TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology

Peter Volostnych 143052IAPB

POTENTIAL FIELD PATHFINDING

Bachelor's thesis

Supervisor: Ago Luberg
MSc
Lecturer

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Pjotr Volostnõh 143052IAPB

TEEKONNA LEIDMINE KASUTADES POTENTSIAALSET VÄLJA

bakalaureusetöö

Juhendaja: Ago Luberg
MSc
Lektor

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Peter Volostnych

[dd.mm.yyyy]

Abstract

This work explores a pathfinding problem in a game scenario where there is a set of agents, a set of static targets that the agents want to reach and a set of moving hazards. A good example of that would be an AI moving units in a Real-Time strategy game. During the course of this work I will explain that convetional algorithms like Dijkstra, BFS, DFS and A* would not be the best course of action because 1) Some of them will need to know which is the closest node to reach and 2) they need to be run for each unit individually since we don't want a situation where we have one path and all the other agents nearby use the same path. Instead I will use a potential field algorithm that fits to this problem really well.

This thesis is written in english and is 29 pages long, including 6 chapters and 21 figures.

Abstract

Teekonna leidmine kasutades potentsiaalset välja

Selle töö eesmärk on analüüsida teekonna leidmise probleemi mängus, milles on mitu erinevat liikuvad agenti, mitu staatilist sihtmärki nin mitu liikuvad ohuobjeti. Agendid proovivad jõuda sihtmärkideni, age peavad vältima ohtusid. Töö kiljeldab, miks traditsioonilised algoritmid nagu A* ja Dijkstra ei sobi selle ülesande lahendamiseks hästi. Töö kirjeldab potentsiaalsete väljade metoodikat ja näitab, kuidas seda kasutades on võimalik ellnevalt kirjeldatud mängus leida hea tulemus.

Antud töö tulemusena on valminud teek, mis realiseerib potentsiaalse väljade algoritmi. Teegi töötamise demonstreerimisekf on loodud simulaator ning tulemuste analüüsiks testimise töörist *(evaluator).* Kõik komponendid on realiseeritud kasutades C++ standardfunktsionaalsust.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 29 leheküljel, 6 peatükki ja 21 joonist.

List of abbreviations and terms

BFS	Breadth-First-Search
DFS	Depth-First-Search
P2P	Point-to-point

Table of Contents

List of Figures

1. Introduction

Pathfinding in general is a problem of finding a path from point A to point B .It is a common and recurring problem in multiple fields of computer science such as logistics[1] - finding the shortest route between cities and military robotics[2] - moving robot-controlled vehicles. There is also a lot of research about pathfinding in games[3][4] since it is an essential part of almost any game[5] - units and every single entity that an AI moves has to calculate a path and more often than not it has to be done in real-time with the resources for computations being very limited meaning that algorithms have to be quite efficient[6].

1.1 Goals

The main goals of this thesis are:

- Build a simulator with a simplistic graphical interface for demonstration purposes that is able to read map information from a text file and choose different algorithms
- Implement a way to evaluate the results of pathfinding algorithms in the simulator
- Create a library with an implementation of a potential field algorithm that is dependent other software as little as possible

1.2 Thesis overview

This thesis has 6 sections:

1. Introduction - Here I state the problem and the goals of this thesis
2. Problem analysis - Information about the problem, choice of algorithm

3. Implementation overview - A bit more technical description of the algorithm used to solve the problem

4. Simulator overview - Description of the simulator

5. Library overview - Very technical description of the solution

6. Summary - a short summary of the solution

2. Problem analysis

In this section I will write about the solution for our problem - why potential field was chosen and what it is.

2.1 Scenario

Instead of basing myself on an abstract scenario I will use a specific scenario in a simplistic game. The scenario is as follows: there are a few goblins (agents) that want to steal treasures (goals) from the dragons (hazards). The dragons don't want that to happen so they are chasing the goblins off. There are also a few walls (obstacles) that neither of them can pass. A more in depth description of this game is presented in section 4.3 of this thesis.

2.2 Algorithm choice

Because of the specifics of the scenario we can't use the most popular pathfinding algorithms. A* can't be used because it is a P2P algorithm[7] meaning it requires us to know *both* ends of the path before we calculate it, at any given point we will only know one. When an agent wants to move it doesn't know which goal to go to exactly, only that it has to be the closest one. When a hazard wants to reach an agent it doesn't care which agent as long as that agent is the closest. This could be solved with algorithms like Dijkstra[8] or BFS[9] but then we will have to calculate them for every single agent separately. With potential field we only need to calculate the field once and then recalculate it only if the goals change. So to put it simply, if we had a situation where we had, say 1 million agents wanting to reach multiple goals all listed algorithms would have to be ran at least once for every single agent, yet with a potential field we only calculate it once and then for every agent we simply look at the cells around and find the one with the smallest value.

