Advances in Computers and Games 2015, Plaat et al. (eds), Springer LNCS (to appear)

Feature Strength and Parallelization of Sibling Conspiracy Number Search

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

Abstract. Recently we introduced Sibling Conspiracy Number Search — an algorithm 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 — and implemented an SCNS Hex bot. Here we show 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.

1 Introduction

Call a heuristic function *local* if it accurately compares the strength of siblings (nodes with the same parent) in the search tree. Recently we introduced Sibling Conspiracy Number Search [19], an algorithm designed for such a heuristic[3], and implemented DeepHex, an SCNS Hex bot.

Our goal there was to introduce a new version of CNS and to implement a competitive Hex bot, so that implementation includes enhancements over basic SCNS. Our implementation, which was single-threaded, was competitive with MoHex.

In this paper we measure the relative contribution of the feature enhancements of our SCNS Hex bot, and describe — and measure the performance of — a parallel implementation.

2 Conspiracy Number Search

In 2-player game search, CNS has shown promise in chess [24,23,13,15,14,17] and shogi [11]. CNS can be viewed as a generalization of PNS.

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) [1,4,18,25,22,26,12]. One can extend

[3] Hex has a good local heuristic. Shannon built an analogue circuit to play the connection game Bridg-it, with moves scored by voltage drop [7]. Adding links between virtual connected cells [2] improves the heuristic, which is reliable among siblings [9].

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.

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 use functions to represent the extended set of values denoted by $\mathbb{V} = \{-\infty\} \cup \mathbb{R} \cup \{+\infty\}$.

Definition 1. 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: If n is a leaf and x is its value (heuristic or actual) then

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

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

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

(Dis)Proof functions can be propagated up from leaves. One way to represent such a function f is as an array of all possible values $f(v)$ for each v. 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.:

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

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 $v = \textsc{Minimax}(n)$.

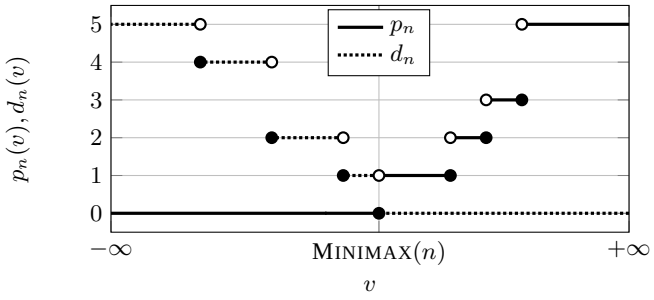

Fig. 1. Each proof function p_n (solid segments) and disproof function d_n (dashed segments) is monotonic staircase. Each black dot belongs to its segment (i.e. closed endpoint), each white dot does not (i.e. open endpoint). The intersection of p_n and d_n is the single point $(\text{MINIMAX}(n), 0)$.

2.1 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 $v_{\mathit{root}} = \text{MINIMAX}(\mathit{root})$. Choose target values $v_{\max}$ for Max (the max player) and $v_{\min}$ for Min so that $v_{\min} < v_{\mathit{root}} < v_{\max}$. We explain how to do this in §2.2. We call $[v_{\min}, v_{\max}]$ *the search value interval* or search interval.[4] 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 SELECTMPN(*root*), with pn $p_n(v_{\max})$ and dn $d_n(v_{\min})$ for every node n. Our CNS implementation — Algorithm 1 — differs from that of McAllester, as we alter both sides of the search interval at once.

