

University of Hertfordshire
School of Computer Science
BSc Computer Science

6COM1053 – Computer Science Project

Final Report
April 2022

A Hierarchical Waypoint Pathfinding
Solution in a 2D Tower Simulation Game

J. C. ASH

Supervised By: Alexios Mylonas

Abstract

One of the most important and common systems in video games is pathfinding. To increase search speed allowing for 1000s of simulated agents, a hierarchical approach can be used, along with an efficient search algorithm. Research conducted showed reducing the graph size and in turn, search space, increased the efficiency of pathfinding. Agents can navigate this graph autonomously through a finite state machine, which can be integrated into the pathfinding system. This paper aims to investigate and implement a pathfinding solution, which can be used in a simulation game with agents.

The literature review conducted looked at past papers and systems in existing video games to determine appropriate strategies moving forward. This secondary research was used to create an understanding for the artefact, which was tested against its requirements through both black and white box tests. It was found that the A* algorithm was the most effective and accessible search algorithm to use for this type of game, and that a globalised scheduling system was best for agent simulation. Alternative systems could be used for differing requirements, and future research should be conducted to determine the most effective systems in each use case.

Contents

Abstract	2
Contents	3
1.0 Introduction.....	6
1.1 Introduction to the Project	6
1.1.1 Topic.....	6
1.1.2 Concepts of Pathfinding.....	6
1.1.3 Concepts of Action Selection	6
1.1.4 Past Games.....	6
1.1.5 Aims.....	6
1.1.6 Objectives.....	7
1.1.7 Goals.....	7
1.1.8 Functional Requirements.....	7
1.1.9 Non-functional Requirements	8
1.1.10 Project Planning	8
1.2 Introduction to the Report.....	8
1.2.1 Terms	8
2.0 Literature Review	8
2.1 Introduction to Literature Review	8
2.2 Pathfinding Approaches	9
2.2.1 Generating the Graph	9
2.2.2 Searching the Graph.....	9
2.3 Action Selection Problem.....	12
2.4 Engine.....	13
2.5 Summary	13
3.0 Methodology.....	14
3.1 Development of Artefact.....	14
3.2 Evaluation of Artefact	14
4.0 Implementation of Pathfinding.....	16
4.1 Generating Graph	16
4.1.1 Build Edges.....	16
4.1.2 Build Graph	20
4.2 Searching Graph.....	22

4.2.1 Implementation of Breadth-First Search	22
4.2.2 Implementation of A*	24
4.2.3 Analysis of Algorithms.....	26
4.3 Agent Simulation.....	27
4.4 Summary	31
5.0 Implementation of Action Selection	31
5.1 Introduction to Action Selection	31
5.2 Implementation	32
5.3 New Graphics	34
5.4 Alternatives.....	34
5.5 New Problem with Lifts.....	34
5.6 Summary	36
6.0 Implementation of User Interaction	36
6.1 Prerequisite	36
6.3 Graph Generating	38
6.4 Summary	38
7.0 Evaluation & Testing	38
7.1 Functional Requirements Test Plan	39
7.2 Performance Tests	40
8.0 Reflection	41
8.1 Introduction	41
8.2 Initial Plan.....	41
8.3 What went wrong.....	42
8.4 What went right.....	43
8.5 Goals	43
8.6 Summary	43
9.0 Conclusion & Future Work	44
9.1 Conclusion	44
9.2 Future Development.....	44
10.0 References	44
11.0 Appendices.....	46
11.1 Appendix A – Gantt Chart.....	46
11.2 Appendix B – Build Edges	46

11.3 Appendix C – Build Graphs	47
11.4 Appendix D – BFS Algorithm.....	48
11.5 Appendix E – A* Algorithm.....	49
11.6 Appendix F – Person Class.....	51
11.7 Appendix G – Lift Class	52
11.8 Appendix H – Global Variables and Objects.....	52
11.9 Appendix I – Agent Simulation	54
11.10 Appendix J – Utilities	63
11.11 Appendix K – Activate Tasks	63
11.12 Appendix L – Find Nearest Room	64
11.13 Appendix M – Check Schedule and Update the Tasks	64
11.14 Appendix N – Adding and Updating Time.....	65
11.15 Appendix O – Room Class.....	66
11.16 Appendix P – On Mouse Down.....	66
11.17 Appendix Q – Evacuate Buildings	69
11.18 Appendix R – Evacuate Buildings.....	70
11.19 Appendix S – Evacuate Buildings	71
11.20 Appendix T – Event Sheet Additional Scripts.....	73

1.0 Introduction

1.1 Introduction to the Project

1.1.1 Topic

The topic of this paper is to look at a comprehensive implementation of waypoint pathfinding, through a hierarchical approach. The pathfinding solution will be implemented in a simple 2D video game, which features a buildable tower that agents can navigate.

Numerous algorithms will be used to create the graph space and to find an appropriate route between a start and destination. An additional sub-problem is to implement a behaviour selection system, which will work in parallel with pathfinding. This will allow the agents to autonomously pick which destinations to travel to, without user input.

1.1.2 Concepts of Pathfinding

Pathfinding is the searching of a graph to find the shortest route. A graph is an abstraction of nodes and edges. These can be contextualised on an environment in a game. For example, buildings in a city could be the nodes and the roads could be the edges. In this case, edges connect at least two nodes together.

Pathfinding spans numerous industries, as it has many use cases. In this paper, pathfinding in video games will be researched, and a suitable artefact will be created. This artefact will be a playable video game, which can be thought of as a playable simulation, rather than a full video game.

1.1.3 Concepts of Action Selection

Action selection is a characterisation of a problem in artificial intelligence (AI), that being: what to do next. Agents in video games need to know what action they should choose next. In this context, an agent is a simulated person, who can act out certain behaviours in accordance with their decisions. For example, an agent can select their destination to path find to, integrating the initial aim.

If an agent can pick what to do next, they have a level of autonomy which allows them to act without user intervention or interaction.

Again, this will be looked at in the context of video games

1.1.4 Past Games

Throughout the history of game creation, many systems and techniques have been used to solve problems. One of the more common and perhaps more sophisticated techniques is pathfinding. Many game engines have built-in algorithms for pathfinding; however, they are often less flexible. A logical step then, is to create a custom pathfinding system, which can be built around a game idea.

In this case, the “idea” is a tower simulation game, in the likes of SimTower or Project Highrise. These games are seemingly able to simulate 1000s of agents going about their day-to-day activities, searching for specific rooms to fulfil their tasks. Therefore, a game will be created with an integrated pathfinding solution and finite state machine to control agents’ tasks.

1.1.5 Aims

There are 3 aims for the project, the primary, secondary and tertiary problems are listed below, in order of relevance.

1. To investigate and implement a comprehensive hierarchical pathfinding solution in a 2D tower building game.
2. To investigate and implement a state machine for autonomous artificial intelligent agents.
3. To have a complete user interface, providing user interaction with the game.

1.1.6 Objectives

To achieve the aims, a list of objectives has been created. These can be thought of as “milestones” to complete the project.

1. Create Placeholder Assets for Game.
2. Create a Basic UI Design.
3. Write Object Classes.
4. Program Pathfinding Algorithm.
 - a. Pathfinding Graph.
 - b. Pathfinding Navigation Script.
5. Create Finite State Machine.
6. Add Graphics to Objects.
7. Implement UI Design.
 - a. Ability for user to pause game, change time scale at runtime.
 - b. Ability to add new rooms to the tower.
 - c. Ability to expand the tower.
8. Test the Implementation.
9. Document Project Implementation in Report Format.

1.1.7 Goals

There are also some personal goals for this project.

- Become more proficient in JavaScript.
- Become proficient in the game engine.
- Learn how to implement a node-based pathfinding algorithm in a game.
 - Acquire a fuller understanding of pathfinding in video games.
- Understand the systems involved in artificial intelligence more generally.

1.1.8 Functional Requirements

There are requirements for the artefact on completion these are shown here.

- **A working graph generation algorithm** – a graph should be generated from a layout of instances.
- **A working graph searching algorithm** – a graph should be able to be searched by either A* or breadth-first search.
- **Have the agents complete actions autonomously** – based on a finite state machine model.
- **Allow the user to control the time scale** – game should be able to be paused, time to be increased.
- **User interaction with creating the building** – the user should be able to place new rooms and expand the building.
- **Display the user interface** – render a user interface to the screen.

1.1.9 Non-functional Requirements

Non-functional requirements are not required for the artefact to work, instead to improve an aspect of the artefact.

- **Clear art** – the game should render the tower with art assets that can be understood easily by the user.
- **Runs well** – the game should run without freezing or slowing down.
- **Ease of use** – the user interface should be clear and understandable, without needing foreknowledge.

1.1.10 Project Planning

To plan for the project, a Gantt chart was created. This features the objectives with time goals. The Gantt chart can be found in the appendix (Appendix A).

1.2 Introduction to the Report

This report documents the stages taken to achieve the objectives. All code has been written within the module and can be found complete in the appendices. The report is largely structured chronologically; showing a narrative of progress, highlighted by the mistakes made and fixed along the way.

Appendices can be found at the bottom of the report, containing additional referenced work, that could not be shown throughout, due to the quantity of material.

1.2.1 Terms

In this paper, “agent” is used interchangeably with “person”, as they are synonymous.

2.0 Literature Review

2.1 Introduction to Literature Review

There are 3 principal areas of research for this topic. Firstly, the game should feature a user interface (UI), where the player can choose to build a new room in the tower. This will offer the basis of user interactivity. A simple UI approach can be used, such as rendering fixed icons to the screen.

The next 2 points of research will cover the majority of the investigation, as they are more complex and are the focus of the paper. The first topic being pathfinding itself. As mentioned earlier, the project is about implementing pathfinding for a 2D tower game. Finding an appropriate search algorithm, generating a path-able graph and exploring advanced optimization techniques are paramount. The second topic is the action selection problem. This is characterised by the ability for agents to choose suitable actions based on a behaviour selection algorithm. This will allow the agents to decide which room to path find to depending on environmental or predetermined factors.

Lastly, there are some questions about how and where this program will be implemented. For example, the engine that will be used and the programming language that the game will be written in. This should also be considered, as an efficient approach is needed to simulate many agents. Drawing on past programming knowledge should also be appraised, as this can make the implementation less demanding.

2.2 Pathfinding Approaches

Pathfinding is defined as the ability for a computer to find the shortest path between 2 points. Before a computer can determine a path, a graph needs to be generated. Then, a search algorithm can be used on the graph to find a suitable path.

2.2.1 Generating the Graph

A paper titled “A Comprehensive Study on Pathfinding Techniques for Robotics and Video Games” (Zeyad Abd Algfoor, Mohd Shahrizal Sunar, Hoshang Kolivand, 2015) documented the fundamental components of pathfinding in different situations. For example, in robotics, graph generation becomes more difficult due to complex terrain environments. In video games, grid-based methods can be used to create a graph. In practice, this would mean that the graph is made up of a grid with vertices and edges. This solution would perhaps be inadequate as a grid restricts the nodes to a set layout.

To avoid this restriction, the paper explains a waypoint system approach, which would have nodes placed irregularly on the graph. A journal titled “Waypoint Graph Based Fast Pathfinding in Dynamic Environment” (Zhu et al., 2015) looks at a waypoint system and notes that a dynamically changing environment makes the pathfinding more challenging. To reduce the complexity, two steps can be taken. First, to eliminate unnecessary waypoints (these are also called nodes) and edges (also called links) in the graph. This will naturally reduce the graph complexity, reducing processing demands. The second way to simplify the graph is to segment the long edges to create localised graphs, which can be translated hierarchically later on.

In summary, the paper found that the dynamic environment caused a “large computational overhead” and that their proposal was to reduce waypoints, simplifying the environment, making it more efficient.

Other programmers have had to deal with reducing the complexity of their path finding implementations. For example, in the book *Game AI Pro 3*, in chapter 34, a game developer talked about their solution to the problem (Zubek, 2017). The chapter was titled “1000 NPCs at 60FPS” and is about a game similar to the one being created. It features a 2D cutaway of a skyscraper that the player is building. The game is called “Project Highrise”. As the name of the paper suggests, the game simulates up to 1000 NPCs (agents) pathfinding simultaneously. To achieve this, the developer decided to simplify the graph by counting each floor as a node and just using linear interpolation to move agents across each contiguous floor. Then, each floor was connected through edges which were created depending on the elevators in the building. In short, this reduced the search space, as it went from “15,000 nodes to 100 nodes and 400 edges” (Zubek, 2017). The performance gains were so notable, that the pathfinding algorithm did not need to be optimised and an algorithm called A* could be used natively. Where this approach lacks, is in the edges, as elevators act by teleporting agents to different floors. This is different to the problem, where there are simulated elevators, which go to different floors and where agents must queue to get on.

In summary, optimisation of the algorithm starts with the generation of the graph. To achieve the desired outcome, an experimental approach can be used, where numerous techniques are tried. The most relevant system can then be applied in the artefact, taking ideas from similar previous games.

2.2.2 Searching the Graph

The graph for pathfinding can be gridded, or irregular. It can feature many edges or be simplified and can be flat or hierarchical to reduce scope. Once a graph is generated, it can be navigated. This is done with a

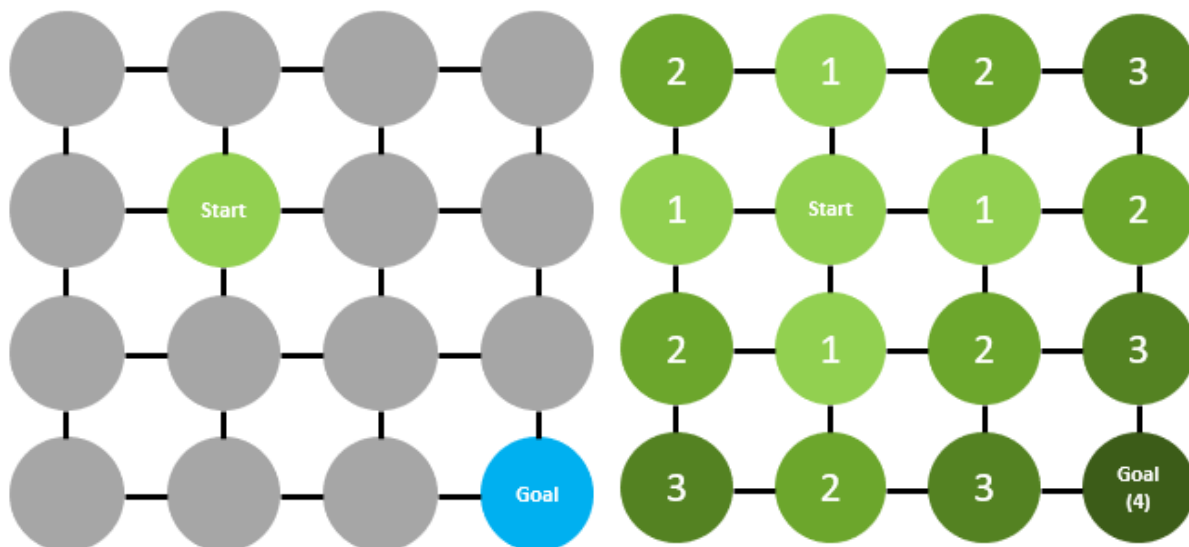
search algorithm. There are many different types of algorithms. These differ by the way that nodes are checked in a graph.

An approach to this research was to look at the simplest algorithms and then work up to more complex, and perhaps more relevant algorithms, disregarding the less efficient predecessors. One of the simplest algorithms is called breadth-first search (BFS). A blog titled “Redblobgames” (Redblobgames.com, 2014) discussed various common pathfinding algorithms. One of which was the aforementioned breadth-first search. This search works by examining each neighbouring node and then when finished, examines the neighbours of the previously explored nodes. This is known as an uninformed search, as it works by brute force, without a heuristic (shortcut).

Suppose we have a start (represented by the green node) and a goal (represented by the blue node). In Figure 1 (below), every node is connected as a grid to all its immediate neighbours. Every time a node is explored it is highlighted green with a number representing the iteration it was checked.

Starting graph:

After 4 iterations:



(Figure 1)

In BFS, all nodes are explored equally. After 4 iterations, the goal is explored, and the algorithm is complete. This is inefficient, as many nodes that do not lead to the goal were explored.

BFS is similar to depth-first search, where instead of searching all neighbours (the breadth of the graph), it instead searches the depth of the graph, which is useful in other situations.

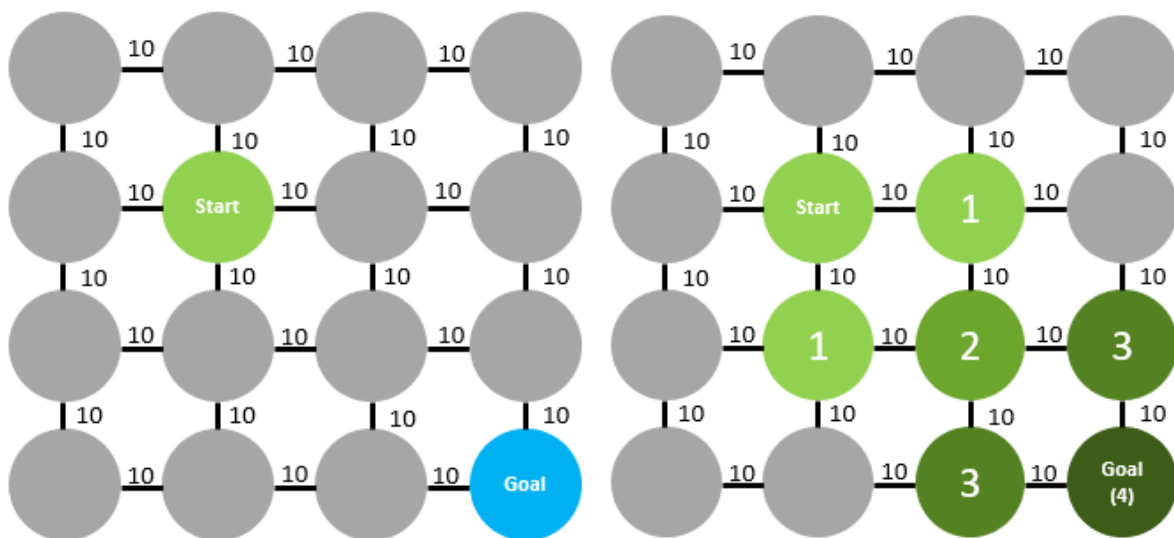
Alternatively, there is Dijkstra's search algorithm. A paper titled “A Method for the Shortest Path Search by Extended Dijkstra Algorithm” (Noto, M. and Sato, H., 2000). Dijkstra's algorithm is the “immediate precursor of A*” it is more informed than BFS, as it has a cost function. Making the algorithm able to find the shortest path. This juxtaposes BFS which does not necessarily find the shortest path. Finding the shortest path is useful, as it prevents agents making irrational decisions such as following a longer path, taking more time to get to their destination.

A* is another algorithm which takes cost into account. What differentiates it from Dijkstra's algorithm is the addition of a heuristic. A paper named “A*-based Pathfinding in Modern Computer Games” (Cui, X. and Shi, H., 2011.) documents this algorithm, which is one of the most popular search algorithms used. This algorithm works by “repeatedly examining the most promising unexplored location”. The algorithm is complete when it has explored the goal destination and then it reverse engineers its steps to return the path. This contrasts BFS, which disregards searching the most promising nodes and chooses to systematically search through every node at the current depth until it reaches its goal. A* can achieve this because of a heuristic and cost approach, which weights the nodes on desirability and distance.

In contrast to BFS, because A* uses both cost and a heuristic, it can orient itself, choosing more desirable nodes. This is shown in Figure 2.

Starting graph:

After 4 iterations:



(Figure 2)

Every node will be connected by edges. These edges have a cost (represented by the number 10). As the graph is a grid, the costs will be uniform. However, this is seldom the case in application. The algorithm will prioritise nodes based on a queue, which is ordered by a function $f(n)$. Here, the function is the cost added to the heuristic, where the heuristic is the distance between the node and the goal. As this algorithm is informed, it will require higher memory usage compared to BFS.