2.3 Potential field

The idea of a potential field is taken from nature. For instance a charged particle navigating a magnetic field like in Figure 1, or a small ball rolling in a hill. The idea is that depending on the strength of the field, or the slope of the hill, the particle, or the ball can arrive to the source of the field, the magnet, or the valley in this example. [10]

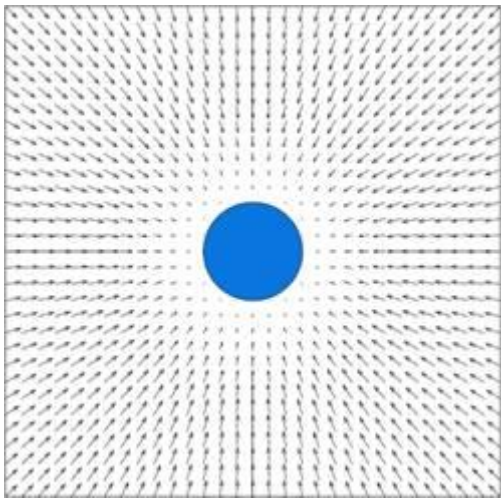

Figure 1: Potential Field concept

This algorithm only cares about the targets that agents have to reach, so the agents themselves do not affect the calculations in any way. To give a short summary on how this algorithm works it is best to imagine a pond over a square grid. Every single goal is like a stone thrown into that pond. All the stones hit the water at the same time and produce waves. Then moving along the waves we assign values to every single cell, incrementing the value as we go further away. So for example the cell with a goal would have a value of 0, then the immediate cells around would have a value of 1 and cells around those cells would have a value of 2 and so on and so on. In case we have multiple goals there will be a few wave sources that will inevitably meet, when that happens the collided waves cancel each other out and do not continue, just like if they have hit a wall. The numbers then represent how far away the cell is from any target and if an agent wants to move to such target all it would have to do is find the cell with lowest (therefore closest) value.

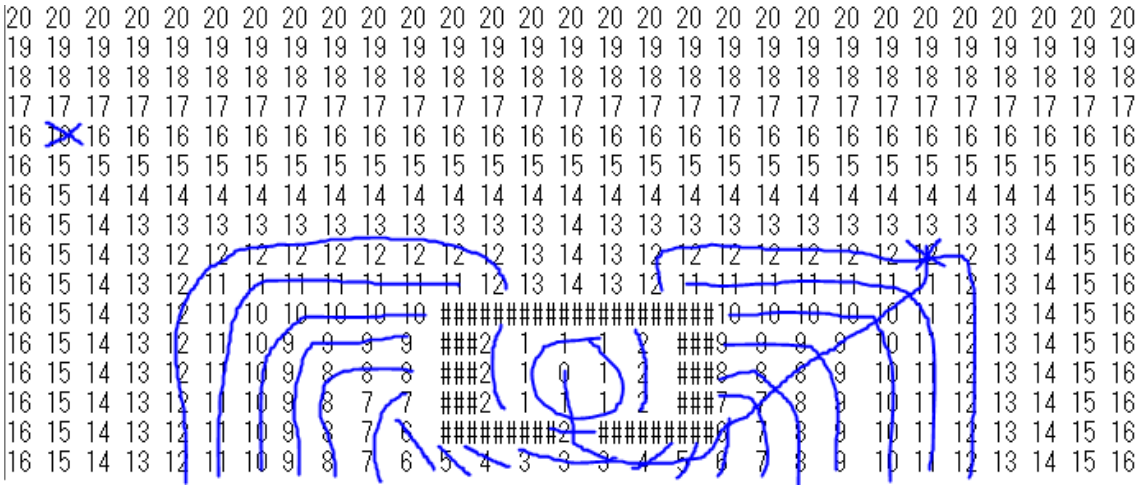

Figure 2: Waves on map "den"

As one can see from an example provided on Figure 2 that is based off a map shown in Figure 3 agents (represented as G for goblins and D for dragons on the map) do not affect the computation of a potential field and when they want to find a path they just have to follow the lowest numbers that will inevitably lead them to a target (represented as T for treasure).

The algorithm works best when there are a lot of agents that we want to find a path for and the goals update very rarely. This is because the potential field does not care about the movement of the agents since it doesn't affect the numbers it holds in any way. If the goal configuration changes then the potential field holds incorrect values and may point to goals that are no longer there so it must be updated which is a computational cost. It is also not particularly effective on bigger maps since it has to process every single cell that targets affect so it means more calculations and more data to store. Therefore it would not be wise to use this algorithm in a situation where we have a huge map and/or where goals change very frequently.

Depending on the situation a potential field might save vectors that point to the closest goal instead of numbers that show how far away the goal is. In this thesis, we use the latter.

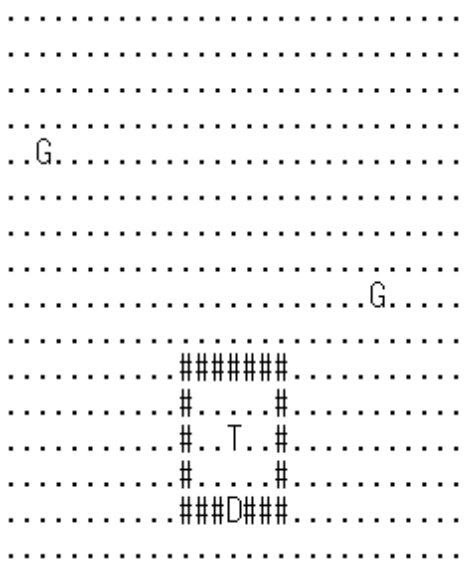

Figure 3: Simulator view of the map "den"

3. Implementation overview

The game simulation utilizes 3 maps to move units. 2 maps for goblins - target map to find treasures and escape map to escape from the dragons, and 1 map for dragons - dragon map to find goblins. Every single map is just a 2D array that holds numbers which represent how far away closest goal is from this cell. All the AI has to do is to look at the neighbouring nodes that an agent can reach and pick the one with the lowest value since it's the closest to the goal. AI will also have to take care of updating the map whenever goals change(goblin reaches treasure/dragon eats a goblin/goblin moves and so on). Maps have to be updated because they might point to goals that are no longer there.

