https://doi.org/10.3311/PPee.11952

Periodica Polytechnica Electrical Engineering and Computer Science, 62(4), pp. 155–164, 2018

Development of an Incremental Pattern Extraction Based Gomoku Agent

János Szőts[1*], István Harmati[1]

[1] Department of Control Engineering and Information Technology, Faculty of Electrical Engineering and Informatics, Budapest University of Technology and Economics, H-1117 Budapest, Magyar Tudósok krt. 2, Hungary
[*] Corresponding author, e-mail: kisszots@gmail.com

Received: 11 January 2018, Accepted: 27 August 2018, Published online: 25 October 2018

Abstract

The subject of this paper is an unusual approach to artificial game playing. Our main goal is to replace exhaustive game tree search with incremental pattern extraction and recognition, thus greatly reducing computation time. This is achieved using search with a depth of 3, together with pattern matching and pattern-based heuristic functions, where patterns are learned through play. We examine the efficiency and efficacy of this method regarding the game Gomoku, also known as Five-in-a-row. To evaluate our agent, we implement two basic reference agents and also incorporate a strong open-source AI called "Carbon" into our environment.

Keywords

Gomoku, machine learning, artificial intelligence

1 Introduction

Game intelligence has always been a widely researched area because it is an easily described environment that provides great challenges for artificial intelligence. It allows researchers to develop AI techniques capable of making strategic decisions. The main element of a game playing program, or agent, is usually tree search, most commonly an alpha-beta minimax search [1]. However, many other basic methods have been developed throughout the past decades, such as proof number search, dependency-based search [2], pn2-search, proof set search [3], Monte Carlo tree search [4]. Recently, learning agents receive the most attention. Learning how to play also ranges widely, from using genetic algorithms [5, 6], supervised training of neural networks [7-9] to the most popular reinforcement learning method [9-12]. The most recent advance in this field isthe result of deep convolutional [8, 9, 13], or residual networks [11]. Another interesting aspect of such programs is their ability to learn play completely without human help, that is, with only the rules and field information of the game provided. Google DeepMind's newest AI, AlphaGO Zero achieved this in the game Go [11], which is considered the most challenging board game for artificial intelligence. Even so, research in this field has not stopped; faster, more efficient algorithms are still to find.

Deep tree searches can be very effective, but also very slow. We aim to develop an agent capable of fast, effective learning and playing by exchanging search depth for pattern recognition. We implement a simple, but feasible pattern-based heuristic function and create an algorithm that extracts winning patterns while playing, called pattern backpropagation.

The pattern extraction based agent uses a search method with limited depth and breadth combined with pattern recognition. Limiting search depth is a widely used method [1], while reducing breadth is similar to conducting a policy based search as in [4, 13]. The evaluation function at the leaves and the policy function are given by our proposed heuristic measure. The heuristics are based on the same patterns as the recognition, thus both hard and soft pattern recognition is realized. Our field for testing this approach is Gomoku, which is suitable in several aspects.

Section 2 describes the game Gomoku and why it is suitable for our research. In Section 3 we specify the heuristic function and the reference agents used to evaluate our method. In Section 4 the pattern extraction based agent is introduced, detailed. Section 5 shows test results; conclusion is discussed in Section 6.

Cite this article as: Szőts, J., Harmati, I. "Development of an Incremental Pattern Extraction Based Gomoku Agent", Periodica Polytechnica Electrical Engineering and Computer Science, 62(4), pp. 155–164, 2018. https://doi.org/10.3311/PPee.11952

2 Gomoku

This section is based on [14]. First, we describe the basic rules of Gomoku, then address the issue of the game being solved and explain why it is still a suitable research field. Lastly, we introduce threat methodology, usually used by agents developed to play Gomoku.

In the original Gomoku game, two players take turns placing black and white stones on either a Go board or a similar, 15-by-15 board, on an empty intersection or square, starting with black. The first player to place their stones in five consecutive squares (horizontally, vertically, or diagonally) wins. The stones placed down will stay there until the end of the game. When there are no valid moves left, meaning the board is filled, the game results in a draw. Gomoku is also popular in schools among students, as it can be played on a grid sheet with two pencils, drawing x's and circles representing black and white stones. Freestyle Gomoku is the variant where a winning line does not strictly have to be five squares long, six or more consecutive occupied fields also win. We consider freestyle Gomoku from here on. An example playout is shown in Fig. 1.

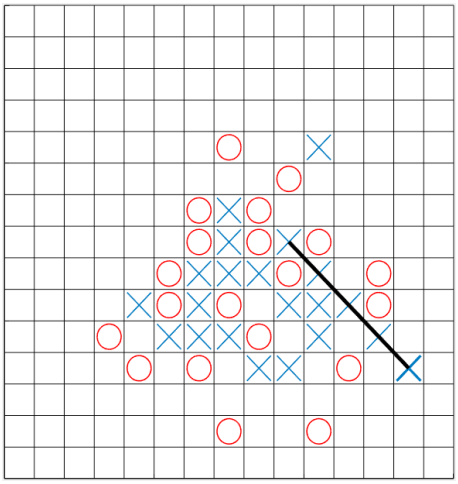

Fig. 1 An example game, where x has won.

