathbb{V} = \{-\infty\} \cup \mathbb{R} \cup \{+\infty\}$.

The simplest possible representation of (value, proof number) — and similarly for (value, disproof number) — pairs for each node would be using a map f from value to to proof number, i.e. for each possible value v which can be output by the evaluation function, there would be an entry in the map f, with $f(v)$ giving the proof number. However, such a representation is feasible only if the the evaluation function has a finite and small number of possible outcomes. That is not the case here, so we will need a different representation, as we explain below. The idea is to represent this set with a step function with a finite number of steps, so the set can be represented with an array with a finite number of entries.

Definition 3. The function $p_n : \mathbb{V} \mapsto \mathbb{N}_0 = \{0, 1, 2, \ldots\}$ is a *proof function* if, for all $v \in \mathbb{V}$, $p_n(v)$ is the minimum number of leaves in the subtree rooted at n that must change value so that $\textsc{Minimax}(n) \geq v$. Similarly, $d_n : \mathbb{V} \mapsto \mathbb{N}_0$ is a *disproof function* if, for all $v \in \mathbb{V}$, $d_n(v)$ is the minimum number of leaves in the subtree rooted 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 $x = \textsc{eval}(n)$ 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 $v \in \mathbb{V}$,

$$\begin{aligned} p_n(v) &= \min_{s \in \text{children}(n)} p_s(v), \quad d_n(v) = \sum_{s \in \text{children}(n)} d_s(v) \quad \text{if } n \text{ is or-node,} \\ p_n(v) &= \sum_{s \in \text{children}(n)} p_s(v), \quad d_n(v) = \min_{s \in \text{children}(n)} d_s(v) \quad \text{if } n \text{ is and-node.} \end{aligned} \tag{5}$$

(Dis)Proof functions can be propagated up from leaves by Fact 2. One way to represent such a function f is as an array of all possible values $f(v)$ for each 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) $\text{MINIMAX}(n)$ is the meet point of p_n and d_n, i.e.:

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

See Figure 1. Following McAllester, the *conspiracy number* $CN_n(v) = p_n(v) +$

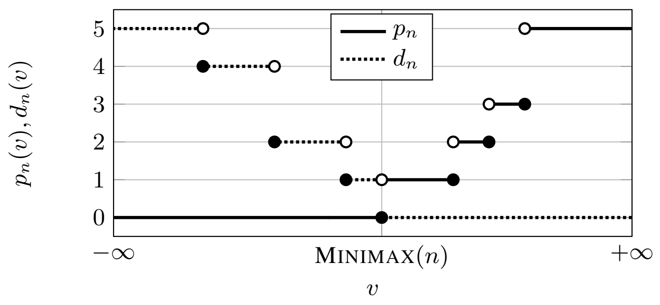

Fig. 1. Each proof function p_n (solid segments) and disproof function d_n (dashed segments) is monotonic staircase. Each black dot belongs to its segment (i.e. closed endpoint), each white dot does not (i.e. open endpoint). The intersection of p_n and d_n is the single point $(\text{MINIMAX}(n), 0)$.

$d_n(v)$ 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 $v = \text{MINIMAX}(n)$.

2.5 Node expansion

Our implementation of CNS follows PNS: iteratively select and expand a most proving node (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 explain how to do this in §2.6. We call $[v_{\min}, v_{\max}]$ *the search value interval* or search interval.[7] 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. Our CNS implementation — Algorithm 3 — differs from that of McAllester, as we alter both sides of the search interval at once.

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:         $n \leftarrow$ 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( $\operatorname{argmin}_{s \in \text{children}(n)} p_s(v_{\max})$ )
12:     else                                               ▷ $n$ is min-node
13:         return SELECTMPN( $\operatorname{argmin}_{s \in \text{children}(n)} d_s(v_{\min})$ )
```

2.6 Choosing the search interval

One way to pick the search interval is to set $v_{\max}$ and $v_{\min}$ a fixed difference from $\textsc{minimax}(root)$, denoted v_{root},

$$\begin{aligned} v_{\max} &= v_{root} + \delta_p, \\ v_{\min} &= v_{root} - \delta_d, \end{aligned} \tag{6}$$

where δ_p and δ_d are possibly equal constants. But it can help to modify the interval during search, e.g. by adjusting according to the root (dis)proof value,

$$\begin{aligned} v_{\max} &= \max_{v \in \mathbb{V}} \{v \,:\, p_{root}(v) \leq P_{\max}\}, \\ v_{\min} &= \min_{v \in \mathbb{V}} \{v \,:\, d_{root}(v) \leq D_{\max}\}, \end{aligned} \tag{7}$$

where $P_{\max}$ and $D_{\max}$ are possibly equal constants. This approach was used in the original CNS algorithm [26,19]. Search proceeds until the interval is sufficiently small, i.e. $v_{\max} - v_{\min} \leq \varDelta$, where $\varDelta$ is a constant indicating an acceptable error tolerance.

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

This method does not always converge, e.g. when the search is close to solving a position, or — if thresholds are too small — when the search stumbles into a stable position; in such cases it is better to increase thresholds and resume the search. Our approach below mixes (6) and (7). Notice that (8) generalizes (7), and also (6) if Pmax or Dmax $= 0$.

