Learning How to Play Hex

Kenneth Kahl, Stefan Edelkamp, and Lars Hildebrand

Computer Science Department
University of Dortmund[⋆]

Abstract. In Hex two players try to connect opposing sides by placing pieces onto a rhombus-shaped board of hexagons. The game has a high strategic complexity and the number of possible board positions is larger than in Chess. There are already some Hex programs of recognizable strength, but which still play on a level below very strong human players. One of their major weaknesses is the time for evaluating a board.

In this work we apply machine learning for the computer player to improve his play by generating an fast evaluation function and lookup procedure for pattern endgame databases. The data structures used are neural networks for the evaluation of a position and limited branching trees to determine if a position can be classified as won or lost.

1 Introduction

In this work we study the strategic game Hex a fully observable two-player zero-sum board game. In the same class of problems we also find Chess and Checkers, which both refer to a long line of AI research. The first article to computer Chess was published by Shannon [14], while the latest result is that a commercial Chess playing program has beaten the world champion in a match on a regular PC[1] [19]. Schaeffer et al. [13] could prove that a certain Checkers opening (called the *White Doctor*) results in a draw, assuming optimal play.

Hex has been invented by the Danish mathematician Piet Hein in the year 1942, calling it *polygon*. Independently, John F. Nash studied the game in 1947, and wrote an article about it in 1952 [10]. In the same year, Parker Brothers firstly sold the game using the name Hex. For the general audience, Hex has been advertised by Martin Gardner [4,5]. The game is played on a squared-sized rhombus-shaped board, consisting of B hexagons. The board size can be scaled, the classical size is 11×11. The two players are called *black* and *white*. Alternatively, each player places one piece on the board of his color. The task in the game is to generate a chain of own pieces that connects two opposite sides of the board (see Figure 1). The number of reachable boards accumulate to

$$\binom{121}{1,0,120} + \binom{121}{1,1,119} + \binom{121}{2,1,118} + \ldots + \binom{121}{61,60,0} \approx 4.7 \times 10^{56}.$$

⋆ The work is supported by DFG in the project ED-74/3.

1 Deep Fritz won against Vladimir Kramnik with 4:2 running on a Intel dual-core with 3 Ghz and 4 GB RAM.

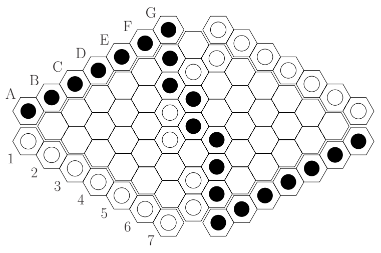


Fig. 1. Terminal position in Hex - game won by black

As intermediate boards may already be classified either to be won for black or for white, the number feasible positions is lower – an upper bound of 2.38×10^{56} has been computed by Browne [2].

As the first player has a big advantage, there is an additional, so-called pie-rule. The first piece is set by player black. Next player white has the choice whether or not he wants to continue the play or swap colors for the rest of the game. The pie-rule is applicable only after the first move. It is well-known that Hex is won by the first player [3]. As the proof is not constructive, for the design of a Hex game playing program the result is of no use. For a growing board, Hex is PSPACE-complete [12]. For board sizes 7×7, 8×8 and 9×9 without pie-rule winning strategies have been provided by [20], together with a winning strategy for the 7×7 game with pie-rule.

In this paper we apply machine learning algorithms to generate a strong Hex player. On the one hand, we train a neural network with the evaluation function computed by the state-of-the-art program Six[2]. The learned function can be evaluated much faster than the original one, which for a given time slot allows to search the game tree much deeper. For Checkers [16] and Backgammon [17] neural networks already yield good players, while for Go so far no strong player could be crated [2]. Moreover, the Chess program *NeuroChess* could not advance to human play [18]. These approaches learn from the final outcome of games, and they recursively learn the evaluation function. As the second learning mechanism, we will construct a database, in which goals are stored in form of sub-boards (patterns). As a compromise between space and time for insertion and lookup, we propose *limited branching trees*. Additionally, when inserting a goal pattern into the database, symmetric patterns are taken care of.

The paper is structured as follows. First we introduce virtual connections, as they play a central role for the evaluation function and goal detection in the world's best Hex playing program Six. Then we turn to learning the evaluation function by training neural networks and cross-validation to detect the best network structure. Next we consider limited branching trees as the data structure for storing goal pattern. This subset dictionary structure supports containment queries and provides a fair compromise between searching time and memory consumption. We then turn to experiments that we obtained by integrating

[2] `http://six.retes.hu`

the above technique into a game playing system. The much larger number of evaluated nodes per second allows deeper searches in the game tree and earlier matches in the databases, and, subsequently, stronger play.

