Mateusz WIŚNIEWSKI
Artur NIEWIADOMSKI

Siedlce University of Natural Sciences and Humanities,
Institute of Computer Science,
ul. 3 Maja 54, 08-110 Siedlce, Poland

Applying artificial intelligence algorithms in MOBA games

Abstract. Multiplayer Online Battle Arena games focus mainly on struggles between two teams of players. An increasing level of cyberbullying [1] discourages new players from the game and they often chose a different option, that is, a match against opponents controlled by the computer. The behavior of artificial foes can be dynamically fitted to user's needs, in particular with regard to the difficulty of the game. In this paper we explore different approaches to provide an intelligent behavior of bots basing on more human-like combat predictions rather than instant, scripted behaviors.

Keywords: Multiplayer Online Battle Arena *(MOBA)*, Artificial Intelligence (AI), Genetic Algorithm (GA), computer game, computer game agents, bots

1. Introduction

The popularity of MOBA games has grown rapidly over the past few years becoming a worldwide trend. Increased usage of social media, and in particular live streaming website – Twitch [2], revealed plenty of gaming social problems [1]. The most experienced players bully not only newcomers but also skillful players what turns the joy of the game into discouragement and frustration. Usually, the disheartened gamesters find serenity in skirmishes against AI-controlled opponents. The companies developing games often do not put a big effort in this mode, because handling a vast amount of game rules is extremely difficult to implement. Moreover, it consumes a significant amount of computing power during the game [3].

The main contribution of this paper is a comparison of several algorithms applied to planning moves of the MOBA game characters. We analyze not only the theoretical complexity of the algorithms, but we also investigate how they deal with a dynamic battlefield environment. To this aim we performed a number of experiments in form of duels between heroes controlled by different algorithms. We compare the behavior and the efficiency of particular planning methods using several statistics collected during individual and team struggles.

The rest of the paper is structured as follows. In the next section we briefly describe the related work. Then, we focus on the general concepts of our approach and we discuss the compared algorithms. The two last sections are devoted to the analysis of the experimental results and conclusions.

2. Related Work

One of the most common solutions to control the behavior of bots, i.e., autonomous game characters, makes use of priority lists dividing the gameplay into long- and short-term objectives. An example of the former is to get necessary means for the battle, or push the front line towards the enemy base. On the other hand, the short term scripts mostly react to the current events, like, e.g., if an opponent approaches too close, then it should be attacked, or the retreat in response to received damage, etc.

This concept has been implemented in a patch for the game *Heroes of Newerth* [4] released in 2013. The possible hero behaviors are stored in a list ordered by their utility. If a tested condition is not satisfied, the algorithm checks the next one. The cases are inspected in short, 250 ms interval, loops. There are two main goals – a team attack or a team defense. The strategic decisions are dictated by a team controller which does not take in consideration the current state of the game. The system follows a simple algorithm where players scattered around the map are formed into a group once every few minutes regardless of the conditions.

Behavior Trees (BTs) [5] can be seen as an extension of the priority lists concept. BTs usually consist of hierarchically ordered nodes containing test conditions and operators controlling the flow of the decision process. The leaf nodes represent specific scripts and commands to be executed by bots. Comparing this approach to ours, BTs are focused on concrete decisions what results in more predictable behaviors. These are desirable features at the design of a tutor agent [6] or medium-advanced opponents.

Besides the low-level bots control, the AI algorithms are often applied to make strategic decisions at higher level of abstraction. For example, the authors of [7] exploit algorithms basing on Influence Maps (IM) to this aim. In the simplest case an IM is a matrix covering the battleground and aggregating information on the current and historical states of the game. An example advantage of IMs is a clear divide of the battlefield into safe and danger zones. In our approach, we implemented to this aim an alternative method based on dynamic computations of distances between bots [8].

