Synthetic Acumen

Wild Racoons

Ryan Dougherty, Kristina Blinova, Craig Belshaw

2018

Table of Contents

[1. Introduction 2](#_Toc512169140)

[2. Software Design 3](#_Toc512169141)

[2.1. Pathfinding: 3](#_Toc512169142)

[2.1.1. Optimizations: 6](#_Toc512169143)

[2.2. Decision Making: 8](#_Toc512169144)

[2.2.1. Requirements 8](#_Toc512169145)

[2.2.2. State machine 8](#_Toc512169146)

[2.2.3. Decision tree 8](#_Toc512169147)

[2.2.4. Optimization 10](#_Toc512169148)

[2.3. Learning: 10](#_Toc512169149)

[3. Implementation Issues 12](#_Toc512169150)

[3.1. Pathfinding: 12](#_Toc512169151)

[3.2. Decision Making: 12](#_Toc512169152)

[3.3. Learning: 12](#_Toc512169153)

[4. Results 13](#_Toc512169154)

[4.1. Pathfinding: 13](#_Toc512169155)

[4.2. Decision Making 14](#_Toc512169156)

[4.3. Learning: 15](#_Toc512169157)

[5. Conclusion 16](#_Toc512169158)

[6. Peer Assessment 17](#_Toc512169159)

# Introduction

The purpose of the report is to discuss the game we, as a team, created for the AI project. The report will cover the software implementation we used to create the game as well as the results we gained from our selection of AI algorithms. We will also be discussing the various implementation issues we faced during our project as well as evaluating the overall experience creating the game as well as the capabilities as team members. The purpose of the project was to create a game that used Pathfinding, Learning and Behavior in some form within the game.

# Software Design

## Pathfinding:

We decided to use pathfinding within our game regarding agent movement. For the player, this would be achieved through right clicking to select a target destination, as seen in traditional RTS games such as StarCraft or Command and Conquer. NPCs also use the same system of movement, but their destinations would need to be determined directly from the game logic.

There are two realistic options which were explored regarding the implementation of our pathfinding algorithm. Those being Dijkstra and A-star. Both are viable options to implement within the game. However, we believe that A-star was more suited to the real-time environment involved within a game and that due to Dijkstra’s lack of a heuristic function we would have had to face greater computation time to generate paths for the AI.

For the algorithm to work a grid of nodes would need to be generated for the algorithm to operate on. Each node within this grid would be required to either hold a reference to, or be able to calculate each of its neighbors. We decided to use a grid size of 100x100. For the purposes of this report the examples shown will use a 5x5 grid.

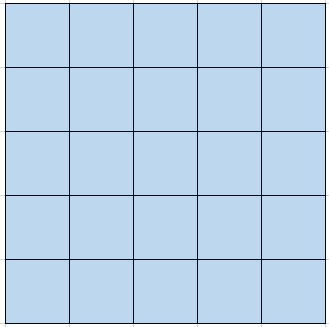


Figure 1 A Grid of 5x5 Nodes.

Nodes typically have 8 neighbors these consist of up, down, left, right and the associated diagonals. Edge cases must also be considered for those nodes which lie on the perimeter of the grid.

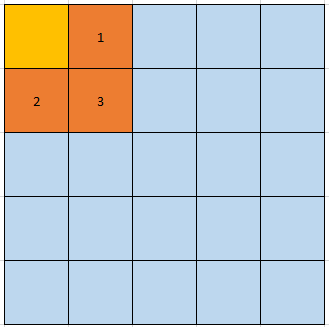
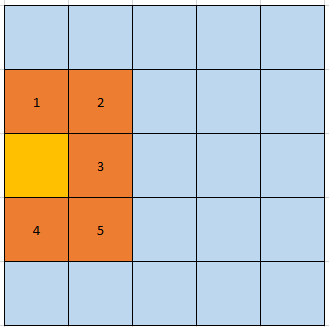
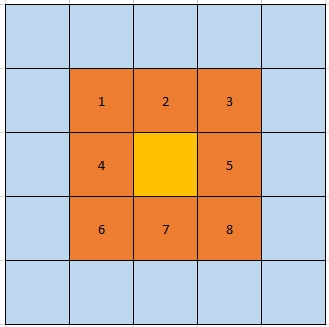


Figure 2 All Possible Cases

Nodes are also flagged as either walkable or unwalkable. This allows the A-star algorithm to navigate around obstacles, opening more gameplay options.