2 Virtual Connections

The current state-of-the-art program Six uses an unusual approach of electrical circuit theory to combine the influence of sub-positions – virtual connections – to larger ones. The program provides an improved implementation of Hexy [1], whose designer has invented the theory of virtual connections.

A *virtual semi-connection* (with respect to a given support cell set) of the board connects two pieces (a.k.a two groups of pieces) provided that the player to close the connection moves first. A (full) *virtual connection* allows to connect pieces even if it is the opponent's turn to move.

An example is given in Figure 2. Black is requested to transform the virtual to a real connection, the shaded cells denote the support. For the first case (a), if its white to move, he cannot avoid black's connection between the two black pieces on the two shaded cells, but he can escape the connection to the right; if it is black to move the connection is trivial. In the second case (b) white can avoid black's connection by placing a piece on the shaded cell in the center.

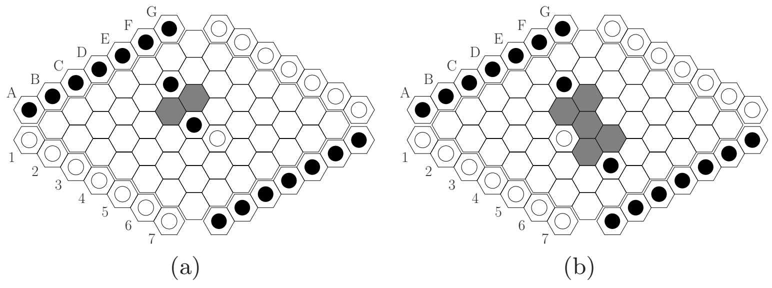


Fig. 2. Virtual connection (left) and virtual semi-connection (right) in Hex

If a virtual connection between opposite boarders for one player is established, the game is won (assuming optimal play). The main aspect of evaluating the strength of the position in Six is to find as many virtual (semi-)connections as possible to merge them to more complex patterns and to combine all established connections to an overall game-playing value that is needed to evaluate leaves in the game playing search tree.

Smaller connections are merged to larger ones by applying *and*-, and *or*-rules. The *and*-rule appends two virtual connections, while the *or*-rule combines two virtual semi-connection to one virtual one. Using an analogy to electronic circuit theory, transforming all virtual connections into a single number is based on computing the resistance for the entire board induced by the resistance for individual cells and connected piece groups.

With this approach Six generates an expressive function to evaluate boards, which itself can look 20 moves (and more) ahead. As a drawback, due to the fact that the evaluation of a single board is quite complex, Six performs a very shallow game tree search, mostly consisting of not more than 1 move for each player. For deeper searches, as e.g. common in Chess, the time for evaluating one position has to be accelerated.

We decided to improve the performance by applying machine learning. As virtual connections actually serve two purposes, namely evaluating a position and showing whether it is terminating, two different strategies have been developed. Firstly, we monitor the evaluation function and model it using function approximation. A natural choice are neural networks. Secondly, we propose a data structure to store virtual connections that represent a terminal position to generate a pattern database that answers so-called containment queries.

3 Learning the Evaluation Function

The purpose to learn the evaluation function of Six with multi-layered feed-forward neural networks is to reduce the time for evaluating boards in order to be able to search deeper in the game tree. We take an input neuron for every cell of the board. The input cell is 1, if a white piece occupies this cell, 0, if the cell is empty, and -1, if the cell is occupied by black. In the example of Figure 3 we see a board that is encoded in the following way:

```
0 1 0 0 0 0 0 0 0 0 -1 0 0 0 1 0 0 0 0 0 0 0 0 0 -1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -1 0 0 0 1 3.220380.
```

The first 49 numbers are the inputs for the neural network, while the last number is the evaluation of Six for this board. The neural network has one output neuron, that provides the learned value of the board.

