Computational Intelligence in Games: The General Video Game AI Competition

Julian Blank, Frederick Sander

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

Abstract. A growing section of Artificial Intelligence is Computational Intelligence in Games. Artificial Intelligence methods are applied to computer games in several different ways. In this paper we deal with the General Video Game Playing of computer games where we cannot be sure how the agent is able to score or to win the game at all. Three different approaches that belong to the research areas Heuristic, Reinforcement Learning and Nature Inspired are being considered. A General Video Game Language environment is used to implement our controllers and to simulate a search of possible next game states. The theories behind the different approaches and the modifications to fit to our problem are described. Finally we evaluate the algorithms by using 20 general video games as a test set and compare the average and standard deviation of the winning rate, score and elapsed time.

1 Introduction

Due to the fact that the game industry has been growing over the past years [**?**] constructing good game Artificial Intelligence (AI) is becoming increasingly important. Commonly a game AI is created for one specific game. If humans play against enemies, there are often varying degrees of difficulty that can be adjusted, such as easy, medium and hard. Our aim of the course "Computational Intelligence in Games" [1] at the Otto-von-Guericke-University Magdeburg in Germany [2] has been the evaluation of three different approaches to the General Video Game AI Competition [3]. The game AI is supposed to play unknown games as effectively as possible. The agent can observe the whole grid with game objects that are moving and gets information of all the collisions that are happening. For each gamestep, the agent has 40 ms to return an action. All actions - depending on the game - are LEFT, UP, RIGHT, DOWN, USE and NIL. In order to find the best action, the agent can simulate steps to predict the next state. However, there is uncertainty because the game objects move randomly.

All these facts form a complex problem to solve. To win most of the games as fast as possible, he should play tight. Before each of the games start the aim of the controller is unknown. The agent should therefore try to dynamically find out what the goal of the current game is.

[1] http://is.cs.ovgu.de/Courses/Team+Projects/Computational+Intelligence+in+Games.html
[2] http://www.ovgu.de/
[3] http://www.gvgai.net/

In this paper are going to introduce three general approaches for a game AI. We will first outline our literary review and will then go on to explain the background. The theory could not always be implemented like it was proposed to, because it is necessary for it to fit the requirements of the games. For this reason we will further explain the details of our implementation in chapter four. We will then evaluate the best parameters of each approach an finally reason which agent we consider best.

2 Literature Review

The field of Computational Intelligence in Games (CIG) comprises a very large area of research and there is a lot of work relating to this topic. We are dealing with General Video Game Playing (GVGP) [**?**], which belongs to CIG. An essential work from T. Schaul [**?**] depicts how our game environment is built up. A General Video Game Language (GVGL) is presented, in which games can be described in a text file without the knowledge of game programming. Different objects can be created and rules can be defined deciding what happens if one object collides with another. It was created for the research of AI in games and creating very fast prototypes of General Video Games. The agent who plays the game gets information about the actual state and can chose an available action. In our case, the agent gets a global observation grid and not only a local one (e.g. first-person view).

There are several conferences dealing with the topic of CIG auch as the IEEE Conference on CIG or the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE). The field of CIG contains a lot of different research areas beside GVGP and agent decision making. Yannakakis, G.N. and Togelius, J. pointed out 10 main research areas of CIG: ”NPC behavior learning, search and planning, player modeling, games as AI benchmarks, procedural content generation, computational narrative, believable agents, AI-assisted game design, general game artificial intelligence and AI in commercial games” [**?**]. This shows the variety of its research field. The subject of GVGP often makes use of Monte Carlo Tree Search (MCTS). The comprehensive description of methods [**?**] describes nearly all approaches for GVGP using MCTS. We used this approach to cover the domain of Reinforcement Learning (RL) based algorithms.

3 Background

In this project we were bound to investigate different ideas concerning three different research areas: heuristic based search, RL and nature inspired algorithm. Of course there is an overlap between these approaches. All of them aim 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 be done by the agent. One naive approach of iterating through the whole search tree is not possible because of time limitation. Generally this leads to the funding of the trade-off between iterating similar to a breadth-first or depth-first search. First of all we will introduce a depth-first approach that uses an estimation function enabling it to look many steps ahead.

3.1 Heuristic Based Search

A Heuristic (HR) 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 called the Manhattan distance [**?**].

In a two dimensional space, the Manhattan 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 always consists 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 path on a map this is always a path with one rectangle waypoint (cf. fig. 1).

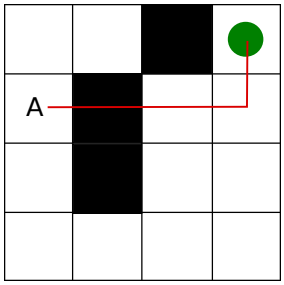

Fig. 1: Manhattan distance

3.1.1 Greedy Greedy-algorithms are an own class of algorithms and strategies. All of them follow a specific scheme/rule. They are iterative and choose the state with the best reward in every step. In most cases, the state is a node representing of the algorithm. The advantage of greedy algorithms is that they are very fast. However, they are not optimal, because they often only find a local optimum instead of the global one. The advantages and disadvantages are caused by the greedy approach.

3.1.2 One Step Lookahead One step lookahead is a very simple tree search algorithm that 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 (cf. fig. 2).

After that the algorithm terminates and we pick the node with the best heuristic value. This algorithm is a special greedy approach that is limited by the first level at the search tree.