It has been known for decades, that through optimal play, the first player always wins. With a proper search method, a database has already been constructed, which contains all the necessary information on how to win [2]. In tournaments, black's advantage is compensated by playing different colors alternately and with opening rules such as "pro" or "swap", or even special restrictions for black, like in the professional version of Gomoku, Renju. However, an agent that plays optimally either requires an unaffordable amount of computation time or great storage capacity. AIs which depend on neither are still up for research.

We chose Gomoku as the environment for our method for several reasons. It is a two-player, zero-sum, non-trivial and well-known game, requiring considerable skill for playing on an expert level. Also, it provides perfect information, is deterministic and a sudden death game, which are crucial in recognizing, and for the existence of winning patterns. Other game properties according to Allis [2] are complexity and convergence, which are less important in our case. An additional aspect is to be considered: our algorithm in its current form only works for monotone games, where a change made to the game field is never reverted.

When aiming to have a deeper understanding of the game Gomoku, one has to be familiar with threats and threat sequences. A threat is a pattern on the board, which, if unaddressed, leads to a win. Threats have descriptive names: a four is an occupied line of four stones, with one open end, a straight four has both ends open, etc. For a more detailed description, see [14]. It is clear that each threat must be blocked immediately. To win the game, one has to create a double threat, which the opponent is unable to counter. This usually happens at the end of a threat sequence, which means a series of turns where an attacking player creates threats in each turn, forcing their opponent on blocking moves. These sequences can be broken when a defensive move also creates a counter-threat.

3 Reference agents

3.1 Simplified threat space evaluation

When developing reference agents for measuring the performance of our pattern-extraction algorithm, we constructed a heuristic evaluation function. This measure is based on and is similar to the threat space approach in some ways, but it is much more general. The key idea behind it is that fours and threes are incomplete fives, and open fours are basically two fours. By specifying how value scales with the incompleteness, we assign values to different patterns relative to the final pattern (five). More complex patterns can also be derived from fives. This way we get a general, simple and feasible method. It also raises the possibility to create a method which works considerably well for different games. This question will be further discussed later.

Following our method, every line of five squares on the board adds the value $m*n$ to each of the five squares when there are n stones of one player and no stones of the other (thus giving five minus n empty squares), where

m is a multiplier. The value of m can be chosen by considering how many weaker patterns should weigh out a stronger one - which contains one more stone of the player. Through playing with a purely heuristic agent, this was tuned to 10, which implies a greedy strategy: the agent prefers fewer, more complete patterns. In Gomoku, it is sensible to account for the opponent's moves similarly, as blocking a potentially good move is also considered advantageous. The easiest way to do so is to simply take the sum of own and enemy values. This way, computing the given multiplications for each line on the board, the values assigned to empty squares represent how beneficial it would be to put a stone there.

The value of an empty square considering the vertical direction, with only own moves can be calculated as

$$V_{vertical,own} = \sum_{i=-4}^{0} \prod_{j=i}^{i+4} F(a+j,b) \tag{1}$$

Where F is a matrix with the size of the board, containing ones for empty squares, zeros for the enemy's stones and a and b represent coordinates. Horizontal and the two diagonal directions and values given by the opponent's moves should be calculated similarly and then added.

We created an agent which only uses this measure to choose its move, which we will refer to as "constant" agent. The pattern extraction based agent uses a generalized version of this measure, where multiplications are not done along straight lines, but along patterns learned through play - practically stored as a vector of coordinates.

3.2 Alpha-beta pruning agent

As a reference, we created an agent that uses classical alpha-beta pruning with two heuristic functions based on the previously described evaluation.

Firstly, the branching factor is greatly reduced by keeping track of possible moves with the biggest potential - given by our heuristic function as an a priori move value -, and only evaluating a fixed number of them. This method also results in a sensible ordering of the move options, which increases the efficiency of alpha-beta search [2]. Secondly, the evaluating function at the leaves is also based on this method.

Running these calculations at every node of every search would be time-consuming, but updating the values can be done incrementally. This means that the value of an empty square only changes when the last move is aligned with it and closer than five squares. These incremental changes are done at each ply of the tree search. It is only possible when a move's potential is symmetrical regarding the players. When selecting sensible moves and thus reducing branching, this is acceptable. However, a move's value is better represented by giving more weight to the patterns of the playing agent. Therefore, the also incrementally changing game value is not modified by the a priori value of the selected move, but the subtraction of the a priori value from a new value calculated with the same method, but with the move in question already made. This new value is only based on the stones of the playing agent, since every multiplication of the opponent would contain at least one zero, the current move; it is also conceivable that this value is m times the player's part of the a priori value.