$$\begin{aligned} v_{\max} &= \max_{v \in \mathbb{V}} \{v \,:\, p_{\mathit{root}}(v) \leq \max(p_{\mathit{root}}(v_{\mathit{root}} + \delta_p), P_{\max})\}, \\ v_{\min} &= \min_{v \in \mathbb{V}} \{v \,:\, d_{\mathit{root}}(v) \leq \max(d_{\mathit{root}}(v_{\mathit{root}} - \delta_d), D_{\max})\}. \end{aligned} \tag{8}$$

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.[8] So we have 2) or a third candidate, namely 3) some form of minimax search. We experimented with forms of 3), including that of [23], 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 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 (See §3.5) 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

$$\hat{f}(v) = \max_{v' \in \mathbb{V}} \{f(v') \,:\, f(v)(1 + \eta) \geq f(v')\}. \tag{9}$$

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

$\hat{f}$ 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 directed acyclic graph of the state space (so nodes are states, and a state is a child of another if it can be reached by a legal move from that state) 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

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.

$$s_0 = \operatorname*{argmax}_{s \in \text{children}(n)} E(n \to s). \tag{10}$$

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

$$e(n \to s) = \log \frac{E(n \to s_0)}{E(n \to s)}. \tag{11}$$

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

$$\mathit{root} = p_0 \to p_1 \to \cdots \to p_k = n. \tag{12}$$

Also, let

$$\sigma(p_i) = \begin{cases} -1 & \text{if we are to move in } p_i \\ 1 & \text{if the opponent is to move in } p_i. \end{cases} \tag{13}$$

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

$$\textsc{Eval}(n) = \sum_{i=1}^{k} \sigma(p_{i-1}) \cdot e(p_{i-1} \to p_i) \qquad (14)$$

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 (7) 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 state via moves

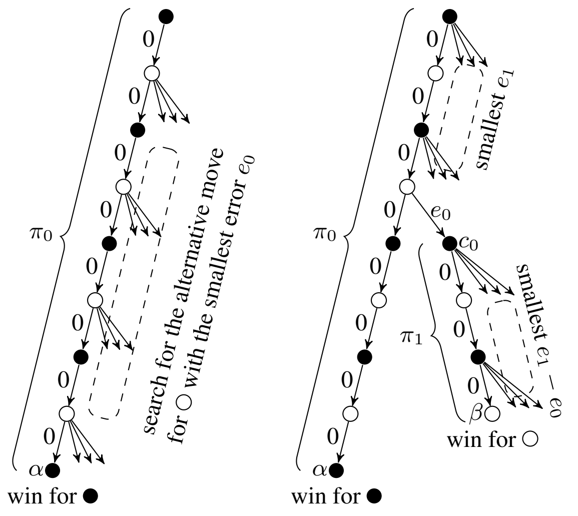

Fig. 2. The first few steps of SCNS.

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 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 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.

3.2 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 3. Now SELECTMPN

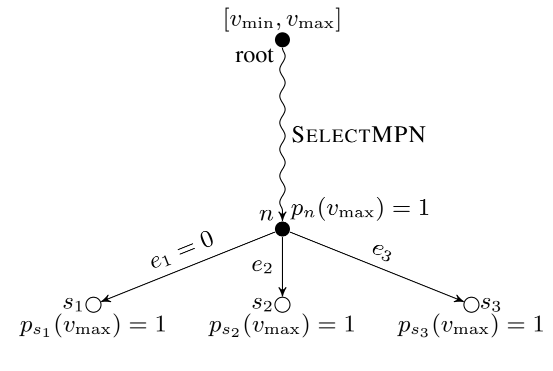

Fig. 3. Illustration of the effect of siblings expansion.

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 $\textsc{Eval}(s_i)$ vary. This is a drawback of CNS in general.

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

$$\begin{aligned} p_n(v) &= \begin{cases} 0 & \text{if } v \leq x \\ i & \text{if } i^\delta < 2^{(v-x)} \leq (i+1)^\delta \end{cases} \\ d_n(v) &= \begin{cases} 0 & \text{if } v \geq x \\ i & \text{if } i^\delta < 2^{(x-v)} \leq (i+1)^\delta \end{cases} \end{aligned} \tag{15}$$

where $x = \textsc{Minimax}(n)$, i is a positive integer and δ a positive rational. See Figure 4. Using (15) to initialize non-terminal leafs, SCNS expands only siblings whose score

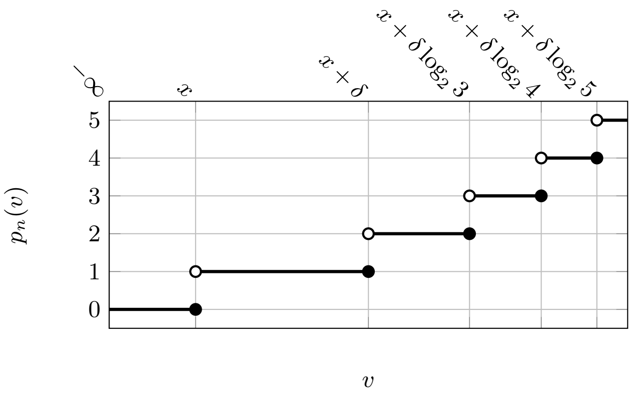

Fig. 4. Proof function p_n for leaf.

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.

3.3 Gradual forgetting of an error

While cell energy is effective in SCEF as a move's error estimate, it can assign a falsely high error to a good move. If SCNS spends much work[9] at such a move the initial error estimate should be corrected. We gradually decrease error as follows,

$$e'(n \to s) = e(n \to s) \cdot \max\Big(1 - \frac{w_s}{W_{\max}}, 0\Big), \tag{16}$$

where w_s is work done at s, $W_{\max}$ is a constant parameter measuring the amount of work after which error should be zero, and $e'(n \to s)$ is the adjusted error estimate.