With this grid of nodes, the A-star algorithm can then be called when required to work out the best path to reach another node within the grid. Once a path is discovered, it can then be used by the agents within our game to navigate around the terrain. It does this by simply moving to each consecutive node within the path, until the final destination is reached.

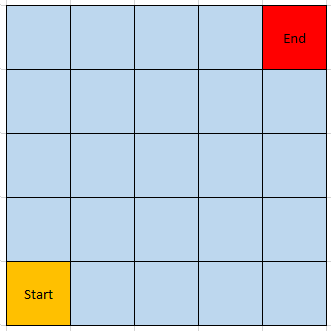


Figure 3 Start and End Node Selection

The algorithm begins when it is provided with both a start and end node. These nodes are identified from a helper function which can take a Vector3 position and return the closest node to that position.

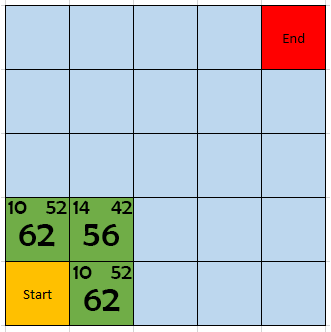


Figure 4 Potential Neighbor Node Evaluation

All neighbouring nodes of the start node are then evaluated to determine the most optimal step.

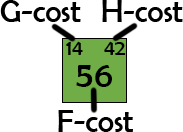


Figure 5 Node Cost Breakdown

G-cost: The total cost to reach this node.

H-cost: The total cost required to reach the end node from this node.

F-cost: The combined total of the G-cost and H-cost.

The heuristic we used to calculate these costs is Manhattan distance. An adjacent move costs 10, whilst a diagonal move costs 14 (**√ (**a2 + b2) = c2).

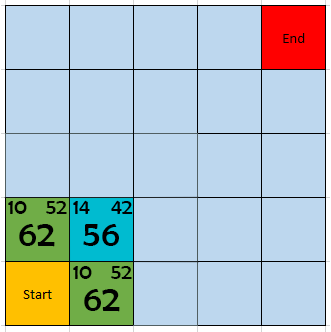
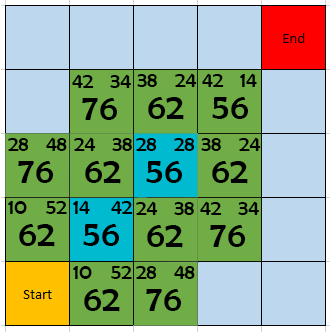
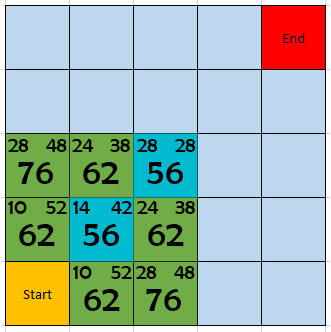
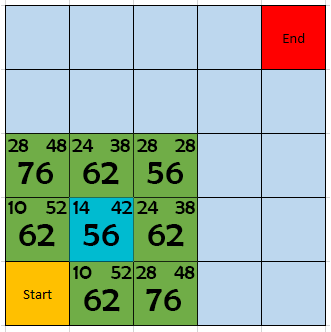


Figure 6 Optimal Node Selection

From this, we can determine the node with the lowest F-cost to be the most optimal node.

This continues until a complete path has been calculated as follows:



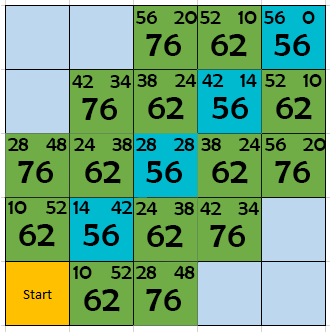
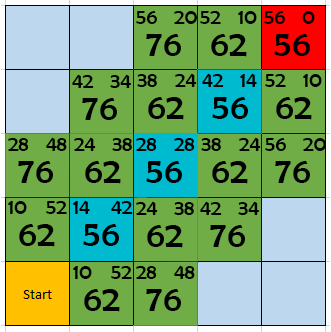
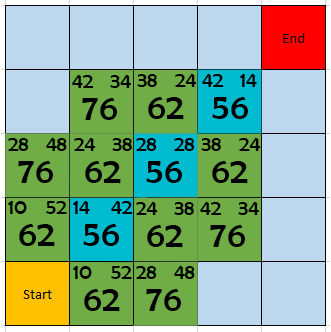


Figure 7 A\* Algorithm Step by Step

### Optimizations:

Removing Irrelevant Nodes from Path

Consecutive nodes which follow the same direction as the previous are unneeded and provide an easy opportunity to optimize the path.

