



Building a multi agent path finding warehouse simulation with artificial intelligence.

CM2302 - Individual Project - 40 Credits

Author - Tom Clare - C1769261

Supervisor - Bailin Deng

Moderator - Helen Phillips

Final Report

Acknowledgement

I would like to thank my supervisor, Bailin Deng, for being a tremendous help over the course of this project. He gave excellent insight into how I could improve my project at several key points during the development. He offered support over the whole process and was able to answer any questions I had quickly and in a comprehensive manner.

Abstract

The purpose of this project is to develop a visualisation and planning tool for large scale robot warehouses. Allowing the user to change characteristics and see how they can affect the efficiency of the warehouse, and in doing so allow users to find optimal characteristics. In addition to this it will also act as a visualisation tool for teaching people how robots can be used for large scale automation in this style of factories. It would also serve to demonstrate the advantages of using artificial intelligence algorithms in these types of scenarios.

Table of Contents

1. Introduction.....	4
1.1 Project Aims	4
1.2 Intended Audience.....	4
1.3 Important Outcomes	4
2. Background	4
2.1 Wider project context	4
2.2 Factors that go into warehouse design	5
2.3 Existing Solutions.....	6
2.3.1 Optimal Solution – Conflict Based Search.....	6
2.3.2 Non-Optimal Solution A* Search	8
2.4 Methods and Tools used.....	9
3. Specification and Design	10
3.1 The static architecture of the system	10
3.2 The user interface	11
3.2.1 Model Control	11
3.2.1 User Changeable Settings.....	12
3.2.3 Warehouse Floor	14
3.2.4 Order labels	14
4. Implementation	15
4.1 Initialising the model.....	15
4.1.1 Starting the server and initialising the model.	15
4.1.2 Initialising agents.....	16
4.1.4 Starting Data collector	17
4.2 Stepping the model.....	18
4.2.1 Blind Goal Step	18
4.2.2 Path Finding Step	19
4.2.2 Planning a route	20
4.3.3 Picking up and dropping items	22
4.3.4 Collecting Data.....	22
4.3.5 Building the Visualisation	23
5. Results and Evaluation.....	24
5.1 Evaluating the import outcomes of the project.	24
Evaluating the user interface of the system.....	29
6. Future Work	30
7. Conclusions.....	31
8. Reflection on Learning	32
9. Reference List	33
10. Appendix.....	34

1. Introduction

1.1 Project Aims

The aim for this project is to simulate a warehouse using robots to pick and sort grocery items into customer orders. The simulation will allow the users to setup starting parameters to be able to measure how they affect the efficiency of the warehouse.

1.2 Intended Audience

The intended audience for this project is companies that use these type of robot swarms. It is intended to be used by these companies to be able to prototype decisions they will need to make about how their swarm operates and see how these decisions impact performance. Other beneficiaries of this work would be people looking to get a greater understanding of how swarm robots operate. This simulation tool would also work as a visualisation for the education of people on how this technology can be implemented.

1.3 Important Outcomes

- Create a working visualisation of a robotic warehouse floor
 - The visualisation should be clear and easy to understand even to people without a technical background.
 - The visualisation needs to clearly show how the robots are operating in the grid.
- Have robots that are able to process customer orders
 - The Robots should be able to fulfil any customer orders generated by the system unless the setup parameters make this impossible.
 - There should be at least two modes of operation for the robots with one utilising artificial intelligence to better move the robots through the warehouse.
- Have a way of seeing how the model is running compared to other models
 - The systems should have graphs and logs to be able to easily compare how changing settings impact performance of the simulation.

2. Background

2.1 Wider project context

In recent years the model that supermarkets use to get business from their customers has changed drastically. This started with allowing customers to pick out their shopping orders online and then “pickers” working in supermarkets would collect the items off the shop floor and pack the customers shopping for them before loading these orders into delivery trucks to be delivered to the customer. This is still the case in some areas not located near to dedicated fulfilment centres.

This was a massive change to traditional shopping and soon became incredibly popular. This left supermarket chains with both massive opportunities and challenges. The idea of having local pickers worked well at first due to having low initial setup costs and low traffic, however as the number of customers that wanted to use the delivery service increased it became difficult to keep up with demand. Human pickers were collecting and packing items too slowly and it was duplicating work for the supermarket as products were being loaded onto shelves and then taken off minutes later by pickers who took them to load into order totes.

The solution for this was to move from a system that utilised human pickers navigating a human environment and instead move towards robots in an environment designed specifically for them that would allow much faster picking times and reduce work duplication. What this meant was huge warehouse spaces designed primarily for robots to navigate were constructed. Companies such as Tesco and Ocado have massive warehouses for this purpose. Doing this meant that items could be sent straight to these warehouses for packing instead of to the supermarket to be put on shelves then picked.

There have been several different approaches for how to design these robotic facilities for example Tesco have a hybrid model that delivers totes of items to a user to hand pick.[1] Whereas Ocado have a facility designed to have as little human robot interaction as possible.[2] There are pros and cons to each implementation, but this project will focus mostly on the robot-centric implementation.