3.3 Carbon AI

In order to fairly evaluate our agents, we searched for an open-source AI of known skill. Among the contestant programs for the Gomocup, a worldwide Gomoku artificial intelligence tournament [15], "Carbon" was available as source code. In order to incorporate given C++ code in the MATLAB environment, certain changes had to be made: the "main" function was exchanged with "mexFunction" with appropriate in- and output arguments, and some datatypes had to be modified. With help of the MATLAB compiler, we built a MATLAB-specific executable containing the given algorithm.

Carbon AI is a really strong advanced artificial intelligence algorithm using some state of the art methods: special evaluation function, minimax with cut offs, alpha-beta pruning, transposition table, situation signatures, candidate generating, expert knowledge and further enhancements [16]. In the 2017 Gomocup Carbon placed 9. in freestyle and 7. in fast game categories, thus it can be considered one of the strongest algorithms in the field.

4 Pattern extraction based agent

4.1 General method

Approaching Gomoku with the classical search methods is infeasible, as the branching factor is very high. Even with the sensible criterion that only squares with occupied neighbors are considered, it can go up to multiple dozens. Computing possible plays to the ending is evidently impossible, but also applying moderate depth limit would need an unacceptable amount of computing time. Depth limit in searches comes with the need of evaluation functions at leaves, which is at least non-trivial. The easiest way to deal with these issues is using heuristic functions.

We use heuristics both to reduce branching - by selecting candidate moves - and to evaluate leaves. The basic method for these will be described later.

When using heuristic functions, however, we risk that our agent could make wrong moves in states that are clear to a human player. We call states where moves leading to certain win or lose exist, critical states. In some sudden-death games like Gomoku or chess, these states contain certain patterns - an example is shown in Fig 2. By incorporating pattern recognition into the heuristic-based search, we get an algorithm that plays intuitively, effectively, and still acts appropriately in critical situations. Learning of these patterns works simultaneously in the search procedure, through a specially developed pattern extraction algorithm.

Our pattern extraction algorithm is based on the observation that for some – sudden death – games when a final, winning pattern is inevitable in the near future, it can be deduced from the current state only. More precisely, by finding a predecessor pattern on the board. A simple example of this is an open four in Gomoku: the next stone put will form a five and win, regardless of the opponent's move – except if that is already a winning move for the opponent. It is clear, that while an open four is also a winning pattern, we must keep track of the number of moves necessary for the win, as an opposing pattern with fewer moves annihilates a weaker one: it simply leads to an earlier win. We define pattern tiers as follows: a (winning) pattern belongsto tier "n", if and only if it can lead to a win in "n" moves, while no proper defensive moves are played. In Gomoku, this is equivalent to considering the longest consecutive occupied line in the pattern: a five leads to a win in zero moves, a four can lead to a win in one move, two open threes lead to a win in two moves, thus belonging to tier 2. The importance of this notion is explained in Fig. 3: here, a tier 1 pattern found a half-ply deeper outweighs the opponent's current, tier 2 pattern. In other words, the agent can ignore the opponent's threat when it can create a winning pattern earlier.

Higher tier patterns exist, but appear seldom – or rather, are rarely recognized. We note, that there is one obvious tier 4 pattern in Gomoku: an empty board with only one black stone, as theoretically it leads to a certain win. It would not be beneficial to store it though, because it obviously appears, and without knowing proper tier 3 and 2 patterns, our agent wouldn't know optimal play from there.

Another important addition to the classic threat definition is that we also consider large patterns which represent

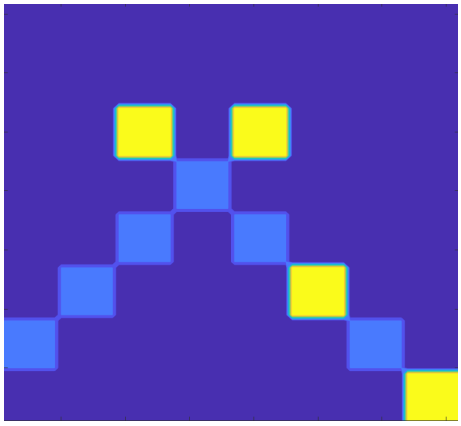

Fig. 2 An example pattern which leads to a win in four moves (two for each player). It is considered a tier 1 pattern.

