Tank Wars – AI report

Authors: Aiden Poonwassie p16201118, Jan Kalinowski p17179637,

Nabil Salam p16220683

Tutor: Dr Liang Hu

Table of content

Introduction

There are many aspects that decide the game is considered to be good. In fact, it is hard to determine what combination of those aspects makes a game good and it is almost impossible to clearly distinguish games between 'good' or 'bad'. In each and every game the developers put the accent on different aspects – sometimes those are stunning visuals and visual effects, in other games, it will be a really engaging, emotional story.

There is, however, no doubt, that AI is a really important aspect in many games' genres. It does not matter whether the AI is simple and only moves NPCs around the screen or it is a really complex built intelligence similar to what humans possess. This paper provides some insight into AI theory and history. Then it moves onto describing several out of many existing AI techniques. At last, it looks at our implementation of tank AI in "Tank Wars".

Games' AI history

Mankind had been fascinated by artificial intelligence for a very long time. Although people did not consider the same way as we do now, the examples of that were middle ages' golems and Mary Shelley's Frankenstein.

The foundations for AI development have been laid in 1936 with the creation of Alan Turing's machine. The machine was able to break down and execute cognitive processes represented by algorithms. The term AI has been created in 1956 by the programmer, John McCarthy.

AI was firstly used to create the world's first chatbot (1966), support treatment (1972) and provide human-like behaviour, such as reading words out loud (1986).

When AI was being developed to perform the above tasks, it was not widely used in games. First video games, such as *Spacewar!* and *Pong* were based on real players' input and did not involve AI at all. That was the case until the mid-1970s – in 1974 Taito created *Speed Race* and Atari released *Qwak* (duck hunting) game. AIs used there were really simple and made use only of different movement patterns. It was, however, a breaking point and since that moment AI started gaining more and more popularity in video games.

The AI was popularized during the golden age of arcade games. *Space Invaders* (1978) supported increasing difficulty level, several movement patterns and events based on the player's actions.

Pac-Man (1980) introduced AI to maze games and have differentiated each enemies' personality.

Over the years AI had been becoming more and more complex introducing new solutions to previous problems. For example, in the 1990s, games started to make use of finite state machines. With the rising popularity of real-time strategies (RTS) have been real a real challenge to AI development, as they introduced many new, previously unknown, problems. It was, amongst others, pathfinding on many, not predefined maps, real-time tactical and economic decisions. The icing on the cake was beating the world's best chess player, Garry Kasparov, by an IBM's supercomputer - named Deep Blue - in 1997.

Modern AI

Nowadays the use of AI in games is not much different than it was back in the 20th century, but the computational power of computers has become much greater, therefore the developers had an opportunity to enhance AI's abilities. Non-player characters (NPCs) present an almost human-like behaviour with – for example – marking a player's position, evaluating the number of resources left or finding a better path to the destination.

In *F.E.A.R.* (2005) enemies are working with each other and they are using the environment to gain maximum advantage over the player and respond to player's actions – for example, they can jump through the window or flank the player. A real-time strategy *StarCraft II* (2010) made use of AI in single-player mode. AI's difficulty range from very easy to insane, where the three highest difficulty levels are giving the AI great advantages over the player, such as vision of the players' units or increased resources profits. *Civilization V* (2010), a turn-based strategy, supports up to 12 Artificial Intelligences in their single-player mode. The amount of decisions the AI has to make in this game is tremendous, varying from declaring wars and looking for alliances, to colonizing new territories on best terrain (resources rich) possible and exploring the map.

Currently, the most common mean of controlling AI are behaviour trees. They are efficient and fairly easy to code but may lead to a certain level of AI's stupidity, which can not respond to the player's actions correctly but performs his actions repetitively. Pathfinding algorithms, such as A*, also suffer in games such as *Civilization* and *StarCraft* due to large amounts of units and obstacles existing in the game, causing the units to get into each other's way.

As mentioned above, the AI is in constant development and everyday developers from various companies are working hard on improving human-made intelligence. A good example of that is London based company DeepMind, which has created several successful Artificial Intelligences. In 2016 an artificial intelligence named AlphaGo, which was making use of a combination of neural networks and machine learning has beaten the best professional Go players. In 2017, another AI from DeepMind, AlphaStar, has beaten two professional StarCraft II players. In the same year, OpenAI from Elon Musk's company has beaten professional Dota 2 team.

