Pathfinding for Dynamic Platformer Game Environments

Tokonaziba Kalango (200300469)

BSc Computer Science, May 2023

Word Count: 14,407

Abstract

This dissertation explores various ways to create a pathfinding algorithm that takes map changes into account.

It describes a solution that efficiently solves this problem, and how the solution can be used to create predictions in the context of platformer games. The results are presented with an analysis of the design’s possibility to become a library for these games.

Declaration

"I hereby declare that this dissertation represents my own work, except where otherwise stated."

Signed: ………………………………..

Dated: …………………………………

Acknowledgements

I would like to acknowledge and thank my supervisor, Dr. Giacomo Bergami, for his willingness and availability whenever I needed guidance with my project.

Table of Contents

Chapter 1: Introduction.

1.1: Dissertation Structure

This Dissertation is split into 6 Sections:

1. The Introduction. This chapter outlines my rationale for selecting this project, my plan for tackling the defined problem and how I will mitigate the risks that may be involved.
2. Background Research. In this section, I will highlight some key information I found that provided general guidance in my design decisions.
3. System Design. As this project focuses mainly on implementation, this is the longest section. It describes the various options I considered for each stage of the design and how each choice affects other design decisions.
4. Testing. This section will provide assessments of each individual design option in section 3 to justify the given solutions.
5. System Evaluation and Plug-and-Play functionality. After the tests have been done on individual components and on the whole system, this section will discuss its effectiveness and efficiency. It will consider the extent that the system accomplished its goal and any additions required to extend the solution into a library.
6. Reflective Report and Conclusion. A personal account of the aspects of my workflow that helped me, and those I believe I could have done better.

1.2: Context for the Project

Artificial Intelligence (AI) is a prominent aspect of modern video games. Non-Playable Characters make some use of AI within their programming to make the game more challenging for the player. AI's functionality in games can broadly be split into 2 areas: finite state machines and pathfinding [1]. The AI uses finite state machines to adapt to its surroundings and can go a step further and use Goal Oriented Action Programming (GOAP) to reach a desired state [2]. Often times, AI agents would also need to be able to know how to move around a game map. This is the pathfinding aspect, and it comprises of 2 steps:

1. Defining the areas where the AI can move along.
2. Setting some heuristics for how to move around that area.

1.3: Definition of the Problem

The problem with the current AI technology in games is that the pathfinding workflow is done almost exclusively in the precomputation phase of the game. Unity (the chosen technology for this project) [3] uses a Navigation System that is made up of 4 main components:

- NavMesh, short for Navigation Mesh, denotes the area that the AI can walk on (step 1 in 1.2)
- NavMesh Agent defines the characters that can walk on that mesh. Typically, this would consist of the players and the NPC agents.
- OffMesh Links represent certain actions that can be done at points around the NavMesh to give its agent some guidance on how to traverse that area. This could include things like jumps or opening doors (step 2 in 1.2)
- A NavMesh Obstacle is a game object that would alter the mesh shape when they come in contact. The NavMesh agent would have to plan to move around that obstacle.

While this existing navigation system is effective at setting the traversable areas and actions for pathfinding, the only map change it can handle is the appearance of obstacles. The NavMesh object and its links are completely static and cannot change at runtime. For the AI to be able to adapt to constant map changes other than obstacles, one would have to manually change the geometry of the NavMesh in the code, a solution too costly to do at runtime. That means that it would be inadequate in environments where the map changes often, which is a gap in industry technology that can be addressed.

1.3: My Proposed Solution

I’ve decided to tackle this problem in the context of a simple video game genre: Platformers. A platformer game is a “style of video game where the player makes a character move through an environment with a series of action-based moves” [4]. This context is an easy starting point because of how basic the geometry of a platform is and the limited number of actions an agent can perform. On the highest level of logic, you can simply define the “traversable areas” as everywhere a few units above a platform floor, and the general area directly below it (of course, there’s more to it). Therefore, I aim to design a system where the AI can “look” at the map, decide for itself where it can or cannot go and ultimately find the shortest path from 1 platform to another… all in real time.

My solution consists of creating a graph data structure where the node/vertex would represent a platform, an edge would represent whether you can get from 1 platform to another, and the cost of that edge would be the distance between them. With that graph, I will be able to update node positions, and consequently edge costs, in order to run the pathfinding algorithm even if the game map should change.

1.4: Project Aims

With the general scope of how I want the system to end up, it's important to break down the major steps into smaller, more manageable ones so that I can keep track of progress and reduce the chance of feeling too overwhelmed. The steps can be broken down into major milestones for the system development which are as follows:

1. **Create nodes out of platforms.** This will be the building block of the system, seeing as the project mainly focuses on pathfinding and the graph that will use the algorithm is made up of nodes.
2. **Create edge system.** The next important step after making the nodes is connecting them together. The edge system would dictate how a platform should determine its neighbouring platforms.
3. **Create cost system.** With those available edges, it's important to know the cost of going from 1 platform to another, in other words the edge weight. This will be distance based such that the further away 2 platforms are, the more it will cost to move between them.
4. **Select graph data structure.** With the 2 components of the graph created, the system needs a dependable data structure to store the graph information. The key features to consider when making this data structure are its memory use and query speed – in order to keep the system efficient.
5. **Design method for AI to navigate map.** This step is independent of all the others. Whatever data structure or traversal algorithm the system uses, the AI still needs to have a set of instructions to follow to know how to move around the map.
6. **Design graph traversal algorithm.** With the given data structure, there also needs to be a way to search the graph and find a path from 1 node to another.

7. **Teach AI to detect map changes.** Once the agent can navigate its way around a static map, it then needs to be able to use that logic to know what to do when platforms move.
8. **Explore further additions.** Continuing to move in a dynamic map is one thing, but a most intuitive AI should be able to use such changes to its advantage to find "shortcuts". This objective isn't necessarily to design such a system, rather to consider how it might be done with the tools from the existing system.

1.5: System Requirements

Aside from how to go about designing a system, it is important to have an idea of its features. I made a "checklist" of all the things my system must have by using 2 requirement types: functional (quantitative attributes) and non-functional (or qualitative attributes). Functional requirements (FR) specify all the things a system needs to be able to do, and non-functional requirements (NFR) describe how it can do those things proficiently [5].

My requirements can be grouped as follows:

1.5.1: Game Environment

FR1. Game map must have platformer attributes, namely gravity and various platforms.

FR2. Game should allow for a change in platform information (platforms should be able to move, disappear and reappear).

NFR1. The AI should be able to detect map movement.

1.5.2: Pathfinding System

FR3. System should be able to find the shortest path from 1 platform to another.

NFR2. The algorithm shouldn't do any more searching than it needs to in order to find a path. Minimum searching.

1.5.3: Agent Movement

FR4. The AI agent should follow the path given to it by the pathfinding system.

NFR3. The system must mimic real reachability area.

NFR4. AI movement should be fluid and player-like within the system

1.5.4: Efficiency

NFR5. The game should be able to run at 30 Frames per second (FPS).

The requirements for the agent movement aren't as crucial to the rest of them because the project is mainly centred around pathfinding. Refer to section 1.7 for more detail.

1.6: Project Plan.

With only 10 weeks to complete this project, it's imperative to pace my workflow along the way to not spend more time on a milestone than I should. Figure 1 illustrates the plan for the project.

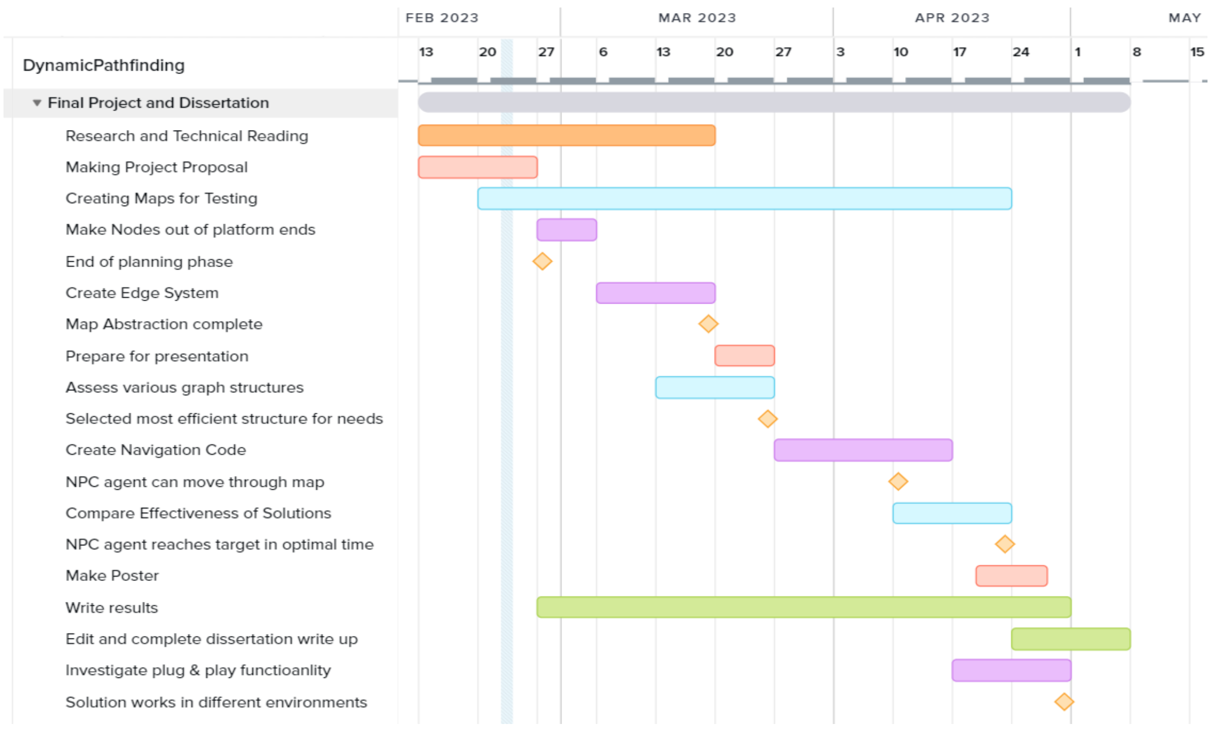


Figure 1: Project Plan Gantt Chart.

1.7: Risk Analysis

The primary risk of this project is the possibility that I would be unable to complete it within the allocated 10-week period. The best way to mitigate this risk is by sticking closely to the Gantt chart plan and prioritising the most important functions in the system. While it is important for the AI to be able to move around the map, the base function is for the system to be able to find the shortest path. Navigating around the map is a secondary feature.

Chapter 2: Background Research.

My research mainly comprised of understanding the mechanics of how platformer games operate and various pathfinding implementations.

2.1: Game Design

2.1.1: Platform Types

There are many types of platforms in platformer games [6], including but not limited to:

- Standard platforms which are horizontal, unbreakable, unmoving and solid all the way through. Examples of these platforms are the ones in Fireboy and Watergirl.

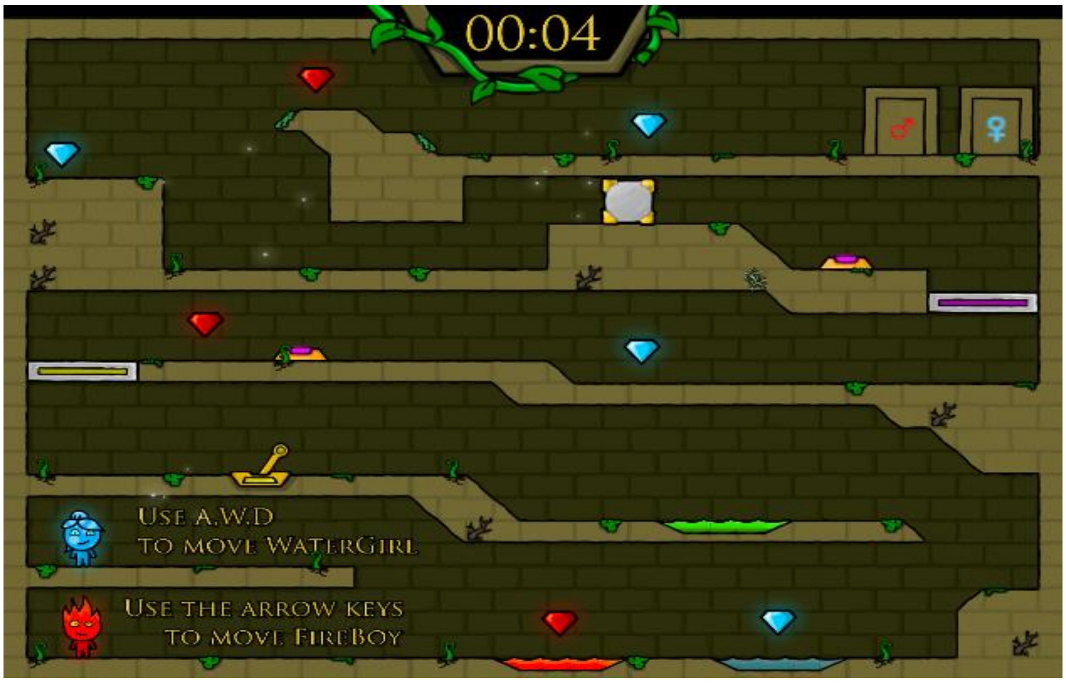


Figure 2: Fireboy and Watergirl

- Jump through (or soft) platforms, which can be a variation of any platform. The bottom of the platform isn't solid, and the player can go through it from the bottom up. The green platforms in Doodle Jump are jump through.

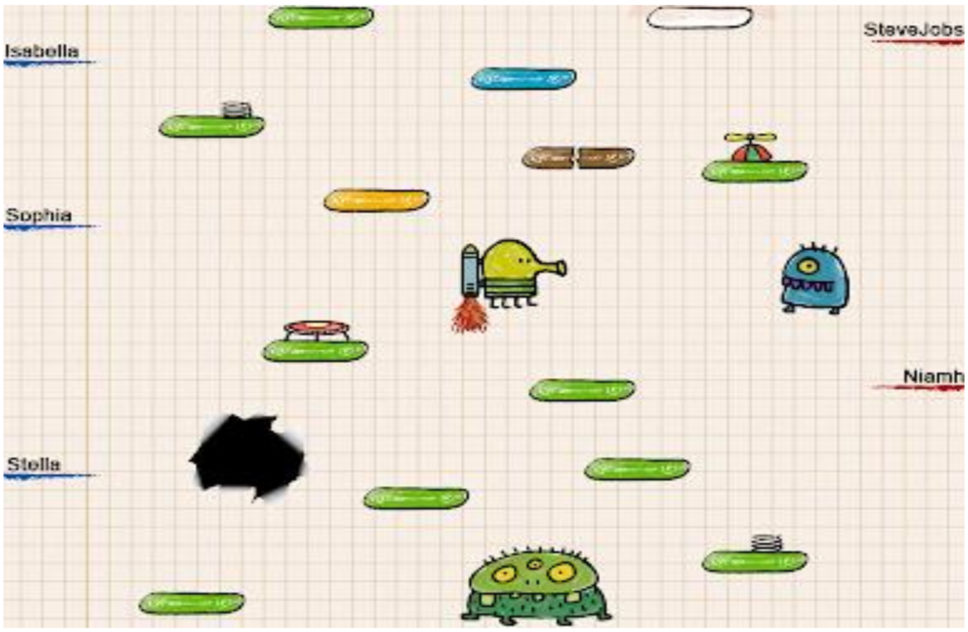


Figure 3: Doodle Jump

- Moving platforms. These normally either go back and forth or just go continuously in 1 direction. The blue platforms in Doodle Jump, for example, move back and forth.
- Disappearing/Falling platforms which can't be walked on more than once. The brown platforms in Doodle jump are falling, but you can't stand on them at the start. The Hex-A-Gone level in Fall Guys consists of these falling platforms.

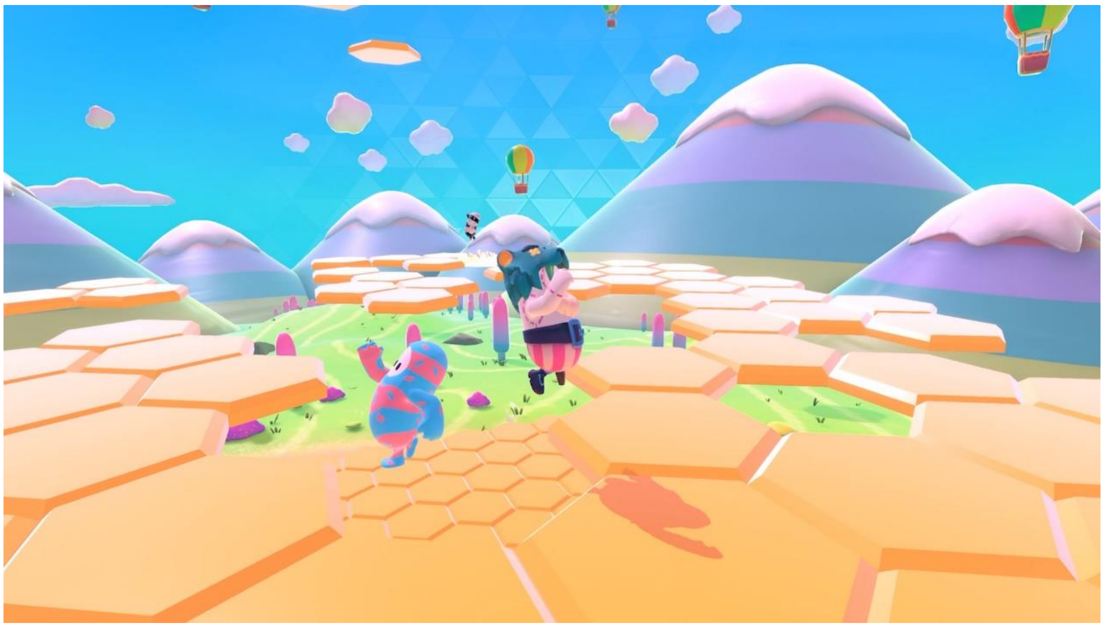


Figure 4: Fall Guys

- Slippery/Sticky platforms where there is less/more friction on the platform surface respectively. Normally, sticky platforms are used on walls, so the character falls more slowly. Multiversus has this feature for the walls of the main platforms.

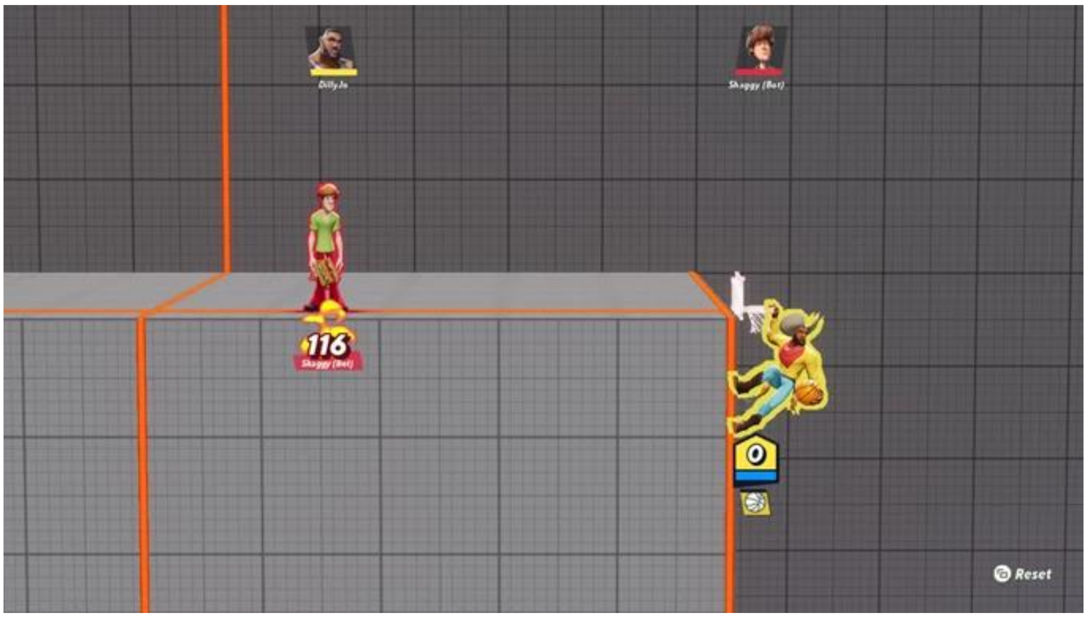


Figure 5: Multiversus

- Conveyer belts/bouncy platforms. These give the player more velocity on the x and y axes respectively. An example of a bouncy surface is the springs in Fancy Pants.