During self- and random play of Six we were able to capture every evaluated board to generate a set of training and validation examples for the neural network. At first we specified the structure of our neural network for the 6×6

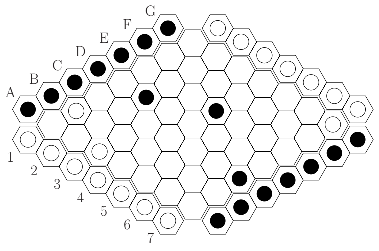


Fig. 3. An evaluated board in Hex

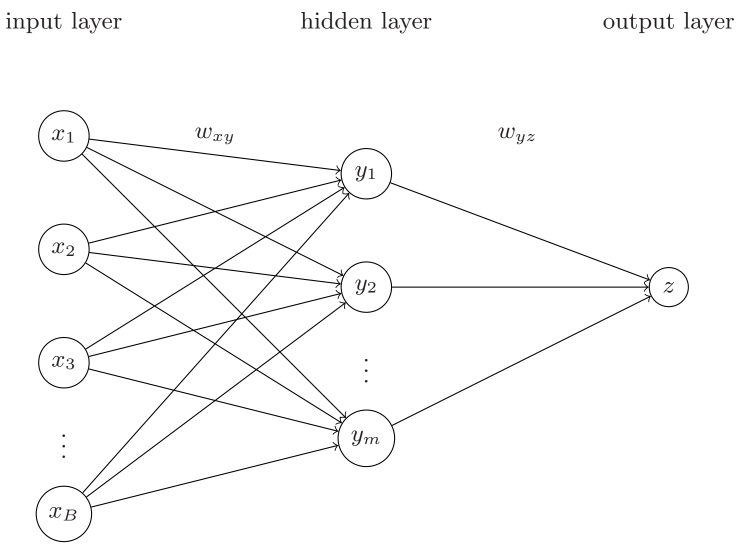


Fig. 4. Structure of the neural network

board. We used a set of 112,737 training examples. After experiments with network structures of up to three hidden layers, we decided to use one hidden layer. We applied a ten-fold cross-validation, where we varied the number of neurons on the hidden layer and the number of epochs, to find a network structure which offers a good generalization. The first criterion for the quality of the generalization was the arithmetic mean of the validation sets, the second criterion was the standard deviation. The best network structure found for the 6×6 board was a network with one hidden layer of $m = 30$ neurons and one output neuron, which was trained for 240 epochs (see Figure 4).

For the 11×11 board we used a set of 142,383 examples. The best network structure we could find is a net with one hidden layer with 50 neurons and one output layer, which was trained over 300 epochs.

Another approach that we implemented was the use of more than one network for evaluating the boards, using a layered partition of the example set. For each search depth (ply) we learned a different evaluation function. Since for the first moves there is only a small number of different boards, the neural network could learn the examples exactly, one property that complies with the experiments.

4 Time Complexity

Now we will show, that our approach outperforms Six. We analyze the complexity for evaluating a board for both programs.

Theorem 1. *(Complexity Neural Network Evaluation) Given that the hidden layer in the network has $m < B$ neurons, the running time of the neural network evaluation amounts to at most $O(mB)$.*

Proof. Since the running time for calculating the output of a neural network depends on the inter-connections of the network (it has to be traversed once), we only need to specify the number of network edges. As the network is fully connected, $O(mB)$ steps are executed to evaluate a position.

In the following, we support the observation that the time for evaluating a board in Six is larger than for the neural network with a theoretical comparison. Later on, we will see how the results match with the experimental outcome.

The running time for the evaluation function in Six is dominated by the algorithm *H-Search* [1]. The procedure is rather complex, and takes a maximum number of virtual connections (θ) as well as the maximal number of virtual connections for applying an *or*-rule (θ') as input parameters. For evaluating a board, three steps are executed:

1. The group partition is renewed and the virtual (semi-)connections are adapted.
2. Procedure *H-Search* is executed.
3. The electrical resistance of the circuit is computed.

Suppose a new black piece is placed on the board. If the piece is adjacent to no other black pieces on the board, an individual group is created. If the piece is, however, adjacent to one or more existing one, the groups have to be merged using any union-find data structure. Let B be the number of board cells and θ be a threshold on the number of virtual connections.

Lemma 1. *(Step 1) Updating the groups in Six requires $O(B^2\theta)$ operations*

Proof. In Six for each adjacent cell c (out of 6 possible), all groups that contain c are traversed. Since the groups consist of less than B cells the update runs in time $O(B)$. To adapt the virtual and virtual semi-connections, all lists for the support of the group pairs are scanned, which accumulates to $O(B^2\theta)$. Therefore, for step 1 Six requires $O(B) + O(B^2\theta) = O(B^2\theta)$ operations.