3.1 Target and Dragon map

```
Assume a 2D array of numbers MAP initialized at -INF
Find all targets there are on a map and save their locations to set of positions CURRENT
Assume empty sets of positions BANNED and NEXT
Assume a number VALUE initialized at 0
while CURRENT is not empty
        for every node NODE in CURRENT
                add NODE to BANNED
                if the NODE is an obstacle, discard it returning back to the loop
                set MAP's value at NODE's position to VALUE
                get all nodes around NODE and add them to NEXT
        set CURRENT to NEXT
        clear NEXT
        increment VALUE by 1
return MAP
```

Figure 4: Target map algorithm

As shown in Figure 4 this algorithm tries to find all the treasures and sets the value at their location to 0. Then it gets all the cells around the ones it had and sets their values to 1, and so on and so forth generating a 2D array of numbers that represent how far is a node from the closest disturbance. An example of a potential field for the map shown in Figure 3 is provided in Figure 5 and Figure 6. The first figure shows the potential field map for dragons who have goblins as their goals and the second one shows the map for goblins who have treasures as their goals. The map also takes walls and such obstacles in regard so agents won't be banging their heads against the wall that has a treasure behind it but instead walk around them to reach their destination

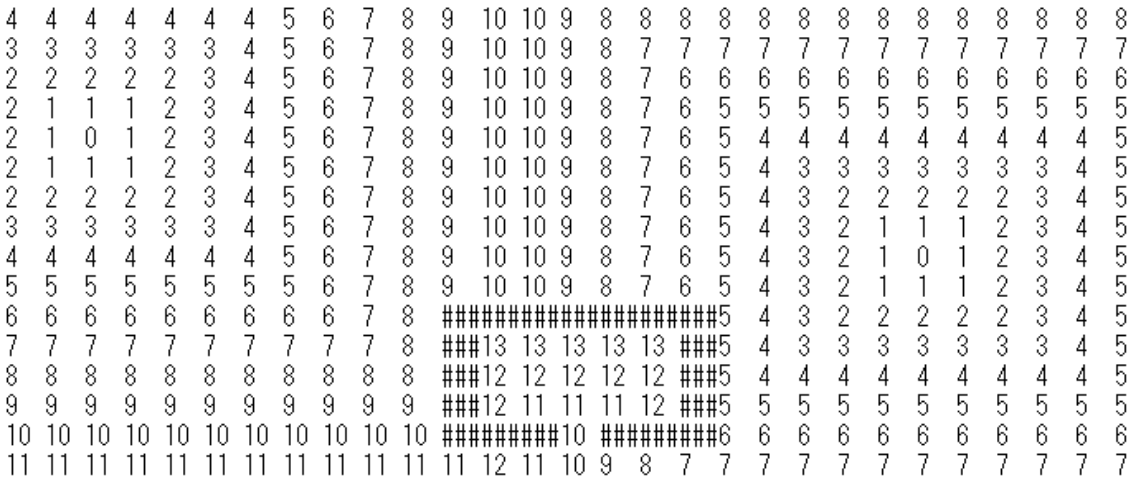

Figure 5: Dragon potential field of the map "den"

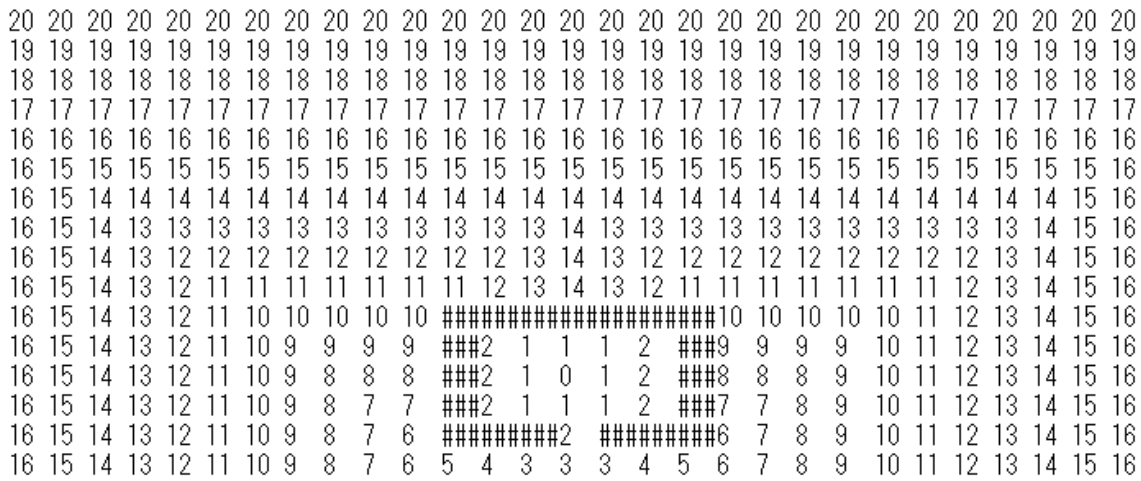

Figure 6: Target potential field of the map "den"

