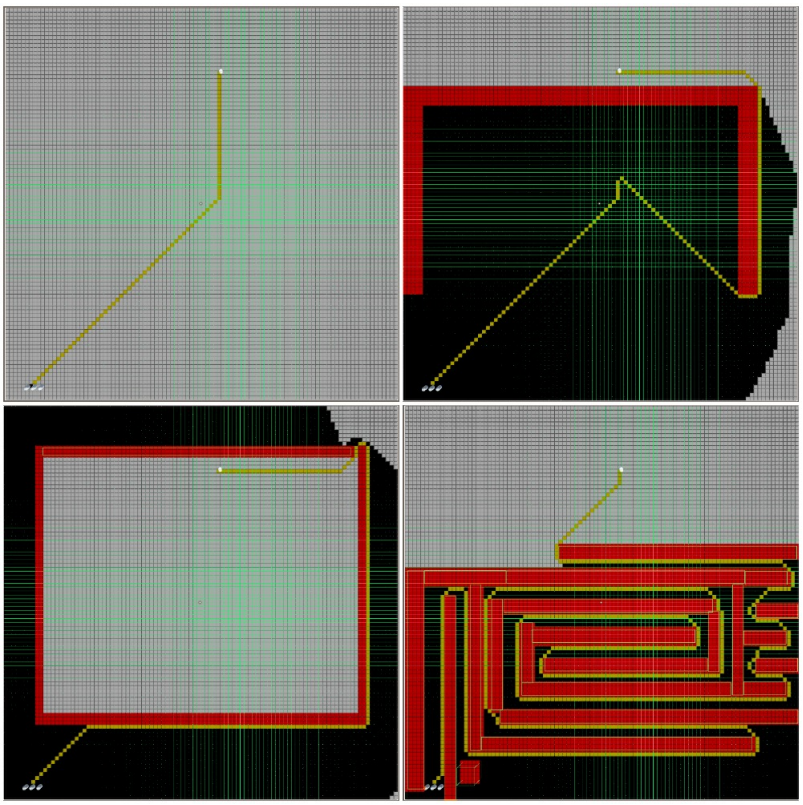
Pathfinding Algorithms in a Unity 3D Environment



Roskilde University
IMT
Humanistic Technology
Computer Science Project

Title: Pathfinding Algorithms in a Unity 3D Environment

Project Period: Spring Semester 2018

Project Group: S1825228105

Members:

Andreas O. Thomsen
Study number: 58650
E-mail: aobelt@ruc.dk

Sebastian A. V. Jakobsen
Study number: 57938
E-mail: savj@ruc.dk

Alex T. K. Wogelius
Study number 58717
E-mail: atkw@ruc.dk

Pelle Schlebaum
Study number: 57306
E-mail: pellesc@ruc.dk

Supervisor: Junia P. G. Silva

Number of pages: 60

Hand in date: 28/5/2018

Abstract

The aim of this report is to test path finding algorithms in four different environments - one with no obstacles, one with a maze, one with a barn shaped obstacle and one with the target enclosed with one entrance. The algorithms under examination is modified versions of respectively breadth first search, dijkstra and A*. furthermore we have implemented a minimal heap based version of A* and Dijkstra, also to compare them. The area of the search is grid based and can therefore be seen as nodes connected with eight other nodes. We use unity game engine as platform, because it provides tools to make good visualizations, and a build-in profiler which, combined with a series of tests of each algorithm in the different environments gives a picture of their completion time. The tests shows, that in a plain grid the heap optimized A* was 4.925 milliseconds faster than the slowest which was breadth first search. In the maze, the heap optimized dijkstra implementation were 1.848 milliseconds faster than the slowest which again were breadth first search. With the barn shaped obstacle the dijkstra without heap optimization was the fastest and 1.235 milliseconds faster than the heap optimized Dijkstra which were the slowest. The test of the enclosed target showed that breadth fist search were the fastest, and 0.974 milliseconds faster than unoptimized A* which were the slowest.

Keywords: Unity, Dijkstra, A*, Breadth first search, pathfinding, algorithms, optimization.

Contents

1 Introduction

This computer science project started out with the intention to research, design, and implement various types of algorithms. "Algorithms are the stuff of computer science: they are central objects of study in the field." (Sedgewick 2011, pp. 4) . Algorithms are present everywhere in today's society from nuclear power plants to coffee machines and in route planners like google maps (Bhasin 2015).

Algorithms are clearly an important field to have an understanding about when one is writing a program. Researching which sort of algorithms and how one should write them can be a complex problem, but the essential goal when writing algorithms is, to minimize computational calculations and memory. Sedgewick and Kayne talk about how upgrading your computer might increase a programs speed by a factor of 10 or 100, but a good algorithm can speed a program up with a higher factor (Sedgewick 2011). This is of course much more economical feasible than upgrading all the current hardware.

It's no secret that the goal intended was to gain enough knowledge about algorithms and how to implement them, in the hope of being able to apply them to NPC's in a 2D video game. This is because all of the group members have an interest in developing computer games, and all have had a course revolving the concepts of basic AI. It seemed like an obvious way to approach this project because we read about algorithms regarding basic AI using the A* algorithm (Sedgewick 2011). But as timed passed we realised that the scope for making a game was too big for us being satisfied with the end result of the semester project and decided that we didn't have the means to accomplishing the idea within one semester.

The scope of this project is about understanding and designing algorithms, but

also about the optimization methods that exists. Finally, the project aims at comparing different algorithms to present test results, so it will be possible to do a comparative evaluation. The algorithms are being programmed in C# through Microsoft Visual Studio which is what is used by Unity so we can get a visualization of the execution of the different algorithms.

1.1 Problem framing

As the scope of the project changed throughout the process, the problem frame naturally changed as well. The first approach remained the same and was about getting to know some classic algorithms, and what components and methods they entail. Then the question arose of where and how in society algorithms are used and what purpose they have. That's where the understanding of the power of algorithms started forming, when it became clear what computational processes, they can save. After exploring different algorithms and understanding the theory behind them, it was clear that the projects scope should be on the theory behind design-techniques and implementation of pathfinding algorithms. The preferred algorithms will be implemented, and some sort of benchmarking test needs to take place. That's another problem, understanding what benchmarking entails and how it should take place. This brings forward additional questions like what parameters need to be set and what criteria needs to be met, to have a benchmark test that produces some results that can be evaluated properly.

1.1.1 Problem definition

How do different path finding algorithms perform in different environments and how do they compare to each other?

2 Theory

In this chapter the concept of three classic algorithms will be presented and their strengths and weaknesses will be evaluated, in the scope of which problems they could be best suited to solve. There will also be a presentation of different optimization methods that can be implemented in an algorithm to save computational time.

2.1 Algorithms

In this section various algorithms will be explored and discussed in relation to each other. This will be done through mathematical representations and graphs. The reasoning behind the algorithms presented in this section is done through graphs, because as Harsh Bhasin puts it "(...) graphs which is the soul of algorithm analysis and design." (Bhasin 2015, pp. 142) . Algorithms can be visualized through many types of graphs. The ones referred to in this chapter will mainly be cyclic and non-cyclic graphs, the latter also known as trees, is when a node (vertex) is not isolated (Bhasin 2015). Graphs can also be a matrix but regarding to this project which has a focus on pathfinding, we will mostly refer to cyclic or non-cyclic weighted graphs. Weighted means there are some costs associated to the nodes or edges in the graph.

An algorithm can be viewed as a series of tasks that needs to be fulfilled to complete an objective (Bhasin 2015). Algorithms are everywhere and as Harsh Bhasin also notes, they exist within everything from coffee-makers to power plants as well as in our search engines and, as most people know, in google maps as a tool for getting directions which are happening due to the pathfinding algorithms

(Bhasin 2015). The pathfinding algorithms we aim to work with are Dijkstras(1959), breadth first search(1945), and A* pronounced "A-Star"(1968), and we will try to implement them to compare them up against each other (*Timeline of algorithms* 2018).

2.2 Pathfinding

In essence, pathfinding is about finding a route from point A to point B. The way to handle this varies from checking all possible directions until point B has been located, to providing search parameters which serves to guide the pathfinder in order to avoid checking all available moves and instead, do a focused search that will, in theory, save computational power and therefore also work faster. “A good algorithm should use the resources such as CPU usage, time, and memory judiciously.” (Bhasin 2015, pp. 2)

2.2.1 Breadth First Search

Harsh Bhasin introduces us to breadth first search and depth first search algorithms through talking about graphs and graph traversals, hence graph traversal can be either BFS or DFS (Bhasin 2015).

The breadth first search algorithm is checking the neighbouring nodes from the root node. It implements a data structure called queue. The queue is used when a node is processed, then the adjacent nodes are placed in a queue (Bhasin 2015).

The queue used in the breadth first search algorithm is an array in order to keep track of the visited nodes - this is a global array (Bhasin 2015). As mentioned above, the breadth first search algorithm checks its neighbouring nodes but the algorithm goes through all the nodes, as going through layers of neighbours. The algorithm starts by selecting the root node and then putting it in the queue. Then "A" is processed and we see that "B" and "C" are adjacent(neighbours) to "A", so now they will be put in the queue and so forth. The reason for the array to keep checking the visited nodes is because, when dealing with graphs as a representa-

tion, then "D" and "A" could be connected but we don't want to process "A" again, when we get to "D". Meaning you can get from "A" to "D", but "D" isn't adjacent to "A" otherwise it would have been put in queue with "B" and "C".

As mentioned breadth first search checks through layers of neighbours and this is its strength. But if the desired node one wants to reach is on the seventh layer and there are for example 256 nodes on each layer, then it has to process 256 nodes on each layer before getting to the desired node (Bhasin 2015).

Following here is a graphical representation where we want to get to the "I" node.

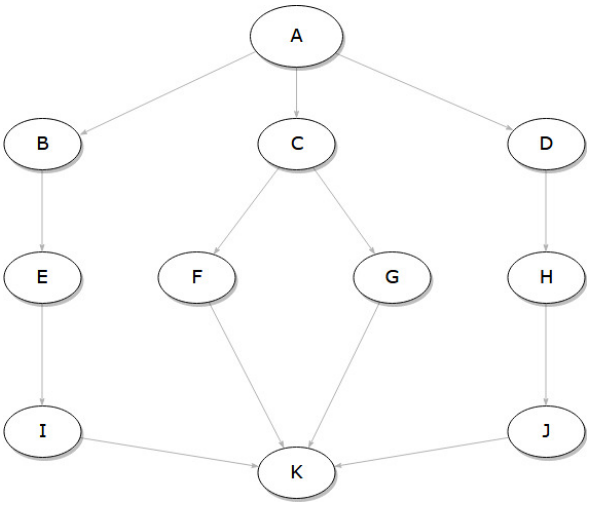


Figure 2.1: Graph/Map

What we see here is a graph, it could represent a road map, where we at our root node "A", wants to visit some family at node "I".

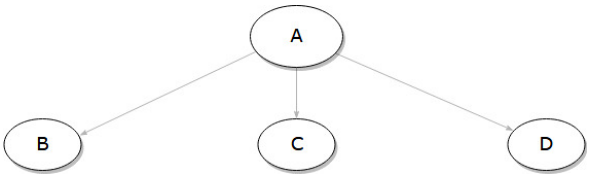


Figure 2.2: Graph/Map

As mentioned in the theory section BFS will search through the adjacent nodes in the next layer. As seen on figure 2.2 it finds "B", "C", and "D".

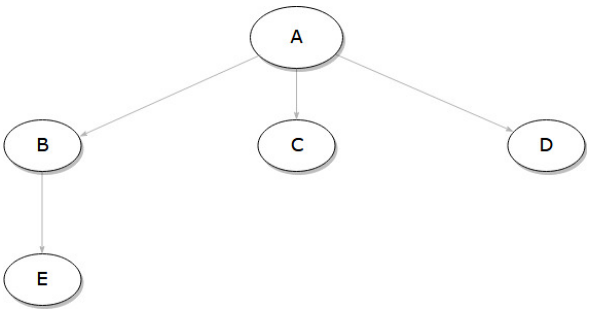


Figure 2.3: Graph/Map

Now the nodes adjacent to "A" have been processed so the algorithm checks the adjacent nodes to "B", which is "E".

