Stronger Virtual Connections in Hex

Jakub Pawlewicz and Ryan Hayward and Philip Henderson and Broderick Arneson

***Abstract*—For connection games such as Hex or Y or Havannah, finding guaranteed cell-to-cell connection strategies can be a computational bottleneck. In automated players and solvers, sets of such virtual connections are often found with Anshelevich's H-search algorithm: initialize trivial connections, and then repeatedly apply an AND-rule (for combining connections in series) and an OR-rule (for combining connections in parallel).**

We present FastVC Search, a new algorithm for finding such connections. FastVC Search is more effective than H-search when finding a representative set of connections quickly is more important than finding a larger set of connections slowly.

We tested FastVC Search in an alpha-beta player Wolve, a Monte Carlo tree search player MoHex, and a proof number search implementation called Solver. It does not strengthen Wolve, but it significantly strengthens MoHex and Solver.

***Index Terms*—Hex, connection games, virtual connection, H-Search**

I. Introduction

HEX is a two-player perfect information connection game invented independently by Piet Hein in 1942 [1] and John Nash in 1948 [2]–[4]. Hex has been an active domain of artificial intelligence research since Claude Shannon's seminal work in the 1950s [5]. It is likely to remain an active domain, as the game is easy to implement yet challenging to master, and solving arbitrary Hex positions is PSPACE-complete [6].

Automated Hex players rely on the computation of connection strategies [7]. This computation is costly and so reduces the number of positions that can be explored in a tree search, whether alpha-beta, monte-carlo, or proof number search, but usually pays off by finding wins early. For 11×11 games, Anshelevich reports that H-search connection computations routinely find a win 20 or more ply before the end of the game and yield significant move pruning [8]. These gains in lookahead and pruning are often worth the computational cost.

But the number of connection strategies of a Hex position can grow exponentially with the number of cells. Even on moderately-sized boards such as 9×9, finding all of a position's connection strategies is computationally infeasible. Finding a critical subset of these connections, and doing so more efficiently than has been done before, would significantly increase the strength of current connection-based players and solvers [9].

In this paper, we give a new method for computing connections. Rather than finding many connections, we find a representative subset of critical connections. And rather than the usual search, we use a more efficient search. We test our method in state-of-the-art Hex players. The strength gains are significant.

In §II we review the rules of Hex and the algebra of computing connection strategies. In §III we review previous implementations, and improvements, for this algebra. In §IV we present our modification of H-search. In §V we describe semis-combiner, a new way of performing the OR-rule that bypasses a computational bottleneck. This new algorithm leads to an explosive growth in the number of connections that can be found in a short time. In §VI we tackle the problem of dealing with such large sets of connections. In §VII we give experimental results and in §VIII we conclude.

II. Connection strategy algebra

A. Rules of Hex

Hex is played on an $n \times n$ board with hexagonal cells. Two players, Black and White, alternate turns coloring any uncolored cell with their color. The winner is the player who forms a path of their color joining their opposing two sides. See Figure 1.

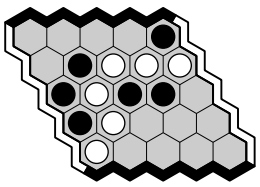


Fig. 1. White has a path of cells joining the two White sides, so White wins.

Hex cannot end in a draw, and for $n \times n$ boards there exists a winning strategy for the first player. This first-player advantage is noticeable in practice, especially on relatively small boards. To mitigate it, the *swap rule* is often adopted: the first player colors a cell black; then the second player chooses whether to be Black; then White colors a cell, and play continues in alternating fashion. When played with the swap rule, the second player has a winning strategy, but to play perfectly must know the win/loss value of every opening move. To date, automated solvers have found all such values for all board sizes up to 9×9. Computer tournaments often use 11×11 boards; 13×13 and 19×19 boards are also popular. On all these boards the average branching factor in a typical game is more than 100.

B. Connection strategies and their application

In a Hex position, a *chain* is a maximal connected group of same-colored cells. A P-chain is a chain with P's color. The position in Figure 1 has 3 black chains and 1 white chain. For a Hex position and a player P, a *connection strategy* specifies the two *endpoints* being connected and the *carrier*, namely the

J. Pawlewicz is with the Institute of Informatics, University of Warsaw. E-mail: pan@mimuw.edu.pl

R. Hayward and B. Arneson are with the Department of Computer Science, University of Alberta. E-mail: hayward@ualberta.ca, broderic@cs.ualberta.ca

Manuscript prepared February, 2014.

set of empty cells P requires to carry out the alternating-turn strategy. An endpoint is either an empty cell or a P-chain or a P-colored side. Following Anshelevich [10], a *virtual connection (VC)* is a second-player connection strategy, and a *virtual semi-connection (VSC)* is a first-player connection strategy. The first move of a VSC — which yields a VC with smaller carrier between the same endpoints — is its *key*.

A VC (VSC) between x and y is an x-y *VC* (*VSC*). An x, y V(S)C carrier is *minimal* if it is not a strict superset of some other V(S)C carrier.

Obviously, recognizing VCs and VSCs is useful. If a player has a *side-to-side* VC — one whose endpoints are the player's two sides — then the player has a winning strategy. We call such a VC a *winning VC*: the player can win even if it is the opponent's turn to play. Similarly, if the player-to-move has a *side-to-side VSC*, then the player-to-move has a winning strategy. We call such a VSC a *winning VSC*.

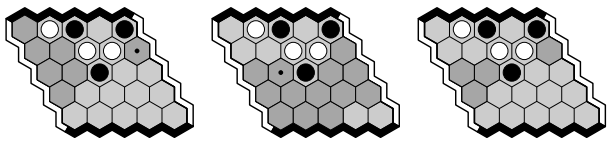


Fig. 2. From left: The first two figures each show the carrier and key of a winning White VSC. So, if Black moves next, Black must play in a cell that intersects both carriers, as shown in the third figure.

If the player-to-move P finds a winning opponent-VSC, P must move within its carrier to avoid reaching a losing position (assuming that the opponent will also find this VSC). Similarly, if P finds several winning opponent-VSCs, P must move in the intersection of the associated carriers to avoid reaching a losing position. Hayward et al. call this intersection the *mustplay*, and use it to prune the list of possible moves (and so reduce the branching factor) when playing and solving Hex positions [11]. See Figure 2.

As in other cell-coloring games such as Go or Havannah, Hex positions can sometimes be decomposed into independent subgames. A four-sided subgame is essentially a smaller subgame on a possibly irregularly-shaped board. For such a subgame, finding a VC for the winner P (who can connect their two opposing P-sides) allows one to *fill* the subgame's empty cells (i.e. P-color them) without changing the position's value, thus pruning all moves in the subgame from future consideration [12], [13]. See Figure 3.

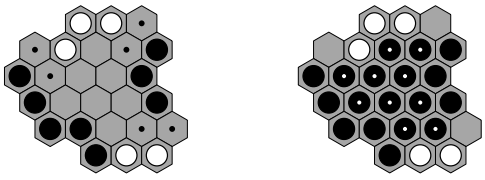


Fig. 3. Left: a four-sided subgame of a larger position. The boundary of the subgame is defined by the two White chains, the two Black chains, and the three dotted cell pairs (each such pair forms a bridge connection between opposite-colored chains). Black has a VC whose endpoints are the two Black bounding chains and whose carrier lies within the subgame interior, so — as shown at right — one can Black-fill the subgame interior without changing the position's value.