3.1.1 Avoidance map

Assume a 2D array of numbers MAP initialized at -INF

Find all hazards there are on a map and save their locations to set of positions CURRENT

Assume empty sets of positions BANNED and NEXT

Assume a number VALUE initialized at 0

while CURRENT is not empty

for every node NODE in CURRENT

add NODE to BANNED

if the NODE is an obstacle, discard it returning back to the loop

```
                set MAP's value at NODE's position to VALUE
                get all nodes around NODE and add them to NEXT
        set CURRENT to NEXT
        clear NEXT
        decrement VALUE by 1
return MAP
```

Figure 7: Avoidance map algorithm

As one can see from Figure 7 this algorithm is very similar to the algorithm shown in Figure 4 with the only exception being that the numbers decrease instead of increasing. The numbers decrease so that we can keep our turn decision function that takes the lowest value possible instead of creating another one that would look for the highest one instead. An example is provided in Figure 8, the !!! in the centre means treasure, it is set to a very big negative number so that when a goblin escapes it would prefer cells with treasures over others if it happens to pass by them.

```
-18-18-18-18-18-18-18-18-18-18-18-18-18-18-18-18-18-18-18-18-18-18-18-18-18-18-18-18-18-18-18-18-18
-17-17-17-17-17-17-17-17-17-17-17-17-17-17-17-17-17-17-17-17-17-17-17-17-17-17-17-17-17-17-17-17-17
-16-16-16-16-16-16-16-16-16-16-16-16-16-16-16-16-16-16-16-16-16-16-16-16-16-16-16-16-16-16-16-16-16
-15-15-15-15-15-15-15-15-15-15-15-15-15-15-15-15-15-15-15-15-15-15-15-15-15-15-15-15-15-15-15-15-15
-14-14-14-14-14-14-14-14-14-14-14-14-14-14-14-14-14-14-14-14-14-14-14-14-14-14-14-14-14-14-14-14-14
-14-13-13-13-13-13-13-13-13-13-13-13-13-13-13-13-13-13-13-13-13-13-13-13-13-13-13-13-13-13-13-13-14
-14-13-12-12-12-12-12-12-12-12-12-12-12-12-12-12-12-12-12-12-12-12-12-12-12-12-12-12-12-12-13-13-14
-14-13-12-11-11-11-11-11-11-11-11-11-11-12-11-11-11-11-11-11-11-11-11-11-11-12-13-13-14
-14-13-12-11-10-10-10-10-10-10-10-10-11-12-11-10-10-10-10-10-10-10-10-10-11-12-13-13-14
-14-13-12-11-10-9 -9 -9 -9 -9 -9 -10-11-12-11-10-9 -9 -9 -9 -9 -9 -10-11-12-13-13-14
-14-13-12-11-10-9 -8 -8 -8 -8 -8 ######################-8 -8 -8 -8 -8 -9 -10-11-12-13-13-14
-14-13-12-11-10-9 -8 -7 -7 -7 -7 ###-3 -3 -3 -3 -3 ###-7 -7 -7 -7 -8 -9 -10-11-12-13-13-14
-14-13-12-11-10-9 -8 -7 -6 -6 -6 ###-2 -2 !!!-2 -2 ###-6 -6 -6 -7 -8 -9 -10-11-12-13-13-14
-14-13-12-11-10-9 -8 -7 -6 -5 -5 ###-2 -1 -1 -1 -2 ###-5 -5 -6 -7 -8 -9 -10-11-12-13-13-14
-14-13-12-11-10-9 -8 -7 -6 -5 -4 ##########0  ##########-4 -5 -6 -7 -8 -9 -10-11-12-13-13-14
-14-13-12-11-10-9 -8 -7 -6 -5 -4 -3 -2 -1 -1 -1 -2 -3 -4 -5 -6 -7 -8 -9 -10-11-12-13-13-14
```

Figure 8: Escape potential field of the map "den"

4. Simulator overview

For the sake of simplicity and to make it easier to get the point across I will use a simulator that depicts a simplistic game utilizing potential field for pathfinding. The game depicts a set of goblins (G) trying to steal treasures (T) from dragons (D) while the dragons are trying to hunt goblins down. Empty cells are represented with a dot (.) and walls are represented with a grid (#). Rules of the simulator are explained in the rules section. The simulator is a part of a program that comes with 2 main modes that alternate between each other - Main Menu and Simulator.