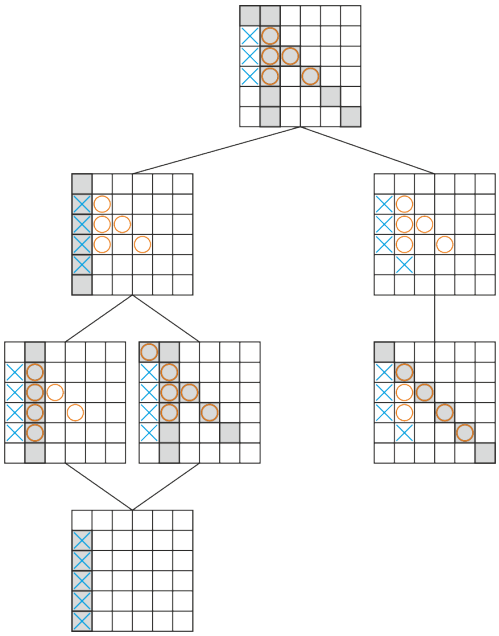

Fig. 3 Importance of tiers: the agent (X) considers pattern tiers to choose an offensive move in presence of an enemy pattern, and wins

threat sequences ending in a double threat, as threats. The example in Fig. 2 is of tier 1 as it contains a four; although in practice, it would usually lead to a win in two moves (of the attacker) because defensive moves would be played.

In our approach, matching patterns add to the heuristic game value, which is computed similarly to that of the alpha-beta agent. Obviously, exact pattern matches are more valuable than the heuristics; this is implemented such that the former values are greater of 3-4 orders. Concretely, heuristic values range around 100-1000 depending on the multiplicator parameter, whereas a winning pattern has the value 1000000. Patterns of tier 1, tier 2, etc. have their value scaled down exponentially with a factor 2. This

means that an open four has the value 500000, two open threes are valued 250000, etc. Basically, these values represent how many turns it takes to win given the patterns when defensive moves are not played. This measure can be further enhanced by scaling pattern values down with size, which causes longer threat sequences to be less favorable, reducing the risk of an unexpected counterattack. We note that the value of the exponential factor alters gameplay only in unhandled situations where one player has multiple independent active threats, which only occurs through suboptimal play. Otherwise, the only purpose of this factor is to separate tiers, for which the value of 2 is feasible.

4.2 Pattern extraction algorithm

The main functionality of the method is pattern-backpropagation. This means that when somewhere in the search tree a known pattern occurs, its value and relevant fields are passed up to the caller node, which then, according to its minimax strategy, either stores or neglects it. Correspondingly to proof number search, when an agent at a specific ply of the search tree discovers a winning pattern at one of the successor states, it will choose it and only store the relevant fields to that one pattern, regardless if there were other, less valuable or even negative valued patterns discovered. However, if all its choices lead to pattern matching for the other player, it has to store the combination of all patterns matched in lower levels, because it's eventual loss is inevitable only if all those patterns are possible. In Gomoku, this means the union of the relevant fields. Such a combined pattern will be on tier $n+1$ – where the most valuable selected sub-pattern is at tier n – and have according value.

Algorithm 1 and 2 describe the pattern extraction algorithm. Algorithm 1 shows the lop-level of the search, move selection and decision on which patterns will be learned. Algorithm 2 describes the recursive search function enhanced with pattern backpropagation. The last lines of code describe a functionality which decides whether lower-level patterns should be considered. If pattern matches found in deeper plies do not outweigh match value at the current level by at least 1.5, they should be neglected. This means that for example, when two patterns of the same tier are found, only the one formed earlier counts. On the other hand, more valuable patterns found in the deeper search have to be considered. Multiplication of propagated values with a parameter between 0 and 1 makes the earlier more valuable of two patterns of the same tier. This parameter is kept close to 1, to not have any other, unwanted effect on pattern weights. To summarize pattern weighing: tiers influence value largely, whereas pattern size and the depth at which they are found have a small effect on it as both only intend to make earlier win preferable. Explanatory flowchart of the algorithm is shown in Fig. 4.

Algorithm 1 Move selection and pattern learning

```
1: procedure SELECTMOVE(board)
2:     options ← best n moves (board)
3:     for i = 1 to n do
4:         [optionValues(i), patternMatches(i)] ←
   patternSearch(options(i))
5:     [best, bestIndex] ← max(optionValues)
6:     if best <= −patternLimit then        ▷ losing
   pattern is inevitable
7:         newPattern.fields                ←
   union of all patternMaches.fields
8:         newPattern.value                 ←
   max of patternMaches:values
9:         learnPattern(newPattern)
10:    if best > patternLimit then          ▷ winning
   pattern found
11:        newPattern ← patternMaches(bestIndex)
12:        learnPattern(newPattern)
13:    return options(bestIndex)
```

Our pattern extraction method is related to conventional reinforcement learning techniques. One can find some similarity with temporal difference learning - TD(0) in particular [17] -, in the sense that the value of a former state is derived from that of a later state. However, there are fundamental differences: in pattern backpropagation, one has to consider all possible (in practice, all potentially chosen) preceding states simultaneously. The extracted value is assigned to the pattern instead of the state and is definite - therefore, no iterative learning is needed.