The paper also mentions how reducing the search space is important in speeding up the algorithm. For example, a grid of 100 by 100 nodes has 10,000 potential places to search. Therefore, reducing the number of nodes will reduce the amount of searching needed before finding the goal.

The paper also touches on hierarchical pathfinding also known as HPA*. This approach starts with a high-level graph, which is more abstract and has less nodes. The path is first found on this graph and then a lower-level graph is used to fine tune the path later on. This reduces computing power but should be reserved for much larger search areas. Also, Zubek (2017) noted that they did not use HPA*, as a specialised and more compact hierarchical graph could be made, which better supported the player

making changes over time. HPA* could be used if the extra efficiency is needed, or an alternative system which is more suitable for the game could be implemented instead.

Comparison of Efficiency in Pathfinding Algorithms in Game Development (Krishnaswamy, 2009) is a paper that looked at different search algorithms in a game. It detailed how the A* algorithm is often regarded as the “de facto” standard in game development. As mentioned above, A* uses heuristics to weight nodes for decisions. This is done by calculating the distance between the current node and the destination node. To do this, the paper describes using Manhattan distance in their tests. However, there are alternatives, and a basic built-in distance function should suffice in any given engine.

Uggelberg, R. and Lundblom, A. (2017) in their paper named “Comparative Analysis of Weighted Pathfinding in Realistic Environments” mention that there are more algorithms, but they tend to be more niche and specialised. For example, D* which employs an “incremental search strategy” which is advantageous in large unknown search areas.

JPS (jump point search) and JPS+ are known to significantly increase search speed by jumping to significant points on a grid and not visiting the same nodes twice (Harabor, D.D. and Grastien, A., 2012), (Rabin and Silva, 2015). These solutions may be unnecessarily complex for the solution and often work exclusively on a grid. JPS+ is significantly faster than A*, but as mentioned by Zubek (2017) and Zhu et al. (2015) it may not be necessary to tweak the final algorithm once a simplified waypoint graph is generated.

In summary, to search the graph the most appropriate algorithm would be A*, due to it being lightweight and versatile, as well as having ample documentation. BFS should also be implemented as a comparand to A*, to test the claims about an efficiency increase from the search algorithm alone. As mentioned previously, emphasis should be put on generating the graph, as a simplified waypoint system will reduce the scope of the search, increasing efficiency. Here, a similar approach to Zubek (2017) can be used for the waypoints.

2.3 Action Selection Problem

Another important topic of research is the action selection problem. This is defined as the means by which an agent chooses which action to perform next. Non-playable characters (NPCs) in video games usually have some form of artificial intelligence. A chief part of this intelligence is the ability for the NPC to select actions from a range of predetermined choices. This is often done through an environmental approach, where the NPC decides what to do next depending on its surroundings.

A paper titled “Action Selection and Individuation in Agent Based Modelling”, (Bryson, J.J., 2003) discusses different models of action selection. These are categorised into “Environmental Determinism” and “Finite State Machines”. The former model is more generic and is defined as actions being mapped to environmental situations. When the environment is enumerated, an algorithm decides which action should be applied depending on the state. A famous example of this would be the cellular automaton “Conway’s Game of Life”.

Alternatively, there is a finite state machine. This is more agent focused, asking the agent what its current state is and how to transition to another state. This is advantageous over the other model, as transitions between states are clearly defined.

Zubek (2017) also talked about the action selection problem in the same chapter mentioned earlier. The solution they found after experimentation, was to implement a “daily script” system which simulates

“entire daily routines at a high level”. Libraries of “stereotyped scripts” were stored in memory and picked out during runtime to create a schedule. These worked by providing block events such as from 8 to 18 [work-at-office] and one-off events with probabilities such as at 13, 80% chance of [buy-coffee]. In a state machine, this would be represented as different states: [buy-coffee] and [work] with transitions based on preconditions, for example the time of day.

There is also goal-oriented action planning (GOAP), which can be optimized by shrinking memory and runtime requirements. This was documented in the chapter “Optimizing Practical Planning for Game AI” in the book Game AI Pro 2 (Jacopin, É. 2015). GOAP is an alternative to a finite state machine and works by allowing the AI to perform sequences of actions (Game Development Envato Tuts+, n.d.). These actions are weighted and the shortest path to the goal is chosen by adding up the weights from each action. This simplifies an overly complex state machine and is implemented using a GOAP planner.

“Behavior Selection Algorithms” a chapter by Dawe et al. (2013) in Game AI Pro lists the action selection algorithms, including GOAP. They mention how GOAP uses a “backwards chaining search”, which is where you start at the goal and work backwards through the actions. Alternatively, there are forward planners such as hierarchical task networks (HTN), which work in an analogous way.

Overall, a finite state machine will be used in the solution, due to its simplicity and efficiency when few actions are needed. GOAP and deterministic models are overly complex or too inefficient to be desirable here.

2.4 Engine

Finally, to implement the solution, a suitable engine should be chosen. A game engine is the framework which the game will be developed in. It will provide libraries with functions and methods. Some engines will have premade pathfinding functions. However, these can be ignored as the solution involves making the algorithm from scratch.

There are several engines, which differ based on the language they use and the features they provide. One of the most notable engines is Unity (Unity Technologies, 2021). This is a proprietary engine with built-in systems, such as a navigation system for pathfinding. It uses C# as the language.

Alternatively, there is Godot (Godot Engine, 2021). This is similar to Unity but is open-source and uses a python-like Godot script language.

There are many more engines including a JavaScript engine called Construct 3 (Construct.net, 2021). JavaScript being the most sought-after language for developers (Hughes, n.d.). Having prior knowledge in Construct 3 makes the engine more approachable as well.

Despite Construct 3 being a 2D engine exclusively, it is desirable for its ease of use and efficiency when compared to alternatives. It features a visual scripting language as well, but JavaScript will be how the algorithms are implemented.

2.5 Summary

After discovering several search algorithms such as A*, Dijkstra and JPS+ in the research, the most appropriate algorithm would be A* for its versatility and ample documentation. This can also be used on a waypoint graph, which is what the game will use.

To determine actions for the agents, a finite state machine will be adequate; GOAP and alternatives are unnecessarily complex for the solution.

The solution will be implemented in the Construct 3 engine using JavaScript. As this engine has the desired features, such as a built-in layout editor, but also provides a platform for complex scripts to be written. The solution will use the algorithms mentioned above and will use an object-oriented approach for the agents.

3.0 Methodology

There were 2 main sections of research in the literature review, the pathfinding and the action selection. The problems being researched were outlined in the aims, and included the questions “What is the most effective pathfinding approach?” and “How can pathfinding be combined with artificial intelligence to create autonomous agents?”

Systems were discovered from the literature review through researching papers, journals and books on the relevant topics. The literature was broadly experimental, introducing methods of developing software given a specific situation. For example, the chapter from Zubek (2017) was written by a game developer outlining his approach to the problem “What is the most effective pathfinding approach?”.

3.1 Development of Artefact

This secondary research can then be used to develop an artefact, which expands on previously explored topics.

Certain systems can be measured quantitatively, such as the time it takes to compute an algorithm, or the size of a graph in memory. This should be used in the development of the artefact to ensure that the solution is optimal for the aims.

For example, both A* and breadth-first search will be implemented and compared. Since they are both search algorithms, they use similar variables which can be compared. The most efficient algorithm should then be used for the continuation of the artefact, after weighing variables appropriately.

When it comes to user interface (UI), a qualitative approach can be used during development. Common practices can be employed, such as positioning UI elements in the top left of the screen, where people often look first, or where people prefer the UI to be.

There have been no surveys, questionnaires or other user input during research and as such, there have been no forms to complete.

3.2 Evaluation of Artefact

As the artefact is a piece of software, for it to be successful, it must meet specific requirements. Such as both functional and non-functional requirements. The requirements were stated in the introduction section. A test plan can then be created from the functional requirements, an empty test plan is shown below.

Testing	Description	Expected Result
Graph generation (main graph)	Run the project after creating a tower in the engine editor with no rooms	A graph is generated

Graph generation (subgraph)	Run the project after creating a tower with rooms in the engine editor	A graph and subgraph are generated
A* pathfinding	Set a start and destination on the same floor	A path is found between the nodes
A* pathfinding	Set a start and destination on different floors, connected by a lift	A path is found between the nodes
A* pathfinding	Set a start and destination, where they are not connected	No path should be found (return no connection)
Agent simulation	Set a start and destination on an agent, so they can move to their destination	Agent should move to their destination
Agent simulation	Make an agent move between floors to test the lift movement	Lift should move to the appropriate floor
Agent simulation	Make an agent move between floors to test lift queue	Lift should pick up agent and add them to the queue
Agent simulation	Make an agent move between floors to test lift states	Lift should drop off passengers, changing state to "waiting for passengers"
Action selection	Wait until 8:00 am to see if agents arrive for work	Agents arrive at work
Action selection	Wait until 12:00 pm to see if agents go to lunch (only workers)	Worker agents go to lunch
Action selection	Wait until 5:00 pm to see if agents go home	Worker agents leave the tower (go home)
User interaction	Press the pause button	Game should pause (freeze time)
User interaction	Press the play (1x speed) button	Game goes at 1x speed
User interaction	Press play (5x speed) button	Game goes at 5x speed
User interaction	Press build button	Game pauses and build menu appears
User interaction	Click on room or building button in the build menu	Game creates an outline of the new room
User interaction	Click after the outline is showing (for room)	A room is placed where the cursor is
User interaction	Click after the outline is showing (for building)	A building is placed where the cursor is

The non-functional requirements are not quantifiable, except for measuring the performance of the artefact. To test performance, the game should be run with varying amounts of agents. A table containing the test information is shown below.

Description	Desired result
Run the game with 5 offices, containing 10 agents each. (50 agents in total)	The game runs with dropping frames and without a CPU utilisation above 10%.
Run the game with 10 offices, containing 10 agents each. (100 agents in total)	The game runs with dropping frames and without a CPU utilisation above 10%.

Run the game with 20 offices, containing 10 agents each. (200 agents in total)	The game runs with dropping frames and without a CPU utilisation above 10%.
Run the game with 50 offices, containing 10 agents each. (500 agents in total)	The game runs with dropping frames and without a CPU utilisation above 10%.

4.0 Implementation of Pathfinding

Now that the research has been conducted, an appropriate plan should be put in place. The breakdown of the project can be summarised as primary, secondary and tertiary problems. Here, the emphasis is on the pathfinding itself. This is broken into the generation of the graph, searching of the graph and then the agents' ability to use the graph in the game.

An exploratory method was used during development, where different approaches to generation were implemented to analyse their efficacy.

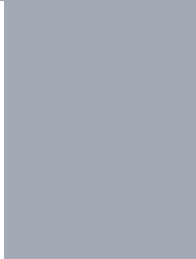
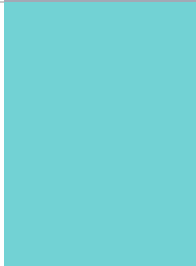

4.1 Generating Graph

Before searching the graph, the graph needs to be created. The game features several instances of objects. These are used together to create a tower.

4.1.1 Build Edges

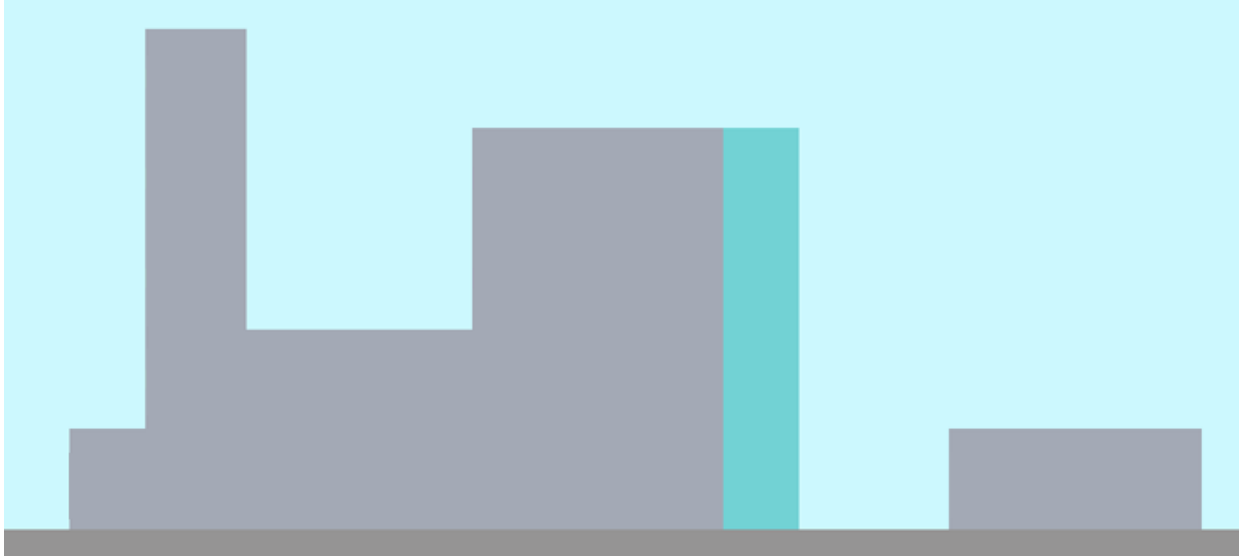
A graph is an abstraction of edges, which are two points connected to each other. For example, (Point A, Point B) is an edge between points A and B. We can store a list of these edges from the graph.

To start off, a tower should be created in the game engine editor. Before art is created, simple shapes can be used to represent the different rooms. A key is found in Figure 3.

Graphic	Meaning
	Building
	Lift shaft
	Node

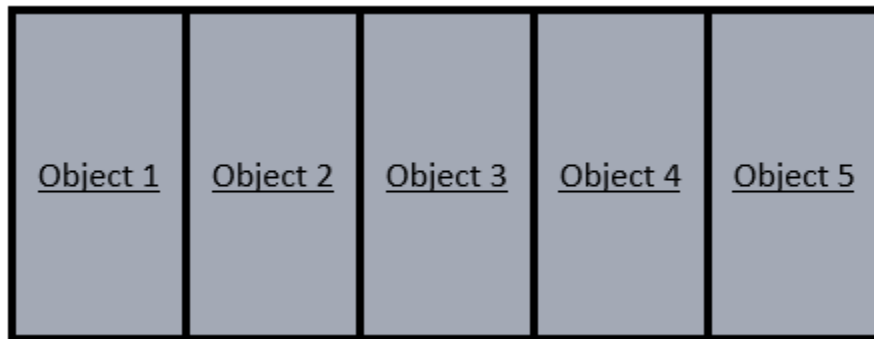
(Figure 3)

The tower created for demonstration is shown in figure 4. You can imagine this as a cross-section of a building, with the ground at the bottom.



(Figure 4)

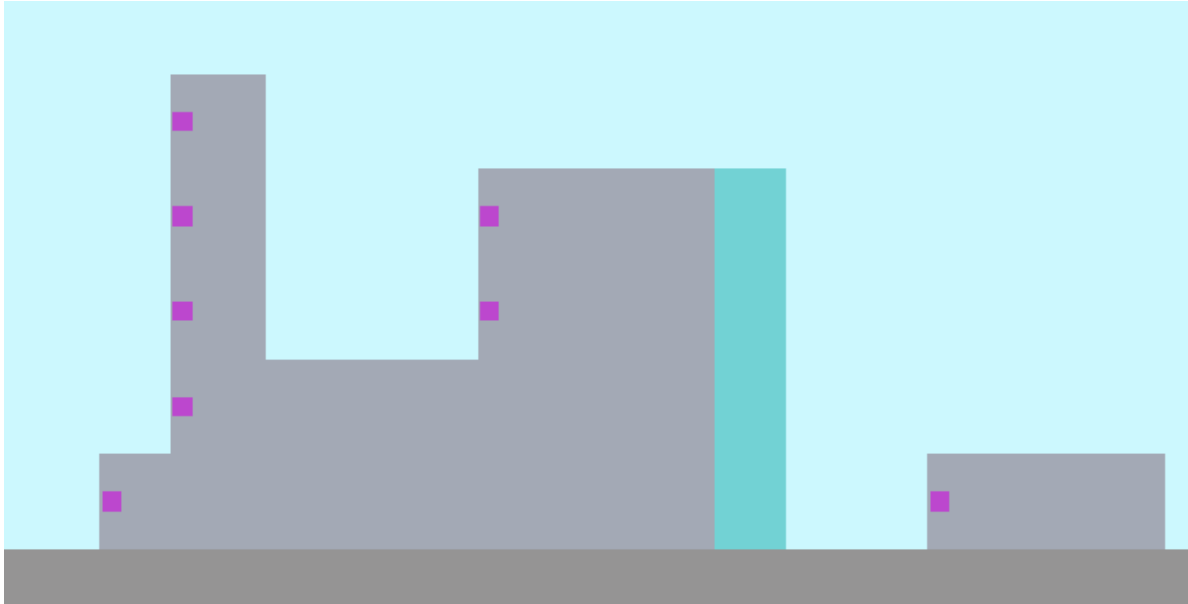
There are several approaches to this problem. Firstly, nodes should be generated. This is so the edges can be developed, as an edge is between two nodes. There are multiple methods of generating nodes, however, and to keep the graph hierarchical, more abstract nodes should be made first. For example, in Figure 4, the tower is created out of many building instances put together, this is visualised in Figure 5.



(Figure 5)

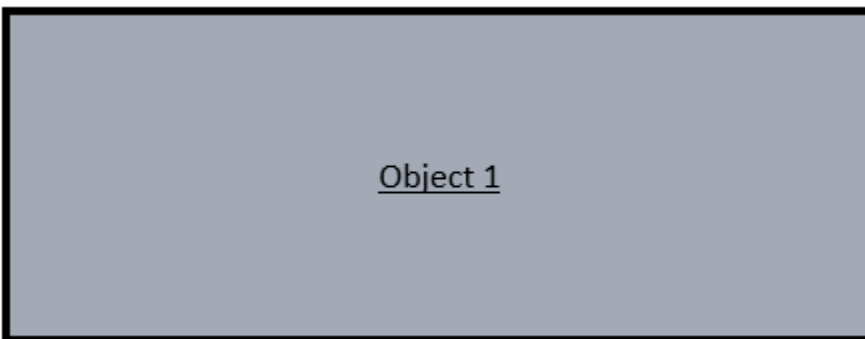
If the graph was not hierarchical, a node for each object can be created. Neighbouring nodes can then be connected by edges, solving the first part of the graph generation. However, to reduce the size of the graph, keeping in line with what was found in the research (Zubek (2017) and (Cui, X. and Shi, H., 2011.)), fewer nodes should be created. To do this, all the objects in Figure 5 should be treated as one.

Originally, the plan was to iterate each object and if there was no object to the left, assign a node. This meant that blocks of contiguous building instances were treated as one node. This is shown in Figure 6.