All of those games – Go, Dota2 and StarCraft – although are different in terms of rules and how to play them, contain vast amounts of decisions to make on every step of the game. In addition, Dota2 and StarCraft II require economic and tactical planning and the decisions amount is even bigger. Those AIs made use of deep machine learning and neural networks in order to learn and duplicate real player's behaviours and tactics. The successes of Alpha and OpenAI Artificial Intelligences were milestones in gaming's AI, as the world has seen, for the first time, that artificial intelligence is able to beat humans and is efficient enough to run on an average user's computer. For sure more research upon this topic will be made and such AIs might become a standard AI in video games.

What makes a good AI?

The creation process of Artificial Intelligence is tough and requires a lot of decision to make. A really important concern is how good the AI should be. It can not be too smart, as a game would become annoying and unplayable. On the other hand, it can not be too stupid, as the game would become too easy and boring. What makes a good AI?

In order to understand what means a 'good AI' we need to remember, that desired AI's behaviour varies from game to game. For some games, AI has to be really simple and straightforward and in the others, it needs to be smart and aggressive. Development of a good AI is not only about technical issues, but mostly about the design. A good example of both behaviours blended together is almost any zombie game – or at least their weakest zombies – where the AI just charges forward whenever it detects the player, whereas in the late stage of the game when players encounter stronger enemies, they need to be smarter than the basic ones.

First of all, AI is not only about enemies. Many games, such as *Mass Effect* (2007) or *Dragon Age: Inquisition* (2014) make use of companions. They not only require to be a bit smarter than an average opponent, but they are also expected to interact with the player and add depth to the game's world. This may be achieved by adding exclusive conversations with them or creating a unique personality for each companion.

It is important, that an Artificial Intelligence reacts to the player and his actions. For instance, in *Shadow of Mordor* (2014) enemy war chiefs remember the player and if – for example – a player had run away from the battle, the next time the chief will be encountered, it will respond with the appropriate dialogue. A similar case occurs in *Metal Gear Solid V* (2015), where enemies wear gear to counter the player. That means, that if a player uses long range rifles and aims for enemies' heads, the NPCs will start wearing helmets. It applies, however, not only to enemy NPCs but also to previously mentioned companions.

That leads us to the third aspect contributing to what we call a 'good AI' – it interacts with game systems. In *Mass Effect* (2007) players are allowed to choose 2 out of several companions to help them during missions. Each of the companions have different abilities and perks, so it is possible to combine them and deal increased damage to the enemies. For example, one of the companions – Liara – uses sort of magic (called *biotics* in the game's world)

and is capable of lifting enemies. The lifted enemy is more vulnerable and therefore players can deal more damage, so Liara is making use of that skill pretty often. Another example of interacting with game systems is *Bioshock* (2007), where enemies may rush to the health dispensers in the middle of the fight. The health dispensers are also free to use for players. By doing so, NPCs appear to be more intelligent and aware of their wounds.

A good AI has its own goals. This is best visible in games with open world and ecosystems, such as *Spore* (2008). In this game, some monsters hunt other monsters but may be more friendly towards others. Those dependencies also affect players. A good example of goal-driven artificial intelligence is *Civilization V* (2010), where each of the leaders in single player modes have their own personalities and may focus on different things. For example, one leader may focus on colonizing new territories, whereas another will maintain a small empire, but will focus on building miracles of the world.

Players expect AI to behave in a certain way, that is why Artificial Intelligence should be predictable. For example, if an NPC is trying to repair a damaged power generator, a player expects it to always do so. That way, players can learn the game dynamics, define their own goals and play intentionally. The AI should remain consistent in its actions. Another level of predictability is achieved by telling the player what is AI thinking. This is called 'barks' and is presented to the player in various ways – by voice, images or even by animations. For example, in *Splinter Cell: Blacklist* (2013) when the AI is going to check something, it asks itself a question if – for instance – the light was switched on when it was there last time. That way, the player can learn about AI's intentions and what action it wants to perform, so that he can plan on his further actions. In order to maintain the game engaging and not too easy is achieved by adding responding to player actions, such as an earlier example of *Metal Gear Solid V* (2015).