3.4 Adding RAVE statistics

One strength of MCTS bots, especially for games in which stones do not move, such as Go or Hex, is their enhancement of move strength by the Rapid Action Value Estimate, an all-moves-as-first statistic [13]. So we added RAVE to SCNS. With each node we store a map from possible moves (cells) to the RAVE statistic, which consists of two integers: RAVE wins and losses. Statistics are updated whenever a terminal node is created by leaf expansion: for each node on the path from root to the node, we update RAVE values for each move played on the rest of the path.

Assume for the move $n \to s$ we have the RAVE win-loss statistic (w^R, l^R) of the player to move. Denote the number of RAVE games as $g^R = w^R + l^R$. We modify move error:

$$e'(n \to s) = (1 - \alpha) \cdot e(n \to s) + \alpha \cdot R_{\text{impact}} e^R(n \to s), \tag{17}$$

where α indicates how quickly we shift into RAVE error

$$\alpha = \sqrt{\frac{g^R(n \to s)}{3R_{\text{shift}} + g^R(n \to s)}}, \tag{18}$$

$e^R(n \to s)$ is a move error computed by RAVE

$$e^R(n \to s) = \text{erf}^{-1}\left(\frac{l^R(n \to s) - w^R(n \to s)}{g^R(n \to s) + 1}\right), \tag{19}$$

erf^{-1} is inverse error function, and R_{shift} and R_{impact} are constant parameters which indicate how quickly we shift to RAVE error and the impact of RAVE error respectively.

RAVE encourages (discourages) moves that are more often involved in winning (losing) lines and gradually diminishes information from cell energy. SCNS often reaches terminal nodes, so RAVE values accumulate quickly. RAVE can be combined with gradual error forgetting by applying (16) on top of (17).

[9] We measure work done at a node as the number of node expansions in the subtree rooted at that node.

3.5 Transposition table and depth-first implementation

PNS assumes (often incorrectly) that the complete tree can be stored in memory. The DFPNS algorithm overcomes this restriction via a depth-first implementation and transposition table [28]. A DFPNS enhancement — the $1+\varepsilon$ method — reduces the tendency of the search to jump around the tree [32]. The resulting algorithm is stronger than PNS and returns to the root only rarely [28,32].

We apply these three enhancements to CNS. Again, search rarely returns to the root, so updates to the search interval $[v_{\min}, v_{\max}]$ are infrequent. It may even happen that search stays too long in one subtree, in which case we want to force the search back to the root after a few expansions (so, small amount of work) in order to refine the interval. A parameter for this is set according to the time-per-move setting. We tuned it for single setup (See §4.6, parameter MaxWorkPerJob), however it could be increased with higher time constraints.

3.6 Parallel SCNS

Our approach[10] is to mimic the parallelization of DFPN [31]: use a quick thread assignment that follows the natural CNS order, and halt thread execution once its task is redundant. This is achieved by using virtual wins and losses, and temporarily halting thread execution — returning the uncompleted portion of thread's task to the thread pool — once the thread has made MaxWorkPerJob recursive calls. So our parallel SCNS works as follows. See [31] for more details.

1. Replace (dis)proof numbers by (dis)proof functions: each operation — leaf initialization, node update, ... — is now done via (dis)proof functions.
2. Whenever search visits the root, set $v_{\max}$ and $v_{\min}$.
3. Navigate the search tree as in DFPNS, but with (dis)proof numbers $p_n(v_{\max})$ and $d_n(v_{\min})$ until search returns to the root.
4. Give each thread its own search interval, based on virtual (dis)proof functions.

4 Experimental Results

Using parallel SCNS, we implemented the Hex bot DeepHex on the Benzene framework [6]. Benzene includes virtual connection and cell energy computations, so as local SCNS heuristic we used the energy drop at each cell as described in §1.

We used two bots as opponents: Wolve and MoHex, each also implemented on Benzene. Wolve uses $\alpha\beta$ Search with max-width pruning, with circuit resistance for heuristic. MoHex — the strongest Hex bot since 2009 — uses MCTS with RAVE, patterns, prior knowledge estimation, progressive bias, and CLOP tuning of parameters [16]. Wolve and MoHex both compute virtual connections that prune moves and solve positions long before the game ends.

[10] Another approach is to dynamically partition the CNS tree and evaluate subproblems in parallel. Lorenz achieved this for the restriction of CNS to 2 conpirators, i.e. effectively bounding proof function numbers at 2 [23].

We ran all experiments on the 11×11 board. For openings, we used 36 relatively balanced single stone openings: a2 to k2, a10 to k10, b1 to j1, and b11 to j11.

First we optimized parameters using CLOP (§4.1). Then we ran a knockout experiment to show feature importance (§4.2). Next we ran two tournaments: the first one is with several single-threaded bot versions and time limits (§4.3) and in the second one we allowed multi-threading to show how strength increases with number of threads (§4.4). Then we ran a DeepHex vs. MoHex tournament at competition settings (§4.5). Because results of multi-threaded tournaments were disappointed we additionally run parameter optimization using CLOP for multi-threaded version to see if they differ (§4.6). Finally, we give some comments on public games played by DeepHex (§4.7).

4.1 Parameter optimization by CLOP

We optimized parameters using CLOP [11]. In the tuning process we played 30s games, used MoHex as the reference opponent, and set the root-interlude (maximum number of node expansions before search must return to the root) to 20. The final parameter settings are based on 30 000 games. Figure 5 shows that CLOP has already found good settings after 20 000 games.