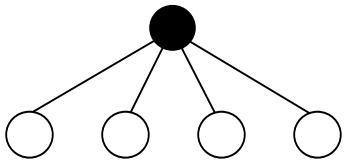

Fig. 2: Search tree for One Step Lookahead

3.1.3 A* The A* tree search algorithm is a modification of the dijkstra algorithm and as well belongs to the class of greedy algorithms. The Algorithm finds the shortest path between two nodes. In contrast to normal greedy algorithms, A* is an optimal algorithm; it finds a solution if 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 father 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

RL is a field in Machine learning which is a section of Artificial Intelligence. RL methods are primarily used by agents in an environment called Markov Decision Process (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 turn a state S to another state S' by any action A can be described as

$$G : S * S * A \to [0, 1] \tag{3}$$

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

$$R : S * S * A \to \mathbb{R} \tag{4}$$

Consequently,

$$(s_a, s_n, a) \to p(s_n | s_a, a) \tag{5}$$

would describe the probability p to turn to the state s_n, given the actual state s_a and the chosen action a and

$$(s_a, s_n, a) \rightarrow r \tag{6}$$

shows its corresponding reward.

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

3.2.1 Monte Carlo Tree Search MCTS is a class of RL algorithms. It is the most important concept considered 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.

The general MCTS algorithm has four steps: selection, expansion, simulation and backpropagation. In the selection the algorithm starts at the root node and traverses down the tree. Goal of this step is to select a node to expand (to generate a child node). Depending on the number of children of every node there are several different paths that can be chosen. Often the $UpperConfidenceBoundforTrees$ (UCT) is used to balance the ratio between exploitation and exploration:

$$UCT = \overline{X}_j + 2 * C * \sqrt{\frac{2 \ln n}{n_j}} \tag{7}$$

$\overline{X}_j$ is the average reward of this node hence the left part of the formula is the exploitation part. The right part generates the value for exploration, where C is a constant (often $\sqrt{2}$), n is the number of times the parent node has been visited and n_j is the number of times the actual node has been visited.This formula has shown to provide good results and it is part of a lot of MCTS algorithms. After the selection of a node one randomly chosen child is generated; this is called the expansion. The third step is the simulation in which we want to know how good the extended node is. In order to do that, we generate children from the expanded node depending on randomly chosen actions. When we reach our simulation depth, we compute the reward of the last simulated node. In the last step, the backpropagation, we start at the expanded node and iterate to the root, guided by the father references. In every node that is visited, the reward of the simulation will be charged with the actual reward of the node. The first two steps are guided by the so called tree policy, the simulation by the default policy.

3.3 Nature inspired

Nature is solving problems by applying different approaches instinctively. Computer scientists used that observed knowledge from nature to write Nature Inspired (NI)

algorithms which follow a similar procedure. Our brain can solve problems that could not be solved by an algorithm until now. Funnily this is often the truth for games such as poker.

3.3.1 Neural nets Many researcher try to explore the process of the human brain. Neural networks try to model the nervous system and to adapt all its processes [**?**]. Neurons are modeled as Threshold Logic Unit (TLU) consisting of several input values x_1 to x_n and one output value y [**?**].

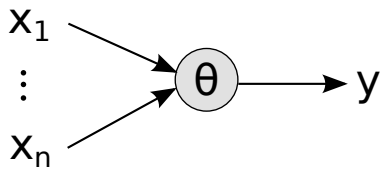

Fig. 3: TLU [**?**]

For computing the output value always the input value, its corresponding weight w_i and the threshold θ is considered. The formula

$$y = \begin{cases} 1, & \text{if } \sum_{i=1}^{n} w_i x_i \geq \theta, \\ 0, & \text{otherwise.} \end{cases} \tag{8}$$

is used to calculate the output y which is either 0 or 1. Normally the weights are learned trough a given input and output. With just one TLU, only linear separable spaces can be learned perfectly. To solve this problem there is the possibility of creating a network of TLUs and thus to map the problem to a higher dimensional space [**?**].

3.3.2 Evolutionary algorithm An Evolutionary Algorithm (EA) tries to utilize the biological behavior of a population [**?**]. These algorithms follow two main ideas. Firstly there are operators like recombination (crossover) or mutation that allow to create different individuals. Secondly every iteration a selection is performed guaranteeing good quality. The procedure is shown in algorithm 1.

First of all there is the initialization phase in which a pool of random individuals is created. After that the score of each of them has to be evaluated. Often it is a problem for optimization problems to evaluate this score quick. This duration is strictly linked to the duration of the whole algorithm.

The while loop needs to have a termination condition. On the one hand this could be a predefined score that the best individual of a generation should possesses. On the other hand it could be a time limit that should not be exceeded. Every iteration starts with a selection of the parents. New individuals are created through the processes of crossover and mutation. The challenge is finding good functions for these operations. Since the pool size is limited, there is a selection of the fittest in every iteration.

Algorithm 1 Evolutionary Algorithm [**?**]

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

Often evolutionary algorithms are used for optimization problems because they are hoped to be more efficient than random search.

4 Techniques Implemented

After explaining the theory of several approaches we will now describe our implementations. The proposed algorithms are not always fitting to the principle of the games or have a lot of parameters that have to be adjusted in order to have good results. We will first point out some basic strategies such as staying alive.