C. Connection strategy algebra

Anshelevich gave a hierarchical algebra that computes some — but not all — VCs and VSCs [10]. A *connection strategy* is a triple $S = (x, C, y)$ where x, y are the endpoints, C is the carrier, and $x, y \notin C$, i.e. neither endpoint is in the carrier. If S is a VSC for player P with key k, then $(x, C \setminus \{k\}, y)$ is a VC in the position obtained by P-coloring cell k.

A *base VC* is a connection strategy $(x, \emptyset, y)$, namely with x, y adjacent. For example, for player P, adjacent empty cells form a base VC, as does an empty cell adjacent to a P-chain or P-side.

Starting with base connection strategies, one can iteratively construct more construction strategies using the AND- and OR-rules. The former combines connections in serial; the latter combines them in parallel.

1) *AND-rule*. If $S_1 = (x, C_1, u)$ and $S_2 = (u, C_2, y)$ are P-VCs, and $C_1 \cap C_2 = \emptyset$, $x \notin C_2$, $y \notin C_1$, then
   a) if u is P-colored, $(x, C_1 \cup C_2, y)$ is P-VC,
   b) if u is uncolored, $(x, C_1 \cup \{u\} \cup C_2, y)$ is a P-VSC.
2) *OR-rule*. If $S_z = (x, C_z, y)$ are P-VSCs, and $\bigcap C_z = \emptyset$, then $(x, \bigcup C_z, y)$ is a P-VC.

Each iteration of this construction algorithm applies the AND-/OR-rules to all possible known connection strategies, using the current strategies to produce a new generation of strategies. Iteration continues until no new strategies are produced. This hierarchical (by generation) AND-/OR- closure algorithm is called H-search [8].

For a position and player, the set of V(S)Cs found by an iteration-limited H-search depends on the order in which the AND- and OR-rules are applied to particular endpoint pairs. But if H-search is computed to completion — i.e. until no new V(S)C can be created — then the set of minimal V(S)C carriers that are found is fixed. For this position and player, we call these the *minimal VC and VSC carrier sets*.

III. Previous approaches

There are many ways to implement H-search. Here are some algorithmic tips from various approaches.

- Discard any V(S)C β that is a superset[1] of another V(S)C α with the same endpoints. Both deduction rules require the carriers being combined to have empty intersection, so the set of strategies generated from a set S containing α is equal to the set generated from $S \cup \{\beta\}$ [14], [15].
- Before applying the OR-rule to all current x-y VSCs, confirm that the intersection of all such carriers is empty. If it is not empty then the intersection of every non-empty subset of carriers is not empty, so no new x-y VCs can be created [15].
- If applying the OR-rule recursively, then backtrack whenever the most recent x-y VSC does not reduce the intersection of all such VSCs. This VSC cannot help construct new VCs, and will only increase the carrier size of any new VC [16].
- Limit OR-rule application by considering VSC subsets of size at most 3 or 4. Checking all 2^k subsets of a set of

[1]We often identify a V(S)C by its carrier.

k VSCs usually takes too long, and yields diminishing strength returns for $k \geq 5$ [15].

- When applying the AND-rule, allow board sides to be midpoints. This might seem counter-intuitive, since sides are final destinations, but it allows the discovery of connection strategies not otherwise deducible by H-search [14], [15].
- Due to diminishing performance returns, limit either the number of V(S)Cs stored for each endpoint pair x, y (hard limit), or the number of such V(S)Cs considered in constructing larger V(S)Cs (soft limit). If such limits are used, sort V(S)Cs by carrier size, so that smaller (more useful) connection strategies will be used in construction [14], [15].

Several variations and/or enhancements of H-search have been proposed, including generalized H-search, the crossing rule, captured set intersection, and common miai carrier intersection [12], [17]–[19].

IV. Computing VCs

A. Our new method

To find set of VCs that strengthened our player and solver, we changed many aspects of the general and/or-rule closure algorithm. One change is to store connections more efficiently: see §IV-B. This allows many search optimizations: see §IV-C. Another change is to apply the OR-rules so that all VCs between two fixed endpoints are created by an operation called *semis-combiner*. Rather than considering separately all VSC subsets that might give rise to new VCs, semis-combiner acts on all input VSCs at once: see §V.

B. Storage

Before outlining our new algorithm, we describe our data structures. To store VCs and VSCs, we use maps M_c and M_s indexed by endpoint pairs. $M_\mathrm{c}(x, y)$ (respectively $M_\mathrm{s}(x, y)$) stores x-y VCs (VSCs). For each pair x, y, we store only the minimal carriers C for which (x, C, y) is a VC (VSC). This set of carriers is stored in a vector. Previous implementations used a linked list, which allowed lists to be easily maintained in sorted order by carrier size. We prefer to use a vector, which allows for faster updates even if elements are occasionally inserted at arbitrary locations. Rather than maintain these lists in guaranteed sorted order, we use a move-to-front technique which is just as effective but does not guarantee the lists are sorted: see §IV-C7. Each carrier has a flag indicating whether it has been processed. In our pseudocode, for $t = \mathrm{c}, \mathrm{s}$ (for VCs,VSCs) we denote by $M_t^p(x, y)$ (resp. $M_t^u(x, y)$) a subset of $M_t(x, y)$ with only processed (unprocessed) VCs/VSCs. A VC is *processed* once it has been used in the AND-rule with all other processed VCs to create new VCs and VSCs. A VSC is *processed* once it has been used by semis-combiner to create new VCs. See §IV-C.

Our approach is minimalist with respect to VC storage. We do not store a VSC's key: if needed, it is recomputed on demand by AND-rule calls. We prefer this approach, as for our purposes it is faster to operate on vectors rather than lists.

For efficiency reasons we need some extra structures. For each endpoint x we maintain the set of all endpoints y with which x is connected via a VC. We denote this *neighborhood* by $N(x)$. For each endpoint pair (x, y) we maintain the intersection of processed VCs and all VSCs, denoted by $I_\mathrm{c}^p(x, y)$ and $I_\mathrm{s}(x, y)$ respectively. As we will see, we will update $I_\mathrm{c}^p(x, y)$ when a VC becomes processed and $I_\mathrm{s}(x, y)$ when a new VSC is created.

We direct our algorithm using queues Q_c and Q_s. Q_c maintains triples (x, C, y) of all unprocessed VCs that are AND-rule candidates. Q_s keeps candidates for semis-combiner, but stores only those endpoint pairs (x, y) for which there is some unprocessed VSC in $M_\mathrm{s}(x, y)$ and $I_\mathrm{s}(x, y) = \emptyset$. Thus there is a chance that semis-combiner applied to all x-y VSCs produces new VCs. Q_s stores only endpoint pairs because semis-combiner will process all such current VSCs at once. See §V.

TABLE I
Summary of data structures

symbol	meaning
$M_t(x, y)$ $M_t^p(x, y)$ $M_t^u(x, y)$	set of carriers C such that (x, C, y) is a VC ($t = \mathrm{c}$) or VSC ($t = \mathrm{s}$); superscript p (u) denotes subset of processed (unprocessed) VCs/VSCs
$N(x)$	VC–neighborhood of x: $\{y : M_\mathrm{c}(x, y) \neq \emptyset\}$
$I_\mathrm{c}^p(x, y)$	intersection of processed x-y VCs: $\bigcap M_\mathrm{c}^p(x, y)$
$I_\mathrm{s}(x, y)$	intersection of x-y VSCs: $\bigcap M_\mathrm{s}(x, y)$
Q_c	queue of unprocessed VCs
Q_s	queue of endpoint pairs (x, y) with an unprocessed x-y VSC, and with $I_\mathrm{s}(x, y) = \emptyset$

C. Search

We now describe the pseudocode of Algorithm 1, our new algorithm.

1) Function VCSearch*:* Create all base VCs, i.e. those between adjacent cells. Mark these as unprocessed and push onto Q_c.