Last, but not least, good AI allows players to cheat. It is, however, done in an unnoticeable way, but also if it was not present in the game, players would feel its consequences. A good example of that is *Far Cry 3* (2012), where only a few opponents are allowed to shoot at the player at one time, so the game does not seem too hard. Another good sample of that is *Uncharted 2: Among Thieves* (2009). In that game, when the player is popping out of the cover, the NPCs have a 0% chance of spotting him, giving the player the chance to fire a few shots at them.

To make a good AI is a tough task and therefore developers often cheat about that. They can, for example, make companions invulnerable, so that it is easier for the players to take control of the game. Developers can also increase enemies' health points (HP) and damage. A playtest on *Halo: Combat Evolved* (2001) has shown, that enemies with those attributes increased appeared a lot smarter to the player, than those with lower HP and damage.

Pathfinding algorithms

Game maps can be full of various obstacles, borders and boundaries, which constrain movement, both players and NPCs. Pathfinding is one of the basic behaviours provided by artificial intelligence.

In order to achieve that, AI may make use of predefined movement patterns, or try to find the shortest path possible. The first solution is quick and easy to code, but may be inefficient and lead to various issues with NPCs blocking themselves in the world.

The solution to that is pathfinding algorithms. In order to achieve that developers need to divide the map into nodes first. They usually have the shape of a square or a hexagon. By doing so, they are easy to represent and efficient to explore and evaluate. There are several algorithms allowing the developers to explore the node and look for the best route to the destination.

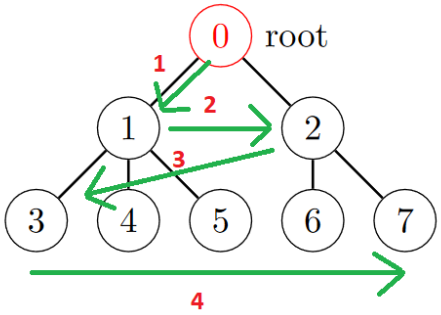

Figure 1. BFS search

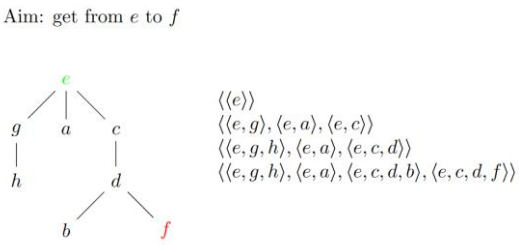

Figure 2. BFS' path

The first algorithm is BFS, which stands for breadth-first search. This algorithm starts at the root and explores each level iteratively and stops when reaches the destination. It is implemented using a queue, which is First In-First Out (FIFO) type of data structure. That means, that the first element that comes in is also the first element to go out of the queue. BFS finds the path to a destination (given node) by putting partial paths onto the queue.

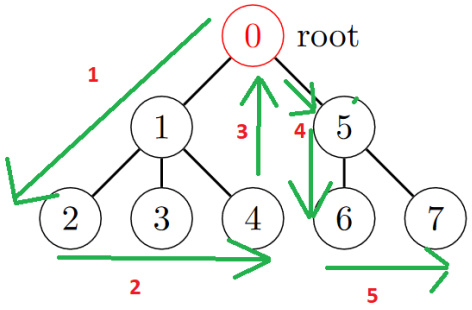

Figure 3. DFS search

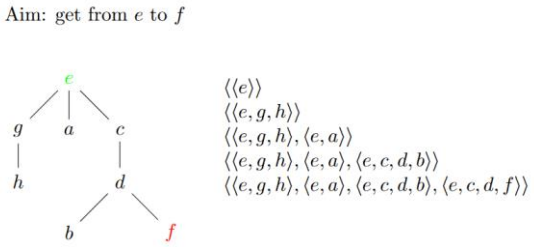

Figure 4. DFS' path

The second pathfinding algorithm is called Depth-First Search (DFS). This is very similar to BFS but first explores as far as it can and checks each path before considering any of its branches. When the algorithm reaches its limit, it backtracks and continues the search, but stops when reaches the goal. It has, however, different implementation to BFS, as it requires stack, rather than the queue. The stack is a Last In-First Out (LIFO) type of data structure, which means, that the last item that comes in is also the first item to go out. As the stack is implemented natively by computers, DFS is naturally recursive, which means, that the computer can recall the DFS execution (with a different starting point at each iteration step) until it finds the path.

