DOI 10.34658/jacs.2018.26.2.107-116

JOURNAL OF APPLIED
COMPUTER SCIENCE
Vol. 26 No. 2 (2018), pp. 107-116

The Use of Heuristic Algorithms: A Case Study of a Card Game

Krzysztof Lichy[1], Marcin Mazur[1], Jan Stolarek[1], Piotr Lipiński[1]

[1]*Institute of Information Technology*
Lodz University of Technology
Lodz, Poland
krzysztof.lichy@p.lodz.pl

Abstract. *In this paper we introduce the results of an experiment consisting in the creation of artificial intelligence using the heuristic algorithm Monte-Carlo Tree Search and evaluation of its effectiveness in the card game Thousand.*
Keywords: *Artificial Intelligence, Monte-Carlo Tree Search, heuristics algorithms.*

1. Introduction

There are many methods of programming artificial intelligence in games, depending on the nature of the game and the requirements to be met by artificial intelligence. In board games and card games, artificial intelligence tends to be based on building a tree of potential game states. Board games, in most cases, are games with excellent information, i.e. they hide no information from the player. In card games, meanwhile, opponents' cards are unknown and one can only try and guess how the deck is distributed, which makes these games an example of games with imperfect information. Building trees of potential game states is quite problematic from the point of view of programming. It is necessary to account

for all game-play possibilities, which may require a large amount of memory and processing power due to the large size of the tree. For games with imperfect information, the size of the tree is additionally increased by various configuration possibilities of hidden information. Monte-Carlo Tree Search (MCTS), a heuristic search algorithm, enables the effective searching of large game trees. Supported by UCT (Upper Confidence Bound applied to Trees), it searches the game tree asynchronously, focusing on the more promising part. It performs simulation of the game on each of the created nodes, based on which the node values are determined. It is characteristic in that it can be interrupted at any time by returning the value close to the optimal one. Expanded by heuristics to support optimization, its duration can be further reduced and its efficiency increased.

2. Theoretical background

Artificial intelligence, as we know it today, dates back to the 1940s. It was then when a model of an artificial neuron, resulting from the combination of knowledge in the field of neurobiology and Turing's theory of computation, was proposed for the first time. However, artificial intelligence as such would not be born until the 1950s when Logist Theorist was created, making it the first program capable of mimicking the problem-solving skills of a human being. It proved this by solving most of the theories described in Whitehead and Russell's Principia Mathematica. Logist Theorist was the first to use, or even define, heuristics in computer programming. Newell and Simon, the creators of the program, realized that the search tree would grow exponentially. Therefore, they needed to introduce something that would allow for limiting the number of branches of the tree. The solution turned out to be the introduction of heuristics that rejected paths most likely not leading towards a solution. In 1957, General Problem Solver (GPS) was designed specifically to mimic human problem-solving skills. It quickly turned out that the program solved puzzles (ones that it supported) in a similar way to a human being. The creation of both Logic Theorist and GPS contributed to the widespread use of search algorithms by artificial intelligence programmers in the 1960s. In 1959, breadth-first search (BFS) was developed by Moore with the purpose of finding the shortest path out of a maze. This algorithm is the first representative of brute-force, or uninformed, search algorithms. The use of heuristic search can be traced back to 1958, and more specifically to the early works of Newell and Simon. However, it was not properly defined or used in heuristic functions until the late 1960s.

In 1987, Bruce Abramson, as part of his doctoral thesis, studies the Monte Carlo method developed in 1940, based on random sampling. In his research, Abramson used, among others, methods that were previously used in heuristic search by Newell and Simon, thus greatly improving the performance of search algorithms such as breadth-first and depth-first [1]. In 1992, Brugmann proposed using the concepts of exploration and exploitation based on the function of UCB (Upper Confidence Bound) in the structuring of sampled trees (Monte Carlo). This, in turn, gave rise to the UCT (Upper Confidence Bound for Trees) function. Based on these concepts, Remi Coulom described the use of the Monte Carlo method for game trees and coined the term Monte Carlo Tree Search [2, 3, 4].

3. Heuristic algorithms

Heuristic algorithms sacrifice finding the best solution for faster and more effective problem-solving. These algorithms usually return a solution close to the optimal one in a quick and simple way. They can be used either to find solutions individually or serve as support functions for algorithm optimization.