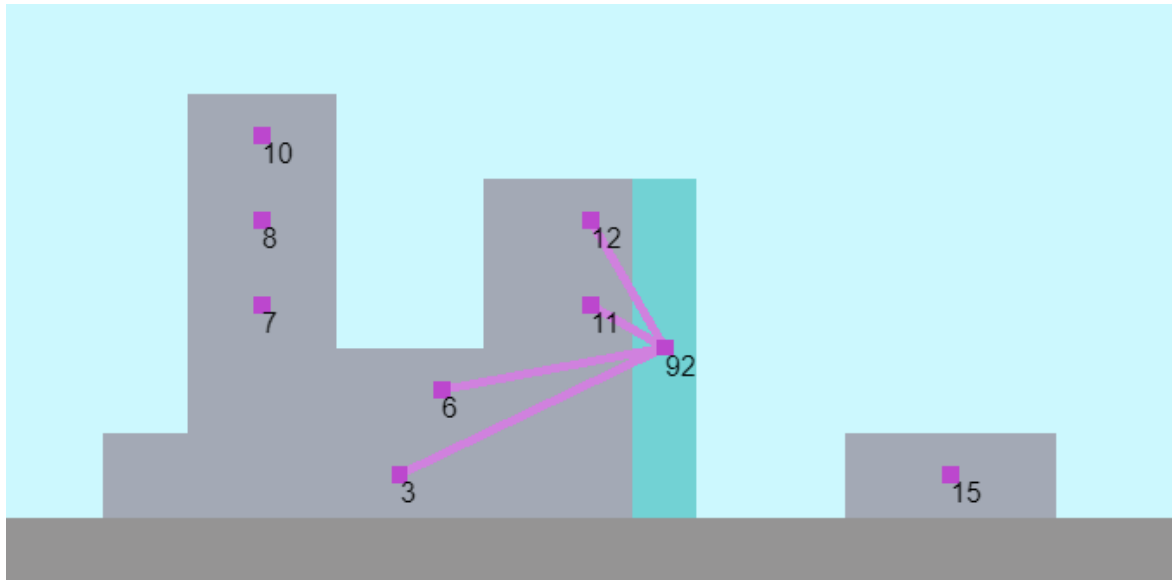
(Figure 6)

However, this solution became problematic once edges had to be generated. This was because relevant nodes had to be connected to the lift shaft. Because there were multiple building objects this was not possible, without an unnecessarily complex function. This was solved by reducing the amount of building objects and just expanding the width of the object when needed. For example, Figure 5 turned into Figure 7 (below).



(Figure 7)

Once this change was made, a simple overlap function would suffice to generate the edges. This worked by checking if the building instance overlapped the lift shaft instance. If so, then an edge would be created. This is shown in Figure 8.



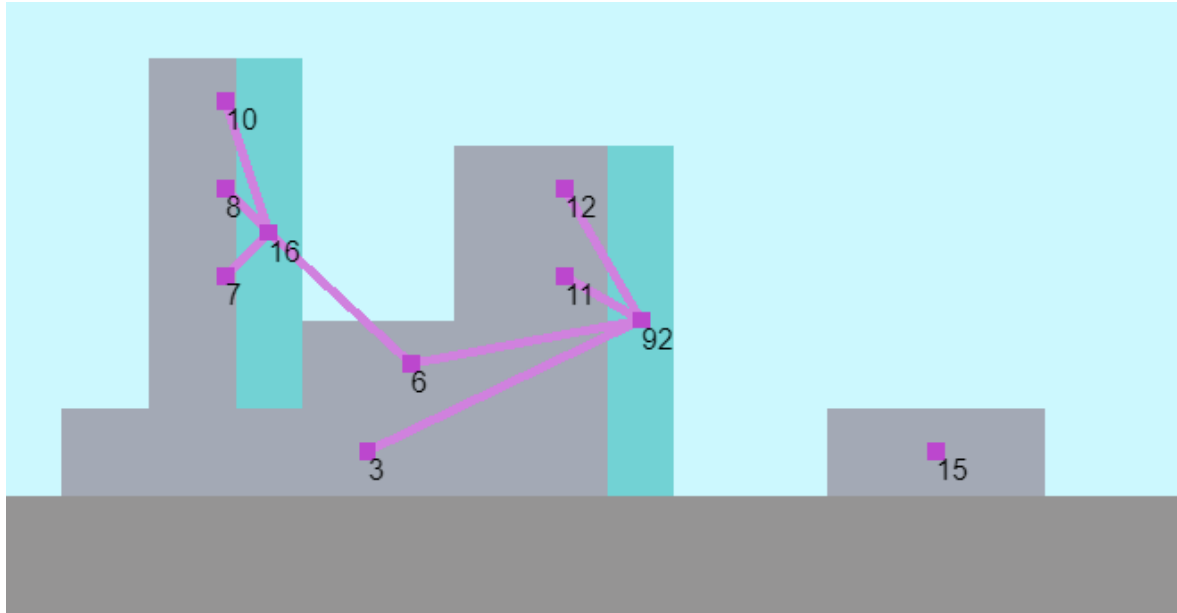
(Figure 8)

The edges can then be visualised, creating a topology. Edges are stored in a two-dimensional array, which is shown in Figure 9. The completed “buildEdges” function is shown in Appendix B.

```
▼ (4) [Array(2), Array(2), Array(2), Array(2)]
  ► 0: (2) ['3', '92']
  ► 1: (2) ['6', '92']
  ► 2: (2) ['11', '92']
  ► 3: (2) ['12', '92']
    length: 4
  ► [[Prototype]]: Array(0)
```

(Figure 9)

Figure 9 shows the 4 elements in the array. Each element stores the two points of the edges. For example, node 3 is connected to node 92. (In this case the nodes are just a random instance ID). Nodes 10, 8 and 7 are not connected to a lift shaft and as such, there is no way of travelling from those nodes to any other node. However, it would be possible to travel from node 3 to node 12, as they are connected through node 92. An additional lift shaft can be used to connect the floors, shown in Figure 10.



(Figure 10)

4.1.2 Build Graph

Now that the edges can be generated dynamically, a graph can be created. This can be made from the edges, which is why they were found first. An algorithm can be used to convert from edges to a fully searchable graph.

The algorithm works by iterating the edges array and adding connections to a dictionary. Because JavaScript does not have dictionaries, an object was used instead. Figure 11 shows the graph that was generated from the edges from Figure 9.

```

{3: Array(1), 6: Array(1), 11: Array(1), 12: Array(1), 92:
(4)}
  ▶ 3: ['92']
  ▶ 6: ['92']
  ▶ 11: ['92']
  ▶ 12: ['92']
  ▶ 92: (4) ['3', '6', '11', '12']
  ▶ [[Prototype]]: Object

```

(Figure 11)

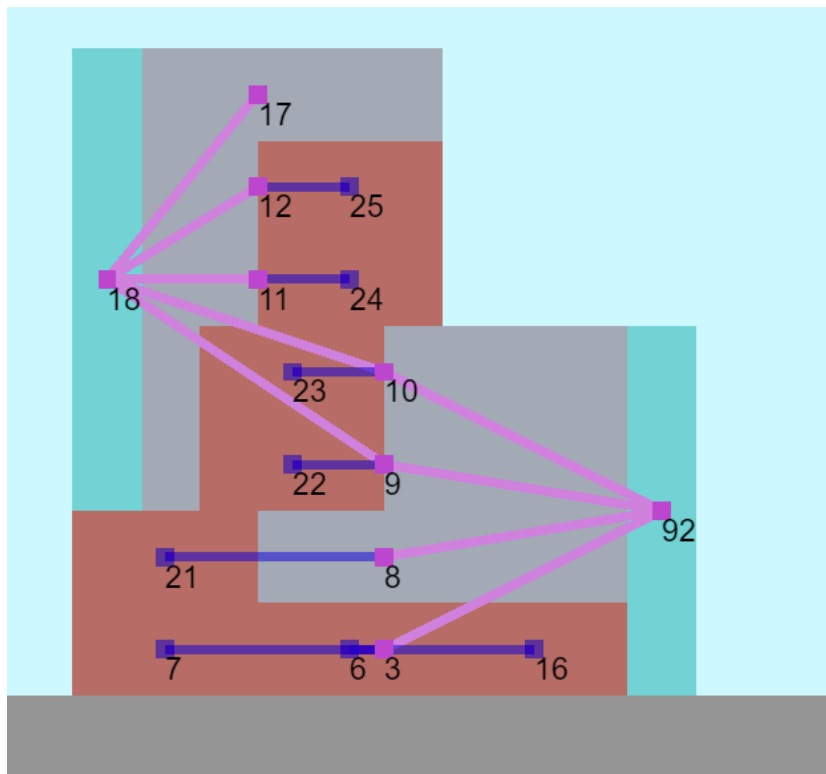
The graph works by storing each node as an object, which has a corresponding 1-dimensional array (list) of the connecting nodes. For example, node 3 connects exclusively to node 92. Whereas node 92 connects to 3, 6, 11 and 12.

If the graph was not hierarchical, it would now be searchable. However, the tower in this game will feature rooms. Rooms are an extension of the building instance that they belong to. In other words, a room instance is a child of a building instance. A key with the new graphics is shown in Figure 12.

Graphic	Meaning
	Room
	Sub node

(Figure 12)

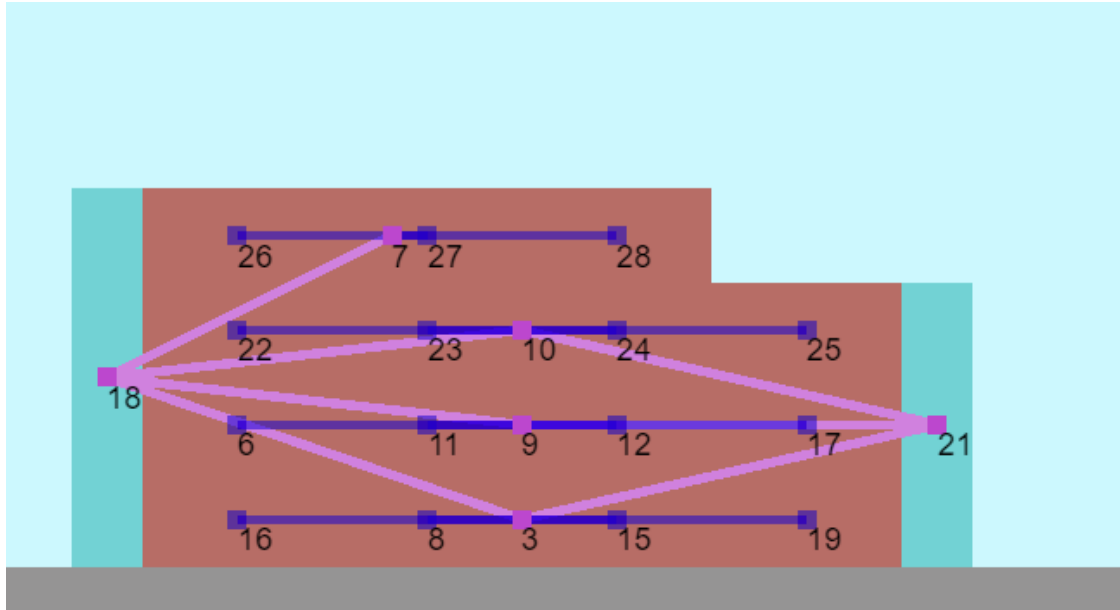
Rooms are represented abstractly as a sub node and will connect to their parent node. Consequently, edges need to be generated between the parent and child nodes, so that a subgraph can be made later on. Figure 13 shows a new topology when rooms are used.



(Figure 13)

The “buildGraphs” algorithm is shown in Appendix C.

As found in the research, reducing the search space reduced the search time (Zubek, 2017). To better illustrate the hierarchy and search space reduction, Figure 14 shows a bigger tower, with many rooms.



(Figure 14)

In this tower, there are 21 nodes, however, the higher-level graph (pink nodes) only features 6. Since only the higher-level graph will be searched, this is a reduction of 3.5x. Usually the bigger the tower, the larger the search space reduction.

4.2 Searching Graph

Now that a graph and subgraph have been generated, they can be searched. As aforementioned, both breadth-first search (BFS) and A* will be implemented to compare the path returned and the speed in which the goal was found.

4.2.1 Implementation of Breadth-First Search

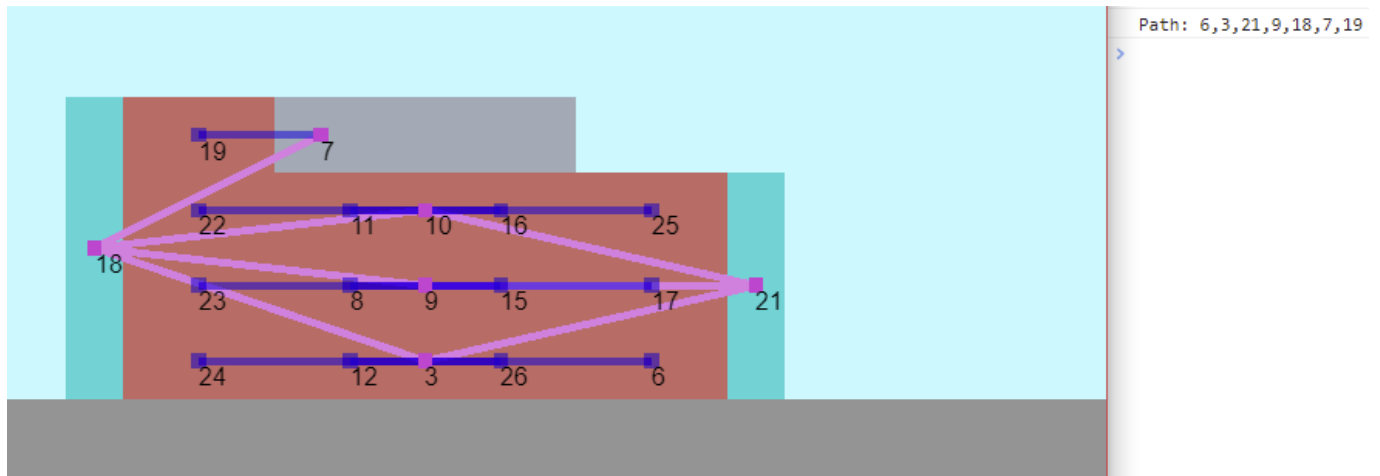
BFS works by brute forcing a path to the goal (or destination). It does this, as it is an uninformed search algorithm. As it is uninformed, there is no need to add additional information to the graph, such as cost.

The algorithm's pseudocode is available many places online. However, different applications require modifications to the algorithm. Because of this a custom algorithm was written based on the requirement to work hierarchically.

In the game, there will be AI agents. They will move between rooms based on their action selection. Because of this, agents will only store what room they are in. However, the algorithm only searches the higher-level graph, which does not feature the rooms. This means that the algorithm must bypass the starting room, instead starting on the parent building node. Similarly, the destination will also be a room, so the parent node should be assumed here as well.

The completed algorithm can be found in Appendix D.

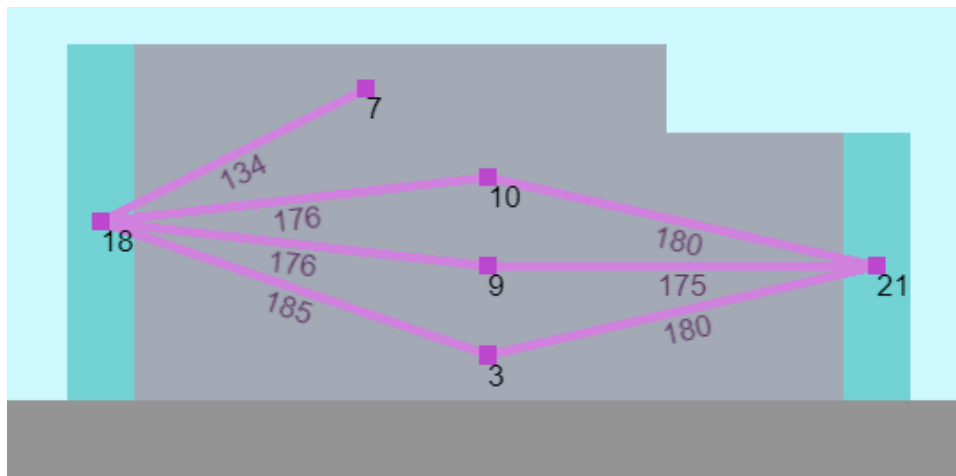
Once BFS algorithm had been implemented, it could be used to find a path between two rooms. As agents have not yet been created, a function call can be manually done through `console.log()`. Figure 15 shows the path between node 6 and node 19.



(Figure 15)

The path is represented as all the nodes that need to be travelled to before the destination is reached. As this is BFS, the shortest route may not necessarily be returned. Figure 15 shows this, as the path returned goes from 3 to 21 to 9 to 18, rather than going directly to node 18.

To remove this problem, cost $g(n)$ can be introduced. The cost of each edge is shown in Figure 16.



(Figure 16)

Cost is calculated as the distance between the two nodes of the edge. There are multiple ways of calculating distance, in this case, Euclidean distance was used.

This additional information needed to be stored in the graph. Allowing for an additional dimension in the array, the cost was added as $(n, 1)$. Figure 17 portrays this change, showing node 3 and its two connections as well as the corresponding cost.

```
▼ 3: Array(2)
  ► 0: (2) ['18', '21']
  ► 1: (2) [185, 180]
```

(Figure 17)

4.2.2 Implementation of A*

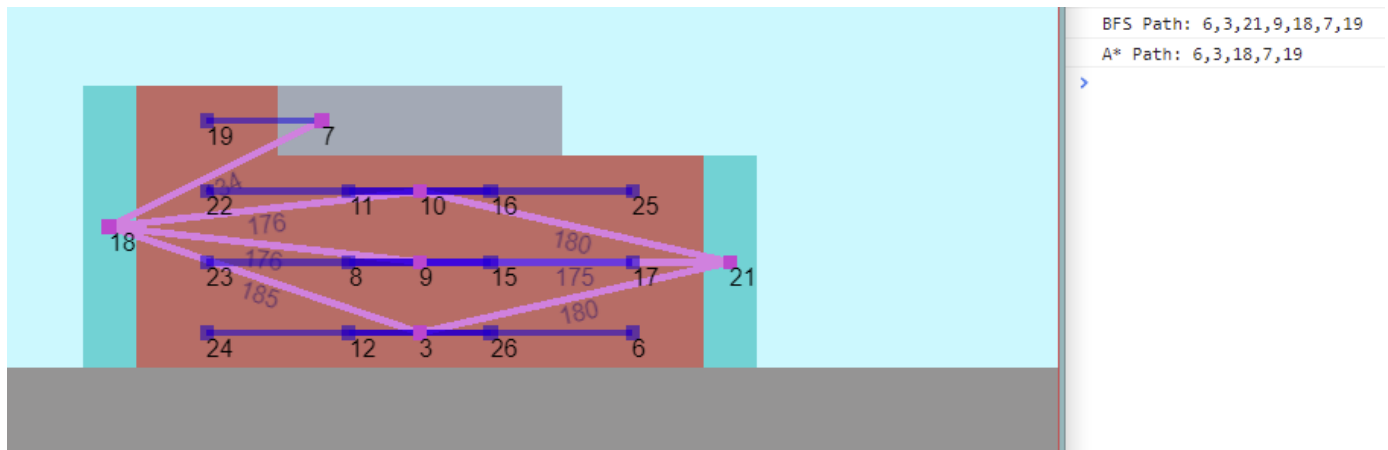
Now that cost had been introduced, other search algorithms can be utilised. For example, Dijkstra's search algorithm uses the addition of cost to find the shortest path. Additionally, more attributes can be used to increase the efficacy of the algorithm. For example, a heuristic can be used, which stores the distance between the current node and the destination. This will give the algorithm an understanding of which node to search next, as closer nodes should be prioritised.

A heuristic $h(n)$ is also a distance, which is usually calculated as the Manhattan distance (see Appendix J for the list of utility functions used).

A* introduces a priority queue, which unlike BFS allows for ordered searching of the nodes. The queue is rank ordered by an fScore $f(n)$, which is calculated as $f(n) = h(n) + g(n)$. In other words, the cost plus the heuristic.

Similar to BFS, there are many versions of A* online. Again, an adapted algorithm was needed to work with the subgraph, as well as the main graph. A* also requires a reconstruct path function, which retraces the shortest path from the destination to the start. This can then be reversed to get a forward travelling path. The final A* algorithm can be found in Appendix E.