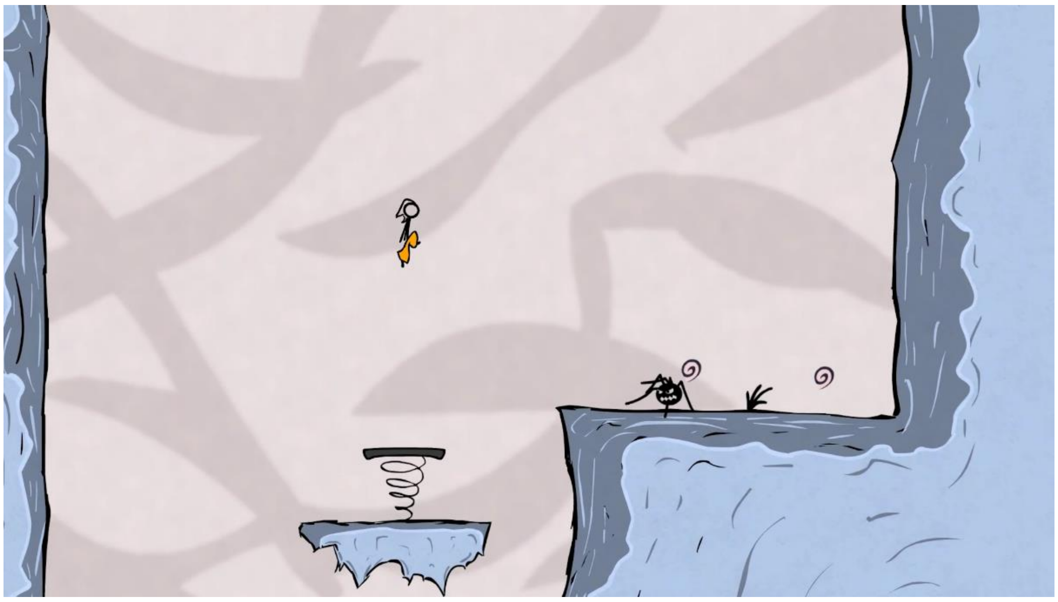


Figure 6: Fancy Pants

2.1.2: Adding Difficulty to Platformer Games

Different combinations of platformer types make for a completely different game feel, and the allure of the game often comes from using unique combinations. Most games have mostly standard platforms (or jump through variations) and will usually include 1 or 2 other types. After doing some research on the various natures of platforms, I thought the number of platforms in a game would also be relevant information to consider. I found that most games would have less than 25 platforms in a 2D environment – aside from Doodle Jump which has practically an infinite number.

Both player vs environment (PVE) and player vs player (PVP) subgenres don't heavily rely on having a lot of platforms to make the game interesting. Rather, they do this by adding a lot of obstacles or a lot of enemies. Plazma Burst, for example, is a PVP game where the goal is to reach a certain door while trying not to die from the enemies shooting at you.

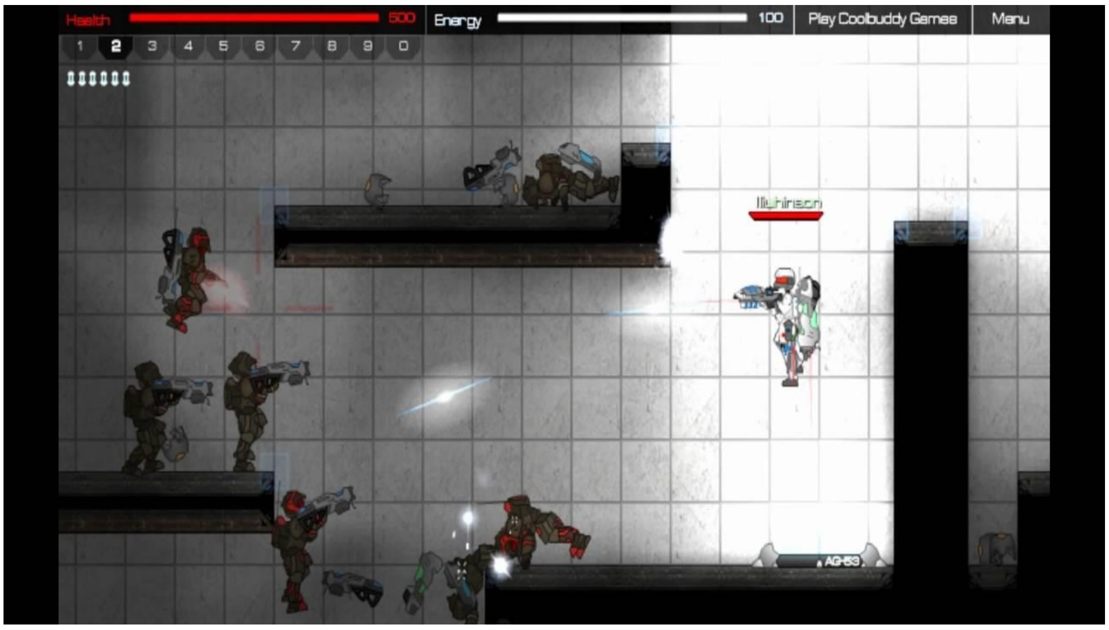


Figure 7: Plazma Burst

While the game is difficult enough to be interesting as it is, it doesn't take much to realise that, as a player, you can simply walk out of the range of the enemies because their movement mechanics are very limited. They do chase you, but without guns it would be nearly impossible for them to catch you unless the level is just a plain surface.

A PVE platformer, such as Ketchapp's Phases (Figure 8), has a similar but opposite limitation.

There are few platforms and a lot of obstacles, a design choice that makes it quite hard for a player to even get to the next platform.

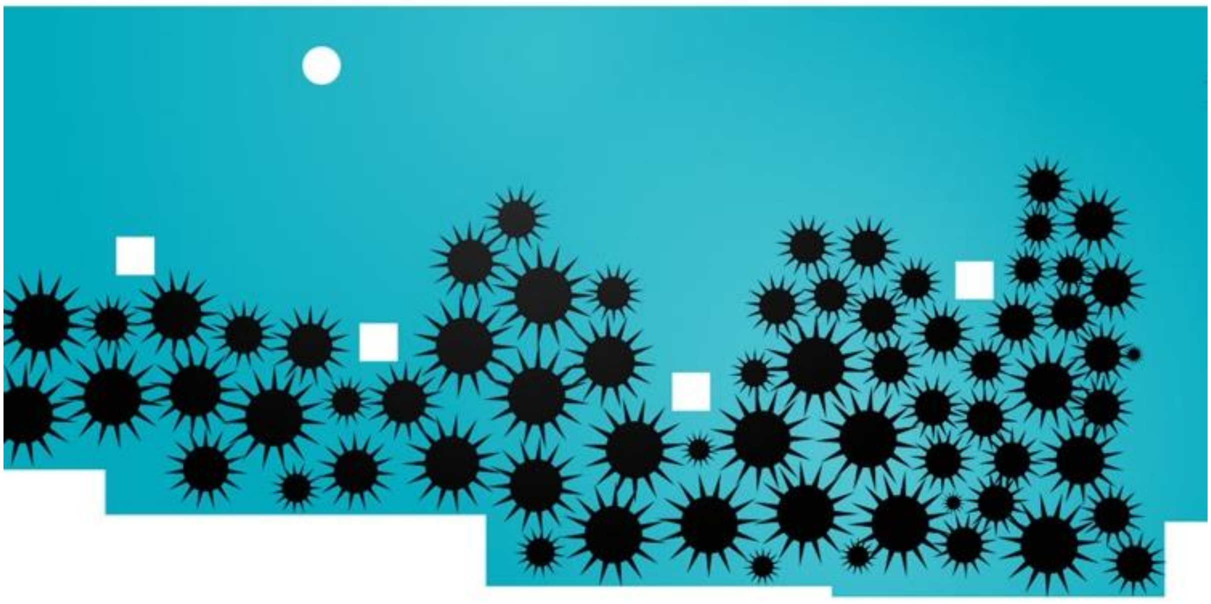


Figure 8: Phases (White ball is player)

While the game has impressive and intricate map designs, they are almost always gone to waste because the player can just wait on one platform for as long as he/she wants. This game doesn't have AI but a concept like this could be heavily improved on if there was some AI chasing you around. As of right now, platformers focus almost exclusively on other aspects to make the game more difficult and a lot of the times these aspects have loopholes as seen in the 2 previous examples. My solution makes for more intuitive AI that can follow the player around accurately, therefore adding more pressure and providing a more interesting play experience.

2.1.3: Agent Reachability

Typically, a platformer agent (either player or AI) would only be able to jump once off a platform, but certain games have a "double jump" feature that allows for jumping in mid-air. Due to gravity being a prominent factor in platformers and the agent's ability to only jump, reachability on the y axis is as straightforward as described in Section 1.3. An agent's x axis reachability, on the other hand requires a bit more consideration. This feature is illustrated best in figure 9, where the shaded blue area represents everywhere a player can reach in relation to a standard platform.

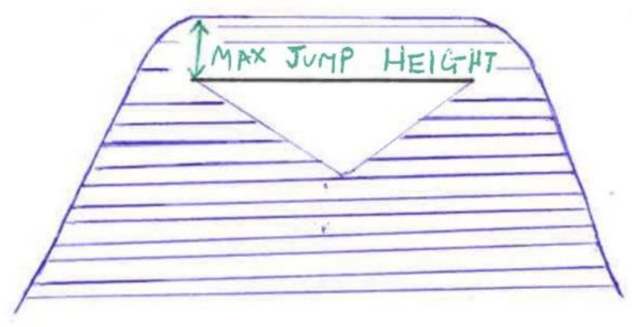


Figure 9: Reachability from a platform

One of the most noticeable features is that not all the areas below the platform are traversable. This is because the agent can't traverse downwards through the middle of the platform (although some soft platforms allow that). Instead, it must always jump off the platform edge and then it can eventually end up directly below it because of characters can move in freefall. The exact distance, D, below the platform where the agent can be directly below its centre can be calculated with the following equation:

$$D = \left(\frac{agent\ speed}{0.5\ platf\,orm\ length} \right)^2 \cdot\ gravity$$

Figure 10: Distance below platform equation

This freefall movement is also the reason why the reachability area looks like an isosceles trapezium (is angled).

2.2: Graph Structure

A graph is the foundation of a pathfinding algorithm. In simple terms, a graph is a mathematical structure consisting of a set of vertices which are interconnected by a set of edges [7]. Edges can have 2 main properties: weight and direction. A graph is said to be weighted if there is a notion of cost associated with each edge and directed if the edge has a tail and a head, demarcating where the edge starts and ends. In a directed graph, there might be an edge from node A to B, but not necessarily vice versa. It is also possible to have an edge where the head node and the tail node are the same, and it leads to itself. This is called a "hook" or "self-edge".

2.2.1: Graph Representation Types

There are typically 2 main ways to abstract a weighted, directed graph into a codebase: an adjacency matrix and an adjacency list [8]. The adjacency matrix has a rather simple concept behind it, you create an array of vertex pairs out of every combination of 2 vertices and then you set the value for the pair as the weight between them. In an unweighted graph, this value would either read 1/true or 0/false as to whether or not there exists an edge between the vertices.

$$Adjacency\ Matrix: [[V][V] = W]$$

Figure 11: Adjacency Matrix

An adjacency list is a bit more complicated; it abstracts the graph as an array of linked lists. In other words, each vertex would have a list where each item would be a connected vertex. This concept is illustrated in Figure 12 [9]

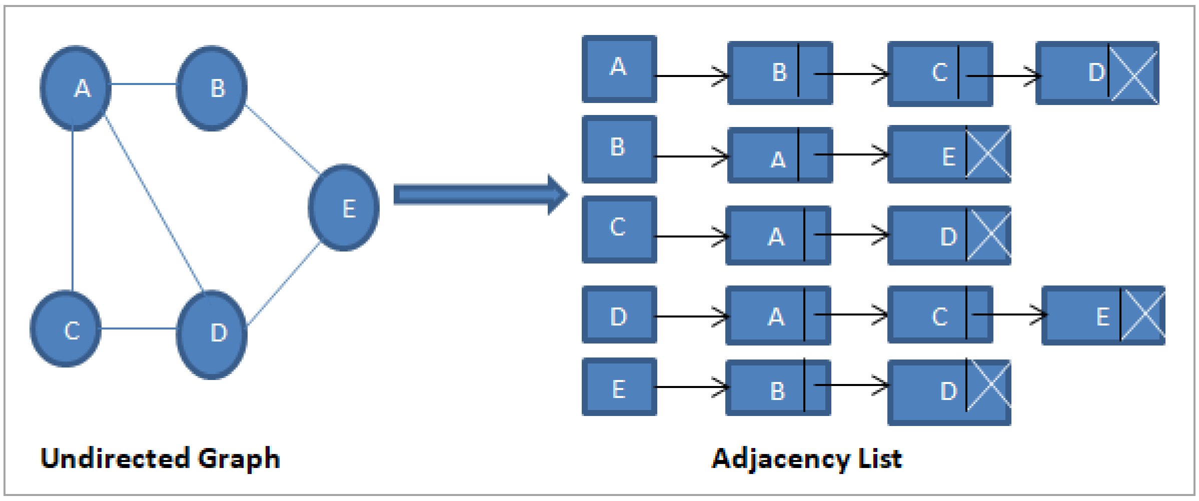
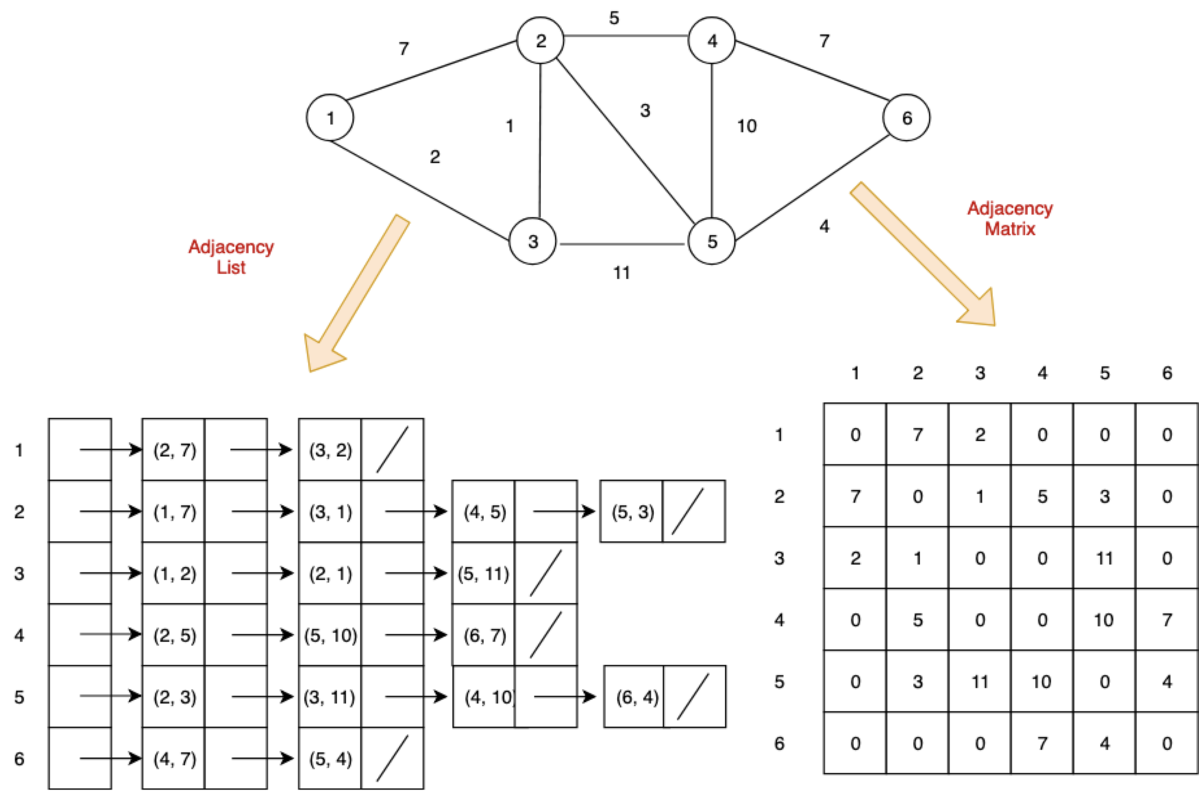


Figure 12: Adjacency List

With the basic idea of how both data structures work, I began to make quantitative comparisons between the 2 to analyse their strengths and weaknesses and choose a suitable option for the project.

2.2.2: Space Complexity

A dense matrix (one which stores all edge values) has storage complexity $O(|V|^2)$, where |V | is the cardinality of the set of vertices. The list on the other hand has a space complexity of $O(|V| + |E|)$, where |E| is the cardinality of the set of edges. At plain sight, the adjacency matrix looks less favourable because it would be populated with many 0/false values (unless it uses a sparse matrix which isn’t available in C#) but that assumption is incomplete because the edges don’t have any weight yet. Figure 13 [12] illustrates how the space complexity will change once weights are introduced into the graph.



	1	2	3	4	5	6
1	0	7	2	0	0	0
2	7	0	1	5	3	0
3	2	1	0	0	11	0
4	0	5	0	0	10	7
5	0	3	11	10	0	4
6	0	0	0	7	4	0

Figure 13: Adjacency Matrix and List Comparison

The difference in how the Adjacency list stores edge information becomes clear immediately. Instead of the graph being a list of lists (list of vertices and list of each connected vertex), it becomes a list of lists of lists, because each vertex is now a list where the second item is the weight. While its size remains $O(|V| + |E|)$, the V has gone from being an integer to being a list. The adjacency matrix, on the other hand, still maintains a size of $O(|V|^2)$ and its vertices don’t take up any more memory even with weights. This is because weights are (sort of) already stored in matrices, the only difference is that instead of the values being limited to 0 and 1, it can range any number.

2.2.3: Time Complexity

The other main aspect to consider between both data structures is how efficient their reachability queries i.e., how fast they can find a path between a given source and destination. This operation is better defined as a structure’s “Time Complexity”. Due to the nature of a list as a data structure, the earlier an item appears on that list, the shorter it’ll take to access it [10]. What this means is that the list searches for items in the order that they appear, and therefore has a linear scan that costs O(|L|), where |L| is the cardinality of the list. It’s a lot faster to use matrices for associating a vertex to its set of edges. Because each edge connection is stored by row, accessing outgoing edges constantly only costs O(1). Visually, trying to find the cost of edge 1->4 would vary for both structures as shown below:

- matrixCost = graph[1][4]
- listCost = graph(1).Find(4)

Would get graph(1) of 2, then 3 to finally arrive at 4. It looked at 2 vertices before getting to the goal vertex.

2.2.4: Other Considerations

While storage and query speed are the 2 main considerations, there are other operations that can help make an informed decision [11][12].

	Adjacency List	Adjacency Matrix
Space Complexity	O(|V| + |E|)	$O(\|V\|^2)$
Time Complexity	O(|V|)	O(1)
Access Outgoing Edges	O(|V|)	O(1)
List all Edges	O(|V| + |E|)	$O(\|V\|^2)$
Add Vertex	O(1)	$O(V^2)$
Remove Vertex	O(|V| + |E|)	$O(\|V\|^2)$
Add Edge	O(E)	O(1)
Remove Edge	O(|V| + |E|)	O(1)

Figure 14: Operation costs for graph representations

2.3: Graph Reachability Queries

Functions such as accessing outgoing edges are useful for pathfinding, but only when put into a reachability query [13]. A reachability query is simply a search to find the shortest path from a source node either to every other node or to a destination node (in this case the system is more interested in the latter). Some of the most popular of such queries include Dijkstra’s Algorithm, A* Search algorithm, Breadth First Search(BFS) and Depth First Search (DFS). Dijkstra’s algorithm finds the shortest path between a node and all other nodes. It does this by keeping track of which nodes the search has or hasn’t already visited and then visiting the nearest unvisited one. A* extends this search by using a set of “heuristics” to optimise the search. Heuristics are intuitive strategies put in place in order to optimise an algorithm – a heuristic can be considered to be a “rule of thumb”. What the A* heuristic does is estimate the distance from each node to the goal node, therefore looking for almost by working backwards from the goal node. The concept of heuristics would be extremely useful in this project because it can help with NFR2 by eliminating any unnecessary checks. A* is more suitable than Dijkstra's for the project because the system doesn't need the lowest cost path to each platform, only the target node.