4. Search algorithms

A characteristic feature of search algorithms is their ability to solve a formulated problem simply by searching for a solution among the possible options. The solution in this case is defined as a sequence of certain actions leading to the expected outcome. Searching for a solution starts from the initial state by forming a search tree, with the root being the initial state. Then, the tree is expanded by subsequent nodes created as a result of applying a certain action to the parent node. Once created the child nodes, one needs to be selected to be checked first, leaving the others that are to be checked for later, which is basically the essence of searching the tree. There are many approaches to choosing a node that is to be checked first, with sequential tree searches being the historically first and most simple among them. They consist in checking the entire tree node by node. In this approach, two sub-approaches can be distinguished: breadth-first and depth-first. Breadth-first search checks the shallowest nodes first and increases tree depth only once all nodes at a given level have been checked. Depth-first search proceeds the other way round, i.e. it creates a sequence of decisions from the root up to the leaf, then moves up the tree looking for a node not yet checked at each level, and once

it has found it, it brings the sequence up to the leaf again. Clearly, these algorithms are not very effective when searching large trees. They take a lot of time and memory, depending on where the solution is found in the tree. However, the tree search process can be improved by using appropriate heuristics, one of the most popular being the A* (A-star) algorithm. It searches the best traversal sequence based on the cost of reaching the goal and reaching individual nodes. The different variants and optimizations of this algorithm find particular application in pathfinding. Unfortunately, it cannot be directly applied to card games because it is impossible to determine or estimate the costs that guide it [5]. Other popular search algorithms are the brilliantly simple Minimax, traditionally used in board games, and Monte Carlo Tree Search [6].

5. Infrastructure for search algorithms

Search algorithms require a certain data structure to enable them to create a tree and track its development. For each node in the tree, there is a structure made up of 4 components:

- State: record of the game state to which a node refers.
- Parent: reference to a node from which a given node was generated. If a given node is also a root, this field is null.
- Children: a list of child nodes of a given node. If a given node is a leaf, this field is null.
- Action: an action that will give a node if called upon the parent object. In the case of a card game, it is playing a card.

Considering these components, it becomes clear how nodes are linked to trees and how easy it is to navigate through them thanks to references [5, 7].

6. Problem formulation

Search algorithms require the right formulation of a problem in order to solve it. For the program to interpret a solution, it must first know what outcome it should expect at the end of the game. Take chess, where one of three states can be expected at the conclusion of the game: win, loss, or tie. The algorithm must know

how to recognize these states so as to react accordingly. In Thousand, such states (excluding tie) can be defined only when a given player robi grę. In the case of a normal game, the only piece of data that allows to determine the value of the node for the final state is the point value of the node [8, 9]. However, interpretation of outcomes is not everything. The algorithm must know how to reach these states in the first place. Depending on the type of game, one needs to properly model the game so that it can be completed and carried out in line with the corresponding rules [10, 11].

7. Thousand game

Thousand is a card game played in Central and Eastern Europe, especially popular in Poland. The game derives from the traditional game of Mariage (Polish "mariasz"), popular in the early 18th century. 2-4 players can play this game and the condition for winning is that the player earns one thousand points. A deck of 24 cards, from 9s to aces, is used in the game. The characteristic feature of this game is "marriages", where each pair of King and Queen of the same suit gives bonus points and gives a trump to the given suit. The detailed description of Thousand goes beyond the scope of this work and can be found in Internet

8. Experiment

Actions of the MCTS algorithm designed to play Thousand will be presented below. Following the confrontation of MCTS with a greedy and a random algorithm, other factors are determined which affect the improvement of the algorithm, in particular the system of awarding the node as well as the exploratory factor. The greedy heuristic algorithm is also compared, and so is the random algorithm in simulating the opponent's moves.

9. Assumptions

The game scenario refers to the setting of the type of players who will participate in the game (MCTS/Greedy/Random). For the MCTS player, this means setting parameters such as the number of simulations, the type of heuristic algorithm used in the simulation, the node value calculation system, and the exploratory factor. Due to the multitude of possible settings for various parameters, the number of

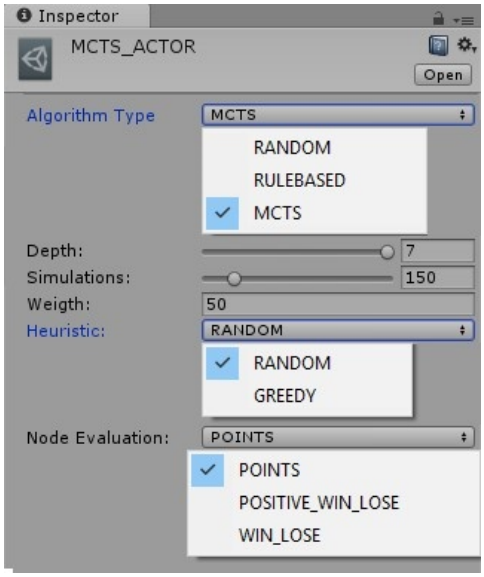

Figure 1: Random Player is a player who does anything randomly, except declare the value of points to be won. This player never raises bets, because then he would only collect negative points.