Although BFS and DFS are easy to understand and implement, they may not be very efficient in the complex scenario, such as the one presented in *Civilization V*. In that case, much more effective is A* algorithm. A* is also the most widely used algorithm for pathfinding in modern video games.

Implementation of A* is a bit more complex than the implementation of BFS or DFS. It makes use of best-first search, heuristic and geographical distance. Heuristic distance is an estimated distance to the goal, while geographical is the distance travelled from starting point. The total cost of the node is calculated from:

$$F = G + H$$

Where:

$$G - geographical\ distance$$

$$H - heuristic\ distance$$

G and H are possible to calculate with several techniques. One of them is Euclidean distance, which calculates the square root of the distance from the currently explored node to the initial explored node (for G) and from the current node to the destination (for H). It can be generally described as:

$$dist = \sqrt{x^2 + y^2}$$

Where:

$$x = goal.x - current.x$$

$$y = goal.y - current.y$$

This is, however, not very efficient, as square root is computationally expensive. It is possible to perform calculations in a different way, for example, Euclidean distance without using square root. This is calculated from:

$$dist = x^2 + y^2$$

Where:

$$x = goal.x - current.x$$

$$y = goal.y - current.y$$

Another way is to use Manhattan distance, which can be calculated from:

$$dist = D * \big(abs(x) + abs(y)\big)$$

Where:

$$D - minimum\ movement\ cost$$

$$abs(x) - absolute\ x\ value$$

$$abs(y) - absolute\ y\ value$$

Slightly different approach to Manhattan is presented by a diagonal approach to distance. It can be calculated using the following formula:

$$dist = D * \max\big(abs(x), abs(y)\big)$$

Where:

$$D - minimum\ movement\ cost$$

$$\max\big(abs(x), abs(y)\big) - bigger\ value\ from\ the\ pair\ of\ numbers$$

$$abs(x) - absolute\ x\ value$$

$$abs(y) - absolute\ y\ value$$

Heuristic	Equation	Result
Euclidean	$\sqrt{\delta x^2 + \delta y^2}$	4.47
Euclidean no sqrt	$\delta x^2 + \delta y^2$	20
Manhattan	$D \cdot (abs(\delta x) + abs(\delta y))$	6
Diagonal	$D \cdot \max(abs(\delta x), abs(\delta y))$	4

Figure 5. Mathematical formulas with their results

Above picture shows different heuristic calculations techniques and their results. In all examples, the starting point was at (0, 0) and the destination was at (4,

2). As we can read, different formulas return different values. It is important to know, that the closer the value of total cost to the true distance, the more accurate A* is. It means, that if the developer chooses less computational expensive formula, he also chooses a less accurate algorithm. It does not, however, mean that the final path will be different. Instead, it means, that more nodes will need to be explored while looking for the shortest path, which results in extending the time of searching.

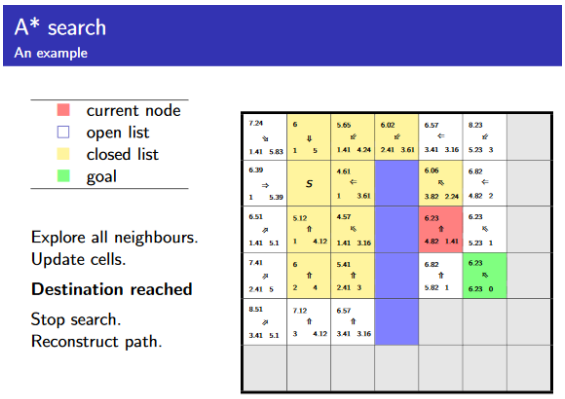

Figure 6. A searching process*

The A* algorithm knows its starting and endpoint. Then it explores nodes around the beginning point and evaluates them in search of the shortest path. When all of the neighbours have been explored it sets the current node to be the parent and moves onto the next node with the best (lowest) total calculated cost. It repeats this process until it reaches the goal node. When that happens, it backtracks, reconstructing the path, back to the node it started at. It requires a list of open and closed nodes for implementation.