The other 2 searches, BFS and DFS, also find the shortest path to a given node. They do this by checking every connection to each node, and then the connections to those connections etc. The difference between the 2 searches is the order they are done in. In BFS, the algorithm visits each connection of 1 node before checking the other node in that iteration whereas DFS checks that node's child, child's child etc before it checks the neighbouring node. In theory, DFS is faster when finding nodes that are further away because it basically looks as far as it can, 1 direction at a time. BFS, on the other hand, is faster if the target is near the node because the search looks around in all directions around a node before doing the same for the next 1.

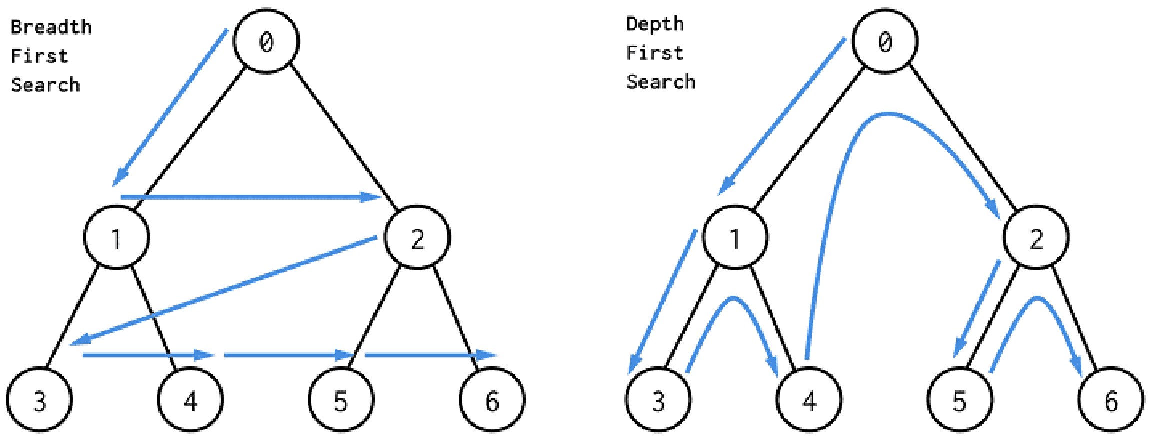


Figure 15: BFS vs DFS

Chapter 3: Implementation.

3.1: Map Design

For my project, I used Unity to create various map scenarios which imitate standard, disappearing and moving platforms[13].

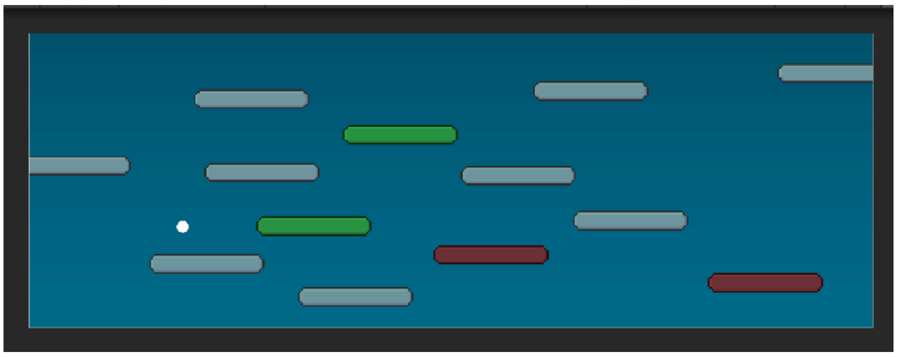


Figure 16: Map example

I chose those platform types because the cost to traverse a series of them remains constant. In other words, these platform types don't have a feature on the surface that would affect the pathfinding calculations. If a platform had surface effects, the system would need to also take the additional cost of traversing the surface into account. For example, it would need to associate a traversal cost for a direction on a conveyer belt because the platform would have a force moving that agent in that direction. Similarly, edge calculations would be different when using a bouncy platform because an agent would be able to jump to heights higher than it would have ordinarily been able to. While edge costs may vary with surface effect platforms, they still operate on dynamic maps and therefore won't prove much for the project. Disappearing and moving platforms, on the other hand, are more interesting to explore because they make the map dynamic, and edge weights still increase/decrease as platforms get closer and further from each other. These changes are more than enough to test my solution on and still maintain shortest path computations.

This environment meets FR1 and FR2. In terms of testing the environment, I created 2 basic operations a user can do:

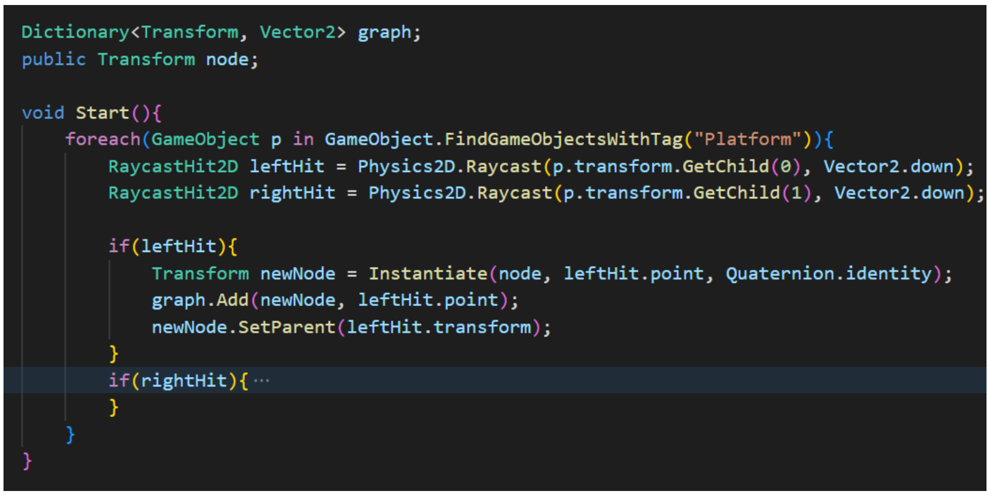
- Left click above the platform you want the AI agent to move to.
- Right click in empty space to create a platform or right click on a platform to make it disappear.

3.2: Creating the Node System

Since pathfinding is the primary matter of my project, it was imperative to create a reliable and efficient model for how I designed my nodes. I came up with 2 separate models: a "Key Point" system with more nodes and a "Platform" system with fewer. I had to be particularly thorough when comparing both systems because the rest of the project relies on it. I mainly considered how the graph would represent and store the nodes, how well I personally understood the chosen system and how easy it would be to implement the AI movement across the map.

3.2.1: System A: The Key Point System (KPS)

This system is high level and not too abstract. It effectively notes each position of interest on a platform and defines it as a "keypoint" for the AI to traverse to or perform some action at. Figure 17 & 18 are programmatic and visual explanations of the system respectively.



```
Dictionary<Transform, Vector2> graph;
public Transform node;

void Start(){
    foreach(GameObject p in GameObject.FindGameObjectsWithTag("Platform")){
        RaycastHit2D leftHit = Physics2D.Raycast(p.transform.GetChild(0), Vector2.down);
        RaycastHit2D rightHit = Physics2D.Raycast(p.transform.GetChild(1), Vector2.down);

        if(leftHit){
            Transform newNode = Instantiate(node, leftHit.point, Quaternion.identity);
            graph.Add(newNode, leftHit.point);
            newNode.SetParent(leftHit.transform);
        }
        if(rightHit){…
        }
    }
}
```

Figure 17: Code snippet of Keypoint function

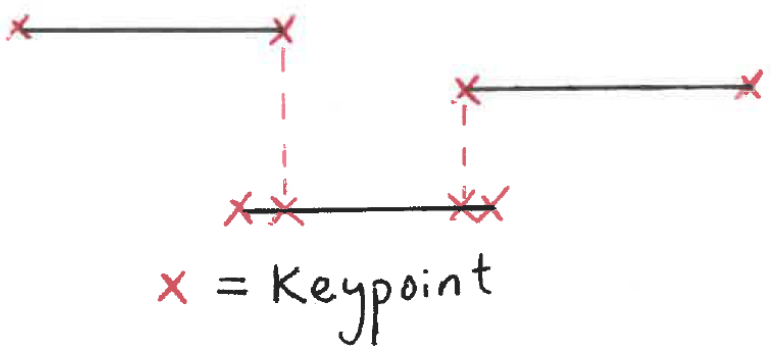


Figure 18: Diagram of Keypoint function

Essentially, keypoints are the positions on a platform that the AI can move between using only 1 action – there is nothing between any 2 keypoints that would cause the AI agent to change its behaviour. A point must meet 1 of 2 conditions in order to be considered a keypoint: it can either be the end of a platform (i.e., the left-most or right-most side) or it can be the point where another platform is directly above it. Unity handles the platform ends in the precomputation simply by setting empty child transforms of both edges. It then sends off RayCasts[3] pointing downwards that create new empty objects if they hit another platform. Those new nodes are the product of the second condition and can be referred to as Contact Points.

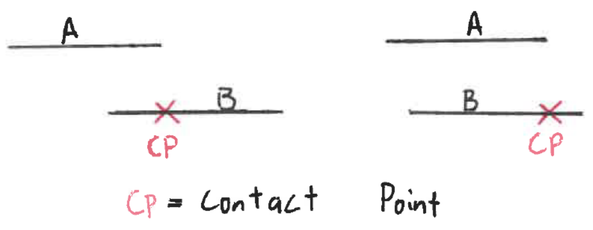


Figure 19: Contact Points

One way could be to represent the graph as:

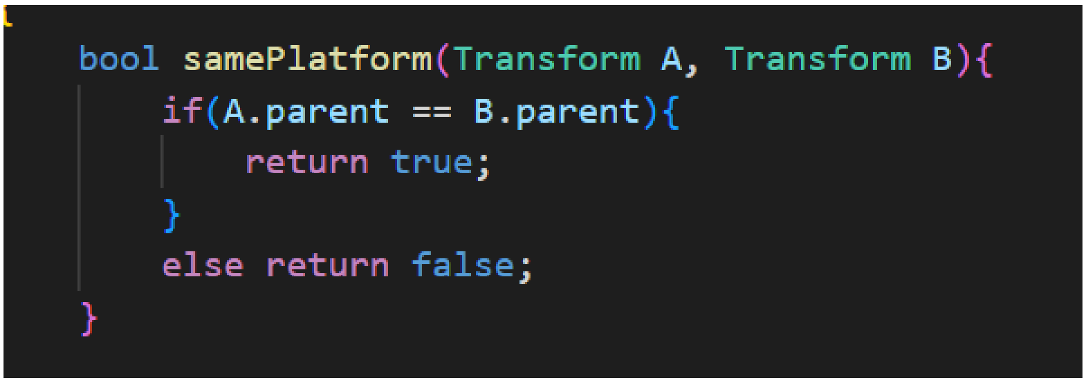
$$Graph = Dictionary<Transform,\ Vector2>$$

Figure 20: Keypoint graph

where the Transform is a node and the Vector2 is that node’s position. Once all the nodes are stored in this dictionary (or key-value map), the next step is to determine whether there is an edge between each node pair, leaving a base calculation complexity of O(|N|^2). This step could be inaccurate if not handled properly. The system cannot, for example, simply compare each node’s position values and conclude that if 2 nodes are near enough to one another, they are neighbours. Instead, the graph can be represented as an adjacency list where nodes are considered neighbours if they meet any 1 of 3 conditions:

1. <u>The samePlatform Relation</u>. Any 2 nodes which are on the same platform will always be neighbours unless there’s an obstacle in the way. Since each node is set to be a child object of a platform, checking this

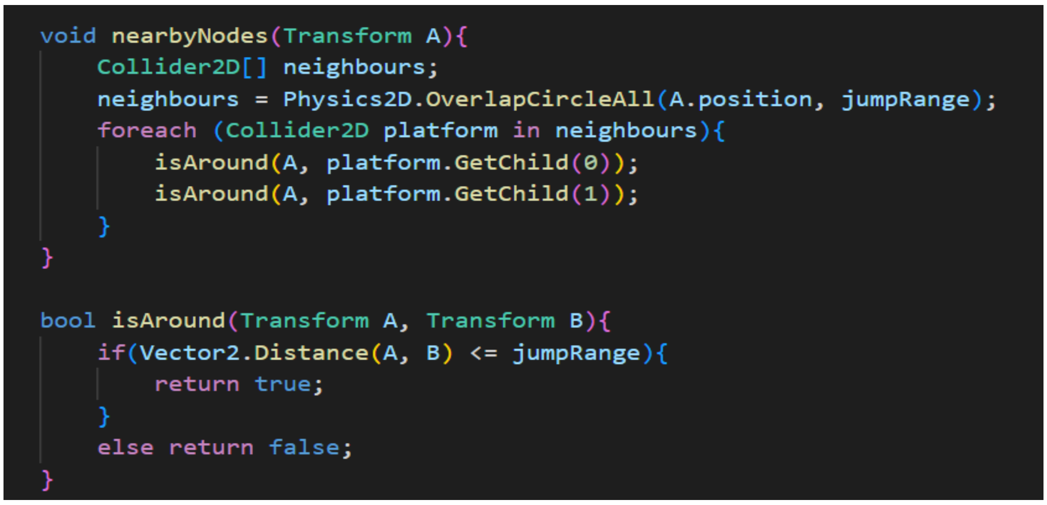
relation will be easy to do in Unity:



```
bool samePlatform(Transform A, Transform B){
    if(A.parent == B.parent){
        return true;
    }
    else return false;
}
```

Figure 21: samePlatform relation

2. <u>The isAround Relation</u>: This is a simple OverlapCircleAll function that will be applied to each node. What this function does is return all colliders within a certain range (jumpRange) of a certain position (node), thereby showing which platforms an AI agent can reach in 1 jump, excluding its ability to indefinitely fall downwards. This relation is necessary for Contact Points because it gives them the ability to determine agent reachability from the middle of the platform (you don't have to be at a platform edge to jump to another one). Using the OverlapCircleAll function alone isn't enough to determine nearby nodes because it returns an array of colliders, which nodes don't have because they are Transforms. This relation would therefore need to determine all platforms within that area and calculate the distance between the given node and each platform's edges.



```
void nearbyNodes(Transform A){
    Collider2D[] neighbours;
    neighbours = Physics2D.OverlapCircleAll(A.position, jumpRange);
    foreach (Collider2D platform in neighbours){
        isAround(A, platform.GetChild(0));
        isAround(A, platform.GetChild(1));
    }
}

bool isAround(Transform A, Transform B){
    if(Vector2.Distance(A, B) <= jumpRange){
        return true;
    }
    else return false;
}
```

Figure 22: isAround relation

An important detail to note is that this relation is only computed between a given node and platform edges. This is the case because you cannot jump upwards to a Contact Point, therefore it would not make sense to compute whether one isAround another.

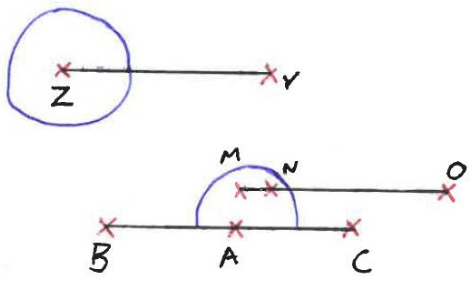


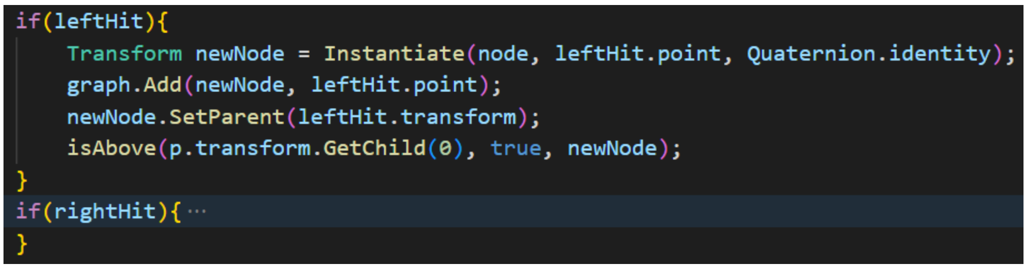
Figure 23: Overlap function

Figure 23 shows a scenario with 3 platforms, where the red crosses represent nodes, and the blue semi-circle represents the jumpRange of node A. In Unity, the RayCast and Overlap functions do not go past any colliders (except with the use of custom settings), that is why there is a semi-circle for node A and a full circle for node Z. Under normal circumstances, calculating nearbyNodes from node A would return nodes B and C, so I added the following condition to the top of the isAround function:

$$if\ (\ samePlatform(\ A,\ B)\ )$$
$$return\ false;$$

After the optimisation, there will are 2 nodes within A’s jumpRange, but from the definition of a Keypoint, we know that an agent will not be able to go from A->N without first traversing M. That is why there is no need to check for an isAround relation between Contact Points. This step is only necessary due to the modelling choices made for the game map. One strength of this Keypoint System is that this relation is easily adaptable to the use of soft/jump through platforms. In which case, the relation wouldn’t need to compute an isAround relation between node and node, it can simply do so between node and platform because an agent would be able to get to the platform from any direction.

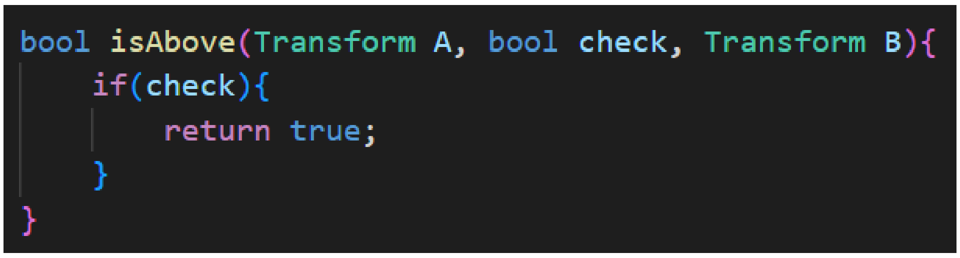
3. The isAbove Relation: This calculation is similar to using the RayCasts to determine Contact Points. The same way isAround can only return nodes on the ends of platforms, only those nodes can have isAbove relations. Contact Points can automatically set their isAbove relation once they’re created by modifying their definition code as seen in Figure 24.



```
if(leftHit){
    Transform newNode = Instantiate(node, leftHit.point, Quaternion.identity);
    graph.Add(newNode, leftHit.point);
    newNode.SetParent(leftHit.transform);
    isAbove(p.transform.GetChild(0), true, newNode);
}
if(rightHit){···
}
```

Figure 24: New Contact Point Definition

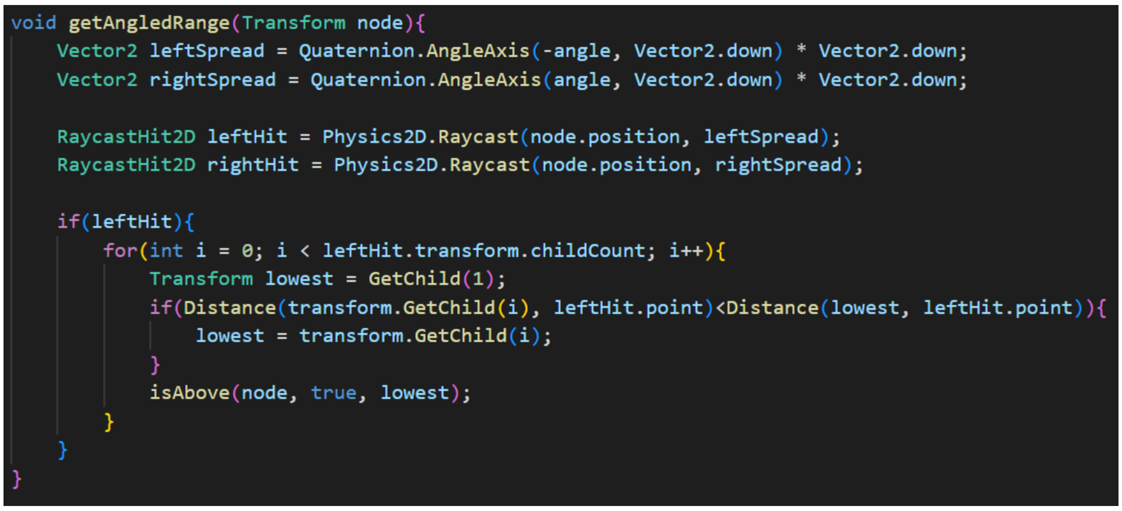
Then, you create the first condition of the isAbove relation as follows.



```
bool isAbove(Transform A, bool check, Transform B){
    if(check){
        return true;
    }
}
```

Figure 25: isAbove function

After creating that first condition, the relation is completed by using 2 additional RayCasts on each platform end node. This step will create the trapezoid reachability area beneath the platform that was described in Section 2.1.3. The ability to move on the x axis can be approximated as an angle offset from the y axis – the further below an AI is from a platform, the higher its possible displacement in the x axis. With that angle, I created a code for the 2 RayCasts of a node to state that it is isAbove the nearest node to the RayCast's hit point.



```
void getAngledRange(Transform node){
    Vector2 leftSpread = Quaternion.AngleAxis(-angle, Vector2.down) * Vector2.down;
    Vector2 rightSpread = Quaternion.AngleAxis(angle, Vector2.down) * Vector2.down;

    RaycastHit2D leftHit = Physics2D.Raycast(node.position, leftSpread);
    RaycastHit2D rightHit = Physics2D.Raycast(node.position, rightSpread);

    if(leftHit){
        for(int i = 0; i < leftHit.transform.childCount; i++){
            Transform lowest = GetChild(1);
            if(Distance(transform.GetChild(i), leftHit.point)<Distance(lowest, leftHit.point)){
                lowest = transform.GetChild(i);
            }
            isAbove(node, true, lowest);
        }
    }
}
```

Figure 26: Second isAbove condition

3.2.2: The Single Platform System (SPS)

This system abstracts the map 1 step further by simply considering a platform as a node. Each platform still has 2 child objects but instead of them being located at the platform’s edges, they represent the top left and bottom right corners of the AI’s reachability range for that particular platform. Figure 27 illustrates this with the crosses representing child nodes and the shaded blue area representing the AI agent’s movement range from that platform.