4.1 Stay Alive

We implemented two different Stay Alive Agents that only have the aim of acting save and are hoped to randomly win the game. Even if there is the possibility to simulate an action, there is a permanent uncertainty. All objects of the games in our competition can - but do not have to - act randomly. If we are using the *advance* function that allows us to simulate one step, the result is just one possible state of the future. This implies that even if we are not dying in our simulation we could still die in the real game.

One approach is to simulate the next action n times. If the agent does not die during this simulations, the next action should be safe. The challenge is to set a good value for n. If the value is to large, not all the possible actions can be tested. If it is to small, the probability that the action is unsafe grows. From the resulting set of safe actions, the final action can be randomly chosen.

Assuming that there is a game situation like in fig. 4, the action *RIGHT* is definitely unsafe, a fact that should be clear after an amount of n-actions.

Another idea is to analyse the grid around the agent. Each enemy could only move one field at the grid in one game step, there is a $5x5$ grid that needs to be analyzed. If an enemy at this grid exists, one action might be unsafe. If not, the avatar will not die.

In fig. 5 you can see the action that can be excluded without doing any simulation at the game. The possible next fields (excluding the agent's position if he is not moving) are marked green. The possible positions of the enemy are red. The action RIGHT by using the grid search is unsafe. At first glance it looks faster and better to only look at the grid, but there might be some problems for unknown games:

Fig. 4: Advancing safe actions

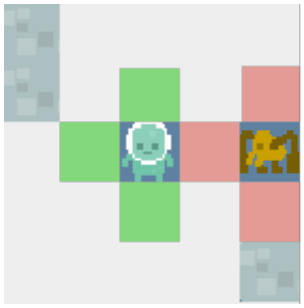

Fig. 5: Grid search for safe actions

One problem lies in classifying the game objects at the grid. The grid search is only successful if we can be certain that this object represents an enemy. Otherwise we also classify walls as unsafe. Furthermore not every enemy can move in all directions but as a agent we would need a special observer to know that. Staying alive is just a basic strategy which is extended with some other approaches.

4.2 Heuristic based

If the action is safe but randomly chosen, the algorithm is only a little bit better than random search. Therefore a heuristic is used to evaluate one state and get a score. Knowing the game in advance would be a substantial advantage. Whenever this is not the case, the results might turn out completely different. One heuristic could be very good, for example in a case in which the agent could kill an enemy. However, if in another game the enemy will kill us, the heuristic will let us commit suicide.

To fix that, we have two different ideas. On the one hand we could implement a dynamic heuristic that is changing over time and is learning from the environment. On the other hand there could be a third instance called *Explorer* that is learning from the environment and creating a knowledge-base. To further use this idea in other approaches, we focussed on the following idea.

During the construction time the agent only explores the environment. First of all he creates a list including all the interesting targets that exist in the level. He sends pheromones into the grid targeting all of them and simulates until the agent dies. For all the simulation there is a global class - the *Simulator* - that automatically makes inferences from the observations. The knowledge base is the environment class that contains information about blocking, scoring, winning and losing game objects (sprites). This two classes follows the singleton pattern and can be created by using the factory (cf. fig. 6). Additional information that can be seen in the figure is the game detection that will be explained later and the field tracker that keeps information about all fields at the grid and how often the agent has visited each one.

When the explorer finds the winning or scoring objects, a heuristic is initialized. We used a heuristic that calculates the distance between two targets by using the Manhattan distance. To reach this target we use an A* algorithm with three modifications.