Heuristic is problem specific and requires to be implemented and adapted to the problem, hence A* is not a universal algorithm. Despite that, it is still widely used and provides the best results in many successful games.

State machines

It is a common practice to control AI's behaviour and it is the desired functionality to know what actions AI will perform. Therefore developers have created a system called 'finite state machine' (FSM), which allowed them to achieve exactly what was needed.

The idea of the system is really simple: it is a discrete set of states and each of the states represents some behaviour. The states can be represented in numerous ways, using Boolean logic (true or false), integers or enums (which are, essentially, also just integers, but in more human-readable format).

```
enum TankState {
    IDLE, MOVE, ROTATE_TURRET,
    ROTATE_TANK, TARGET_FOUND,
    AIM, FIRE }
state;
```

Figure 7. An example of enum containing states

The above picture displays an enum containing states. Finite state machines are easy to understand, quick to code and exceptionally easy to debug. They are, however, unscalable and contain large amounts of redundant code.

In order to perform an action, the states need some conditions in order to know when to transfer to each other. This could look as follows:

Initial state	Transition	Final state
Idle	Move destination set	Move
Move	Target in range	Aim
Aim	Facing the target	Shoot

A great feature of FSM is, that developers can encapsulate them. It means, that although the system is simple in its idea, the structure may become really complex. For example, if a developer requires to have miners in their RTS game, but apart from mining he wants to give them real-life behaviours, such as hunger, tiredness or tools usage, he can create a couple of mother states with many sub-states inside.

FSMs are used in most games in order to create bots (enemies) or units in strategy games. They suffer from lack of scalability and redundancy, but it is being made up by their simple systems, which are easy to understand and quick to code.

Rule-based system

Similar to any computer program, AI requires some rules to operate on. At the end of the day, the program needs to know what to do in case of certain conditions appearance.

Rule-based systems make use of Boolean logic, such as OR, AND or NOT. One of the most common examples of Boolean logic is login to the website:

$$IF\ UserName(correct)$$

$$AND\ Password(correct)$$

$$THEN\ Login(successful)$$

And ∧

A	B	A∧B
0	0	0
1	0	0
0	1	0
1	1	1

Figure 8. Boolean AND table

Or ∨

A	B	A∨B
0	0	0
1	0	1
0	1	1
1	1	1

Figure 9. Boolean OR table

Above figures display examples of some of the Boolean operations and their results.

Rule-based systems can be chained forwards or backwards. In the forward chaining, the antecedents (predefined set of facts) of each chain trigger the result (known as consequence). A perfect example of that is above example of login system. It can be represented by a simple IF statement, such as:

```
if (turretAngle < 0.f)
{
    turretAngle += 360.f;
}
```

Figure 10. An example of if statement

On the other hand, backward chaining first checks the fact and if it is false, then looks for the assertion of consequences. This could be represented by a boolean function, such as:

```
bool SmartTank::OutOfAmmo()
{
    if (getNumberOfShells() == 0)
    {
        state = MOVE;
        return true;
    }
    return false;
}
```

Figure 11. Boolean function

Rule-based are the opposite of previously mentioned finite state machines – rather than defining NPC's behaviour based on the current state it is a goal-driven system. That means, that an NPC has a final goal – such as 'find food' or 'mine resources' – and the backward chained rule-based system will determine this NPC's current behaviour.

Behavioural trees

Above described systems are useful and necessary, but they are pretty unextendible nor complex enough. Both of them also have their advantages and disadvantages. The answer to this is behavioural trees, which are a compromise between finite state machines and rule-based systems.