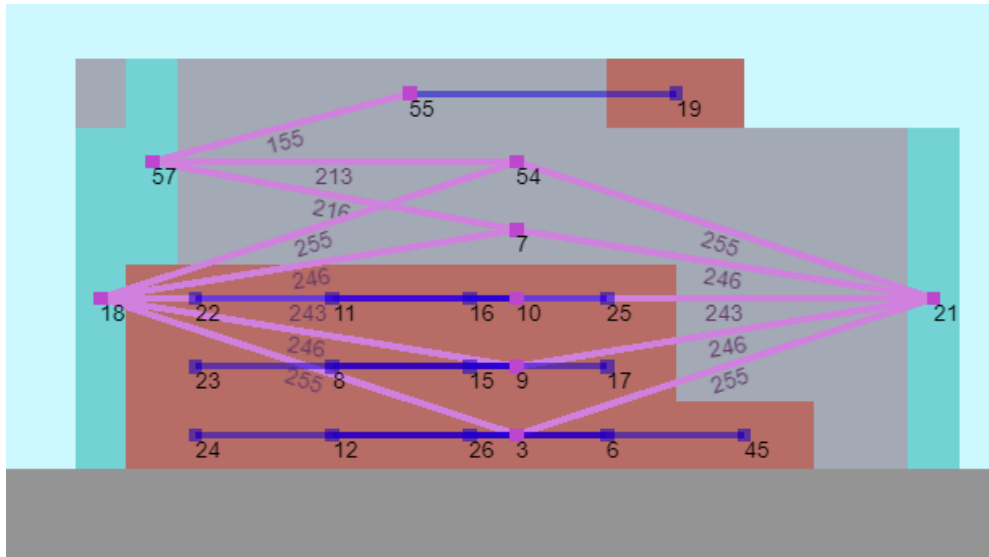
Once implemented, both A* and BFS can be run to show the difference in outcome. Figure 18 represents this difference, as A* returns an alternative path to its counterpart.



(Figure 18)

In this case, A* returns the shorter more sensical path to take.

Upon further testing, it was discovered that on complex graphs, A* would get stuck in an infinite loop, crashing the game. For example, in Figure 19, a complex graph is shown.



(Figure 19)

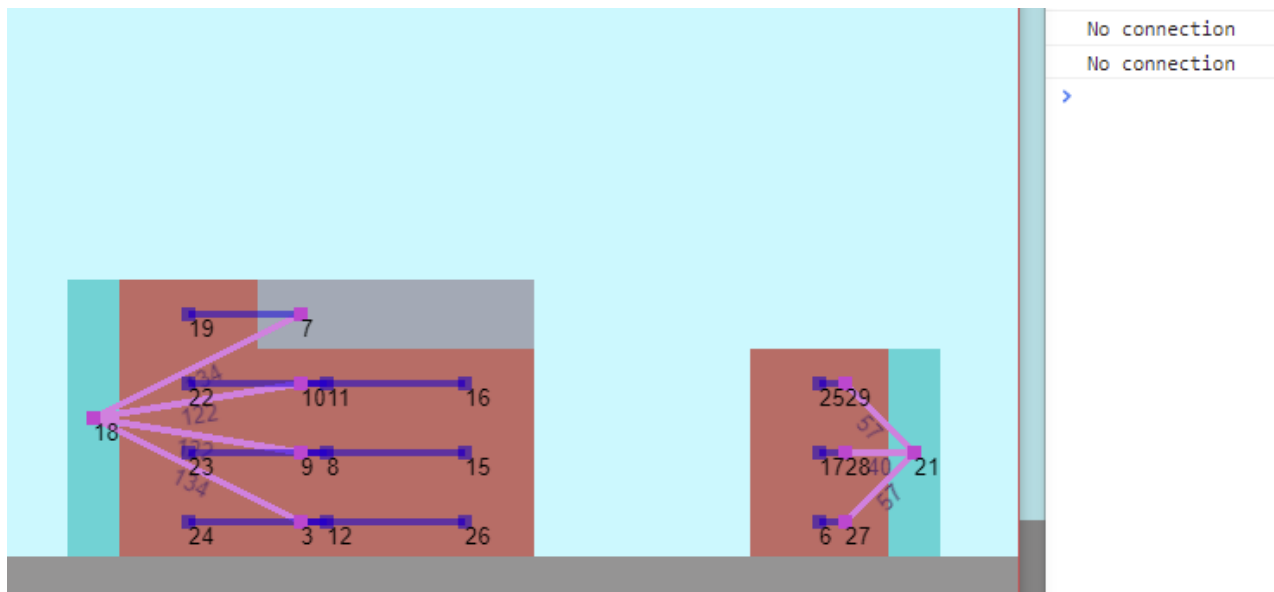
When running A*, no connection was found between node 3 and node 19. This was due to the priority queue being picked not based on nodes from the openSet, rather, just nodes in general. This created a misalignment between what should have been the openSet node with the lowest fScore and just constantly choosing the node with the lowest fScore so far (not necessarily in the openSet). A function called "returnLowestFScore" was created for the solution. This function can also be found in Appendix E, as it is part of the main A* algorithm.

This can be visualised by finding the correct node and comparing it to the node that was picked by the older, broken method. Figure 20 demonstrates this misalignment (item on left is the node being checked; right is the node that should have been checked).

3	3
18	18
18	54
18	7
3	18 21
No connection	

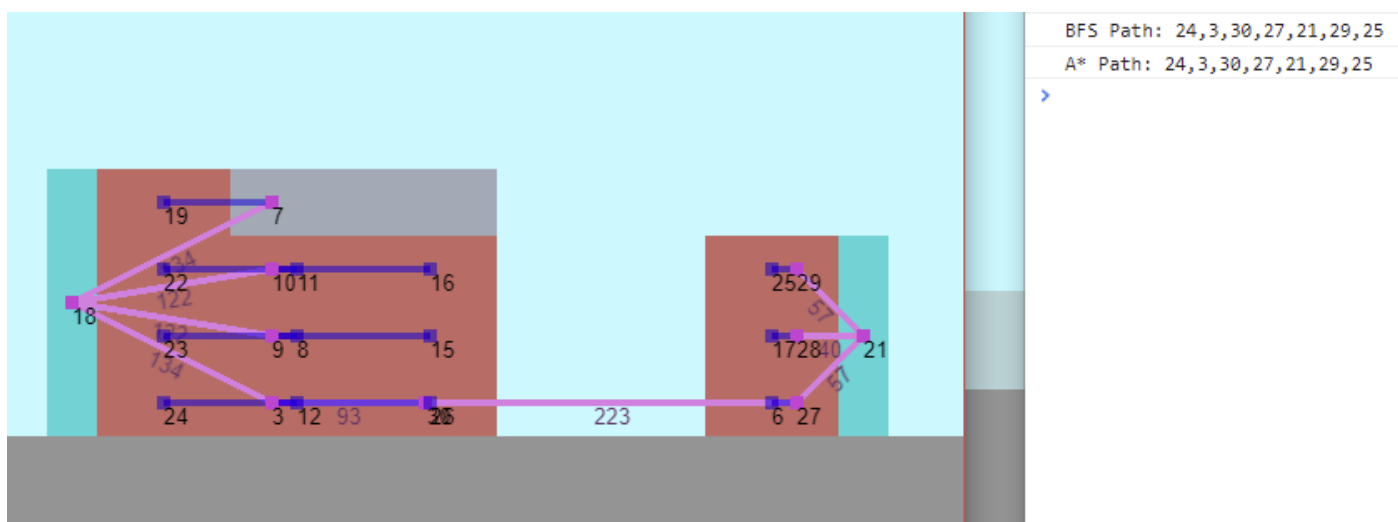
(Figure 20)

Finally, and to expand on a previous section, there was another problem with the graph generation. For example, the start node in Figure 21 is connected to its destination by the ground. However, because there is no connection through the ground floor, no path can be found.



(Figure 21)

To fix this, all buildings on the ground floor should be connected to each other by a ground floor node. Figure 22 demonstrates the change and consequent finding of the paths.



(Figure 22)

4.2.3 Analysis of Algorithms

Now that both algorithms have been fully implemented, they can be compared. A* has two main advantages over BFS: a heuristic and cost. The cost allows for the finding of the shortest path, by comparing the total cost so far to a newer cost during the exploration of the nodes. The lower the cost, the shorter the journey. BFS can sometimes find the shortest path as well, however, there is no guarantee. This was found during the development of the algorithms, where A* was found to return different paths to BFS. It is important to find the shortest path, as the agents should not be taking an unnecessary long route to their destination. This would, in a game environment infuriate the player, as they would be planning a tower to have more lift shafts at busier areas for example, and the agents would be ignoring these routes.

The second advantage A* has over BFS is that it introduces the heuristic. This allows A* to find the destination faster, as it is making estimates of which node should be explored first. This considerably reduces the computing time needed to find the destination. However, because A* is constantly iterating through nodes that it has already searched, in some very rare instances it could be slower than BFS. Greedy best first search fixes this problem by having a heuristic, but no cost, which would likely make it faster than A*. However, as mentioned earlier, it is important to find the shortest path.

A* is also more memory intensive, as multiple new objects are introduced and stored, such as the storing of current costs. This, however, is not a problem with modern hardware.

In summary, A* will be used for the game, as it contains necessary components for the agent simulation.

4.3 Agent Simulation

Now that a path can be returned and A* is established to be the most suitable algorithm, agents can be added to utilise the paths and move between rooms. An agent is a simulated person, who will be able to move left and right. They should be able to ride lifts to their desired floors and make their way to their destination.

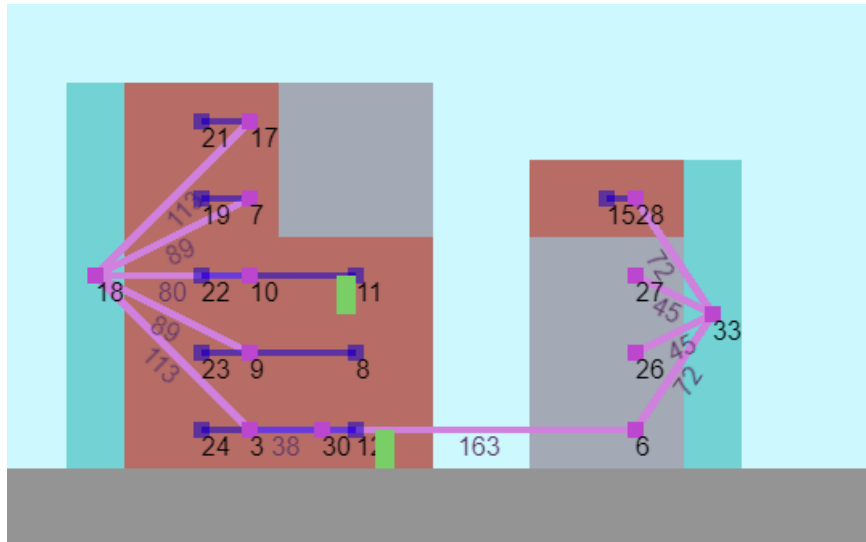
In this case, a custom class was created called “person”. This contained the properties that are instance specific. For example, the agent’s current path (currentPath). Appendix F contains the person class.

To further update the key, Figure 23 shows the new graphics.

Graphic	Meaning
	Person or agent
	Lift

(Figure 23)

To start off, the A* function can be run, and the output stored in the agent’s currentPath property. Figure 24 demonstrates this, as the agent’s current path is shown.



```

PersonInstance {currentRoom: '11', currentPath: Array(9), runtime: IRuntime
  d: 16, ...}
  ▶ currentPath: (9) ['11', '10', '18', '3', '30', '6', '33', '28', '15']
    currentRoom: "11"

```

(Figure 24)

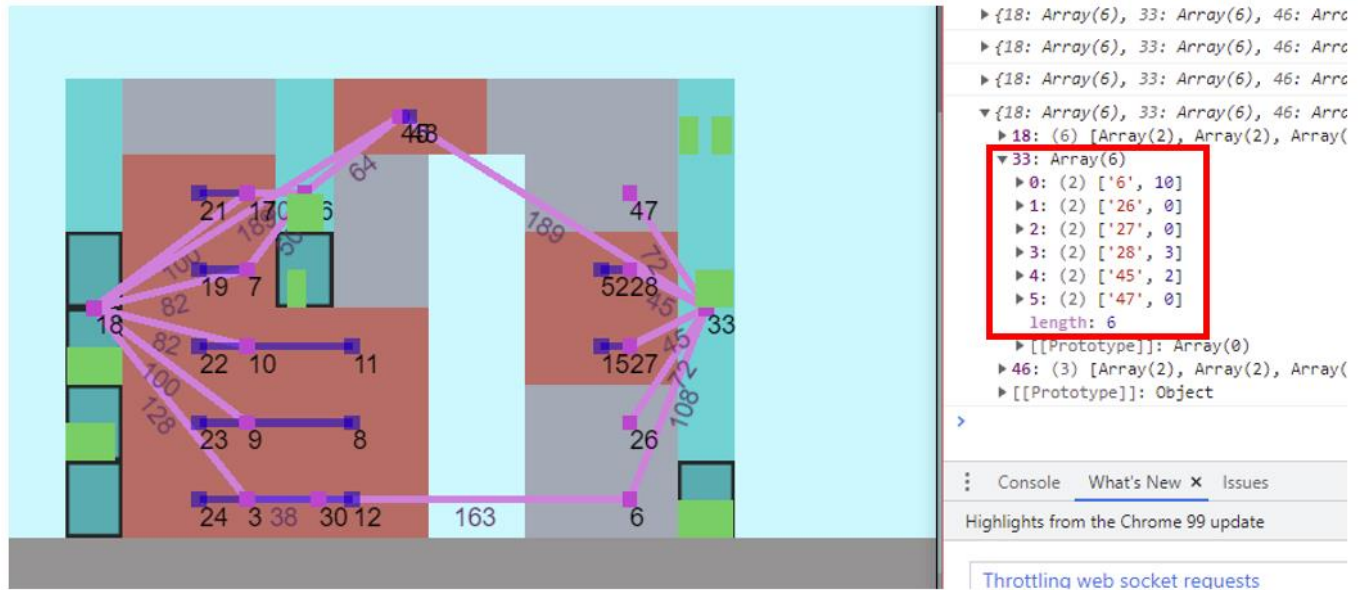
Now the people know where to go, they need to know how to get there. To do this, a lift object needs to be created. The lift will travel vertically on the lift shaft. Here, another class is used to define several properties that the lift itself should store (see Appendix G).

The lift can also store a queue list, which will have the list of floors it needs to visit. When a person enters the lift, they will tell the lift which floor they need to go to. We can assume that the lift is able to go to the desired floor, as the person successfully found a path using that specific lift.

This creates an immediate problem, as when an agent enters a lift, they tell the lift where to go. This means that the lift's queue is not ordered, and the lift will travel to whichever floor was chosen first by an agent. This is dissimilar to lifts in real life, where a lift will go in one direction, until all the desired floors have been visited, then change direction.

To call the lift, the current floor an agent is on needs to be added to the lift's queue. This, however, is also problematic, as suppose person A wants to go from floor 1 to floor 2 and person B also wants to go from floor 1 to floor 2, the lift's queue will end up looking like this: [1, 2, 1, 2]. Which means the lift will go back and forth, even though person B will have already been picked up, when the lift arrived for person A.

The solution was to store how many people were on each floor in another array, stored in a lift shaft object, rather than the lift (see Appendix H for global objects code). This was so the queue was not localised to each lift, as a lift shaft may have multiple lifts on it. Figure 25 shows the new queue (highlighted in red), with the corresponding amount of people waiting. For example, on node 6, ten people are waiting.



(Figure 25)

The lift shaft is then responsible for sending lifts to the floors. This allows for the introduction of lift capacity, as floor 6 has ten people, if the lift's capacity is five, then the lift shaft should send two lifts.

However, the lift itself still needed a localised queue, so it new which floors to visit. Instead of storing a dynamic list of places to explore, a static binary queue could be utilised. This is shown in Figure 26.

```
▼ queue: Array(9)
  ► 0: (2) ['3', 1]
  ► 1: (2) ['10', 1]
  ► 2: (2) ['9', 1]
  ► 3: (2) ['7', 0]
  ► 4: (2) ['17', 0]
  ► 5: (2) ['45', 1]
  ► 6: (2) ['34', 1]
  ► 7: (2) ['46', 1]
  ► 8: (2) ['51', 0]
  length: 9
```

(Figure 26)

For example, nodes 3, 10, 9, 45, 34 and 46 should be visited. This list can then be iterated from top to bottom and vice versa when it hits the last node which needs to be visited. Of course, the list should be ordered by the y axis. This contrasts the older system, where a dynamic queue was held, which would have nodes added and deleted to. This old queue can be seen in Figure 27.

```
► queue: (4) ['7', '45', '17', '45']
```

(Figure 27)

Once this was done, capacity was added. If a lift visits a floor but is at capacity and more people want to get on, then the lift will remember that floor as a missed floor and return later (see the “moveLifts” function in Appendix I).

For the agents to move in the first place, they use linear interpolation, also informally known as tweening. A person will loop through their list of nodes to get to their destination and depending on the type of node, perform a certain action. For example, if the node is a room or a lift shaft, then the person will move to that instance's x axis. Otherwise, the person will ignore the node, removing it from their queue. They will repeat this process until all nodes have been iterated, then the last node in the list must be the destination. This is demonstrated with pseudocode in Figure 38 below.

```
While person's list is NOT empty,  
    If the next node is the destination,  
        They have arrived.  
    Otherwise,  
        If the node is a lift shaft, move to it and leave the while loop.  
        Remove node from list.
```

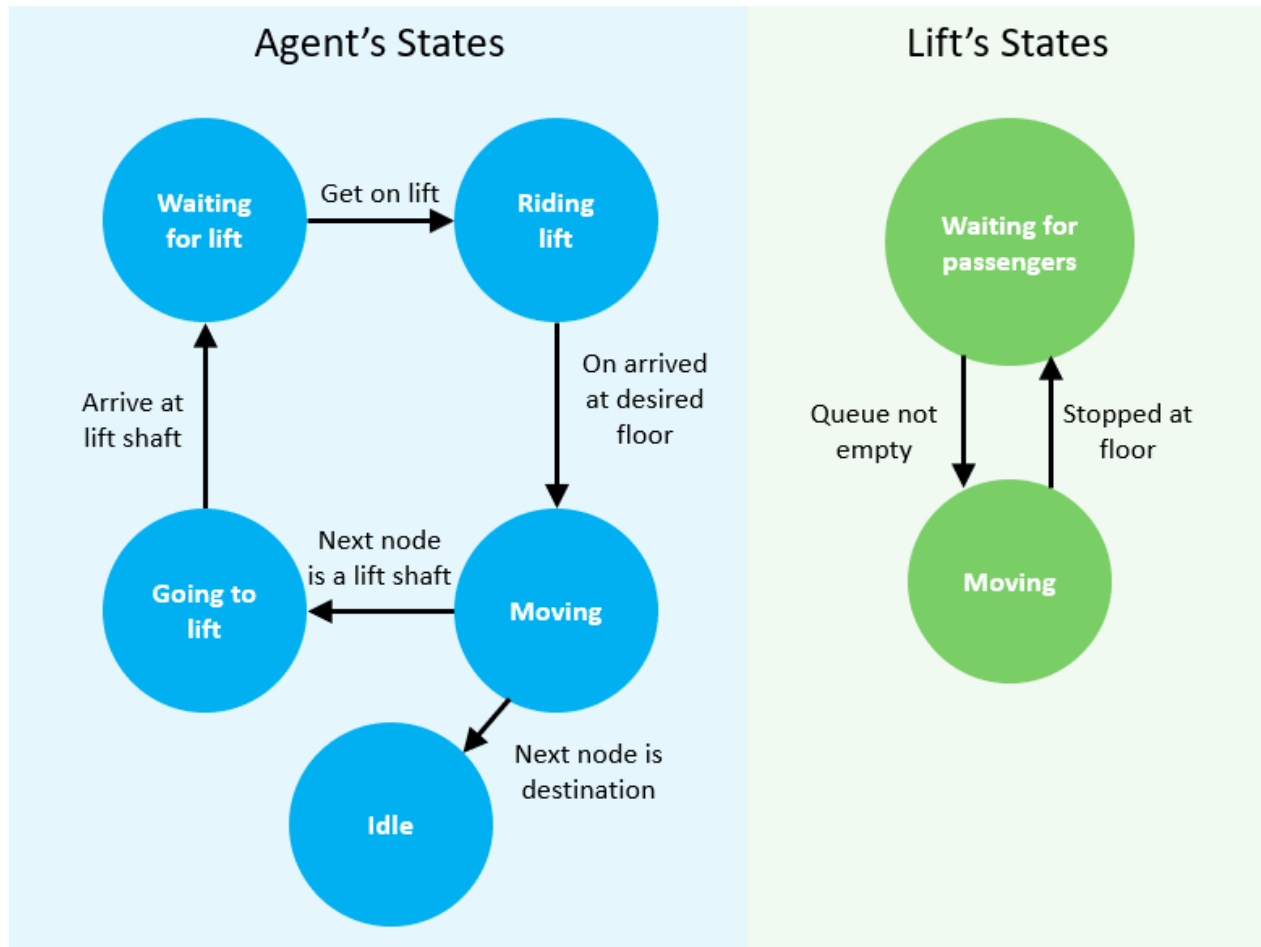
(Figure 38)

If the next node is a lift shaft, then a different function takes over. This checks if they have arrived at the lift shaft, and if so, to take a lift. Once arrived at the desired floor, then the function from Figure 38 is run again. This will happen in a cycle until the destination is reached.