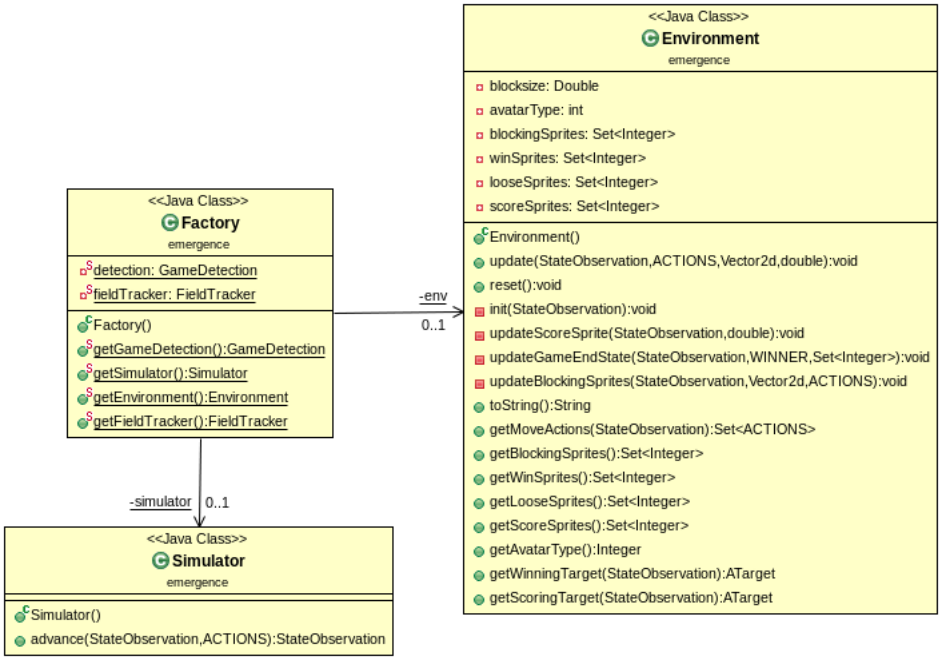

Fig. 6: Class diagram of the simulator and environment structure

Every game step the A* algorithm is executed from scratch even if the target was found in the last iteration. We use a safety approach for the first action to ensure that the next action we my choose is safe. For that we use the principle of the stay-alive agent. The A* algorithm has to keep all the states in memory. This leads to a very fast growing open list with a multitude of states. We introduce a *maxStates* variable. Whenever the open list of the A* algorithm is larger than this variable, the list is cut in a half. For our simulations, we additionally use the simulator as well as the knowledge base. Normally all children of a state are added to the open list. However, we only add those children to the list of whom we assume we will get a new position. Since we know from the Environment class which objects are blocking, we can use that information for a smarter iteration.

4.3 MCTS

The theory of MCTS is described in section 3. The standard approach was modified to fit our issue. The whole MCTS algorithm is computed in the class *MCTSStrategy* which is used by the *MCTSAgent* and the *MCTSHeuristicAgent*.

Our **tree policy** iterates trough the tree, starting at the root node. When a node is not fully expanded, a randomly chosen child node is generated. We tried out some other

ways to choose the children, but random selection worked best. If we reach a fully expanded node the tree policy uses a modified Upper Confidence Bound (UCT) formula

$$UCT = \frac{Q_c}{V_c + \epsilon \cdot r} + \sqrt{\frac{\ln (V_n + 1)}{V_c}} \tag{9}$$

to chose the children node, where *n* is the node and *c* the actual children. *Q* is the reward of a node and *V* the number of visits, more precisely the number of times the *tree policy* had chosen this node. The exploitation term is computed by dividing the reward of the child by the number of visits and a random factor to avoid divisions by zero. This term is big for nodes whose reward is high on average. When the children number of visits are small in comparison to the others, the second term (exploration) will be big. We tried out some other variants and factors, but this solution has produced best results. It is very close to the original UCT formula in section 3.

The **default policy** tries to figure out how good the given node from the tree policy is. To do that, a simulation with the correspond state observation and random actions needs to be done. It is repeated until the hypothetical level of the simulated node reaches the before defined maximum (maximal depth of the tree). At the beginning, when the tree is small, more simulations are executed. Later, when we have already grown our tree, only a few simulations are executed.

When the simulation is finished, a reward is generated from the last state observation by the delta heuristic which will be described later. The win, loss and score are being considered. The **backpropagation** iterates from the node which was expanded by the tree policy to the root, guided by the father references of every node. To every visited node the reward is added and weighted with a before specified value.

The approach of a **rolling horizon** has been modified and applied to our problem. The goal of this rolling horizon technique is to save computing power. When the act method from the *MCTSAgent* or *MCTSHeuristicAgent* is called, a new *MCTSStrategy* (search tree) is constructed or rolling horizon is executed. The new search tree contains only the root node and has to be built up normally; this costs a lot of computing power and time. The rolling horizon does not build a whole new tree, but it uses the information from the previous call of the act method (from the last gametick). One subtree of the previous root node will be the new search tree; it is chosen by checking the last action that was executed and takes the corresponding child as the new root node. After that the levels of the nodes are updated and the tree is built up normally. This saves some computing power and the according tree is larger than the normal one which yields more checked actions and better results.

We used the **open loop** approach, and in contrast to the closedloop approach, the state observation is not stored in the actual node. We only store a sequence of actions in every node which leads from the actual state observation (which represents the root node) to the appropriate node. The tree policy does not need the state observation to compute the UCT value. Before the default policy is able to generate a random path, the state observation of the expanded node has to be generated. All actions in the path of the node are applied to the actual state observation (from the root node) leading a state observation that represents the expanded node. After that, the random path is simulated and the backpropagation updates the reward values.

When no more time is available the MCTS algorithm stops and the agent calls the act method from *MCTSStrategy* to choose the action which will be executed given the MCTS search tree. The child node from the root which was **most visited** is taken, some other approaches that take the node with the highest reward tend to produce only poor results. The *MostVisitedNodeComparator* is used to compare the number of visits. When there are more nodes with equal (and highest) number of visits, the heuristic value is used to select a node. With no heuristic given, a random node from the list of the most visited nodes is selected. The depending action is executed by the agent.

4.4 Evolutionary algorithm

The theory of 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 using the simulation function that is provided by the game. The score

$$s = \sum_{t=0}^{n} (H(s_t) - H(s_{t-1})) \tag{10}$$

is calculated by using the function

$$H(s_i, s_{i-1}) = \begin{cases} 10, & \text{if isWinner} \\ -10, & \text{if isLooser} \\ score(s_i) - score(s_{i-1}), & \text{otherwise.} \end{cases} \tag{11}$$

that we called delta heuristic function.

We implemented two very naive operations on the pool. The crossover is done by selecting the action of the first or the second individual by 50 % (cf. fig. 7).