2.2 Factors that go into warehouse design

This project is intended as a tool for the use in planning these large-scale picking facilities, and so it needs to consider the factors that have impacts on the process's efficiency. The one with the largest impact is the number of robots to be used on the factory floor. Having too many robots on the warehouse floor can cause large amounts of congestion within the model as robots block each other's movement and make finding clear paths to their goals very difficult, and on the other side having too few robots means that the advantages that come with swarm robotics are greatly lessened. What this visualisation can show is the optimum number of agents to use in a warehouse of a given size.

The size of the warehouse floor is in itself another consideration, larger sizes mean that more items could be stocked, and more robots can be used to pick customer orders. However, a consideration of having a large grid is how the items will be dropped off from the robot into the customer orders, having a large warehouse with few drop offs could lead to bottlenecks. In addition to this in a real-world scenario cost would massively increase with larger spaces, not just in building the facility but maintaining it as well.

Restrictions on the types of orders that can be placed is also another factor that would affect the speed at which the warehouse is able to process orders. For this visualisation the user has control over the number of unique types of items that can be in each order, along with the quantity of each that can be ordered. These can have impacts on the optimum warehouse size as if you only have a small number of unique types that can be ordered then having a smaller warehouse would be more efficient however if lots of different types of items are required then a large space would work better, equally in a real world scenario if item quantity was more common than robots might be able to be designed to pick up more than one item however that is not the case in this visualisation each robot can only pick one item at a time.

There is also perhaps the biggest factor which is what type of algorithm will be used to control the movement of the robots within the warehouse space. Having the robots simply knowing their current position and their goal position and moving towards where they need to be is easy to implement and can be fast with smaller warehouse sizes and smaller number of robots however as the number of agents increase so does the complexity that goes into route planning.

2.3 Existing Solutions

2.3.1 Optimal Solution – Conflict Based Search

CBS search is a common approach for multi agent pathfinding problems.[6] It works by using a path finding algorithm most commonly A* search with an appropriate heuristic then once it has the optimal routes for the agents to reach their goals it will evaluate the paths found against each other.

Figure 2.1 shows an example of a CBS tree with two agents attempting to path find to their goals. In the first node of the tree, Agent 1 and Agent 2 have their optimal paths however they have found there would be a collision at the 3rd node B2 which means that the routes are invalid. Since the routes are not valid then two new nodes are added each with a different constraint, each agent will then find their optimal path whilst obeying the node conditions, in this case again collisions are found so each lowest cost node splits again. This continues until the cost of the child nodes can't be reduced meaning a solution has been found.

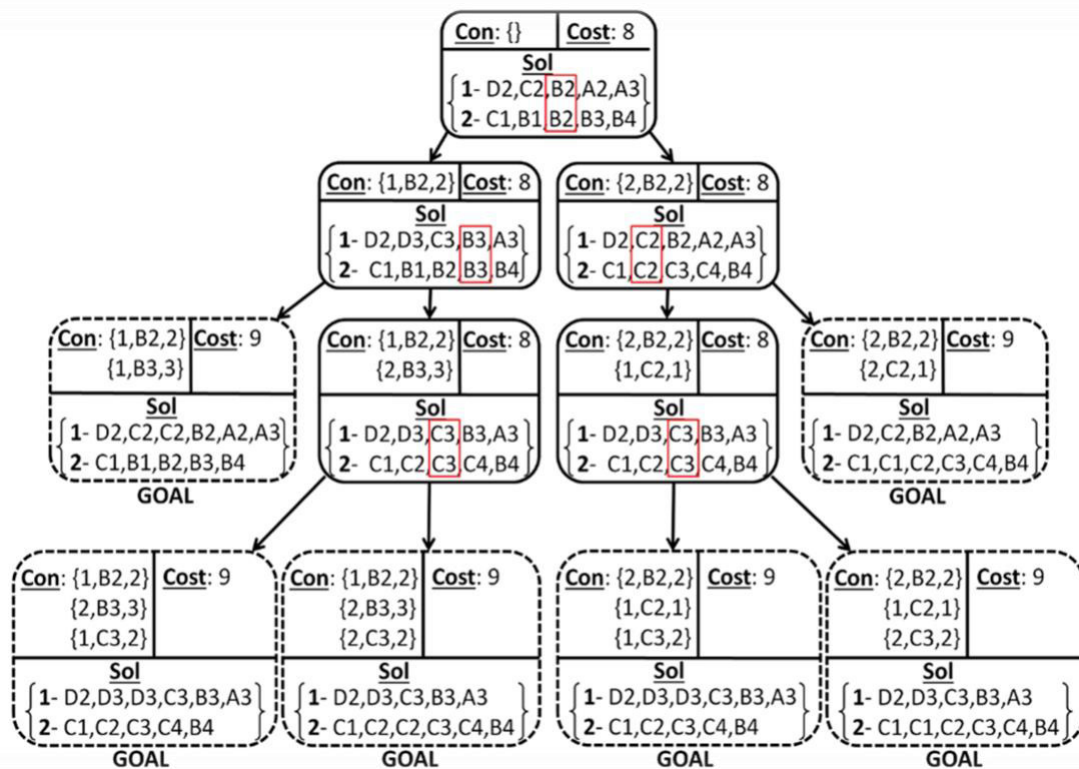


Figure 2.1 Constraint tree for example CBS problem [6]

In practice this CBS search algorithm has several limitations such as: The performance of CBS depends on the structure of the problem. Cases with bottlenecks are where CBS performs well, whereas in open spaces CBS performs poorly.[6] Figure 2.2 shows a comparison of different path finding algorithms and how their performance compared on different map types. The graphs are plotting the success rate of the algorithms against the number of agents used in the test. This shows how much the performance of the CBS algorithm was affected by the different map types.