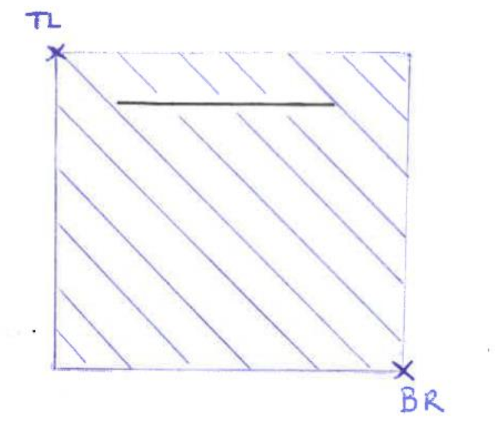
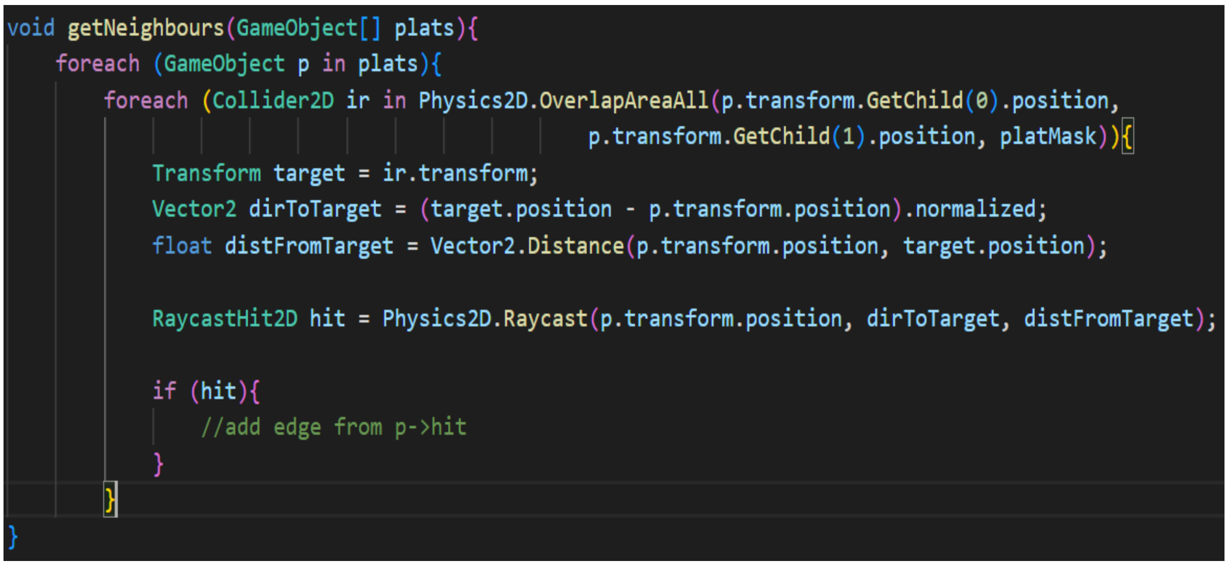


Figure 27: Platform System reachability

Neighbouring platforms are then defined in 2 steps:

1. Does a platform exist in this area?
2. Are there no platforms between it and the platform related to this area?

Figure 28 shows how Unity handles these steps in C# using an OverlapAreaAll where the area is defined by the child objects of a platform.



```
void getNeighbours(GameObject[] plats){
    foreach (GameObject p in plats){
        foreach (Collider2D ir in Physics2D.OverlapAreaAll(p.transform.GetChild(0).position,
                                                          p.transform.GetChild(1).position, platMask)){
            Transform target = ir.transform;
            Vector2 dirToTarget = (target.position - p.transform.position).normalized;
            float distFromTarget = Vector2.Distance(p.transform.position, target.position);

            RaycastHit2D hit = Physics2D.Raycast(p.transform.position, dirToTarget, distFromTarget);

            if (hit){
                //add edge from p->hit
            }
        }
    }
}
```

Figure 28: getNeighbours function

The second step makes the check similar to a RayCast and is necessary for the context of platformers. It has logic similar to the nodes in the Keypoints System, where certain nodes might be within a range but not directly traversable. Consider the scenario in Figure 29, for instance:

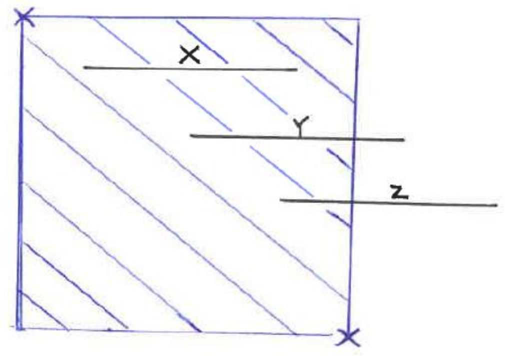


Figure 29: Indirect neighbours in Platform System

There exists a path from node X to node Z but the 2 of them are not adjacent because it is impossible to move between them without coming in contact with node Y. This range covers both the isAbove and isAround relations, and the system nullifies the notion of samePlatform altogether.

I was initially worried that this system would make inaccurate Edge comparisons between each platform by only considering the centre of a platform to be within range. That wasn't the case because the OverlapAreaAll function returns an array of Colliders, so it doesn't consider the centre of the platform at all. The downside to this calculation is that when a node discovers an edge, its cost would be the cost between the 2 platform positions and not the 2 nearest points between both platforms.

3.2.3: Solution Comparisons

Both systems' strengths can be summed up as follows: the Keypoint System is more accurate and versatile; the Single Platform System is more efficient. Using keypoints creates a much more accurate picture for reachability. While there is a chance that a platform can exist between 1 of the 3 RayCasts used in the isAbove relation, that chance is slim and can be reduced by using a smaller offset angle for the downwards casts. The reachability described in Figure 23 is nearly identical to the realistic reachability described in Figure 9. The Single Platform System, however, has a very limited reachability range and doesn't really account for freefall movements – The Keypoint System has the accuracy advantage and therefore yields better results for NFR3.

It is also more robust in its traversal calculations because it considers the cost to traverse a platform. This advantage maximises the results of FR3 and NFR3. Its cost calculations are also more accurate than the Single Platform System because of how the nodes are a single point in space and not a range. Having more nodes on a platform also means that the system is versatile enough to incorporate different node types like conveyer belts or sticky/slippery surfaces, and it can allow for various platform lengths. The SPS is extremely restrictive in this sense because all platforms must be standard and have the exact same length in order for the system to work. This system doesn't take surface traversal into account at all.

The area where the SPS is better than the KPS is its efficiency (NFR5). This system is able to collapse all Keypoint operations into 1 function. Doing this saves so much time for computation because each platform has at least 8 RayCasts (4 for each platform end node, excludes Contact Points) and 3 relations that each node would have to check for. Not only that, but the Unity game environment has much fewer Transforms because each platform doesn't gain/lose child platforms. This system also has a much more straightforward way to consider node positions (used in Objective 7). In a dynamic map environment, platform movements add an extra layer of complication to the KPS. Consider the situation in Figure 30:

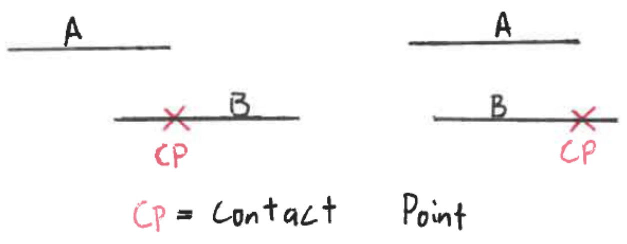


Figure 30: Movement in KPS

Once platform A moves, the equivalent Contact Point on platform B needs to move as well in order to keep this model consistent. This alone adds enough computational complexity to the system but once a platform moves in/out of range to another one, it would have to insert/delete the Contact Point each time. This would be an extreme waste of computation and memory. The Single Platform System doesn't require any such allocation/deallocation.

After taking the strengths of both systems into consideration, I decided that the Single Platform System was the way to go. The versatility that the Keypoints provide goes to waste for this project because I already decided to design the map with only standard platforms (Section 3.1). Without that, all that's left is the comparison between accuracy and performance. That comparison might be equal in certain contexts, but the difference between the computational complexity of both systems is too great to even consider the Keypoint System.

3.3: Creating the Cost System

For the cost system, I also tested 2 different methods and weighed the pros and cons to choose the most suitable one for my project. While this wasn't as intricate as creating the nodes system, it was still important to create a realistic system that emphasizes the gravity factor in the calculation. I came up with 2 principles that I wanted my cost system to adhere to:

- The higher the x axis difference of 2 nodes, the higher the cost.
- Going up on the y axis should cost more than going down.

At first, I wanted to use Newtonian Physics to get life-like accuracy for the gravity, with an exact downwards acceleration of 9.81 m/s^2. This calculation would have been useful for finding the AI's jump angle to strengthen the approximation of Figure 9, but that level of accuracy isn't worth its use in the cost calculation. With the Single Platform node system (Section 3.2.2), there is a limit to the range that adds edges and gravity doesn't accelerate the speed of an AI agent too much within that range. Instead, I modelled both systems directly from the distance between the 2 nodes.

Both systems use a linear relation for the difference in the x axis and cost.

$$\left| first_x - second_x \right|$$

Figure 31: Absolute difference on x axis

Where they differ is how the difference in the y axis is modelled.

$$\left| first_x - second_x \right| \times \frac{second_y}{first_y}$$

Figure 32: Cost System 1

$$\left| first_x - second_x \right| + \left(second_y - first_y \right)$$

Figure 33: Cost System 2

I hypothesised that both Systems would maintain the 2 principles discussed at the start of this section and that the second one would be more efficient because addition and subtraction are computationally much cheaper than multiplication and division. I tested both of these hypotheses (Section 4.1) and ultimately decided to use Cost System 2.

3.4: Choosing an Appropriate Graph Data Structure

3.4.1: Review of QuikGraph

QuikGraph [14] is a .NET library that "provides generic directed/undirected graph data structures and algorithms" [14]. Initially, I planned to use QuikGraph and its existing functionalities to abstract my system. This was a sensible choice because the library comes with various data structures and search algorithms which would have allowed me to test/compare them all almost simultaneously. It would also have meant I saved myself the time and trouble of making my own graph structure from scratch, instead I would've just needed to find a way to incorporate my system into the library. In spite of all its positives, I opted not to use QuikGraph in order to mitigate the main project risk: not finishing in time. While QuikGraph would've meant less time on making the structures and algorithms, that time would have still gone towards becoming comfortable with the library and figuring out how to model my graph in .NET. For a 10-week dissertation project, it didn't seem like a wise decision to take an indefinite amount of time getting used to a new technology and adapting it – however good that technology may be. Nonetheless, I briefly looked through it and doing so gave me more reason to stick to my decision. As I had imagined, the library seemed a bit too complex to familiarise myself with in a short amount of time. I also noticed that it had classes for both nodes and edges and at first sight, it didn't seem like the edges had costs associated to them. With further research, I understood that the library used an external map for the costs which the traversal algorithms used over each associated edge. I felt that all this structure was unnecessary, and that QuikGraph is slightly too robust for my project. The cost map, node class and edge class were ultimately uses of memory I concluded to be a waste.

3.4.2: My Graph Solution

For my graph, I went with the simple solution of using a dictionary. The dictionary is a map of string to string to float, which represents a start node, an end node and the cost of their edge, as shown in Figure 34.

$$Graph = Dictionary[start -> Dictionary[final -> cost]\ ]$$

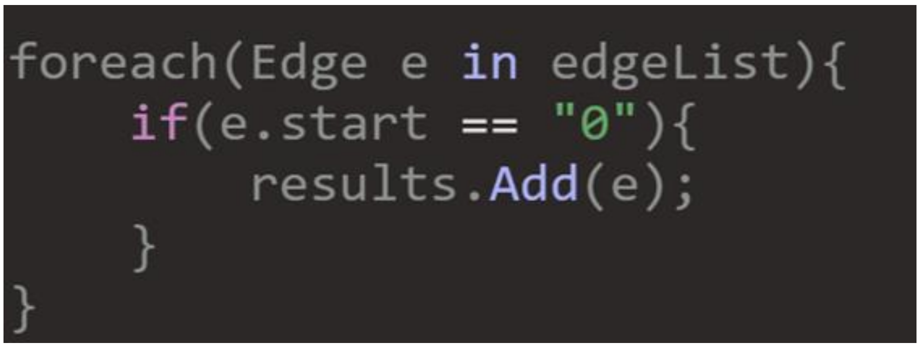
Figure 34: My Graph Abstraction

This implementation extends Adjacency Lists over Edges by representing the set of edges as another Dictionary, mapping each target node to the cost of the source->target edge. It is more efficient than all previous solutions because the scanning cost is significantly reduced as it can access the target vertex with O(1), but it doesn't waste memory by storing non-existent edges. Figure 35 shows the performance of this solution in comparison to the first 2 discussed in Section 2.2.1.

	Adjacency List	Adjacency Matrix	Adjacency Dictionary
Space Complexity	$O(\|V\| + \|E\|)$	$O(\|V\|^2)$	$O(\|V\| + \|E\|)$
Time Complexity	$O(\|V\|)$	$O(1)$	$O(1)$
Access Outgoing Edges	$O(\|V\|)$	$O(1)$	$O(1)$
List all Edges	$O(\|V\| + \|E\|)$	$O(\|V\|^2)$	$O(\|V\| + \|E\|)$
Add Vertex	$O(1)$	$O(V^2)$	$O(1)$
Remove Vertex	$O(\|V\| + \|E\|)$	$O(\|V\|^2)$	$O(\|E\|)$
Add Edge	$O(E)$	$O(1)$	$O(1)$
Remove Edge	$O(\|V\| + \|E\|)$	$O(1)$	$O(1)$

Figure 35: New graph solution comparisons

Using this abstraction means that updating cost values can be done much faster. It doesn't take as long as a list would to find the Edge to update values for, and it doesn't require as much complexity as a matrix would to add a new vertex (adding an edge would be done when a platform appears, therefore a new vertex). Another important advantage is that you can easily find all the connections from a given platform. If I used QuikGraph, for example, which has edge lists, finding all the connections from node "0" would have been a process Figure 36:



```
foreach(Edge e in edgeList){
    if(e.start == "0"){
        results.Add(e);
    }
}
```

Figure 36: Finding connections with Edge List

Whereas the computation in an Adjacency Dictionary is simply:

$$results = Graph\left["\ 0\ " \right].Keys$$

Figure 37: Finding connections in Adjacency Dictionary

This solution is efficient enough for a platformer abstraction while still maintaining all the functions it needs to use. It still has notion of nodes, edges and costs, having everything organised by their start node which makes accessing them more efficient.

3.5: Creating a Graph Traversal Algorithm

I did all the other steps in Section 3 in the order that they were listed in in the objectives (Section 1.4), but I did this before working on the navigation mechanics. As discussed in Section 1.4, designing the navigation is the only objective which can be done independently, and so I decided to first work with the new graph solution.

The first solutions algorithms I thought of were Dijkstra's and A*. Despite these being some of the most reliable and recognisable pathfinding algorithms, some of their functionality would have gone to waste, thus not adhering to NFR2. For one thing, they include hooks in their pathfinding, but I modelled my graph to have no self-edges and that calculation will therefore be unnecessary. The second inefficiency of those searches is that the shortest path between 2 nodes might not be a direct edge between them. Using my cost system and the logic of platformer games, I wanted to set a heuristic for the algorithm that states "if there are multiple paths between 2 platforms, the path with more nodes always has a higher cost than the path with fewer nodes".

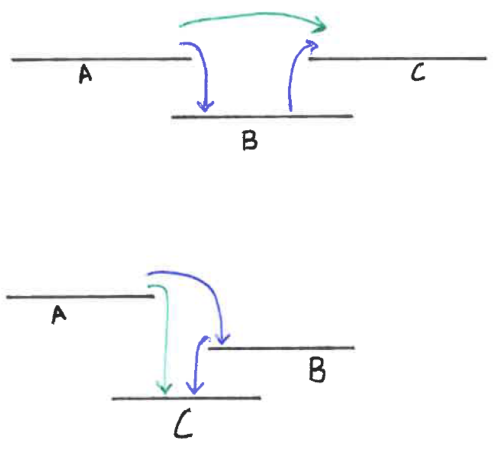


Figure 38: Shorter Path Heuristic

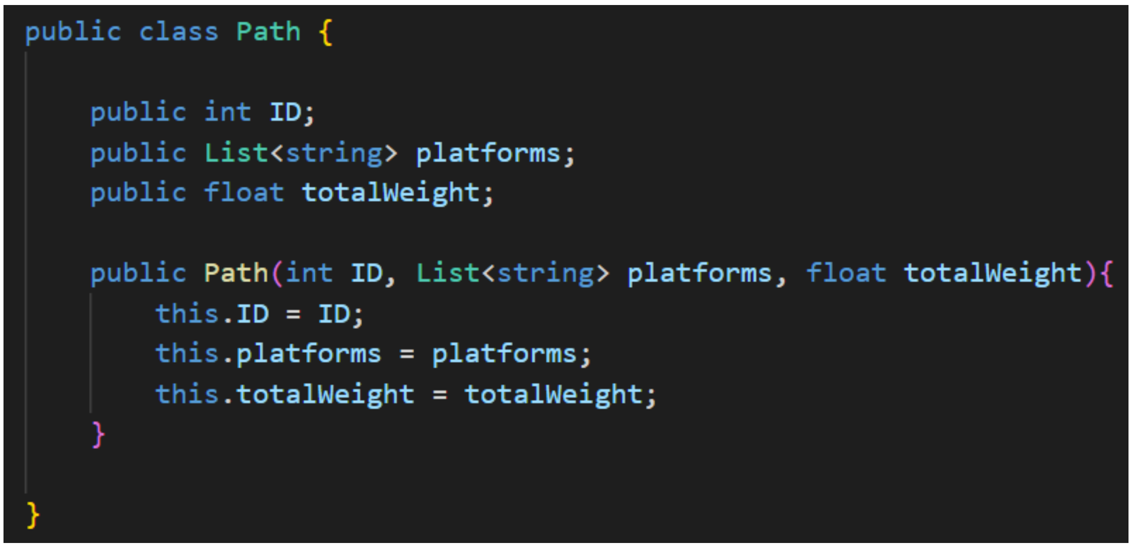
With that heuristic, the green paths in Figure 38 should be preferable to the blue ones. While costs are distance based, they should also be related to the time it takes to complete a path. The reality of a platformer game is that a direct jump to another platform takes less time than an indirect series of jumps. If this heuristic holds true, the algorithm will no longer need a notion of cost and a simple BFS or DFS search would suffice. I tested this with multiple scenes in my game environment, results can be found in Section 4.2.