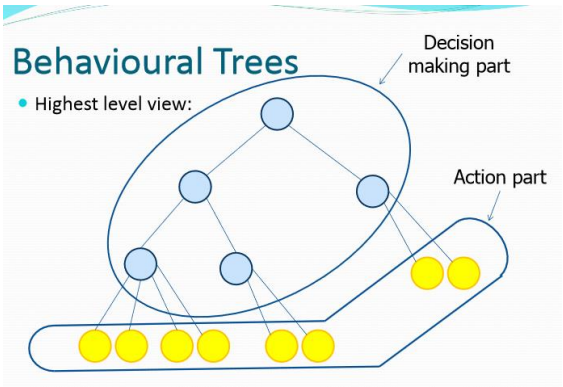

Figure 12. An example of behavioural tree construction

Above picture displays an example of such a behavioural tree. They are basic in terms of complexity but may create really complex structures, which will present human-like decision making. Each branch contains so-called leaves, which may be both conditions or actions. They are processed being processed step by step. A good example is an NPC trying to shoot at the player. In order to do that an NPC requires to have a line of sight, a weapon and ammunition. If all of those conditions are met, then the NPC will shoot. Otherwise, NPC will be forced to choose a different action, for example running away or melee attack at the player.

The flow of that may be controlled by introducing decorators. They are another type of nodes, which may be of different purpose. For instance, if all of the conditions of the previous example are met, developers may use an iterator in order to shoot not once, but 3 times. Decorators, however, are not only iterators, and as mentioned above, they may be of different purpose. They may check if an NPC has recently taken a similar action or randomize the outcome of an action, such as a dialogue line.

There is also a special type of a decorator called a lookup decorator. It allows the developers to encapsulate behaviour and trigger them when they are needed. It adds an extra level of abstraction and separates the goals from behaviours. It also significantly reduces the amount of redundant code.

Behavioural trees are a great tool to build NPCs' behaviours on. It is abstract, yet modular and has a built-in safety mechanism, which prevents NPCs from performing weird behaviour or freezing without knowing what to do next. By encapsulation, it becomes easy to debug. It is also goal driven and responds to the environment, which creates an impression of human-like behaviour.

Tank wars AI

With all of the information above, it is time to take a look at our implementation of tank's AI in 'Tank wars' game.

We have started the development by establishing the desired artificial

intelligence. We have thought about what kind of behaviour do we want to implement and how smart the enemy's tank should be. Keeping in mind the good AI rules we have decided not to go for a too smart tank, as that would make the game annoying. We wanted, however, the AI to present similar functionality as the player's tank – for instance, we wanted the AI to remember the friendly buildings' positions or to be actively looking for targets. We also have thought about AI techniques we wanted to implement.

Figure 13. Screenshot of the game

We have decided to make use of finite state machines in order to control what tank will be doing, rule-based system in order to provide conditions of what actions to perform, such as in what direction the turret should be rotated and behavioral trees in order to provide safety to the tank's behavior and reduce amount of unintended behaviors.

Firstly, we have introduced FSM, as we have decided this is the very basics of the AI we wanted to develop. For quick pathfinding, created however just for the sake of testing at the beginning – we have made a random number generator, where the generated number was tank's destination. After that, we have created a rule-based system by introducing controlling flags to the program so that we could control the AI in each of the states. For the states used in our game please see figure 7.

```
bool tankSpottedFlag = false;
bool scanFlag = false;
bool scanCheckpoint = false;
bool eBaseSpottedFlag = false;
bool fBaseSpottedFlag = false;
bool lineOfSightFlag = false;
bool firingFlag = false;
bool movementTargetFound = false;
```

Figure 14. Flags in the game

After creating the basic movement we have made a targeting system. As soon as the player's buildings come in range the tank stops the movement, aims at the closest of them and shoots. The system is also designed to prioritize the player over buildings. If the player is in range of AI's vision, it constantly aims the turret at the player, so that player feels endangered and needs to continuously respond to AI's behaviour. These attributes make AI seem more aggressive and the game is more challenging, as the enemy is more likely to gain a higher score. When there are no more targets for the AI, it continues the interrupted movement.

It was reasonable to make AI scan the area as long as it reaches the destination. As the tanks have 2 kinds of visions: one is around them, and the other one goes from turret forwards and has a shape of a triangle. Keeping that in mind we have made the turret rotate 360 degrees around the tank so that the AI is

scanning the area, searching for possible targets in the firing range.