To make this possible, states were added. These are an added property to both the lift and the person. A state represents the current action of an instance. For example, a person may be "moving" or "waiting for lift" etc. This makes it possible for more complex behaviour. For instance, once a person is on the lift, they change state to "riding lift". If a person is in this state, then every tick, they will set their position relative to the lift (ie. moving with the lift).

These states in effect create a state machine. This, however, does not lead to agent autonomy, which is discussed in section 5.0 "Implementation of Action Selection". The state machine is shown in Figure 28. Possible states are connoted by circles and the transition between the states is shown by an arrow.

All of the code for agent simulation can be found in Appendix I.



(Figure 28)

4.4 Summary

In summary, a graph is generated from the object instances in the layout. A graph, being a mathematical abstraction, contains nodes and edges. This graph is then used to find a path between two of the nodes.

With a basic state machine, agents can move along their paths, as stored individually in memory. They do not yet move autonomously and require a debug user input to move.

5.0 Implementation of Action Selection

5.1 Introduction to Action Selection

Now that the people are able to move between two points, they should have some level of autonomy so that they can choose where to move to themselves. To do this, action selection can be introduced to the game. This was the secondary aim of this project.

As introduced during the research, a finite state machine would be suitable for this task. This state machine will extend what was shown in Figure 28, by adding new states and transitions. Similar to what was found in the paper by Zubek (2017), a daily schedule can be implemented, which will allow agents to transition states based on the time of day.

5.2 Implementation

This can be hard coded as an object embedded in other objects. To do this, agent types were introduced. For example, an agent can be a worker or a resident. Residents live in the tower, in flats. They will leave the tower frequently and make trips to the local food court within the tower (if one exists). On the other hand, workers will spend most of their time working in their workplace, taking lunch and bathroom breaks. In this game, residents cannot work in the tower, and workers cannot live in the tower (apart from to go home at the end of the day).

Each agent of a different type will have a separate schedule, which is generalised to all people of that particular type.

For example, Figure 29 shows the schedule that a worker would have in a day (also found in Appendix H).

```
schedule: {  
  "worker": {  
    // morning  
    480: ["out", 0.5, "go to work"],  
    540: ["out", 1, "go to work"],  
    // midday  
    660: ["working", 0.2, "go to bathroom"],  
    720: ["working", 0.5, "go to eat"],  
    780: ["working", 0.5, "go to eat"],  
    800: ["eating", 0.3, "go to bathroom"],  
    810: ["working", 0.3, "go to eat"],  
    830: ["eating", 0.3, "go to bathroom"],  
    // evening  
    960: ["working", 0.3, "go to bathroom"],  
    1020: ["working", 0.5, "go home"],  
    1050: ["working", 0.5, "go home"],  
    1080: ["working", 0.8, "go home"],  
    1110: ["working", 1, "go home"]  
  },  
}
```

(Figure 29)

As aforementioned, the agents will change state based on the time of day. The simulation runs on a 24-hour clock which is stored in minutes. These become the name of the objects. For example, 480 is 8:00 AM. This allows for minute precision in the scheduling of tasks.

Each object stores a 1-dimensional array, or list, which stores three elements. The first of which is the current state of the agent. This is a prerequisite requirement for the agent to change state. For example, only an agent with the current state “out” can transition on the first task.

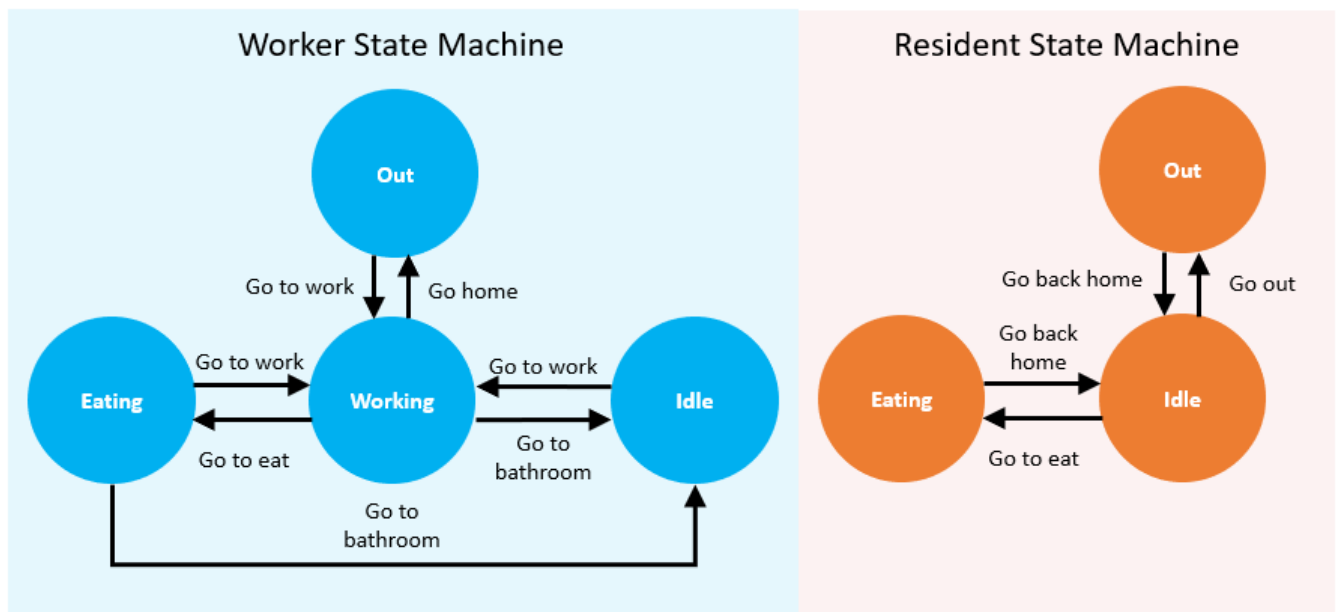
The next element is the chance of transition happening. For example, 0.5 is a 50% chance of changing state. Finally, the last element refers to the transitional action or task to go between states. This begins with the verb “go”. For example, “go to work” is a task, which when checked, will make the agent go to work.

The task is not a state in itself, as the agent will not change their state to this. The task is only used as an argument to tell the simulation what the agent wants to do. Here, the task is queried to see what it is and then a specific action takes place. For example, if the task is “go to eat”, then the agent will find the closest food court and path find to that room. Once they have arrived, they change their state to “eating” (see Appendix K for this function). However, to better imitate human behaviour, they sometimes do not find the closest room, opting to sometimes go to another room instead (see Appendix L).

Certain tasks, however, need to run systematically every hour. This is accomplished by running certain events if the time is a multiple of 60 (as it would be at the start of the hour). For example, “go back home” is run every hour if a resident is eating (see Appendix M).

Of course, for this to work, time needs to be implemented. This was done through waiting a set interval and then running a function, which incremented the time (in minutes) by one. When the time hit 1440 (24 hours) it would reset to zero. Every time the time is incremented, the global schedule is checked to see if anyone has tasks that need to be run (see Appendix N).





This can be conceptualised as a state machine. We can think of the previous states from Figure 28 as transitional (except for “idle”). For example, “moving” is not a permanent state, but rather leads to another state. Figure 30 demonstrates a state machine without these transitional states, where the arrows represent changing first to the “moving” state, and then the consequent state. For example, the state “eating” transitions to “working” through the “go to work” task. Which requires the agent to first change their state to “moving”, and the subsequent cycle as shown in Figure 28. On arrival, the agent changes to the appropriate state (in this case “working”).



(Figure 30)

5.3 New Graphics

To make the buildings clearer, the graphics have been outlined. A new key featuring the added rooms is shown in Figure 31.

Graphic	Meaning
	Office – where worker agents go to work.
	Flat – where resident agents live.
	Bathroom
	Food court

(Figure 31)

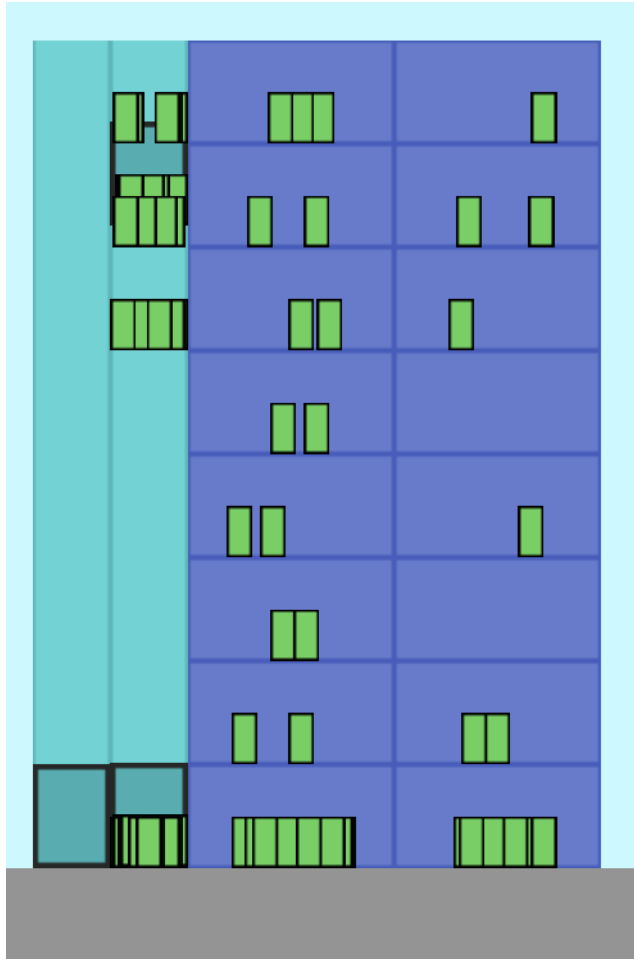
5.4 Alternatives

The advantage of the current system is that it is light weight and customisable. It is light weight, as it requires less memory than a localised system, where each agent has their own schedule. It is customisable, as the schedule can be modified, where new events can be added. The system also has a built-in probability, which allows some events to run less frequently, preventing all of the agents doing the same task at the same time.

An alternative would be to have a needs-based system. Where each person would store a list of properties called needs. These would be things like hunger, happiness etc. And would dictate where the people would go. For example, if a person was low in hunger, then they would go to a food court. As mentioned before, this would not be as light weight as the current system, as each person would need to be constantly updating their needs. This system may make the people more personable and realistic, however, it is unnecessarily complex for this game.

5.5 New Problem with Lifts

Now that the agents have autonomy and are moving around the building, a new problem has been found. The problem is due to the use of A*, as the algorithm does not consider the occupancy of different rooms. For example, Figure 32 shows how no agents are taking the lift shaft on the left.



(Figure 32)

The fix was to increase the $g(n)$ cost of travelling to a lift shaft node by the amount of agents currently waiting for a lift. This amount is calculated by incrementing when people decide which lift to path find to; decrementing when people get off.

This would in effect, make travelling on a crowded lift more costly and as such, agents would avoid these lifts. Of course, this does not make the lift shaft on the left in Figure 32 necessarily more appealing, rather it allows agents to avoid busy lifts, as they increase the travel time to their respective destinations. A potential downside to this solution is that agents choose their path at the start, not altering it on the way. So, by the time they arrive at a lift shaft, it could be fuller than expected, consequently leading them on a slower route. However, this is avoided by having an independent array recording how many agents are on the lift, which is modified as agents find paths, rather than when they get on the lift. For example, person A may choose lift shaft 1 in their route, this increments the “expected load” variable. Then, person B can instantly see the change and fullness of the lift shaft before the person arrives at the lift.

Figure 33 demonstrates the change and consequent fix.



(Figure 33)

Notice also that there can be multiple lifts on a singular lift shaft. This was intentional, as this is a common feature in this game genre.

5.6 Summary

In summary, the action selection problem was overcome, and a working state machine has been implemented for the agents.

The agents can now autonomously choose where to go to, as they now have artificial intelligence.

6.0 Implementation of User Interaction

After completing the primary and secondary problems. A few quality-of-life features can be implemented. One of the aims was to have a level of user interactivity with the game. For example, the user should be able to create new rooms in the tower during runtime. This will ensure that another objective is met; the ability to expand the tower.

6.1 Prerequisite

Before the ability to modify the tower is added, time control needs to be introduced. This involves the ability to change the in-game time, from paused to normal speed and then a fast speed. This is so when the user is building, the time can be stopped.

To do this, the engine features a built-in time scale property. This is defaulted to 1. If the time scale is changed to 0, then the game pauses; 2, then the game runs twice as fast.

So the user can control time, some buttons were created. An event listener can then check for mouse input and check if the mouse overlaps any of these buttons. If so, then the consequent time can be changed. Some `setInterval()` functions are disabled, such as the time changing every second, and then re-enabled when the game is played again (see Appendix P for mouse control code).

The UI (user interface) elements are shown in Figure 34.



(Figure 34)

6.2 Building

Originally, the building becomes customisable once the game was paused. However, a separate build button was added later instead.

There were many ways to implement a building system. Traditionally, when building, the people in the building stay where they are. However, when this method was used, the people would often get lost, as nodes that they were using during pathfinding were deleted. A solution would be to update the pathfinding every time the game is resumed after building. Another solution would be to evacuate the building entirely, changing the agent's state to "idle", and then allowing them to return once the game resumes. This was the solution that was used, as it minimised the number of agents who became lost and stopped moving entirely.

To have this work, an evacuation function was used (see Appendix Q for code), which teleports all the agents to a spawn location (of the screen). Once resumed, the agents path find to their primary room, either a flat or an office (see Appendix R for code). This solution is not perfect, as they stop what they are doing and effectively reset. However, it is fully working and again, lacks the "lost" state problem.

When the user is building, a new UI is shown. This can be seen in Figure 35



(Figure 35)

Clicking on one of the buttons will create an outline where the building will be placed. Left-clicking will place the building, right-clicking will cancel the placement. Already existing buildings can be moved around or removed by right-clicking.

There are several systems to make this work. For example, the outline needs to be the same size as the potential room, and when the room is placed, it needs to be on top of other instances.

Certain rooms can be adjusted once placed. For example, the building instance can be stretched horizontally, by using the scroll wheel. This is similar to how the lift shaft will stretch vertically with the scroll wheel. The code for both the placing and expanding the tower can be found in Appendix S).

Some rooms need to be placed under specific conditions, such as a flat must be placed in a building. If not, then that respective room will be removed once the game is resumed.

6.3 Graph Generating

Once the user has changed the building, the program regenerates the graph. This can be done by calling the same function used at the start. However, global variables must be reset. Despite originally, some variables were missed and not reset, leading to a few problems later on. For example, the lift's queue was not reset which led to it getting longer with repeats, as shown in Figure 36.

```
▶0: (2) ['7', 0]
▶1: (2) ['7', 0]
▶2: (2) ['6', 0]
▶3: (2) ['6', 0]
▶4: (2) ['9', 0]
▶5: (2) ['9', 0]
▶6: (2) ['3', 1]
▶7: (2) ['3', 1]
▶8: (2) ['11', 0]
▶9: (2) ['11', 0]
▶10: (2) ['73', 0]
▶11: (2) ['73', 0]
length: 12
```

(Figure 36)

You can see that each number is repeated, which meant lifts would stop twice as long at each floor.

This was fixed by resetting the individual lift queue, as each lift has its own queue. Once this was resolved, the generation could be done without error, and buildings could be changed at runtime.

6.4 Summary

In summary, the tower can now be modified through user interactive buttons which are rendered in a fixed position to the screen.

On modification, the game will regenerate the graph as developed in section 4.0.

7.0 Evaluation & Testing

The artefact has now been completed, as all the objectives have been achieved. As mentioned in the introduction to the report, there were functional and non-functional requirements.

The objectives were largely listed in the order that they were pursued. This was to demonstrate the idea that the previous task was a prerequisite of the current task. For example, the objective to create a “Pathfinding Graph” is a prerequisite of the subsequent task of implementing a “Pathfinding Navigation Script”.

The initial objectives changed very little, with the biggest change being implementing 2 search algorithms as opposed to 1. This was to create an environment for analysis.

Part of evaluating an artefact is testing. During the development of the artefact, white box testing was carried out frequently. Every time a problem arose, a solution was put forward. Now that the artefact is complete, final black box tests can be conducted. This is to ensure that the end product is working as intended, and meets the criteria set out in the introduction.

7.1 Functional Requirements Test Plan

A test plan created from the functional requirements can be found in the methodology section and is used below on the artefact.

Testing	Description	Expected Result	Actual Result	Pass/Fail
Graph generation (main graph)	Run the project after creating a tower in the engine editor with no rooms	A graph is generated	A graph is generated	Pass
Graph generation (subgraph)	Run the project after creating a tower with rooms in the engine editor	A graph and subgraph are generated	A graph and subgraph are generated	Pass
A* pathfinding	Set a start and destination on the same floor	A path is found between the nodes	A path is found between the nodes	Pass
A* pathfinding	Set a start and destination on different floors, connected by a lift	A path is found between the nodes	A path is found between the nodes	Pass
A* pathfinding	Set a start and destination, where they are not connected	No path should be found (return no connection)	No path is found	Pass
Agent simulation	Set a start and destination on an agent, so they can move to their destination	Agent should move to their destination	Agent moves to their destination	Pass
Agent simulation	Make an agent move between floors to test the lift movement	Lift should move to the appropriate floor	Lift moved to the appropriate floor	Pass
Agent simulation	Make an agent move between floors to test lift queue	Lift should pick up agent and add them to the queue	Lift picks up agent and adds them to the queue	Pass
Agent simulation	Make an agent move between floors to test lift states	Lift should drop off passengers, changing state to "waiting for passengers"	Lift drops off passengers, changing state to "waiting for passengers"	Pass
Action selection	Wait until 8:00 am to see if agents arrive for work	Agents arrive at work	Agents arrive at work	Pass

Action selection	Wait until 12:00 pm to see if agents go to lunch (only workers)	Worker agents go to lunch	Worker agents go to lunch	Pass
Action selection	Wait until 5:00 pm to see if agents go home	Worker agents leave the tower (go home)	Worker agents leave the tower (go home)	Pass
User interaction	Press the pause button	Game should pause (freeze time)	Game pauses	Pass
User interaction	Press the play (1x speed) button	Game goes at 1x speed	Game goes at 1x speed	Pass
User interaction	Press play (5x speed) button	Game goes at 5x speed	Game goes at 5x speed	Pass
User interaction	Press build button	Game pauses and build menu appears	Game pauses and build menu appears	Pass
User interaction	Click on room or building button in the build menu	Game creates an outline of the new room	Game creates an outline of the new room	Pass
User interaction	Click after the outline is showing (for room)	A room is placed where the cursor is	A room is placed where the cursor is	Pass
User interaction	Click after the outline is showing (for building)	A building is placed where the cursor is	A building is placed where the cursor is	Pass

7.2 Performance Tests

The artefact is working, and all the functional requirements have been maintained. There were also non-functional requirements to fulfil, most of which are not quantifiable. A table shown in the methodology section can be used to measure the performance of the game, which was part of the non-functional requirement that it “runs well”.