I designed the algorithm with that principle in mind and opted to use a Depth First Search, but I still kept the weights in the calculations in case the heuristic wasn't the case.

I defined this algorithm within the context of my system as a recursive function that consists of 2 simple steps. In the context of finding a path between node "start" and node "final":

1. Check if start = final. Return start if so.
2. If not, run the function on all the test on all unvisited nodes connected to start.

As it is right now, all the function does is return the penultimate node in the path from start to finish. The AI would need to keep track of all the nodes on each path and how much it costs to get from 1 node to another in order to determine a complete path with a list of nodes and a total cost. In order to note those details, the system needs a "Path" class.

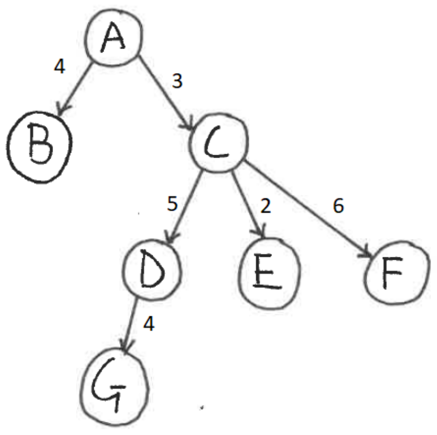


```
public class Path {

    public int ID;
    public List<string> platforms;
    public float totalWeight;

    public Path(int ID, List<string> platforms, float totalWeight){
        this.ID = ID;
        this.platforms = platforms;
        this.totalWeight = totalWeight;
    }

}
```

Figure 39: Path class

With this class in place, I was able to make a lot of modifications to the search algorithm so that it works accurately.

In order to store information about list of platforms and total weight, the search needs to be able to send all the information from a Path object to all the child nodes where that object would branch off to. Figure 40 illustrates what this would look like in practice.



$$\begin{tikzcd}
 & A \arrow[dl, "4"'] \arrow[dr, "3"] & & \\
B & & C \arrow[dl, "5"'] \arrow[d, "2"] \arrow[dr, "6"] & \\
 & D \arrow[d, "4"] & E & F \\
 & G & &
\end{tikzcd}$$

P1 = A->B with cost 4

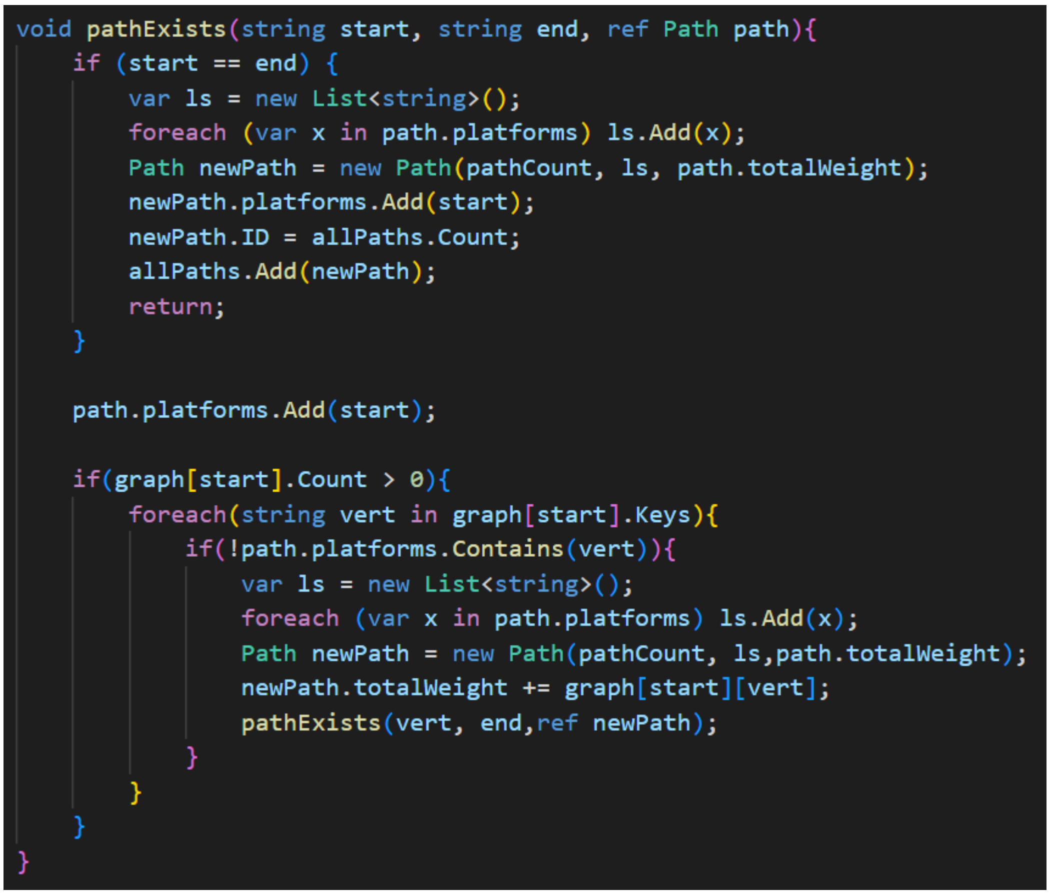
P2 = A->C->D->G with cost 12

P3 = A->C->E with cost 5

P4 = A->C->F with cost 9

Figure 40: Path continuation

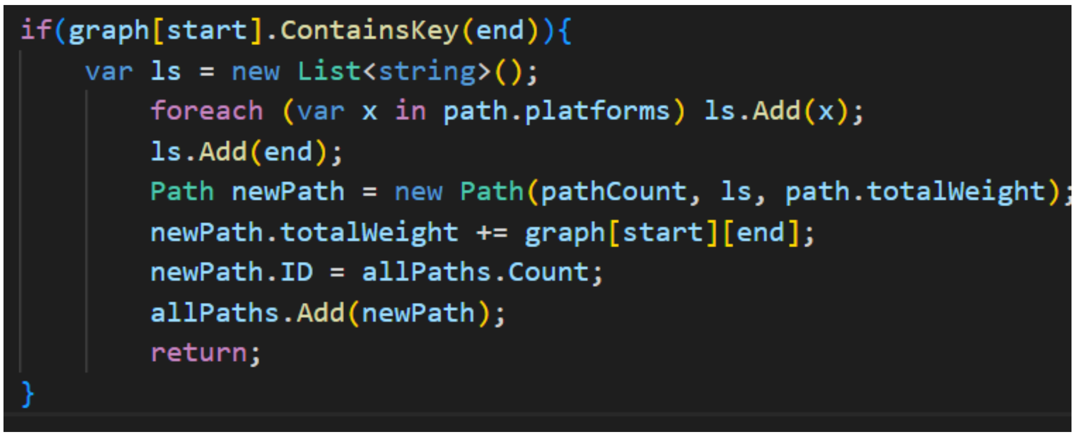
With this clear picture of how the search should behave and yield results, I filled up the recursive function with the following information:



```
void pathExists(string start, string end, ref Path path){
    if (start == end) {
        var ls = new List<string>();
        foreach (var x in path.platforms) ls.Add(x);
        Path newPath = new Path(pathCount, ls, path.totalWeight);
        newPath.platforms.Add(start);
        newPath.ID = allPaths.Count;
        allPaths.Add(newPath);
        return;
    }

    path.platforms.Add(start);

    if(graph[start].Count > 0){
        foreach(string vert in graph[start].Keys){
            if(!path.platforms.Contains(vert)){
                var ls = new List<string>();
                foreach (var x in path.platforms) ls.Add(x);
                Path newPath = new Path(pathCount, ls,path.totalWeight);
                newPath.totalWeight += graph[start][vert];
                pathExists(vert, end,ref newPath);
            }
        }
    }
}
```

Figure 41: pathExists function

What this function does is return a list of paths from start to finish. The idea is to take a list of all the paths the search finds and then compare their weights to choose the shortest 1. However, there are 2 possible optimisations I found that would make this function run a lot faster. The first one is to implement the heuristic I defined at the start of this section. Doing so would mean that the function stops running as soon as it finds a path, adhering to NFR2. This can be done in the code by adding the following check:



```
if(graph[start].ContainsKey(end)){
    var ls = new List<string>();
        foreach (var x in path.platforms) ls.Add(x);
        ls.Add(end);
        Path newPath = new Path(pathCount, ls, path.totalWeight);
        newPath.totalWeight += graph[start][end];
        newPath.ID = allPaths.Count;
        allPaths.Add(newPath);
        return;
}
```

Figure 42: DFS optimisation

Since the graph uses dictionaries, it is easy to check whether a node is connected to the end node, and therefore doesn't require any further iterations. Adding this reduces the cardinality of the allPaths list because once 1 path is found, it doesn't check for paths with more platforms than in the current path. allPaths, therefore, will only contain 1 path. I left the function as it was because adding a return type would have complicated things in the recursion.

The final optimisation is to remove the idea of costs altogether. Because it is a depth first search and returns a path as soon as it finds one, costs might seem insignificant. While this is somewhat applicable in the current context, I chose not to include that optimisation for the sake of making the system more robust and more versatile. The model used to determine the DFS heuristic is only accurate if all platforms are of the same length. If the platforms vary in length, the time to traverse A->C might not necessarily be lower than the time to traverse A->B->C. allPaths can, however, return more than 1 path so my solution orders them all by lowest to highest cost and returns the first item.

3.6: Creating AI Navigation Mechanism

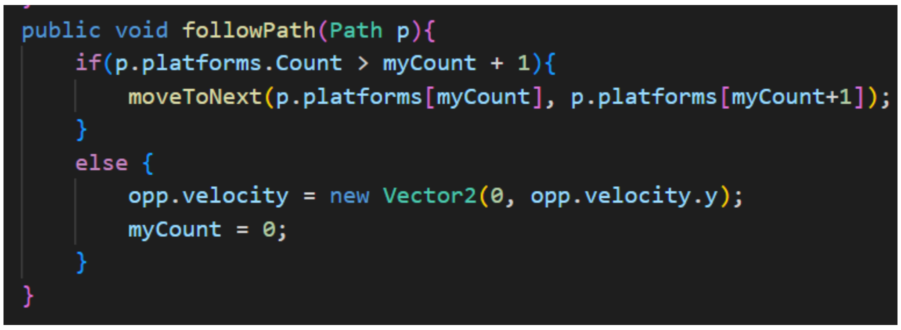
Now that there is a graph and a function to find the shortest path from 1 node to another, all that's left is to set rules for the enemy agent to traverse the given path (if there is one). Following a path consists of 2 simple steps:

1. Break the list of platforms down into pairs of current to next platform.
2. Move from current until agent reaches the destination.

I initially planned to attach an action to each edge in order to reduce the amount of computation needed for this step. This approach would have worked well if my nodes used the Keypoint System but using the Platform System means that, more often than not, an edge would need more than 1 action. Therefore, I hardcoded specific instructions for step 2 of the navigation.

3.6.1: Following Path Platforms in Order

The pathExists function simply find the shortest path from point A to B. The AI agent needs another function for it to follow that path 1 platform at a time. The first step of this function can be done with a very basic set of instructions as seen in Figure 43.

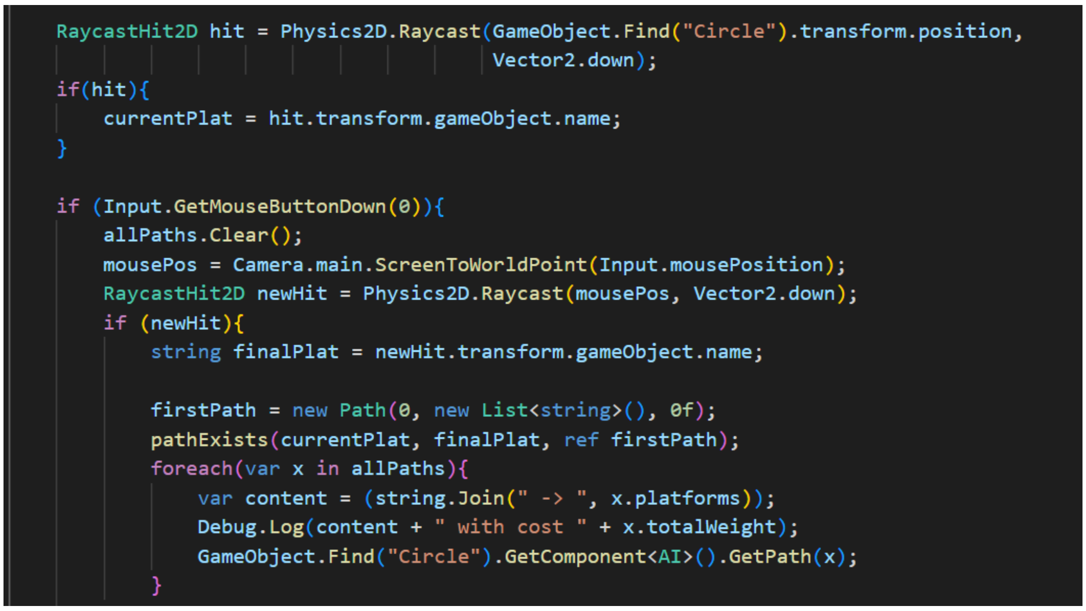


```
public void followPath(Path p){
    if(p.platforms.Count > myCount + 1){
        moveToNext(p.platforms[myCount], p.platforms[myCount+1]);
    }
    else {
        opp.velocity = new Vector2(0, opp.velocity.y);
        myCount = 0;
    }
}
```

Figure 43: followPath function

In followPath, myCount is just an integer variable that increments each time the AI agent moves to the next platform on the Path's list. Once it equates to the number of platforms, the AI stops moving and resets myCount to 0. Most of the heavy lifting is done before and after this function, namely in retrieving the path to follow and running the moveToNext function.

I designed the system such that the path to retrieve always starts from the enemy agent's current platform and ends at a platform the user clicks on. In practice, the end platform would be some target; it could be the player in a game or maybe some powerup. I left it as a click in this project so that I could easily do navigation tests. As of right now, calculating what path to follow is done with the code snippet in Figure 41.



```
RaycastHit2D hit = Physics2D.Raycast(GameObject.Find("Circle").transform.position,
                                     Vector2.down);
if(hit){
    currentPlat = hit.transform.gameObject.name;
}

if (Input.GetMouseButtonDown(0)){
    allPaths.Clear();
    mousePos = Camera.main.ScreenToWorldPoint(Input.mousePosition);
    RaycastHit2D newHit = Physics2D.Raycast(mousePos, Vector2.down);
    if (newHit){
        string finalPlat = newHit.transform.gameObject.name;

        firstPath = new Path(0, new List<string>(), 0f);
        pathExists(currentPlat, finalPlat, ref firstPath);
        foreach(var x in allPaths){
            var content = (string.Join(" -> ", x.platforms));
            Debug.Log(content + " with cost " + x.totalWeight);
            GameObject.Find("Circle").GetComponent<AI>().GetPath(x);
        }
```

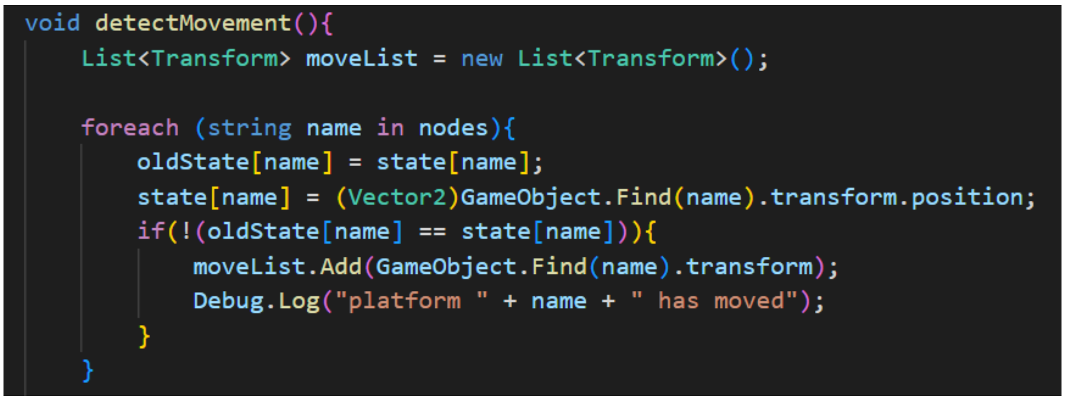
Figure 44: Getting a path to follow

This current system is adequate for following a path in a static game environment, but paths might change once the environment does. In order to detect environment changes, I used a new data structure: State. The State of the map can be represented as a Dictionary of string to Vector2, where the string is a platform name and Vector2 is its position.

$$State = Dictionary[string -> Vector2]$$

Figure 45: State

With a notion of States, the system can detect when a platform moves by simply creating another structure for oldState. Then, it sets oldState to state and update State at regular intervals. Whenever State is updated, any platform whose value differs between State and oldState is said to have moved, and the path needs to be recalculated. Using states helps us accomplish Objective 7 efficiently because of the O(1) complexity of finding a value for a key in Dictionaries.



```
void detectMovement(){
    List<Transform> moveList = new List<Transform>();

    foreach (string name in nodes){
        oldState[name] = state[name];
        state[name] = (Vector2)GameObject.Find(name).transform.position;
        if(!(oldState[name] == state[name])){
            moveList.Add(GameObject.Find(name).transform);
            Debug.Log("platform " + name + " has moved");
        }
    }
```

Figure 46: detectMovement code

Each time a platform movement is detected, followPath runs again from the current AI platform to the given final node. A weakness to doing this computation is that the previous path calculation is wasted each time a platform moves. The system has to constantly recalculate paths. I managed to reduce the calculation's effect on the efficiency by using regular intervals instead of checking this directly in Update. My system does this check every 1.5 seconds, meaning it's done 1/36 as often as it would've been done in Update.

3.6.2: Moving to the Next Platform

Moving to a platform's neighbour can be grouped into 1 of 3 actions:

- Jump from the end of a platform
- Jump from some point in the middle of a platform
- Walk off the platform (no jumping necessary).

Doing any one of those actions isn't too complicated, the intricacy of the moveToNext function comes from determining which of those 3 it needs to do.

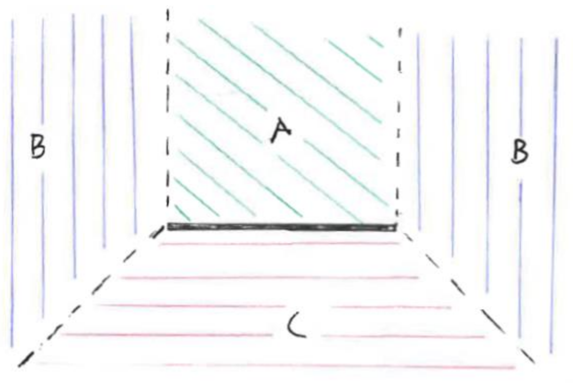
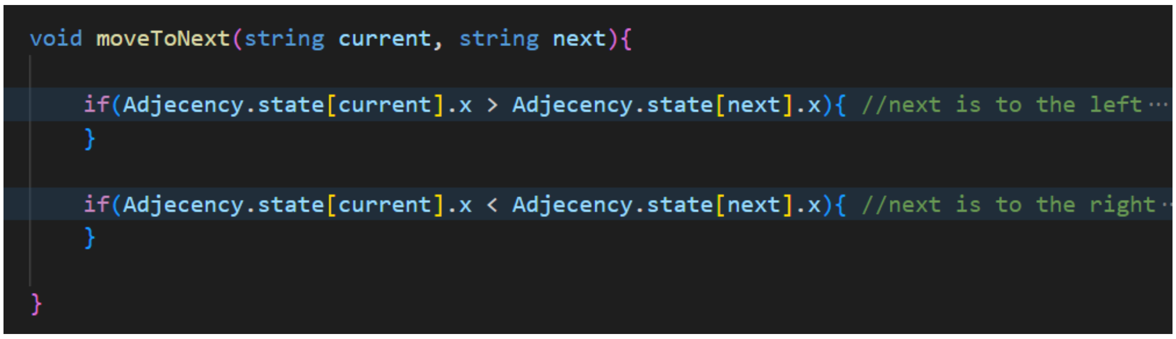


Figure 47: moveToNext check

The moveToNext function can determine which of the actions the AI needs to do to get to another platform depending on what area it resides in based on Figure 47. If a platform is in area A, it jumps from the middle. If in B or C, it jumps or rolls (respectively) off the edge. Platforms can, however, exist in more than 1 area, therefore we would need to check for a position of interest rather than the platform object as a whole. To calculate this in C#, I used both platforms' Colliders to get their edges and set them as the positions of interest. The first thing that the function needs to check is if the next platform is to the left or the right of the current platform. This step can be done using the notion of states rather than real time values, so it computes this check 1/36 as often as Update as well. Doing this also strengthens the functionality of the State which is useful for further additions (3.7).