Loop until both queues are empty (line 3). The loop invariant is that processed VCs/VSCs have been used in all possible AND-rule/OR-rules with all other processed VCs/VSCs, and unprocessed VCs/VSCs have never been used in either rule.

At each iteration, try the AND-rule, and — via semis-combiner — try the OR-rule only if the AND-rule fails. We postpone applying the OR-rule as long as possible, because semis-combiner uses all VSCs between a given endpoint pair and finds all VCs at once. See §V.

If some VC remains unprocessed (so NonEmpty(Q_c) is true), pop a VC (x, C, y) from Q_c (line 5) and in the rest of this iteration try the AND-rule only on this VC together with all processed VCs. Try both ends as the AND-rule midpoint (line 6). After the two DoAnd calls finish, mark the VC as processed (line 7) and update $I_\mathrm{c}^p(x, y)$ (line 8).

After all VCs are processed — the AND-rule has been tried on all pairs of current VCs — try the OR-rule. Pop the endpoint pair (x, y) from Q_s (line 10) and call DoOr, which applies semis-combiner (line 11) to the sets of all VSCs. Mark these VSCs as processed (line 12).

Algorithm 1 FastVC Search

```
1: function VCSEARCH
2:     initialize structures with base VCs
3:     while NONEMPTY(Q_c) or NONEMPTY(Q_s) do
4:         if NONEMPTY(Q_c) then
5:             (x, C, y) ← POP(Q_c)
6:             DOAND(x, C, y), DOAND(y, C, x)
7:             mark VC (x, C, y) processed
8:             I_c^p(x, y) ← I_c^p(x, y) ∩ C
9:         else
10:            (x, y) ← POP(Q_s)
11:            DOOR(x, y)
12:            mark all x-y VSCs as processed
13: function DOAND(x, C_1, u)
14:     for all y ∈ N(u) − (C_1 ∪ {x}) do
15:         if (C_1 ∪ {x}) ∩ I_c^p(u, y) = ∅ then
16:             for all C_2 ∈ M_c^p(u, y) do
17:                 ANDRULE(x, C_1, u, C_2, y)
18: function ANDRULE(x, C_1, u, C_2, y)
19:     if (C_1 ∪ {x}) ∩ C_2 ≠ ∅ then return
20:     if u is colored then
21:         TRYADDVC(x, C_1 ∪ C_2, y)
22:     else
23:         TRYADDVSC(M_s(x, y), C_1 ∪ {u} ∪ C_2)
24: function DOOR(x, y)
25:     for all C ∈ SEMISCOMBINER(M_s(x, y), M_c(x, y)) do
26:         TRYADDVC(x, C, y)
27: function TRYADDVC(x, C, y)
28:     if TRYADD(M_c(x, y), C) then PUSH(Q_c, (x, C, y))
29: function TRYADDVSC(x, C, y)
30:     if TRYADD(M_s(x, y), C) then
31:         I_s(x, y) ← I_s(x, y) ∩ C
32:         if I_s(x, y) = ∅ and (x, y) ∉ Q_s then
33:             PUSH(Q_s, (x, y))
34: function TRYADD(C, C_new)
35:     for subsequent C ∈ C do
36:         if C ⊆ C_new then
37:             move C to front of C
38:             return false
39:     C ← {C ∈ C : C_new ⊈ C} ∪ {C_new}
40:     return true
```

2) Function DOAND(x, C_1, u)*:* For the AND-rule, given a VC (x, C_1, u), find a companion VC with endpoints u, y. Iterate over all feasible y (line 14). Iterate over the VC-neighborhood $N(u)$, rather than all cells on the board; this saves time.

Fix y and iterate over all u, y VCs, applying the AND-rule by calling ANDRULE. First check whether the intersection $I_c^p(u, y)$ of these VCs is small enough (line 15) to allow a suitable VC. This check often avoids a futile loop.

3) Function ANDRULE(x, C_1, u, C_2, y)*:* Check the remaining AND-rule condition and try to add new a VC or VSC. Depending on whether u is colored, call TRYADDVC (yes) or TRYADDVSC (no).

4) Function DOOR(x, y)*:* SEMISCOMBINER returns all newly created VCs. These might not be minimal, so consider them one by one with TRYADDVC.

5) Function TRYADDVC(x, C, y)*:* If C is a new minimal x-y VC carrier, as checked by TRYADD, push it onto Q_c. Mark newly created VC as unprocessed.

6) Function TRYADDVSC(x, C, y)*:* Check whether C is minimal by calling TRYADD. If yes, update the intersection of all x-y VSCs; if this set is empty then push $x - y$ onto Q_s. Mark newly created VCS as unprocessed.

7) Function TRYADD$(\mathcal{C}, C_{new})$*:* This is the function in which the most time is spent. Check whether a newly created carrier C_{new} is minimal, i.e. is not the superset of an existing carrier. This must be done efficiently, so we use a move-to-front technique.

Iterate over all carriers in the vector $\mathcal{C}$. The first carrier C found that is a subset of C_{new} (line 36) is a *rejecting* carrier. It rejects the new connection, and might reject future connections, so move it to the front of $\mathcal{C}$ (line 37). This takes amortized constant time, since we iterated over all carriers preceding C. Using move-to-front, the best rejecting carriers quickly collect at the front of $\mathcal{C}$, significantly reducing the rejection time of non-minimal carriers. This method is crucial, since new carriers are usually non-minimal, with new minimal VCs/VSCs discovered rarely.

A new connection that is not rejected is added as a new carrier. Also, all carriers which become non-minimal are removed (line 39). Again, this is amortized-time efficient, as we already iterated over all carriers. However, we need to update VC information carefully: if filtering removes a processed VC, recalculate $I_c^p(x, y)$; if it removes an unprocessed VC, remove the VC from Q_c.

V. SEMIS-COMBINER

A. The previous approach

Consider x-y connections. Let $\mathcal{C}$ (resp. $\mathcal{S}$) be the set of all x-y VC (VSC) carriers. A straightforward application of the OR-rule is to iterate over all subsets $\mathcal{S}'$ of $\mathcal{S}$. If the intersection of the subsets of $\mathcal{S}'$ is empty then combining these strategies yields a new VC C' whose carrier is the union of these subsets. One must check whether C' *covers* (is a superset of) the carrier of any current VC; if not, then C' is minimal and so can be added to $\mathcal{C}$.

There are $2^{\#\mathcal{S}}$ such subsets, so iterating over all of them is in general infeasible. A usual previous approach is to iterate only over subsets $\mathcal{S}' \subseteq \mathcal{S}$ of bounded size, say 3 or 4. But this reduces the set of new connections that can be found, which in turn reduces the strength of the program that uses the VC engine, while still leaving iteration over VSCs as a computational bottleneck.

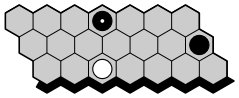
B. Our new approach

We now describe our semis-combiner algorithm. The main idea of our approach is to focus on what we call blocked cells, as we shall explain.

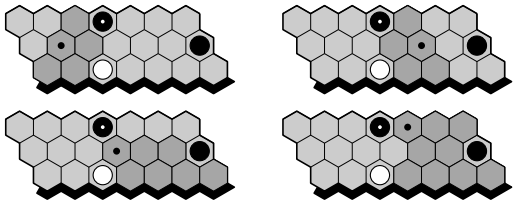
To start, suppose $\mathcal{C}$ is empty (so there is no x-y VC). Then we create a new VC C' whose carrier is the union of all

strategies of $\mathcal{S}$ only if the intersection of all corresponding carriers is empty.