Procedure H-search updates the lists of virtual (semi-)connections. More precisely, the lists of supports of all virtual connections $VC(g_1, g_2)$ (for all groups g_1, g_2), and all virtual semi-connections $SC(g_1, g_2)$ (for all groups g_1, g_2) are updated. Initially, these lists are empty. Unfortunately, the running time of this approach is large.

Lemma 2. *(Step 2) H-Search as implemented in Six requires $O(B^3\theta^{2+\theta'})$ steps.*

Proof. See [8]

To generate the circuit for each two groups, a wire is inserted into table T, if there is a virtual connection between the two groups by scanning the list of groups (computed by H-Search). The list consists of at most B^2 pairs of groups. By playing pieces the number of group pairs can only decrease. According to the Kirchoff's rule, starting with a set of linear equations Six computes the overall resistance [9].

Lemma 3. *(Step 3) The computation of the resistance takes $O(B^3)$ operations.*

Proof. For the insertion of wires all groups are scanned, which takes $O(B^2)$ operations. Solving the set of linear equations can be done in $O(B^3)$ steps.

Theorem 2. *(Time Complexity Six) Computing the evaluation function for requires $O(B^3\theta^{2+\theta'})$ operations.*

Proof. The evaluation function executes the above three steps. Using the lemmas 1, 2 and 3, we arrive at $O(B^2\theta) + O(B^3\theta^2\theta^{\theta'}) + O(B^2) = O(B^3\theta^2\theta^{\sigma})$ steps to evaluate a board.

Even if θ and θ' are constants, H-Search requires $O(B^3)$ steps.

5 Goal Pattern Databases

The problem of finding an element in a set of elements such that this element is a subset (or a superset) of the query occurs in many applications, e.g., the matching of a large number of production rules, the identification of inconsistent subgoals in AI planning, and the detection of potential periodic chains in labeled tableau systems for modal logics. Moreover, efficiently storing and searching partial information is central to many learning processes.

5.1 Subset Dictionaries

For state space search the stored sets often correspond to partially specified state vectors or *patterns*. As an example consider the solitaire game SOKOBAN [7], together with a selection of dead-end patterns. As every given state is unsolvable, if the dead-end pattern is a subset of it, we wish to quickly detect, whether or not such dead-end pattern is present in the data structure.

Definition 1. *(*SUBSET QUERY *and* CONTAINMENT QUERY *Problem, Subset Dictionary) Let D be a set of n subsets over a universe U. The* SUBSET QUERY *(*CONTAINMENT QUERY*) problem asks for any query set $q \subseteq D$ if there is any $p \in D$ with $q \subseteq p$ ($p \subseteq q$). A* subset dictionary *is an abstract data structure providing insertion of sets to D, while supporting subset and containment queries.*

Since p is a subset of q if and only if its complement is a superset of the complement of q the two query problems are equivalent.

For HEX, we have that each board is an element of U. Inserting a pattern to the goal database amounts to inserting a subset of U to the subset dictionary. In HEX a goal pattern is a virtual connection between both sides of the board (see Figure 7). Subsequently, determining whether or not a state has a match with a stored pattern in the dictionary, is a containment query.

Definition 2. *(*PARTIAL MATCH*) Let $*$ denote a special* don't care *symbol that matches every character contained in an alphabet Σ. Given a set D of n vectors over Σ, the* PARTIAL MATCH *problem asks for a data structure, which for any query $q \in \Sigma \cup \{*\}$ detects if there is any entry p in D such that q matches p.*

The application for this problem is to solve approximate matching problems in information retrieval. A sample application is a crossword puzzle dictionary. A query like `B*T**R` in the Crossword Puzzle would be answered with words like `BETTER`, `BITTER`, `BUTLER`, or `BUTTER`.

Theorem 3. *(Equivalence* Partial Match *and* Subset Query *Problems) The* Partial Match *problem is equivalent to the* Subset Query *problem.*

Proof. As we can replace any algorithm for solving the Partial Match problem to handle binary symbols by using their binary representation, it is sufficient to consider the alphabet $\Sigma = \{0, 1\}$.

In order to reduce the Partial Match to the Subset Query problem, we replace each $p \in D$ by a set of all pairs (i, p_i) for all $i = 1, \ldots, |U|$. Moreover, we replace each query q by a set of all pairs (i, q_i) provided that q is not the don't care symbol $*$. Solving this instance to the Subset Query problem also solves the Partial Match problem.

In order to reduce the Subset Query to the Partial Match problem, we replace each database set with its characteristic vector, and replace query set q by its characteristic vector in which zeros are replaced with don't cares.