Table 1: Win ratio for greedy player against random players

	greedy	random 1	random 2
winsy	99.9%	0.0%	0.1%

simulations was set to a fixed value of 150 iterations, experimentally selected during the creation of the program. That number of simulations is considered the sweet spot between the optimality of card selection and the decent simulation length.

10. Greedy algorithm

First, we will show how the algorithm works in comparison to Random Player. In this scenario, Greedy Player plays against two Random Players.

Greedy Player wins almost 100% of the time, on average in 24 rounds. To compare, the average game time for MCTS Player against Random Players is 20.53 rounds. The exploratory factor is determined experimentally. However, it is often

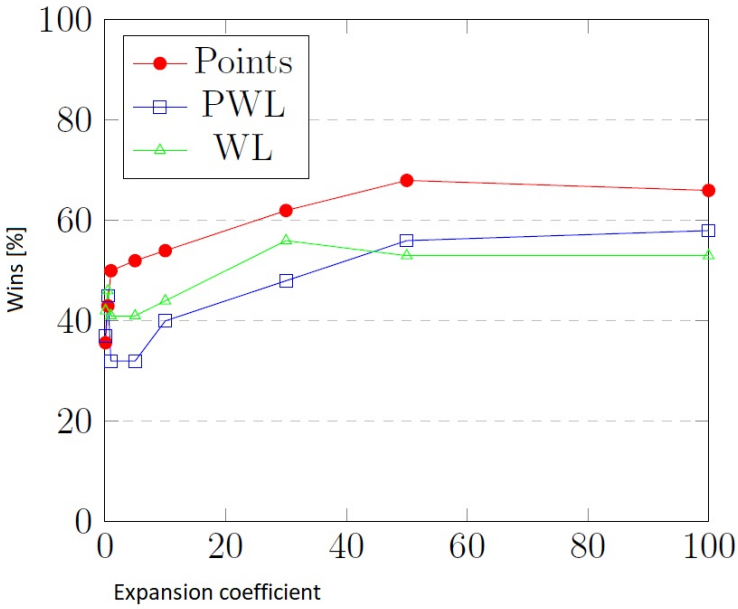

Figure 2: Ratio of wins to exploratory factor in different reward systems.

encountered in the default value 1. This is because in many games one can apply a reward system that assigns values to nodes that do not exceed the range [-1; 1]. In Thousand, two out of three games that are wins depend directly on the number of points scored, where the range can be [0; 360] points. When a player make game, that range can include values from [-360; 360], although one can then successfully make the reward system binary. The player will either accumulate a required number of points or not. This is why determining the right exploratory factor is important in Thousand. To determine the exploratory factor, game simulations are performed for several selected exploratory factors for each of the reward tactics. In these scenarios, MCTS Player plays against two Greedy Players

As seen in Fig. [2], the player who bases the rewarding system on points fared better than the other players. This is because the point-based reward system is more universal than the other systems. Even if make game, we still strive to win as many points as possible. In addition, the remaining systems are binary and point-based reward hybrids, because the player of the win-loss system will not always make game. In the other cases, the player must use a point-based system. The

Table 2: Percentage of wins by mcts players with different algorithms used for simulation

	MCTS greedy	MCTS random
winsy	64.0%	51.0%

point-based system also influences the order of magnitude of the exploratory factor. Better results pertain to values that are expressed in tens. Considering that the value of the node is given in points scored for each simulated game, the exploratory factor would result in exploring only the promising node and omitting the other potentially good nodes. The Monte-Carlo Tree Search algorithm must mimic the opponent's moves during simulation. The program allows two ways to approach this problem. The first is to randomly select a card from the pool of available opponent cards selected by the rules of the game. The second is to use a greedy algorithm, in which the player always chooses a card that gives him the biggest profit or the smallest loss. In Thousand, these are probably the only options. Table [2] shows the percentage of wins in 100 games played by the MCTS algorithm with greedy heuristics and a random algorithm. In both cases, opponents to artificial intelligence were two players using a greedy algorithm to make decisions.