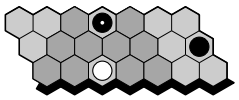
Next, suppose $\mathcal{C}$ is non-empty. Assume $\mathcal{C}$ contains exactly one carrier, say C. The carrier of a new VC C' cannot cover C, so there must be some cell a that is in C but not in C', and C' can be created only from VSCs not containing a. We call a a *blocked cell* for constructing C'. Let $\mathcal{S}' = \{S \in \mathcal{S} \mid a \notin S\}$ be the set of all such VSCs not containing some blocked cell. Create a new VC only if the combined intersection of $\mathcal{S}'$ is empty. So if $\mathcal{C}$ contains one carrier C, it suffices to iterate over all a in C.



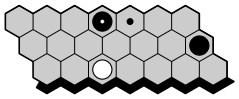
Consider an example. Above, between the black side and the dotted stone, we create VCs from four VSCs A, B, C, D (left to right, top to bottom):



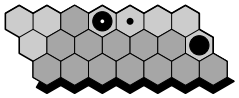
The intersection of A, B, C, D is empty, as is the intersection of A, B, so $A \cup B$ is the carrier of a VC, say I:



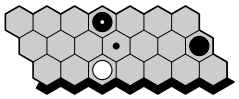
Now block a cell (dotted) from I:



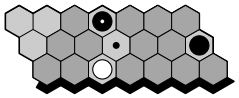
This cell intersects B, D but misses A, C, whose intersection is empty, so $A \cup C$ is the carrier of a new VC, say II:



Now block a cell from II. A good option is a cell that also belongs to I:



This cell misses A, D, whose intersection is empty, so $A \cup D$ is the carrier of a new VC:



In general, finding a single blocked cell is not enough to construct all new VCs. Instead, we need to find a set of blocked cells $B = \{a_1, \ldots, a_n\}$ that satisfies the following: each C in $\mathcal{C}$ contains at least one blocked cell (so $C \cap B \neq \emptyset$) and the intersection of all VSCs which do not contain any blocked cell is empty (so $\bigcap \mathcal{S}' = \emptyset$, where $\mathcal{S}' = \{S \in \mathcal{S} \mid S \cap B = \emptyset\}$). Given a set B of blocked cells, the union of the strategies of $\mathcal{S}'$ is the carrier of a new VC.

Following this approach, we implement Algorithm 2 in a backtracking manner. The main function is BACKTRACK,

Algorithm 2 SEMISCOMBINER$(\mathcal{S}, \mathcal{C})$

Require: VSCs $\mathcal{S}$ and existing VCs $\mathcal{C}$ carrier sets
Ensure: Return a set of carriers of newly created VCs

```
1:  function SemisCombiner(S, C)
2:      return Backtrack(∅, S, C) − C
3:  function Backtrack(F, S, C)
4:      if ⋂S ≠ ∅ then return C
5:      if C = ∅ then C ← {⋃S}
6:      loop
7:          A ← a smallest set from {C − F : C ∈ C}
8:          if A = ∅ then return C
9:          a ← choose element from A
10:         F ← F ∪ {a}
11:         S' ← Filter(S, a),  C' ← Filter(C, a)
12:         C ← C ∪ Backtrack(F, S', C')
13: function Filter(A, a) return {A ∈ A : a ∉ A}
```

called initially at depth 0 in SEMISCOMBINER. When called at depth j, the set of blocked cells $B = \{a_1, \ldots, a_j\}$ has size j. From here, all possible supersets of B are searched. B is not stored explicitly. Instead, filtered VSC and VC carrier sets are passed as arguments $\mathcal{S}$ and $\mathcal{C}$, where a *filtered* carrier set is one in which each carrier contains no blocked cell (so is disjoint with B). Additionally, we pass a set F of forbidden choices of a_{j+1}. F contains all cells a such that the set of blocked cells $B \cup \{a\}$ has been already searched.

BACKTRACK returns all of the VC's carriers, both original and newly created from VSCs, that are disjoint with B.

We now explain BACKTRACK step by step. To start, test whether a VC can be created (line 4). If filtering removes all VCs, create a new VC (line 5). This VC will be disjoint with any VC created so far, because here $\mathcal{S}$ contains only VSCs that are disjoint from B, whereas all previous VCs contain at least one cell from B.

Next, loop over all possible choices of a_{j+1} (variable a). In order to create a new VC, we must filter out all VCs from $\mathcal{C}$. So for each $C \in \mathcal{C}$ we must at some point block a cell of C. But we cannot block any cell from F. In order to minimize the branching factor, we want the smallest possible set difference $C - F$ among all possible C; this is A (line 7). The next blocked cell must be in A.

If A is empty, there is at least one VC that cannot be filtered, so end the search (line 8). If A is not empty, pick an arbitrary a from A (line 9) as a_{j+1}. Now recursively call BACKTRACK on the carrier sets $\mathcal{S}', \mathcal{C}'$ obtained from $\mathcal{S}, \mathcal{C}$ by removing connections containing a (lines 11–12).

We forbid a as a candidate for each future selection of a_i, namely for $i > j$, both in deeper recursive calls and in local future choices of a_{j+1} (line 10), because including it would

produce no new blocked cells, and so no new VCs. Thus search a set B of blocked cells only once.

C. Efficiency

The total number of recursive calls is limited in several ways. We never duplicate a set B of blocked cells, so one upper bound is the number of such sets, namely 2^t, where t is the size of a smallest set that hits all VCs. The number of blocked cell sets that correspond to recursion calls is smaller than this, as the recursion is often broken by the condition at line 4: if B is the blocked set corresponding to the current recursive call and the intersection of VSCs is not empty, then we abort the search for supersets of B. While the input VSCs thus limit the search, the main bounding factor is the number of VCs, both current and newly created. For example, recursion depth is bounded by the number of VCs, so SEMISCOMBINER runs quickly when creating initial connections. This bounding effect is difficult to measure precisely. The number of recursive calls depends not on the number of VSCs, but rather on the number of newly created VCs.

We performed an experiment to measure typical performance, using the position in Figure 4, which has many V(S)Cs.

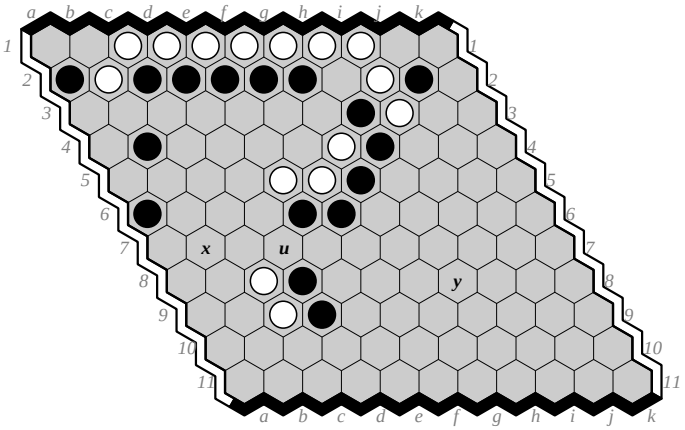


Fig. 4. A position with many VCs and VSCs.

For this position, for each endpoint pair with empty VSC intersection, we called SEMISCOMBINER and gathered data on the $2\,780$ resulting calls. Each data point includes the number of input VSCs, the number of produced final VCs, and the number of recursive BACKTRACK calls made. For each data point, the number of recursive calls was more than $1\,000$.