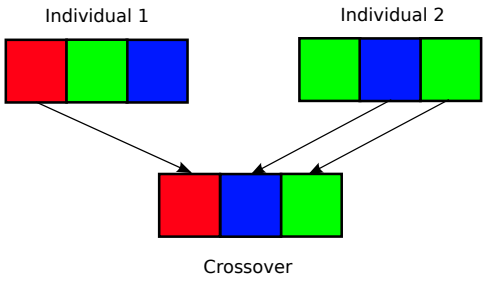

Fig. 7: Crossover of an individual

The mutation is implemented by doing a crossover with a random individual. For all of our evaluation processes, the mutation probability was set to 0.7, thus maintaining a crossover probability of 0.3.

We had several problems with the limited time for the evolution. For the evolution we want to have a good starting pool and want to reuse our information from the last

game step. All candidates from the last final pool of the last game step have been used for the initialization.

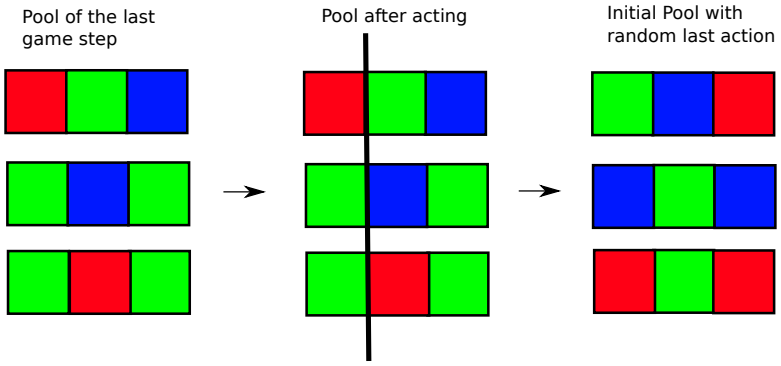

Fig. 8: Sliding Window

For that we use the principle of a sliding window (cf. fig. 8). The first action from the last pool is removed because the agent was acting like this at the last game step. To keep the action length fix, one random action is added to the action path. By using that principle we are not throwing our last simulations away.

Another problem that arises are the differing simulation times for the different games. If we have many game objects that need to be updated and checked for collision, the simulation time increases. However 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 must always get the fourth generation. On the one hand, staying at the initial pool because the evaluation takes to long, will reduce the path length. On the other hand if the agents acts like the seventh generation we are bound to have a long period of planning by increasing the path length.

Another problem that we tried to fix is the occurrence of situations where no individual has a positive score or many have a score of zero. This means that no candidate is gaining points. Our first implementations solved such circumstances randomly. The randomness could be disabled by using a heuristic. Always using the same heuristic would not work for all the games because they have different aims. This is why we used a heuristic switch after n time steps to have a long time planning strategy.

4.5 Game Detection

Another technique we have implemented is the 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 were 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

The approaches are not easy to compare at all. As a fair guideline each approach should have the same knowledge of the future e.g. the same look-ahead depth. But since the approaches uses completely different strategies this is not possible. For example our EA algorithm could have a dynamic path length what means the look-ahead depth is changing over time. This idea does not make sense for the MCTS approach because there it needs to be fixed. Otherwise we have different *Evaluation State Functions* because our HR approach only looks for the distance to an object. By contrast the other approaches tries to figure out if this chain of action is gaining points or not. Because of that problem we decided to compare each approach themselves with different parameter and evaluate after that the best candidate of each algorithm. We are fixing for all the algorithm the time limitation which is in our opinion enough a fair comparison.

To test how well the implemented algorithms work, we executed the games multiple times. We have the 20 games from the trainings- and validation set to test the algorithms. We have no access to the original test set. First of all we want to compare the 3 different controllers with each other, but the several different parameters we can change in the controllers are of significance as well. To test an agent we ran 1000 Games; one game 50 times and one single level 10 times. The following machine was used to run all of the simulations:

CPU	4x Intel(R) Core(TM) i5-4210U CPU @ 1.70Ghz
Memory	8 GB DDR3 L
Operating System	Ubuntu 14.04.1 LTS
Java Version	1.7.0_65

Table 1: experiment setup

Only this machine was used, because with the same agent and parameters we achieved very different results on different machines. The computational power from the CPU was decisive for our choice. The reason for the different results is that the agent every gametick, only has 40 ms to return an action. The results from a powerful CPU were better and more stable (the derivation was lower) than the results from a worse CPU. To evaluate the results, we stored them in csv files and generated tables.

For every evaluation there is a table with all the parameter settings. Additionally there are the average winning rate and the standard deviation listed. All the results are

also visualized by boxplots. There you can see the minimum and maximum average wins overall iterations. Furthermore the blue rectangle shows the qartile values. The red line in the middle is the median and the blue dot is the mean (average wins). It should be taken into account that quartiles and the standard derivation are completely different things and should not be confused. In the following sections each of the approaches will be evaluated.

5.1 Heuristic based Algorithm

The heuristic based algorithm has several parameters that change the behaviour of the agent (cf. table 2). Firstly we changed the maximal states field that provides a limit for saved states at the A* algorithm. It can be seen that the average winning rate increases by changing to a higher maximal states limit of 20. However, setting the variable to 25 lead to worse results. It might be that if this value is to low, the target is not found by the A* algorithm. Moreover if it is too large, to many states are saved and we encounter troubles with the memory or the garbage collector.