Another interesting example of application AI algorithms in MOBA games is the paper [9] where the authors exploit Genetic Algorithm (GA) [10] to change parameter values of procedures controlling the basic bots behavior. The algorithm makes use of non-standard genetics operators. The crossover-like operator is used when a character interacts with an ally to mimic the behavior of a better acting team mate. On the other hand, an interaction with an enemy results in applying a mutation-like operator. Thus, GA runs and improves the characters' behavior during the whole game.

The crucial difference between [9] and our approach is that we use (and compare each other) several algorithms, including GA, working in a short-term scale to control the basic, low-level bots behavior and react to the state of the battlefield. In order to evaluate different

planning methods, we implemented a simple MOBA game using the Unity 3D [11] environment, as shown in the next section.

3. Solution overview

Our approach focuses on planning and executing of combat tasks in team fights. Instead of checking rigid scripts, several AI algorithms (from simple heuristics to Genetic Algorithm) try to estimate the hero's chances and predict the nearest moves. The designed game characters work in a loop consisting of computing multiple solutions (called also paths or plans), assessing them, and choosing and executing the best one before a new iteration starts. A new plan is computed also in the case when a new threat appears on the battlefield. An example path of length 3 is depicted in Fig. 1. The cross-marked green line corresponds to the movement track, and the numbers show the estimated arrival time to the subsequent points of the path. The red line visualizes the planned attack while the short, yellow line on the right shows the distance between the actual and the expected position of the opponent.

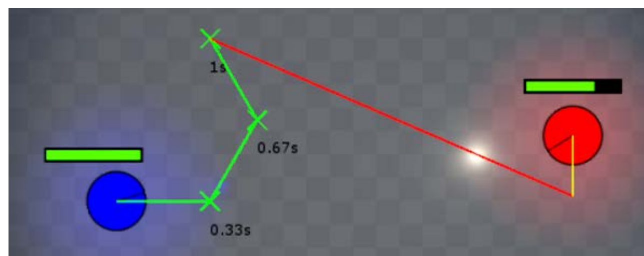

Figure 1. A currently executed solution by the hero on the left. After moving to the convenient position it will fire a projectile towards the opponent on the right. The green line shows the planned moves and the numbers correspond to estimated arrival time to the points marked with the crosses. The red line represents the planned attack and the yellow line corresponds to the estimated move of the opponent.

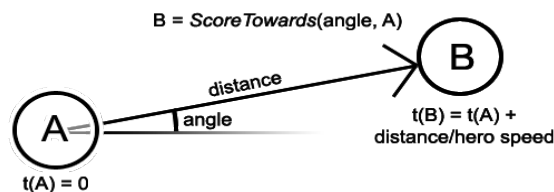

Figure 2. The *ScoreTowards* function computes the next step of the plan (B) basing on the previous location (A), the chosen direction (angle), the distance to cover in one step, and the movement speed.

A single track consists of several steps, each one reachable after some delay. The coordinates of the destination point and the estimated arrival time is calculated basing on the

current location, the speed of the hero, and the chosen movement direction by the ScoreTowards function (Fig. 2).

In every step of the plan, a hero can use its “abilities” which allow to attack an opponent or to protect itself. In order to select the best combination of movement and retaliation each solution has to be rated. The scoring process consists of additions of the benefits and subtractions of the downsides of the assessed plan. For example, some considered destination point may be a good place for an assault, but moving there would be unprofitable because it is located on a trajectory of an enemy’s projectile. Even a short stay in a danger zone has a significant impact on the overall plan evaluation.

The rating of an offensive ability is based on the evaluation of profitability, expressing to what extent the potential hit will affect the target, and effectiveness - estimating the chances to hit the target. Similarly we can assess the threats. For example, if the enemy missile cannot affect the hero, then there is no point in dodging it.

However, our experiments have shown that relaying too excessively on the prediction can be disastrous, because it may turn out that a currently executing solution is already outdated. Since the AI agents have to recalculate plans whenever a new threat appears on the battleground, the duels involving a significant number of players and abilities enforce switching the CPU context very often, resulting in heavy computations. Thus, in our real-time battle environment, the algorithms providing a solution quickly often take advantage over the slower ones, even if the quality of the latter is better.