4.1 Main Menu

Main purpose of this mode is to let the user define the rules for the simulator. It is essentially a configuration screen to avoid configuration files. It has a simplistic work behaviour that can be described with a flowchart provided in Figure 9

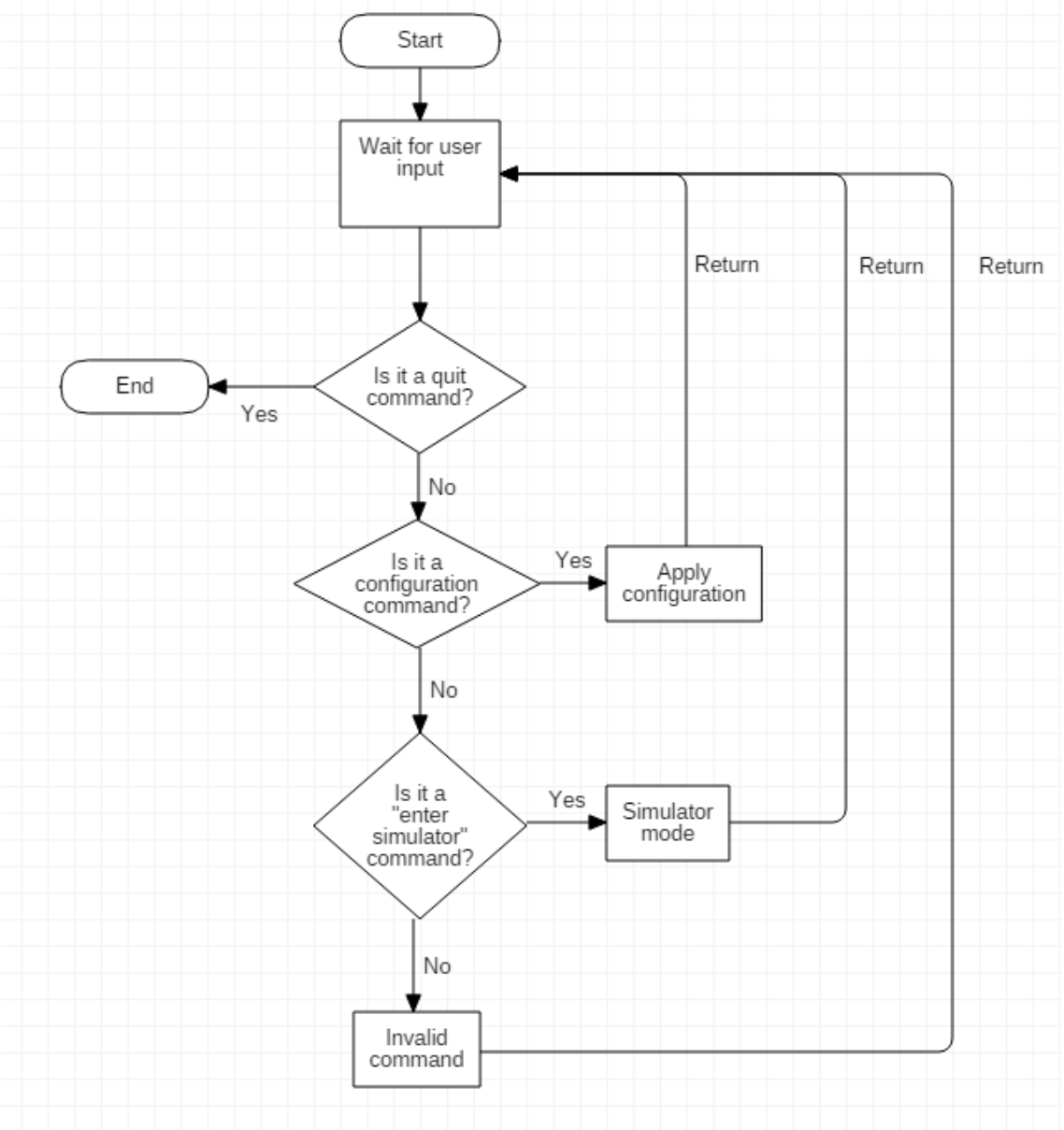

Figure 9: Main Menu Flowchart

It has a few commands that are available to the user:

- q/quit/exit - quits the program.
- h/help - prints out the list of all commands that are available in this mode
- o/open/m/map <mapname> - tries to load the map called <mapname>, if no such map exists then no map is loaded.
- a/algo - sets the algorithm for the simulator.

- g/game - moves the user to the simulator mode
- fp/firstparam <number> - sets the first parameter to <number> if possible, if not then the first parameter is not changed.
- ev/eval <eval> - sets the evaluation to <eval> if it exists
- qev/quickeval <eval> - sets the evaluation to <eval> and does a quick simulation of said evaluation
- aev/autoeval - runs qev command for all available evaluations