ID	Max. States	Safety Strategy	Safety Iterations	Avg Wins	Std of Wins
1	5	SafetyAdvance	5	0.517	0.026
2	15	SafetyAdvance	5	0.529	0.020
3	**20**	**SafetyAdvance**	**5**	**0.532**	**0.027**
4	25	SafetyAdvance	5	0.521	0.019
5	20	SafetyGridSearch	-	0.498	0.024
6	20	SafetyIntelligent	5	0.527	0.018

Table 2: results of the HR algorithms

Afterwards we tried to change the safety strategy to the grid and the intelligent search. Both results are worse than the SafetyAdvance Strategy. This could result from the high dependence of the SafeGridSearch on the Explorer and the classification. With the SafetyIntelligent we got no further improvement but it proved nearly as good as the SafetyAdvance approach.

The boxplot (cf. fig. 9) visualizes the quartiles. The best parameter setup also has the largest quartile range. When looking at the table it also has the largest standard deviation (which do not have to be the same). If the target is not found the agent is staying alive. The risk of a longer search to reach a simulation state that collides with the target could be the reason for the higher variance of the results.

5.2 MCTS

For the test of the MCTS algorithm we modified three different parameters. First of all we tried to figure out if the rolling horizon idea makes sense. After that we changed the gamma variable that is used as a *discounting factor*. Since there were no further

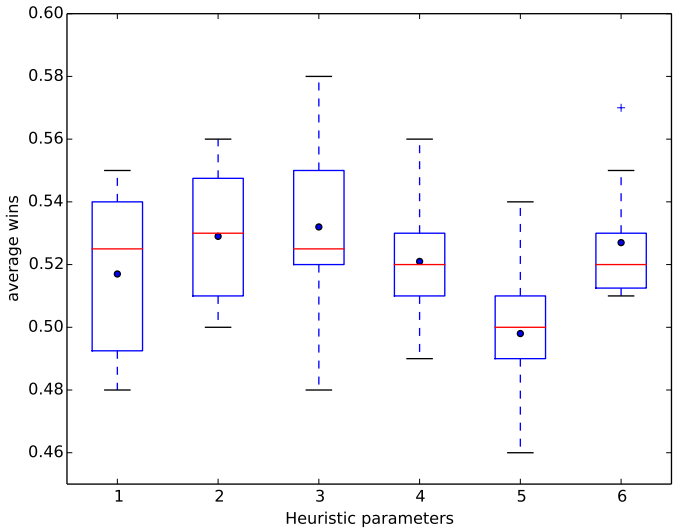

Fig. 9: boxplot of the heuristic based algorithms

improvements, the discounting was disabled by setting it to 1. As a third and very important variable we experimented with the maximal tree depth. Due to the fact that we used an open loop approach this determines the size of the tree with recommended actions.

We achieved the best result by using an agent with rolling horizon, a maximal tree depth of 7 and no discounted reward (cf. table 3). A higher maximal tree depth sticks together with a depth first search. Therefore the winning rate becomes lower at some value. If the tree is too small (e.g. hight of one), it really is a breadth-first search.

We also tried to combine several approaches such as using the heuristic explorer for a better MCTS iteration. But that idea is difficult to implement because this it is a trade off of exploration, exploitation and heuristic search needed. Since that is not as easy to handle, a first naive ideas brought worse results; we were not able to evaluate the combination of these two approaches. It might be a future work to try and implement these.

5.3 Evolutionary Algorithm

For the EA algorithm many parameters exists that can be modified. In order to find a good setup, we created an evolutionary algorithm over that parameter of the EA algorithm. This meta-evolution uses the winning rate as a score function and varies the different parameters. Since we need more computational power to achieve really good results, we were not able to use that idea. Instead we defined the different agents (cf. table 4).

As a first parameter we modified the *SafetyIterations* which ensure that the agent do not choose an action that causes death. Furthermore the path length of each individual

ID	rolling Horizon	maximal TreeDepth	gamma	Avg Wins	Std of Wins
1	0	10	1	0.383	0.033
2	1	10	0.9	0.401	0.026
3	1	3	1	0.416	0.014
4	1	5	1	0.402	0.030
5	**1**	**7**	**1**	**0.422**	**0.033**
6	1	10	1	0.412	0.025
7	1	15	1	0.394	0.014
8	1	20	1	0.395	0.025
9	1	25	1	0.384	0.026

Table 3: MCTS result

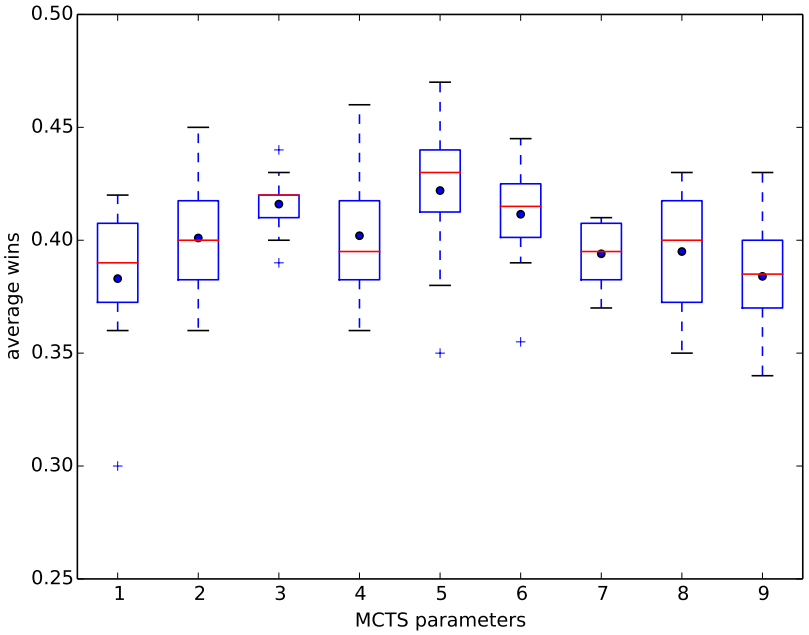

Fig. 10: boxplot of the MCTS algorithm

and the size of the pool were changed. Since not all candidates are shifted to the next generation we introduced a variable that manages that quantity. At least we evaluated if our dynamic path length what ensures the achievement of a specific generation makes sense.