4. State space search algorithms

The game characters usually traverse a two dimensional surface divided into a grid of squares or other geometric shapes forming a *Navmap* [12]. Both of the approaches are very popular, however we decided to follow another, more flexible concept. Our MOBA game environment, developed to compare the considered algorithms, builds a graph-like structure dynamically. Moreover, describing the agent position with float precision numbers allows the characters to wander through the battleground in a more organic, human-like, unpredictable manner. This also opens endless possibilities of choosing the points on the map.

Thus, to conduct performance tests comparing different algorithms, some variables need to be fixed. Those include: the number of steps in the plans, the distance covered in each step, and the number of possible directions. Below, we describe several methods of searching the state space. In the following five figures we have assumed 6 possible movement directions.

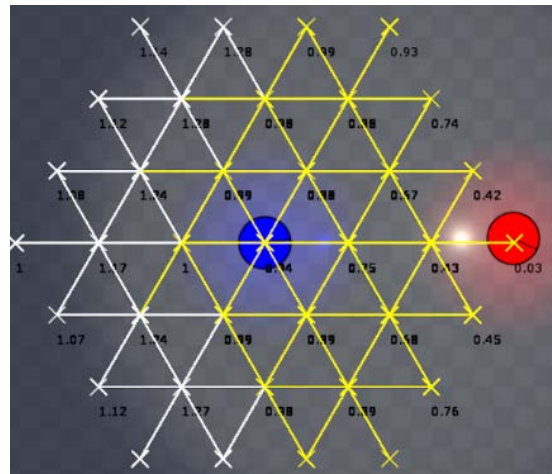

Figure 3. A visualization of paths evaluated by the Brute Force algorithm. The numbers correspond to ratings computed for the points marked with the crosses. All possible combinations of moves are considered.

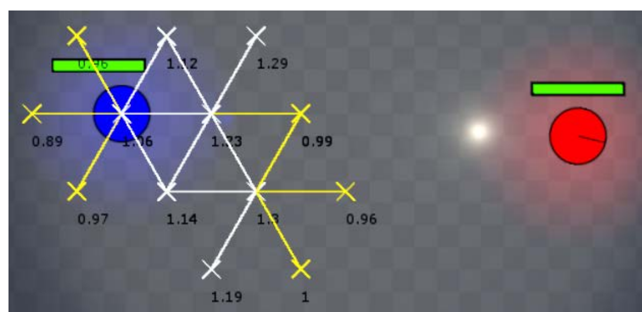

Figure 4. The paths considered by the Directional algorithm. The numbers correspond to ratings computed for the points marked with the crosses. The algorithm chooses a direction with the best rating and explores it further ignoring the remaining directions. In this example the direction "right" leads to the point with the highest rating (1.23), and it has been chosen in the first step. In the next steps "right-down" (with rating 1.3) and then "left-down" (with rating 1.19) have been selected.

The *Brute Force* (BF) algorithm, see Fig. 3, follows the simplest concept, but it is the most computationally complex of the implemented methods. To estimate the computational complexity we take into account the number of potential locations assessed by the particular algorithms. In the case of BF, all possible combinations of the available moves are considered. Notice that the same points can be calculated multiple times with different delays or various arrival directions. Assuming that n represents the number of possible directions and k stands for the length of the plan, the first step of BF is n computations concerning the locations reachable from a current position. Then, every point computed and assessed in the previous step is treated as a start location to plan the next move. That is, in every of the n new points again n directions are considered. Thus we have to compute and assess n^2 locations in the second step, and n^k positions in the general case. Thus, the computation complexity of BF is $\mathbf{O(n^k)}$. It is not surprising that, due to its exponential complexity, the BF algorithm is extremely inefficient.