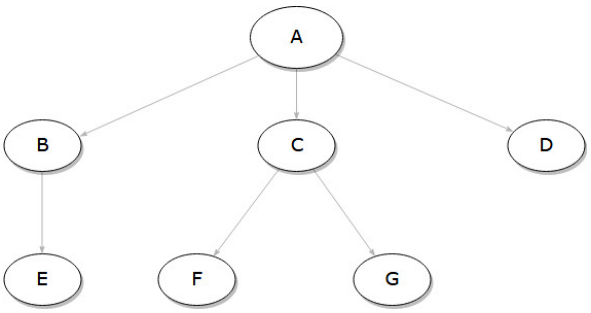


Figure 2.4: Graph/Map

Since the only adjacent node to "B" was "E", we go back to "C" and process the adjacent nodes to "C", which is "F" and "G".

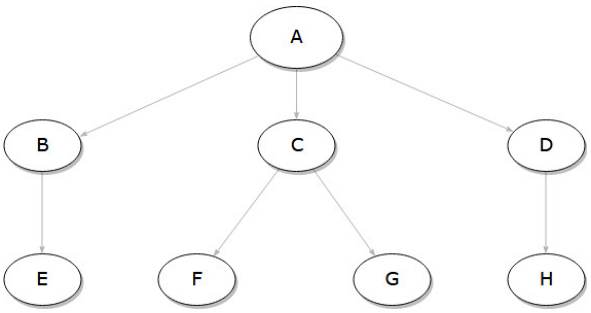


Figure 2.5: Graph/Map

In figure 2.5 the algorithm is back at the "D" node and are processing the adjacent nodes to "D", which is the "H" node.

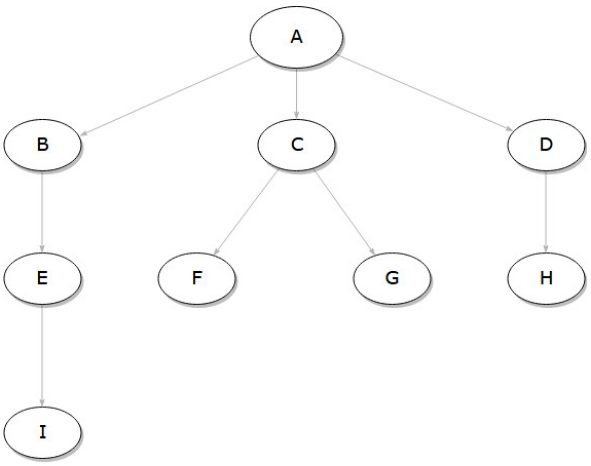


Figure 2.6: Graph/Map

Finally when the nodes on layer 1 "B", "C" and "D" has been processed and the ones on layer 2 "E", "F", "G" and "H", the algorithm goes to the "E" node and processes the ones adjacent to "E". Now the algorithm has made it to "I". That's how breadth first search executes in theory.

2.2.2 Dijkstra

Dijkstra's algorithm is a single-source shortest path algorithm which was invented to solve "the single-source shortest-path problem in edge-weighted digraphs with non-negative weights." (Sedgewick 2011, pp. 652) dijkstra's algorithm is useful for finding the shortest path or the lowest cost path through a digraph when the weights are non negative. An edge-weighted digraph has a cost or weight connected to a directed path. Regarding the cost/weights and the nodes, the weights do not represent the distance between edges. They are Euclidean distances (Sedgewick 2011). The reason for this is important because dijkstra's algorithm uses the weights from the edges to get to a node, to find the minimal cost path i.e. the shortest path. The quality of dijkstra is that it's able to build a shortest-paths tree where it's possible to find the shortest path from the source node to any given node (Sedgewick 2011).

When applying dijkstra's algorithm to an edge-weighted graph it's possible to apply a greedy approach to finding the shortest path, which aims at either minimizing costs or maximizing profit (Bhasin 2015). The greedy approach on edge-weighted graphs work by using heuristics from the edges when calculating the shortest path between two nodes.

The guide for using the greedy approach is starting from a source node and then selecting a node as the goal, then the algorithm will find the path with the lowest cost.

2.2.3 A*

The A* algorithm finds its path by maintaining two lists. An open list and a closed list described further below. Also it has a set of heuristics which helps determine the direction of its search towards its end-goal. This for an example, separates it from dijkstra's algorithm which searches in all directions until it reaches its end-goal and are therefore - in theory faster.

> This algorithm is an extension of dijkstra's algorithm, which aims to improve upon the running time using heuristics. dijkstra's algorithm was in some sense blindly searching the graph for the goal node. It did not use the location of the goal node to guide its search, which meant it would search with equal priority in all directions.
>
> Vinther, Vinther, and Afshani 2015, pp. 18

Open-list

The purpose of the open-list is to store all the neighbouring nodes of the current node under investigation. As the search expands, more neighbours will be added but to the end of the list. With sufficient heuristics, we can loop through the open-list, and take the "closest" neighbour and make that one the next current node / "node of interest". (Oksa 2014)

Closed-list

The closed-list serves as a back stopper meaning, that its purpose is to add the nodes that have already been investigated so - when a loop through neighbours of the current node is initiated, and other nodes have already been examined, they will be skipped.

Heuristics

The heuristics are the logic the A* follows in order to generate the shortest path. The heuristics are described as:

$$F(n) = G(n) + H(n) \tag{2.1}$$

The cost of a move

Moving horizontally or vertically have a cost of 1 and moving diagonally have a cost of 1.4. The reason for choosing 1.4 for the diagonal move is that we consider our map, and the nodes it consists of, as angles of 90 degrees. Therefore to calculate the diagonal movement we need to find "C" in Pythagoras theorem. As it is - the theorem says that "C" to the power of 2 is equal to "A" to the power of 2 plus B to the power of 2.

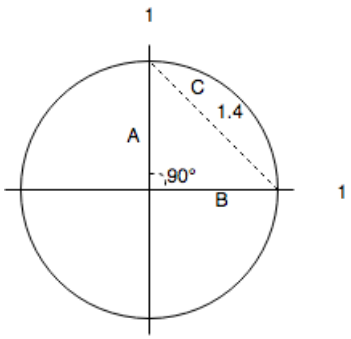


Figure 2.7: Unit Circle

since our movement cost to go along either "A" or "B" is 1, then those two powered by 2 would be 2. Now to find the square root of 2, ending up at 1.414213. By a principle of convention we multiply these costs by 10. So to simplify it, we choose to make the diagonal movement cost 14 and the cost of both vertical and horizontal moves 10.

beneath we will describe F-, G- and H-cost individually.

G cost

The G cost is the cost from any given node to the starting node. As we move further away from the starting node this number will grow according to horizontal, vertical and diagonal costs.

H cost

The H cost is the distance from any given node to the end node, and therefore the H cost equips the algorithm with a kind of compass telling it which way to search. As we get closer to the end node, the H cost will get smaller and smaller according to the horizontal, vertical and diagonal costs.

F cost

F is the combined cost of the H- and G cost. The F cost gives the A* algorithm its potential to find the shortest path without searching the entire area. So with the right heuristics A* works really well and thus are flexible because only with slight changes to the heuristics you can change and modify the search criteria (Vinther, Vinther, and Afshani 2015, pp. 19).

2.3 Optimization

In the following section we wish to investigate one way of optimizing a couple of our algorithms for further comparison of algorithmic design and structure. We want to optimize to compare the results of a regular loop-search of a list of nodes, to the presumably optimized version using heap sort(1968).(*Timeline of algorithms* 2018) The results of these comparisons will be used in our analysis. The section will contain image examples of a maximum heap structure, whereas we are using a minimum heap structure - the principles are exactly the same. The only difference is that in a max-heap structure we want to sort for the highest value and in the min-heap we sort for the lowest value.

2.3.1 The Heap

Since pathfinding algorithms are searching for the lowest cost path we have applied a min-heap sort structure to our optimization of two of the algorithms. The term "heap" was first used in the relation with heap-sort, and is not to be confused with the heap as garbage-collected storage as it is used in various programming languages such as Java (Cormen et al. 2009, pp. 151). To see why using a heap is much faster than a search through a list, it can be useful to illustrate the structure of a heap. A heap can be thought of as an advanced array of items with specific index relations. The visualisation of a heap is a reversed binary tree-structure with every node having a relation to two other nodes. Nodes are therefore not separate items in a list, but are connections of relations. The index relations are visualized in 2.8.

As shown in 2.8, the first index holds a relation to the second and third index, the second index holds a relation to the fourth and fifth index, the third to the seventh and eighth index and so on. In practice we refer to any node linking two other nodes as the parent of those nodes. (A parents index will always be less than its children's and therefore to the left of them in the list). The goal is to satisfy the heap property, that the value of any node i other than the root, is at most equal to its parent - and in the case of a Min-Heap structure it shall be lower if it is not equal:

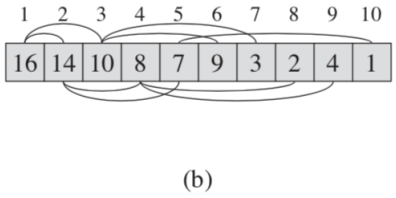
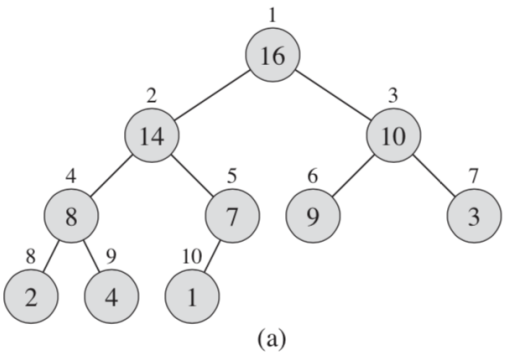


Figure 2.8: The heap as an array list, Cormen et al. 2009(pp. 152).

$$A[Parent(A)] \leq A(i) \tag{2.2}$$

Cormen et al. 2009, p. 153 When looking at the tree structure it becomes clear that we can always define the parent / children relationships in a fairly simple way. The number within the grey circle is the stored value of the node, while the number above is the index number.



To get to any current-node-*[i]*s position through the index relations shown in fig. 2.8

Figure 2.9: heap as conceptual tree,
(Cormen et al. 2009, pp. 152)

$$parent(i) = i/2 \tag{2.3}$$

$$leftChildNode(i) = 2 * i \tag{2.4}$$

$$rightChildNode(i) = 2 * i + 1 \tag{2.5}$$

Cormen et al. 2009, pp. 152

Notice that moving to the parent node from the right-child node we end up at a .5 index number which is rounded down(by automatic integer division in our code).

Usually when applying heap-sort it involves building the heap, a method to "heapify" the heap - meaning that the heap property will be maintained, and a way to extract the top heap item and "re-sorting" to find the new lowest value.

This is the structure of max-heapify:

```
MAX-HEAPIFY(A, i)
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.heap-size and A[l] > A[i]
 4       largest = l
 5  else largest = i
 6  if r ≤ A.heap-size and A[r] > A[largest]
 7       largest = r
 8  if largest ≠ i
 9       exchange A[i] with A[largest]
10       MAX-HEAPIFY(A, largest)
```

(Cormen et al. 2009, pp. 154)

> When it is called, MAX-HEAPIFY assumes that the binary trees rooted at LEFT(i) and RIGHT(i) are max-heaps, but that A[i] might be smaller than its children, thus violating the max-heap property.
>
> (Cormen et al. 2009, pp. 154)

The next sub-section is a basic description of how to create a heap and how to extract the minimum F cost, as well as a visualization of the "heapify" method.

2.3.2 Heap-sort

.

The first step of the heap sort procedure is to specify the size of the heap that is to be built.

$$A.heap.size = A.length \tag{2.6}$$

Whenever an element is passed into the heap the "heapify" method is called and as the heap is filled the heap property is being maintained through that method. As long as the heap property is maintained, it is possible to remove the top item in the heap and then update the heap to maintain its property.

The algorithm begins with (a) where A(i) is at index five (The highest index number that has children to compare with). Again, A(i) assumes that its children are both max-heaps but that itself is possibly a smaller value - this might sound contradictory but it basically just means that A(i) is the one that "checks" if it breaks the heap property. In the "heapify" example, shown in Figure 2.10, index five with a value of 16 is being compared with its children (child in this case). 16 is a bigger value than 7 so nothing is swapped. In the next run (b) of the loop A(i) changes to index four - compares with its children and swaps places with its highest-value child which is 14, and so on. As the heap is filled with nodes as an open list (list of nodes/neighbours to be evaluated), we can begin to extract the top item, sort the list anew and keep doing so till the heap is sorted. In the case of pathfinding we only need to extract the shortest path from A to B.