We used Gnuplot [20] to analyze the $1\,500$ data points with the most recursive calls, finding a function that correlates the number of recursive calls with the number of VSCs and the number of VCs. After drawing various 3D plots, we considered several functions with varying numbers of parameters. For each such function we used Gnuplot's fit command to establish the best value for parameters. We found a good approximation for the number of recursive calls to be $\max(20 \cdot (\#\text{VCs})^{5/4}, 2 \cdot \#\text{VSCs})$. To show how well this fits, for each data point we calculate a fitting factor: a proportion of actual number of recursive calls and its approximation. The maximum value of the fitting factor is 2.98. There are 8 data points with fitting factor over 2, 53 with fitting factor over 1.5, and 294 with fitting factor over 1. Notice that this function is polynomial in the number of VCs and VSCs, and — as expected from our discussion above — reflects that the number of initial VCs have a greater impact on the final number of connections than the number of initial VSCs.

D. Remarks and minor optimizations

1) VC minimality: During intermediate processing, SEMISCOMBINER can construct non-minimal VCs. So as to avoid using these in the construction of further strategies, we discard them before the final set of VCs is returned.

2) Subsequent use of semis-combiner: During the connection construction process, SEMISCOMBINER can be called with the same endpoint pair more than once. In such cases, each subsequent SEMISCOMBINER call in argument $\mathcal{S}$ partitions the VSCs as either processed (already present in the previous call) or not. This allows the search to be stopped whenever BACKTRACK is called with $\mathcal{S}$ containing only processed VSCs, as this cannot produce any new VC.

3) Greedy sum: Whenever a new connection is created by summing VSCs in line 5 of Algorithm 2, rather than sum all VSCs, we greedily add each successive carrier only if it reduces the current intersection of all VSCs. See Algorithm 3. This optimization decreases runtime.

Algorithm 3 Greedy sum

```
1: function GREEDYSUM(S)
2:     X ← ∅, I = U
3:     for all S ∈ S do
4:         if I ∩ S ≠ I then
5:             X ← X ∪ S
6:             I ← I ∩ S
7:     return X
```

VI. LIMITING CONNECTION GROWTH

Semis-combiner runs quickly on large VSC sets and finds all possible H-Search VCs. But this can be too many connections to be useful to a game player.

Consider the position from Figure 4. Here computing black VCs takes 8 minutes, too long for any playing or solving application. This runtime is due simply to there being $696\,901$ VCs and $4\,550\,587$ VSCs.

Between cells x (b7) and y (i8) black has $65\,698$ VSCs. Applying Algorithm 2 to these yields $6\,925$ VCs. The number of recursive BACKTRACK calls is $860\,264$. There are $14\,238$ $x-y$ VSCs with key u (d7); this is the most common key for $x-y$ VSCs. These are created by applying the AND-rule to the 60 x-u and 266 u-y VCs; most such VC pairs yield a new VSC. Thus the number of x-y VSCs with key u can be almost as large as the product of the number of x-u and u-y VCs, which explains the exponential growth of the number of VSCs.

Because of this growth, it is necessary to limit the creation of new VCs.

A. New VC acceptance heuristic

Previous approaches to limit the number of new VCs used soft and hard limits. See §III. We propose another approach. Assume that $\mathcal{C}$ is the set of VC carriers between two fixed cells, and that a new VC with carrier C is created. When is it worth adding C to $\mathcal{C}$? Always, if C is a strict subset of a carrier in $\mathcal{C}$. Never, if C is a superset of such a carrier. But what if C is neither such a strict subset nor a superset? Then C is useful only if it leads to the creation of any new VSCs by the AND-rule. A set of VSCs combine to form a new VC only if their combined intersection is empty; hence it is useful to collect VCs whose intersection (with others with the same endpoints) is as small as possible. So our *acceptance heuristic* is as follows: add a new VC only if

(1) it is a strict subset of an existing VC, or
(2) it reduces the combined intersection of all current VCs.

With this heuristic, the number of VCs stored between two given cells does not exceed the number of cells on the board, which is a constant. So, it limits to a polynomial in the number of cells the total number of connections created, thus avoiding exponential growth.

Not surprisingly, given unlimited computation time, this heuristic weakens the resulting VC engine. However, for fixed time computations, this heuristic generally strengthens the VC engine. See §VII.

This heuristic is not universally better than the standard connection-finding algorithm. Figure 5 shows a position from the 2011 Computer Olympiad Hex competition, with White to move. Here our heuristic fails to find a white side-to-side VSC. Such VSCs are useful in pruning losing moves, and a solver using our new heuristic is slower here than a solver using the standard VC computation.

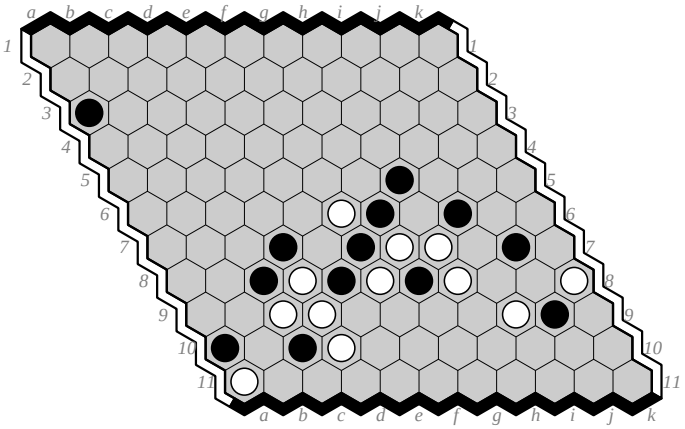


Fig. 5. A VC engine using the acceptance heuristic fails to find a white side-to-side VSC here. White wins, e.g. `i10 e9 e10 b10 b9 a9 b8 a8 b6 b7 c6 c7 e5 d5 e4 d6 d4 c5 c4 b4 b5 h9 i8 f5 e6`.

B. Fast semis-combiner

In our acceptance heuristic, (2) often implies (1). This suggests a simpler heuristic: use only (2). This allows a faster implementation of semis-combiner.

Let $\mathcal{S}$ and $\mathcal{C}$ be SEMISCOMBINER arguments, i.e. existing VSCs/VCs with fixed endpoints. Let $A = \bigcap \mathcal{C}$ be the intersection of these VCs. We want to create a VC C that is not a superset of A. For this we need to find a blocked cell $a \in A$ such that the intersection of VSCs not containing a is empty. We do this by iterating over $a \in A$. After each iteration we either remove a from A or, if we added a new VC C, update A to $A \cap C$. In the latter case, since $a \notin C$, the size of A is reduced by at least one.

We stop the loop whenever we are unable to create a new VC fulfilling (2). This happens if $A = \emptyset$ or $\bigcap \mathcal{S} \neq \emptyset$. We speed up this process by filtering out of $\mathcal{S}$ all supersets of A, since from now on they are useless. See Algorithm 4.

Algorithm 4 FASTSEMISCOMBINER$(\mathcal{S}, \mathcal{C})$

Require: VSCs $\mathcal{S}$ and existing VCs $\mathcal{C}$
Ensure: Return new VCs reducing intersection of $\mathcal{C}$

```
function FastSemisCombiner(S, C)
    C_new ← ∅
    A ← ⋂C
    while A ≠ ∅ do
        S ← {S ∈ S : A ⊈ S}
        if ⋂S ≠ ∅ then return C_new
        a ← choose element from A
        S' ← Filter(S, a)
        if ⋂S' = ∅ then
            C ← GreedySum(S')
            C_new ← C_new ∪ {C}
            A ← A ∩ C
        else
            A ← A − {a}
    return C_new
```

FASTSEMISCOMBINER has at most as many iterations as the size of the VCs intersection, but usually much fewer.