The *Directional* algorithm (Fig. 4) is a greedy version of BF. In each point of each path, the algorithm considers all the available directions, chooses only the most promising one and explores it further. That is, in every of k steps of the algorithm the n new points are computed and assessed. Thus, the computation complexity of this algorithm is **O(k*n).** A good performance, the straightforward implementation and a linear scalability makes it a very viable option to consider while designing a game logic.

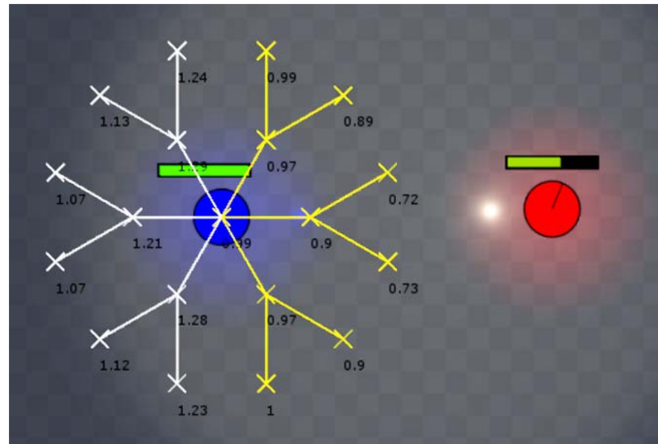

Figure 5. The paths generated by the *Split* algorithm. The computed points appear beyond the "hexagonal grid" created by the other methods. Every point computed in a previous step of the algorithm is considered as a start location for two potential moves using two new directions.

The *Split* algorithm starts from computing single steps using all available directions. Then, every obtained point is explored further, but only in two, new directions. An angle formed between the new directions is the same as the angles between the subsequent directions of the first step. For example, in Fig. 5 which depicts a plan of length 2 considering 6 possible directions, the angles formed between the subsequent directions in the first step equal 60 degrees, and so are the angles formed between the pairs of new directions in the second step. This algorithm prevents from generating backtrack paths, what allows to reach a safe distance from threats. Analyzing the computational complexity, we start from n computations in the first step. It is easy to observe, that the number of points computed in each subsequent iteration is two times greater than the number of locations generated in the previous step. Therefore, for n directions and k steps the computation complexity of *Split* equals:

$$n + n * 2 + n * 4 + \dots + n * 2^{k-1} = n * \sum_{i=0}^{k-1} 2^i,$$

what, using the O notation, gives us the computational complexity of **O(n*2^{k-1})**. Thus, *Split* scales worse than *Directional* according to the length of the plan, since every new step doubles the size of the state space.

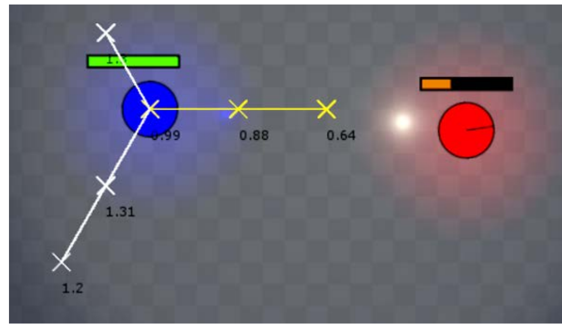

Figure 6. Example paths considered by the *Monte Carlo* algorithm. The numbers correspond to ratings computed for the points marked with the crosses. The algorithm randomly selects the number of explored directions and the lengths of the paths.

The *Monte Carlo* algorithm (Fig. 6) is the most unpredictable of the described algorithms. The number and lengths of the paths are chosen randomly. A hero controlled in this manner often stays still for some time, if none of the calculated destinations is better than a currently occupied point. Staying in the same place increases the recalculation frequency which makes this method extremely responsive to the environment changes.

The pessimistic computation complexity of *Monte Carlo* is **O(n*k)**, because the algorithm randomly selects the number of considered directions from 1 to *n*, and then, for every direction, it chooses randomly the number of steps from 1 to *k*. Thus, the pessimistic complexity is the same as for the *Directional* algorithm, however the experiments show that the average complexity is much lower.