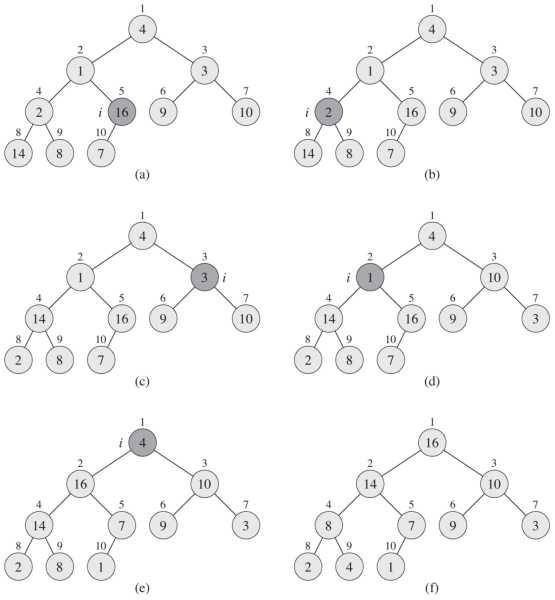


Figure 2.10: Heapify, (Cormen et al. 2009, pp. 158)

2.3.3 Extraction and updating

The following section is meant to illustrate how to extract and re-heapify the heap. These are the final steps for our heap-sort algorithm in order to apply it to pathfinding.

As seen in figure 2.11 line two it keeps going through the nodes as long as there are nodes in the heap to compare (down to 2). The top-heap element is then sent to an array list in line 3 just before decrementing the list by one. Now it heapifies the heap again to

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

Figure 2.11: Extract and update, (Cormen et al. 2009, pp. 160)

"sort up" the new highest value.

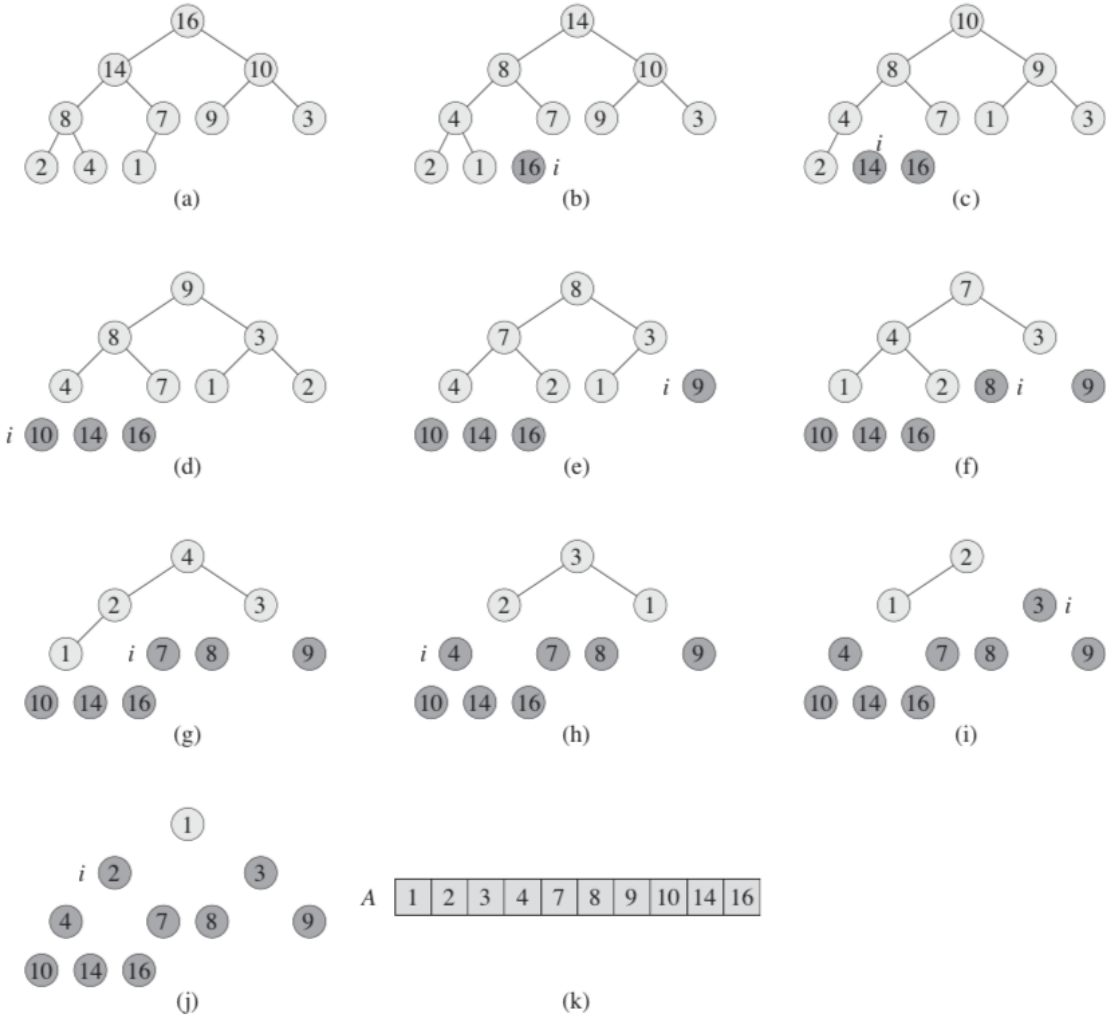


Figure 2.12: Extract and update (graph), (Cormen et al. 2009, pp. 161)

3 Design

In this section, there will be elaborated on our code and highlighted the areas which are important. The approach will be to bring out methods from the respective scripts, and dive into their individual functionality for the program as a whole. It's necessary to point out, that in the cases where vector3 is mentioned it contains three values - X, Y and Z. However in this case Z is treated as Y because the visualization of the pathfinding is two dimensional and Y is then the depth which is always set to 1. There is also two Node scripts, but in order not to repeat the same topics there will only be described one in this section. The reason for this is, that the other Node script - Dnode ("dijkstra node" used for the dijkstra algorithm) is similar, but a bit simplified compared to the Node script we will elaborate on here.

3.1 Node

First of, the Node script implements the IHeapItem interface with a type <*Node*>. The reason for this is that this script now have to implement a get and a set method. Further more we can see that the IHeapItem script on line 1, implements the Icomparable interface from where we also need to implement the CompareTo method that we will use to sort our Nodes in our Heap optimization of both the A* and the dijkstra scripts. This will be explained further in this section.

```
1 public interface IHeapItem<T> : IComparable<T>{
2
3     int HeapIndex { get; set; }
```

A Node contains five integers. The Hcost and Gcost meant for the A star algorithm. GridX and GridY so it knows its own position in the Grid array, and lastly a HeapIndex so it can be stored in, and knows its own place in the Heap. Then it contains a boolean which dictates if a node is walkable or not, a vector3 to store its

position in the game world and a reference to its parent Node - used for tracking its path once it is found.

3.1.1 Constructor

for each Node instantiated we have a constructor that - when initialized sets its walkable boolean, worldPosition, GridX and GridY.

```
1   public Node(bool walkable, Vector3 worldPosition, int gridX, int gridY){
2     this.walkable = walkable;
3     this.worldPosition = worldPosition;
4     this.gridX = gridX;
5     this.gridY = gridY;
6   }
```

3.1.2 FCost

Also for each Node, there is a FCost get method in order for the A* pathfinding script to calculate each of its next moves. As can be seen on line 4, it returns the gCost added the hCost because that is in fact the fCost seen in the Theory section under A*2.2.3

```
1   //-- || fCost get-method ||--\\
2   public int FCost{
3     get{
4       return gCost + hCost;
5     }
6   }
```

3.1.3 HeapIndex

The HeapIndex method is a getter and setter and is given its value in the pathfinding script.

```
1     public int HeapIndex
2     {
3         get {  return heapIndex;}
4
5         set  {heapIndex = value;}
6     }
```

3.1.4 CompareTo

the CompareTo method was created because it implemented the IHeapItem interface which implements the IComparable interface. This method allows one to sort Nodes in the heap optimization. The principle it works by is that it takes a node as parameter. It then initializes an integer called *compare* which stores the value returned from the CompareTo method, used on the FCost of the node which contains this method to the node which is sent through the method. If the cost is less than its compared node it returns -1, if it is equal it returns 0 and if its larger it returns 1. There's also implemented a tie breaker seen on line 4 which purpose is, if the FCost of the two nodes under comparison is equal, then there is a comparison to the hCost which is the distance to the goal from the Nodes under comparison. Since our heap is reversed meaning that Nodes with the lowest Fcost should be first, then it returns -*compare* so that it returns -1 if it's 1.

```
1      public int CompareTo(Node nodeToCompare)
2      {
3          int compare = FCost.CompareTo(nodeToCompare.FCost);
4          if (compare == 0){
5              compare = hCost.CompareTo(nodeToCompare.hCost);
6          }
7          return -compare;
8      }
9  }
```

3.2 Grid

The Grid1 script works as a command script, understood in the sense that it contains nine different booleans that through an inspector can be given the value true or false. Their values determine which algorithm is running, and further more for visual purpose, also show the path and searched area of each algorithm. It also has 2 two-dimensional arrays of types Node and Dnode, a float that holds the Node diameter, and a vector2 that holds the grid's size. There's also a layer-mask with the purpose of setting each node to either walkable or un-walkable - this will be further explained later in this section. Finally two integers that hold the grid size X and the grid size Y.

3.2.1 CreateGrid

The script contains two methods that create a grid. One that creates a grid of Dnodes which is made for the dijkstra algorithm, and one for the two remaining algorithms that consists of Nodes. They are similar in structure besides the population of either Nodes or Dnodes in the two two-dimensional arrays.

Besides setting the grid's X and Y to the same size as gridSizeX and gridsizeY we also start the nodes population at the bottom left corner. In order to set worldBottomLeft we need to use unity's *transform.position* which gives the game objects their position in the game world - (x = 0, y = 0, z = 0) being its centre/ "world origin". Then we subtract half the width of the gridWorldSizeX and half the height of gridWorldSizeY. The calculation can be seen on line 4 and sets the starting point for the Node or Dnode population to the left bottom corner. In the nested for loop, one can see the calculation done for giving each node its world point in the grids.Hereafter we initialize a temporary boolean *walkable* which is set to either true or false with unity's Physics.CheckSphere in which it passes the world point, the node radius and the unwalkableMask and this checks if a given node should be walkable or not. Lastly we populate the grids by giving it a place in the grid's X and Y and instantiate a Node with walkable - either true or false, the world point and its X and Y.

```
1   void CreateGrid(){
2
3     grid = new Node[gridSizeX, gridSizeY];
4     Vector3 worldBottomLeft = transform.position -   Vector3.right * gridWorldSize.x
          / 2 - Vector3.forward * gridWorldSize.y / 2;
5
6       for(int x = 0; x < gridSizeX; x ++){
7         for ( int y = 0; y < gridSizeY; y ++){
8         Vector3 worldPoint = worldBottomLeft + Vector3.right * (x * nodeDiameter +
              nodeRadius) + Vector3.forward * (y * nodeDiameter + nodeRadius);
9
10
11      bool walkable = !(Physics.CheckSphere(worldPoint,nodeRadius, unwalkableMask));
12        grid[x,y] = new Node(walkable,worldPoint, x, y);
13
14        }
```

```
15        }
16    }
```

3.2.2 GetNeighbours

The next method that will be presented is one the group made in order to get the neighbours of the current node under examination in our pathfinding. It also exists in two similar versions so that our dijkstra algorithm and the two remaining pathfinding algorithms can use their function.

The methods takes a Node or Dnode as parameter, this is done so that it's possible to pass the current node from our search algorithms. It starts by initializing a list of Nodes. Thereafter follows a nested for loop, and inside this we first off want to skip to check if both X and Y are 0 because this means, that it is in fact the node that we passed in and therefore there is no need to check it. After this we initialize two integers called *CheckX* and *CheckY* with their values set to the node's X position + the X value that are in the scope of the nested for loop, so we get the neighbour node's X position and the same goes for the Y value.