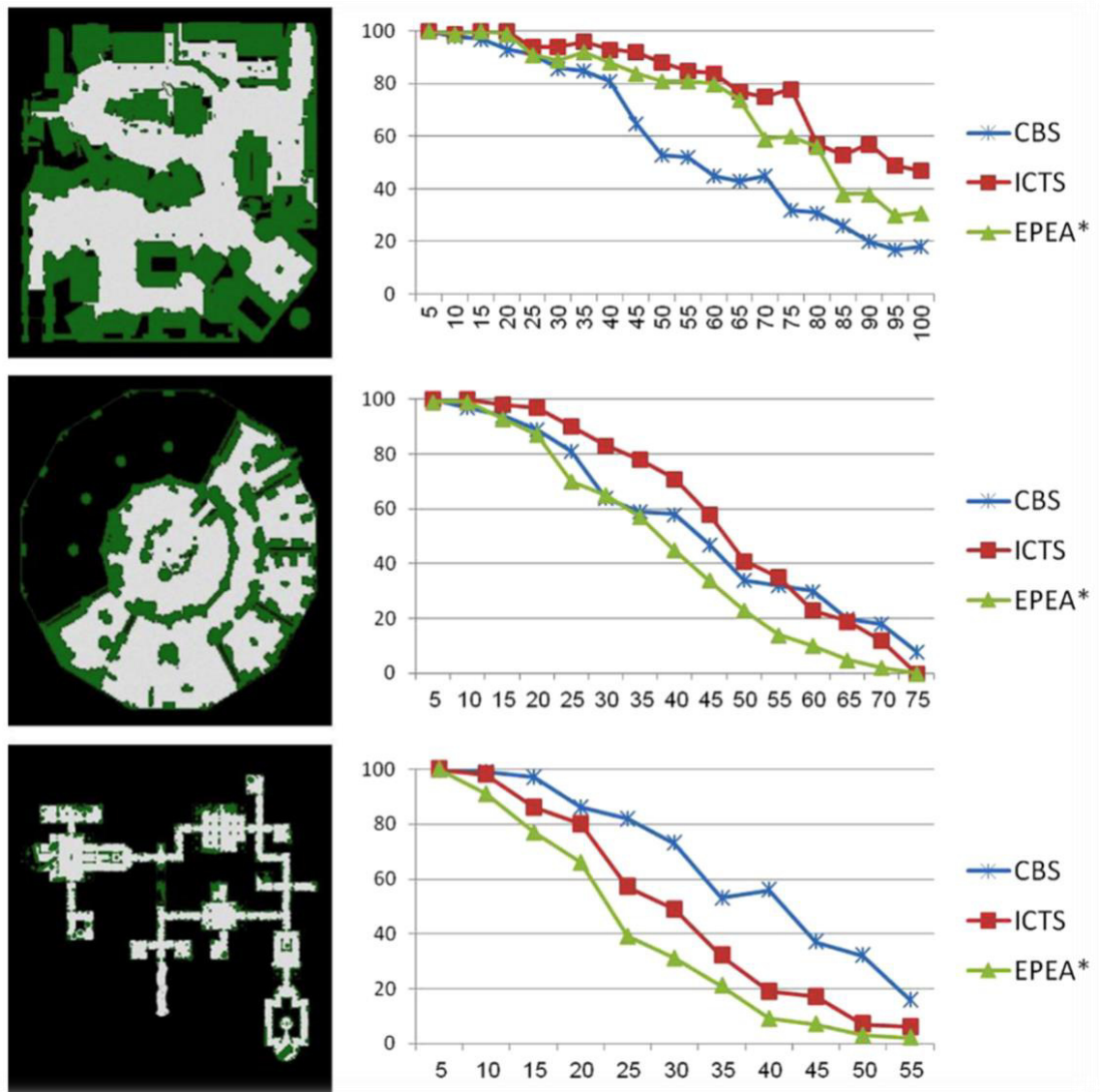


Figure 2.2. Comparison of path finding algorithms, plotting number of agents against success rate.[6]

The conclusions from the paper were that the CBS algorithm performed worst on the first map due to the large open spaces with comparatively few bottle necks, then the performance was middling on the second map as it still had wide open spaces, but this time had more bottlenecks. Finally, the third map is made up largely of narrow corridors and bottlenecks and the CBS algorithm outperformed the others by a decent margin.