Description	Desired result	Actual Result	Pass/Fail
Run the game with 5 offices, containing 10 agents each. (50 agents in total)	The game runs with dropping frames and without a CPU utilisation above 10%.	Game runs at about 1.5% CPU utilisation, 3% when agents are moving	Pass
Run the game with 10 offices, containing 10 agents each. (100 agents in total)	The game runs with dropping frames and without a CPU utilisation above 10%.	Game runs at about 1.5% CPU utilisation, 3% when agents are moving	Pass
Run the game with 20 offices, containing 10 agents each. (200 agents in total)	The game runs with dropping frames and without a CPU utilisation above 10%.	Game runs at about 1.5% CPU utilisation, 3.8% when agents are moving	Pass
Run the game with 50 offices, containing 10 agents each. (500 agents in total)	The game runs with dropping frames and without a CPU utilisation above 10%.	Game runs at about 2% CPU utilisation, 5% when agents are moving	Pass

Surprisingly, the game ran better than expected, being able to run more efficiently than first thought. This is due to several optimisations. For example, when an agent is waiting for a lift, they stop performing any

checks that run every tick. These checks consume CPU time and so avoiding them increased the performance. Because of the performance, the minimum running requirements for the artefact should be no higher than the requirements to run the engine. These can be found at Construct (www.construct.net, n.d.).

During the development of the artefact, it was important to keep this in mind, as to avoid any CPU intensive operations. As stated in the introduction, this game was inspired by other games that are able to simulate 100s if not 1000s of agents. This game, whilst simplistic, has been able to achieve a similar feat.

8.0 Reflection

8.1 Introduction

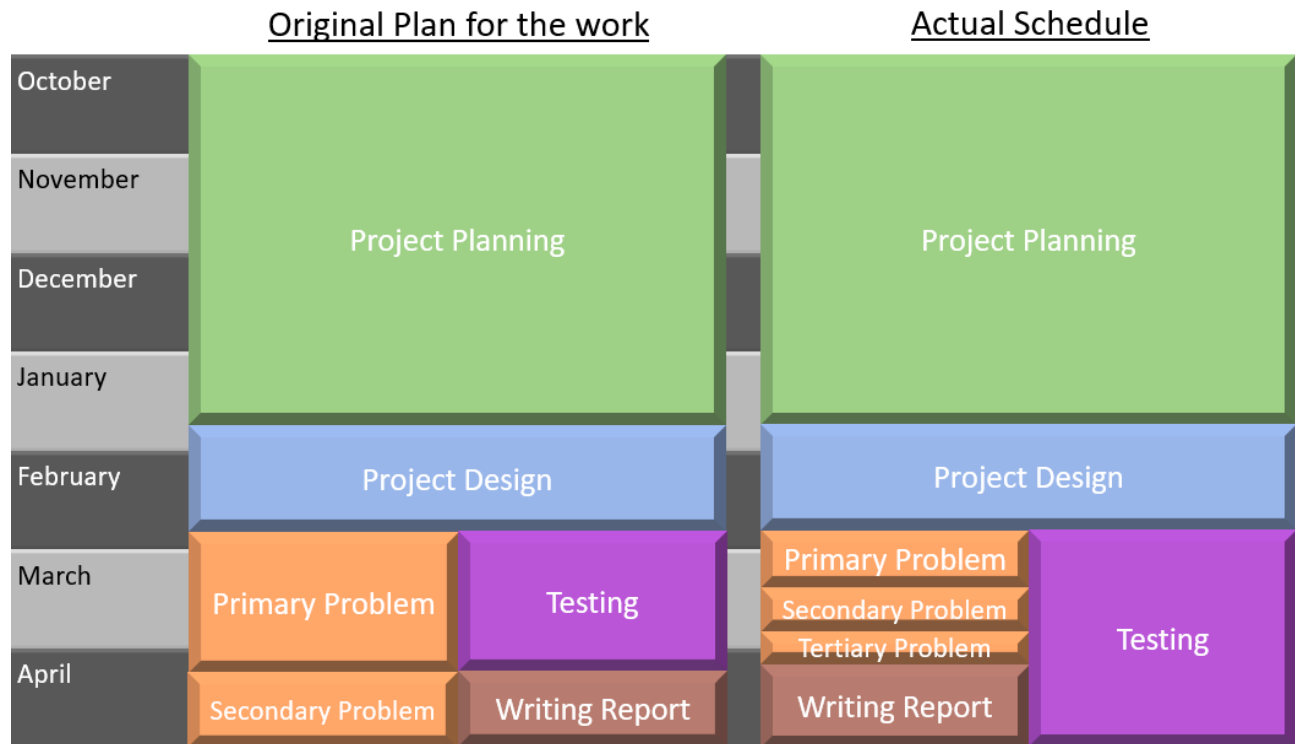
The project has been worked on since September 2021 (last year), with the initial project planning phase taking place first. Here, a proposal was put forward and some research was conducted into the topic. It was here when the aims and objectives were chosen. These guided the project, making sure that the artefact was relevant to the problems needed to be solved.

The initial aims and objectives were very similar to how they ended up in the final project, with only the introduction of a new pathfinding algorithm and slight modifications to the user interaction being added. However, at first, the aims and objectives were vague at best and should have been defined more clearly.

Very early on, the primary problem was split into 2 sections: the generating and searching of the graph. This was to break up the research into more appropriate topics, as they can, and often are separated in literature.

8.2 Initial Plan

To plan the development of the project, a Gantt chart was created. This was done to estimate the time needed to complete each task. On later analysis, the original plan differed from what actually happened. Figure 37 displays the original plan, from the Gantt chart, with what happened on the right.



(Figure 37)

With prior experience working on games, the estimated time in the Gantt chart was realistic, and did not cause any problems with having to rush the work at the end.

Figure 37 only displays the days worked on the project, rather than the hours spent each day. Often, the artefact was worked on for long periods of time, causing unnecessary stress, but allowing bugs to be quickly quashed. In retrospect, working fewer hours over more days may have made for a better workflow.

When doing the project, an additional tertiary problem was worked on. This was outlined as an extension in the original proposal. This was the ability for the user to build and modify the tower during runtime. Working on this was possible, as progress had been swift on the initial problems.

The writing of the report happened after the artefact was completed. This was different from the initial plan, but because the artefact had been created earlier than expected, doing the writing afterwards made more sense. In retrospect, this was only possible due to comprehensive notes being taken during development. If notes were not made, the report would have had to be written during the development, so important ideas and issues were not forgotten.

8.3 What went wrong

Even though the artefact fulfils all the requirements, it has not been stress tested. An intervention to prevent any problems last minute was to introduce the “lost” state to the agents. If an agent was lost, then they could be reset, reducing the chance of serious bugs whilst pushing the game to its limits. This was originally added because of myriad problems which arose from an earlier prototype which tested the ability for a user to change the building without first having the agents evacuated. Agents would often

become lost, as their current route was interrupted or destroyed. The intervention here, was to remove the system altogether, as discussed in the “Implementation of User Interaction” section.

This removal did not affect the requirements of the artefact, so these were not updated. For future reference, adding more specific specifications to the requirements would make the plan clearer, as it would prevent a disorganised scope.

Scope creep is the idea that as a project develops, its objectives increase in quantity. This became a slight issue about halfway through the artefact. This is often due to vague objectives, too open to interpretation. For example, the complexity of the user interface was never discussed in the initial objectives. At first, a simple UI sufficed, but as time progressed, the thought of adding comprehensive drop-down menus and categorising the building options became potential new objectives. Preventing this involved reclassifying the objectives and limiting scope where possible.

8.4 What went right

The artefact has been completed and a report has been written within the time given. Time management and planning were at the forefront of progress on the artefact, allowing quick development ahead of the expected time needed on each task. As mentioned above, prior experience on game development was key to some of this success, as it allowed for more accurate and realistic planning and workflow.

The primary problem was the main experiment during the project, with the additional problems being explored as there was enough time. These additional problems extended the scope and complexity of the artefact, leading to a less generalised artefact. Working on these problems led to more knowledge and understanding being gained on the topic.

8.5 Goals

Before starting, there were several personal goals for the project. These were outlined in the 1.0 introduction section.

The main goals surrounded the idea of increasing proficiency in something. In this case, JavaScript and the Construct game engine. These are immeasurable, but likely achieved during the project, due to the necessity to use these tools during development.

Another goal was to increase understanding in artificial intelligence in games. This was achieved primarily through the literature review, as well as exploring the systems used whilst implementing. For example, a choice was made for the action selection, as to choose between using a needs-based system or a time schedule for the agents. A thorough understanding of AI was needed to be able to make this choice, and to accurately predict the performance increase in the outcome.

8.6 Summary

Overall, and to reflect, the project was a success. This is both in terms of requirements, objectives, and personal goals. Proper pacing and a plan allowed the project to be conducted with minimal problems along the way. A flexible approach also meant the changing of objectives could be done early on, to better reflect the artefact.

9.0 Conclusion & Future Work

9.1 Conclusion

In conclusion, the artefact has been completed and all the initial aims, objectives, functional and non-functional requirements have been fulfilled.

The initial and primary problem, to create “A Hierarchical Waypoint Pathfinding Solution in a 2D Tower Simulation Game” has been solved. The artefact creates a hierarchical waypoint graph which is searched using the A* search algorithm, as was found to be most effective. This is done in the context of a 2D tower simulation game, where the agents are simulated to move around the tower.

Additionally, agent autonomy, through a finite state machine was used to allow agents to move without user interaction. This was completed as a secondary problem, which added to the primary problem, expanding on the premise.

To round off the game, user interaction was required. Allowing the user to modify the existing building is paramount in these sorts of games, and so became a necessity early on. This was implemented during the final stages of development, as before then debug commands were used during tests.

The game is playable, but features little in-game objectives, such as earning money and spending that on new rooms. Therefore, the term “simulation” is more appropriate at this stage.

9.2 Future Development

For future development, the game can be expanded. Additional user interactions can be added to support more room types. A tweak of the agent behaviour could also be added to allow agents to be both residents and workers, letting them work and live in the tower.

Further down the line, in-room actions can be supported, which would make agents perform actions depending on the rooms they are in. For example, an agent at a restaurant could sit at a table or wait to be seated. This would also require more detailed art, which would be something needed in the future as well.

At the moment, development will cease and can be picked up at a later date, as the code is well commented, and the work is documented in this report.

10.0 References

Bryson, J.J., 2003. Action selection and individuation in agent-based modelling. In Proceedings of agent (pp. 317-330).

Construct.net. (2021). Game Making Software - Construct 3 [online] Available at: <https://www.construct.net/en>.

Construct.net. (2021). System requirements - Construct 3 Documentation ★★★★★. [online] Available at: <https://www.construct.net/en/make-games/manuals/construct-3/getting-started/system-requirements>.

Cui, X. and Shi, H., 2011. A*-based pathfinding in modern computer games. International Journal of Computer Science and Network Security, 11(1), pp.125-130.

Dawe, M., Gargolinski, S., Dicken, L., Humphreys, T. and Mark, D. (n.d.). 4 Behavior Selection Algorithms An Overview. [online] Available at:

http://www.gameapro.com/GameAIPro/GameAIPro_Chapter04_Behavior_Selection_Algorithms.pdf

Game Development Envato Tuts+. (n.d.). Goal Oriented Action Planning for a Smarter AI. [online]

Available at: <https://gamedevelopment.tutsplus.com/tutorials/goal-oriented-action-planning-for-a-smarter-ai--cms-20793>.

GamesIndustry.biz. (n.d.). What is the best game engine: is Unity right for you? [online] Available at:

<https://www.gamesindustry.biz/articles/2020-01-16-what-is-the-best-game-engine-is-unity-the-right-game-engine-for-you>.

Godot Engine (2021). Godot Engine - Free and open source 2D and 3D game engine. [online]

Godotengine.org. Available at: <https://godotengine.org/>.

Harabor, D.D. and Grastien, A., 2012, July. The JPS Pathfinding System. In SOCS.

Hughes, O. (n.d.). Top programming languages: Most popular and fastest growing choices for developers. [online] ZDNet. Available at:

<https://www.zdnet.com/article/top-programming-languages-most-popular-and-fastest-growing-choices-for-developers/>

Jacopin, É. (2015). Optimizing Practical Planning for Game AI. [online] Available at:

http://www.gameapro.com/GameAIPro2/GameAIPro2_Chapter13_Optimizing_Practical_Planning_for_Game_AI.pdf

Krishnaswamy, N. (2009). Comparison of Efficiency in Pathfinding Algorithms in Game Comparison of Efficiency in Pathfinding Algorithms in Game Development. [online] Available at:

<https://via.library.depaul.edu/cgi/viewcontent.cgi?referer=https://www.google.com/&httpsredir=1&article=1010&context=tr>

Noto, M. and Sato, H., 2000, October. A method for the shortest path search by extended Dijkstra algorithm. In Smc 2000 conference proceedings. 2000 ieee international conference on systems, man and cybernetics.'cybernetics evolving to systems, humans, organizations, and their complex interactions'(cat. no. 0 (Vol. 3, pp. 2316-2320). IEEE.

Rabin, S. and Silva, F. (n.d.). Introduction 14.2 Pruning Strategy 14.3 Forced Neighbors 14.4 Jump Points 14.5 Wall Distances 14.6 Map Preprocess Implementation 14. [online] Available at:

http://www.gameapro.com/GameAIPro2/GameAIPro2_Chapter14_JPS_Plus_An_Extreme_A_Star_Speed_Optimization_for_Static_Uniform_Cost_Grids.pdf

Redblobgames.com. (2014). Red Blob Games: Introduction to A*. [online] Available at:

<https://www.redblobgames.com/pathfinding/a-star/introduction.html>.

Uggelberg, R. and Lundblom, A., 2017. Comparative Analysis of Weighted Pathfinding in Realistic Environments.

Unity Technologies (2021). Unity - Unity. [online] Unity. Available at: <https://unity.com/>.

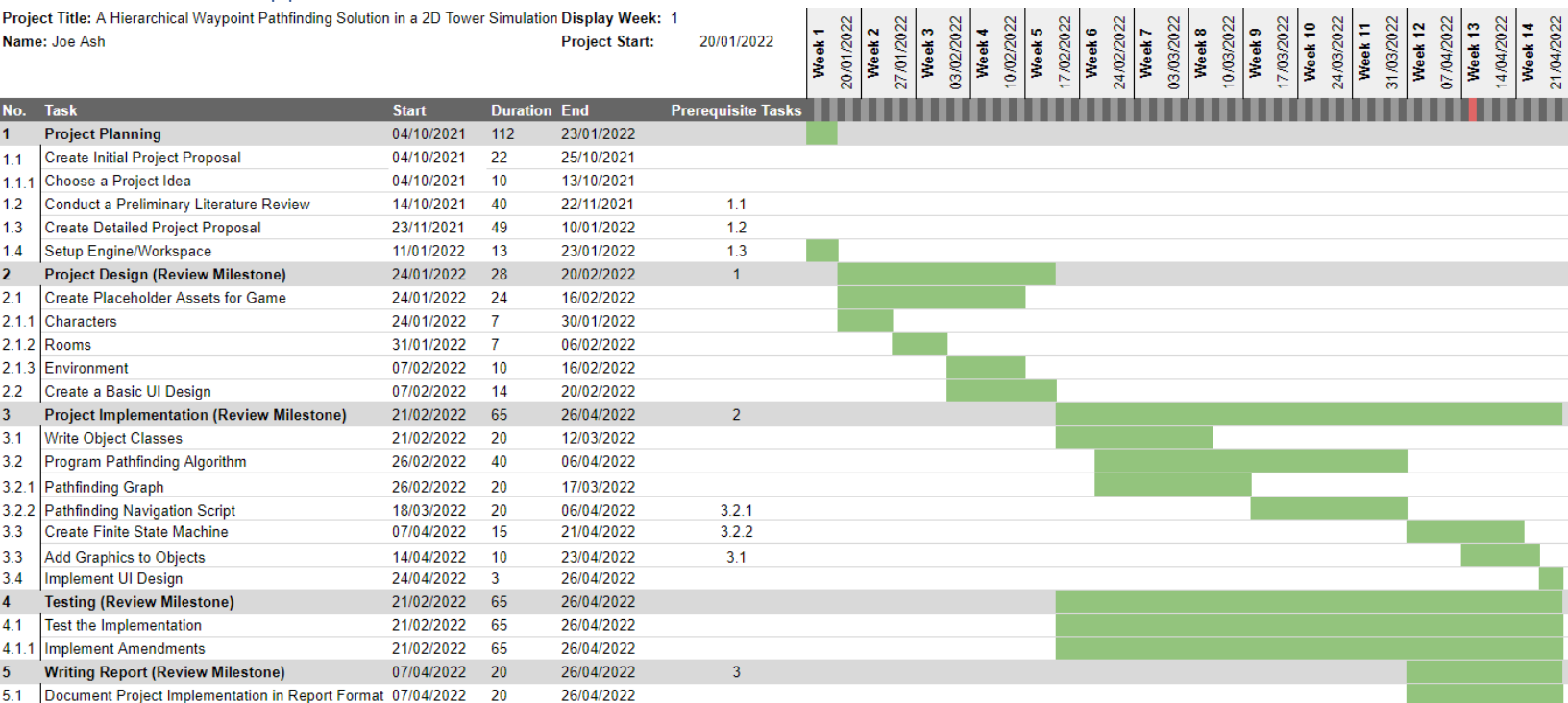
Zeyad Abd Algfoor, Mohd Shahrizal Sunar, Hoshang Kolivand, "A Comprehensive Study on Pathfinding Techniques for Robotics and Video Games", *International Journal of Computer Games Technology*, vol. 2015, Article ID 736138, 11 pages, 2015. <https://doi.org/10.1155/2015/736138>

Zhu, W., Jia, D., Wan, H., Yang, T., Hu, C., Qin, K. and Cui, X. (2015). Waypoint Graph Based Fast Pathfinding in Dynamic Environment. *International Journal of Distributed Sensor Networks*, 11(8), p.238727.

Zubek, R. (2017). 34 1000 NPCs at 60 FPS. [online] Available at: http://www.gameai.pro/GameAIPro3/GameAIPro3_Chapter34_1000_NPCs_at_60_FPS.pdf.