ID	Safety Iterations	Path Length	Size of Pop.	Fittest Pop.	Dyn. Path Length	Min. Gen.	Avg Wins	Std of Wins
1	2	6	14	5	no	-	0.459	0.023
2	4	6	10	4	no	-	0.458	0.024
3	5	4	14	5	no	-	0.466	0.028
4	5	6	10	5	no	-	0.459	0.032
5	5	6	13	4	yes	5	0.451	0.041
6	5	6	14	3	no	-	0.471	0.018
7	5	6	14	5	no	-	0.461	0.025
8	**5**	**6**	**14**	**5**	**yes**	**2**	**0.479**	0.021
9	5	6	14	5	yes	4	0.463	0.027
10	5	6	14	5	yes	6	0.449	0.041
11	5	6	14	7	no	-	0.474	0.028
12	5	6	18	5	no	-	0.462	0.035
13	5	8	14	5	no	-	0.453	0.029
14	8	6	14	5	no	-	0.477	0.022

Table 4: results of the EA algorithms

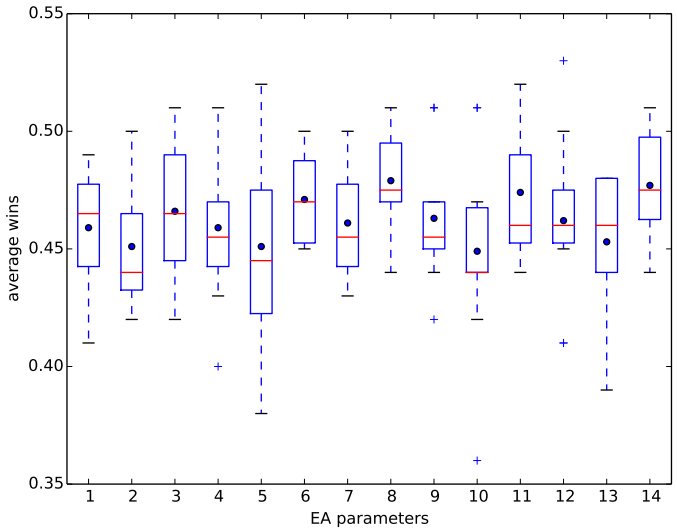

Fig. 11: boxplot of the EA algorithms

As you can see (cf. fig. 11), the EA approach has a higher interquartile range than the others. We assume that this is caused by the idea of EAs concerning the high importance of randomness. We generally reached the best results with the parameters of *ID* 8 (cf. table 4). Since another agent 14 with no dynamic path length reached a very good result as well, this idea might not help for winning the games. This is assured by the best one that only has a min generation value of 2, a value that should normally be the case for all agents. Otherwise it is only a random approach.

5.4 Evaluation of all approaches

Finally we will compare the best parameter setups of the three approaches. For that we increased our experiment to 30 iterations which means 3000 games, each game with 150 runs and each single level 30 times. We defined the best as the agent with the highest average winning rate. The selected agents are printed bold at each of the result tables. For a complete comparison, the average score and the played time steps are evaluated.

approach	Avg Wins	Std Wins	Avg Score	Std Score	Avg timesteps	Std timesteps
HR	**0.527**	0.029	165.05	59.51	**695.86**	36.17
MCTS	0.467	0.034	**230.69**	74.64	942.06	**34.00**
EA	0.470	**0.026**	178.33	**51.85**	818.72	38.47

Table 5: results of all algorithms

The highest winning rate is reached by the heuristic controller (cf. table 5). But for the highest score we can see that the MCTS approaches reaches more points than the others. We suppose that the HR is playing tighter than the other. This is supported by the least played time steps as well. Playing tight leads to an earlier win. But there are some games where we could first of all collect points, increase the score and win at the end.

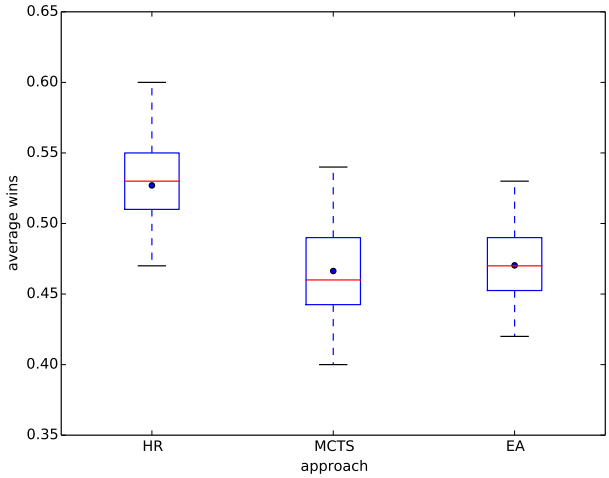

Fig. 12: boxplot of wins

By looking at the boxplot of the winning rate (cf. fig. 12) the winner of all agents is without any doubt the HR agent. In the best run he won 60 percent of all the games which is a very good rate. The mean is of course lower than this value but still better than the other approaches. The MCTS and the EA agent have both quite the same rates with similar quartile values.