Since in a warehouse environment there are very few bottle necks the CBS algorithm would not be the best to use. Also, it was in looking at how CBS worked finding optimal paths that I realised that this solution would not best fit for this project. In all of the examples given the agents were started in a randomised location and given a randomised goal to go to. This is similar to the needs of this system however a key difference is that in this example is that once a goal state is reached the robot has not finished its task. It would need to interact with the goal and then would be assigned a new goal to head to. This would mean the entire CBS tree would need to be created from scratch anytime a robot reached its goal. Also, with multiple robots filling customer orders it's possible that more than one robot could have the same goal which with CBS would mean that the path would never be found as the constraints would include the

goal. Having multiple goals for the robots to navigate to, makes it much harder to solve this problem optimally. This means that a non-optimal solution was much more appropriate for this implementation.

2.3.2 Non-Optimal Solution A* Search

A* search is one of the most commonly used path finding algorithms using heuristics to find optimal paths to goals.[9] “A* is complete and optimal on graphs that are locally finite where the heuristics are admissible and monotonic.”[10] An admissible heuristic is where the estimated cost will never be greater than the cost of reaching the goal state. Monotonic or consistency as it is sometimes referred to, means that the estimated cost of reaching the goal from a given node A is no greater than the cost of going from node A to node B plus the estimated cost of reaching the goal from B. Example of devising heuristics can be seen in Figure 2.3.

$$Hueristic(A) \leq Cost(A \rightarrow B) + Hueristic(B)$$

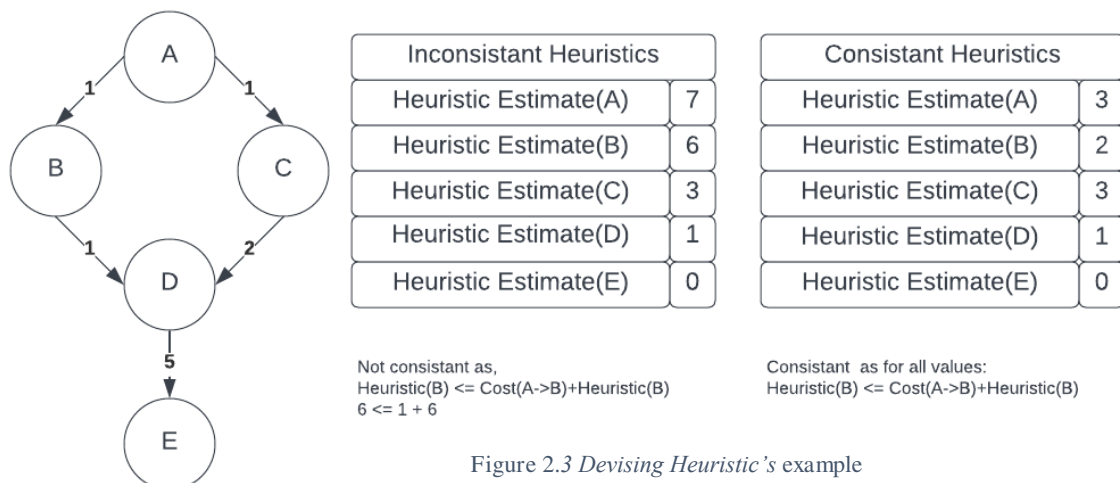


Figure 2.3 Devising Heuristic's example

There are several options for heuristics to use with A* search and they can have effects on the solution found. Some of the most widely used are Manhattan Distance, and Euclidean Distance.

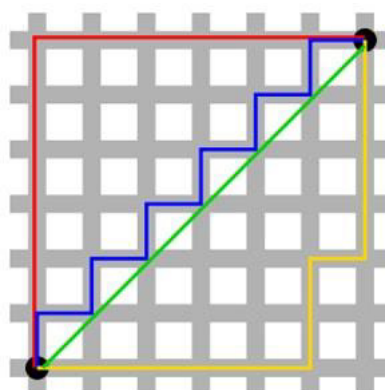


Figure 2.4 Differences between Manhattan and Euclidean Heuristics. [13]

The Modelling section is the base of the library, it contains the model and agent classes as well as the sub modules, meas.space and mesa.time, these modules are where the simulation environment is defined as well as the schedule that it runs on.[4]

The Analysis section is used for data collection whilst running the model and has built in support with the visualisation module for displaying the recorded data.

Finally, the Visualisation section of the module is the main visualization tool uses a small local web server to render the model in a browser, using JavaScript.[5] In my project I use all of these modules.

3. Specification and Design

3.1 The static architecture of the system

Figure 3.1 shows a UML diagram of part of the system. Figure 3.2 shows an example UML diagram of a MESA project [8] and so only the parts that are different to the example model are shown in Figure 3.1. The main difference between the example project and the system I have created is that multiple different agents are used in this project, these are the Robot workers, the Bin agents that control what item is held at which location, the drop off agents which are where items are deposited by the robot workers, the label agents which shows the dynamic order tracking on the right hand side of the grid, and finally the floor agent which are included to make it easier to tell the order tracking and warehouse floor apart.

