Computational Intelligence in Games

Julian Blank, Frederick Sander

Otto-von-Guericke-University Magdeburg, Germany
julian.blank@st.ovgu.de
frederick.sander@st.ovgu.de

Abstract. Summarize the paper in a paragraph of two. It should contain at least 70 and at most 150 words. You should motivate the research done, give some details about the experiments run and briefly mention the most important findings.

1 Introduction

Give an introduction to the problem tackled. Why is it important? Outline the rest of the document.

2 Literature Review

Perform a literature review: What has been done before in this field? What is the main technique/s used in the paper, and what has it/they been used for in the literature before? Give references to the most relevant work published. Example: [2].

3 Background

There are several basic approaches to write an artificial intelligence (AI) for a game. In this project we were forced to investigate different ideas at three different research areas: heuristic based search, reinforcement learning and nature inspired algorithm. Of course there is an overlap between these approaches. All of them try to find the next best step for the agent by iterating through a search tree. This is build in base of all possible game steps that could done by the agent. One naive approach of iterating through the whole search tree is not possible, because there is a time limitation. Generally this leads to solve the trade-off between iterating similar to a breadth-first or depth-first search. The heuristic approach forces the second by using a estimation function for looking many steps ahead.

3.1 Heuristic Based Search

A heuristic is used to evaluate a game state by putting several facts into one number. When we have to decide which current active branch of a search tree should be iterated this score might help us. One common idea to estimate the distance to the target is the manhatten distance [1].

In a two dimensional space the manhatten distance is calculated by

$$dist(u, v) = |x_1 - x_2| + |y_1 - y_2| \tag{1}$$

adding the absolute value of the difference for the x and the y axis. The input consists always of the points that have one value for each dimension. This could be extended for a n-dimensional space as well. When thinking of a way at a grid this is always a path with one rectangle waypoint (see Figure 1).

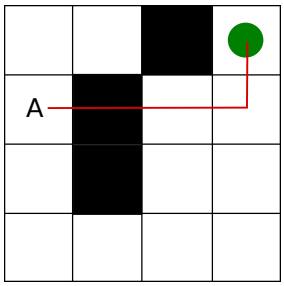

Fig. 1: Manhatten distance

3.1.1 Greedy Greedy-algorithms are a whole class of algorithms and strategies. All of them follow a specific scheme/rule. They are iterative and choose in every step the state with the best reward. The state is in most cases a node which represents the state of the algorithm. The advantage of greedy algorithms is that they are very fast but on the other hand they are not optimal they often only find a local optima and not the global one. The advantage and disadvantage is caused by the greedy approach.

3.1.2 One Step Lookahead One step lookahead is a very simple tree search algorithm which follows the greedy approach. The actual state is the root node. From this node we only look one step ahead to all nodes which are connected by one edge and compute a heuristic value or another kind of reward value for these nodes.

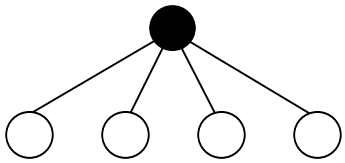

Fig. 2: Search tree for One Step Lookahead

After that the algorithm terminates and we pick the node with the best value.

3.1.3 AStar The A* tree search algorithm is a modification of the dijkstra algorithm and also belongs to the class of greedy algorithms. The Algorithm finds the shortest path between two nodes. In difference to normal greedy algorithms A* is a optimal algorithm, it finds a solution when a solution exists (in this case the shortest path). The algorithm uses a heuristic to estimate the shortest path. The value f(x) of a node N is the sum of its heuristic value h(x) and the costs from the start-node to N g(x).

$$f(x) = h(x) + g(x) \tag{2}$$

A* contains two sets of nodes, the openlist and the closedlist. In every step of the algorithm the Node N with the lowest f(x) value in the openlist is put on the closedlist and all its connected nodes, which are not in the closedlist, are put in the openlist (with reference to their fahter N). If a connected node is already in the closedlist but the new generated value f'(x) is lower than f(x) then f(x) will be replaced by f'(x) and the new father-reference is N. The openlist contains all the nodes to which the path is already known and which can be checked in the next step, the closedlist contains every visited and checked node. When the actual node is the goal-node, the algorithm terminates. To generate the path, the algorithm goes back from the goal-node to the start node (guided by the father-references).