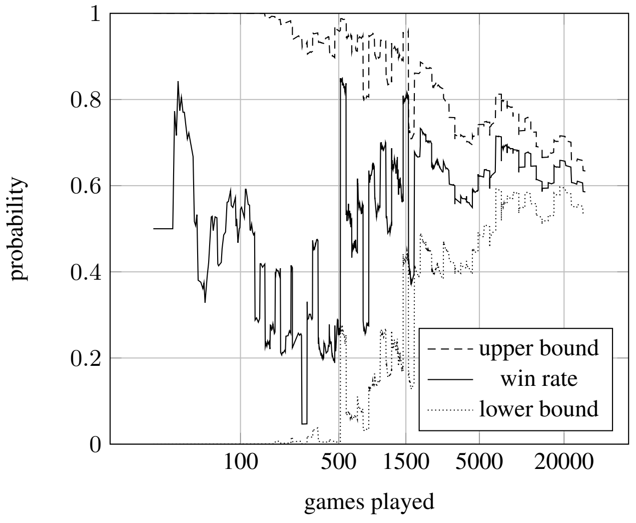

Fig. 5. Win rate with 95% lower and upper confidence bound of the best parameters found by CLOP after playing the given number of games.

DeepHex won .45 of these CLOP-tuning games. The estimated win rate using the best set of parameters is .59. The final settings are shown in Table 1.

The final CLOP-tuned values hint at the effect of the various parameters. δ measures the urgency of sibling expansion: 103 seems small, as moves become easily distinguishable with δ about 300.

parameter	value	description
ε	0.41	ε tolerance
η	0.30	allowed relative error of numbers in proof function
δ	103	the end of the first step in leaf initialization
$P_{\max}$	3	proof threshold when setting $v_{\max}$
$D_{\max}$	4	disproof threshold when setting $v_{\min}$
δ_p	8	extending the search interval on max side
δ_d	7	extending the search interval on min side
R_{shift}	211	RAVE error shift factor
R_{impact}	782	impact of RAVE error
$W_{\max}$	1824	error forgetting threshold

Table 1. Parameters tuned by CLOP for DeepHex.

$P_{\max}$ and $D_{\max}$ are also small, so DeepHex prefers exploring promising lines deeply before diverging. A hand-tuned version of DeepHex with $P_{\max} = D_{\max} = 1$ was strong, so we expected these CLOP-tuned values to be close to 1. The CLOP values 3,4 suggest that for DeepHex the best CNS behaviour is not far from that of PNS. The CLOP values show that optionally extending the search interval by δ_p, δ_d is practically useless, since values 8,7 have negligible effect on performance.

Surprisingly, the RAVE impact is small. We guessed it would be important to incorporate the outcome of terminal nodes quickly, but values 211,782 show this is better done slowly. A similar conlusion holds for gradual error forgetting.

4.2 Knockout experiment

Here we measure feature importance and accuracy of CLOP tuning. We tested many versions of DeepHex, each with either a feature off or a parameter slightly changed. For each version we played 720 matches against MoHex (10 times for each opening) at 30s/move and then — to measure scaling — at 60s/move. See Table 2.

As expected, at 30s/move the CLOP-tuned version is strongest. The most critical feature is better leaf initialization via the multi-step proof function. RAVE is beneficial at 30s/move but less so at 60s/move. This behaviour can be result of too deterministic play of DeepHex with fixed time settings. However, the parameter values are confirmed by another tuning using CLOP in §4.6.

Our goal here was to use CLOP to find — within a relatively short period of time — a reasonable tuning for 30s/move. Given more time, to find a tuning that works well over wide range of time settings, it would have been better to use randomly selected time settings for CLOP instances (as in §4.6). It would also be better to use more than one opponent during CLOP tuning, but we are not aware of any other non-deterministic Hex bots that are comparable in strength to MoHex.

4.3 Strength increase with time constraints

To measure the effect of time constraints on playing strength we ran a round-robin tournament. For DeepHex we used the parameter settings found in §4.1. For each bot,

id	version	30s	60s
(base)	CLOP tuned	60.0	52.6
(a)	2-step (dis)proof function in leafs	20.3	23.1
(b)	no $1+\varepsilon$ method ($\varepsilon = 0$)	57.5	50.6
(c)	exact proof functions ($\eta = 0$)	56.1	48.9
(d)	no gradual error forgetting	56.7	51.9
(e)	no rave	51.9	57.5
(f)	pure SCEF	52.4	55.6
(g)	$P_{\max} = D_{\max} = 1$	52.1	52.5
(h)	$P_{\max} = D_{\max} = 1$ and $\delta_p = \delta_d = 50$	59.4	51.5
(i)	$P_{\max} = D_{\max} = 5$	56.0	53.9

Table 2. Knockout experiment results showing win percentage over MoHex by differ settings in DeepHex. Every winning percentage has $\pm 2.4\%$ confidence bound with 80% confidence. The DeepHex versions are: (base) all features, all parameters with CLOP settings, (a) basic leaf initialization, (b) $1+\varepsilon$ method off, (c) exact proof functions (approximation off), (d) gradual error forgetting off, (e) RAVE off, (f) gradual error forgetting and RAVE both off, (g) smallest possible thresholds inducing smaller search interval, (h) as in (g) but extending search interval to at least 100 on each side, (i) larger thresholds for setting the search interval.

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 [10] in Figure 6.

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 [5].

Figure 7 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.

4.4 Multi-threaded tournament

Here we show how program strength scales with number of threads. 13 bots competed: 1,2,4,8-thread DeepHex; 1,2,4,8-thread MoHex; 1,3,7-thread MoHex plus 1 thread for solver; 1-thread Wolve; 1-thread Wolve plus 1 thread for solver. In each game each bot had 30s/move. Each bot played each other bot two times on each opening, once as black (1st-player) and once as white (2nd-player). So each bot played 864 of the 5616 tournament games.