As the Subset Query problem is equivalent to the Containment Query problem, the latter one can also be solved by algorithms designed for the Partial Match problem.

One possible implementation that immediately comes to mind is a *trie*[3]. It compares a query string with all stored entries. Unfortunately, tries for the Partial Match problem can introduce large searching times as each don't care symbol induces a branching.

The next option is to store all possible queries in an array. This solution has constant search time but an obvious problem – the structure can become very large. An alternative to reduce the space complexity for the array representation is to hash the query sets into a smaller table. The lists in the chained hash tables again correspond to database sets. However, the lists have to be searched to filter the elements that match.

5.2 Limited Branching Trees

Our compromise between a trie and a hash table subset dictionary data structure consists of an ordered list of tries. Insertion is similar to ordinary trie insertion with the exception that we maintain a distinctive root for the first element in the sorted representation of the set.

This choice of the data structure adapts *unlimited branching tree* as [6] to our requirements. As the branching factor is bounded, our data structure is referred to as *limited branching trees*, LB trees for short.

[3] A trie is a lexicographic search tree structure, in which each node spawns at most $|\Sigma|$ children. The transitions are labeled by $a \in \Sigma$ and are mutually exclusive for two successors of a state. Leaf nodes correspond to stored strings.

The access operation *insert* and *search* in such limited branching trie are realized as follows. Insertion simply traverses the root list to find whether or not a matching root element is present. In case we find a root element the implementation of the ordinary insert routine for the corresponding trie (not shown) is called. In case there is no such element a new one is constructed and inserted to the list. The running time of the algorithm is $O(k + l)$, where k is the size of the current trie list and l the number of elements in the inserted set – plus the time $O(l \log l)$ to sort the elements. As with Hex it is often the case that all elements are selected from the set $\{1, \ldots, n\}$ such that the running time is $O(n)$ altogether (given that all elements are distinct, linear sorting algorithms such as bucket sort apply).

```
Algorithm: Insert
Input: Limited branching tree $L = (T_1, \ldots, T_k)$, sorted set $p = \{p_1, \ldots, p_l\}$
Output: Modified data structure
1 foreach $i$ in $\{1, \ldots, k\}$ do
2     if $(p_1 = \text{root}(T_i))$ then
3         return Trie-Insert$(T_i, p)$
4     end
5 end
6 Generate new trie $T'$ from $p$
7 Insert $T'$ into list $L$
```

Algorithm 1. Inserting a set in a limited branching tree

In Algorithm 2 we show a possible implementation for the lookup. First all root elements matching the query are retrieved. Then the corresponding tries are searched individually for a possible match with the query. As both the query and the stored set are sorted, the match is available in linear time with respect to the query set. The number of root elements that have to be processed can grow considerably and is bounded by the size of the universe U.

The worst-case running time of the algorithm is $O(km)$, where k is the length of the trie list and where m is the size of the query set – plus the time $O(m \log m)$ to sort the query elements. If all set elements are drawn from the set $\{1, \ldots, n\}$ the worst-case running time is bounded by $O(n^2)$.

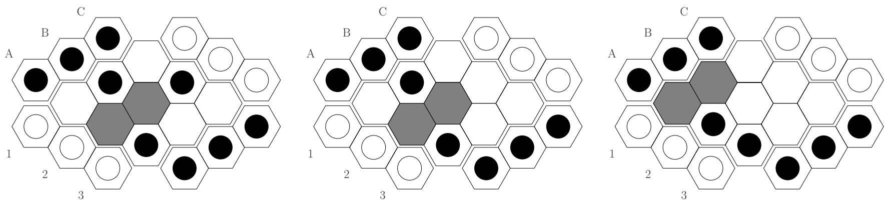
5.3 Adaption to Hex

For adapting the above structure to a goal pattern database for Hex we remind that a goal pattern represents a virtual connection and consists of a set s of cells *and* a set of boolean variables b for their occupation. As we store many patterns, we actually maintains a set of sets S for the cells and a set of sets B for the occupation. For each $s \in S$ we have a matching $b \in B$.