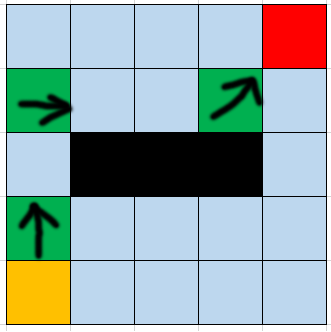
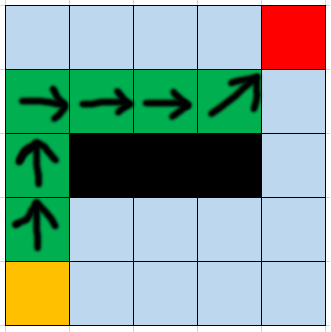


Figure 8 Node Optimization

Heap Data Structure

Nodes are stored using a heap rather than a list, allowing for faster iterations through the open set, as the first element can be guaranteed to have the lowest, or equal to lowest F-cost. This optimization provides a great reduction on path calculation time, making it an obvious choice due to our decision to run the game in real-time. This data structure is only valid for our situation since we are only ever interested in searching for the absolute lowest F-cost. A search for a different value would not be possible with this data structure.

Path Request Queue

Path requests are stored in a queue, rather than waiting for an immediate response. This prevents a drastic loss of FPS when several agents request a path within a small-time frame, allowing us to maintain the smoothness which would be both expected by the player and required for this style of game.

Dynamic Control

Rather than updating the grid of nodes at a regular time interval, they are updated only when the environment changes. This prevents any unnecessary recalculations which would simply provide the same result.

## Decision Making:

### Requirements

Decision making within the game is used to determine which attack the boss should perform at any given time. To decide on the best implementation to use, the following list of criteria was developed during the design stage:

1. The agent has many actions that it can perform.
2. Only one action can be performed at a time
3. A new action can start only after the previous action has finished
4. Decision of which action to perform must be based on both player’s data and agent’s data.
5. Algorithm must be easily adjustable after the initial implementation (e.g. swap attacks)

Based on these requirements we decided to implement a decision tree that is combined with a finite state machine (FSM).

### State machine

Each attack is designed as a state and has three flags: “started”, “in progress” and “finished”. Once “finished” the state is changed to “Stand by”. Only one state can be active at a time. Figure 9 illustrates a high-level design of the state machine. Transition from the “Stand by” to one of the “Attack” states is determined by the decision tree.

Figure 9 State machine

Decided on action (Decision tree)

Stand by

Attack finished

Attack

Attack A

Attack B

### Decision tree

Figure 10 demonstrates the high-level principles of the decision tree implementation. The decision maker takes both internal and external knowledge as input and outputs an action request. For our game purposes the action always triggers a state change.

Once the general concept was established, a more detailed diagram was developed. As more boss attacks were created, the tree gradually expanded and was modified multiple times throughout the testing period to provide better experience. The final design is shown on Figure 11.

No

Yes

Melee

Attack

Shoot 5 bullets

Shoot 3 bullets

Right hand grab

Left hand grab

Yes

No

Yes

No

Which zone player is at?

Left

Melee

Back

Middle

Was the last attack melee?

Was the last attack spawning enemies at left?

Was the last attack spawning enemies at right?

Was the last attack shooting three bullets?

Is player closer to the right hand than to the left hand?

Yes

No

Yes

No

Spawn enemies right

Teleport

Teleport

Spawn enemies left

Does player have 20 mana?