4.2 Simulator mode

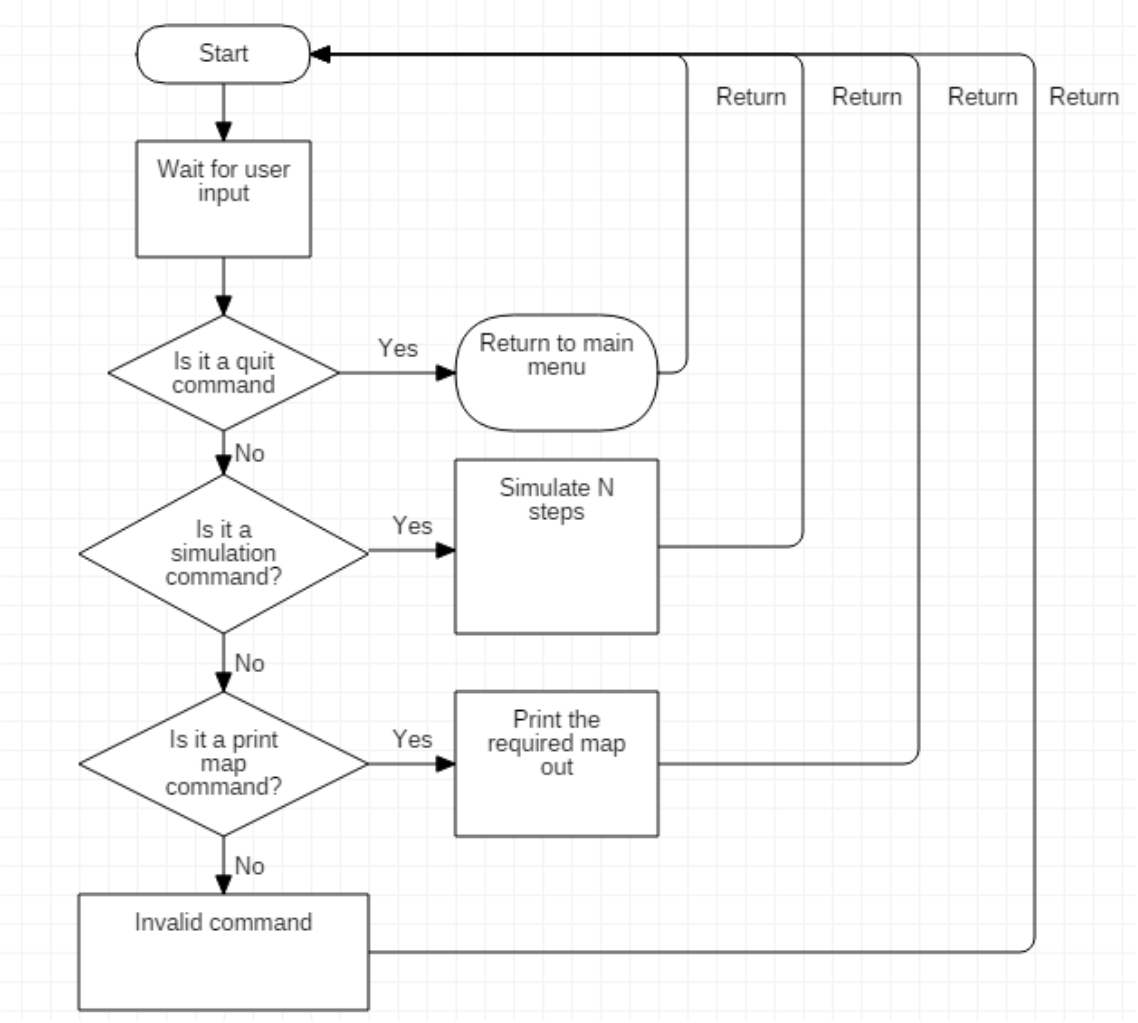

Figure 10: Simulator Flowchart

The simulator can only be entered from the Main Menu mode and may not quit the command, when it's done it returns control to the main menu as seen in Figure 10. The simulator also has a set of commands that are available to the user:

- q/quit/exit - quits the simulator and puts the user back to the main menu
- h/help - prints out all the commands
- qs/qsim N - Simulates N steps without printing out all the steps in between, only the last step is printed out.
- s/sim N - Simulates N steps and prints every single simulation step out
- p/print <g/goblin/d/dragon/e/escape> - prints out the required potential field

4.3 Simulator rules

Simulator is based upon the following rules:

- The map is a 2D array of cells
- Cells can have one value out of 5 total: goblin (G), dragon (D), empty (.), wall (#) and treasure (T)
- There are only 2 entities that can move: goblins and dragons
- The simulation is turn based, goblins move first then the dragons
- All entities may only move once each turn
- All entities can only move in a 1 cell radius each turn (so they have 9 cells they can move to - 8 around them and the one they are standing on)
- All entities may not move onto walls
- All entities may not move onto the cells with the same values (goblins can't eat goblins and dragons can't eat dragons)
- Dragons may not move onto treasures

- If a goblin moves onto a treasure that treasure is removed
- If a dragon moves onto a goblin or a goblin moves onto a dragon that goblin is removed
- Trying to do an invalid move results in that move being ignored and the entity not moving

4.4 Evaluation mode

Evaluation is simply an addon onto the simulator mode that gives some sense of benchmark. Each evaluation holds 3 numbers - number of steps, number of alive goblins and number of treasures. The number of alive goblins and present treasures is implied to be a "normal" outcome of the simulator running for the specified amount of turns. Quick evaluation does the required amount of steps and then dumps the differences in amount of goblins and treasures. It is somewhat hard to say automatically which result is better than the other because 2 values may vary. For example one algorithm may get more treasures than the other one while losing more goblins. Depending on the requirements for the game that may be good or bad - good if we want goblins to get as many treasures as possible regardless of the cost but bad if we want to have as many goblins survive as possible.

4.5 Input/Output

The simulator assumes that there are 2 folders where the main executable is located: mapfiles and eval - both containing an index.txt file that simply lists all the available files in those directories, each one on it's own line, no extension.

4.5.1 Evals

Each eval file must be in the eval folder and have an extension of .txt. The program only reads the first line and assumes that there are 4 fields in order: name of the map, turns, amount of alive goblins and amount of present treasures. In case the formatting is wrong the file is considered invalid and is ignored.