It is important to state that this pattern extraction algorithm works for Gomoku because the game is monotone, and future patterns are located exactly on the fields of their predecessors. However, we propose a potential method for future research regarding the monotony of feasible games. In some games, future patterns can be deducted from present patterns, although they do not explicitly contain the latter like in Gomoku. Initial patterns may even contain more fields than their successors. If this is the case, relevant fields cannot be determined as easily. Pattern backpropagation must be modified such that for each potentially relevant field in the initial state, their values have

Algorithm 2 Search function

```
 1: procedure PATTERNSEARCH(board, move)
 2:     board ← makeMove(board, move)
 3:     currentMatch
    patternMatch(board, move)
 4:     if match found then
 5:         currentValue
    value of currentMatch
 6:     if currentMatch is a final pattern then
 7:         return currentValue, currentMatch
 8:     if searchDepth > maxDepth then
 9:         value ← heuristicValue(board)
10:     else
11:         options ← best n moves (board)
12:         for i = 1 to n do
13:             [optionValues(i), patternMatches(i)] ←
    patternSearch(options(i))
                ▷ Pattern extraction showed for the agent,
    works similarly for the opponent
14:         [searchValue, maxIndex] ← max(optionValues)
15:         if searchValue > 0 then
16:             newPattern ← patternMatches(maxIndex)
17:         else
18:             newPattern.fields                ←
    union of all patternMaches.fields
19:             newPattern.value                 ←
    max of patternMaches.values
20:     if |searchValue| > |currentValue| then
21:         value ← 0.99 ∗ searchValue + currentValue
22:     else          ▷ search results are neglected
23:         newPattern ← currentMatch
24:         value ← currentValue
25:     return value, newPattern
```

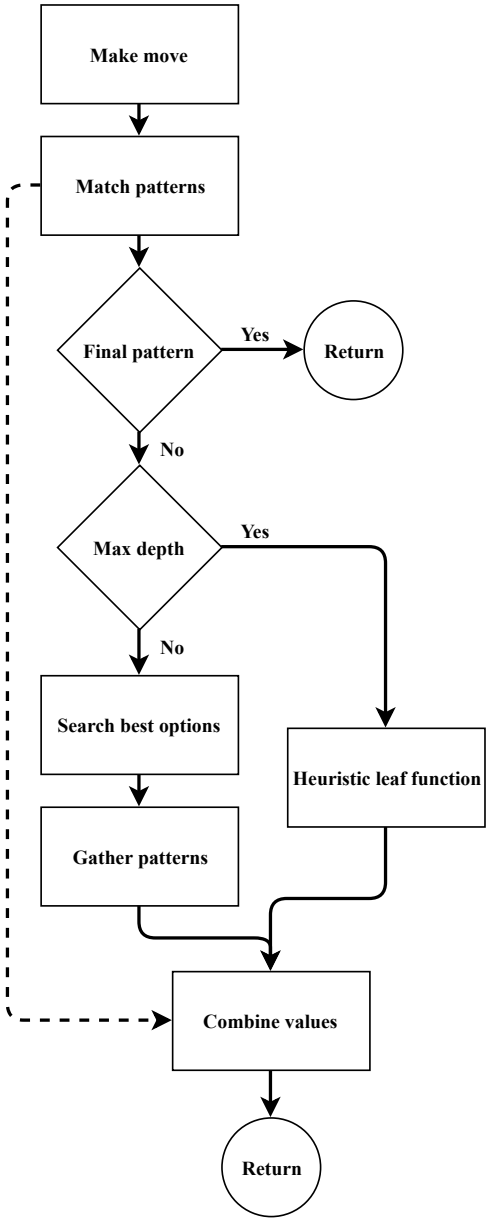

Fig. 4 Flowchart of Algorithm 2.

to be changed one at a time, and for every modified state the same search has to be conducted. This way, when the modification prevents future patterns, the modified field belongs to the relevant set.

4.3 Pattern based agent, heuristic functions, applicability to different games

So far we have introduced patterns and how to acquire them but did not state why. The computational advantage comes from the search depth of our agent: only a 3-ply search is necessary (considering nodes for both players belonging to separate plies). Theoretically, starting with only the final patterns, the agent incrementally learns to recognize patterns which lead to the game's ending in more moves. The 3 plies are sufficient to deduce tier 1 patterns from tier 0 – final – patterns, and inductively, tier $n + 1$ or tier n patterns from tier n patterns. Two plies are enough to detect patterns achieved by the opponent, the third is necessary to extract own future patterns. Also, a tier 1, two-step threat chain can be deducted from a tier 1, one-step threat. Practically, there are too many patterns to store and check all of them, so the database of the agent is limited by maximizing pattern size: this means neglecting long threat sequences. We will further discuss this issue in the conclusion.