The last of the considered planning methods is *Genetic algorithm* [10]. It maintains a population of individuals, where each one of them is a potential solution encoded as a vector of *k* integers. Every integer, i.e., a gene, stands for a move direction chosen in the subsequent steps, thus there are *n* possible values for a single gene. The evolutionary process is divided into several iterations. In every step of the algorithm, the individuals are assessed and modified using standard genetic operators, becoming a new generation processed in the next iteration. The implemented operators include a roulette wheel selection, a single-point crossover, and a random mutation. The initial population is generated randomly. Basing on several experiments, we set the crossover probability to 0.8, and the mutation probability to 0.05. Since the computations in the real-time environment have to be quick, we set the number of iterations (*I)* and the population size (*S)* to 10.

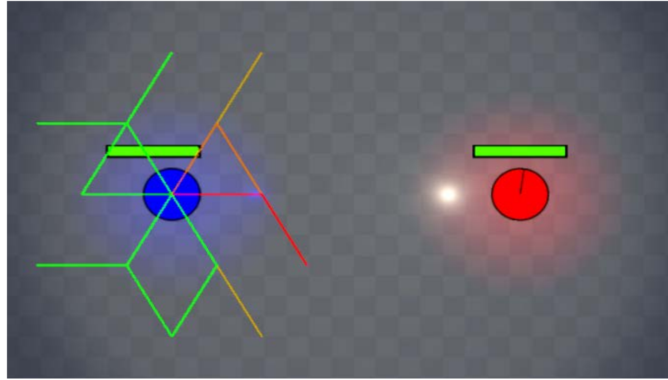

Figure 7. Example paths generated by GA. After several iterations, the trend to keep a safe distance from the opponent can be observed. The colors of the tracks correspond to the ratings of the particular points: the green lines are of high ratings, while the red ones are of low ratings.

Estimating the computational complexity using the same assumptions as for the other algorithms, it is easy to observe that the GA complexity depends on the *I, S,* and *k* parameters. During every iteration, so *I* times, the algorithm computes and evaluates *S**k* points. Moreover, at the start of the algorithm the initial population has to be assessed, so additionally *S**k* locations are processed. This gives us the computational complexity equals **O((I+1)*S*k)**.

5. Experimental Results

The experiments have been conducted in the form of short (5 min.) skirmishes between bots having the same abilities but controlled by different algorithms. The tests have been performed using several different values of the parameters: the algorithm to control the bots in each squad, the number of team members (b), the number of available movement directions (n), and the length of planned paths (k). The other worth mentioning parameters are, e.g., the movement speed (set to 1.5 body length per second), the update interval (250 ms), the initial number of life points (1000), the damage made by a projectile (120 life points), and the minimal interval between subsequent shots (1 second). Overall, 600 encounters have been performed for all possible combinations of different algorithms, with the following values of the parameters: b ϵ {1, 2}, and (n, k) ϵ {low = (6, 2), high = (8, 3)}. That is, one-on-one and two-on-two duels, with low and high settings, have been fought.

The results are summarized as four the most expressive statistics showing the average percentage of: the life points remaining after the fight (Fig. 8), the games won by particular algorithms (Fig. 9), the accuracy of fired projectiles (Fig. 10), and the efficiency of performed dodges (Fig. 11).