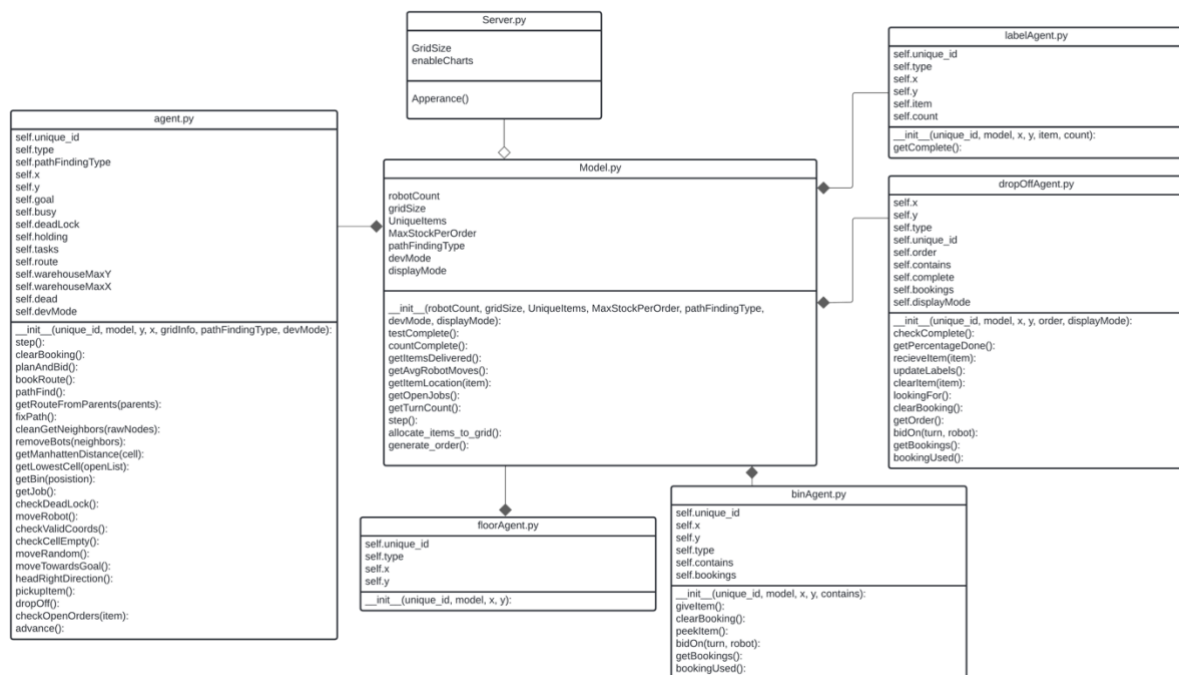


Figure 3.1 UML Diagram showing how the non-MESA classes interact.

The model class is initialised when the program starts, it takes two parameters from within the code itself, these are the size of the grid and whether to activate the charts for the visualisation. These two settings have to be initialised at the start of the program and cannot be changed while the program is running. However, all other settings about the model are set using sliders and

drop-down menus on the visualisation. When started the model uses the MultiGrid class from the MESA space module and begins by populating the grid with agents, to start with drop off agents are assigned into a specific column of the grid such that marks the furthest right the robots are able to go. Then if the visualisation mode is enabled the cells to the right of these drop offs will be populated with floor agents and then label agents matching the order currently being requested by the drop off cell. After that the model will place a bin agent in each cell of the grid with each agent assigned an item to store. Then the robot agents are placed into the grid, these are distributed randomly to a start location. Only the robot agents are added to the scheduler to activate every step, they trigger the activation of other agents when needed, for example when the robot drops an item off it the drop off agent will activate to test whether it has a completed order.