C. Comparison

A VC engine using either FASTSEMISCOMBINER or SEMISCOMBINER together with the acceptance heuristic is usually weaker than a VC engine using only SEMISCOMBINER. We call these VCEs respectively *fast*, *limited*, and *unlimited*. For the position in Figure 4, Figure 6 shows cells VC-connected to the bottom for these VCEs. Notice that unlimited VCE finds more complicated VCs. A white mustplay set for this position is shown in Figure 7. Here fast VCE finds no VSCs between the black sides and so finds no mustplay set, while the mustplay of limited VCE is twice as large as that of unlimited VCE.

Although unlimited VCE is the strongest of these three VCEs, for fixed time computations the speed gains of limited VCE and fast VCE more than compensate for the loss in found connections. As we show in §VII, fast VCE is the strongest of these VCEs.

VII. EXPERIMENTAL RESULTS

We performed experiments to show the strength of our methods. We used these VC engines: *base*, the standard engine from §III; *vc1*, in which base is improved by using move-to-front and by making other minor improvements; *vc2*, a totally

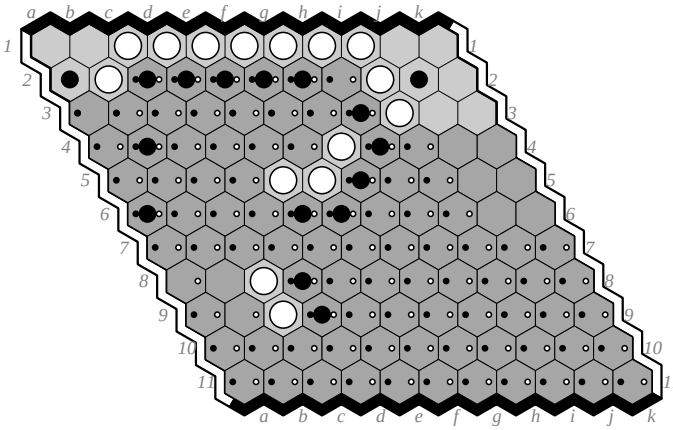


Fig. 6. Cells black-VC-connected to the bottom, as computed by unlimited VCE (shaded), limited VCE (black dot), and fast VCE (white dot).

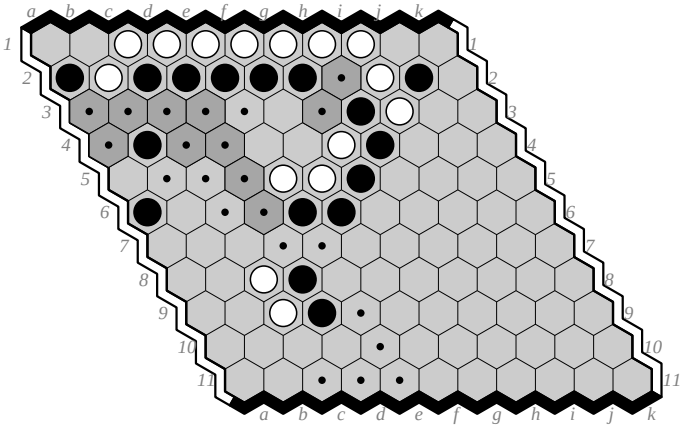


Fig. 7. White mustplay, as found by unlimited VCE (shaded), limited VCE (dotted), and fast VCE (void: no mustplay).

rewritten engine, with our new data structures from §IV. We considered the three variants of vc2 from §VI-C.

We implemented algorithms in the open-source Hex repository Benzene [19], which in turn is built on the open-source game-independent framework Fuego [21]. Tests were performed on three Benzene programs that rely on their VC engines: Solver, MoHex and Wolve.

A. Solver

Solver finds the theoretical value of a position. It uses Focused DFPN [9], [12], a version of DFPN search [22] that is enhanced by the $1+\varepsilon$ trick [23], a VC engine, an inferior cell engine, and electric circuit resistance for move ordering. VCs play a central role: computational efficiency and the strength of the computed VC sets are the main factors in determining runtime. Because Solver depends so critically on VC computation, it is a good benchmark for our improvements.

We used positions of various difficulty, with solving times of vc2-unlimited varying between 400 and 20 000 seconds. We used only relatively challenging positions, since on simple positions there is often a large variance in runtime for different runs.

The first set of positions consists of the ten hardest 8×8 1-move openings. See Figure 8. The second set of positions consists of six 11×11 positions from the 2011 ICGA Olympiad Hex competition [24]. These were found by starting with the final position and proceeding backwards to a moderately difficult position. See Table II.

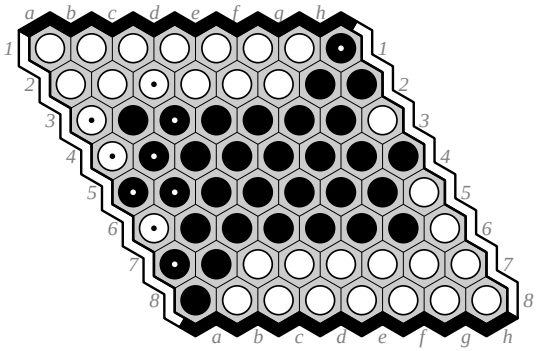


Fig. 8. 1-move 8×8 Hex openings (dotted). Cell color shows winner if black opens there.

TABLE II
POSITIONS FROM THE 2011 OLYMPIAD. THE GAME 6 POSITION HAS BEEN MODIFIED. GAME 3 WAS NOT USED IN THIS BENCHMARK, BUT WAS USED IN §VI-B.

gm.	move sequence black moves first
3	a3 f6 d7 d8 h5 g7 c8 c9 i6 h8 j7 i9 g8 h7 f7 k8 g6 f8 e8 d10 a10 a11 c10 d9 j9
5	a2 g5 d7 d8 e7 e8 f7 f8 g7 g8 c8 c9 h8 h7 a10 a11 b10 b11 c10 c11 i7 i6
6	e2 g5 e6 f7 d8 d7 e7 d9 c9 b11 e8 e9 f8 f9 g8 g9 c10 f4 d5 d6 e5 c11 d10 d11 e10 e11 f10 f11 a11 b10 i9 h10 g10 g11 i10 h9 i8 h8
7	a2 e7 f7 f6 h5 h4 g5 g6 h6 g4 f5 f4 j3 j2 i3 i2 h3 h2 g3 i4 j4 i5 j5 i6 j6 i8 i7 g2 f3 h8 h7 f2 e3 g8
9	c1 f6 g6 g5 i4 i3 h4 h3 g4 g3 f4 f3 d4 e4 c6 d5 c5 c7 d6 d7 e6
10	c1 e7 g6 g7 f7 e9 f8 g5 h5 h4 f6 f5 e6 e5 c6 d4 b5 c3 j3 i4
11	a2 f6 g6 g5 d7 d8 e5 f7 c8 b10 c9 c10 d9 d10 e9 e10

Table III shows our results. We ran our experiments on an Intel Xeon 2.4 GHz. The transposition table size was 2^{25}, which is more than sufficient. All averages are geometric means, since these are more suitable than arithmetic means when measuring a speedup ratio.

TABLE III
AVERAGE SOLVING TIMES AND VC BUILDS FOR DIFFERENT VERSIONS OF SOLVER.

	8×8 openings		Olympiad Games	
version	time	VC builds	time	VC builds
base	4819	355322	10834	203820
+ semis-combiner	5442	294169	10783	177722
+ AND-rule priority	5901	289586	12828	171459
+ store VC-neighbours	4946	290862	10411	166276
+ move-to-front (vc1)	3926	289366	7733	171593
vc2-unlimited	2015	183144	5099	140475
vc2-limited	2398	294490	4808	244548
vc2-fast	1695	295163	3185	244791

The first part of the table shows the improvements obtained by adding various features to the base version VC engine. Semis-combiner yields only a small reduction in runtime, and only for the Olympiad Games set, but for both sets it reduces the number of searched states (which, with succeeding optimizations, resulted in significant runtime reduction).