Algorithm 1 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 (1)
6:          Update nodes along path to the root using (2)
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) )
```

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

2.2 Choosing the search interval

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

$$\begin{aligned} v_{\max} &= v_{root} + \delta_p, \\ v_{\min} &= v_{root} - \delta_d, \end{aligned} \tag{3}$$

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,

$$\begin{aligned} v_{\max} &= \max_{v \in \mathbb{V}} \{v \,:\, p_{root}(v) \leq P_{\max}\}, \\ v_{\min} &= \min_{v \in \mathbb{V}} \{v \,:\, d_{root}(v) \leq D_{\max}\}, \end{aligned} \tag{4}$$

where $P_{\max}$ and $D_{\max}$ are possibly equal constants. This approach was used in the original CNS algorithm [16,13]. Search proceeds until the interval is sufficiently small, i.e. $v_{\max} - v_{\min} \leq \Delta$, where Δ is a constant indicating an acceptable error tolerance.

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 (3) and (4). Notice that (5) generalizes (4).

$$\begin{aligned} v_{\max} &= \max_{v \in \mathbb{V}} \{v \,:\, p_{root}(v) \leq \max(p_{root}(v_{root} + \delta_p), P_{\max})\}, \\ v_{\min} &= \min_{v \in \mathbb{V}} \{v \,:\, d_{root}(v) \leq \max(d_{root}(v_{root} - \delta_d), D_{\max})\}. \end{aligned} \tag{5}$$

To use CNS as a bot, we search until error tolerance is reached or time runs out and then pick the best move. Experiments show the best criterion for best move is the branch on which most time (leaf expansions in the subtree) is spent.

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 by adding relative errors, as follows. The evaluation of non-terminal game tree node n is given by

$$\text{EVAL}(n) = \sum_{i=1}^{k} \sigma(p_{i-1}) \cdot e(p_{i-1} \to p_i), \tag{6}$$

where $p_0 \to p_1 \to \cdots \to p_k = n$ is the path from root p_0 to n, $\sigma(p_j) = 1(-1)$ if we (the opponent) are to move at p_j, and, for any child s of n, $e(n \to s) = \log \frac{E(n \to s_0)}{E(n \to s)}$ is the relative error at n with respect to s, where s_0 is a child of n with best score. We call this *siblings comparison evaluation function* (scef).

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.

Although SCNS — CNS with scef — explores good lines of play, the version we have described so far is wasteful, as CNS tends to expand all siblings whenever a new child is expanded. To avoid this, especially for unpromising children, 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 via a two-step staircase function (1), we use a multi-step staircase function, with the number of steps logarithmic in the difference between current and minimax values. Hence

$$p_n(v) = \begin{cases} 0 & \text{if } v \leq x \\ i & \text{if } i^\delta < 2^{(v-x)} \leq (i+1)^\delta \end{cases} \quad d_n(v) = \begin{cases} 0 & \text{if } v \geq x \\ i & \text{if } i^\delta < 2^{(x-v)} \leq (i+1)^\delta \end{cases} \tag{7}$$

$x = \textsc{Minimax}(n)$, i is a positive integer and δ a positive rational. Using (7) to initialize non-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.

3.1 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[5] at such a move the initial error estimate should be corrected. We gradually decrease error as follows,

$$e'(n \to s) = e(n \to s) \cdot \max\Big(1 - \frac{w_s}{W_{\max}}, 0\Big), \tag{8}$$

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.

3.2 Adding RAVE statistics

One strength of MCTS Hex bots is their enhancement of move strength by the Rapid Action Value Estimate, an all-moves-as-first statistic [8]. 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:

$$e'(n \to s) = (1 - \alpha) \cdot e(n \to s) + \alpha \cdot R_{\text{impact}} e^R(n \to s), \tag{9}$$

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

where α indicates how quickly we shift into RAVE error