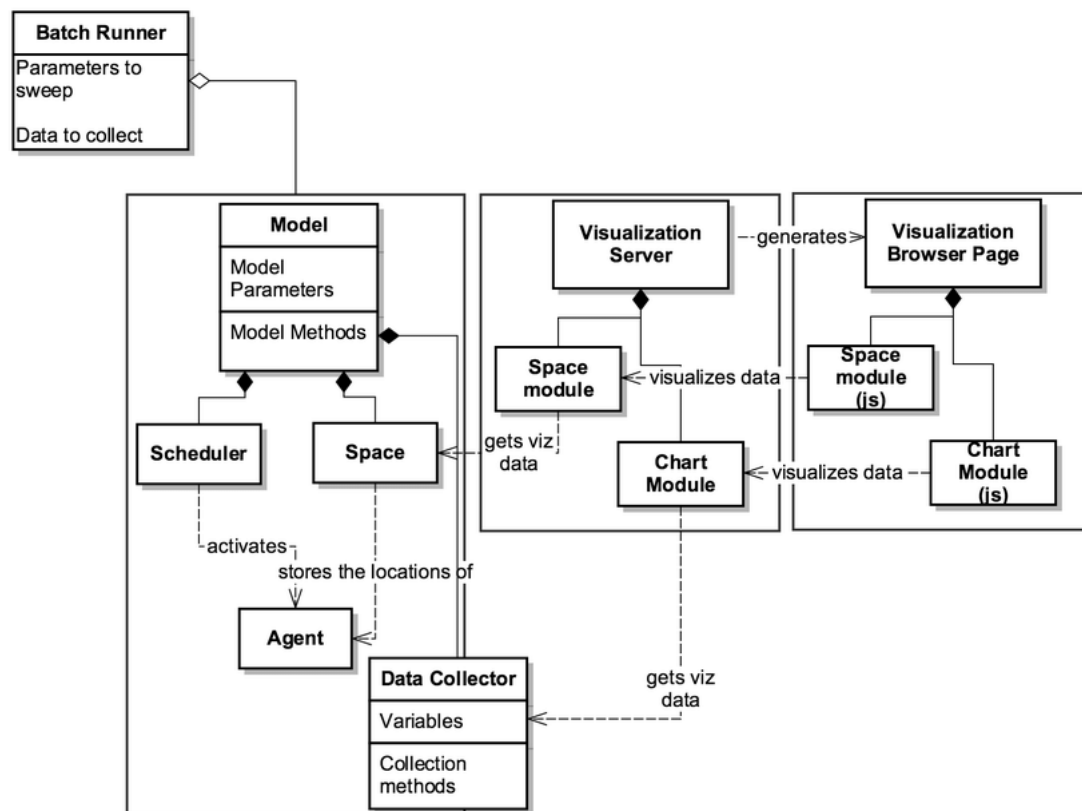


Figure 3.2 UML Diagram of a simple MESA Model [8]

While running at every step of the program data is being collected by the data collector module of the MESA library, three things are collected, the average number of steps taken by the robots in the system, the number of items that have been delivered to the drop off agents and the percentage of orders that have been successfully filled. This data is then fed to the MESA visualisation chart module which in turn feeds into the browser web page to be able to show the collected data as a graph on the page in real time.

3.2 The user interface

3.2.1 Model Control

The MESA visualisation module has built in buttons for controlling the model these are: Start, Stop, Step, Reset and About. The Start button starts the simulation and then is replaced with the stop button which in turn when clicked stops it. The Step button causes the model to allow

all agents to take a single step and the Reset button stops everything and reinitialises the simulation, this is required each time the user changes a setting. The About button shows a small dialogue box with a description of what the project is doing. Along with the buttons there is also a slider to control the speed at which the simulation runs. This is on a scale from 0 – 20, however setting the slider to 0 will disable the control and run the simulation as fast as possible.

3.2.1 User Changeable Settings

The user interface is most affected by two different modes, there is the Display mode which is shown as a checkbox that the user can toggle on for off while the program is running. And then there is also enableCharts, this is a setting inside of server.py and cannot be changed while the program is running, this is what controls if the data collected during the running of the visualisation is shown to the user or not.

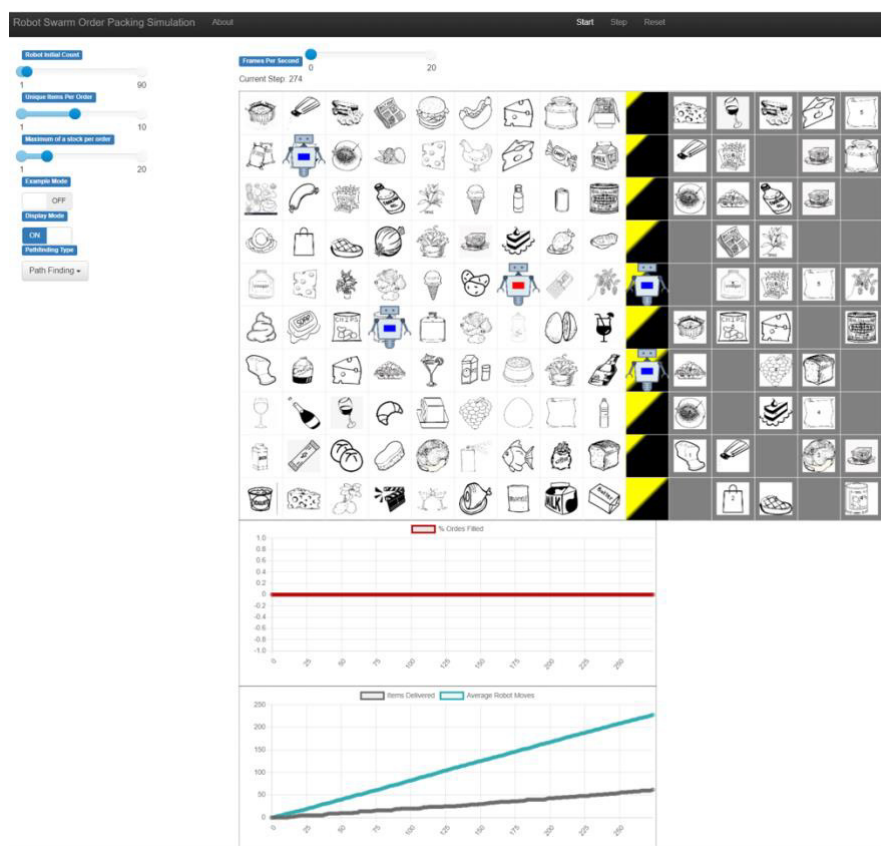


Figure 3.3 Screenshot of an example visualisation with display and chart mode enabled.