Figure 8 shows tournament results. Scores are BayesElo [10] with respect to reference player Wolve (win rate .31, score 0); Wolve and Wolve+solver (score 23) are not

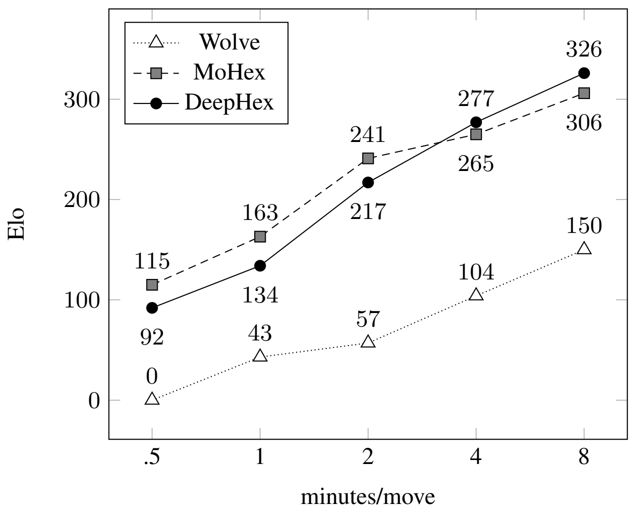

Fig. 6. Tournament results. Each point has error within 14 Elo, with 80% confidence.

shown. MoHex scales well up to the maximum 8 threads; this is perhaps not surprising, as MCTS strength typically increases uniformly with number of simulations and parallelizes relatively easily. MoHex+solver scales well up to 4 threads, but is only slightly stronger at 8 threads. The latter is perhaps because, with solver effectively taking over end games, the difference in opening play between 3-thread MoHex and 7-thread MoHex is not enough to change many outcomes.

DeepHex scales as well as MoHex up to 2 threads, but then more poorly. This drop in scaling efficiency is more pronounced than a similar drop in scaling efficiency of parallel DFPN [31], perhaps due to overfitting (i.e. training only single-threaded, only 30s/move, and only against MoHex), and perhaps because the method we used to parallelize CNS — prevent search tree thread convergence via virtual wins and losses — works better in PNS than in CNS. More research is needed to explore this drop in scaling efficiency.

4.5 DeepHex versus MoHex

Here we simulated a competition tournament on a 12-thread machine. MoHex used its strongest settings: 1 thread for its DFPNS solver and 11 for MCTS. DeepHex does not yet have game-length time control; in almost all games, each bot knows the winner before its 20th move, so we allowed DeepHex (30/20) m/move = 90 s/move.

We played a first tournament using DeepHex settings found by CLOP tuning. However, §4.2 results suggest that — as thinking time increases — $P_{\max}$ and $D_{\max}$ should increase and RAVE weight should decrease. So, we played a second tournament with parameters as in Table 3.

Each tournament had 9 rounds. In each round, each bot played 72 games, i.e. 2 games per opening — once as black (1st-player), once as white (2nd-player) — for a

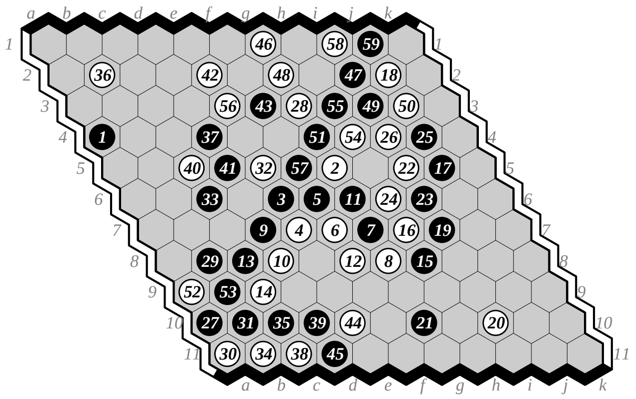

Fig. 7. An 8min/move DeepHex (Black) win over MoHex. With **26**.i4 MoHex has winrate .51 and PV a10 g3 g4 f4 a8 c7 d5 d4 a6 a5 b5 c5 e4 f2 i3 i2 j3 a11 f5 h3 h4 h9 g9 d10 e9 e10. With **27**.a10 DeepHex finds a loss – unproven due to pruning – with PV f4 h4 h5 g4 f5 b8 d5 a6 a5 b5 a11 b10 b11 c10 c11 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.

parameter	value
$P_{\max}$	6
$D_{\max}$	8
δ_p	50
δ_d	50
R_{shift}	500
R_{impact}	500

Table 3. Hand selected parameters for the second tournament.

total of 648 games. DeepHex had a .448 (.457) win rate in the first (second) tournament. So perhaps CLOP tuning is most effective with shorter time limits or as a starting point; for longer time limits or more than one thread hand tuning, especially for parameters such as search interval or RAVE weight, might be more effective.

Under tournament conditions MoHex seems stronger in early play but DeepHex sees further in complicated positions. Figure 9 shows a typical game, where MoHex pushes DeepHex into a losing position before DeepHex escapes.

In the first tournament the average game length for a MoHex (DeepHex) win is 48.6 (61.2) moves, while in the second it is slightly longer 49.6 (62.1). MoHex wins almost all short games, DeepHex wins almost all long ones. See Table 4. MoHex seems strategically stronger, often — perhaps because it is ahead — making simplifying moves. DeepHex seems tactically further-sighted, often — perhaps because it is behind — making complicated moves. A research challenge is to mix these two behaviours.