Then we make a check to see if the *CheckX* and *CheckY* are within the boundaries of the grid array that holds all our nodes and if it is, we add that node to the neighbour list and return it.

```
1   public List<Node> GetNeighbours (Node node){
2
3     List<Node> neighbours = new List<Node> ();
4
5     for (int x = -1; x <= 1; x++) {
6       for (int y = -1; y <= 1; y++) {
7
8         if (x == 0 && y == 0)
9
10          continue;
11
12
13          int checkX = node.gridX + x;
14          int checkY = node.gridY + y;
15
16
17          if(checkX >= 0 && checkX < gridSizeX && checkY >= 0 && checkY < gridSizeY
                ){
```

```
18              neighbours.Add (grid [checkX, checkY]);
19            }
20        }
21      }
22      return neighbours;
23    }
```

3.2.3 NodeFromWorldPoint

This method also appear in two versions like the two prior methods, and for the same reasons. The purpose of these methods are to provide each of the pathfinding algorithms with the exact positions of the start point and the end point of their search. They take a vector3 as parameter but only uses X and Z. Then we initialize two float variables with the percentage of the X and Y position, basically it can be a float within 0 and 1. To make sure it doesn't give a value out of the grid's area, if let's say, either the seeker or the target are not within the grid area because it would then throw errors. It implements the Mathf.Clamp01 that clamps the value within 0 and 1 for this purpose. Finally it initializes two integer values X and Y which takes the whole gridSizeX - 1 because arrays are 0 indexed. Then it's multiplied by percentageX and thus get the X position. The same counts for the Y position and then it returns the Grid position with the X and Y coordinates.

```
1   public Node NodeFromWorldPoint(Vector3 worldPosition){
2
3     float percentageX = (worldPosition.x + gridWorldSize.x / 2) / gridWorldSize.x;
4     float percentageY = (worldPosition.z + gridWorldSize.y / 2) / gridWorldSize.y;
5
6     percentageX = Mathf.Clamp01 (percentageX);
7     percentageY = Mathf.Clamp01 (percentageY);
8
9     int x = Mathf.RoundToInt((gridSizeX-1) * percentageX);
10    int y = Mathf.RoundToInt((gridSizeY-1) * percentageY);
11
12    return grid [x, y];
13  }
```

3.3 Heap

The following section will cover our implementation of a min-heap in our pathfinding program. It differs from the theoretical section not only by being a min-heap

(in regards to the examples used being of a max-heap) but also by being built up of more methods. These differences will be explained along the way.

First of all the heap class "communicates" with the nodes through an interface that any object using the heap, has to implement.

```
1         public class Heap<T> where T : IHeapItem<T> {
2
3     T[] items;
4     int currentItemCount;
```

This is a very flexible way of constructing the heap class. It uses T to represent a generic object so that it can fit different objects into the heap, as long as they implement the *IHeapItem* interface along with its methods. And this is basically the meaning of the first line, the public class *Heap* takes in a parameter of type T where the object T implement the *IHeapItem* interface. So far so good.
The heap consists of an array of T-items which will respectively be the node and D-node in the use of the heap. Besides that, the list needs a counter to keep track of how many T items there are in the heap, hence the *currentItemCount*.
The functionality that has to be implemented via the interface consists of a getter and a setter of the heap-index as well as a compare method so that its possible to compare the two nodes with each other.

```
1 public interface IHeapItem<T> : IComparable<T>{
2
3     int HeapIndex { get; set; }
4 }
```

3.3.1 Heap constructor

In the heap constructor the heap is set to be an array list of the parameter type T with a "passed-in" size of *maxHeapSize* - when the heap is constructed in the A* and dijkstra algorithms, the *grid.maxSize* will be passed in, meaning that the heap has indexes to contain all nodes in the grid.

```
1     public Heap(int maxHeapSize){
2         items = new T[maxHeapSize];
3     }
```

3.3.2 Add an item to the heap

This method takes in an item parameter of type T. When a node is passed through this method it is assigned its *heapIndex*-number through the *HeapIndex* setter method in the respective node's classes. The index will be equal to the *currentItemCount* which is incremented in the end of the method. The item can now be inserted into the heap in its correct position before being sorted up according to its value/ cost.

```
1     public void Add(T item){
2
3         item.HeapIndex = currentItemCount;
4         items[currentItemCount] = item;
5         SortUp(item);
6         currentItemCount++;
7     }
```

3.3.3 Remove the top item in the heap

This method has a return type T since the functionality needed is to return the first item of the sorted heap to the path list in the pathfinding classes (A* and dijkstra). It begins by defining a new variable of type T called firstItem which occupies index zero in the heap, the lowest cost item of the sorted list. Then it proceeds by decrementing the *currentItemCount* by one. Now that the *firstItem* variable holds the top node, it can give away its place in the heap, it gives it to the *currentItemCount* item which is in theory the most expensive node. This doesn't really matter since it is being sorted down in the heap in a couple of steps. It is being done this way to figuratively decrement the heap in the bottom instead of the top, since the agenda is to always be able to find the lowest cost value on top of the heap. Before sorting it down though, it is being set to its new *heapIndex* at zero since it comes with a *heapIndex* of *currentItemCount* which isn't its correct position any more. The method is ended by returning our *firstItem*, the original items[0].

```
1         public T RemoveFirst(){
2
3         T firstItem = items[0];
4         currentItemCount--;
5         items[0] = items[currentItemCount];
6         items[0].HeapIndex = 0;
7         SortDown(items[0]);
8         return firstItem;
```

```
9       }
```

3.3.4 Sort clarification

As the attentive reader might have noticed, there hasn't really been mentioned anything about the "heapify" concept described in our theory section. In that section the heapify method is being showcased as a recursion which has not been necessary in our use, since items are being added to the heap and in that method sorted in a while-loop in the respective pathfinding classes. Another difference is that we want to be able to both sort an item up the heap and down the heap. This is due to the fact that it is not necessary to sort and deliver a complete list from the heap to an array list. When the path has been found, the heap will still be filled with "neighbour nodes" and at that point it is not meaningful to transfer the items one by one into a list. Therefore, when an item is added it should be sorted up the heap, but when the first item is removed and replaced with *items[currentItemCount]* it needs to be sorted down through all the "neighbour nodes" that are still in the heap.

3.3.5 Sort up

The sort up method takes in an item parameter of type T. The parent to the item that are being passed in will have its index defined by the formula shown in the heap theory section, equation 2.3. (The calculations in the code looks a bit different, since the heap-index in the code starts at index zero contrary to the formulas shown in that figure). A while loop is then initiated where a new local *parentItem* is defined to be *items[parentIndex]*. Now it is possible to compare the incoming item with its parent and swap it if necessary, (if it has a lower cost than its parent) through our *Swap* method.

```
1     void SortUp(T item){
2
3         int parentIndex = (item.HeapIndex - 1) / 2;
4          while(true){
5              T parentItem = items[parentIndex];
6              if (item.CompareTo(parentItem) > 0){
7                  Swap(item, parentItem);
8              } else {
```

```
9                   break;
10              }
11              parentIndex = (item.HeapIndex - 1) / 2;
12          }
13      }
```

3.3.6 Sort down

This method is like a reversed version of the "sort up" method. It also takes in an item parameter of type T that actually comes from the bottom of the heap when the top heap-item has been removed. A while-loop is also initiated where it's possible to define the incoming item's children by the formula shown in the theory section, equations 2.4 and 2.5. A *swapIndex* is then created which is comparable with the *largest* variable in the theory description of "heapify". The "heapify" concept is actually a combined version of our sort up and sort down method.
The next step is to check if the incoming item even has a left and a right child. If there is a left child, which is being checked for first, it just sets the *swapIndex* to this child. If there also is a right child we then compare the left- and right child node with each other. The lowest cost of this comparison will then be stored in the *swapIndex* variable and lastly, we can compare the *swapIndex* with the item that is being passed through the method to begin with (i.e the parent). If the *swapIndex* has the lowest cost we then swap it with its parent, and if not we return from the loop.

```
1       void SortDown(T item){
2
3           while(true){
4                   int childIndexLeft = item.HeapIndex * 2 + 1;
5                   int childIndexRight = item.HeapIndex * 2 + 2;
6                   int swapIndex = 0;
7
8               if (childIndexLeft < currentItemCount){
9
10                  swapIndex = childIndexLeft;
11
12                  if (childIndexRight < currentItemCount){
13
14               if(items[childIndexLeft].CompareTo(items[childIndexRight]) < 0){
15
16                              swapIndex = childIndexRight;
17                      }
18                  }
```

```
19
20                  if (item.CompareTo(items[swapIndex]) < 0){
21
22                      Swap(item, items[swapIndex]);
23                  } else {
24                      return;
25                  }
26
27              } else {
28                  return;
29              }
30          }
31      }
```

3.4 Swap items

The swap method takes in two item parameters of type T, an *itemA* and an *itemB*. The method begin by defining the incoming items as each others index, getting ready for the switch. Before the actual swap a new int variable is defined (*itemAIndex*), to hold the index of *itemA* since it's about to be overwritten with the index number of *itemB*. The last step will be to overwrite *itemB* with the stored value of *itemA* which is held by *itemAIndex*.

```
1       void Swap(T itemA, T itemB){
2
3           items[itemA.HeapIndex] = itemB;
4           items[itemB.HeapIndex] = itemA;
5
6           int itemAIndex = itemA.HeapIndex;
7           itemA.HeapIndex = itemB.HeapIndex;
8           itemB.HeapIndex = itemAIndex;
9       }
10  }
```

3.5 Additional methods

There are a couple of small additional methods. There is a count method that returns the number of items in the heap. This is for the while loop in the pathfinding classes that uses the heap. The while loop runs as long as the count is bigger than zero. There is also an *update* method which will actually just sort up an item if we have found a neighbour to our *currentNode* with a lower cost in our pathfinding.

Last but not least there is a *contains* method with a return-type "boolean". This will return true if the heap contains a specific item T which can be passed through the method. This is to check if a lower cost neighbour of a *currentNode* is already in the heap.

3.6 Pathfinding

We consider the A* algorithm as the most complex of the three algorithms. Therefore we will present this one first in detail, and then follow up by describing our implementation of the dijkstra algorithm and lastly - the Breadth first search algorithm.

The Pathfinding script contains three data structures. A List, a Heap, and a HashSet. The List and the Heap's function is that of the openlist 2.2.3. and the HashSet is the closedList 2.2.3. We then instantiate a grid of Grid1 because we want to be able to use the methods from the Grid1 in our pathfinding and initialize it in our awake method.

most noticeable in this script is the two different pathfinding methods. We are using !Enumerator for calling the optimized and not optimized version of the A*. The logic behind using the IEnumerator was because we wanted the seeker to be able to follow the path, however this is for future improvements and we are not utilizing its properties.

3.6.1 FindPathLoop

FindPathLoop takes in two vector3, startPos and targetPos. These are used on line 9 and 10 to provide the method from grid with the coordinates to get the position for the start, and the end of the search. Also there is an instantiation of Stopwatch which is provided to see the time it takes the Unity to run the code, this is to make us able to compare the different pathfinding algorithms.

The first that need to be checked is if the two points of the search is walkable,

because if this isn't the case - there is no need to start the search. Now it initialize the openList and the closedSet, and the loop of the search begins and will run for now, until openList size is bigger than 0. Now it has to loop over the openList and set *currentNode* equal to the element in the openList with either the lowest FCost or, if the *currentNode* and a given element in the openList loops through have a equal FCost, then it checks if the hCost in the element is less than the *currentNode* hCost and set the *currentNode* equal to that element in the openList. After the loop, the *currentNode* will be removed from the openList and added to the closedSet to mark it as visited and not loop over it in the loops to come. On line 34 there's created a stop for the search, if the *currentNode* is equal to the *targetNode* then it prints the time it took to run in milliseconds, and set *pathSucces* to true which, at line 68 and triggers the if statement and it sets the closedSet in the grid equal to the closedSet which is the area searched through so its possible to visualize it.