3.2 Reinforcement learning

Reinforcement learning, below RL, is a field in Machine learning which is a section of Artificial Intelligence. RL methods are mostly used by agents in an environment called Markov decision process, below MDP. MDP is a mathematical description of decision processes. They have different states S and some actions A which are available in the actual state. Every timestep the agent chooses an action a and the process switches from state s_a to s_n. The probability to go over form a state S to another state S' by any action A can be described as

$$G : S * S * A \to [0, 1]$$

and the reward given to the agent can be described by this:

$$R : S * S * A \to \mathbb{R}$$

So that

$$(s_a, s_n, a) \to p(s_n | s_a, a)$$

would describe the probability p to go over in the state s_n, given the actual state s_a and the choosen action a and

$$(s_a, s_n, a) \to r$$

shows its corresponding reward.

In differ to other learn methods and approaches like the (semi)supervised learning, RL algorithms never use information which they do not figured out themselves, so no correct samples were given to the algorithm. The only information is the reward given to the agent and some additional information like heuristic values, depending of the specific algorithm. A big problem problem in RL is the conflict between exploration of new and unvisited areas of the solution room and exploitation which is the improvement of already found solutions. ...

3.2.1 Monte Carlo Tree Search Monte Carlo Tree Search, below MCTS, is a class of RL algorithms. It is the most important concept in this paper. MCTS needs a tree of nodes which represent the different states, the edges represent the actions used by the agent to get to this node. The MCTS algorithm traverses to this tree and expands it. To find the global optimum a good balancing ratio between exploration and exploitation is required.

[maybe picture from his paper and pseudocode]

The general MCTS algorithm has four steps, selection, expansion, simulation and backpropagation. In the selection the algorithm starts at freddy working on it

3.2.2 Temporal difference methods freddy working on it

3.2.3 Q-learning freddy working on it

3.3 Nature inspired

The nature is solving problems by applying different approaches instinctively.

3.3.1 Neural nets Our brain solves many problems that could not be solved by algorithms yet. Many researcher tries to explore the process of the human brain. Neural networks tries to model the nervous system and to adapt all the processes [5]. The neurons are modeled as *threshold logic units* (TLU) that consists of several input values x_1 to x_n and one output value y [4].

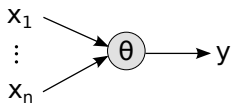

Fig. 3: threshold logic units [4]

For computing the output value is always for every input value one weight w_i and the threshold θ. The formula

$$y = \begin{cases} 1, & \text{if } \sum_{i=1}^{n} w_i x_i \geq \theta, \\ 0, & \text{otherwise.} \end{cases} \tag{3}$$

is used to calculate the output y that is either 0 or 1. Normally the input and the output is given and the weights has to be learned. With only one TLU there only linear separable spaces could be learned perfectly. To solve that problem there is the possibility to create a network of TLU's and map that problem to a higher dimensional space [4].

3.3.2 Evolutionary algorithm An *Evolutionary Algorithm* (EA) tries to use the biological behavior of the population [3].

Algorithm 1 Evolutionary Algorithm [3]

Initialize Population with random candidate solutions;
Evaluate each candidate;
while Termination condition not satisfied **do**
    Select parents;
    Recombine pairs of parents;
    Mutate the resulting offspring;
    Evaluate new candidates;
    Select individuals for the next generation;
end while

3.3.3 Pheromones

4 Techniques Implemented

After explaining the theory of several approaches we now describe our implementations. The proposed algorithm are not always fitting to the principle of the games or has a lot of parameter that has to be adjusted to have good results. We first point out same basic strategies for example staying alive.