A VC engine can postpone running semis-combiner until a sufficiently large set of VSCs has accumulated. We implemented such an *AND-rule priority* scheme. While this scheme slightly decreased the number of VC builds, there was no runtime reduction, presumably because the base version lacks some optimizations that are implemented in vc2.

The other two base optimizations achieved a significant speedup, especially *move-to-front*. *Storing VC-neighbours* allows for iteration only over cells that are VC-connected with a given endpoint. This is useful mainly within application of the AND-rule. As mentioned earlier, move-to-front requires the use of a vector instead of a linked list; however, it eliminates the sorting of V(S)Cs by carrier size, which is no longer necessary because of the natural order found by move-to-front. Its high performance is as expected.

As explained in §IV, we designed new data structures to exploit our numerous optimizations. The second part of the table shows the improvement in solving time and the number of visited states. Version vc2-fast visits a comparable number of states to vc2-limited, suggesting that the former might be superior to the latter. While vc2-limited gives no improvement over vc2-unlimited, vc2-fast is faster, especially on larger boards. Thus vc2-fast is arguably the best of these variants for use in Solver. However, the superiority of vcs-fast over the other variants is not universal: there are positions — e.g. in Figure 5 — where it fails due to missing VCs.

B. Playing programs

In order to measure the comparative strengths of these variants in head-to-head competition, we played a large tournament involving many versions of the two Benzene players, namely MoHex and Wolve. For each of these two players, we created various versions by selecting a VC engine, selecting a VC computation variant, and deciding whether to use parallel Solver. The VC engine is one of base, vc1 and vc2. For vc2, we used one of the variants: unlimited, limited, or fast. Thus, for each VC engine, we considered 5 different versions of the VC engine, which together with the parallel Solver choice yields 10 different players. So in total 20 different players (10 MoHex, 10 Wolve) competed in the tournament.

The tournament was played with 10 seconds per move on an 11×11 board. Because we are interested in playing strength per unit time, and because the different VC variants take differing amounts of time and find different sets of connections, using fixed time limit is a better measure than either fixing the number of playouts in MoHex or fixing the search depth in Wolve. Each two players played 72 games with each other. For openings, we used 36 relatively balanced single stone openings: a2 to k2, a10 to k10, b1 to j1, and b11 to j11. For each opening and each pair of players, two games were played: each player once as black, and once as white. So, each player played 1368 games, and the total number of tournament games was 13 680.

Table IV shows the results. The Elo score is computed by BayesElo [25] with error ±11 and confidence 80%. A base Elo score of zero was assigned to Mohex with the base VC engine and no solver. The results are discussed separately for MoHex and Wolve.

TABLE IV
RESULT OF MOHEX AND WOLVE TOURNAMENT FOR DIFFERENT SETTINGS.

Rank	Program	VC engine & variant	solver	Elo score
1	MoHex	vc2-unlimited	yes	111
2	MoHex	vc2-limited	yes	93
3	MoHex	vc2-fast	yes	88
4	Wolve	base	yes	85
5	Wolve	vc1	yes	70
6	Wolve	base	no	69
7	MoHex	vc2-unlimited	no	48
8	MoHex	vc1	yes	47
9	Wolve	vc1	no	46
10	MoHex	vc2-fast	no	44
11	MoHex	vc2-limited	no	40
12	MoHex	vc1	no	37
13	MoHex	base	yes	37
14	Wolve	vc2-unlimited	no	33
15	Wolve	vc2-unlimited	yes	28
16	Wolve	vc2-limited	yes	27
17	Wolve	vc2-fast	yes	27
18	Wolve	vc2-fast	no	11
19	Wolve	vc2-limited	no	1
20	MoHex	base	no	0

1) MoHex: MoHex [7] uses Monte Carlo tree search [26], [27]. The VC engine is used as follows. As soon as node is visited 400 times, VCs are built for the node, and a mustplay region is computed. This allows the pruning of inferior moves, and detects win or loss long before the board is full.

The parallel Solver — if used — executes on a separate thread. If Solver detects a win or loss, the associated move is used by the player; otherwise, the player uses MCTS to select its move.

It is no surprise that — irrespective of whether Solver is used — base is the weakest VC engine. It is also no surprise that Solver increases playing strength: base is 37 Elo stronger (with Solver on than with Solver off), vc1 is 10 Elo stronger, and the three variants of vc2 are 40–48 Elo stronger. With Solver, vc2 is much stronger than other VC engine versions. Among vc2 variants, vc2-unlimited is best, while vc2-fast and vc2-limited are comparable.

Our experiment shows that MoHex benefits greatly by increasing the strength and/or speed of its VC engine, and suggests that vc2-unlimited is the preferred variant. However, under real tournament settings — those used in competitions, typically 16 MCTS threads for MCTS and about 1 minute per move — the number of VCs occasionally explodes to the extent that MCTS search is crippled. Thus, in such tournaments, vc-fast is preferred: this gives the fastest VC computation, allowing the highest number of playouts per second, yet maintains relatively accurate move selection.

2) Wolve: Wolve uses electric circuit resistance evaluation enhanced by adding edges for VC-connected cells [10]. Move ordering and pruning is based on this evaluation. VCs are also used for move pruning, by finding the mustplay region. The quality of the computed VCs influences move ordering and pruning, while the depth of search depends on speed. Thus the choice of best VCE variant is not obvious.

As with MoHex, Solver can be run in parallel, but here its benefit is questionable. This is presumably because of the alpha-beta search used by Wolve often yields similar

results to the DFPN search used by Solver, yielding little extra information. By contrast, MCTS often yields different results from DFPN search, so using Solver adds much extra information.

Here, the results are counter-intuitive: the stronger or faster the VC engine, the weaker the variant of Wolve. In particular, the variants vc2-limited and vc2-fast, which build sparse and so relatively low quality sets of VCS, perform very poorly. Thus one might conclude that the accuracy of the VC engine is important in Wolve.

However vc1 and vc2-unlimited, which have the highest possible accuracy in building VCs, are weaker than base. Moreover, vc1 is stronger than vc2-unlimited; the main difference between these variants is that vc2-unlimited is faster than vc1, which means that alpha-beta can search more deeply.

This counter-intuitive behaviour has been observed in players similar to Wolve such as Six [15], in which a deeper search can yield a weaker program. One possible explanation is the peculiarities of the evaluation function, which is based on an electric resistance model of the position. Nodes that are far apart in the tree, especially if they are at different depths, are often incorrectly ranked by this function, and so deep searches may deviate significantly from correct play. We suspect that this evaluation is biased towards positions with more stones, and that the relative error among positions with many stones is much greater than the relative error among positions with few stones. Each of these factors could lead to instability in deep searches.

VIII. Conclusions

A. Our Results

We have presented an improved algorithm for finding and using virtual connections in a connection-game automated player. Our algorithm reorganizes H-search and stores connections more efficiently. It uses move-to-front, which speeds the verification of new minimal connections.

Our most important contribution is arguably our new OR-rule (semis-combiner), which finds a restricted set of new minimal connections, but more quickly than ordinary H-search. This allows us to apply our OR-rule for arbitrary large set of VSCs; previous techniques had to limit the number of VSCs used in an OR-rule to four.

Semis-combiner can produce an intractably large number of connections, becoming too slow for practical usage. Thus we designed a VC acceptance heuristic that allows the application of a new OR-rule (fast semis-combiner). This final optimization yields a DFPN-based solver that is about three times faster than with the previous approach.