From line 43 to 64 the *Getneighbours* method from grid is used in order to get the neighbour nodes of the *currentNode*. If one of the neighbours are either not walkable or are in the closedSet then its skipped, and proceeds to the next. Then an integer is initialized, *newMovementCostToNeighbour* and set its value equal to the *currentNode* gCost and use the GetDistance method from the grid, and add the distance between the *currentNode* and the *neighbour*. If the *newMovementCostToNeighbour* value is less than the *currentNode* gCost, or the openList doesn't contain the *neighbour* then the *neighbour* gCost is set equal to *newMovementCostToNeighbour*, and the hCost equal to the distance from the *neighbour* to the *targetNode* and the the parent of *neighbour* to *curretnNode*. If the openList doesn't contain the *neighbour* then its added. This will run until the *currentNode* is equal to the *targetNode* or the openList is empty.

```
1
2       IEnumerator FindPathLoop(Vector3 startPos, Vector3 targetPos){
3
4       Stopwatch swLoop = new Stopwatch();
5       swLoop.Start();
6
7       Vector3[] waypoints = new Vector3[0];
8           bool pathSuccess = false;
9
10          Node startNode = grid.NodeFromWorldPoint (startPos);
11          Node targetNode = grid.NodeFromWorldPoint (targetPos);
12
```

```
13          if (startNode.walkable && targetNode.walkable) {
14
15        openList = new List<Node>();
16        closedSet = new HashSet<Node> ();
17              openList.Add (startNode);
18
19        while (openList.Count > 0) {
20
21          Node currentNode = openList[0];
22
23          for (int i = 1; i < openList.Count; i++) {
24
25            if (openList[i].FCost < currentNode.FCost || openList[i].FCost ==
                  currentNode.FCost && openList[i].hCost < currentNode.hCost) {
26
27              currentNode = openList[i];
28                          }
29                      }
30
31          openList.Remove (currentNode);
32
33                      closedSet.Add(currentNode);
34
35                      if (currentNode == targetNode) {
36
37            swLoop.Stop();
38            print("Path found in: " + swLoop.ElapsedMilliseconds + " ms");
39            AStarMilliseconds = swLoop.ElapsedMilliseconds;
40                          pathSuccess = true;
41                          break;
42                      }
43
44                  foreach (Node neighbour in grid.GetNeighbours(currentNode)) {
45                  if (!neighbour.walkable || closedSet.Contains (neighbour)) {
46                          continue;
47                      }
48
49                      int newMovementCostToNeighbour = currentNode.gCost + GetDistance
                            (currentNode, neighbour);
50
51            if (newMovementCostToNeighbour < currentNode.gCost || !openList.Contains
                  (neighbour)) {
52
53                          neighbour.gCost = newMovementCostToNeighbour;
54                          neighbour.hCost = GetDistance (neighbour, targetNode);
55                          neighbour.parent = currentNode;
56
57              if (!openList.Contains(neighbour))
58
59                openList.Add(neighbour);
60
```

```
61                      }
62                  }
63              }
64          }
65
66          yield return null;
67          if (pathSuccess) {
68              waypoints = RetracePath (startNode, targetNode);
69        grid.closedSet = closedSet;
70          }
71      }
```

FindPathHeap

3.6.2

The optimized version essentially works by the same principles. The major difference is, that the list openList is substituted with our Heap instead. Then it's not needed to loop over the openList because, it sorts the Heap by values so the lowest always is at the top and therefore as seen on the code snippet below the first item from the heap is removed, and sets the *currentNode* equal to that item.

```
1         Node currentNode = openSet.RemoveFirst();
```

Dijkstra

3.7

The dijkstra implementation which has been made during this project uses the Dnode, and all Dnodes gCost are set to the maximum value of the variable type integer. This algorithm make use of the same principles as the approach on the A* algorithm. It contains a openList and a closedList, for the same purpose and it also instantiates a grid to use its methods. What's different here is that, instead of using the fCost and hCost, it only make use of the gCost. Therefore essentially we search in all directions, but find the path through the assignment of parents. This can be seen from line 3 to 7 below. Here it's looping through the openList to check which element gCost is lower than the *curretnNode* gCost. And if this is the case, *currentNode* is set equal to that element.

```
1       for (int i = 1; i < openList.Count; i++){
2
3         if(openList[i].gCost < currentNode.gCost){
4
```

```
5             currentNode = openList[i];
6           }
7         }
```

Next it loops through all the *neighbour* Dnodes of the *curretnNode*, and again, if a *neighbour* is either not walkable, or is in the closedList it will ignore it and continue to examine the next *neighbour*. That's where it initializes an integer *tentative _dist* which is the temporary value consisting of the *currentNode* gCost added to the movement cost of the *neighbour*. Then if the *tentative_dist* is less than the *neighbour* gCost which is the maximum value of integers. Then the *neighbour* gCost is set to the value of *tentaive_dist*. Then we assign the parent of *neighbour* to *currentNode*, and finally adds the *neighbour* to the closedList. If the openList doesn't contain the neighbour we wrap the loop up by adding it for the next iteration of the loop.

```
1         foreach (DNode neighbour in grid.GetDNeighbours(currentNode)){
2
3         if(!neighbour.walkable || closedList.Contains(neighbour) ){
4               continue;
5           }
6         int tentative_dist = currentNode.gCost + GetDistance(currentNode, neighbour);
7
8             if(tentative_dist < neighbour.gCost){
9               neighbour.gCost = tentative_dist;
10              neighbour.parent = currentNode;
11              closedList.Add(neighbour);
12
13              grid.dClosedList = closedList;
14            }
15            if(!openList.Contains(neighbour)){
16              openList.Add(neighbour);
17            }
18        }
19      }
20      }
```

3.7.1 Optimized

The optimization of our implementation of dijkstra is essentially the same as the A*. But the openList is substituted with the Heap. The Difference is, that the sorting of Dnode isn't based upon the fCost, but the gCost instead.

3.8 Breadth first search

Like dijkstra, the logic behind Breadth first search is essentially the same, there is a while loop and the openList and closedList are also present. What separates it is, that instead of having a focused search, it searches in all directions like in the implementation of the dijkstra algorithm. This can be seen on line 1 to 3 where we assign *currentNode* to the first index of the openList. We do not loop over the openList, but simply use the first index, assign it and remove it from the openList and finally add it to the closedList.

```
1         currentNode = openList[0];
2         openList.Remove(currentNode);
3         closedList.Add(currentNode);
```

After this, the *GetNeighbours* method is used again from the grid which we've also instantiated in this script to get the surrounding neighbours. The check to see if the neighbour is walkable or if the closedList is also used. But after that we only check if the neighbour is in the openList. And if it isn't the case, it will be assigned the *currentNode* to the parent of the *neighbour* and adds it to the openList.

```
1         foreach (Node i in grid.GetNeighbours(currentNode))
2         {
3           if (!i.walkable || closedList.Contains(i))
4           {
5             continue;
6           }
7
8
9           if (!openList.Contains(i))
10          {
11            i.parent = currentNode;
12            openList.Add(i);
13          }
14        }
15      }
16    }
17  }
```

4 Methodology

4.1 Journal

As a group it was decided to keep a journal of the progress throughout the months of this project. Several group members have tried this method before, and it turned out to be a good way of keeping track of every decision and move throughout the project work. The point of the journal is to give an honest impression of how the project have evolved. The journal is not meant to be a way of making our work effort look especially impressive. The important thing is to be completely honest of how the weeks have looked. It will therefore show weeks where there was done very little on the project, as well as weeks where there have been more intense work on the project. While this is a method to show readers how the work have progressed, it is also working as a way for the group to see how well it spent the time. To see if the work hours could have been prioritized differently or made other choices, that would perhaps have had a more beneficial outcome. It also works so that the group in the end of the project, can read through the journal, and make a reflection of the entire project. Often it is hard to remember how every week of the project have been, and here the journal is working phenomenally. It keeps track of every single step that we as a group have made. But its also useful regarding looking up design decisions, because remembering all the small decisions in a progress can be difficult.

With a group project like this, the believe is that it's often interesting to know how people are working together. Working together as a group, can at times be difficult. Personal interests and different schedules can interfere, and in worst case cause trouble between members. With the journal the intentions are to give

an impression of how everything have been worked out during the elapse of the semester. All members of the group have different courses to attend. This means that meetings can't always be scheduled and exams are also occurring at different dates. As shown in the journal, this has at times caused the work to be paused. However, we have still managed to keep close contact with each other and remembered to update the journal as often as possible.

4.2 Benchmarking

In this project its relevant to test our code and how fast it runs. This is especially with how fast the algorithms are. When benchmarking takes place in this project then its using the build-in Unity profiler which acts as a benchmarking mechanism where its possible to see, in great detail how the CPU runs the code with different algorithms. Benchmarking, simply put is a standard or point of reference against which things may be compared. The focus of the tests is only how fast the CPU will run the different algorithms from start to finish, but there is a lot of other stuff to test besides how fast they are.

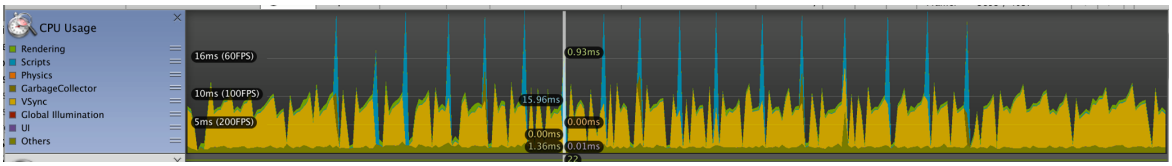


Figure 4.1: How the CPU tab looks in the Unity profiler

As seen in figure 4.1 above there is eight different things to measure in the "CPU Usage" tab. The one being focused on in this report is the "scripts" section which is seeing how fast the scripts were run with the CPU but there are two other parameters which also could have some usefulness in testing the algorithms in this case. The first one is the rendering which is used for rendering the visual image on the screen. That one is a bit more specific to how long it takes for the CPU to render the visual path which we have in Unity. The second one is the "Garbage collection" section which is how it possible to optimise the speed of the algorithms. The garbage collector works with the memory to go through it to find methods and in

general lines of code which is not in use any more and then remove them so not to slow the process of the continued use of the algorithms.

When conducting the tests there will be done a total of 86 tests to see how fast the CPU runs the algorithms. There will be made four different environments for testing the code, one which is empty, one with a barnyard environment, one with a enclosure environment, and finally one with a maze-like environment.

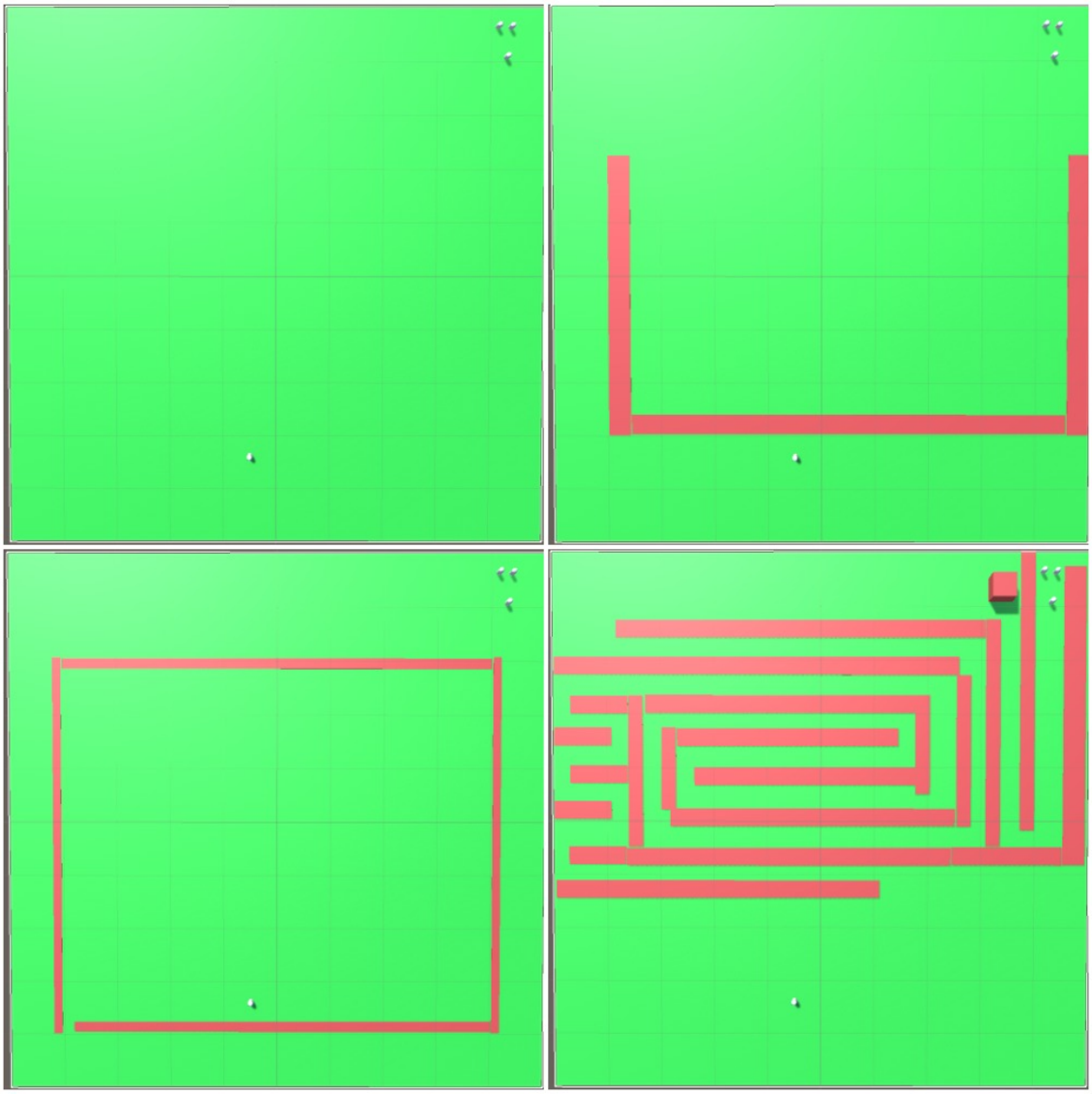


Figure 4.2: The 4 different environments tested in

The four different environments used in the tests is ran about 28-30 times with each algorithm which means each environment is tested 84-90 times. This means that there are 14-15 test runs per algorithm with 4-5 warm up runs to make sure that there will be as small a chance of deviation as possible. It will then be run on all of the environments which ends up being a total of 336 to 360 tests ran in order to get the numbers used for analysing.