Introducing scanning behaviour made us also think of behavioural trees. Keeping in mind what makes a good AI, we wanted the tank to perform correct behaviour if certain conditions occur so that players could learn game dynamics. For example, when the tank finds a target it checks what to aim at, aims at that and shoots.

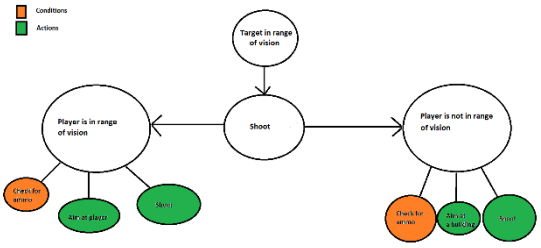

Figure 15. Behavioural tree graph made upon the game's example

At this point, we have decided to redesign movement to an A* algorithm. We had a couple of requirements for that: we wanted the AI to avoid friendly buildings, but mark and remember their position. We also did not want to give the buildings' position at the start, but we wanted to make the AI unaware of that and find them itself.

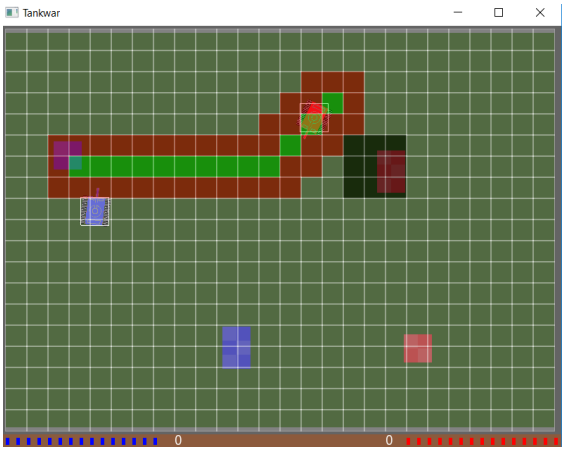

Figure 16. A in the game*

In order to do that we have divided the map into the nodes. Each node is of size 30x30, which created a grid made of 26 columns and 19 rows. This is slightly less than the tank's size but makes a fair amount of nodes to choose from. The above picture displays how the grid looks like – green nodes in the open list create a path, red nodes are those in the closed list. White nodes are nodes that will be considered as a path, while black ones are those excluded from being a possible path. In the picture we can also see, that only 1 out of 2 red buildings' positions are of a different colour – it is due dynamically locating system we have created. The AI does not know, that those buildings are there so that its gameplay is similar to the players.

The reason for making the AI initially unaware of the environment was not trying to make it too smart. This does not seem like a big deal, but excluding the nodes at the start from being a possible path did greatly include the chance of choosing the destination on the left-hand side of the map where the player's bases are. It was the same reason that made us make all of the nodes of equal chance to be chosen as a possible destination.

In terms of coding itself, we tried to keep everything easy to understand. We have made use of functions previously implemented to the game, for example for clearing movement or checking amount of shells left to fire. For example, we have got rid of magic numbers, which allowed us to quickly debug the code whenever the calculations were off. We have reduced the amount of redundant code by wrapping FSM in a switch statement, which executes the code whenever it should be executed. We also have used pointers, so that the

amount of memory needed to run the game is decreased and the memory is being cleaned after the program is closed, so there are no memory leaks. We did not alter the game's behaviour, but in order to make the AI work, we had to slightly alter the game's code. This includes, for example, adding if statements in game.cpp for demarking enemies or deleting bases.

There was also a need of thinking about the scenario, where the AI runs out of shells, but the game keeps going on, as the player still has some shells. In that case, we made a quick check if the AI is out of ammo before performing any action and if it is, it ignores everything, but just keeps moving around the map to new destinations. This makes the AI harder to find and target, hence players have to think more about what to look for in order to win the game.

We have not, however, make use of two functions defined in the class. Those are MarkShells and score. The reason for not using the score is, that the score was actually already being handled by the game, so it did not require any further changes. Marking shells have not been used due to the nature of the game itself – the shells are simply moving too fast to avoid them. This was noticed during many playtests and not using this function was a collective decision. Besides, if AI was able to avoid shells, there was a danger of making the game too hard.