11.0 Appendices

11.1 Appendix A – Gantt Chart



11.2 Appendix B – Build Edges

```
import Globals from "./globals.js";
import * as Ut from "./ut.js";

export function buildEdges(runtime){
  defineNodes(runtime);
  for (const building of runtime.objects.sprBuilding.getAllInstances()){
    // add building to list of buildings
    for (const lift of runtime.objects.sprLiftShaft.getAllInstances()){
      // if a building is overlapping a lift
      if (building.testOverlap(lift)){
        Globals.edges.push([String(building.uid),String(lift.uid),Ut.distanceTo(building.x,building.y,lift.x,lift.y)]);
      }
    }
  }
}
```

```

        // otherwise just put it on the graph with no connections
        else{Globals.graph[String(building.uid)] = [[],[[]]}
    }

    // testing if a room overlaps a building (to know what building the room
is in)
    for (const room of runtime.objects.sprRoom.getAllInstances()){
        if (room.testOverlap(building)){
            Globals.subEdges.push([String(room.uid),String(building.uid)]);
        }
    }

    // testing for ground overlap
    const groundInstance = runtime.objects.sprGroundFloor.getFirstInstance();
    if (building.testOverlap(groundInstance)){
        Globals.edges.push([String(groundInstance.uid),String(building.uid),U
t.distanceTo(building.x,building.y,groundInstance.x,groundInstance.y)]);
    }
}
//console.log(Globals.edges, Globals.subEdges, Globals.nodes);
}

function defineNodes(runtime){
    for (const building of runtime.objects.famBuilding.getAllInstances()){
        if (building.objectType.name === "sprRoom"){
            addToNodes(building.uid, "room", building.x, building.y);
        }else if (building.objectType.name === "sprBuilding"){
            addToNodes(building.uid, "building", building.x, building.y);
        }else if (building.objectType.name === "sprLiftShaft"){
            addToNodes(building.uid, "lift", building.x, building.y);
        }else if (building.objectType.name === "sprGroundFloor"){
            addToNodes(building.uid, "groundFloor", building.x, building.y);
        }
    }
}

function addToNodes(node, type, localX, localY){
    if (!Globals.nodes.hasOwnProperty(node)){
        Globals.nodes[node] = [];
    }
    Globals.nodes[node].push(type, localX, localY);
}
}

```

11.3 Appendix C – Build Graphs

```

export function buildGraphs(){
    populateGraph(Globals.edges, Globals.graph, true);
}

```

```

    populateGraph(Globals.subEdges, Globals.subGraph, false);
}

function populateGraph(LocalEdge, graph, cost){
    // loop through every edge
    for (const edge of LocalEdge){
        let a = edge[0];
        let b = edge[1];

        // if doesn't exist, initialize
        if (!graph.hasOwnProperty(a)){
            if (cost){graph[a] = [[],[]];}
            else{graph[a] = [];}
        }
        if (!graph.hasOwnProperty(b)){
            if (cost){graph[b] = [[],[]];}
            else{graph[b] = [];}
        }

        // adding cost to the graph
        if (cost){
            // creating the adjacency list
            graph[a][0].push(b);
            graph[b][0].push(a);
            // adding cost
            graph[a][1].push(Math.round(edge[2]));
            graph[b][1].push(Math.round(edge[2]));
        }else {
            // creating the adjacency list
            graph[a].push(b);
            graph[b].push(a);
        }
    }
    //console.log(graph);
}

```

11.4 Appendix D – BFS Algorithm

```

import Globals from "./globals.js";
import * as Ut from "./ut.js";
import * as ActionSelection from "./actionSelection.js";

// breadth first search algorithm
export function bfs(graph, start, dest){
    let explored = [];
    let searched = [];
    // change the room into its corresponding building

```



```

let queue = [[Globals.subGraph[start]]];
let newDest = Globals.subGraph[dest][0];

if (start === dest){
  console.log("Already there")
  return
}

while (queue.length !== 0){
  let path = queue.pop();
  let node = path[path.length-1];
  if (searched.indexOf(node) === -1){searched.push(node);}

  if (!explored.includes(node)){
    let neighbours = graph[node][0];

    for (const neighbour of neighbours){
      if (searched.indexOf(neighbour) === -1){searched.push(neighbour);}
      let newPath = Array(path);
      newPath.push(neighbour);
      queue.push(newPath);

      if (neighbour === newDest){
        // original dest and start
        newPath.push(dest);
        newPath.unshift(start);
        console.log("BFS Path: " + newPath);
        return
      }
    }
    explored.push(node);
  }
}
console.log("No connection");
return
}

```

11.5 Appendix E – A* Algorithm

```

export function astar(graph, start, dest, person, runtime){
  let newStart = Globals.subGraph[start][0]
  let openSet = [newStart];
  let cameFrom = {};
  let gScore = {[newStart]: 0};
  let fScore = {[newStart]: Infinity};

  // change the room into its corresponding building

```

```

    let newDest = Globals.subGraph[dest][0];

    while (openSet.length !== 0){
        // node with lowest f(n)
        let current = returnLowestFScore(openSet, fScore);
        // old system (broken) let current = Object.keys(fScore).reduce((key, v) =>
        fScore[v] < fScore[key] ? v : key);

        if (current === newDest){
            //console.log("A* Path: " + reconstructPath(cameFrom, current, start,
            dest));
            person.currentPath = reconstructPath(cameFrom, current, start, dest);
            // send the person on the path
            person.currentState = "moving";
            ActionSelection.goToNextNode(runtime, person);
            return
        }
        openSet.splice(openSet.indexOf(current), 1);
        let neighbours = graph[current][0];

        for (const neighbour of neighbours){
            let localGScore = gScore[current] + returnCost(current, neighbour);
            if (!gScore.hasOwnProperty(neighbour)){gScore[neighbour] = Infinity;}
            // if a lift, we increase the cost based on the lift's queue
            if (Globals.nodes[neighbour][0] === "lift"){
                localGScore = localGScore + Globals.liftShaftAmount[neighbour]
            }
            if (localGScore < gScore[neighbour]){
                cameFrom[neighbour] = current;
                gScore[neighbour] = localGScore;
                fScore[neighbour] = localGScore + returnManhattan(neighbour,
newDest);
                if (!openSet.includes(neighbour)){
                    openSet.push(neighbour);
                }
            }
        }
    }
    person.currentState = "lost";
    return
}

function reconstructPath(cameFrom, current, start, dest){
    let path = [current];
    let localCurrent = current;

```

```

    while (cameFrom.hasOwnProperty(localCurrent)){
        localCurrent = cameFrom[localCurrent];
        path.push(localCurrent);

    }
    // original dest and start (of the rooms)
    path.push(start);
    path.unshift(dest);
    return path.reverse();
}

function returnLowestFScore(openSet, fScore){
    let min = fScore[openSet[0]];
    let node = openSet[0];
    for (const open of openSet){
        if (fScore[open] < min){
            min = fScore[open];
            node = open;
        }
    }
    return node;
}

function returnCost(node, dest){
    return
    Ut.distanceTo(Globals.nodes[node][1],Globals.nodes[node][2],Globals.nodes[dest][1]
    ],Globals.nodes[dest][2]);
}

function returnManhattan(node, dest){
    return
    Ut.manhattanDistance(Globals.nodes[node][1],Globals.nodes[node][2],Globals.nodes[
    dest][1],Globals.nodes[dest][2]);
}

```

11.6 Appendix F – Person Class

```

import Globals from "./globals.js";

export default class PersonInstance extends globalThis.ISpriteInstance{
    constructor(){
        super();

        // for primary problem
        this.currentRoom = "";
        this.currentPath = [];
        this.currentState = "";
    }
}

```

```

        this.previousNode = "";
        this.liftOn = [];
        this.chosenLift = "";

        // for secondary problem
        this.type = "";
        this.primaryBuilding = "";
    }
}

```

11.7 Appendix G – Lift Class

```

export default class LiftInstance extends globalThis.ISpriteInstance{
    constructor(){
        super();
        this.parentLiftShaft = "";
        this.goesTo = [];
        this.queue = [];
        this.currentState = "waiting for passengers";
        this.currentFloor = "";
        this.pointer = 0;
        this.capacity = 10;
        this.onBoard = 0;
        this.missPeopleOn = "";
    }
}

```

11.8 Appendix H – Global Variables and Objects

```

const Globals = {
    CURRENTLY_PLACING: 1,
    GRID_SIZE_X: 5,
    GRID_SIZE_Y: 20,
    EDGE_COUNT: 0,
    GROUND_LEVEL: 360,
    TIME: 475,
    SPEED: 1,
    TIMER: "",
    LIFTTIMER: "",
    JUST_BUILT: false,
    CENTER_X: 320,
    CENTER_Y: 240,
    IS_PLACING: false,
    LAST_SELECTED: 0,

    // graph object storing the nodes and corresponding connecting nodes
    graph: {},
    // information about the nodes

```

```

nodes: {},
// sub graph storing nodes of rooms and corresponding building nodes
subGraph: {},

// 2d array of edges between nodes
edges: [],
// 2d array of edges between rooms and buildings (format: "r, b")
subEdges: [],

// lift shaft stores the queue
liftShaft: {},

// stores the quantity of people waiting for each lift
liftShaftAmount: {},

// global schedule (secondary problem)
// in minutes, so 480 = 8am, 540 = 9am etc.
schedule: {
  "worker": {
    // morning
    480: ["out", 0.5, "go to work"],
    540: ["out", 1, "go to work"],
    // midday
    660: ["working", 0.2, "go to bathroom"],
    720: ["working", 0.5, "go to eat"],
    780: ["working", 0.5, "go to eat"],
    800: ["eating", 0.3, "go to bathroom"],
    810: ["working", 0.3, "go to eat"],
    830: ["eating", 0.3, "go to bathroom"],
    // evening
    960: ["working", 0.3, "go to bathroom"],
    1020: ["working", 0.5, "go home"],
    1050: ["working", 0.5, "go home"],
    1080: ["working", 0.8, "go home"],
    1110: ["working", 1, "go home"]
  },
  "resident": {
    // morning
    480: ["idle", 0.3, "go to eat"],
    485: ["idle", 0.5, "go out"],
    540: ["idle", 0.3, "go to eat"],
    545: ["idle", 0.5, "go out"],
    600: ["idle", 0.2, "go out"],
    660: ["idle", 0.2, "go out"],
    // midday

```

```

710: ["idle", 0.2, "go out"],
720: ["idle", 0.5, "go to eat"],
780: ["idle", 0.5, "go to eat"],
810: ["idle", 0.3, "go to eat"],
820: ["idle", 0.2, "go out"],
    // evening
1020: ["idle", 0.2, "go to eat"],
1050: ["idle", 0.2, "go to eat"],
1080: ["idle", 0.2, "go to eat"],
1110: ["idle", 0.2, "go to eat"],
1200: ["idle", 0.2, "go out"],
1260: ["idle", 0.2, "go out"]
    }
  }
};

// Export the object representing global variables.
export default Globals;

```

11.9 Appendix I – Agent Simulation

```

import Globals from "./globals.js";
import * as Ut from "./ut.js";
import * as Searching from "./searchingGraph.js";

export function generatePeople(runtime){
  let spawn = "";
  for (const room of runtime.objects.sprRoom.getAllInstances()){
    if (room.animationName === "spawn"){
      spawn = room;
    }
  }
  // loop through existing rooms in order to generate new people
  for (const room of runtime.objects.sprRoom.getAllInstances()){
    // generate resident
    residentCheck: if (room.animationName === "flat"){
      if (room.capacity !== 100){break residentCheck;}
      room.capacity = Ut.randomNum(1, 5);
      for(let i = 0; i < room.capacity; i++){
        const lastPersonInstance =
runtime.objects.sprPerson.createInstance("characters", spawn.x + Ut.randomNum(-
10,10), spawn.y, false);
        lastPersonInstance.y = lastPersonInstance.y -
(lastPersonInstance.height/2)
        lastPersonInstance.currentRoom = String(spawn.uid);
        lastPersonInstance.type = "resident";

```

```

        lastPersonInstance.primaryBuilding = String(room.uid);
        lastPersonInstance.currentState = "out";
        Searching.astar(Globals.graph, lastPersonInstance.currentRoom,
lastPersonInstance.primaryBuilding, lastPersonInstance, runtime);
        room.occupancy++;
    }
}
// generate worker
officeCheck: if (room.animationName === "office"){
    if (room.capacity !== 100){break officeCheck;}
    room.capacity = 10;
    for(let i = 0; i < room.capacity; i++){
        const lastPersonInstance =
runtime.objects.sprPerson.createInstance("characters", spawn.x + Ut.randomNum(-
10,10), spawn.y, false);
        lastPersonInstance.y = lastPersonInstance.y -
(lastPersonInstance.height/2)
        lastPersonInstance.currentRoom = String(spawn.uid);
        lastPersonInstance.type = "worker";
        lastPersonInstance.primaryBuilding = String(room.uid);
        lastPersonInstance.currentState = "out";
        room.occupancy++;
    }
}
}
}

export function checkForHomeless(runtime){
    let allRooms = [];
    for (const room of runtime.objects.sprRoom.getAllInstances()){
        if (room.animationName === "office" || room.animationName === "flat"){
            allRooms.push(String(room.uid));
        }
    }
    for (const person of runtime.objects.sprPerson.getAllInstances()){
        if (!allRooms.includes(person.primaryBuilding)){
            person.destroy();
        }
    }
}

export function findLiftShaft(runtime){
    for (const lift of runtime.objects.sprLift.getAllInstances()){
        for (const liftShaft of runtime.objects.sprLiftShaft.getAllInstances() ){
            if (lift.testOverlap(liftShaft)){

```

```

        lift.parentLiftShaft = String(liftShaft.uid);
        lift.goesTo = Globals.graph[liftShaft.uid][0];

        // create corresponding queue amount
        if
(!Globals.liftShaftAmount.hasOwnProperty(String(liftShaft.uid))){Globals.liftShaftAmount[String(liftShaft.uid)] = 0;}
        lift.queue = [];
        lift.onBoard = 0;
        // add queue (in order of y)
        for (const i of Globals.graph[liftShaft.uid][0]){
            let relevantIndex = 0;
            let c = 0;
            for (const q of lift.queue){
                c++;
                if (Globals.nodes[i][2] > Globals.nodes[q[0]][2]){
                    relevantIndex = c;
                }
            }
            lift.queue.splice(relevantIndex, 0, [i,0]);
            //lift.queue.push([i,0]);
        }
    }
}

}

}

}

export function addLiftShaftQueue(runtime){
    for (const liftShaft of runtime.objects.sprLiftShaft.getAllInstances()){
        Globals.liftShaft[String(liftShaft.uid)] = [];
        // add the floors it goes to with the corresponding passengers waiting
        (at the start 0)
        for (const i of Globals.graph[liftShaft.uid][0]){
            Globals.liftShaft[String(liftShaft.uid)].push([i,0]);
        }
    }
}

export function findRooms(runtime){
    // finding the room each person is on
    for (const person of runtime.objects.sprPerson.getAllInstances()){
        for (const room of runtime.objects.sprRoom.getAllInstances() ){
            if (person.testOverlap(room)){

```



```

        person.currentRoom = String(room.uid);
    }
}
}
for (const lift of runtime.objects.sprLift.getAllInstances()){
    for (const building of runtime.objects.sprBuilding.getAllInstances()){
        if (lift.testOverlap(building)){
            lift.currentFloor = String(building.uid);
        }
    }
}
}

function liftArrived(lift, runtime){
    for (const person of runtime.objects.sprPerson.getAllInstances()){
        if (person.currentState === "waiting for lift"){
            if (person.testOverlap(lift)){
                if (lift.currentState === "waiting for passengers"){
                    if (lift.onBoard < lift.capacity){
                        // remove old node (in this case the lift they are about
to ride)

                        person.previousNode = person.currentPath[0];
                        person.currentPath.shift();
                        // tell the lift where to go
                        activateQueue(lift, person.currentPath[0]);
                        // change state
                        person.currentState = "riding lift";
                        // give some variation to position and pin them to lift
                        person.liftOn = [lift, Ut.randomNum(-10,10), 10];

                        //update amount on board
                        lift.onBoard++;
                    }
                    // if it has left people behind, remember that floor and add
it back to the queue later
                    else{
                        lift.missPeopleOn = lift.currentFloor;
                    }
                }
            }
        }
    }
}

export function stillWaiting(runtime){
    for (const person of runtime.objects.sprPerson.getAllInstances()){

```

```

        if (person.currentState === "waiting for lift"){
            addToLiftShaftQueue(person.currentPath[0], person.previousNode);
            console.log("I'm still waiting!");
        }
    }
}

export function ridingLift(runtime, person){
    person.x = person.liftOn[0].x + person.liftOn[1];
    person.y = person.liftOn[0].y + person.liftOn[2];
    // on arrived at desired floor
    if (person.liftOn[0].currentFloor === person.currentPath[0]){
        person.currentState === "moving";
        // reducing amount of people in queue
        Globals.liftShaftAmount[person.previousNode]--;

        goToNextNode(runtime, person);

        //update amount on board
        person.liftOn[0].onBoard--;
    }
}

export function arrivedAtLift(person){
    // if they have arrived at the lift
    if (person.currentState === "going to lift" &&
    !person.behaviors.MoveTo.isMoving){
        let nextNode = person.currentPath[1];
        // tell the lift shaft you are waiting (push the button)
        addToLiftShaftQueue(person.currentPath[0], person.previousNode);
        person.currentState = "waiting for lift";
    }
    // addToLiftQueue(runtime, nextNode, person.previousNode,
    person.currentPath[1], person)
}

function addToLiftShaftQueue(liftShaft, currentFloor){
    let liftQueue = Globals.liftShaft[liftShaft];
    let c = 0;
    for (const floor of liftQueue){
        if (floor[0] === currentFloor){
            Globals.liftShaft[liftShaft][c][1] =
Globals.liftShaft[liftShaft][c][1]+1;
            //Globals.liftShaftAmount[liftShaft]++;
        }
    }
}

```

```

    }
    c++;
}
}
export function goToNextNode(runtime, person){
    while (person.currentPath.length !== 0){
        let nextNode = person.currentPath[0]
        // if the next node does not exist anymore, make the agent lost
        if
(!Globals.nodes.hasOwnProperty(nextNode)){person.currentState="lost";break;}
        // if they have arrived
        if (person.currentPath.length === 1){
            person.behaviors.MoveTo.moveToPosition(Globals.nodes[nextNode][1]
+ Ut.randomNum(-20,20), person.y, true);
            person.currentRoom = person.currentPath[0];

            // depending on where they arrive, change their current state
            let arrivedRoom = getArrivedRoomAnimationName(runtime, person,
person.currentPath[0]);
            if (arrivedRoom === "foodCourt"){
                person.currentState = "eating";
            }else if (arrivedRoom === "office"){
                person.currentState = "working";
            }else if (arrivedRoom === "spawn"){
                person.currentState = "out";
            }else{
                person.currentState = "idle";
            }
            // remove the last node from their path
            person.currentPath.shift();
            return
        }else{
            // check what the node is
            if (Globals.nodes[nextNode][0] === "room"){
                person.behaviors.MoveTo.moveToPosition(Globals.nodes[nextNode][1]
, person.y, true);
                person.previousNode = nextNode;
                person.currentPath.shift();
            }else if (Globals.nodes[nextNode][0] === "building"){
                person.previousNode = nextNode;
                person.currentPath.shift();
            }else if (Globals.nodes[nextNode][0] === "groundFloor"){
                person.previousNode = nextNode;
                person.currentPath.shift();
            }else if (Globals.nodes[nextNode][0] === "lift"){

```

```

        Globals.liftShaftAmount[nextNode]++;
        person.behaviors.MoveTo.moveToPosition(Globals.nodes[nextNode][1]
+ Ut.randomNum(-10,10), person.y, true);
        person.currentState = "going to lift";
        return
    }
}
}
}

function getArrivedRoomAnimationName(runtime, person, currentRoom){
    for (const room of runtime.objects.sprRoom.getAllInstances()){
        if (String(room.uid) === currentRoom){
            return room.animationName;
        }
    }
}

export function callLiftToFloor(runtime){
    // go to each liftShaft and look through all of the floors that have people
    // waiting there, then send relevant lifts to pick them up
    for (const liftShaft of runtime.objects.sprLiftShaft.getAllInstances()){
        const floors = Globals.liftShaft[String(liftShaft.uid)];
        for (const floor of floors){
            // if at least one person is waiting
            if(floor[1] > 0){
                //console.log(findLeastEmptyLift(runtime, liftShaft));
                const leastEmpty = findLeastEmptyLift(runtime, liftShaft);
                activateQueue(leastEmpty, floor[0]);
                reduceWaiting(liftShaft, floor[0], leastEmpty.capacity);
            }
        }
    }
}

function reduceWaiting(liftShaft, floor, amount){
    // reduces the amount of people listed as waiting
    let c = 0;
    for (const i of Globals.liftShaft[String(liftShaft.uid)]){
        if (i[0] === floor){
            Globals.liftShaft[String(liftShaft.uid)][c][1] =
Globals.liftShaft[String(liftShaft.uid)][c][1] - amount;
        }
        if (i[1] < 0){
            Globals.liftShaft[String(liftShaft.uid)][c][1] = 0;
        }
    }
}