4.1 Stay Alive

We implemented two different Stay Alive Agents that has the aim just to act safe and hopefully randomly win the game. Even if there is the possibility to simulation an action, there is all the time an uncertainty. All objects of the games in our competition could - but has not - to act randomly. If we are using the *advance* function that allows us to simulate one step that just one possible state of the future. This implies even if we are not dying at our simulation we could die at the real game.

One approach is to simulate the next action n times. If the agent does not die for this simulations the next action should be safe. The problem is to set a good value for n. If the value is to large not all the possible actions could be tested. Otherwise if it's to small the probability that it's even unsafe grows. From the resulting safe action set there could be randomly choosen the final action.

Suppose there is a game situation like in figure 4 the action *RIGHT* is definitely unsafe what should after an amount of n-action be clear.

Another idea is to analyse the grid around the agent. Assume that each enemy could only move one field at the grid at one game step there is a $5x5$ grid that needs to be analyzed. If an enemy at this grid exists one action is maybe unsafe. If not the enemy won't die for sure.

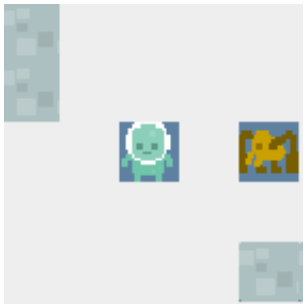

Fig. 4: Advancing safe actions

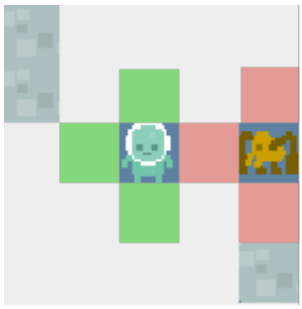

Fig. 5: Grid search for safe actions

At the figure 5 you can see the action that could be excluded without doing any simulation at the game. The possible next fields (excluded the current position if agent does nothing) are marked green. The possible positions of the enemy are red. The action RIGHT is by using the grid search unsafe. At first glance it looks faster and better to only look at the grid, but there might be some problems for unknown games:

- One problem is to classify the game objects at the grid. The grid search is only good if we know that this objects represents an enemy. Otherwise we also classify walls as unsafe.
- Not every enemy could move to all directions. have a look at the figure

4.2 Heuristic based

4.3 MCTS Tree

- act choosig -¿ most visited node
- tree policy
- default policy

4.4 Evolutionary algorithm

The theory of the evolution has to be mapped to the gaming problem. Each candidate of our population is a list of actions that could be executed. To evaluate the fitness of a candidate we are just using the simulation function that is provided by the game. The score

$$s = \sum_{t=0}^{n} (H(s_t) - H(s_{t-1})) \tag{4}$$

is calculated by using the function

$$H(s_i, s_{i-1}) = \begin{cases} 10, & \text{if isWinner} \\ -10, & \text{if isLooser} \\ score(s_i) - score(s_{i-1}), & \text{otherwise.} \end{cases} \tag{5}$$

what we called delta heuristic function.

We implemented two very naive operations on the pool. The crossover is done by select for 50 % the action of the first or the second individual (Figure 6).

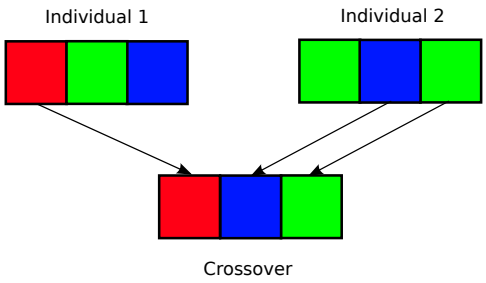

Fig. 6: Crossover of an individual

The mutation is implemented by doing an crossover with an random individual. We fixed for all evaluations runs the mutation probability to 0.7 which implies a crossover probability of 0.3.

We had several problems with the limited time for the evolution. For that we want to have always a good starting pool and use our information from the last game step. All candidates from the last final pool of the last game step are used for the initialization.