4.5.2 Maps

Each map file must be in the mapfiles folder and have an extension of .txt. The map file must simply contain the map represented by the allowed characters separated by a newline, in case the character is not permitted or the map is not a valid rectangle unspecified cells are set to EMPTY. If we take map shown in Figure 3 as an example the file must contain 16 lines of text each line containing 29 characters. The characters must be exactly the same as shown in the figure

5. Library overview

The library is written in C++11, GCC version 4.9.2, no dependencies and consists of a single header file that specifies the potential field class named PFMap as a template based around a map cell class. Since the library is only one header file simply including it in a project is enough.

5.1 Typenames

PFMap uses a few custom datatypes, Figure 11 shows one outside the class - just a shortening for an unsigned int datatype and Figure 12 shows the ones inside - I used std::vector for storing cells to keep myself within the standard library and avoid memory allocation problems that normal arrays have, there is also a shortening for a position variable and a definition of a pointer. I also used one enumeration presented in Figure 13 to avoid hard coding weird numbers to behaviours based on cell values.

```
typedef unsigned int ui; //just so that I don't have to spell out unsigned int
```

Figure 11: typedef in PFMap

```
using cellptr = T; //T is the template typename, preferably a pointer to ease updating
using lineVec = std::vector<PFMap::cellptr>; //An array for a single line in a map
using dataVec = std::vector<PFMap::lineVec>; //2D array representing the map


using vlineVec = std::vector<int>; //A line of integers for MAP
using vdataVec = std::vector<PFMap::vlineVec>; //2D array of ints
```

```
using Pos = std::pair<unsigned int, unsigned int>; //2D position


using ptr = std::shared_ptr<PFMap>; //Pointer type for our PFMap
```

Figure 12: PFMap types

```
enum CELLTYPE
{
	WALL = 0, //Walls, unpassable
	IGNORE, //Allies, other units that we can't move onto but they don't influence the map
	DANGER, //Dangerous tiles, needed in avoidance map, otherwise treated as IGNORE
	GOAL, //Places we want to reach
	EMPTY, //Empty tiles that we can move onto
};
```

Figure 13: PFMap enumeration

5.2 Initialization

```
PFMap(PFMap::dataVec map, std::function<PFMap::CELLTYPE(PFMap::cellptr)> _heur, bool _avoid = false)
```

Figure 14: PFMap constructor

As shown in Figure 14 to create a PFMap one would need to provide 2 arguments with an optional third one. The arguments are:

- PFMap::dataVec - a 2D array that describes the map, a cell should be a pointer if one aims to update the map

- std::function<PFMap::CELLTYPE(PFMap::cellptr)> - a function that allows us to convert PFMap::cellptr (a cell of a given map) to a PFMap::CELLTYPE enumeration described above that tells the map how it should treat the given cell

- bool - an optional boolean value signalling whether this is an escape map or not defaulted to a false value

5.3 Usage

The map has 7 public methods that allow it's basic usage

5.3.1 void update()

This is a simple function that takes no arguments and returns nothing, used to update the map. This function should be called by the game when a change of targets is detected.

5.3.2 Pos getNextPos(Pos current, ui lookRad = 0)

This function takes 2 arguments: a position and an optional radius. This function looks at the 8 cells surrounding the current cell and return the one with the lowest value. If a radius greater than 0 is supplied then neighbour's value is represented by the lowest value around the neighbour in the radius.

5.3.3 bool isGoal(Pos pos)

Returns true if value at pos is equal to CELLTYPE::GOAL

5.3.4 std::vector<Pos> getAround(Pos pos, int rad = 1)

Returns all the cells around pos in a rad radius ignoring the walls

5.3.5 std::vector<Pos> getFillAround(Pos pos, int rad = 1)

Returns all the cells that pos could reach in rad steps.

5.3.6 vdataVec getVData()

Returns the potential field - 2D array of integers

5.3.7 int getValue(Pos pos)

Returns the value at pos

5.4 Problems

The way the algorithm is implemented causes certain problems to occur which will be discussed in the following subsections

5.4.1 Escape decision

Escaping is not particularly effective with this algorithm but that's also due to the fact that it's impossible to escape the dragon forever, it will eventually catch up because of the way the game is laid out, unless map plays in favour of goblin and provides loops or encloses goblins away from dragons

A good example of this problem is presented on Figure 15 and Figure 16. A goblin is escaping from a dragon and comes to a crossroad. It can either go up or down. Going up would end up in it's demise, but at the point of choice it doesn't exactly know that as both going up and down have values of -3. This problem can be solved by looking at the 4 next neighbours of those cells. Then the value above will end up at -6 yet the value below will end up at -7 resulting in goblin going down to safety.