Besides checking for exact patterns, "pattern" agent uses the same heuristic branch reduction and evaluation at the

leaves as our alpha-beta agent, but the already mentioned pattern-based generalized version. This way, it can be easily altered by changing the patterns along which the heuristic values are calculated. When using only tier 0 patterns it is the same as the basic heuristic, but there's an option to use higher tier patterns in either the branching or the evaluation heuristic. Our experiments showed that this modification results in less improvement than it is increasing computation time. Still, using patterns instead of straight lines is preferable, as it makes the algorithm more general.

When choosing the maximum branching parameter, we have to consider that the pattern recognition algorithm assumes that every sensible move is evaluated. With the branching factor set too low, false deductions happen, leading to critical errors. This is analogical to neglecting possibly but not probably true children of an "or" node in an and-or tree. Our experiment showed that a branching factor of 5 is already applicable if occasional mistakes are admitted. These mistakes can, however, lead to learning false patterns, which is unacceptable. With the branching factor set to 6, we never experienced this, but we propose a more sophisticated solution. Our planned method is to repeat the search with considerably larger branching factor every time a new pattern is to be learned, thus verifying it. Even in a strictly timed tournament, losing due to violating time limits is preferable to risking an incorrect pattern. This proving search has however not yet been implemented.

4.4 Further details

When using higher tier patterns in the branching heuristic, we used a preliminary pruning, further lowering the branching factor. In practice, this means that if the heuristic value of the next move to be evaluated is lower than half of the previously evaluated move, it can be neglected. Since in this case, the heuristic is more reliable, this lead to a minimal decrease in efficacy, and improved speed. This kind of pruning is only permitted when not in a critical state, meaning the algorithm has not discovered a matching pattern so far.

When the algorithm returns with a potential new pattern similarity check and symmetrical pattern calculations occur. These are highly Gomoku-specific parts of our method. First, we check if there exists a pattern which is equal to or part of the newly found candidate. One winning pattern being a subset of another means in Gomoku that the larger one is redundant since only a part of it already leads to victory. If the candidate is accepted, it is mirrored and rotated multiple times, creating all its symmetrical versions, which are then checked one by one and stored. Storing all symmetrical patterns is not memory efficient, but this way we compute them only once and not at every evaluation. Also, we avoid using symmetrical patterns twice or four times.

5 Evaluation

5.1 Evaluation framework, rules

To evaluate, we implemented our algorithm in MATLAB. Choice of the programming language was based merely on former experience with the environment and greater possible speed of development. MATLAB supports object-orientated programming, which is essentialin programming game playing agents. It also enables the developer to integrate codes written in Fortran, C, C++, which was of great importance in our project.

Our main goal was reducing computation time in playing games. Therefore, when examining the performance of our agents, we set up a rather strict timing rule: each player has got at most five seconds to act in each turn. Time limit for the whole match was set to two minutes. Time limit for our agents was set practically by tuning depth and breadth parameters.

5.2 Evaluating pattern recognition and learning

First, we compared our agent with a "constant" and an "$\alpha - \beta$" agent. The main agent is set to use only top-tier patterns, therefore all three agents are based on the same heuristic measure. Agent “constant” uses no search, only the heuristic function. The second opponent uses a naive $\alpha - \beta$ search with branching reduction, whose depth and branching parameters are tuned for keeping time limits. Playing against this agent represents the dominance of pattern recognition over traditional search.

In both cases, the testing process was following: the pattern extracting agent started with zero known patterns. The opponent played first in every second match. We monitored the cumulative win percentage of our agent and known patterns in each round. Draws were accounted for as 0.5. In theory, our agent learns a new pattern every time it loses and ultimately knows all possible patterns, thus ensuring victory. In practice, this is limited by disabling learning of big patterns to stop the patterns database from growing unlimitedly. Our assumption was that when the majority of allowed patterns are learned win chance becomes constant, meaning that the cumulative win percentage approaches this value. On Figs. 5 and 6 cumulative win percentage can be seen along with the gathering

of patterns. Playing against a weaker opponent results in learning fewer patterns: in 200 matches against "constant", our agent learned 516 patterns as opposed to learning 696 against "$\alpha - \beta$", although there is at most slight improvement after 300. This implies, that when playing against weaker agents, complex patterns - indicating longer threat sequences - are unnecessary. It is also an interesting attribute of the heuristic function that enhancing it with 5-ply deep search does not lead to significant improvement; percentage limits are similar in both cases (84% and 79%). When playing against each other, "$\alpha - \beta$" only beat "constant" in 63 out of 100 games, with 8 draws.

5.3 Play against independent reference agents

Next, we tested the overall efficiency of our agent against the 2017. Gomocup-contestant CarbonAI. Test setting was similar to the previous tests. Since "carbon" is a significantly stronger opponent we expected that the win percentage limit would be reached later, through more rounds. This would be because against a stronger opponent, more complicated patterns are still crucial to know, which would be learned after more matches. Test results of 1000 rounds are shown in Fig. 7.