Let B be the number of cells on the board. We assume queries of the form $q = (q_s = (q_{s_1}, \ldots, q_{s_B}), q_b = (q_{b_1}, \ldots, q_{b_B}))$ with $q_{s_i} \in \{1, \ldots, B\}$ and $q_{b_i} \in \{0, 1\}$ for $i \in \{1, \ldots, B\}$. The answer given is whether or not $s \subseteq q_s$ for one $s \in S$, given that the Boolean assignments in q_b match with the one in b.

```
Algorithm: Lookup
Input: Limited branching tree $L = (T_1, \dots, T_k)$, sorted query $q = \{q_1, \dots, q_m\}$
Output: Flag indicating whether or not $p$ contained in $L$ with $q \supseteq p$
1  $Q \leftarrow \emptyset$
2  foreach $i \in \{1, \dots, k\}$ do
3      if (root($T_i$) $\in q$) then
4          $Q \leftarrow Q \cup \{T_i\}$
5      end
6  end
7  foreach $T_i \in Q$ do
8      if Trie-Lookup($T_i, q$) then
9          return true
10     end
11 end
12 return false
```

Algorithm 2. Searching for subsets in a limited branching tree



(a) Element $A = (s_A, b_A)$ (b) Element $B = (s_B, b_B)$ (c) Element $C = (s_C, b_C)$

Fig. 5. Goal patterns database example

A node K in the LB tree for the pattern search in Hex represents a cell on the board and consists of three components: the index $I(K)$ of the node K, a sentinel $E(K)$, and a boolean flag $U(K)$ denoting either a black (*true*) or an empty cell (*false*).

As an example, we illustrate the insertion of the patterns of Figure 5 into the empty dictionary. Three patterns, which all show a clear win for black on a 3×3 board, are processed one-by-one. The resulting LB-tree is shown in Figure 6. It consists of two tries, on of which contains a branching.

Beside the color of the pieces there is no difference between a black and a white winning position, such that in order to save memory we only stored black's winning connections. By swapping the colors and reflecting the board before querying the database, its entries can be reused for white. During the training the opposite strategy applies, all winning connections for white are stored as if they were won by black.

As illustrated in Figure 7 there are many further patterns that can be obtained through translation, reflection, and rotation. For example, the goal pattern in (a) is translated into the pattern in (b). Depending on the width of the pattern,

$$\begin{tikzcd}
& \boxed{\text{root}} \arrow[dl] \arrow[dr] & \\
\boxed{\begin{array}{c} I(K_1) = 2, \\ E(K_1) = \text{false} \\ U(K_1) = \text{true} \end{array}} \arrow[d] & & \boxed{\begin{array}{c} I(K_7) = 1, \\ E(K_7) = \text{false} \\ U(K_7) = \text{false} \end{array}} \arrow[d] \\
\boxed{\begin{array}{c} I(K_2) = 4, \\ E(K_2) = \text{false} \\ U(K_2) = \text{false} \end{array}} \arrow[d] & & \boxed{\begin{array}{c} I(K_8) = 2, \\ E(K_8) = \text{false} \\ U(K_8) = \text{false} \end{array}} \arrow[d] \\
\boxed{\begin{array}{c} I(K_3) = 5, \\ E(K_3) = \text{false} \\ U(K_3) = \text{false} \end{array}} \arrow[d] \arrow[dr] & & \boxed{\begin{array}{c} I(K_9) = 4, \\ E(K_9) = \text{false} \\ U(K_9) = \text{true} \end{array}} \arrow[d] \\
\boxed{\begin{array}{c} I(K_4) = 6, \\ E(K_4) = \text{true} \\ U(K_4) = \text{true} \end{array}} \arrow[d] & \boxed{\begin{array}{c} I(K_6) = 7, \\ E(K_6) = \text{true} \\ U(K_6) = \text{true} \end{array}} & \boxed{\begin{array}{c} I(K_{10}) = 7, \\ E(K_{10}) = \text{true} \\ U(K_{10}) = \text{true} \end{array}} \\
\boxed{\begin{array}{c} I(K_5) = 7, \\ E(K_5) = \text{true} \\ U(K_5) = \text{true} \end{array}} & &
\end{tikzcd}$$

Fig. 6. LB tree for the example

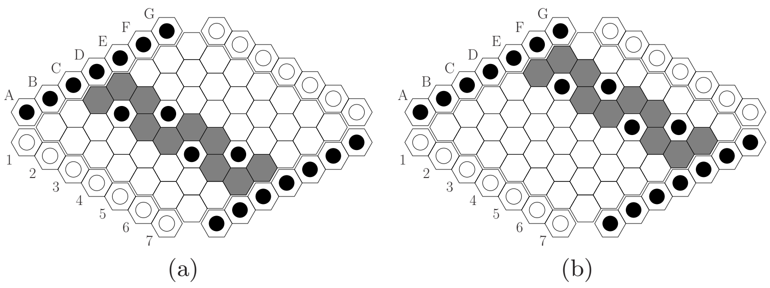


Fig. 7. Hex goal patterns

there are up to $B-1$ translations possible. For each pattern we have one that is obtained by reflection along the middle axis, such that we insert at most $2(B-1)$ entries for each established winning position for black. All symmetric patterns are additionally inserted into the limited branching tree.