By combining a couple of AI techniques and utilizing A* algorithm we have created a reasonable AI. We tried to make the game challenging and interesting, but not annoying, hence – as mentioned above – we did not make the AI too smart. We have tried to make AIs gameplay and 'experience' similar to the player's and our AI answers to our expectations.

Summary

In order to create a good AI, we had to investigate many new to us areas, such as basic AI techniques and pathfinding algorithms. We had to understand exactly how they work and how we can utilize them to our best. We also had to understand the game and how its data flow between classes so that we could make use of them in our game.

We introduced 3 new classes in order to implement A* pathfinding and we made classes' data flow between one another. During many playtests we had performed, both individually and collectively, we have located and solved all of the encountered issues, so the game should be bug-free.

With that said, we found the assignment really challenging, but also fun to develop and work with. We have acquired a lot of new knowledge about the AI in general, its implementation, but also about teamwork and project management. We agreed, that gained experience will surely help us in our future career.

References

[1] Greene, K. (2018). *How AI is used in video games and what we should expect in the future*. [online] Kulture Hub. Available at: https://kulturehub.com/ai-games-artificial-intelligence/ [Accessed 11 Apr. 2019].

[2] Dickson, B. (2018). *Why Teaching AI to Play Games Is Important*. [online] PCMag UK. Available at: https://uk.pcmag.com/opinions/116551/why-teaching-ai-to-play-games-is-important [Accessed 10 Apr. 2019].

[3] Allen, T. (2019). *The future of AI is not in sentient robots, but it might be in gaming | Computing*. [online] Computing. Available at: https://www.computing.co.uk/ctg/feature/3063723/the-future-of-ai-is-not-in-sentient-robots-but-it-might-be-in-gaming [Accessed 11 Apr. 2019].

[4] ITU News. (2018). *How video games can help Artificial Intelligence deliver real-world impact*. [online] Available at: https://news.itu.int/video-games-artificial-intelligence/ [Accessed 03 Apr. 2019].

[5] Roesler, Z. (2019). *Video game AI will change the future of work. Here's how*. [online] World Economic Forum. Available at: https://www.weforum.org/agenda/2019/02/ai-beat-professional-gamers-at-starcraft-ii-here-s-why-that-matters/ [Accessed 10 Apr. 2019].

[6] Walch, K. (2018). *Use of AI in video games boosts playing experience*. [online] SearchEnterpriseAI. Available at: https://searchenterpriseai.techtarget.com/feature/Use-of-AI-in-video-games-boosts-playing-experience [Accessed 10 Apr. 2019].

[7] Lou, H. (2017). *AI in Video Games: Toward a More Intelligent Game [online].* Available at: http://sitn.hms.harvard.edu/flash/2017/ai-video-games-toward-intelligent-game/. [Accessed: 02 Apr. 2019]

[8] Butcher, C. i Griesemer, J., n.d. *Slideplayer.com.* [Online] Available at: https://slideplayer.com/slide/1406867/[Accessed: 28 Mar. 2019].

[9]DeepMind, 2019. *YouTube.com.* [Online] Available at: https://www.youtube.com/watch?v=cUTMhmVh1qs&t= [Accessed: 10 Apr. 2019].

[10] Liang, H., 2019. *Graph searching.* Leicester: De Montfort University.

[11] Liang, H., 2019. *Behaviour tree.* Leicester: De Montfort University.

[12] Liang, H., 2019. *Finite State Machines.* Leicester: De Montfort University.

[13] Liang, H., 2019. *Rule based systems.* Leicester: De Montfort University.

[14] Liang, H. i Radoja, M., 2019. *A* search.* Leicester: De Montfort University.

[15] AlphaStar team, 2019. *DeepMind.com.* [Online] Available at: https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/ [Accessed: 10 Mar. 2019].

[16] Game Maker's Toolkit, 2017. *Youtube.com.* [Online] Available at: https://www.youtube.com/watch?v=9bbhJi0NBkk&t= [Accessed: 27 Mar. 2019].

[17] Unk., 2018. *Bosch.com.* [Online] Available at: https://www.bosch.com/stories/history-of-artificial-intelligence/ [Accessed: 10 Apr. 2019].