B. Future Work

Our algorithm is not universally better than previous methods for some positions, because it limits the set of VCS it finds, it fails to solve the position in a reasonable time. Other approaches to limiting the set of discovered VCs might be successful.

Further minor optimizations might be possible. For example, changing the order in which connections are processed might improve runtime, or the strength of new connection sets.

FastVC Search could be applied to other programs which, like MoHex and Wolve, rely on connection computations. MoHex spends most of it time in connection computation, so our new algorithms reduce its running time significantly. By contrast, our new algorithms did not strengthen Wolve, presumably because for Wolve the critical factor is not how long it takes to compute connections, but rather the richness of the discovered set than occasionally increased search depth. Exploring various aspect of connection strategies and how they contribute to the strengths of various players and solvers is another area for study.

Acknowledgements

We thank Martin Mueller for the use of his machines for some of our experiments.

References

[1] P. Hein, “Vil de laere Polygon?” *Politiken*, December 1942.
[2] J. Nash, “Some games and machines for playing them,” RAND, Tech. Rep. D-1164, February 1952.
[3] H. W. Kuhn and S. Nasar, Eds., *The Essential John Nash*. Princeton University Press, 2002.
[4] S. Nasar, *A Beautiful Mind: A Biography of John Forbes Nash, Jr.* Simon and Schuster, 1998.
[5] C. E. Shannon, “Computers and automata,” *Proceedings of the Institute of Radio Engineers*, vol. 41, pp. 1234–1241, 1953.
[6] S. Reisch, “Hex ist PSPACE-vollständig,” *Acta Informatica*, vol. 15, pp. 167–191, 1981.
[7] B. Arneson, R. B. Hayward, and P. Henderson, “Monte Carlo Tree Search in Hex,” *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 4, pp. 251 –258, 2010.
[8] V. V. Anshelevich, “The game of Hex: An automatic theorem proving approach to game programming,” in *AAAI/IAAI*. Menlo Park: AAAI Press / The MIT Press, 2000, pp. 189–194.
[9] B. Arneson, R. B. Hayward, and P. Henderson, “Solving Hex: Beyond humans,” in *Computers and Games 2010*, ser. LNCS, H. J. van den Herik, H. Iida, and A. Plaat, Eds. Springer, 2011, vol. 6515, pp. 1–10.
[10] V. V. Anshelevich, “A hierarchical approach to computer Hex,” *Artificial Intelligence*, vol. 134, no. 1–2, pp. 101–120, 2002.
[11] R. Hayward, Y. Björnsson, M. Johanson, M. Kan, N. Po, and J. van Rijswijck, “Solving 7×7 Hex with domination, fill-in, and virtual connections,” *Theoretical Computer Science*, vol. 349, no. 2, pp. 123–139, 2005.
[12] P. Henderson, “Playing and solving Hex,” Ph.D. dissertation, University of Alberta, 2010, http://webdocs.cs.ualberta.ca/~hayward/theses/ph.pdf.
[13] P. Henderson and R. B. Hayward, “Captured-reversible moves and star decomposition domination in Hex,” *Integers*, vol. 13, p. #G1, 2013.
[14] G. Melis and R. Hayward, “Six wins Hex tournament,” *ICGA Journal*, vol. 26, no. 4, pp. 277–280, 2003.
[15] G. Melis, “Six,” six.retes.hu/, 2006.
[16] R. Rasmussen, “Algorithmic approaches for playing and solving Shannon games,” Ph.D. dissertation, Queensland University of Technology, Brisbane, Queensland, Australia, 2007.
[17] V. V. Anshelevich, “The game of Hex: The hierarchical approach,” July 2000, combinatorial Game Theory Workshop. MSRI, Berkeley. http://www.msri.org/publications/ln/msri/2000/gametheory/anshelevich/1/.
[18] P. Henderson, B. Arneson, and R. Hayward, “Hex, braids, the crossing rule, and XH-search,” in *ACG*, ser. Lecture Notes in Computer Science, J. van den Herik and P. Spronck, Eds., vol. 6048. Springer, 2010, pp. 88–98.
[19] B. Arneson, P. Henderson, and R. B. Hayward, “Benzene,” 2009–2012, http://benzene.sourceforge.net/.
[20] T. Williams, C. Kelley, and many others, “Gnuplot 4.4: an interactive plotting program,” http://gnuplot.sourceforge.net/, March 2011.
[21] M. Enzenberger, M. Müller, B. Arneson, R. Segal, F. Xie, and A. Huang, “Fuego,” 2007–2012, http://fuego.sourceforge.net/.

[22] A. Nagai, "Df-pn algorithm for searching and/or trees and its applications," Ph.D. Thesis, Dept. of Information Science, University of Tokyo, Tokyo, Japan, 2002.
[23] J. Pawlewicz and L. Lew, "Improving depth-first pn-search: $1+\varepsilon$ trick," in *Computers and Games 2006*, ser. LNCS, H. J. van den Herik, P. Ciancarini, and H. Donkers, Eds. Springer, 2007, vol. 4630, pp. 160–170.
[24] R. B. Hayward, "Mohex wins Hex tournament," *ICGA Journal*, vol. 35, no. 2, pp. 124–127, June 2012.
[25] R. Coulom, "Bayesian elo rating," 2010, http://remi.coulom.free.fr/Bayesian-Elo.
[26] ——, "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search," in *Proc. 5th Int. Conf. Comput. and Games, LNCS 4630*, Turin, Italy, 2007, pp. 72–83. [Online]. Available: http://portal.acm.org/citation.cfm?id=1777826.1777833
[27] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *ECML*, ser. Lecture Notes in Computer Science, J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, Eds., vol. 4212. Springer, 2006, pp. 282–293.

Jakub Pawlewicz received the Ph.D. degree in Computer Science from University of Warsaw, Poland, in 2009.

His main field of interest is Artificial Intelligence in Games. In 2011-2012 he held a post-doctoral position in Computing Science at the University of Alberta in Canada, during which time he became a co-author of the Benzene code project.

Currently he is an adjunct professor at University of Warsaw. His main research interest is the design of new tree search algorithms.

Ryan B. Hayward is a professor in the Department of Computing Science at the University of Alberta. He holds a PhD in computer science (McGill 1987, supervisor Vasek Chvatal) and an MSc (Queen's/Kingston 1982, supervisors Peter Taylor and Selim Akl) and a BSc (honours, Queen's/Kingston 1981) in math.

Before joining the department in 1999, he was an assistant and then associate professor at Lethbridge (92-99), asssistant professor at Queen's/Kingston (90-92) and Rutgers (86-89), and an Alexander von Humboldt Fellow in Bonn (89/90).

His research interests include discrete algorithmics and game-tree search. Together with Broderick Arneson, Philip Henderson, Shih-Chieh Huang, and Jakub Pawlewicz, he is a co-author of MoHex, the gold-medal winning Hex player, and Solver, the Hex solver that has solved all 9x9 and two 10x10 openings.

Philip Henderson is a software engineer at Google, improving the machine learning systems that predict the clickthrough rate for search ads.

He is interested in graph theory and combinatorial game theory, and in 2010 completed his PhD on the game of Hex at the University of Alberta.

Broderick Arneson holds an MSc in Computing Science (Alberta 2007, supervisors Piotr Rudnicki and Lorna Stewart). From 2007 until 2014 he was a research assistant in the Hex and Go groups in Computing Science at the University of Alberta. In particular, he implemented most of the code for Benzene, the primary code base for the Hex group. Since 2014 he is a software engineer at Google.