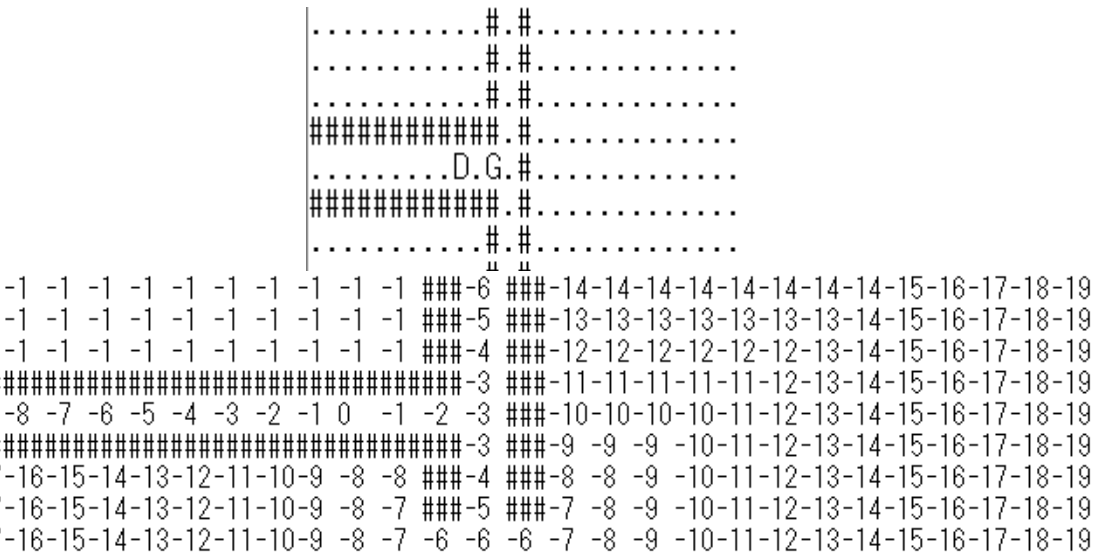

Figure 16: Escape map view of the map tonnel

5.4.2 Corner fondness during escape

The algorithm seems to be fond of corners even though more often that not they are effectively a very bad choice. If we take a scenario presented in Figure 17 one can easily see how the goblin can keep on going in circles with the dragon forever and ever as long as they both skip corners and opt for the diagonal instead. That doesn't happen however because as we can see in Figure 18 and Figure 19 both the corner itself and the diagonal path that skips the corner weigh the same so as far as the decision making is concerned both are equally "good" even though choosing the corner makes the distance

between dragon and the goblin shorter. This particular scenario ends with dragon catching up to the goblin in 29 turns instead of chasing after him forever. The dragon is not affected by this problem as one can see in Figure 20 and Figure 21.

```
.....
.###.
D###G
.###.
.....
```

Figure 17: Simulator view of the map "circle"

```
.....
G###.
.###.
.###.
..D..
```

Figure 18: Simulator view of the map "circle" after 9 turns

```
-5 -5 -6 -5 -5
-4 #########-4
-3 ###-1 ###-3
-2 #########-2
-2 -1 0  -1 -2
```

Figure 19: Escape map of the map "circle" after 9 turns

```
G....
.###.
.###.
.###.
.D...
```

Figure 20: Simulator view of the map "circle" after 10 turns

```
0  1  2  3  4
1  #########4
2  ###-1 ###5
3  #########6
4  4  5  6  7
```

Figure 21: Dragon map of the map "circle" after 10 turns

6. Summary

During the course of this work I managed to implement a simulator with a configuration screen that has a few options and a simplistic evaluation mode without using any 3rd party libraries on pure C++11. The simulator uses an implementation of a potential field algorithm to move the units around and works decently. The implementation of the potential field was done in a way that allows it to be separated from the main simulator code into a stand-alone single-file library that doesn't even need a .dll to function.

The goals of this thesis have been achieved - a simulator has been built, the pathfinding algorithms can be evaluated both manually with the usage of a simulator and with a simple evaluation tool and the potential field algorithm implementation is a library that doesn't need the simulator to operate.

There a lot of ways that this work could be improved: a better GUI could be implement and would make this an overall better tool, more pathfinding algorithms could be added to the simulator, choice which algorithms to use for escape, dragon and goblin targetting could be separated, the evaluation tool could measure more data (such as time) about the algorithms' performances. The implementation of the potential field could use more testing and debugging not to mention optimizing it for bigger maps and maybe finding a way to keep some of the previous values during an update.

References

1: Xiaoge Zhang, Zili Zhang, Yajuan Zhang, Daijun Wei, Yong Deng, Route selection for emergency logistics management: A bio-inspired algorithm,
2: A.M. Mora, J.J. Merelo, C. Millan, J. Torrecillas, J.L.J. Laredo and P.A. Castillo, Enhancing a MOACO for Solving the Bi-Criteria Pathfinding Problem for a Military Unit in a Realistic Battlefield,
3: Vadim Bulitko, Yngvi Bjornsson, Ramon Lawrence, Case-Based Subgoaling in Real-Time Heuristic Search for Video Game Pathfinding,
4: Ramon Lawrence, Vadim Bulitko, Database-Driven Real-Time Heuristic Search in Video-Game Pathfinding,
5: Johan Hagelback, Potential-Field Based navigation in StarCraft,
6: Adi Botea, Martin Muller, Jonathan Schaeffe, Near Optimal Hierarchical Path-Finding,
7: Xiao Cui and Hao Shi, A*-based Pathfinding in Modern Computer Games ,
8: E. W. Dijkstra, A Note on Two Problems in Connexion with Graphs, 1959
9: Pat Morin, Open Data Algorithms,
10: Hani Safadi, Local Path Planning Using Virtual Potential Field, 2007

Appendix

This work as well as the the simulator sources are available from https://gitlab.cs.ttu.ee/Pjotr.Volostnoh/thesis.git. The library consists of 1 header file 380 lines long. Simulator + Library consist of 11 + 1 header files, 698 (318 + 380) lines and 11 source files, 1227 lines.