Magic shield

Physical shield

Right

Yes

No

Figure 11 Detailed decision tree

Figure 10 High-level decision making

Decision Making

\* Agent’s position

\* Previous attack

\* Player’s position

\* Player’s mana

\* Attack state

Internal knowledge

External knowledge

Action request

### Optimization

N-ary elements

As shown on Figure 10 player’s position always influences the decision. Therefore, it was decided to use an N-ary tree approach for the first check. From the implementation point of view, all the zones are labeled using an enum class, which allows the use of a switch statement. This way the tree required less comparison when compared to a sequence of if statements and is therefore more efficient.

Balanced tree

There was an effort to balance the decision tree which resulted in the average of two checks before the decision is made. The only exception is the melee branch of the tree, which has an additional check for the shield type. Though it eliminates a hundred percent probability of the perfect O(log2 (n)) time, this features enhances the gameplay.

Additionally, the costlier actions are located closer to the leaves. For example, in the melee branch, the agent checks the internal knowledge first (previous attack) and only then requests external information from the player’s script (mana left).

## Learning:

When coming up with the learning we used in the game we eventually decided on using Naive Bayes algorithm. However, we originally decided to use the ID3 algorithm as it can be used in tandem with decision trees. From this we created a modular ID3 algorithm that in theory could be used by any decision tree. Unfortunately, having created the decision trees separately from the ID3 they were unable communicate correctly due to the differences in the data the two approaches were expecting. We then decided use Naive Bayes to do this because it was easy to set up and due to the fact, the algorithm requires only a small amount of training work. The learning method is used by the AI boss agent to assess which of its attacks have connected with the player the most and then doubles the damage of that attack. The data is collected when the boss starts an attack and when the attack connects with the player. This data is then stored in a list that is used to train the boss when a set amount of damage is done to it.



Figure 10 The Naive Bayes algorithm used to calculate the Posterior probability that an outcome would accrue

The equation in Figure 12 shows how the algorithm works out the probability that something would occur based on the information that it already has access too. Naive Bayes is a collection of algorithms that are based on the common principle that any feature is independent of the value of any other feature. This meaning that a feature is known by the values applied to it. In the game we do this by looking for what if the ability has hit and if it is an attack. The returned probability that those would relate to a specific attack is what we use to then decide what the boss will do. The boss then finds the attack with the highest probability of matching those values. In the implementation we used, there are three main functions that are needed to get the algorithm to work. These are:

* Teach – This function takes a list of data and stores the feature as the key and the associations as the values.
* Calculate probability – This function is used by the classify function and takes a label and the features being searched for. It returns the probability that the feature would be related to those associations.
* Classify – This function returns all the probability that the searched values would be linked to a feature using the calculate probability and the information gained from the teach function.

The functions must be used in a specific order that being because the classify function requires data that is provided by the Teach function and without that data it wouldn’t be possible to check the probability a hit had happened.