An example of the user interface can be seen in Figure 3.3, this is with both display mode active and with charts enabled. In Figure 3.4 you can see the user interaction section, this is where the user is able to change the settings that can affect the efficiency of the simulated warehouse.

The first slider is Robot Initial Count, this controls the number of robots that are loaded into the system at the start of the simulation. The upper limit is set by taking the height of the grid squared minus itself. This means it's impossible for a robot to be spawned into a drop off cell at the start of the visualisation, but the rest of the grid can be completely filled. Next is the Unique items per Order slider, this controls how many different types of items can be ordered,

this does not affect the quantity of items only the shopping list length. This slider starts full at the beginning of the program as it affects the length of the labels on the right-hand side of the grid. Because the grid size cannot be changed whilst the server is running the max size available on the slider needs to be used to ensure that if the user increased the slider, then the max value would fit. Once initialised the slider can be adjusted to any length the user would like to use. After that is the last slider, this is the maximum stock per order slider, this denotes the maximum amount of any one item that can be ordered in a single order, which combined with unique items per order give the user complete control over the example generated orders.

After the sliders are two toggles, these allow the user to activate settings to help understand how the robots work. The first setting is example mode. In this mode only one robot is able to move, this makes it easier to track its movements across the grid. The path that it has found to its next obstacle is also highlighted on the grid. The other robots are unable to move and act as obstacles, this is to show how the robots are able to navigate around the grid. The other toggle is for the display mode, this allows the user to deactivate the icons used in the visualisation and instead have a much simpler view of the robot movements on the grid. These two modes work well with each other to simplify the visualisation and can be seen in Figure 3.5.

The final setting the user can change is the path finding type they wish for the robots to use. There are two options, Blind Goal in which the robot has no real intelligence and will just move towards its goal each step with no planning, it has some deadlock detection and so given time it will be able to process all orders. The other type is Path Finding, this uses artificial intelligence to find routes through the grid.

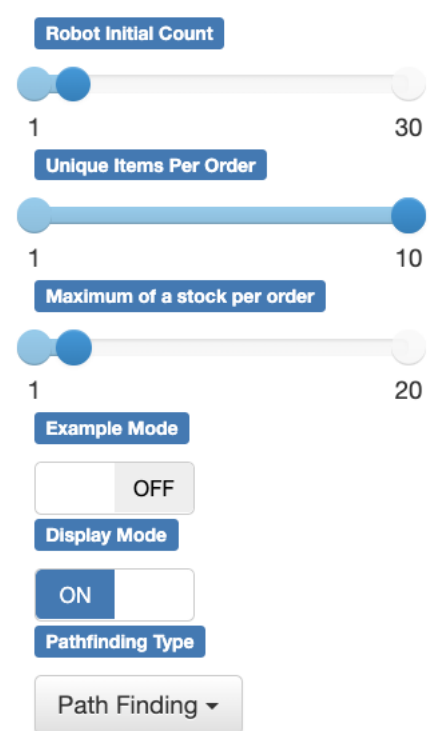
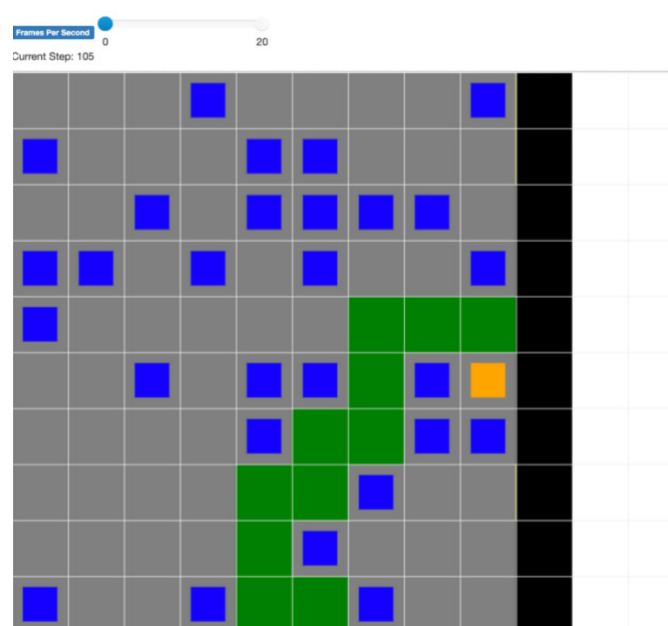


Figure 3.4 User Interaction



3.5 Simplified visualisation view with Display mode and enableCharts off and Example mode on

3.2.3 Warehouse Floor

The layout of the warehouse floor is based on the Ocado Hive factory an image of which can be seen in Figure 3.6 [14]. This factory layout was the main inspiration for this project. In the image you can see how each cell has totes loaded with items. This is shown in the visualisation by the image loaded onto each cell. Since the system has only a finite number of items with larger grid sizes some items will be held in multiple bins clustered together. As the robots move around the simulation, they also change colour to indicate what their current task is. Blue robots symbolise that they are currently not holding any item. Red robots means that the robot is currently carrying an item to a drop off point.