As seen from the results presented above, the player using a greedy algorithm for simulation won more games than MCTS using a random algorithm. However, even with the use of random algorithm, MCTS was able to win the majority of games against players using the greedy algorithm. This shows how big of an advantage predicting game states is in the game.

11. Summary and conclusions

The purpose of this study was to create a competitive artificial intelligence based on heuristic algorithms. We showed that the heuristic search algorithm Monte-Carlo Tree Search works very well in programming games in which we can distinguish individual states described by the same criteria. These can be card games as well as board games. The basic application of the algorithm for games with perfect information can also be used for games with imperfect information, by sampling hidden information, i.e. by determination. The additional use of the greedy algorithm to mimic the opponent's moves in simulations gives more accurate results

and allows to optimize the algorithm in terms of computational complexity. Monte-Carlo Tree Search works as a universal algorithm for card games and board games. All that is required is to define the rules of the game and use appropriate heuristics in predicting the opponent's moves. Programming any tactical behaviours is not necessary because the algorithm itself, although unknowingly, applies these behaviours by analysing the game tree. The algorithm clearly shows the importance of predicting the course of the game in playing this type of games. The Monte-Carlo Tree Search algorithm is very promising, although it has yet to be fully used in the programming of artificial intelligence for the card game Thousand. This game is not particularly complicated, with 24 cards in the deck and "marriages" as the only mechanics that pushes it forward. Still, it is surprising how the same algorithm can be reused for different stages of the game, only slightly modifying what we want it to return. This can also mean modifying the initial output, although the core of the algorithm's operation never changes. Heuristic algorithms are very effective as the right decision-making algorithms, as well as algorithms supporting or optimizing other algorithms. They give satisfactory results at a low computing cost. For MCTS, the user can define how many computational resources he or she will allocate to it. Time constraints are most commonly used. These resources were limited in the experiment by the number of algorithm iterations. This is because the algorithm will be performed the exact specified number of times on any hardware, consequently giving similar results.

References

[1] Russel, S. and Norvig, P., *Artificial Intelligence: A Modern Approach (Third Edition*, Prentice Hall, 2009.

[2] Palma, D. S., *Monte Carlo Tree Search algorithms applied to the card game Scopone*, Ph.D. thesis, Politecnico Di Milano, Italy, 2014.

[3] Puchala, D. and Yatsymirskyy, M., *Fast Neural Networks Learning Techniques For Signal Compression*, Przegląd Elektrotechniczny, Vol. 86, No. 1, 2010, pp. 189–191.

[4] Wawrzonowski, M., Daszuta, M., and Szajerman, D., *Mobile devices' GPUs in cloth dynamics simulation*, In: Proceedings of the 2017 Federated Conference on Computer Science and Information Systems" (FedCSIS), 2017, pp. 1283–1290.

[5] Long, J., Sturtevant, N., and Buro, M., *Search in games with incomplete information: a case study usingBridge card play*, In: Proc. Assoc. Adv. Artif. Intell., 2010, pp. 134–140.

[6] Frank, I. and Basin, D., *Search in games with incomplete information: a case study usingBridge card play*, In: Artificial Intelligence, 1998, pp. 87–123.

[7] Powley, E., Whitehouse, D., and Cowling, P., *Determinization in Monte-Carlo TreeSearch for the card game Dou Di Zhu*, In: Proc. Artif. Intell. Simul. Behav, 2011, pp. 573–580.

[8] Porola, M. and Wojciechowski, A., *Real-Time Hand Pose Estimation Using Classifiers*, In: Computer Vision and Graphics Volume: 7594, 2016, pp. 573–580.

[9] Staniucha, S. and Wojciechowski, A., *Mouth features extraction for emotion classification*, In: Proceedings of the 2016 Federated Conference on Computer Science and Information Systems (Fedcsis), 2016, pp. 573–580.

[10] Napieralski, P. and Kowalczyk, M., *Detection of vertical disparity in three-dimensional visualizations*, Open Physics, Vol. 15, No. 1, 2017.

[11] Napieralski, P. and Kowalczyk, M., *Efficient rendering of caustics with streamed photon mapping*, Bulletin of the Polish Academy of Sciences-Technical Sciences, Vol. 65, No. 3, 2017.