6 Experiments

We implemented a Hex-playing program on top of Six. Six applies limited depth alpha-beta game tree search. To deal with the large branching factor in HEX only the k-best successors are kept for each depth. We chose the Fast Artificial Neural Network Library (FANN) [11] as the basis for learning the evaluation function and its subsequent use. The Java Native Interface [15] was used to connect a flexible user interface (written in Java) to the underlying system (written in C).

All experiments were run on a Linux PC with 3 GHz Intel Pentium IV with 512 MB RAM. From the range of different board sizes, we selected two case studies: the 6×6 board, which is small enough to completely explore the search tree, and the 11×11 board that is used for tournament play.

The decision in HEX on a 6×6 board can be make after a few moves. The game is won[4] after three played pieces. Both our HEX-system and Six actually terminate after the third move. For the first move, both systems perform a game-tree search to depth one, where 30 boards are evaluated. In Figure 8 we show a comparison of the evaluation of our system and Six.

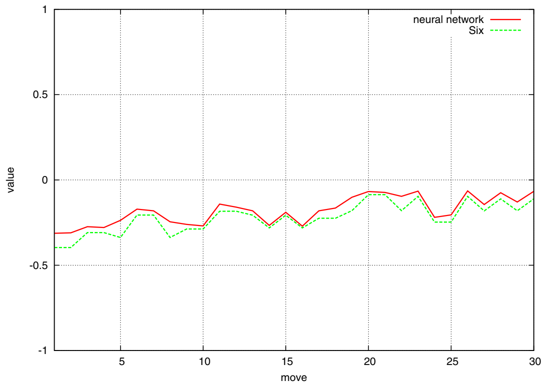


Fig. 8. Comparison between the evaluation function values of our system and Six for a selection of boards

On the one hand, we observe that our system approximates the true value quite good. On the other hand, there is a large difference between the evaluation times. While the evaluation of a board in our system needs less than one millisecond, Six needs up to three seconds and more. In Figure 9 we display the times for evaluating the boards for the first move in Six. The overall time for the first move in our system was 0.57 seconds, while Six required 47.28 seconds.

After the second player has moved, with the third move both systems create a virtual connection between the edges. Our system finds an entry the goal pattern database 0.015 seconds lookup time on the average.

[4] We say a game is won, if there is a virtual connection between the sides of the board.

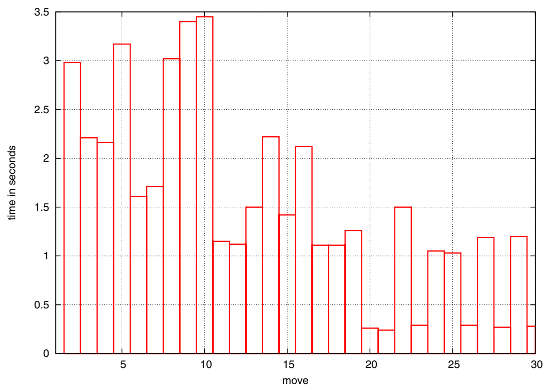


Fig. 9. Distribution on evaluation times for game tree leaves when evaluating the first move in Six

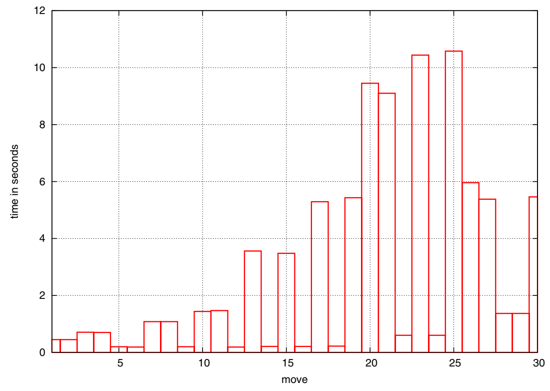


Fig. 10. Distribution on evaluation times for game tree leaves when evaluating the first move on the 11 × 11 board in Six

Our system can classify the initial state as won for the first player: in a search depth of three without any limit the branching factor on the first move, the root of the game tree evaluates to a definite win for the first player. The search time for this proof was 18.2 seconds, while expanding 1,389 nodes.