The results have a surprising tendency: above a certain level, more known patterns do not improve play. Also, learning of patterns became slower than against the simpler agents. The latter can be explained with "carbon's" ability to construct longer threat sequences than our size-limited patterns can handle, causing that from the majority of lost games our agent can not learn. The former is supposedly due to not handling counterattacks (mentioned in Section 2) which break threat sequences. This means that while larger patterns imply smarter planning, they are less sure to result in a win. At such low percentages, discreteness makes the curve rough, which makes it harder to draw conclusions, but the win percentage would supposedly stay between 3 and 4 percent. Our agent won 36 out of 1000 matches during the evaluation. There were no draws.

Additionally, we compared our agent with 104 patterns to some of the other Gomocup contestants. These tests were run manually, with a human operator linking the two different game environments; therefore they were not thorough, serve only as an approximation of the strength of our agent. We will be referencing these testing agents with numbers according to their place in the 2017 Gomocup competition. Our agent scored 2-0 against numbers 37,32,29; 1-1 against 39,33,27,26,21; and 0-2 against 31, 19 and below. Repeated matches were played the same in most cases. This result

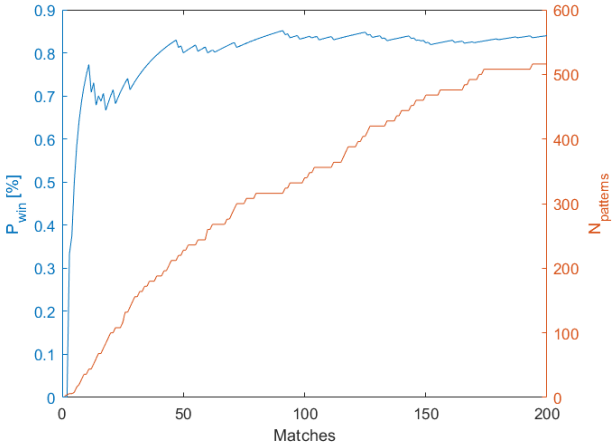

Fig. 5 Cumulative win percentage of the patterns extraction based agent and number of patterns learned against agent "const".

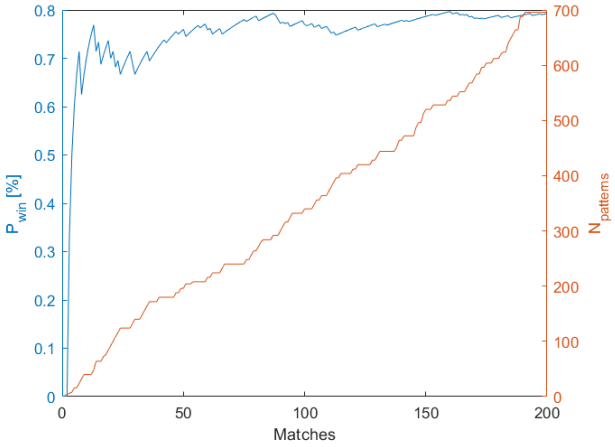

Fig. 6 Cumulative win percentage of the patterns extraction based agent and number of patterns learned against agent "$\alpha - \beta$".

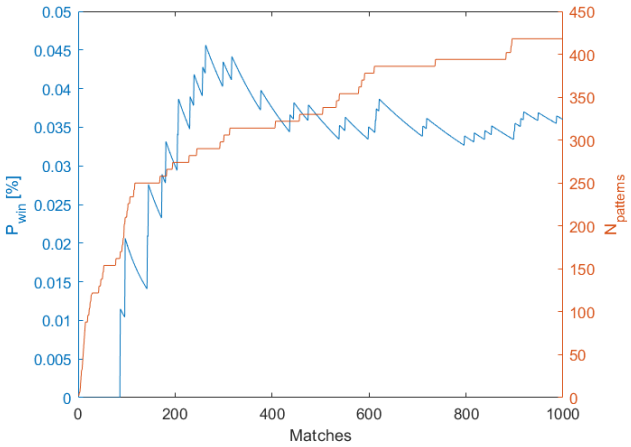

Fig. 7 Cumulative win percentage of the patterns extraction based agent and number of patterns learned against agent "carbon".

makes us believe that if implemented in the appropriate programming language our agent could prove as a considerable contestant in the competition.

We conclude that while the described pattern extraction algorithm works as planned, it succumbs to the methods more sophisticated Gomoku AIs are using - transposition tables for example. This is mostly because their search is considerably deeper, even considering the depth embedded in patterns.

6 Conclusion

In this paper, we presented our pattern extraction based Gomoku agent. The two main contributions of this work are a limitedly general pattern based heuristic and the pattern extraction algorithm. The algorithm works for monotone sudden death games with no topological differences between fields of the game. These differences may also be taken into consideration with proper extensions of the structure of the patterns, but that is not our research goal.