```

```

        c++;
    }
}

export function activateQueue(lift, dest){
    let localQueue = lift.queue;
    let c = 0;
    for (const q of localQueue){
        if (q[0] === dest){
            lift.queue[c][1] = 1;
        }
        c++;
    }
}

function findLeastEmptyLift(runtime, LiftShaft){
    let smallestQueueLift = 0;
    let smallestQueue = Infinity;
    for (const lift of runtime.objects.sprLift.getAllInstances()){
        // if its a child of the liftShaft
        if (String(LiftShaft.uid) === lift.parentLiftShaft){
            // get the lift with the smallest queue
            if (countQueue(lift) < smallestQueue){
                smallestQueueLift = lift;
                smallestQueue = countQueue(lift);
            }
        }
    }
    return smallestQueueLift;
}

function countQueue(lift){
    let c = 0
    for (const floor of lift.queue){
        if (floor[1] === 1){
            c++;
        }
    }
    return c;
}

function isQueueNotEmpty(lift){
    for (const floor of lift.queue){
        if (floor[1] === 1){
            return true;
        }
    }
}

```

```

    }
    return false;
}

export function moveLifts(runtime){
    for (const lift of runtime.objects.sprLift.getAllInstances()){
        // if there is something in the queue -> move to it
        if (lift.currentState === "waiting for passengers" &&
isQueueNotEmpty(lift)){
            // keep incrementing the pointer until it gets to the size of the
queue, then reverse the queue and keep going
            updatePointer(lift);
            // once at a floor that someone is waiting at, move to it
            lift.behaviors.MoveTo.moveToPosition(lift.x,
Globals.nodes[lift.queue[lift.pointer][0]][2], true);
            lift.queue[lift.pointer][1] = 0;
            lift.currentState = "moving";
        }

        // stop the lift when at destination
        if (!lift.behaviors.MoveTo.isMoving){
            if (lift.currentState === "moving"){
                lift.currentState = "waiting for passengers";
                lift.currentFloor = lift.queue[lift.pointer][0];
                // if it has missed people
                if (lift.missPeopleOn !== "" && lift.missPeopleOn !==
lift.currentFloor){
                    activateQueue(lift, lift.missPeopleOn);
                    lift.missPeopleOn = "";
                }
                liftArrived(lift, runtime);
            }
        }
    }
}

function updatePointer(lift){
    while (lift.queue[lift.pointer][1] !== 1){
        lift.pointer++;
        if(lift.pointer === lift.queue.length){
            console.log("reverse");
            lift.queue.reverse();
            lift.pointer = 0;
        }
    }
}

```

```
}
```

11.10 Appendix J – Utilities

```
export function distanceTo(x1, y1, x2, y2){
  return Math.hypot(x2 - x1, y2 - y1);
}

export function angleTo(x1, y1, x2, y2){
  return Math.atan2(y2 - y1, x2 - x1);
}

export function manhattanDistance(x1, y1, x2, y2){
  return Math.abs(x2-x1) + Math.abs(y2-y1);
}

export function randomNum(min, max){
  return Math.floor(Math.random() * (max - min + 1) + min)
}
```

11.11 Appendix K – Activate Tasks

```
// what to do with specific tasks
// some tasks, a person goes to a prestored room, other times they find the
// nearest relevant room
function activateTask(runtime, task, person){
  if (task === "go to work"){
    Searching.astar(Globals.graph, person.currentRoom,
person.primaryBuilding, person, runtime);
  }else if (task === "go to eat"){
    let nearestRoom = findNearestRoom(runtime, person, "foodCourt")
    // if there is a nearest room,
    if (nearestRoom !== "no rooms"){
      Searching.astar(Globals.graph, person.currentRoom, nearestRoom,
person, runtime);
    }
  }else if (task === "go back home"){
    Searching.astar(Globals.graph, person.currentRoom,
person.primaryBuilding, person, runtime);
  }else if (task === "go home" || task === "go out"){
    let nearestRoom = findNearestRoom(runtime, person, "spawn")
    // if there is a nearest room,
    if (nearestRoom !== "no rooms"){
      Searching.astar(Globals.graph, person.currentRoom, nearestRoom,
person, runtime);
    }
  }else if (task === "go to bathroom"){
    let nearestRoom = findNearestRoom(runtime, person, "bathroom")
  }
```

```

        // if there is a nearest room,
        if (nearestRoom !== "no rooms"){
            Searching.astar(Globals.graph, person.currentRoom, nearestRoom,
person, runtime);
        }
    }
}

```

11.12 Appendix L – Find Nearest Room

```

// to find the nearest relevant room, this function is called
function findNearestRoom(runtime, person, type){
    let currentLowestDist = Infinity;
    let currentRoom = "";
    let chance = 0.3;
    for (const room of runtime.objects.sprRoom.getAllInstances()){
        if (room.animationName === type){
            if (Ut.distanceTo(room.x, room.y, person.x, person.y) <
currentLowestDist){
                currentLowestDist = Ut.distanceTo(room.x, room.y, person.x,
person.y);
                currentRoom = String(room.uid);

                // chance of leaving the loop and retaining the current smallest
(sometimes they don't find the smallest)
                if (Math.random() < chance){
                    break;
                }
            }
        }
    }
    if (currentRoom !== ""){
        return currentRoom;
    }else{
        return "no rooms";
    }
}

```

11.13 Appendix M – Check Schedule and Update the Tasks

```

// every time the time changes, check if a corresponding schedule event needs to
be checked
export function checkSchedule(runtime){
    let workerSchedule = Globals.schedule["worker"];
    let residentSchedule = Globals.schedule["resident"];

    if (workerSchedule.hasOwnProperty(Globals.TIME)){
        updateTasks(runtime, workerSchedule[Globals.TIME], "worker");
    }
}

```



```

    }
    if (residentSchedule.hasOwnProperty(Globals.TIME)){
        updateTasks(runtime, residentSchedule[Globals.TIME], "resident");
    }

    // certain tasks always run every hour
    if (Globals.TIME % 60 === 0){
        updateTasks(runtime, ["eating", 1, "go back home"], "resident");
        updateTasks(runtime, ["eating", 1, "go to work"], "worker");
        updateTasks(runtime, ["out", 0.5, "go back home"], "resident");
        updateTasks(runtime, ["idle", 1, "go to work"], "worker");

        //if they are lost
        updateTasks(runtime, ["lost", 1, "go to work"], "worker");
        updateTasks(runtime, ["lost", 1, "go back home"], "resident");

        // after a certain time, on the hour certain tasks can run
        if (Globals.TIME > 1110 || Globals.TIME < 480){
            updateTasks(runtime, ["working", 1, "go home"], "worker");
        }
    }
}

// if an event is checked, then check requirements and activate if possible
function updateTasks(runtime, tasks, type){
    for (const person of runtime.objects.sprPerson.getAllInstances()){
        if (person.type === type){
            if (person.currentState === tasks[0]){
                if (Math.random() < tasks[1]){
                    activateTask(runtime, tasks[2], person);
                }
            }
        }
    }
}
}

```

11.14 Appendix N – Adding and Updating Time

```

// change the time forward
export function addTime(runtime){
    if (Globals.SPEED !== 0){
        Globals.TIME++;
        if (Globals.TIME > 1440){
            Globals.TIME = 0;
        }
    }
}

```

```

    }
    // update clock
    const timeText = runtime.objects.txtTime.getFirstInstance();
    timeText.text = String(Math.floor(Globals.TIME / 60)).padStart(2, "0") +
    ":" + String(Globals.TIME % 60).padStart(2, "0");

    // check schedule
    checkSchedule(runtime);
  }
}

```

11.15 Appendix O – Room Class

```

export default class RoomInstance extends globalThis.ISpriteInstance{
  constructor(){
    super();
    this.capacity = 100;
    this.occupancy = 0;
  }
}

```

11.16 Appendix P – On Mouse Down

```

import Globals from "./globals.js";
import * as ActionSelection from "./actionSelection.js";
import * as Generating from "./generatingGraph.js";
import * as Searching from "./searchingGraph.js";
import * as Ut from "./ut.js";

// what happens when the mouse is clicked
export function onMouseDown(e, runtime){

  // if right click pressed
  if (e.button === 2){
    console.log(runtime.layout.getLayer("ui"));
    // if trying to place a building, cancel it
    if (Globals.IS_PLACING){
      Globals.IS_PLACING = false;
      runtime.objects.sprBuildOutline.getFirstInstance().destroy();
      return;
    }

    if (Globals.SPEED === 0){
      let highestInstance = "";
      let c = 0
      for (const building of runtime.objects.famRooms.getAllInstances()){
        if
        (building.containsPoint(runtime.mouse.getMouseX(), runtime.mouse.getMouseY())){

```

```

        if (c===0){highestInstance = building;}
        c++;
        if (building.zIndex > highestInstance.zIndex){
            highestInstance = building;
        }
    }
}
if (highestInstance !== ""){highestInstance.destroy();return;}

// since buildings are on a lower layer, they should be destroyed
separately
for (const building of
runtime.objects.sprBuilding.getAllInstances()){
    if
(building.containsPoint(runtime.mouse.getMouseX(),runtime.mouse.getMouseY())){
        building.destroy()
        return;
    }
}

}
}
// check for left mouse button
if (e.button === 0){
    const mouseX = runtime.mouse.getMouseX();
    const mouseY = runtime.mouse.getMouseY();

    // controlling time
    if (!Globals.IS_PLACING){
        for (const timeControl of
runtime.objects.sprTimeControl.getAllInstances()){
            if (timeControl.containsPoint(mouseX,mouseY)){
                if (timeControl.animationName === "pause"){
                    Globals.SPEED = 0;
                    clearInterval(Globals.TIMER);
                    clearInterval(Globals.LIFTTIMER);
                    runtime.layout.getLayer("buildingButtons").isVisible =
false;
                }else if (timeControl.animationName === "play"){
                    Globals.SPEED = 1;
                    clearInterval(Globals.TIMER);
                    Globals.TIMER = setInterval(() =>
ActionSelection.addTime(runtime), 300);

```

```

        clearInterval(Globals.LIFTTIMER);
        Globals.LIFTTIMER = setInterval(() =>
ActionSelection.moveLifts(runtime), 300);
        justBuilt(runtime);
        runtime.layout.getLayer("buildingButtons").isVisible =
false;

        }else if (timeControl.animationName === "faster"){
            Globals.SPEED = 3;
            clearInterval(Globals.TIMER);
            Globals.TIMER = setInterval(() =>
ActionSelection.addTime(runtime), 50);
            clearInterval(Globals.LIFTTIMER);
            Globals.LIFTTIMER = setInterval(() =>
ActionSelection.moveLifts(runtime), 50);
            justBuilt(runtime);
            runtime.layout.getLayer("buildingButtons").isVisible =
false;

            }else if (timeControl.animationName === "build"){
                Globals.SPEED = 0;
                clearInterval(Globals.TIMER);
                clearInterval(Globals.LIFTTIMER);
                evacuateBuilding(runtime);
                Globals.JUST_BUILT = true;
                runtime.layout.getLayer("buildingButtons").isVisible =
true;

                }
            return;
        }
    }
}

// adding new parts to the building
if (Globals.IS_PLACING){
    const outline = runtime.objects.sprBuildOutline.getFirstInstance();
    placeBuilding(Globals.LAST_SELECTED, runtime, outline.width,
outline.height);
}
if (Globals.SPEED === 0){
    for (const buildingButton of
runtime.objects.buildingButton.getAllInstances()){
        if (buildingButton.containsPoint(mouseX,mouseY)){
            if (!Globals.IS_PLACING){
                const outline =
runtime.objects.sprBuildOutline.createInstance("characters", mouseX, mouseY,
false);

```

```

        outline.width = returnBuildingWidth(runtime,
buildingButton.animationFrame);
        outline.height =
returnBuildingHeight(buildingButton.animationFrame);
        Globals.IS_PLACING = true;
        Globals.LAST_SELECTED = buildingButton.animationFrame;
    }
}
}

// logging information about something
for (const person of runtime.objects.sprPerson.getAllInstances()){
    if (person.containsPoint(mouseX,mouseY)){
        console.log(person);
        return;
    }
}
for (const room of runtime.objects.sprRoom.getAllInstances()){
    if (room.containsPoint(mouseX,mouseY)){
        console.log(room);
        return;
    }
}
}
}
}

```

11.17 Appendix Q – Evacuate Buildings

```

function evacuateBuilding(runtime){
    for (const person of runtime.objects.sprPerson.getAllInstances()){
        person.x = -60;
        person.y = 370;
        person.currentState = "idle";
    }
}

// resetting global objects
export function resetGlobal(){
    Globals.graph = Object();
    Globals.nodes = Object();
    Globals.subGraph = Object();
    Globals.liftShaft = Object();
    Globals.liftShaftAmount = Object();
    Globals.edges = [];
    Globals.subEdges = [];
}

```

11.18 Appendix R – Evacuate Buildings

```
function justBuilt(runtime){
  if (Globals.JUST_BUILT){
    Globals.JUST_BUILT = false;

    // if buildings are in invalid locations
    let overlap = false;
    for (const liftShaft of runtime.objects.sprLiftShaft.getAllInstances()){
      overlap = false;
      for (const building of
runtime.objects.sprBuilding.getAllInstances()){
        if (liftShaft.testOverlap(building)){
          overlap = true;
        }
      }
      if (!overlap){
        liftShaft.destroy();
      }
    }
    for (const lift of runtime.objects.sprLift.getAllInstances()){
      overlap = false;
      for (const liftShaft of
runtime.objects.sprLiftShaft.getAllInstances()){
        if (lift.testOverlap(liftShaft)){
          overlap = true;
        }
      }
      if (!overlap){
        lift.destroy();
      }
    }
    for (const room of runtime.objects.sprRoom.getAllInstances()){
      overlap = false;
      for (const building of
runtime.objects.sprBuilding.getAllInstances()){
        if (room.testOverlap(building)){
          overlap = true;
        }
      }
      if (!overlap){
        room.destroy();
      }
    }
  }

  resetGlobal()
}
```

```

// generate the graph
Generating.buildEdges(runtime);
//Generating.visualizeNodes(runtime, true);
Generating.buildGraphs();

// use the graph to search
ActionSelection.addLiftShaftQueue(runtime);
ActionSelection.findLiftShaft(runtime);
ActionSelection.findRooms(runtime);

// secondary problem
ActionSelection.generatePeople(runtime);
ActionSelection.checkForHomeless(runtime);

// checks for problems in positioning and queue's needing to be updated
again
let roomX = -10;
let roomY = -10;

// if they are waiting for the lift, the lift needs to be told again
for (const person of runtime.objects.sprPerson.getAllInstances()){
    person.currentState = "idle";
    if (person.type === "resident" || (person.type === "worker" &&
(Globals.TIME < 1110 && Globals.TIME > 480))){
        Searching.astar(Globals.graph, person.currentRoom,
person.primaryBuilding, person, runtime);
    }
}
}
}

```

11.19 Appendix S – Evacuate Buildings

```

export function placeBuilding(value, runtime, width, height){
    if (Globals.SPEED === 0){
        const mouseX = runtime.mouse.getMouseX();
        const mouseY = runtime.mouse.getMouseY();
        let built = false;
        let tempBuilding = "";
        // place building
        if (value === 0){
            // set width to the last one created
            let oldWidth =
getLastInstanceWidth(runtime.objects.sprBuilding.getAllInstances());
            tempBuilding = runtime.objects.sprBuilding.createInstance("building",
mouseX, mouseY, false);
            built = true;

```

```

        }else if (value === 1){
            tempBuilding = runtime.objects.sprLiftShaft.createInstance("rooms",
mouseX, mouseY, false);
            built = true;
            tempBuilding.moveToBottom();
        }else if (value === 2){
            tempBuilding = runtime.objects.sprLift.createInstance("rooms",
mouseX, mouseY, false);
            built = true;
        }else if (value === 3){
            tempBuilding = runtime.objects.sprRoom.createInstance("rooms",
mouseX, mouseY, false);
            built = true;
            tempBuilding.setAnimation("flat", "beginning")
        }else if (value === 4){
            tempBuilding = runtime.objects.sprRoom.createInstance("rooms",
mouseX, mouseY, false);
            built = true;
            tempBuilding.setAnimation("office", "beginning")
        }else if (value === 5){
            tempBuilding = runtime.objects.sprRoom.createInstance("rooms",
mouseX, mouseY, false);
            built = true;
            tempBuilding.setAnimation("bathroom", "beginning")
        }else if (value === 6){
            tempBuilding = runtime.objects.sprRoom.createInstance("rooms",
mouseX, mouseY, false);
            built = true;
            tempBuilding.setAnimation("foodCourt", "beginning")
        }
        if (built){
            tempBuilding.x =
(Math.round(tempBuilding.x/Globals.GRID_SIZE_X))*Globals.GRID_SIZE_X;
            tempBuilding.y =
(Math.round(tempBuilding.y/Globals.GRID_SIZE_Y))*Globals.GRID_SIZE_Y;
            tempBuilding.width = width;
            tempBuilding.height = height;
        }
    }
}

export function expandRoom(e, runtime){
    if (Globals.SPEED !== 0){return;}
    if (Globals.SPEED === 0){

```



```

        if (e.deltaY < 0){
            for (const liftShaft of
runtime.objects.sprLiftShaft.getAllInstances()){
                if (liftShaft.containsPoint(runtime.mouse.getMouseX(),
runtime.mouse.getMouseY())){
                    liftShaft.height = liftShaft.height + 40;
                    return;
                }
            }
            for (const building of
runtime.objects.sprBuilding.getAllInstances()){
                if (building.containsPoint(runtime.mouse.getMouseX(),
runtime.mouse.getMouseY())){
                    building.width = building.width + 10;
                    return;
                }
            }
        }else{
            Globals.JUST_BUILT = true;
            for (const liftShaft of
runtime.objects.sprLiftShaft.getAllInstances()){
                if (liftShaft.containsPoint(runtime.mouse.getMouseX(),
runtime.mouse.getMouseY())){
                    if (liftShaft.height > 40){
                        liftShaft.height = liftShaft.height - 40;
                        return;
                    }
                }
            }
            for (const building of
runtime.objects.sprBuilding.getAllInstances()){
                if (building.containsPoint(runtime.mouse.getMouseX(),
runtime.mouse.getMouseY())){
                    if (building.width > 40){
                        building.width = building.width - 10;
                        return;
                    }
                }
            }
        }
    }
}

```

11.20 Appendix T – Event Sheet Additional Scripts

There were some additional in-engine scripts that were not possible by script (such as changing the time scale).

1	→  Mouse	On Left button Clicked on sprTimeControl	Add action
2	 sprTime...	Is animation "pause" playing	 System Set time scale to 0  famMov... Set  DragDrop Disabled Add action
3	 sprTime...	Is animation "play" playing	 System Set time scale to 1  famMov... Set  DragDrop Disabled Add action
4	 sprTime...	Is animation "faster" playing	 System Set time scale to 6  famMov... Set  DragDrop Disabled Add action
5	 sprTime...	Is animation "build" playing	 System Set time scale to 0  famMov... Set  DragDrop Enabled Add action
6	 famMov...	 DragDrop is dragging	 famMov... Set position to $(\text{round}(\text{Self.X} \div 5) \times 5, \text{round}(\text{Self.Y} \div 20) \times 20)$ Add action
7	→  famBuild...	On created	 famMov... Set  DragDrop Enabled Add action
8	→  sprBuildi...	On created	 sprBuildi... Set  DragDrop Enabled Add action
9	→  sprRoom	On created	 sprRoom Set  DragDrop Enabled Add action
10	→  sprLift	On created	 sprLift Set  DragDrop Enabled Add action
11	→  sprLiftSh...	On created	 sprLiftSh... Set  DragDrop Enabled Add action
12	→  System	On start of layout	 famMov... Set  DragDrop Disabled Add action