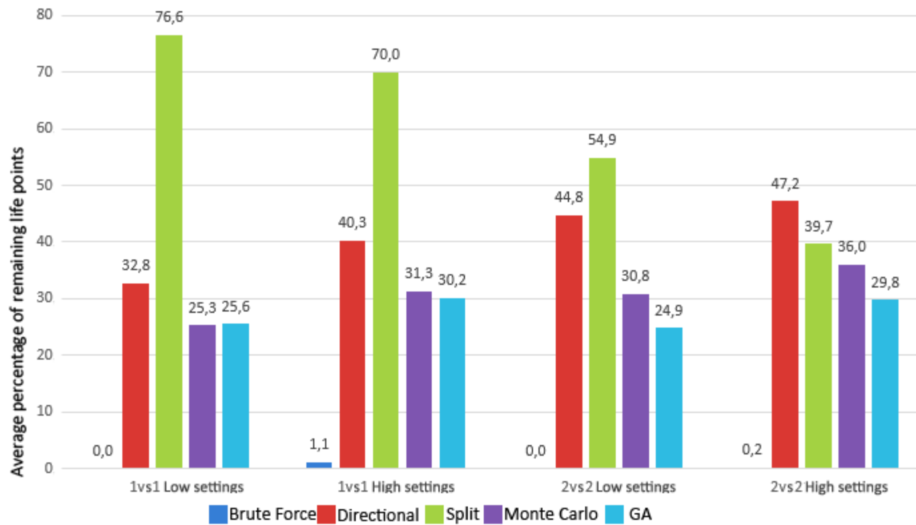

Figure 8. The average percentage of life points remaining after the fight. This chart shows the dominance of particular algorithms slightly better than win / lose ratio.

The teams controlled by the *Brute Force* algorithm have not won barely any battle. This can be explained by the excessive search for the best place to take a shot which is computationally expensive, even for small parameter values. The executions of the computed plans have been often interrupted by the emergence of new factors on the battlefield. The methods generating the paths faster fired bullets more frequently and fared much better. The quicker opponents had more time to react and change positions for counterattack. What would benefit BF, is a longer time needed to regenerate the shooting ability. However, even from its defeat we can learn that predicting too far ahead and offensive abstention can be disastrous.

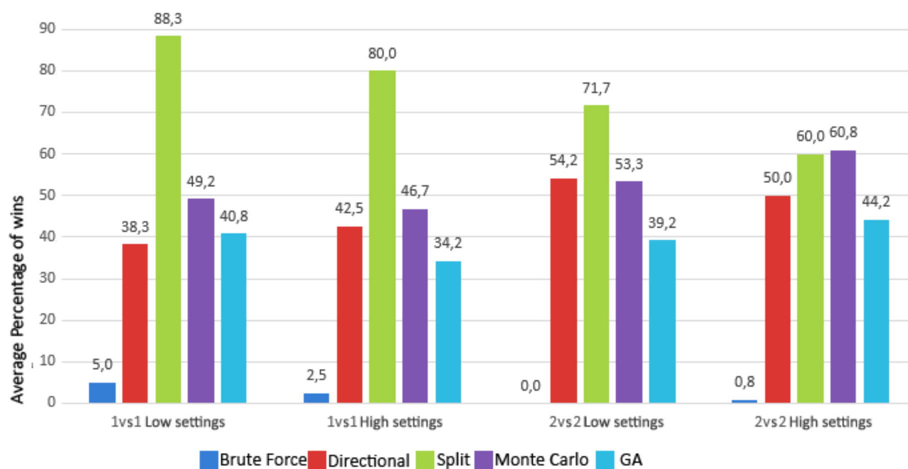

Figure 9. The average percentage of games won by particular algorithms.