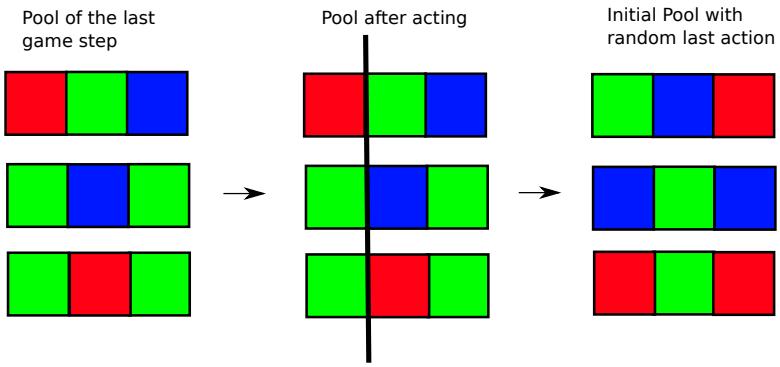

Fig. 7: Sliding Window

For that we use the principle of a sliding window (Figure 7). The first action from the last pool is removed because the agent was acting like this at the last game step. Then to keep the action length fix there is added one random action to the action path. By using that principle we are not throwing our last simulations away.

There is also the problem that for different games the simulation time differs. If we have many game objects that need to be updated and checked for collision, the simulation time increases. But to find a good value for the pool size we need to know the calculation time. As a fix for this we introduced the adaptive path length that reduces or increases the actions of every individual. For that we forced to get always the fourth generation. On the one hand if we stay for example at the initial pool because the eval-

uation takes to long we will reduce the path length. On the other hand if the agents acts like the seventh generation we force to have a long time planing by increasing the path length.

Another problem that we tried to fix is to avoid situations where no individual has a positive score or many has a score of zero. That means each candidate is not gaining points. Our first implementations solved such a tie randomly. The randomness could be disabled by using a heuristic. If we would always use the same heuristic that would not work for all the games because they have different aims. So we used a heuristic switch after n time steps to have also a long time planing strategy.

4.5 Game Detection

Another technique we have implemented is a detection of a known game. We know the 10 games from the test-set and the 10 games from the validation-set, the 10 test-set games are unknown. The Goal of the Gamedetection is to improve the score and the number of wins in the two known game-sets without decreasing the score and the number of wins in the unknown test-set. To do that, the standard parameters of the Algorithm are used when no game is detected, otherwise the optimal parameters for this game are used. These parameters we figured out for some difficult games in which the standard parameters give bad results. To detect a game we generate a String of all Objects (npc, movable, immovable, ect...) and store the Hash value of this String. All the hashes from the known games are stored, in a running Game we generate another String of these Objects and compare the generated hash value to the stored ones.

5 Experimental Study

Detail the experimental setup used to test the different algorithms. Present the results in an understandable manner (graphics, tables, etc.). Draw conclusions about what things worked (and why) and which didn't (and why).

5.1 Among each Approach

5.2 General

6 Conclusions and Future Work

Explain the main contributions of this work: what are the most important findings. Finally, explain how could this work be extended. What would be the next steps?

References

1. P. Barrett, "Euclidean distance: raw, normalized, and doublescaled coefficients," September 2005, pbarrett.net/ [Online; posted 7-January-2015].
2. C. Browne, E. J. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods." *IEEE Trans. Comput. Intellig. and AI in Games*, vol. 4, no. 1, pp. 1–43, 2012. [Online]. Available: http://dblp.uni-trier.de/db/journals/tciaig/tciaig4.html#BrownePWLCRTPSC12
3. A. E. Eiben and J. E. Smith, *Introduction to Evolutionary Computing*. SpringerVerlag, 2003.
4. R. Kruse, C. Borgelt, F. Klawonn, C. Moewes, M. Steinbrecher, and P. Held, *Computational Intelligence - A Methodological Introduction.*, ser. Texts in Computer Science. Springer, 2013.
5. R. Rojas, *Neural Networks: A Systematic Introduction*. New York, NY, USA: Springer-Verlag New York, Inc., 1996.