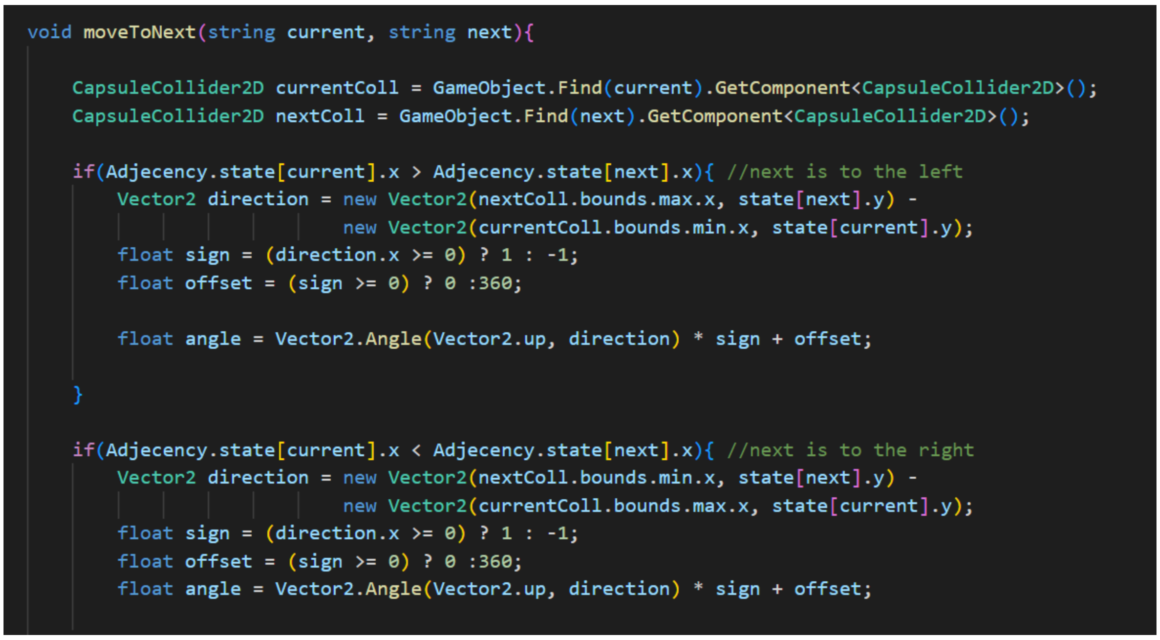


```
void moveToNext(string current, string next){

    if(Adjecency.state[current].x > Adjecency.state[next].x){ //next is to the left···
    }

    if(Adjecency.state[current].x < Adjecency.state[next].x){ //next is to the right·
    }

}
```

Figure 48: First step of moveToNext

Note: "Adjecency" is the file that contains the pathfinding information, and this navigation code is attached to the AI. I also spelled it wrongly but realised too late.

This check is used to determine what edges the function needs to find an angle between. If the next platform is to the left of the current one, we calculate the angle between the left side of current and right side of next and then vice versa. Figure 49 shows this step in C#.



```
void moveToNext(string current, string next){

    CapsuleCollider2D currentColl = GameObject.Find(current).GetComponent<CapsuleCollider2D>();
    CapsuleCollider2D nextColl = GameObject.Find(next).GetComponent<CapsuleCollider2D>();

    if(Adjecency.state[current].x > Adjecency.state[next].x){ //next is to the left
        Vector2 direction = new Vector2(nextColl.bounds.max.x, state[next].y) -
                            new Vector2(currentColl.bounds.min.x, state[current].y);
        float sign = (direction.x >= 0) ? 1 : -1;
        float offset = (sign >= 0) ? 0 :360;

        float angle = Vector2.Angle(Vector2.up, direction) * sign + offset;

    }

    if(Adjecency.state[current].x < Adjecency.state[next].x){ //next is to the right
        Vector2 direction = new Vector2(nextColl.bounds.min.x, state[next].y) -
                            new Vector2(currentColl.bounds.max.x, state[current].y);
        float sign = (direction.x >= 0) ? 1 : -1;
        float offset = (sign >= 0) ? 0 :360;
        float angle = Vector2.Angle(Vector2.up, direction) * sign + offset;
```

Figure 49: Calculate angle for moveToNext

I had to incorporate in idea of ‘sign’ and ‘offset’ in order to get the angles in 360 degrees because Unity sets angles as a range between 180 and -180 which would have been very difficult to work with. With that problem out of the way, I was able to set 3 checks to see what area the next platform is in, and consequently what action the AI agent should perform to get to it. The first area to check is area C. This check can be done as seen in Figure 50, where σ represents the ‘slant’ of the line between B and C.

$$In\ C\ if$$

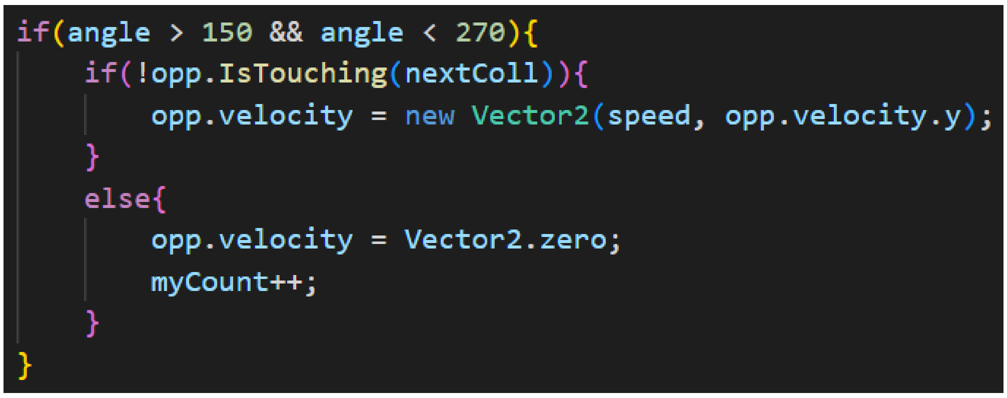
$$90 < angle < 180 + \sigma \ \ (\,going\ right)$$

$$or$$

$$180 - \sigma < angle < 270\,(\,going\ left)$$

Figure 50: Check for section C

With the knowledge of when the next platform is in this section, the AI needs to know what actions to perform to get there. I set the instruction for this situation to omit a jump from the platform because I think rolling off looks more natural, aiming to meet NFR4. It also salved navigation time and reduces the possibility of the AI ending up on the wrong platform. This action can be described in Figure 51, where the next platform is to the right. The only difference for if the next platform is to the left is that opp.velocity would use ‘speed’ instead of ‘-speed’.

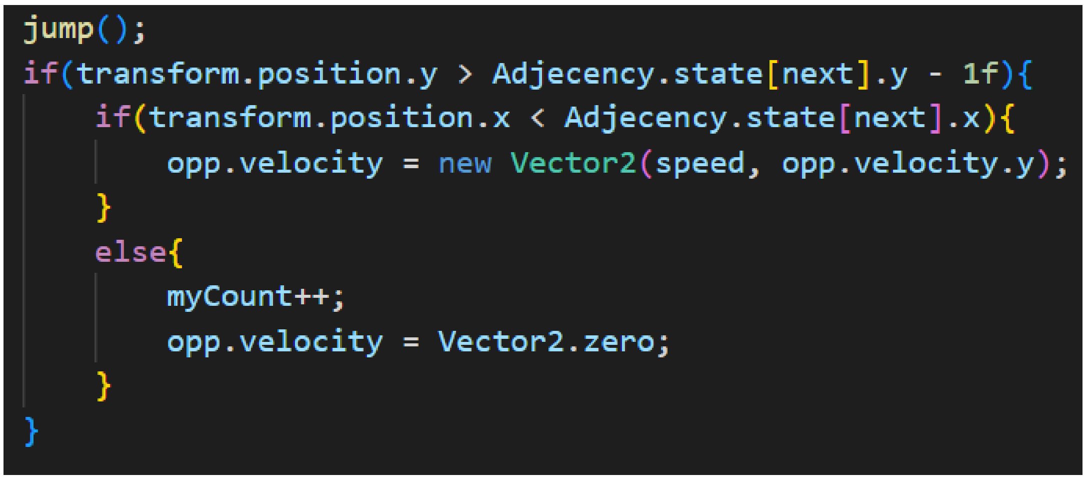


```
if(angle > 150 && angle < 270){
    if(!opp.IsTouching(nextColl)){
        opp.velocity = new Vector2(speed, opp.velocity.y);
    }
    else{
        opp.velocity = Vector2.zero;
        myCount++;
    }
}
```

Figure 51: Basic movement code

This code snippet adds a velocity to the AI agent in the direction of the next platform until it touches that platform, at which point it stops moving. It is important to note that myCount increments once the AI reaches its destination. What this does is tell the followPath function that this first step has been completed and that the AI needs to start moving to the next platform on the list. Using myCount is necessary because movement is something that can only be done in Update, but you can’t simply plug the list directly into Update as it would “navigate” between all platforms at once. With this, the AI only starts moving to another platform once it reaches the one it’s meant to reach and so followPath can go directly into Update.

For platforms in both areas A and B, the AI agent would need to jump and then move. The jump-and-move mechanism is like the basic movement function except the AI doesn’t start moving until it has a y value greater than 1 unit below the next platform.



```
jump();
if(transform.position.y > Adjecency.state[next].y - 1f){
    if(transform.position.x < Adjecency.state[next].x){
        opp.velocity = new Vector2(speed, opp.velocity.y);
    }
    else{
        myCount++;
        opp.velocity = Vector2.zero;
    }
}
```

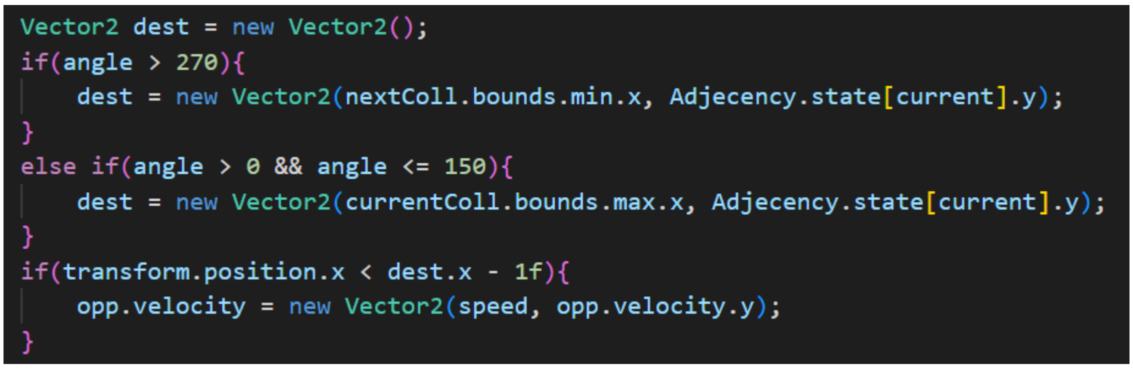
Figure 52: jump-and-move code

The need to wait to be within 1 unit below the next platform is a result of the Single Platform node system. The reachability range in this system is a rectangle, therefore it doesn't have the detail of curved corners from a platform edge that the real reachability range has (as seen in Figure 9). The AI cannot start moving on the x axis as soon as it jumps because it might end up below the next platform if that platform is in area A. On the other hand, the next platform might be on the corner of the current platform's reachability range, and so the AI cannot wait to get to its max height before it starts moving. A solution might have been to make separate jump-and-move mechanisms for both platforms as follows:

```
if next in A:                    if next in B:
jump                             jump & move
move when $AI_y > next_y$
```

Figure 53: Possible jump-and-move solution

This solution provides adequate instructions for both scenarios in order to negate their weaknesses, but the solution in Figure 52 strengthens the entire system. I edited the reachability range of each platform by making them slightly wider and reducing the distance above the platform such that the AI can actually jump above the range. By doing this, the range becomes closer to that in Figure 9 by not only accounting for more freefall movement but also the parabola of a jump. This modification might mean that there are platforms which won't be registered as adjacent even though they are in actuality (within the jump height but out of the reachability query) but that has its benefits as well. It reduces the size of the graph dictionary, therefore making the system more efficient, and it means that the jump-and-move mechanism can be collapsed into 1 operation instead of 2. All the AI needs to do in order to differentiate between the 2 scenarios is determine the spot it needs to jump from.



```
Vector2 dest = new Vector2();
if(angle > 270){
    dest = new Vector2(nextColl.bounds.min.x, Adjecency.state[current].y);
}
else if(angle > 0 && angle <= 150){
    dest = new Vector2(currentColl.bounds.max.x, Adjecency.state[current].y);
}
if(transform.position.x < dest.x - 1f){
    opp.velocity = new Vector2(speed, opp.velocity.y);
}
```

Figure 54: Finding jumpSpot

While it might seem unclear at first sight, all this code is doing is finding the nearest position on the current platform to the next one and moving the AI there. If the next platform is in area B, the nearest position is the current one and if next is in A, the nearest position is just below the edge of B. This is essentially like finding one of the Contact Points described in Section 3.2.1.

The navigation function can be summarized as the following:

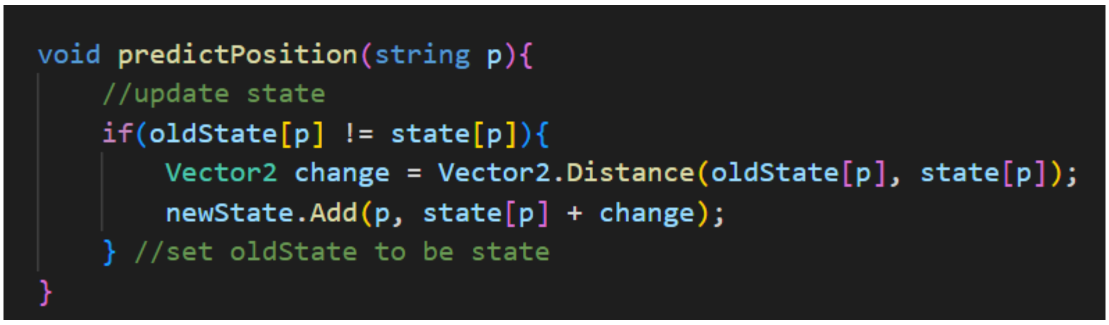
- Determine if the next platform is to the left or right of the current one.
- Find which Section the platform is in.
- If it's in Section C, let the AI roll off the current platform.
- If it's in A or B, move the AI to 1 unit before the jumpSpot.
- Once at jumpSpot, jump directly up and start moving towards next.
- Stop calculation once the AI reaches next platform.

3.7: Exploring Possibilities for Additional Features

While the current system is able to adapt to map changes such as movement or destruction of platforms, it doesn't use those changes to its advantage, as it still only views all the platforms as stationary, just at various points in time. To get a complete sense of the shortest path, the AI should be able to predict whether or not a moving platform can provide . This section covers Objective 8 from Section 1.4 by examining how path prediction can be achieved using the existing tools in the system.

3.7.1: Finding a Platform’s Trajectory

Movement in the map environment is handled by State structures of past and present positions of platforms. The pathfinding system would be able to predict a platform’s trajectory by calculating the change in position from the old to the current State with the following calculation:

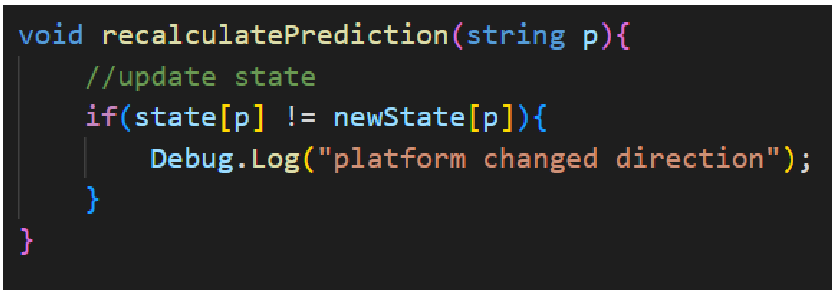


```
void predictPosition(string p){
    //update state
    if(oldState[p] != state[p]){
        Vector2 change = Vector2.Distance(oldState[p], state[p]);
        newState.Add(p, state[p] + change);
    } //set oldState to be state
}
```

Figure 55: Predict Position

Using this new State structure can help predict where the platforms will be the next time it checks. This is why the notion of States and regular check intervals is extremely important for movement detection because it can help with trajectory estimation as well.

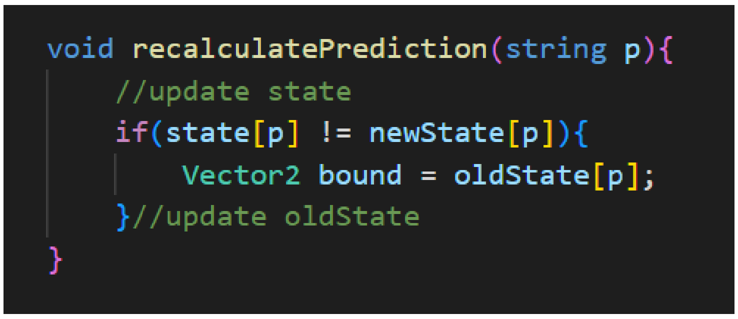
At first glance, it is easy to criticise this estimation in the context of a platformer game because the function cannot say for certain that the platform will be at that future position. While this criticism is true, this estimation is also as accurate as what a human can do in the present moment. If a human plays in an unfamiliar platformer game environment, all he/she can infer is that a moving platform will continue moving in that direction until proven otherwise. From my experience of using platformer games, and my research done in Section 2.1, moving platforms normally have constant speeds and move along a path – either in 1 direction or back and forth. This addition, therefore, is accurate enough for the first principle, and the second can use the following adaptation:



```
void recalculatePrediction(string p){
    //update state
    if(state[p] != newState[p]){
        Debug.Log("platform changed direction");
    }
}
```

Figure 56: Noting change in platform direction

This function could perform a calculation similar to predictPosition but doing so would only constantly tell us where a platform will be 1.5 seconds after each check, and that won't be of much help. Instead, the algorithm can take note of the last position that the AI was in before it changed direction and approximate it to be a 'bound' of the platform's trajectory. This notion of bounds works under the assumption that a moving platform follows a back-and-forth motion and would not work if platforms move in a looped shape like a square or a circle.



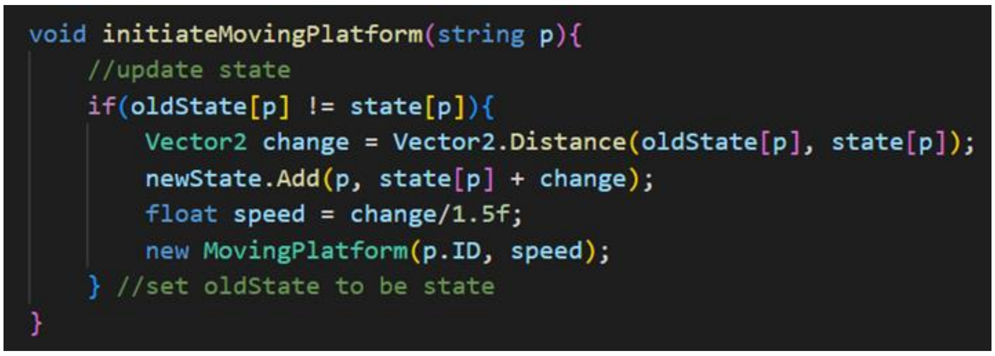
```
void recalculatePrediction(string p){
    //update state
    if(state[p] != newState[p]){
        Vector2 bound = oldState[p];
    }//update oldState
}
```

Figure 57: Storing bound information

Storing the bounds of a platform's trajectory opens the doors to a lot of possibilities for path calculation. In order to maximise those possibilities, the algorithm would need to have a "Moving Platform" class that constitutes of and ID and 3 other attributes:

- Speed (not velocity).
- Bound1.
- Bound2.