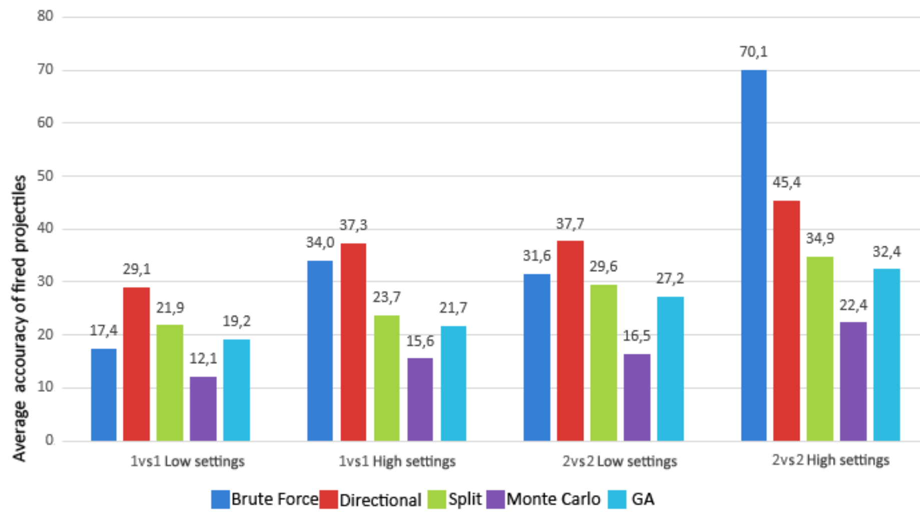

Figure 10. The average accuracy of fired projectiles.

On the other hand, the *Directional* algorithm has had the longest average encounter time of all considered methods. This success results from performing the best dodges. However, similarly to BF, it also restrained from shooting which did not allow it to show its full potential.

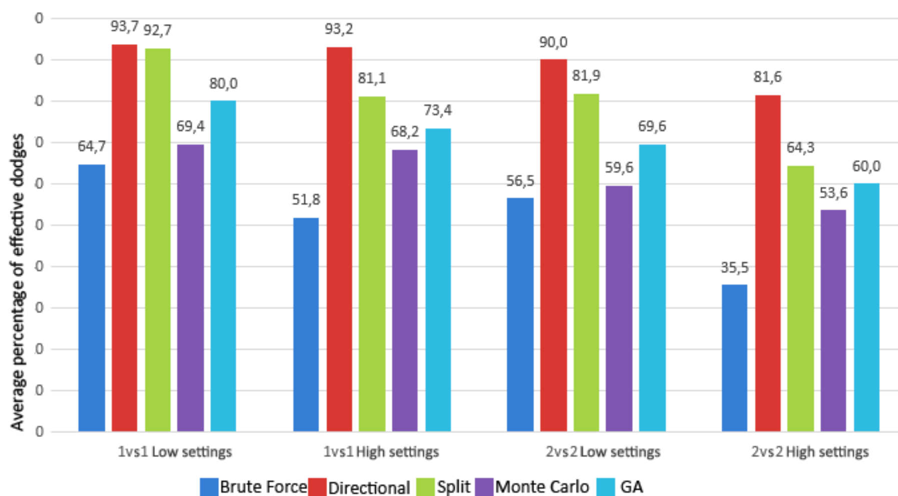

Figure 11. The average percentage of efficient dodges.

The *Split* algorithm appears the definite winner of duels with small parameter values. Its biggest competitor is *Directional*, but looking at the number of life points remaining after 5 minutes timeout it is easy to deduce that *Split* would have won if it had more time. One of its advantages is using the offensive abilities more often than BF and *Directional*.

Monte Carlo has the highest ratio of fired projectiles, but the lowest accuracy, just opposite to BF. The “stand still” policy often presenting by this method causes a high rate of recalculations and shooting, believing that the current place occupied by the character is the best. This method would cope much worse if the minimal interval between two attacks was longer. It also revealed how important are the often updates. On the other hand, standing still makes the bot an easy target for the enemy’s projectiles.

The *Genetic Algorithm* certainly has behaved as intended, however our expectations were definitively higher. It seems that the designed GA implementation is too heavy to cope with the real-time constraints properly. We plan to further investigate and optimize it, however, other more lightweight solutions, like, e.g., simulated annealing [13], seem to be also worth to follow.

6. Conclusion and future work

In this paper we have introduced a new approach to controlling heroes of MOBA games. The results obtained can be a basis to build lightweight AI-based game systems without using Navmaps or grids surprising players with its quality and uniqueness.