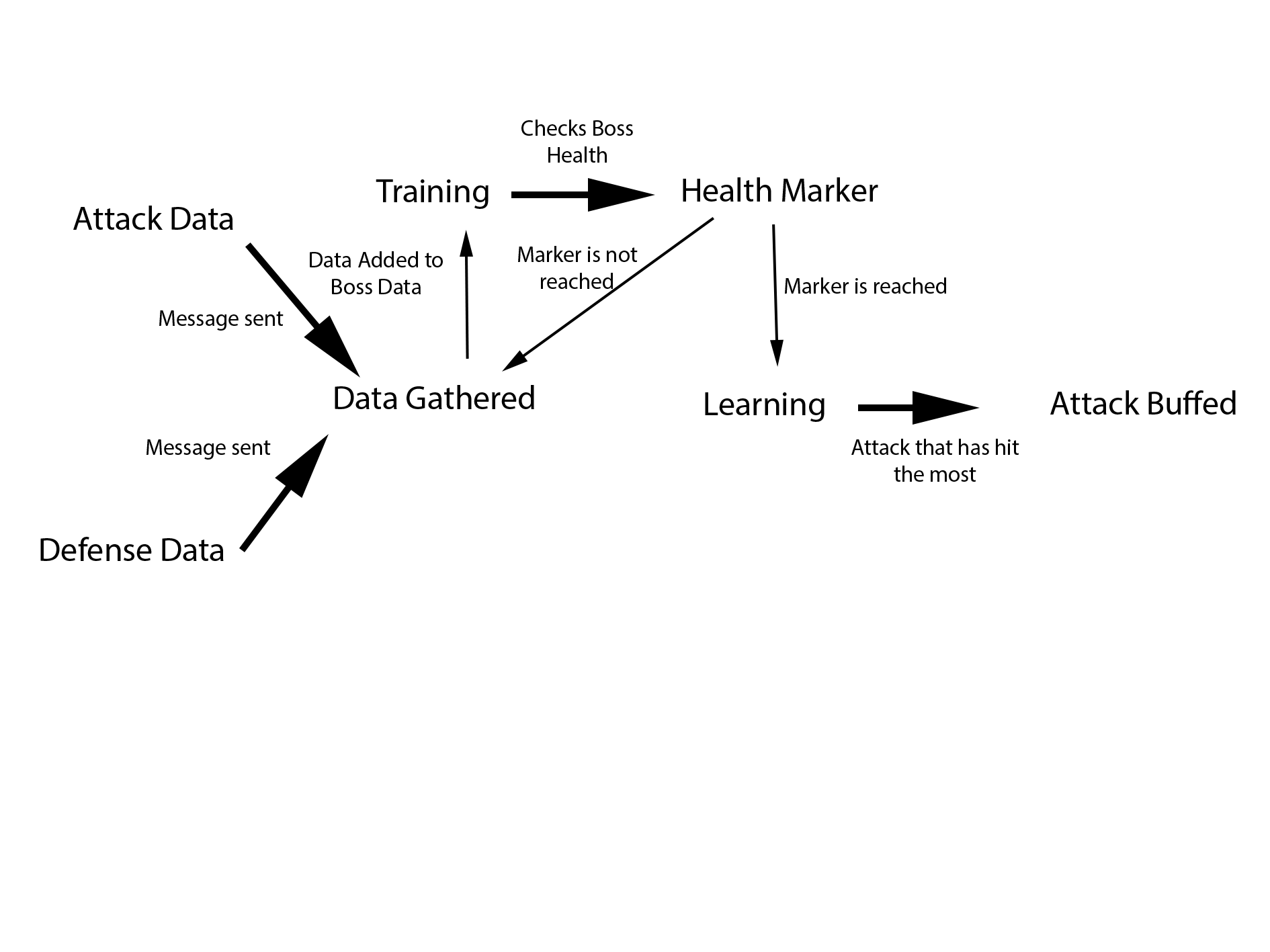


Figure 11 The image shows a diagram of how the learning works within the game

Figure 13 shows how the boss in the game figures out which attack would be best to have its damage increased. It does this by looking at all the player data and works out based on the data which attack will most likely hit the player based on the data collected over the round.

# Implementation Issues

As in any team project, ensuring that all the group members are working on the correct version of the project required us to use source control. After the initial discussion it was decided we would use gitkraken to store the repository as everyone was familiar with the software. This decision allowed us all to work on the same project without causing merging issues. However, to achieve this we eventually had to implement a check in/check out system as merging branches caused many merging conflicts. The changes were logged both in the differences in check ins and our comments in the code.

## Pathfinding:

No technical implementation issues other than ensuring the algorithm was working correctly. This can be attributed to it being the first time any of our group had to implement the A-star algorithm.

## Decision Making:

After the code for the state machine was in place, it was relatively easy to build the decision tree. The only issue was testing that all the paths worked correctly, which had been very time consuming, due to the size of the tree. Another challenge was modifying the tree to accommodate for the testing results. The process was simplified by extensive refactoring, commenting and the regularly updating the documentation diagram.

## Learning:

When creating the learning for the project one of the major implementation issues that we faced was implementing the ID3 algorithm with the decision tree created for the boss behaviour. Due to the difficulties for trying to merge the two we decided to step away from ID3 and instead use Naive Bayes algorithm as it better fit our needs for the project. This also influenced another area that became another issue. The other problems we had during the implementation of learning was deciding what we wanted the boss to learn. This was problematic as it prevented us from deciding what the best option to use was and therefore what data we needed to collect. To solve this, we looked at what we had and decided to have the boss log its attacks so we can then use that to increase the damage of the most successful attacks. Another issue was the segmented approach to coding as it caused clashes in coding styles and differences in how we solved issues. To overcome this, we decided to focus on specific areas and anything related to that was then tasked to that person.