4.3 Statistics

Statistics is a very broad term which can be used in many different instances of daily life. Statistics can also be used in Technical problem, which is how this project are going to use it in this report. To simplify it, statistics consists of a body of method for collecting and analysing data.

Statistics also consists of a population and samples. These two things can be seen in several different ways. First of all, population consists of the groups of individuals or object which are being used for gathering data. Next up is the samples which is the specific objects in the population which are being collected upon. Since the term population can be very vague and undefinable there is two ways of viewing the population. First is the finite population. this population indicates a population which is countable and therefore have a precise number over how large the population is. That could for example be students at RUC, this is a finite population and would be countable. Second is the hypothetical population. The hypothetical population consists of of a much more abstract concept and would normally be a population built upon the consideration under consideration. In this case, the entire population is finite since it's largely around how the algorithms are doing in comparison with each other and in comparison with how they, according to the literature, should be doing.

With that said, there is two different types of statistics. Descriptive and Inferential statistics. Where descriptive statistics use methods for summarizing and organizing information, inferential statistics use methods for measuring and drawing conclusions based on the information gathered. As seen in the way the two kinds of statistics works, they overlap on a large scale since they in some sense, are in a need of each other. Descriptive statistics uses different kinds of graphs, charts, and tables, and also the calculation of stuff like averages, measures of variation, and percentiles. Inferential statistics are all methods for point and interval estimation and hypothesis testing. In this report both have been used since there is a need to test and chart all those tests to find the variation in the algorithms while also looking at the intervals, And since a lot of the testing is based on the hypothesis

that some of the algorithms will do better than the others there's need to test that as well.

When the population is being chosen there is a specific set of numerical parameters which are unknown and the samples is what's going to be used to make inference about them. A sample is used to describe the characteristics of it self which then can be used to make inference about the parameters.

> A parameter is an unknown numerical summary of the population. A Statistic is a known numerical summary of the sample which can be used to make inference about parameters.
>
> Isotalo 2014, pp. 7

As the quote clearly states, the parameters and the samples statistics are overlapping and are in need of each other to answer the research question. The question in this case is more or less quite simple, which algorithm is the "best" there are three different algorithms and with the use of benchmarking it will be possible to see which is fastest and takes most effort to draw out.
Lastly is a model showing how one of the main objectives of statistics is to make inferences about the population from all the sample statistics in question and how the process from formulating the research problem to reporting the results from the data collected and the analysis described from that (Isotalo 2014, pp. 2-8).

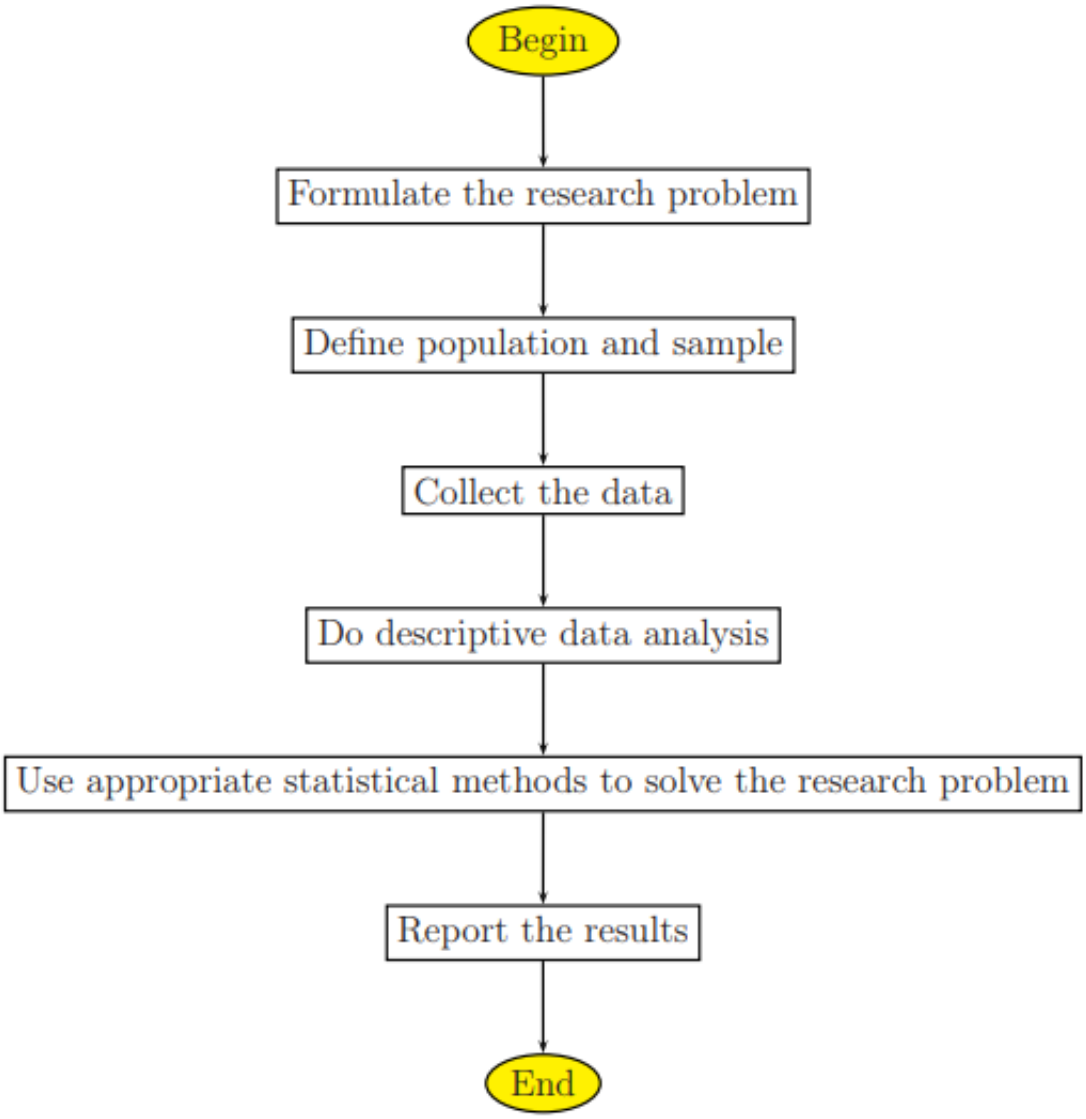


Figure 4.3: Statistical research from start to finish (Isotalo 2014, pp. 8)

5 Analysis & Discussion

In this chapter a decision was made to merge the discussion with the analysis since there are large similarities between the two, and it seemed difficult to do one without the other.

5.1 Introduction to tests

The foundation of the analysis is Unity's build-in profiler to see how fast each pathfinding algorithm is run. Each algorithms performance will be presented with ten runs of each, both the optimized and not optimized versions. Thereafter a comparing to the theory will be included to see, if the results are consistent with the theory, and finally see the difference between the implementations of the different algorithms. The progress of the analysis section will be with an presentation of the results, but in order not to repeat the same statements throughout this section, there will be a more in depth analysis of the first test, in regards to the theory and the other test results will be explained, with some minor elaborations.

5.2 No obstacles

In the first test, it's clear that the heap optimized A* implementation out performed the other algorithms. As seen on the graph, the algorithm closest is the A* without optimization. So in this case, the heuristics of both of the A* implementations was an advantage based upon the time it took to finish. Even though the A* implementation is going through more calculations than Breadth first search, it found the target almost 5 times faster. If we examine the two dijkstra algorithms, the Heap optimized version were around 0.6 milliseconds quicker than the one with-

out, so it made a difference but not as noticeable as with the A*.

The Breadth first search implementation algorithm took around the same completion time as the unoptimized dijkstra and the explanation for this can be that their search patterns are similar, while the A* implementation have a direction of its search. But breadth first search seems like it's executing according to the theoretical basis. On the figure of the grid, there are a path of yellow nodes surrounded by black nodes. The black nodes are on the closed list of the algorithm which means that they have been processed. Since the whole grid is black then we can safely say that BFS has run through the whole grid, and afterwards it found our path. But contrary to both dijkstra and the A* algorithm, breadth first search does not include weights to find the shortest path. It just makes sure what out of all the given options it is possible to find a path.

In regards to dijkstra then it's clear that it has the potential to run through the whole grid, or in other words, process all the nodes. Contrary to the explanation about dijkstra in the theory section where the concept behind it is explained through examples with weighted-edge graphs. In the case of this projects dijkastra algorithm it was run through a grid, where there were no cost assigned before the execution. The algorithm assign costs to edges it follows, and they have a cost of 10 for moving vertically or horizontally and 14 for a diagonal edge. dijkstra runs through the nodes and processes all the nodes that could have a path from the root node to the goal node.

Regarding the A* algorithm in this test. On the figure it could seem like it just takes the shortest path without checking the neighbouring nodes, But when the contents of the heap are printed then it shows, that it contains more items, than the nodes which generates the path. And that's also what the theory says, that from A*'s root node it will process the neighbouring nodes with the finishing node as a reference point. And that's why the path looks like it does and the reason why there's no black tiles in the grid, is because in the programming of the heap it takes processed nodes and sort it by the useful ones.

environment 1	A*		A* heap		Dijkstra		Dijkstra Heap		BFS
t 1. ms.	6,3	t 1. ms.	1,98	t 1. ms.	6,62	t 1. ms.	4,84	t 1. ms.	8,1
t 2. ms.	6	t 2. ms.	0,71	t 2. ms.	5,77	t 2. ms.	7,64	t 2. ms.	7,33
t 3. ms.	5,49	t 3. ms.	2	t 3. ms.	6,24	t 3. ms.	7,11	t 3. ms.	6,99
t 4. ms.	5,06	t 4. ms.	1,83	t 4. ms.	7,52	t 4. ms.	7,77	t 4. ms.	6,97
t 5. ms.	6,12	t 5. ms.	1,82	t 5. ms.	7,87	t 5. ms.	4,5	t 5. ms.	7,07
t 6. ms.	6,07	t 6. ms.	1,61	t 6. ms.	7,5	t 6. ms.	5,15	t 6. ms.	7,7
t 7. ms.	5,44	t 7. ms.	1,64	t 7. ms.	5,55	t 7. ms.	4,83	t 7. ms.	5,59
t 8. ms.	5,19	t 8. ms.	1,91	t 8. ms.	6,77	t 8. ms.	7,37	t 8. ms.	5,46
t 9. ms.	6,18	t 9. ms.	1,72	t 9. ms.	5,29	t 9. ms.	5,97	t 9. ms.	5,7
t 10. ms.	6,13	t 10. ms.	2,03	t 10. ms.	7,37	t 10. ms.	5,3	t 10. ms.	5,38
	5,798		**1,725**		**6,65**		**6,048**		**6,629**

Figure 5.1: The total measurements of time to complete all the algorithms, including the optimized ones in an environment without obstacles. The average time can be seen at the bottom

5.3 Maze environment

The second test was done in a maze environment. The test showed that the optimized implementation of dijkstra was the fastest. It performed at 4.364 milliseconds which is 0.7 milliseconds faster than the unoptimized A* implementation which were second in completion time. In this case, it seems that a not focussed search, in a area with many obstacles is a little faster but overall the average completion of the different algorithms were within 1.848 milliseconds of each other.