There are many possible space search algorithms to be exploited in MOBA games. However, due to tight time constraints related to the dynamics of battlefield, the methods quickly providing a solution would be preferred. The experiments have shown that the prediction time should be rather short, and the update intervals should be based on the status of the opponent abilities.

The implementation can be further developed in several directions, e.g., by modifying the function used to assess the plans, applying new planning algorithms (like, e.g. [14], [15]), or by introducing new variables to simulate more complex bots behavior. The latter include estimating positions of players hidden behind walls, increasing rating in a safer position near walls, covering wounded teammates from inevitable death or even simply learning of successful solutions to mimic them in future. We plan also to develop and evaluate the approach based on machine learning techniques, switching between several planning algorithms in reaction to the environment changes.

References

1. Kwak Haewoon, Blackburn Jeremy, Han Seungyeop. Exploring Cyberbullying and Other Toxic Behavior in Team Competition Online Games, CHI '15 Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems, pp. 3739-3748, 2015.

2. Hamilton A. William, Garretson Oliver, Kerne Andruid, Streaming on Twitch: Fostering Participatory Communities of Play within Live Mixed Media, CHI '14 Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 1315-1324, 2014.

3. Nae Vlad, Iosup Alexandru, Prodan Radu. Dynamic Resource Provisioning in Massively Multiplayer Online Games, IEEE Transactions on Parallel and Distributed Systems: 22(3), pp. 380–395, 2011.

4. Kvanli Erik, Hammerstad Eirik M. A Coalition based Agent Design for Heroes of Newerth, Norwegian University of Science and Technology, Master's Thesis, 2014.

5. Lim, C. U., Baumgarten, R., Colton, S. Evolving behaviour trees for the commercial game DEFCON. In Applications of Evolutionary Computation, pp. 100-110, Springer, 2010.

6. Victor do Nascimento Silva, Chaimowicz Luiz. A Tutor Agent for MOBA Games, Proceedings of the XIV Brazilian Symposium on Computer Games and Digital Entertainment (SBGames 2015), pp. 220-223, ISSN: 2179-2259, 2015.

7. Victor do Nascimento Silva, Chaimowicz Luiz. On the Development of Intelligent Agents for MOBA Games, Proceedings of the XIV Brazilian Symposium on Computer Games and Digital Entertainment (SBGames 2015), pp. 131-139, ISSN: 2179-2259, 2015.

8. Wiśniewski Mateusz. Zastosowanie Algorytmów Sztucznej Inteligencji w Grach typu MOBA (in Polish), Siedlce University of Natural Sciences and Humanities, Master's Thesis, 2016.

9. Siddhesh V. Kolwankar. Evolutionary Artificial Intelligence for MOBA / Action-RTS Games using Genetic Algorithms, International Journal of Computer Applications (IJCA) (0975 – 8887), pp. 29-31, 2012.

10. Whitley Darrell. A genetic algorithm tutorial, Stat Comput. 4: 65, doi:10.1007/BF00175354, 1994.

11. Creighton Henson Ryan. Unity 3d Game development by Example, ISBN 9781849690546, Packt Publishing, 2010.

12. Brand Sandy. Efficient obstacle avoidance using autonomously generated navigation meshes, Delft University of Technology, Master's Thesis, 2009.

13. Kirkpatrick, S., Jr, C. G., Vecchi, M.: Optimization by simulated annealing, Sci., 220, pp. 671–680, 1983.

14. Niewiadomski A., Skaruz J., Świtalski P., Penczek W. Concrete Planning in PlanICS Framework by Combining SMT with GEO and Simulated Annealing, Fundam. Inform. 147 (2016), pp. 289–313, IOS Press, DOI 10.3233/FI-2016-1409, 2016.

15. Niewiadomski A., Skaruz J., Penczek W., Szreter M., Jarocki M. SMT Versus Genetic and OpenOpt Algorithms: Concrete Planning in the PlanICS Framework, Fundam. Inform. 135(4), pp. 451-466, 2014.