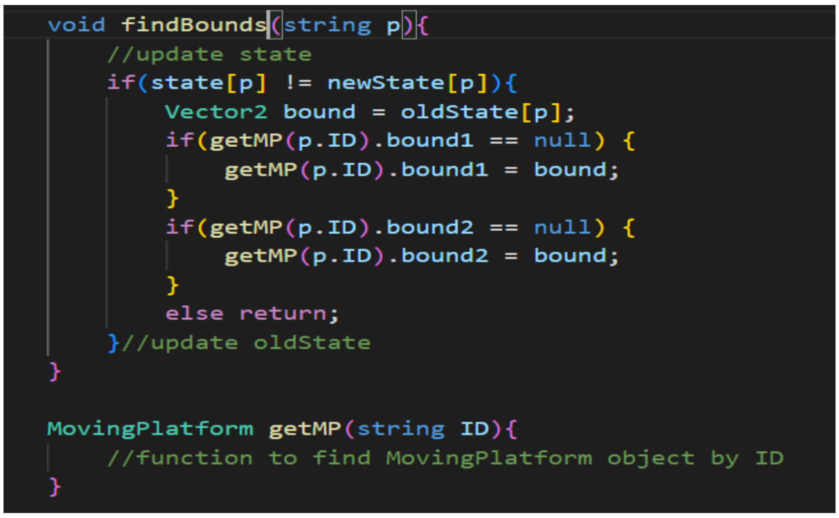
Path prediction can be accomplished using just these 3 attributes. Once the Moving Platform object has all 3 of them filled out, it no longer needs to predictPosition for newState and can simply do so for any point in time. Storing bound information is done as shown in Figure 57 but the algorithm would need to store the speed and ID once it finds a Moving Platform.



```
void initiateMovingPlatform(string p){
    //update state
    if(oldState[p] != state[p]){
        Vector2 change = Vector2.Distance(oldState[p], state[p]);
        newState.Add(p, state[p] + change);
        float speed = change/1.5f;
        new MovingPlatform(p.ID, speed);
    } //set oldState to be state
}
```

Figure 58: initiateMovingPlatform function

The recalculatePosition function can then be renamed findBounds, and instead of just creating a Vector2 bound, we put it in the coinciding Moving Platform object and set it as 1 of the bounds, the other to be determined if the platform changes direction again.



```
void findBounds(string p){
    //update state
    if(state[p] != newState[p]){
        Vector2 bound = oldState[p];
        if(getMP(p.ID).bound1 == null) {
            getMP(p.ID).bound1 = bound;
        }
        if(getMP(p.ID).bound2 == null) {
            getMP(p.ID).bound2 = bound;
        }
        else return;
    }//update oldState
}

MovingPlatform getMP(string ID){
    //function to find MovingPlatform object by ID
}
```

Figure 59: findBounds function

The initiateMovingPlatform and findBounds functions combine to do the following:

- Once a platform moves, note its ID and speed.
- The first time a platform stops moving as predicted, note the last predicted location as a bound.
- The second time it does that, note the second bound for that platform.

As previously discussed, this calculation only works if a moving platform only moves back and forth along the same path and with constant speed, but it has another weakness to it. The bounds are not calculated as the actual position when a platform changes, rather the last stored correct location. This approximation is inaccurate but sufficient under the assumption that the platforms do not move too quickly. If a platform moves slowly enough, the time range between 2 checks should not be enough to greatly affect adjacency calculations for that inaccuracy. This way just makes the system a bit faster because finding the exact position where a platform changes direction would have to be done in Update.

3.7.2: Using Trajectory to Determine Position

A system can compute a platform's position at any given point in time using standard speed, distance and time equations. Calculating this for a platform whose trajectory doesn't change direction is extremely straightforward:

$$pos\ at\ \Delta time = currentPos + (\,speed \times \Delta time)$$

Figure 60: Calculating Position

Platforms that move back and forth still use this principle but extend it quite a bit. The difference between the 2 scenarios is comparable to the difference between an athlete running and swimming 100m. The athlete travels the same total distance in both cases, but his total displacement (if swimming in a 25m pool) would be 0m. I applied that principle to finding the displacement in a back-and-forth loop by enclosing the new position within both bounds, i.e., switching the sign of the added distance whenever the platform reaches a bound. This wasn't too dissimilar to the calculation needed to get angles in the range of 0-360. If Δpos represents the total distance a platform moves from the time of the check and $boundDist$ is the distance between a its bounds, displacement can be calculated as such:

$$\left(\frac{\Delta pos}{boundDist} > 1\right)?\ newPos = \frac{\Delta pos}{boundDist} : newPos = \frac{-\Delta\ pos}{boundDist}$$

Figure 61: Calculating pos over loop

This way, every time the platform reaches a bound and changes direction, the displacement goes from being added to subtracted and vice versa.

3.7.3: How Algorithm Uses Prediction

Once the system can determine where a moving platform will be at any given point in time, it will be able to calculate adjacencies over a range of positions for that moving platform ahead of time. At face value, doing this calculation might seem like a waste of time because the system does this computation on a regular basis anyway, so what is the point of doing a more at a time but less often? Looking ahead of time can help the AI find "shortcuts" on a path, and that is what this entire project is about. Consider Figure 62:

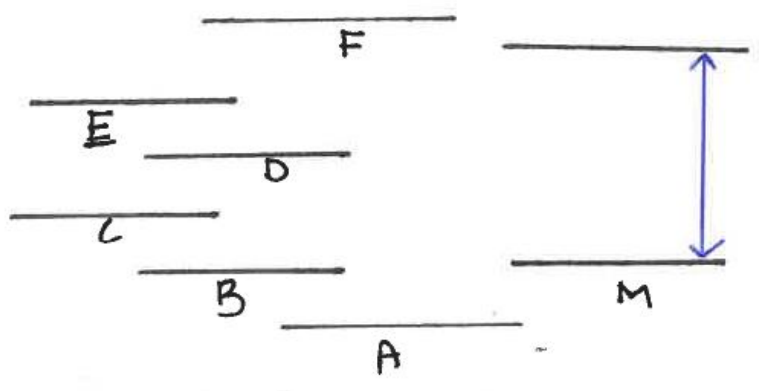


Figure 62: Static vs Dynamic path

There exists a path between platforms A and F that goes between B, C, D and E but there is no point in time where there is a path from A->M->F. There are, however, instances where there is a path from A->M and M->F separately. In order to justify adjacency precalculation of moving platforms, I had to think 1 step ahead of those separate A, M and F connections. To determine if taking M would be faster than taking the static path, the system needs to introduce a notion of cost per second estimation. With a reliable cost system, one would be able to run a series of tests for the AI to follow paths of various costs and find an approximate ratio of how many seconds it takes the AI to traverse 1 cost metric (since costs are distance based). Once there is a good approximation for this, the system can compare the time taken to traverse a static path to how long it would take to traverse a dynamic path (if there is one). This is as simple as getting the total time needed to traverse the static path and determining what the state would look like at that time. This can be done by taking the speeds of every moving platform and finding their position by using the equation in Figure 60.

With that justification in mind, I needed to explore possible ways to determine if there is, indeed, a 'dynamic' path between 2 platforms.

3.7.4: Determining Dynamic Path Existence

Predicting a path is much more intricate than finding and adapting to one, therefore the existing pathfinding algorithm will not suffice. Instead of doing a DFS from the root node to the target, this algorithm would need to find a series of 'reverse' connections from the target to the root node. It looks for 'reverse' connections in the sense that it doesn't recursively check adjacency from target->neighbour, but instead it checks for neighbour->target for all platforms near the target. This is best explained in Figure 63:

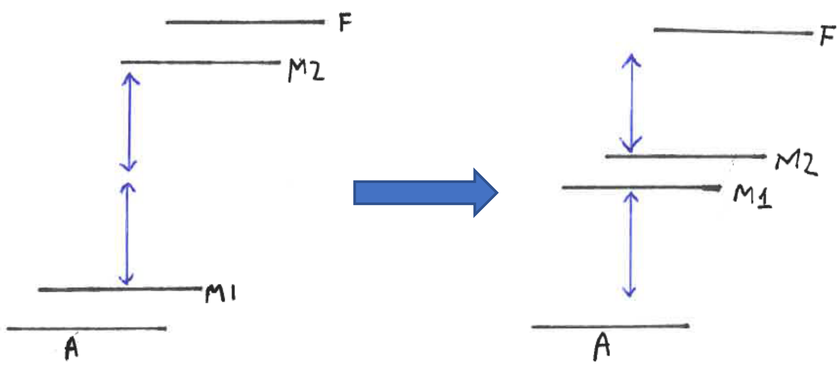
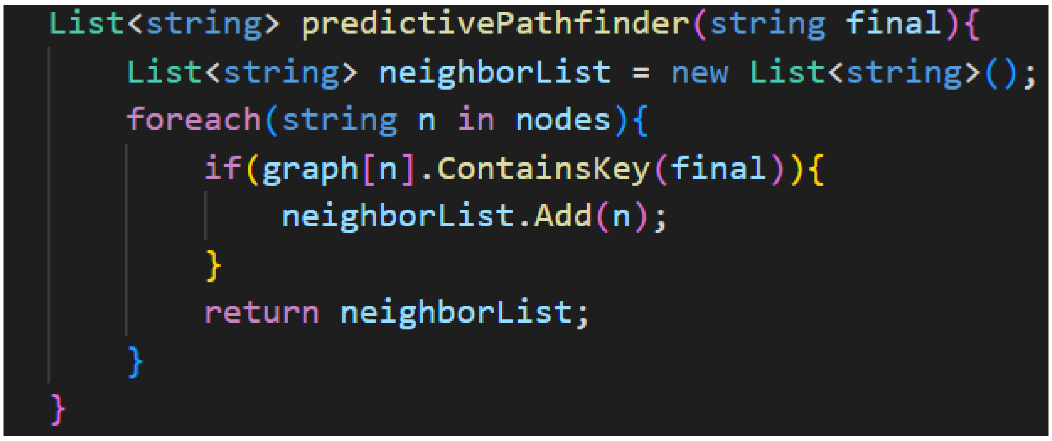


Figure 63: 2 Moving platforms

If the AI needs to determine a path between A and F in this map, it will never find one using the regular search algorithm because paths A->M1, M1->M2 and M2->F never exist at the same time. What the prediction pathfinder does is look for all nodes that can reach platform F. This can be done in 1 of 2 ways. The first is to constantly check for any connections by doing the following check over the graph:



```
List<string> predictivePathfinder(string final){
    List<string> neighborList = new List<string>();
    foreach(string n in nodes){
        if(graph[n].ContainsKey(final)){
            neighborList.Add(n);
        }
        return neighborList;
    }
}
```

Figure 64: predictivePathfinding 1

The idea here is that each item in neighborList will be considered a node and the search would work backwards. With this, the pathfinder will find a path from A->F in Figure 63, but it will be inefficient because it would have to keep checking for each new state. This search can be optimised by using precomputation for platformer trajectory. Doing so would involve 2 Linecasts – 1 for each child object of a moving platform. A Linecast is similar to a RayCast, but it is more suitable in this scenario because it uses both a start and end position instead of just a start. What this means is that the algorithm can create lines whose start and end points are related to bound1 and bound2 of a moving platform, making an extended reachability range. What each node needs to then do is check for any collisions with a Linecast and then add its coinciding node to the neighborList.

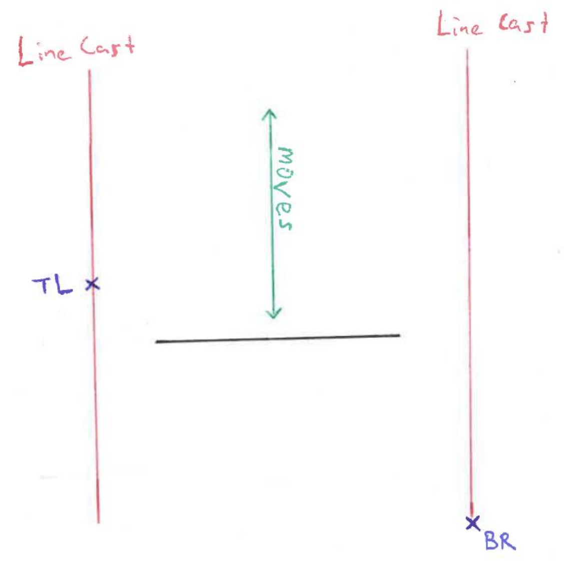


Figure 65: Use of Linecasts

This solution is a lot more efficient because it uses all the information of a platforms movement without the platform needing to move, therefore negating the need for any real time calculations. It is also straightforward and accurate under the assumption that the platform will follow the estimated path and the path bounds are of negligible inaccuracy.

3.7.5: Conclusion

In terms of adding functionality for predicting a path, it is theoretically feasible and doesn't require many more data structures. States already exist and speeds can be found by comparing difference in position at different states. Once there is a speed, the algorithm can determine where a platform will be at a given time. This is more useful for platforms that don't move in a loop, as those platforms already have precomputed ranges.

The weaknesses of this solution are that it is restrictive to simple platform movements to and fro along a line. If a platform moves in any other way or changes its speed, the system cannot work anymore. It is also not as easy to consider how long it would take an AI to traverse over a series of moving platforms because I haven't yet accounted for the time spent waiting for a platform to become traversable. This calculation should theoretically be possible but going into too much detail for an Objective that simply explores possibility and meets no requirements would not be a wise use of my time (Sections 1.6 & 1.7). I can conclude, however, that this addition is possible, and Objective 8 has been met.

Chapter 4: Testing

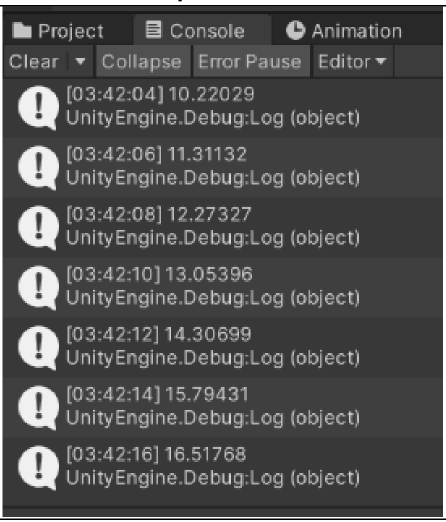
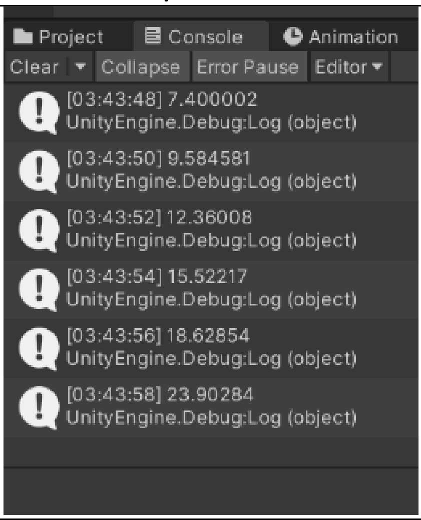
With all the objectives met, I needed to test the reliability of my system by checking how well it achieves each of requirement set in Section 1.5. Certain requirements, such as FR1, FR2, and NFR3, do not need tests because they are already predetermined. The requirements which can be tested (FR3, FR4, NFR1, NFR4, and NFR5), however, can only really be done once the entire system is complete. There are system features, namely Cost System and Heuristic evaluation (NFR2) which I need to do before testing the system's integrity for those requirements.

4.1: Cost System Comparisons

To conduct this test, I created 2 sample scenes in Unity to run 2 tests. The first scene had 1 stationary platform and another that I could move by holding down the movement keys. This tested the consistency of both systems by logging a cost every so often to see if the systems both obeyed the principles defined in section 3.3. The second scene had 100 platforms where the systems had to constantly log the costs between them all. This was to test both systems' efficiency, thus contributing to NFR5.

4.1.1: Integrity Test

I tested both systems to see how the cost from the static to moving platform might change as the moving one goes from below to above the static, and then from left to right. The results can be seen using a series of screenshots from Unity's Console as seen in Figure 66.

	System 1	System 2
Moving platform going from below to above static.	 [03:42:04] 10.22029 UnityEngine.Debug:Log (object) [03:42:06] 11.31132 UnityEngine.Debug:Log (object) [03:42:08] 12.27327 UnityEngine.Debug:Log (object) [03:42:10] 13.05396 UnityEngine.Debug:Log (object) [03:42:12] 14.30699 UnityEngine.Debug:Log (object) [03:42:14] 15.79431 UnityEngine.Debug:Log (object) [03:42:16] 16.51768 UnityEngine.Debug:Log (object)	 [03:43:48] 7.400002 UnityEngine.Debug:Log (object) [03:43:50] 9.584581 UnityEngine.Debug:Log (object) [03:43:52] 12.36008 UnityEngine.Debug:Log (object) [03:43:54] 15.52217 UnityEngine.Debug:Log (object) [03:43:56] 18.62854 UnityEngine.Debug:Log (object) [03:43:58] 23.90284 UnityEngine.Debug:Log (object)

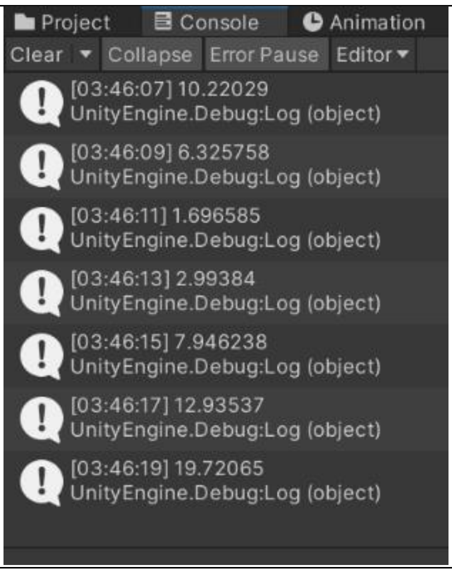
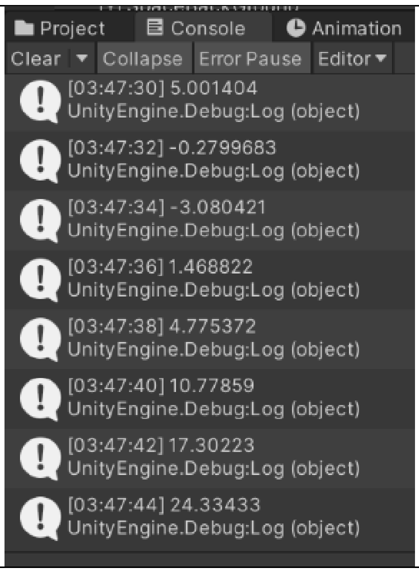
Moving platform going from left to right of static	 [03:46:07] 10.22029 UnityEngine.Debug:Log (object) [03:46:09] 6.325758 UnityEngine.Debug:Log (object) [03:46:11] 1.696585 UnityEngine.Debug:Log (object) [03:46:13] 2.99384 UnityEngine.Debug:Log (object) [03:46:15] 7.946238 UnityEngine.Debug:Log (object) [03:46:17] 12.93537 UnityEngine.Debug:Log (object) [03:46:19] 19.72065 UnityEngine.Debug:Log (object)	 [03:47:30] 5.001404 UnityEngine.Debug:Log (object) [03:47:32] -0.2799683 UnityEngine.Debug:Log (object) [03:47:34] -3.080421 UnityEngine.Debug:Log (object) [03:47:36] 1.468822 UnityEngine.Debug:Log (object) [03:47:38] 4.775372 UnityEngine.Debug:Log (object) [03:47:40] 10.77859 UnityEngine.Debug:Log (object) [03:47:42] 17.30223 UnityEngine.Debug:Log (object) [03:47:44] 24.33433 UnityEngine.Debug:Log (object)

Figure 66: Cost Integrity results

As the table shows, both systems obey the principle of how distance should affect cost on the y and x axes. Some differences in System behaviour include the existence of negative costs in System 2 and not System 1, and the range difference of cost values as platform gets higher up. Due to its linear nature, System 2 experiences the difference in the y axis much more than System 1, therefore giving a much broader set of cost values as the y difference increases.