$$\alpha = \sqrt{\frac{g^R(n \to s)}{3R_{\text{shift}} + g^R(n \to s)}}, \tag{10}$$

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

$$e^R(n \to s) = \text{erf}^{-1}\left(\frac{l^R(n \to s) - w^R(n \to s)}{g^R(n \to s) + 1}\right), \tag{11}$$

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 (8) on top of (9).

3.3 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 [18]. A DFPNS enhancement — the $1+\varepsilon$ method — reduces the tendency of the search to jump around the tree [21]. The resulting algorithm is stronger than PNS and returns to the root only rarely [18,21].

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.

3.4 Parallel SCNS

Our approach[6] is to mimic the parallelization of DFPN [20]: 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 [20] 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.

[6] 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 [14].

4 Experimental Results

Using parallel SCNS, we implemented the Hex bot DeepHex on the Benzene framework [3]. 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 [10]. Wolve and MoHex both compute virtual connections that prune moves and solve positions long before the game ends.

For openings, we used 36 relatively balanced single stone openings: a2 to k2, a10 to k10, b1 to j1, and b11 to j11.

We optimized parameters using CLOP (§4.1). Then we ran a knockout experiment to show feature importance (§4.2). Next we ran a tournament to show how strength increases with number of threads (§4.3). Finally, we ran a DeepHex vs. MoHex tournament at competition settings (§4.4).

4.1 Parameter optimization by CLOP

We optimized parameters using CLOP [6]. 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; CLOP already found good settings after $20\,000$ games. DeepHex won 45% of these CLOP-tuning games. Final settings are shown in Table 1.

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.

The CLOP-tuned values hint at the effect of various parameters. δ measures the urgency of sibling expansion; 103 seems small, as moves become easily distinguishable with δ about 300. $P_{\max}$ 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 that the CLOP-tuned values to be close to 1; CLOP values 3,4 suggest that for DeepHex the best CNS behaviour is not far from that of PNS. CLOP values show optional extension of the search interval by δ_p, δ_d is practically useless, as values 8,7 have negligible effect on performance. Surprisingly, 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.

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. 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.

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.

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. 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 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.

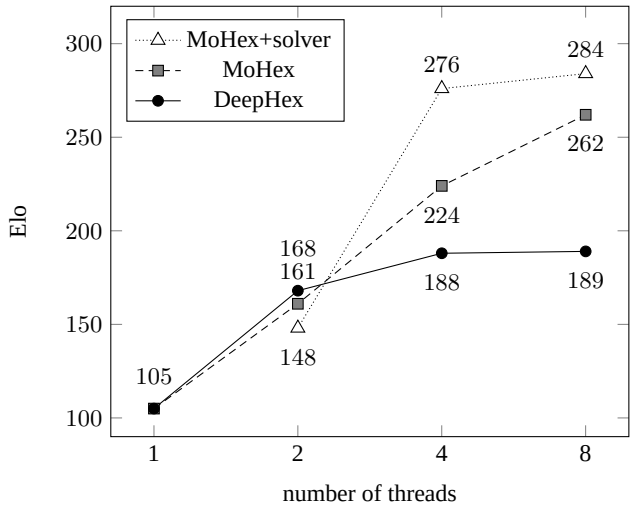

Fig. 2. Tournament results. Each point has error within 14 Elo, with 80% confidence. Reference player is 1-thread Wolve, BayesElo score 0.

Figure 2 shows tournament results. Scores are BayesElo [5] with respect to reference player Wolve (win rate .31, score 0); Wolve and Wolve+solver (score 23) are not 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 [20], perhaps due to overfitting (i.e. with training settings only single-threaded, only 30s/move and training opponent only MoHex), and perhaps because the parallelization method — prevent search tree thread convergence via virtual wins and losses — works better in PNS than in CNS. More work is needed to explore this behaviour.

4.4 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.

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.

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 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 4.4 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.

length	26-40	41-50	51-60	61-70	71-80	81-95
1st tournament win rate	.04	.20	.52	.73	.83	1.00