Similar results have been obtained for the 11 × 11 board. The evaluation time of our system is less than one millisecond, while Six needs sometimes more than 10 seconds for evaluating one of the 30 boards for the first move. The time distribution for evaluating these 30 boards for the first move is illustrated in Figure 10. On the same depth limit our system did not find the best move (placing the piece in the middle of the board [2]), so we allowed a search depth of three. Now the best move was found after 44.26 seconds, which is still faster than the 87.64 seconds that Six needs to calculate the first move with search depth one.

Our system can search at least two levels deeper than Six and, therefore, classify some goal states earlier. The largest amount of time is needed for the lookup in the goal pattern database, with an average of 0.1 seconds.

7 Conclusion

With the combination of a neural network evaluation function and a goal pattern database based on limited branching trees we were able to generate a perfect player for the 6×6 board. For the 11×11 board, a player was generated that is able to choose good moves. Due to the limited range of the first two moves the neural nets computed the evaluation functions precisely without generalization.

We have successfully reduced the time for node expansion and, therefore, increased the search depth. An evaluation on a larger scale (larger number of training example) will produce a much better player. For example, TD-gammon was trained by 1.5 million self-playing games [17].

For a very strong Hex player, the design of a hybrid of Six and our learned system, seems the most plausible avenue for future research. Both systems can concurrently execute game tree search given a fixed time frame. Comparing the evaluations can then exploit the advantage of both approaches. Especially for ending games, our approach searches deeper, which lead to an earlier classification of a board.

References

1. Anshelevich, V.V.: The game of hex: An automatic theorem proving approach to game programming. In: National Conference on Artificial Intelligence (AAAI), pp. 189–194 (2000)
2. Browne, C.: Hex Strategy: Making the Right Connections. A. K. Peters (2000)
3. Gale, D.: The game of hex and the brouwer fixed point theorem. American Mathematical Monthly 86, 818–827 (1979)
4. Gardner, M.: The Scientific American Book of Mathematical Puzzles and Diversions. Simon and Schuster, New York (1959)
5. Gardner, M.: The Second Scientific American Book of Mathematical Puzzles and Diversions. Simon and Schuster, New York (1961)
6. Hoffmann, J., Koehler, J.: A new method to index and query sets. In: International Joint Conference on Artificial Intelligence (IJCAI), pp. 462–467 (1999)
7. Junghanns, A.: Pushing the Limits: New Developments in Single-Agent Search. PhD thesis, University of Alberta (1999)
8. Kahl, K.: Maschinelle Lernverfahren für das Strategiespiel Hex. Diplomarbeit, Universität Dortmund (2007)
9. Litovski, V.B., Zwolinski, M.: VLSI Circuit Simulation and Optimization. Kluwer Academic Publishers, Dordrecht (1996)
10. Nash, J.: Some games and machines for playing them. Technical report, Rand Corporation (1952)

11. Nissen, S.: Implementation of a fast artificial neural network library (FANN). Technical report, Department of Computer Science University of Copenhagen (2003)
12. Reisch, S.: Hex ist PSPACE-vollständig. Acta Informatica 15(2), 167–191 (1981)
13. Schaeffer, J., Björnsson, Y., Burch, N., Kishimoto, A., Müller, M., Lake, R., Lu, P., Sutphen, S.: Solving checkers. In: International Joint Conference on Artificial Intelligence (IJCAI), pp. 292–297 (2005)
14. Shannon, C.E.: Programming a computer for playing chess. Philosophical Magazine 41, 256–275 (1950)
15. Java Native Interface., Javasoft's Native Interface for Java (1997)
16. Sutton, R.S.: Learning to predict by the methods of temporal differences. Machine Learning 3, 9–44 (1988)
17. Tesauro, G.: Practical issues in temporal difference learning. Machine Learning 8, 257–277 (1992)
18. Thrun, S.: Learning to play the game of chess. Advances in Neural Information Processing Systems, vol. 7 (1995)
19. Tischbierek, R.: Nur das Schach hat verloren. Deutsche Schachzeitung 1, 4–14 (2007)
20. Yang, J., Liao, S., Pawlak, M.: On a decomposition method for finding winning strategy in hex game. In: Internat. Conf. Application and Development of Computer Games, pp. 96–111 (2001)