# Results

## Pathfinding:

The choice of A-star was justified, as the pathfinding in our game resulted in the best path, with a path computation time which did not impact on gameplay at all. This is true for both the player character, and AI agents. Below I have visualized the different elements of the pathfinding. Additional obstacles have been included for demonstrative purposes.



Figure Obstacles In Game View

Grid nodes are coloured to show walkable/unwalkable terrain.

Green - Walkable.

Red - Unwalkable.

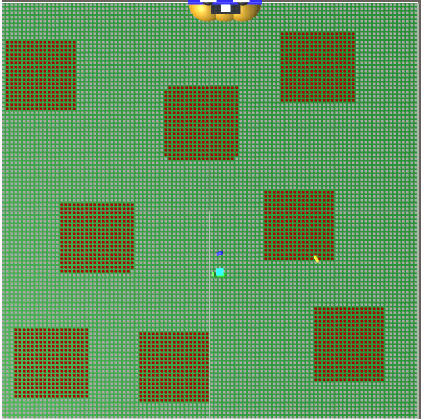


Figure Grid of Nodes

Path calculation disregards unwalkable nodes. Provides the shortest path without intersecting unwalkable nodes.

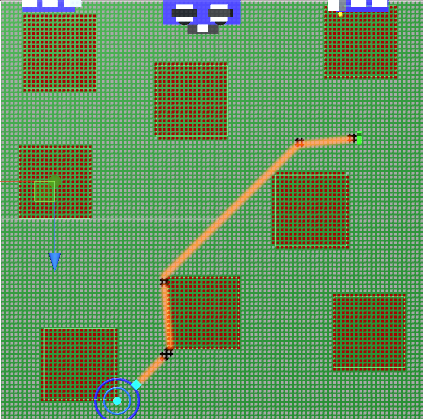
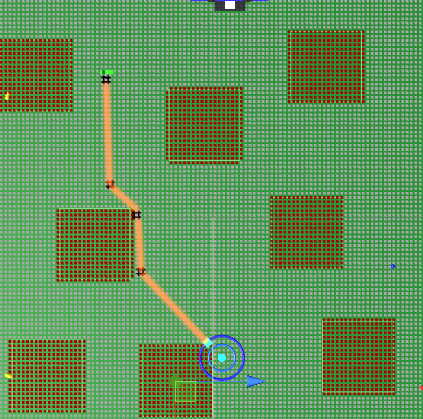
** **

Figure Path Examples

## Decision Making

**A screenshot of a computer screen

Description generated with very high confidence**The implementation of the decision-making algorithm provides unique boss behavior and greatly expands the replay ability of the project. It can be easily expanded or modified, which allows good potential for future work. Currently there are 10 different actions which can be performed by the boss based on a combination of two internal and two external to the agent factors. Screenshots below show some of the in-game attack examples.

Figure 15 Boss performs left hand grab

**A screenshot of a computer screen

Description generated with very high confidenceA screenshot of a computer screen

Description generated with very high confidence**

Figure 16 Boss shoots 3 bullets

Figure 17 Boss spawns enemies from the right

**A screenshot of a computer screen

Description generated with very high confidence**

Figure 18 Boss teleports

## Learning:

Due to the ease to build the Naive Bayes algorithm and the small amount of training needed to run the algorithm meant it was the perfect match for what we wanted. The results gained from the classify function are accurate and have allowed us to implement the double boss attack damage based on which attack was the most successful. Due to the real-time nature of the game and the inability to predict the amount of data that will be gathered each game the savings made on computational power using the Naive Bayes algorithm means there is no noticeable effect on the game play. Figure 19 shows the output from the boss when a specific amount of damage is done. The output shows the name of the attack that gets buffed.



Figure Output of value when bosses health reaches health marker

# Conclusion

In conclusion, looking back at what we developed for the project we could create a game that contained all the elements of AI that was requested. The process of implementing the AI in the project may not have been a smooth process as covered in the report but we were able to come together as a team and solve all the issues that arose. The main thing that is shown from our work is that we could pick the best algorithms for what we wanted to achieve.

# Peer Assessment

|  |  |  |
| --- | --- | --- |
| Peer Name | Percentage Grade | Comment |
|  |  |  |
|  |  |  |