One main disadvantage of our developed method is that the agent has to build a large database of patterns in order to avoid every possible double threat, let alone those at the end of a threat sequence. When playing against a simpler opponent with a constant strategy, the number of occurring patterns is limited and they can be all learnt until the agent is sure to win. In general, however, we would like to create an agent capable of playing against any opponent. For greater efficiency, one would want an algorithm capable of recognizing unseen patterns as combinations of its known patterns, somehow having a deeper understanding of the game. Even for Gomoku, this question leads very far and even if a solution was available, it could probably not be extendedfor other games.

Acknowledgements

The research reported in this paper was supported by the Higher Education Excellence Program of the Ministry of Human Capacities in the frame of Artificial Intelligence research area of Budapest University of Technology and Economics (BME FIKP-MI/SC).

References

[1] Tan, K. L., Tan, C. H., Tan, K. C., Tay, A. "Adaptive game AI for Gomoku", In: 2009 4th International Conference on Autonomous Robots and Agents, Wellington, New Zealand, 2000, pp. 507–512. https://doi.org/10.1109/ICARA.2000.4804026

[2] Allis, L. V. "Searching for Solutions in Games and Articial Intelligence", PhD Thesis, Maastricht University, 1994.

[3] Müller, M. "Proof-set search", In: Computers and Games: Third International Conference, CG 2002, Edmonton, Canada, July 25-27, 2002. Revised Papers, volume 2883, pp. 88–107. Springer, Berlin, Heidelberg, 2003.

[4] Tesauro, G., Galperin, G. R. "On-line Policy Improvement using Monte-Carlo Search", In: Mozer, M. C., Jordan, M. I., Petsche, T. (eds.) Advances in Neural Information Processing Systems 9, MIT Press, 1997, pp. 1068–1074.

[5] Ferrer, G. J., Martin, W. N. "Using genetic programming to evolve board evaluation functions", In: Proceedings of 1995 IEEE International Conference on Evolutionary Computation, Perth, Australia, 1995, pp. 747–752. https://doi.org/10.1109/ICEC.1995.487479

[6] Wang, J., Huang, L. "Evolving Gomoku solver by genetic algorithm", In: 2014 IEEE Workshop on Advanced Research and Technology in Industry Applications (WARTIA), Ottawa, Canada, 2014, pp. 1064–1067. https://doi.org/10.1109/WARTIA.2014.6976460.

[7] Tesauro, G. "Connectionist Learning of Expert Preferences by Comparison Training", In: Touretzky, D. S. (ed.) Advances in Neural Information Processing Systems 1, Morgan-Kaufmann, 1989, pp. 99–106.

[8] Shao, K., Zhao, D., Tang, Z., Zhu, Y. "Move prediction in Gomoku using deep learning", In: 2016 31st Youth Academic Annual Conference of Chinese Association of Automation (YAC), Wuhan, China, 2016, pp. 292–297. https://doi.org/10.1109/YAC.2016.7804906

[9] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., Hassabis, D. "Human-level control through deep reinforcement learning", Nature, 518(7540), pp. 529–533, 2015. https://doi.org/10.1038/nature14236

[10] Freisleben, B. "A neural network that learns to play ve-in-a-row", In: Proceedings 1995 Second New Zealand International Two-Stream Conference on Artificial Neural Networks and Expert Systems, Dunedin, New Zealand, 1995, pp. 87–90. https://doi.org/10.1109/ANNES.1995.499446.

[11] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T., Hassabis, D. "Mastering the game of Go without human knowledge", Nature, 550, pp. 354–359, 2017. https://doi.org/10.1038/nature24270

[12] Vodopivec, T., Šter, B. "Enhancing upper confidence bounds for trees with temporal difference values", In: 2014 IEEE Conference on Computational Intelligence and Games, Dortmund, Germany, 2014, pp. 1–8. https://doi.org/10.1109/CIG.2014.6932895

[13] Silver, D., Huang, A., Maddison, S. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., Hassabis, D. "Mastering the game of Go with deep neural networks and tree search", Nature, 529, pp. 484– 489, 2016. https://doi.org/10.1038/nature16961

[14] Allis, L. V., van den Herik, H. J., Huntjens, M. P. H. "Go-moku and threat-space search", Department of Computer Science, Faculty of General Sciences, Rijksuniversiteit Limburg, Maastricht, The Netherlands, Technical Report CS 93-02, 1993.

[15] Gomocup "Gomocup homepage", [online] Available at: http://gomocup.org/ [Accessed: 10 January 2018].

[16] Czardybon, M. "CarbonAI repository", [online] Available at: https://github.com/gomoku/Carbon-Gomoku [Accessed: 10 January 2018].

[17] Sutton, R. S. "Learning to Predict by the Methods of Temporal Differences", Machine Learning, 3(1), pp. 9–44, 1988. https://doi.org/10.1023/A:1022633531479