4.1.2: Efficiency Test

I was able to judge both Systems' effects on efficiency by keeping track of the FPS fluctuations. Due to the FPS fluctuating very often, I was not able to note down every value. What I did instead was note the average FPS range, the lowest and the highest value both Systems produced after 10 seconds of checking.

	Average FPS	Lowest FPS	Highest FPS
System 1	32-37	28.4	39.3
System 2	35-42	32.1	50.9

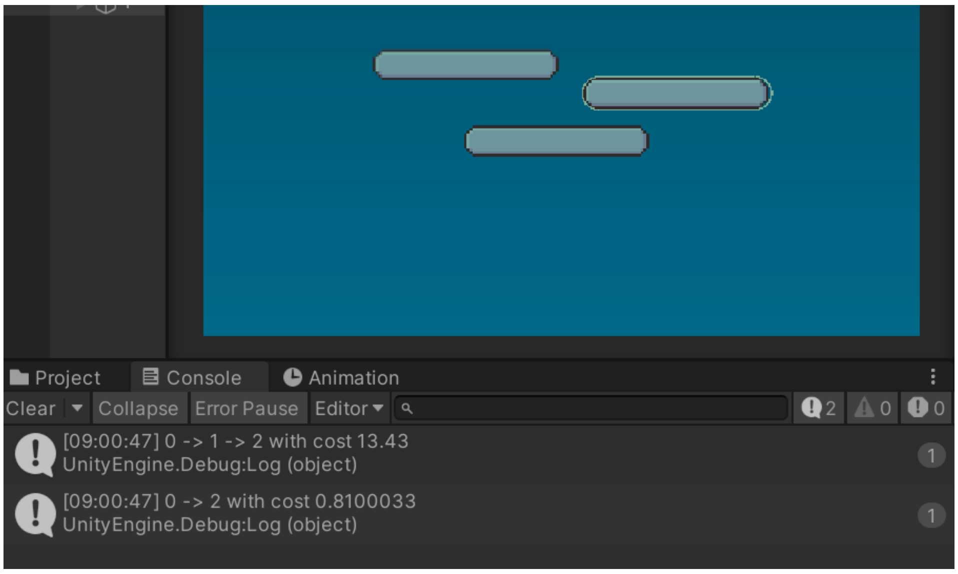
Figure 67: Efficiency Test of both Cost Systems

As predicted, System 2 outperformed System 1 by a considerable amount in each category and is therefore definitively more efficient.

As both systems obey the set cost principles, but System 2 is more efficient, I decided to use System 2 for my solution so that it may have a better chance of meeting NFR5.

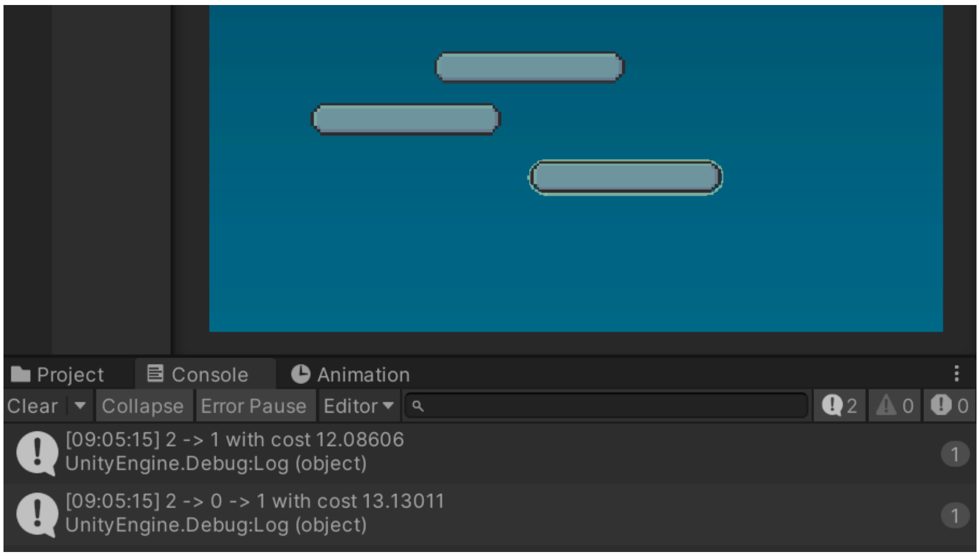
4.2: Path Heuristic Test

In order to determine whether the heuristic described in Section 3.5 holds, I simply created a series of sample scenes in Unity where the pathfinder would return every path from node A to node B. Some, including the scenarios from Figure 38, had fewer platforms than others. All I did was check if every path with x number of platforms had a lower cost than every path with x+1 number of platforms. Below are the results from 2 scenarios where the AI calculates this over just 3 platforms.



```
[09:00:47] 0 -> 1 -> 2 with cost 13.43
UnityEngine.Debug:Log (object)
[09:00:47] 0 -> 2 with cost 0.8100033
UnityEngine.Debug:Log (object)
```

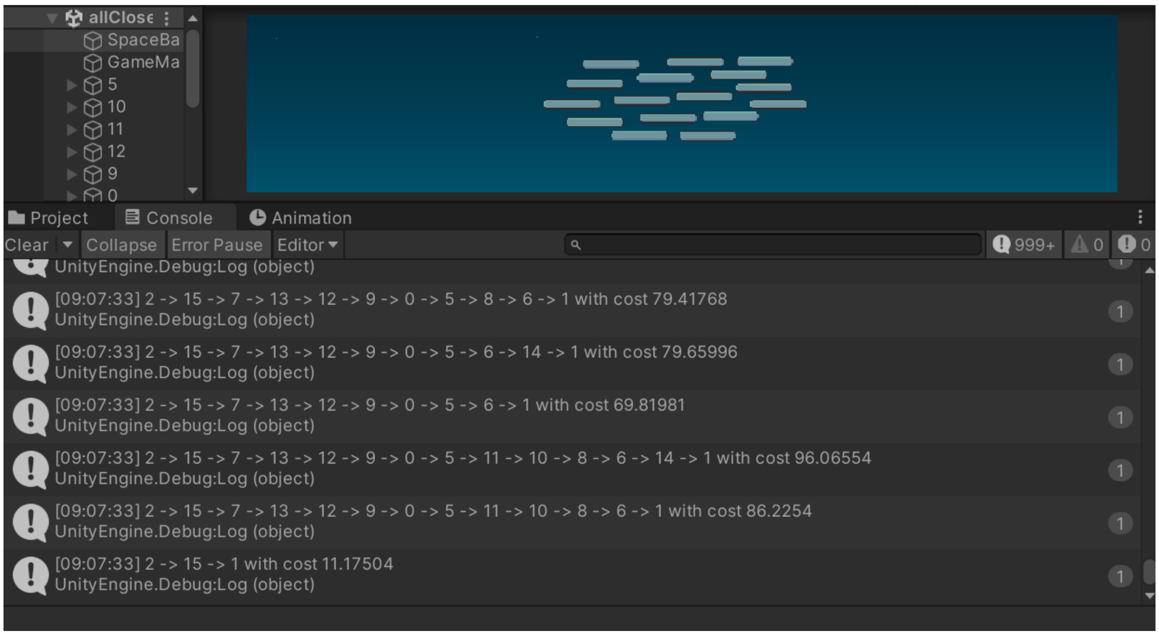
Figure 68 Indirect test 1



```
[09:05:15] 2 -> 1 with cost 12.08606
UnityEngine.Debug:Log (object)
[09:05:15] 2 -> 0 -> 1 with cost 13.13011
UnityEngine.Debug:Log (object)
```

Figure 69 Indirect test 2

It is immediately clear that the heuristic works in scenarios of 3 platforms where the direct path is preferable to the indirect one, but the test won't be accurate without scalability.



```
UnityEngine.Debug:Log (object)
[09:07:33] 2 -> 15 -> 7 -> 13 -> 12 -> 9 -> 0 -> 5 -> 8 -> 6 -> 1 with cost 79.41768
UnityEngine.Debug:Log (object)
[09:07:33] 2 -> 15 -> 7 -> 13 -> 12 -> 9 -> 0 -> 5 -> 6 -> 14 -> 1 with cost 79.65996
UnityEngine.Debug:Log (object)
[09:07:33] 2 -> 15 -> 7 -> 13 -> 12 -> 9 -> 0 -> 5 -> 6 -> 1 with cost 69.81981
UnityEngine.Debug:Log (object)
[09:07:33] 2 -> 15 -> 7 -> 13 -> 12 -> 9 -> 0 -> 5 -> 11 -> 10 -> 8 -> 6 -> 14 -> 1 with cost 96.06554
UnityEngine.Debug:Log (object)
[09:07:33] 2 -> 15 -> 7 -> 13 -> 12 -> 9 -> 0 -> 5 -> 11 -> 10 -> 8 -> 6 -> 1 with cost 86.2254
UnityEngine.Debug:Log (object)
[09:07:33] 2 -> 15 -> 1 with cost 11.17504
UnityEngine.Debug:Log (object)
```

Figure 70: 15 close platforms

The third test, as shown in Figure 70 proves that the heuristic is accurate. I cannot show all the results because the map returned over a thousand possible paths, but from even that small sample, it becomes obvious that each extra platform in a list adds drastically to the cost. The next thing I did was compare the time taken and minimum FPS to find a path with and without the optimisation in Figure 42 for Figure 71 where there are 15 platforms and a lot of different paths to get from 1 to another. This comparison gives a feel of how much more efficient the search is with the optimisation.

	Time taken	Minimum FPS
Optimised Algorithm	14.1s	19.7
Base Algorithm	~1s	29.4

Figure 71: Optimisation difference

From this test, I can conclude that adding this optimisation completely accomplishes NFR2. In an environment where all the platforms are the same length, the optimisation also achieves FR3 but the same cannot be said if platforms vary in length. The map design, however, models the prior scenario and so FR3 is also completely met with this optimised pathfinding algorithm.

4.3: AI Navigation Test.

This is the section that determines the integrity of the entire project. It combines the tests for each requirement to see if AI in platformer game environments can actually adapt to map changes and accurately move around the map, and how well my solution handles this if so.

The first requirement I was able to easily determine was NFR5. Unity has a function that allows a user to see the FPS as the game is running and at no point in any of the tests did the FPS drop below 130FPS. The benchmark I planned to test for is 30FPS, so it is safe to say that this system reached and surpassed its efficiency quota.

The system, however, doesn't completely fulfil NFR4. Upon running the tests, I came to realise that the AI agent always moves to the middle of its next platform when following a path. While this might not seem like a big problem, this detail makes the AI waste time when it moves to a platform behind itself. This is best illustrated in Figure 72 where the platform goes from A->B->C.

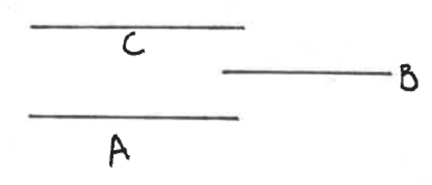


Figure 72: Wasted traversal

In order for the motion to be fluid, the enemy agent should only move a short distance into platform B, but the navigation code states that it should go to the middle.

The AI also jumps each time it reaches another platform. This doesn't necessarily have a direct impact on the movement time because movement in the x axis is unaffected, but a player would not behave like that and so NFR4 isn't met.

FR4, on the other hand, is met. As I noted in Section 3.7, the AI is able to detect map movements (accomplishing NFR1), but its idea of states still paints the map to be static. To expand, the AI knows that a platform has moved, but until the next time it checks, it views that platform as stationary. The effect of this miscomputation is apparent because an AI would not do any movement at all until there is a full path from current to target platform. That means that in situations like Figure 73 it will simply wait for the moving platform to get in range of its neighbours.

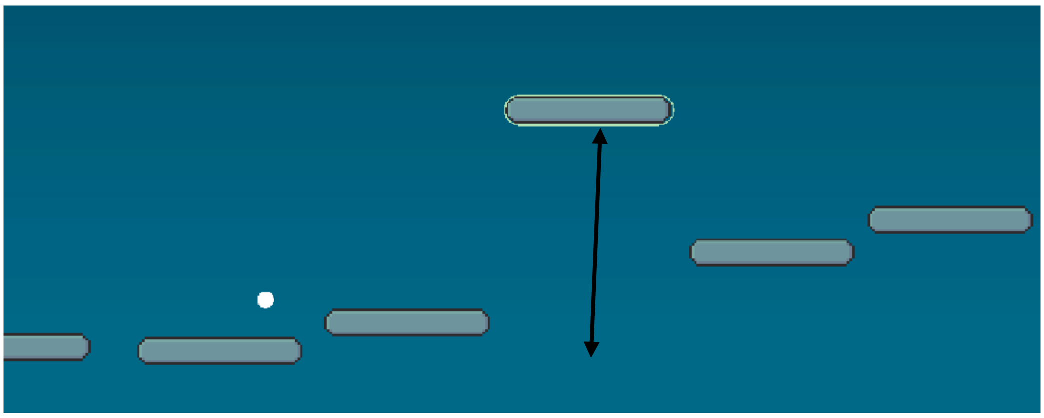


Figure 73: Wait for move problem

The positive aspect to this is that the system is able to realise when there is no longer a path, and therefore will not act on previous information. This problem isn't too costly if the moving platform is right next to the AI because the AI will be able to overcome it as soon as that platform is back in range. The main issue comes when the AI is a few platforms away from the moving one, and therefore its window of opportunity to move is limited to a check for if there is a platform. This problem can be fixed with 2 solutions:

1. Once a path no longer exists, determine which of its platforms has moved and set a path to the platform just before that. Doing this will mitigate the second problem, meaning the AI can still follow the path up until it needs to get to the moving platform.
2. Calculate a path to the target platform that doesn't include the moving one. This can be done by excluding that platform from the search in the recursive pathExists function.

Aside from that, however, the AI always eventually gets to its target platform. Even if it misses a platform when it jumps (which might happen if it tries to go to a platform that's moving too fast), it will simply land on a platform below and recalculate the path. The only time that this would fail is if there does not exist a path from the floor platform to the target one. There is the issue of the AI object becoming a child object of a moving platform thus carrying on its motion, but that is a weakness of Unity and doesn't affect the result I am interested in, so I left it.

Chapter 5: Evaluation.

5.1: Qualitative Requirements Assessments:

As mentioned in Chapter 4, not all the requirements could be assessed by making tests with visible results. Some of them are qualitative, meaning their proficiency is more judgement-based than result based. I have made the following assessments for these requirements:

For FR1, I was able to ensure the rules of a platformer game environment by giving the AI agent a RigidBody component, which allows gravity to act on it, and giving all platforms and the AI Collider Components, which means the AI can stand on a platform without falling through. The platforms themselves don’t have RigidBody components because gravity is not supposed to act on them in a platformer environment.

I was also able to meet FR2 by creating sample scenes (which can be viewed as different maps) with different platform types. Some scenes have moving platforms and platforms can mimic appearing and disappearing features with a click from the user.

NFR3 also doesn’t require a test because the AI’s reachability range is set in the creation of the game environment. I discuss the assessment of its quality in Section 3.2.3, but I was able to improve on it as discussed in Section 3.6.2. As a final remark, the system does not completely fulfil the functional requirement that states it must accurately mimic a reachability range. One weakness is that it doesn’t account for all the freefall reachability. While this problem isn’t completely solved, I believe that using a maximum -y helps somewhat because the AI won’t have too much room to freefall with that abstraction.

5.2: Evaluation summary

In conclusion, all the game environment requirements were fully met. A lot of the project relied on these requirements as the model would not be possible to use unless it had some rules for how the map operates.

Both pathfinding requirements were fully met. The optimisation algorithm optimisation to only return 1 path went a long way in ensuring this.

The agent movement requirements were partially met, with NFR4 being the most unsuccessful of all requirements. I believe this was a result of the lack of priority I placed on navigation because the project is primarily about pathfinding.

The efficiency requirement is completely met. With the optimisation, the whole system is able to operate and barely affect the FPS rate of the game environment.

5.3: Plug and Play functionality

I believe that, with a few adjustments, this solution can be extended to become a library for platformer game environments.

1 of such adjustments would be to edit the reachability range. If Unity had some sort of OverlapShapeAll function that could take in 4 distinct points to create a specific shape, that would have been perfect for modelling the trapezoid abstraction of Figure 9.

There would also have to be a lot of changes made in order to represent various platform lengths and surface effects. Most of the work required to accomplish that level of versatility can be covered by the Keypoint System for the nodes.

The last major change required to make this solution available for plug-and-play functionality would be the need to change the hardcoded values for jump height, angle, time etc. In order to really make this solution versatile enough to be a used as a library, it would need to be flexible enough for a developer to use custom values for things like platform speed.

Chapter 6: Reflective Report and Conclusion.

6.1: What went well?

I believe that I was able to achieve accuracy in this project due to the amount of consideration I gave the system before I even began creating it. Taking time to plan out the ideal solution by considering different strengths and weaknesses of each one really gave me the ability to break the project down into smaller steps. Rigorous planning was definitely a factor in gaining understanding of the problem and solution.

6.2: Where my methodology could have improved.

While it paid off to only consider various Node and Graph solutions without actually implementing both, this was the opposite case for the traversal algorithms. The solution could have benefited from comparing BFS and DFS at the very least. They both used optimisations in the search to find the nearest node and I cannot say for certain which search is better than the other.

I also believe that I could have paid more attention to the navigation algorithm. Although Section 4.3 notes what improvements could have been made, these should ideally be reflected in the code. Finding a path is one thing but adapting to the changes is what makes this project peculiar and interesting within platformer environments.

6.3: Conclusion

In conclusion, this solution proves that pathfinding can be done over a dynamic map. Albeit not every requirement was perfectly met, the project mainly focused on the graph representation, pathfinding algorithm and efficiency – 3 areas where the solution successfully accomplished these objectives.

References

[1] Jones, J (2023) “The Future Of AI In Gaming”. URL: https://www.gamedesigning.org/gaming/ai-in-gaming/

[2] Orkin, J. (2008) “Applying Goal-Oriented Action Planning to Games.” URL:

https://alumni.media.mit.edu/~jorkin/GOAP_draft_AIWisdom2_2003.pdf

[3] Unity Manual. URL: https://docs.unity3d.com/Manual/index.html

[4] Masterclass (2021) “Learn About Platform Game.” URL: https://www.masterclass.com/articles/platform-game-explained

[5] Becker, P., Tebes, G., Peppino, D. and Olsina, L. (2019) “Applying an Improving Strategy that embeds Functional and Non-Functional Requirements Concepts”, Journal of Computer Science and Technology, 19(2), p. e15. doi: 10.24215/16666038.19.e15.

[6] iD Tech (2012) 10 types of platformers in Platformer Video Games. URL:

https://www.idtech.com/blog/10-types-of-platforms-in-platform-video-games

[7] Zhang, P. and Chartrand, G., (2006) “Introduction to graph theory.” Tata McGraw-Hill.

[8] Singh, H. and Sharma, R., (2012) “Role of adjacency matrix & adjacency list in graph theory.” International Journal of Computers & Technology, 3(1), pp.179-183.

[9] Software Testing Help (2023) Graph Implementation In C++ Using Adjacency List .URL: https://www.softwaretestinghelp.com/graph-implementation-cpp/

[10] Koganti, H. and Yijie, H. (2018). “Searching in a Sorted Linked List.” 2018 International Conference on Information Technology (ICIT) (pp. 120-125). IEEE.

[11] Behlanmol (2021) Comparison between Adjacency List and Adjacency Matrix representation of Graph. URL: https://www.geeksforgeeks.org/comparison-between-adjacency-list-and-adjacency-matrix-representation-of-graph/

[12] Algorithm Tutor (n.d.) Graph Representation: Adjacency List and Matrix. URL: https://algorithmtutor.com/Data-Structures/Graph/Graph-Representation-Adjacency-List-and-Matrix/

[13] Mehta, P., Shah, H., Shukla, S. and Verma, S. (2015), A Review on Algorithms for Pathfinding in Computer Games, Conference: IEEE Sponsored 2nd International Conference on Innovations in Information Embedded and Communication Systems ICIIECS'15

[14] QuikGraph library information page. URL: https://github.com/KeRNeLith/QuikGraph