2nd tournament win rate	.03	.21	.46	.75	.89	.89

Table 4. DeepHex win rate by game length.

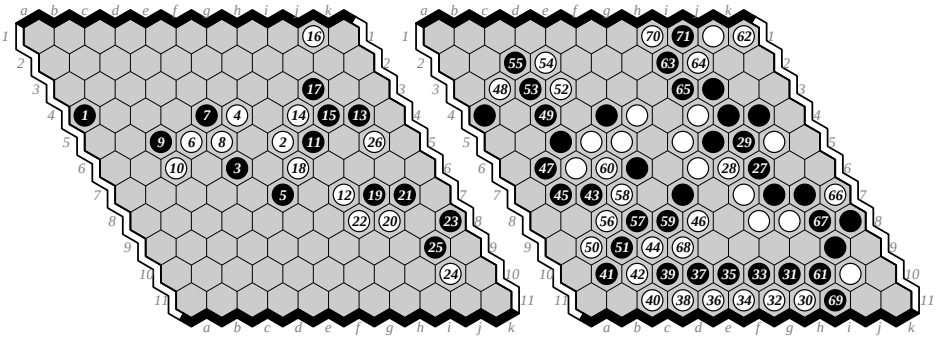

Fig. 3. DeepHex (Black) escapes against MoHex. After 26.j5 DeepHex sees its loss with PV i6 h6 i5 i10 i9 h10 h9 g10 g9 f10 g8 h2 k1 e9 c9 d8 a8 d7 d9 e8 c7 d6 a7 b5. But MoHex sees neither this nor its win after 28.h6 and blunders with 30.h11 instead of i10. After 60s of search DeepHex sees a win 31.h10 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.

5 Conclusions and further research

We showed the strength of Sibling Conspiracy Number Search features by competing our SCNS Hex bot with MCTS bot MoHex and $\alpha\beta$ bot Wolve. 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

We thank the referees for their helpful comments, including the suggestion of a second tournament with hand-tuned parameters.

References

1. L. Victor Allis. *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis, University of Limburg, Maastricht, Netherlands, 1994.
2. Vadim V. Anshelevich. A hierarchical approach to computer Hex. *Artificial Intelligence*, 134(1–2):101–120, 2002.
3. Broderick Arneson, Philip Henderson, and Ryan B. Hayward. Benzene, 2009. `http://benzene.sourceforge.net/`.
4. Dennis M. Breuker. *Memory versus Search in Games*. PhD thesis, Maastricht University, Maastricht, Netherlands, 1998.
5. Rémi Coulom. Bayesian elo rating, 2010. `http://remi.coulom.free.fr/Bayesian-Elo`.
6. Rémi Coulom. CLOP: Confident local optimization for noisy black-box parameter tuning. In *Advances in Computer Games*, Springer LNCS 7168, pages 146–157, 2011.

7. Martin Gardner. *The 2nd Scientific American Book of Mathematical Puzzles and Diversions*, chapter 7, pages 78–88. Simon and Schuster, New York, 1961.
8. Sylvain Gelly and David Silver. Combining online and offline knowledge in UCT. In *24th ACM ICML*, pages 273–280, 2007.
9. Philip Henderson. *Playing and solving Hex*. PhD thesis, UAlberta, 2010. `http://webdocs.cs.ualberta.ca/~hayward/theses/ph.pdf`.
10. 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.
11. Hiroyuki Iida, Makoto Sakuta, and Jeff Rollason. Computer shogi. *Artif. Intell.*, 134(1-2):121–144, 2002.
12. 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.
13. Norbert Klingbeil and Jonathan Schaeffer. Empirical results with conspiracy numbers. *Computational Intelligence*, 6:1–11, 1990.
14. Ulf Lorenz. Parallel controlled conspiracy number search. In *Euro-Par 2002*, Springer LNCS 2400, pages 420–430. 2002.
15. Ulf Lorenz, Valentin Rottmann, Rainer Feldman, and Peter Mysliwietz. Controlled conspiracy number search. *ICCA Journal*, 18(3):135–147, 1995.
16. David McAllester. Conspiracy numbers for min-max search. *Artif. Intell.*, 35(3):287–310, 1988.
17. David McAllester and Denize Yuret. Alpha-beta conspiracy search. *ICGA*, 25(1):16–35, 2002.
18. A. Nagai. *Df-pn Algorithm for Searching AND/OR Trees and Its Applications*. Ph.d. thesis, Dept. Info. Science, University Tokyo, Tokyo, Japan, 2002.
19. Jakub Pawlewicz and Ryan Hayward. Sibling conspiracy number search. In *SoCS 2015: The 8th Annual Symposium on Combinatorial Search*, 2015.
20. Jakub Pawlewicz and Ryan B. Hayward. Scalable parallel dfpn search. In *Computer and Games*, Springer LNCS 8427, pages 138–150, 2013.
21. 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.
22. 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.
23. Jonathan Schaeffer. Conspiracy numbers. *Artificial Intelligence*, 43(1):67–84, 1990.
24. Maaretn van der Meulen. Parallel conspiracy-number search. Master's thesis, Vrije Universiteit Amsterdam, the Netherlands, 1988.
25. Mark Winands. *Informed Search in Complex Games*. PhD thesis, Universiteit Maastricht, Maastricht, Netherlands, 2004.
26. Mark Winands and Maarten Schadd. Evaluation-function based proof-number search. In *Computers and Games 2010*, Springer LNCS 6515, pages 23–35. 2011.