environment 2	A*		A* heap		Dijkstra		Dijkstra Heap		BFS
t 1. ms.	0,55	t 1. ms.	6,51	t 1. ms.	6,19	t 1. ms.	0,4	t 1. ms.	7,22
t 2. ms.	5,15	t 2. ms.	4,97	t 2. ms.	7,44	t 2. ms.	0,95	t 2. ms.	5,95
t 3. ms.	7,35	t 3. ms.	5,99	t 3. ms.	7,15	t 3. ms.	7,49	t 3. ms.	6,31
t 4. ms.	0,56	t 4. ms.	5,62	t 4. ms.	5,8	t 4. ms.	5,05	t 4. ms.	6,72
t 5. ms.	5,41	t 5. ms.	6,13	t 5. ms.	0,44	t 5. ms.	5,06	t 5. ms.	7,66
t 6. ms.	6,66	t 6. ms.	5,27	t 6. ms.	0,45	t 6. ms.	5,38	t 6. ms.	7,3
t 7. ms.	5,12	t 7. ms.	7,44	t 7. ms.	5,38	t 7. ms.	7,79	t 7. ms.	8,08
t 8. ms.	5,85	t 8. ms.	5,77	t 8. ms.	6,29	t 8. ms.	6,07	t 8. ms.	5,15
t 9. ms.	7,59	t 9. ms.	5,11	t 9. ms.	8,51	t 9. ms.	4,67	t 9. ms.	7,18
t 10. ms.	6,4	t 10. ms.	6,63	t 10. ms.	6,6	t 10. ms.	0,78	t 10. ms.	0,55
	5,064		**5,944**		**5,425**		**4,364**		**6,212**

Figure 5.2: The total measurements of time to complete all the algorithms, including the optimized ones in a maze environment. The average time can be seen at the bottom

5.4 Barnyard environment

The average completion of the third test showed, that the barnyard environment in general took a longer time to complete. The best completion time was performed by the unoptimized dijkstra implementation and was 5.482 milliseconds. The second best completion time was performed by the breadth first search implementation and was 5.663 milliseconds. Both the optimized dijkstra and A* implementation, and the unoptimized A* implementation were all above 6 milliseconds.

environment 3		A*		A* heap		Dijkstra		Dijkstra Heap		BFS	
t 1. ms.	7,26		t 1. ms.	6,37	t 1. ms.	5,47	t 1. ms.	7,94	t 1. ms.	6,32	
t 2. ms.	6,81		t 2. ms.	6,25	t 2. ms.	5,3	t 2. ms.	5,38	t 2. ms.	7,26	
t 3. ms.	6,63		t 3. ms.	5,6	t 3. ms.	5,16	t 3. ms.	7,87	t 3. ms.	0,45	
t 4. ms.	7,15		t 4. ms.	7,72	t 4. ms.	0,68	t 4. ms.	4,67	t 4. ms.	6,29	
t 5. ms.	8,2		t 5. ms.	6,64	t 5. ms.	7,47	t 5. ms.	8,68	t 5. ms.	5,25	
t 6. ms.	6,01		t 6. ms.	5,31	t 6. ms.	4,76	t 6. ms.	6,24	t 6. ms.	7,44	
t 7. ms.	7,78		t 7. ms.	7,17	t 7. ms.	7,36	t 7. ms.	7,91	t 7. ms.	5,33	
t 8. ms.	7,36		t 8. ms.	5,06	t 8. ms.	6,19	t 8. ms.	7,48	t 8. ms.	6,75	
t 9. ms.	6,31		t 9. ms.	7,45	t 9. ms.	5,12	t 9. ms.	6,54	t 9. ms.	5,77	
t 10. ms.	0,78		t 10. ms.	4,33	t 10. ms.	7,31	t 10. ms.	4,46	t 10. ms.	5,77	
	6,429			**6,19**		**5,482**		**6,717**		**5,663**	

Figure 5.3: The total measurements of time to complete all the algorithms, including the optimized ones in a barnyard environment. The average time can be seen at the bottom

5.5 Enclosure environment

Fourth and last test, was performed looking for a target hidden in a square with only one entrance. This was the most demanding task for the pathfinding implementations. The results show, that the breadth first search implementations completion time was 5.888 milliseconds. The optimized A* implementation came second and performed at 6.11 milliseconds followed by both dijkstra implementations. Only slight fluctuations can be seen between the average completion time. The only result that stands out is the unoptimized A* implementation which performed at 6.862 milliseconds.

environment 4		A*		A* heap		Dijkstra		Dijkstra Heap		BFS
t 1. ms.	7,32		t 1. ms.	6,26	t 1. ms.	5,01	t 1. ms.	4,86	t 1. ms.	4,33
t 2. ms.	7,09		t 2. ms.	6,24	t 2. ms.	7,29	t 2. ms.	4,76	t 2. ms.	5,37
t 3. ms.	7,63		t 3. ms.	7,4	t 3. ms.	5,65	t 3. ms.	5,04	t 3. ms.	6,19
t 4. ms.	7,3		t 4. ms.	5,07	t 4. ms.	6,28	t 4. ms.	7,46	t 4. ms.	7,14
t 5. ms.	7,46		t 5. ms.	5,16	t 5. ms.	8,85	t 5. ms.	6,39	t 5. ms.	4,91
t 6. ms.	6,31		t 6. ms.	7,86	t 6. ms.	4,61	t 6. ms.	6,48	t 6. ms.	6,11
t 7. ms.	4,68		t 7. ms.	6,29	t 7. ms.	5,74	t 7. ms.	7,83	t 7. ms.	4,95
t 8. ms.	5,66		t 8. ms.	4,63	t 8. ms.	4,92	t 8. ms.	4,92	t 8. ms.	5,32
t 9. ms.	7,12		t 9. ms.	4,16	t 9. ms.	5,35	t 9. ms.	5,86	t 9. ms.	7,33
t 10. ms.	8,05		t 10. ms.	8,03	t 10. ms.	8,11	t 10. ms.	8	t 10. ms.	7,23
	6,862			**6,11**		**6,181**		**6,16**		**5,888**

Figure 5.4: The total measurements of time to complete all the algorithms, including the optimized ones in an enclosure environment. The average time can be seen at the bottom

5.6 With or without heap

Our initial thought of the heap implementation was, that it would make the dijkstra and A*'s completion time faster. However, it seems that the heap optimization only had an effect on the A* algorithm in test 1 where the search area was clear of obstacles. In the case of the dijkstra optimization, on the test where it showed a noticeable difference were in test 2. Therefore we would like to mention that, calling it a optimized version depends on the context in which the algorithms perform, rather than an overall optimization of the algorithm.

5.7 Hashset and lists

In the first implementation of both dijkstra and breadth first search, the completion time was above 9 seconds in a simple maze like area. It lead to a lot of frustrations, we realized that we were using a list as the closedlist. This lead us to think that, since we in both of the algorithms when we check for neighbours, essentially in a worst case scenario, use the contains method 8 times. What we then realized was, that the list's contain method's performance fell every time a element is added. We then tried to use a hashset instead and realized that, the 9 seconds fell to around 30 milliseconds. We can't explain the exact behaviour of the hashset besides the

completion time. But what we did realize when using a list was, that when we moved the target closer, the completion time was quicker for both dijkstra and breadth first search in an linear pattern. So for each element added to the list it seems that the lock up time is getting slower on a linear pattern.

5.8 Sources of errors and uncertainties

In retrospective after generating the results it was discussed that having a different seeker/start-node per algorithm, may not have been the best setup for the tests. Although they are placed right beside each other, and there's presumably talk about a difference of microseconds its still considered as a source of error.
In a perfect world we would have had our own benchmarking script working which would have equipped us with a lot more test results (We were striving for 240,000 results in total). This didn't come to fruition partly due to unforeseen time constraints in the last part of our project work - we lost a group-member and we spent a week trying to make our own benchmarking class work without any success. We are very much aware that creating a time-average for our algorithms would have been even more precise with a lot more tests.

5.9 Expectations

When we first started testing all the algorithms with and without optimization there was a clear expectation from everyone in the group that the A* algorithm would out perform the other algorithms. It turned out that, it wasn't the case and there was only one time where it actually was the fastest and that was the completely empty environment. When the A* was the fastest it was with the use of the heap optimization and it severely outperformed the others at that time and that is because the heap wouldn't get just as filled as it would when there is obstacles and since the environment is empty it didn't have to work through every node.

6 Conclusion

Based on our analysis, we can conclude that our different implementations of pathfinding algorithms serves different purposes. For an example, we can see from our data, that the optimized version of our A* implementation outshines the other algorithms on a grid with no obstacles. But as soon as the search area becomes more complex with obstacles of different shapes, the A* gets competition in completion time by the remaining algorithms. The breadth first search algorithm have a average completion time in all our four tests - this can be explained by its search pattern, because it doesn't do calculations, it simply searches in all directions until it reaches its end point. Furthermore it doesn't provide us with the optimal path but rather guarantees that the target is reachable. The dijkstra implementations works by the same principle, but assign a gCost to each node the further we get away from the starting point, and will choose the one from the openList with the lowest gCost. Throughout the search it assigns the child nodes according to the gCost - and through this assignment of child nodes we are able to track back the shortest path. Dijkstra also has an average that's stable across the tests, with the exception of the maze.
So what this report can conclude is, the A* algorithm with optimization outshines the others on the empty environment, the BFS algorithm makes sure that there is a way, it doesn't necessarily find the most optimal way but a way is found, and lastly the dijkstra is the best in maze environment. But overall the algorithms performed, for the most part, very similarly.

7 Perspectivation

In this chapter we would like to come with some perspective for the entire report and everything we would have liked to do as well as some future possibilities for our product.

7.1 The game idea

This project started back in February with the idea of us creating a game of some sorts, this was for a long time the path of the project. We had the idea of a 2D RPG game with the player fighting different kinds of monsters and those monster would have had the A* algorithm set on them so they would be hard to avoid and they would chase the player, if he was inside the radius of them. That was all the way up until the realisation that what we had in mind for a game and how high our expectations were, we would never reach an acceptable level of game either for our selves or the report. That brought some changes to the project which we thought would bring the academical level a bit higher in the report. We implemented two more algorithms in the forms of Breadth First Search and dijkstra's algorithm and changed our focus from wanting to create a game to wanting to compare algorithms.

7.2 Benchmarking

When it comes to comparing the algorithms there was the idea of using benchmarking for comparing the algorithms. At first it seemed that using benchmarking in Unity was not something we could do easily, and there seemed to be a problem with the way that Unity is built up from within. First we gave it a try to make it

work with the ".NET Framework" which is a special cross-platform open source developer platform, and following many different guides and reading a lot about it, it turned out that within Unity there is a built-in framework that is too old to follow along with newer frameworks. This meant that there were no way to install the "BenchmarkDotNet" package from NuGet to get some kind of .NET benchmark code going.

The next thing we tried was to take the individual scripts from Unity and copying them out into a new project created in Microsoft Visual Studio 2017. This idea was followed by a different kind of problem. First of there was the issue of all the different kinds of methods and variables. Take for example our grid, which is built from the bottom left corner using the "Vector3" data type in a 3D environment. Other than that there arose a whole new problem of us using unity not only for the visuals but also to see how fast the algorithms could create a visual path. If we were to benchmark the code using only Microsoft Visual Studio without Unity, we would not get the full picture.

In the end we found the profiler which we used to get the accurate numbers we did. We did not code the benchmarking which isn't optimal considering that when we use the profiler we really don't know what is going on behind the scenes; we know what happens and what it calculates upon, but we do not know how it does it other than on a theoretical level.

7.3 Google Maps comparison

After we got rid of the idea of producing our own game and took up the task of creating more algorithms for comparisons we were suggested by our supervisor, that we should try and get the algorithms to work with google maps and compare that code. The whole idea arose around researching how much faster the Doogle maps algorithm was compared to our algorithms and then analysing on those data. In order to see how we either would be able to improve on the algorithms or how Google Maps worked and if it took different routes from our algorithms and why it did that. In the end, we decided that we couldn't do that because we felt

that it would split the waters of the report too much. We wanted to keep the focus on which of the algorithms did the best compared to each other and compared to what the theory says, instead of seeing how they faired against Google Maps which would always come out superior.

7.4 Alternative uses

7.4.1 A* Algorithm

What the A* algorithm really excels at, is finding the shortest path from A t B. With this in mind there is an alternative use that quickly springs to mind, A* could have served a purpose of finding way in a city in just like a GPS. Even though a GPS always would be faster there is the argument that we could have created a GPS like application to see if there was any rivalling the Google Maps algorithm and compare it to real GPS' to see if they found the same.