On an 11×11 board, the shortest possible win would be 21 moves: 11 for the 1st player and 10 for the second. So any game that is less than around 30 moves long prob-

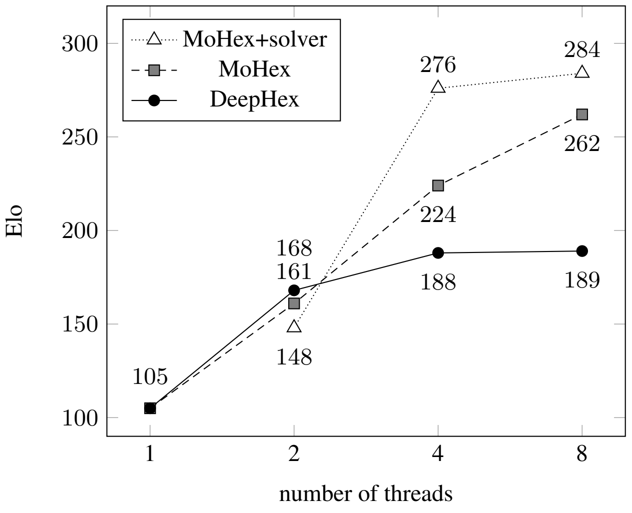

Fig. 8. Tournament results. Each point has error within 14 Elo, with 80% confidence. Reference player is 1-thread Wolve, BayesElo score 0.

length	26-40	41-50	51-60	61-70	71-80	81-95
1st tournament	.04 / 75	.20 / 186	.52 / 195	.73 / 128	.83 / 59	1.00 / 5
2nd tournament	.03 / 62	.21 / 166	.46 / 213	.75 / 145	.89 / 53	.89 / 9

Table 4. DeepHex win rate / the number of games by game length.

ably indicates a weak move the part of the loser. Also, the virtual connection module (VCM) that is common to both MoHex and DeepHex is strong, so weak moves are usually punished when a winning move is found soon after.

For example, consider the 26-move game in Figure 10, the shortest in our experiments. Postgame analysis with an endgame solver shows that in fact move 13.Bj2 is losing. During the experiment, after move 14 the VCM finds quickly that all but sixteen moves are losing, but cannot solve the state in the time allotted; then, after move 15, the VCM finds quickly that 16.Wa4 wins.

Figure 10 also shows the 74-move variation after move 12 that occurs with 60s/move MoHex self-play, where instead of 13.Bj2 MoHex plays 13.Bj4. Here Black loses again, but more slowly, so perhaps Black is already losing after move 11, in which case move 13 is not a blunder. In any event, this game shows that against MoHex a weak move can lead quickly to a loss.

4.6 Multi-threaded parameter optimization with CLOP

The performance of multi-threaded DeepHex using the CLOP optimization settings found with 1-thread tuning yielded disappointing results (§4.1), so we retuned with more threads. This time we ran 4-thread DeepHex against the most powerful 4-thread

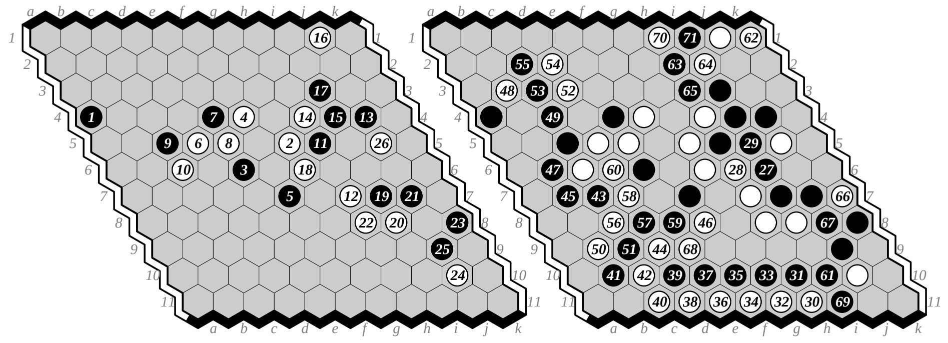

Fig. 9. DeepHex (Black) escapes against MoHex. After **26**.j5 DeepHex sees its loss with PV i6 h6 i5 i10 i9 h10 h9 g10 g9 f10 g8 h2 k1 e9 c9 d8 a8 d7 d9 e8 c7 d6 a7 b5. But MoHex sees neither this nor its win after **28**.h6 and blunders with 30.h11 instead of i10. After 60s of search DeepHex sees a win **31**.h10 with PV g11 g10 f11 f10 e11 e10 d11 d10 c11 b10 c10 c7 d9 b7 f8 b6 d3 d4 d1 b9 f2 f3 g2 g3 c8 b8 h2 b2 b3 a3 b5 c4 b4 c3. With **34**.f11 the MoHex search threads score .62 before the solver thread finds the loss.

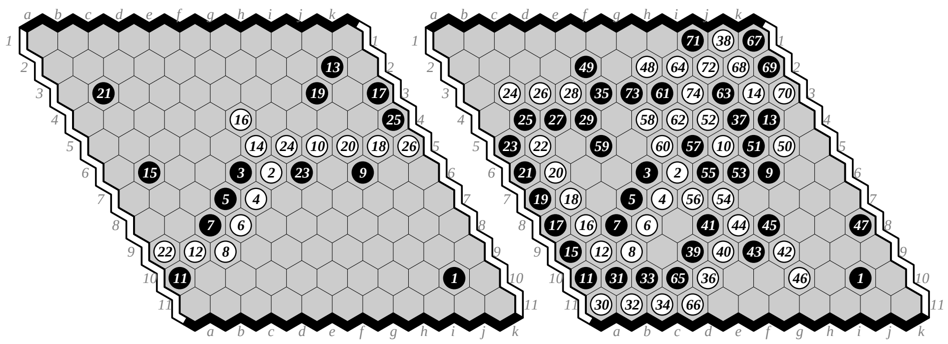

Fig. 10. Left: the shortest game in our experiments. Right: starting after **12**.Wb9, the variation found by 60s/move MoHex self-play.