In the overall evaluation we also have a look at the score (cf. fig. 13). There we can see a surprise since the HR agent has as an average value the lowest score at all. This might be because if he finds the target to win the game he will directly try to find it. By contrast the MCTS agent will look for the best action that has a score or a win as a consequence. You can also see three outliers of the EA that are caused by the randomness of that approach.

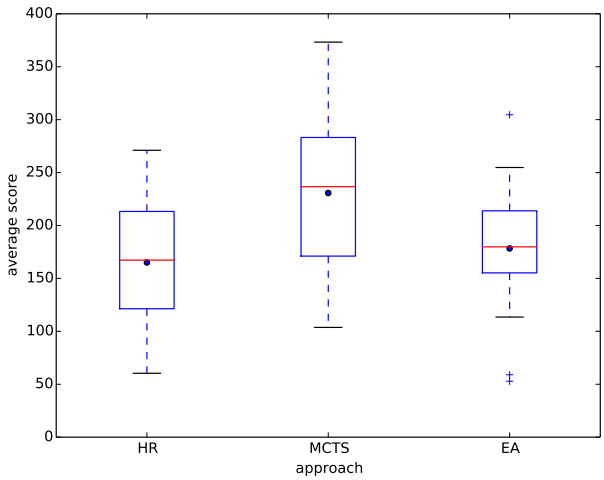

Fig. 13: boxplot of score

At least we have a look at the elapsed time in each game (cf. fig. 14). The HR approach plays very tight what has less timesteps as a consequence. Besides the MCTS agent needs the most timesteps but as we evaluated before with quite the same winning rate in comparison to the EA. This time is also used to increase the score.

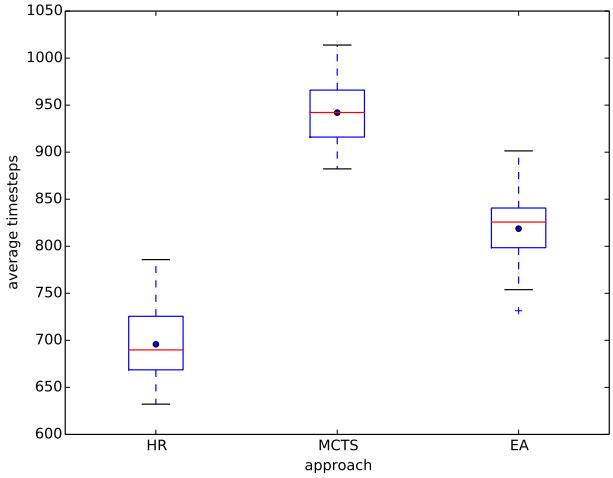

Fig. 14: boxplot of timesteps

The Evaluation of the winning rate, score and elapsed timesteps showed that each approach has a different strategy and trade off. playing tight could resolve in only try to win and not collecting a lot of points where it would be possible.

6 Conclusions and Future Work

This paper presents a comparison of several approaches concerning the playing of general video games. For each research area (HR, RL, NI) we selected one algorithm that we implemented.

For the evaluation we have to look at the data such as average wins, score and time steps. We ignored the in-game behaviour of our agents completely and only considered the statistics. Furthermore we were limited by time and computational power which restricts the iterations of our experiment. In order to produce more significant results, a experimental setup with more than 1000 iterations (like 3000 in the comparison between the best agents) has to be set up. Some future work could be to set this setup up and to test the algorithms with more computing power. The HR approach achieved the best results (depending on the number of wins) and proved to be better than the NI and MCTS. This might have several reasons. Firstly our implementation of MCTS and NI is might not be very effective or we might have failed to find the optimal parameters to fit our problem. The parameter problem, which occurs due to the lack of computing power, could be solved with a more powerful machine and a loop over a set of possible parameter-combinations. Another reason why the HR agent wins most of the games is that often there are only very few winning-sprites in the game which makes it hard for MCTS and NI to get a reward and win the game. These games are easy for the heuristic controller. The average score of the MCTS controller is higher because the HR controller wins very fast without collecting extra points. We cannot make general statements about the accuracy of the approaches (HR, RL, NI) using our controllers as quality criterion, but we can say that all of them are useful for GVGP. For future tasks, it would be interesting to compare the implementation of other controllers and methods (Neuronal Nets, Pheromone) to the actual ones. Furthermore a test with unknown games would be interesting, to compare the agents and the methods on completely unknown games. Another idea for future work is to combine the best properties of the three agents and and create a combined agent.

Acronyms

AI Artificial Intelligence
CIG Computational Intelligence in Games
EA Evolutionary Algorithm
GVGP General Video Game Playing
GVGL General Video Game Language
HR Heuristic
MDP Markov Decision Process
MCTS Monte Carlo Tree Search
NI Nature Inspired
RL Reinforcement Learning
TLU Threshold Logic Unit
UCT Upper Confidence Bound