7.4.2 Breadth First Search

Breadth First Search really excels at finding a way, not necessarily the fastest way but a way never the less. Breadth first search would be great to be used in a game in the lines of a tower defence. A tower defence game works in the lines of enemies going from one end of a map to the other, while the player has to kill them using towers along the way. In the development of such a game there is the argument for using BFS and apply it on the map since it would then check if there is a way from A to B and stopping the player form placing a tower if that is not the case.

7.4.3 Dijkstra's Algorithm

Staying on the track of tower defence, dijkstra's algorithm would also serve a purpose in the developing of a tower defence game. Usually in a tower defence game the towers have a certain range for shooting the enemies, so since dijkstra calculates how long there is from a certain point on the map to everywhere else the algorithm could be put on the each individual tower and then it would be able to,

calculate how long there is from that specific tower to anywhere else and therefore know when enemies are in range.

7.5 Costs

We find it important to mention that the costs could have been created in a more dynamic manner for different applications be it computer games or real-life implementation. In the same sense that we created a layer of un-walkable nodes, we could have created layers with different costs representing varied terrain, traffic considerations or other external factors that could influence a travel between point "A" and point "B". Whether this would have given us much different comparison results in times and paths are unclear at this moment, but these speculations could have laid the ground for further experimentations.

Bibliography

Bhasin, Harsh. 2015. *Algorithms, design and analysis* [**inlang**eng]. Oxford: Oxford University Press. ISBN: 0199456666.

Cormen et al. 2009. *Introduction to algorithms*. Third edition. The MIT press.

Isotalo, J. 2014. *Basics of Statistics*. CreateSpace Independent Publishing Platform. ISBN: 9781502424655. https://books.google.dk/books?id=uUSvoQEACAAJ.

Oksa, Sampsa, et al. 2014. "Pathfinding in a 3D-environment Using Unity3D".

SebLague. 2016. *Pathfinding-2D*. https://github.com/SebLague/Pathfinding.

Sedgewick, Robert. 2011. *Algorithms* [**inlang**eng]. 4. ed. Upper Saddle River, NJ: Addison-Wesley. ISBN: 032157351X.

Timeline of algorithms [**inlang**en]. 2018. Page Version ID: 839113808. Visited on 05/27/2018. https://en.wikipedia.org/w/index.php?title=Timeline_of_algorithms&oldid=839113808.

Vinther, Anders Strand-Holm, Magnus Strand-Holm Vinther, and Peyman Afshani. 2015. "Pathfinding in Two-dimensional Worlds". *no. June*.

Appendix

A Journal

Week 6

This week we started the making groups for the project. Before the day, several members of the now existing group, had talked together about making a group for the semester. We did this, because we made a project together the previous semester, and it was therefore a good opportunity to make a group together again.
Because we already knew each other, we quickly started talking about what kind of project we could be interested in. There wasn't any need for talking about how we were going to meet, or talk about other social relations.
After the group were made, we came to talk about the idea of a pathfinding algorithm. We liked the use one, and with possibility in a game.

Week 7

This week all the courses starts, and this brings a talk about how and when we can have meetings. We also got assigned a different project supervisor than we wanted. That made some troubles, since we were prepared to have a project in english, but was assigned a danish supervisor. After a little struggle, we got the supervisor we wished for, and Junia is guiding us.
We started slowly with programming this week as well. We want to get a good feeling out of Unity and how the program works. Everything is still in discussion, so nothing is being sat in stone this week.

Week 8

A group meeting this week. We talked about what kind of methods were going to be used, when the programming starts.
There were talk about using a SCRUM method and pair programming. The pair programming is optimal for this group

Week 9

Short meeting where we talked about the methods we would like to use. Also delegated work regarding future work -as in where would we like to move towards: make a UML (class diagram) for the code we have so far. Think this could provide us with a picture of how far we are, regarding future pair programming.

Week 10

This week nothing really happened with the project.

Week 11

Preparation for midt term. Talking about what we want to present, and what features we find important in the work we have done so far.
Had a meeting with our supervisor Junia.

Week 12

Midt term this week. Some group members were away, and others were sick, so we were quite challenged. Thankfully, one person carried the project to victory. We got a lot of feedback and reflected on it at later point.

Week 13

Week 13. This week had easter holidays, and therefore didn't ford for much meeting. Several members of the group were working on an assignment for at course, and the project were therefore put on hold this week.
However we did manage to agree on a meeting the following week, to talk about how we were going to progress further.

Week 14

We started early wednesday with a meeting. Connected one of the computers to a tv screen and went through Unity. We want to make sure that everybody is comfortable with Unity. Because of that, we have talked about taking a day, where we complete one of the tutorials associated with Unity. That should hopefully provide a good understanding of how the program works, and with that, ensure a more smooth transition from IntelliJ to Unity.

The basic parts of the game are being discussed. How the player is going to be set up, what kind of interactions there will be, the idea of a backpack in a game. With these classes, we are trying to figure out how the code is being designed and what code is usable where. Also how the different classes are losing health. A talk about using a temporary enemy in the game is being discussed. This could potentially test how the interaction is sorting out, and see if we can find a pathfinding algorithm that suits the purpose.

Week 15

Tuesday a short skype meeting, to set in date what to do now. Talking about if we are actually using SCRUM, since so far it hasn't been used at all. We would like to start on some writing, besides the coding. Have found some algorithm books and will see if we can use them, and divide the work between group members.
Starting to enter a period where we have close to no courses, and are talking about have meetings on mondays now, in addition to tuesday/wednesday, where we usually meet.
Talking about setting up a meeting with our supervisor for possibly wednesday or friday week 16. At this point it has been some time since we last talked with our supervisor, so it would be good to update her on where we are now.
We are setting up future programming days. Some of us are planning on trying to find out how unity works. Other are going to read in some books and the rest is trying to implement an enemy in the game we are working on in Unity.

Week 16

So far, 4 out of 5 members are taking a computer science course, which is taking up a big amount of time. We are finding it hard to make time for the project, since we are struggling with the computer science course.
The plan as of tuesday 17/4, is to try and have a clear plan for the next week or two. At this point, we are spending close to all time on the course, and are setting aside some to little time for the project. We want to have a clear plan for what is going to happen, so we don't get too many surprises. After today tuesday, everybody in the group should know what they should be doing for the next 7 days.
Today we also talked about an upcoming supervisor meeting on thursday 19/4. We wish to prepare for the meeting, and have some questions ready.

Monday 07-05-2018

Today we have worked with pathfinding. We are working with unity game engine, and since we have no prior experience with it we decided, as a starting point, to use Sebastian Lague's Github repository (SebLague 2016)
we got the path finder to work. It should be noted, that we haven't implemented the heap optimization, since it's to complex for us at this given time. instead we focused on the PathRequestManager script and the Unit script. the idea here is, that if we get it to work properly according to our expectations, we have a solid foundation, and this we give us the tools we need to rework it and implement some more personalized code (standing on the shoulders of giants).
For the moment the non playable characters are called seekers, and is a cylinder Object. We fear, that if we have more than a few "seekers" we will see performance issues if we do not queue them, and they can calculate their paths sequentially instead of simultaneously.
In order to get our pathfinder working, we've created a 2d node array, that is out grid and it's from here we get the X and Y coordinates that feeds the pathfinder.
We have also created a cylinder target object so we have two points that makes up the start and end of the path. we have not been able to ge the seeker objectto move yet, but tommorow tuesday the 08-05-2018 we will continue and focus on that.

Wednesday 09-05-2018

Today our main focus where on getting the two pieces of code stitched together - those are the pathfinding with its respective scripts, and the player movement.

first off, we wanted to get more than one npc moving. after that where sorted, we merged the scripts of both unity projects, and ended up struggling with the camera movement. we havn't figured out exactly why, but we think it has something to do with the fact that we want to make the camera static on the player game object *to be continued tommorow.*

Then we decided, that we need a finite state machine. After some search we located a template that makes us able to create an easy flexible state machine,

and the second goal for tommorow will be on how we can implement it in our code, and understand the principles behind it.

Friday 10-05-2018

Today we implemented a very simple version of a statemachine. At the moment it works by simple boolean's that tells every state when to activate and when to be on idle. The next iteration will be to implement it with our pathfinding, and make it work by adding a proximity circle to each game object, that tells it when to activate the pathfinding, if the player moves within range of the proximity.
We have also been working on merging our different code snippets, since we are working on several computers, we need to see if it works when we combine it.

Monday 14-05-2018

Today we are working at home. The reason for this is, that we all have something to do individually so we focus on finishing the theory section of the report so that we can focus on our technical section which will describe our code and how it works.
also one of the groups members are struggling to merge our code, so this will hopefully finished today.

Tuesday 15-05-20180

Two group members are writing and coordinating their process among each other, the remaining members are at Roskilde University. Today marks a change in the project - we have come to the conclusion, that making a full on game have proven to be too big of a challenge for us. Therefore a discussion about our focus of the rapport have started. What we have agreed on is, that tomorrow 16-05-2018 we will bring our thought with us and share our concern with our supervisor.

Wednesday 16-05-2018

Meeting with our supervisor. After sharing our thought and concerns, we are advised to build upon what we already have, and that is a working A* path finding algorithm. So if we can implement more algorithms - our personal goal i at least three preferably five, benchmarking could be a tool to compare each

algorithms performance and also we aim to get a visual output to check the paths that are found. finally we are also encouraged to compare google's path finding algorithm with our. So we now have a new direction for the project and most important alot of work to do.

Thursday 17-05-2018

Today we have restructured our focus from game creation to implementing path finding algorithms. Our first goal is to implement breath first search. the plan is obviously to do that as quick as possible, and when that is done to implement dijkstras algorithm to be able to compare those, and afterwards see what time we have remaining if other algorithms shall be implemented.
We have started to research on benchmarking and how we can use it in unity.
A members of our group have decided that he is not going to continue on this education, and therefore have left the group.

Friday 18-05-2018

We now have a working breath first search algorithm that searches outwards, until it reaches the goal node - we have problems with setting the right parent node, so for now the path we visualize are some places two nodes wide, we suspect that it have something to do with our assignment of parent nodes when we check for neighbours.
Our research on benchmarking continues. we havn't been able to get it to work yet.

Monday 21-05-2018

Some work have been done over the weekend on Dijkstras algorithm. It isn't complete, but we are working on it while also trying to structure our rapport. Benchmarking research is still going on.

Tuesday 22-05-2018

Dijkstra is working. Our theory is in the process of being restructured to fit our new approach. Benchmarking have not yet been a success so we have decided, to implement a text field to show the time it takes to complete the algorithm in unity, and this will be our tool to compare the algorithms. further more, we have two version of A* algorithm one with a heap optimization and one

without. Also Dijkstra got the heap optimization and one without for further analyses.
We now decide, that it is not realistic for us to compare our algorithms with the google path finding algorithm unfortunately.

Wednesday 23-05-2018

Wednesday has been spent on trying to get the "BenchmarkDotNet" to work with the Unity game engine too see if we can test our codes speed. There has been created an issue on the GitHub website where the code for BenchmarkDotNet is found in the hopes of us getting some answers to our issues with using the correct framework for implementing .NET properly. Other than that there has been made great strides towards the finishing of the theory.

Thursday 24-05-2018

The work towards getting the BenchmarkDotNet working is still in progress. Meanwhile the rest of the group are working full power to get the report ready for our test day, which have - for now been scheduled for Saturday , unless of course we are ready before that.

Friday 25-05-2018

BenchmarkDotNet has been dropped becasue of the problems it caused in Unity and the answer we got on GitHub but Junia pointed out something we could do instead. We have tried taking the code we use in Unity outside of Unity and open it up in a new project in Visual Studio. That aswell did not work since there are a lot of datatypes and methods which we have in the code and only works in Unity as a starting point so in the end of the day the Benchmarking has been put away. We have decided to focus our energy on getting results with our existing algorithms and design some different environments to test them in.

Saturday 26-05-2018

Test day. Today we've planned to perform the test of our implementations. Originally the benchmarking where dropped yesterday, but fortunately we have now discovered that unity have an in build profiler, which enables us to see how much time is spent on each of our pathfinding scripts when they

are runned. So we quickly saddled the tests horse and decided that each implementations should have 4 warm up runs, before the actual test and then followed up by 10 tests each. The tests have been carried out, and all focus is now on getting the report finished.

Sunday 27-05-2018

We are putting the finishing touches on the report and are starting the whole correction process. We have looked at doing some quality of life improvements on the report and we have finished correcting the report tonight and are handing in the report.