version of MoHex: 3 MCTS threads and 1 solver thread. In the tuning we also added another parameter to optimize: MaxWorkPerJob. This parameter controls how often a thread breaks the search in order to refine the search interval and find a new subtree to work on. Smaller values of MaxWorkPerJob yield more efficient parallelization: the search is partitioned more evenly but has higher communication costs. For each game, time per move was selected uniformly randomly in the interval $[30, 40]$ seconds. In this way we hoped to reduce overtuning the deterministic DeepHex.

In this experiment we played 35 000 games. See Figure 11. Table 5 shows the final selection of best parameters, yielding estimated win rate .64. Although the learning has not yet converged well after 35 000 games, most parameters have value close to that found when tuning with 1 thread (Table 1). So it is possible that the values are close to optimal; further testing would be needed to confirm this.

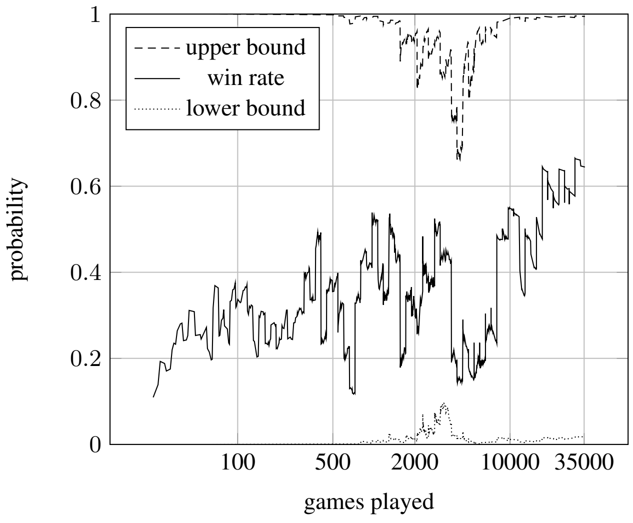

Fig. 11. Win rate with 95% lower and upper confidence bound of the best parameters found by CLOP after playing the given number of games.

Let us compare parameter values here to those of Table 1. Only $W_{\max}$ is higher, which may indicate that we do not need to discard an initial heuristic error from cell energy in order to focus on the most promising moves. A multi-threaded search tends to explore more than a single-threaded one, so the higher value of $W_{\max}$ is expected. The value of MaxWorkPerJob is relatively small, introducing some communication cost between threads, so perhaps the nature of SCNS is such that it is relatively stronger when single-threaded than when multi-threaded. Further research is needed to explore this.

4.7 Public tournaments

DeepHex, MoHex, and a third program Ezo by Kei Takada and Masahito Yamamoto competed in the two computer Hex tournaments — 11×11 and 13×13 — in the 2015 International Computer Game Association's Computer Games Olympiad in Tilburg, Netherlands. In these tournaments MoHex narrowly defeated DeepHex for first place, with a head to head record of 3-1 on 11×11 and 4-2 (the last 2 games were a playoff) on 13×13. This tournament confirmed our opinion that MoHex and DeepHex are evenly matched but with different styles: MoHex seems a bit stronger in opening and early middle play, but its Monte Carlo simulations cannot handle tactical positions. By contrast DeepHex thrives on tactical positions and in complicated positions is particularly strong in the late middle game.

Immediately after the tournament, DeepHex and MoHex each played two 11×11 30-minute exhibition games against Tony Van der Valk, one of the top-ranked Hex players on the online game site Little Golem. DeepHex and MoHex each won both games.

parameter	value	description
ε	0.42	ε tolerance
η	0.21	allowed relative error of numbers in proof function
δ	132	the end of the first step in leaf initialization
MaxWorkPerJob	11	max work for single job for worker
$P_{\max}$	3	proof threshold when setting $v_{\max}$
$D_{\max}$	4	disproof threshold when setting $v_{\min}$
δ_p	8	extending the search interval on max side
δ_d	4	extending the search interval on min side
R_{shift}	297	RAVE error shift factor
R_{impact}	1013	impact of RAVE error
$W_{\max}$	4334	error forgetting threshold

Table 5. Parameters tuned by CLOP for DeepHex running on 4 threads.

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.

We showed the strength of Sibling Conspiracy Number Search features. By far the most critical feature is to initialize leaf (dis)proof functions via a multi-step — rather than 2-step— staircase function. Also, we showed a parallel version of SCNS. Our parallel SCNS Hex bot scales well — as well as MoHex — with 2 threads, but less efficiently with 4 or 8 threads. An open problem is to parallelize SCNS more effectively.

Acknowledgements

This journal paper is a revised version of the conference papers Sibling Conspiracy Number Search [30] and Feature Strength and Parallelization of Sibling Conspiracy Number Search [29]. This research was supported by an NSERC Discovery Grant. We thank the referees for their detailed and helpful comments, including the suggestion of a second tournament with hand-tuned parameters. We are grateful to Kei Takada and Masahito Yamamoto for competing against our programs and to Tony van der Valk for volunteering to play an exhibition against our programs on short notice, and for his helpful postgame analysis.

References

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

2. L. Victor Allis, Maarten van der Meulen, and H. Jaap van den Herik. Proof-number search. *Artificial Intelligence*, 66(1):91–124, 1994.
3. Vadim V. Anshelevich. A hierarchical approach to computer Hex. *Artificial Intelligence*, 134(1–2):101–120, 2002.
4. Broderick Arneson, Ryan B. Hayward, and Philip Henderson. Wolve 2008 wins Hex tournament. *ICGA*, 32(1):49–53, March 2009.
5. Broderick Arneson, Ryan B. Hayward, and Philip Henderson. Monte Carlo Tree Search in Hex. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):251 –258, 2010.
6. Broderick Arneson, Philip Henderson, and Ryan B. Hayward. Benzene, 2009. `http://benzene.sourceforge.net/`.
7. Dennis M. Breuker. *Memory versus Search in Games*. PhD thesis, Maastricht University, Maastricht, Netherlands, 1998.
8. Cameron Browne. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–49, 2012.
9. Rémi Coulom. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In *Computers and Games*, Springer LNCS 4630, pages 72–83, 2007.
10. Rémi Coulom. Bayesian elo rating, 2010. `http://remi.coulom.free.fr/Bayesian-Elo`.
11. Rémi Coulom. CLOP: Confident local optimization for noisy black-box parameter tuning. In *Advances in Computer Games*, Springer LNCS 7168, pages 146–157, 2011.
12. Martin Gardner. *The 2nd Scientific American Book of Mathematical Puzzles and Diversions*, chapter 7, pages 78–88. Simon and Schuster, New York, 1961.
13. Sylvain Gelly and David Silver. Monte-carlo tree search and rapid action value estimation in computer go. *Artif. Intell.*, 175(11):1856–1875, July 2011.
14. Ryan B. Hayward. Mohex wins Hex tournament. *ICGA Journal*, 36(3):180–183, Sept 2013.
15. Philip Henderson. *Playing and solving Hex*. PhD thesis, UAlberta, 2010. `http://webdocs.cs.ualberta.ca/~hayward/theses/ph.pdf`.
16. Shih-Chieh Huang, Broderick Arneson, Ryan B. Hayward, Martin Müller, and Jakub Pawlewicz. Mohex 2.0: A pattern-based mcts hex player. In *Computers and Games*, Springer LNCS 8427, pages 60–71. 2014.
17. Hiroyuki Iida, Makoto Sakuta, and Jeff Rollason. Computer shogi. *Artif. Intell.*, 134(1-2):121–144, 2002.
18. Akihiro Kishimoto, Mark Winands, Martin Müller, and Jahn-Takeshi Saito. Game-tree searching with proof numbers: the first twenty years. *ICGA Journal*, 35(3):131–156, Sept 2012.
19. Norbert Klingbeil and Jonathan Schaeffer. Empirical results with conspiracy numbers. *Computational Intelligence*, 6:1–11, 1990.
20. Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.
21. Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *ECML*, Springer LNCS 4212, pages 282–293, 2006.
22. Richard Korf and David Chickering. Best-first minimax search. *Artif. Intell.*, 84(1-2):299–337, 1996.
23. Ulf Lorenz. Parallel controlled conspiracy number search. In *Euro-Par 2002*, Springer LNCS 2400, pages 420–430. 2002.
24. Ulf Lorenz, Valentin Rottmann, Rainer Feldman, and Peter Mysliwietz. Controlled conspiracy number search. *ICCA Journal*, 18(3):135–147, 1995.
25. David McAllester. A new procedure for growing min-max trees. Technical report, Artificial Intelligence Laboratory, MIT, Cambridge, MA, USA, 1985.

26. David McAllester. Conspiracy numbers for min-max search. *Artif. Intell.*, 35(3):287–310, 1988.
27. David McAllester and Denize Yuret. Alpha-beta conspiracy search. *ICGA*, 25(1):16–35, 2002.
28. A. Nagai. *Df-pn Algorithm for Searching AND/OR Trees and Its Applications*. Ph.d. thesis, Dept. Info. Science, University Tokyo, Tokyo, Japan, 2002.
29. Jakub Pawlewicz and Ryan Hayward. Feature strength and parallelization of sibling conspiracy number search. In Aske Plaat, H.J. van den Herik, and W. Kosters, editors, *Proceedings of the 14th Advances in Computer Games Conference (ACG14)*, Springer LNCS 9525. 2015.
30. Jakub Pawlewicz and Ryan Hayward. Sibling conspiracy number search. In Levi Lelis and Roni Stern, editors, *Proceeding of the 8th International Symposium on Combinatorial Search (SOCS 2015)*, 2015.
31. Jakub Pawlewicz and Ryan B. Hayward. Scalable parallel dfpn search. In *Computer and Games*, Springer LNCS 8427, pages 138–150, 2013.
32. Jakub Pawlewicz and Lukasz Lew. Improving depth-first pn-search: $1+\varepsilon$ trick. In *Computers and Games 2006*, Springer LNCS 4630, pages 160–170. 2007.
33. Jahn-Takeshi Saito, Guillaume Chaslot, JosW.H.M. Uiterwijk, and H.Jaap van den Herik. Monte-carlo proof-number search for computer go. In *Computers and Games*, Springer LNCS 4630, pages 50–61. 2007.
34. Jonathan Schaeffer. Conspiracy numbers. *Artificial Intelligence*, 43(1):67–84, 1990.
35. Maarten van der Meulen. Parallel conspiracy-number search. Master's thesis, Vrije Universiteit Amsterdam, the Netherlands, 1988.
36. Mark Winands. *Informed Search in Complex Games*. PhD thesis, Universiteit Maastricht, Maastricht, Netherlands, 2004.
37. Mark Winands and Maarten Schadd. Evaluation-function based proof-number search. In *Computers and Games 2010*, Springer LNCS 6515, pages 23–35. 2011.