Figure 3.6 Ocado Hive robot packing facility.[13]

The cells on the grid that are filled in black at the beginning of the simulation are the drop off points, this can be seen in figure 3.5. They are the right-hand boundary of the simulated warehouse and in my implementation, they function in a similar way to the totes in that the robot drive over them and lower the picked-up item down into a tote collecting the order. As the order is filled, they begin to fill up like a progress bar to show the percentage of items that have been delivered to them. This can be seen in Figure 3.7. This give the viewer a better understanding of how far along the order is to being complete as it progresses not according to the types of items delivered but the proportion of items delivered.

3.2.4 Order labels

The labels that make up the right hand wide of the visualisation detail the outstanding items needed to complete the order help in their corresponding drop off cell. The labels contain an image of the items in the order and also a number indicating the amount of that item currently needed in the order. This is updated every time an item is dropped off by a robot to give a live view on how the order is progressing. Sometimes the numbers are not clear due to last minute icon changes, they appear more clearly on larger grid sizes.



Figure 3.7. Screenshot of the visualisation label section

4. Implementation

4.1 Initialising the model

4.1.1 Starting the server and initialising the model.

The program is started by running the file `Server.py`, this is where the settings that can't be changed while the simulation is running are set. When the program starts it uses these values and the default values from the user settings to create the grid object for the visualisation and also the model object. These setting can be seen in Figure 4.1 showing part of the `init` function for the model. If the display mode is enabled will change how the grid for the project is generated, to make room for the label agents.

```
def __init__(self, robotCount, gridSize, UniqueItems, MaxStockPerOrder, pathFindingType, devMode, displayMode):
    # Allows the model to continue to run.
    self.running = True

    # Number of robots in the warehouse
    self.num_agents = robotCount

    # Open jobs is the items currently needed, will be assigned later
    self.openJobs = []

    # The size of the warehouse floor.
    self.width = gridSize
    self.height = gridSize

    # If the display mode is on will create a longer grid to place the labels in
    self.widthWithOrderQueue = gridSize + UniqueItems

    # Creates the grid object from the MESA Space module
    if displayMode:
        self.grid = MultiGrid(self.widthWithOrderQueue, self.height, False)
    else:
        self.grid = MultiGrid(self.width, self.height, False)

    # Creates the scheduler to activate the agents
    self.schedule = RandomActivation(self)

    # Initialises the count that is used by the data collectors to plot the graphs
    self.turnCount = 0
    self.itemsDelivered = 0
    self.robotMoves = 0
```

Figure 4.1 Model Class `init()` function showing the parameters passed to generate the model.

Multi Grid

`MultiGrid` is an object from the MESA Space module[15], in this case it is initialised with the `Torus` parameter set to `False`, this means that the agents cannot wrap around from one side of the grid to another, if enabled this would allow for an agent to move from opposite sides of the grid in one move. For example, with a 4 by 4 grid an agent would be able to go from (1,1) to (4,1) in a single step. Figure 4.2 shows an example of how a torus grid works. This is not what is needed for this project and so is disabled. Also, the reason that `MultiGrid` is used is because while only one robot can occupy a cell at a time, there are also bin agents or drop off agents at every cell, so `MultiGrid` is needed.

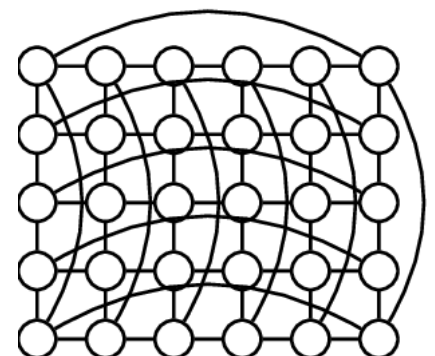


Figure 4.2 Example of how a Torus grid operates

Random Activation

The scheduler used for the model is random activation, every time the model is stepped forward the scheduler runs and any agents that have been added to the schedule have their step function run. For this project only the robot agents are added the schedule and they trigger the activation

of other agents when needed as most steps in the model they don't need to be activated. The reason that random activation is used instead of the alternative simultaneous activation is because it is easier for the path finding the robots use to avoid getting into deadlocks. Sometimes when using simultaneous activation situations could occur where a robot higher up in the schedule hierarchy would trap one that was lower. With random activation being used instead deadlock are much reduced as in a situation where one robot is blocking another, they are both equally likely to be the one to move first so even if a robot pair blocks each other a few times eventually the alternate one will activate first and escape the lock.

4.1.2 Initialising agents

```
for i in range(self.height):
    DropOffCell = DropOffPoint('Drop off point ' + str(i),
                               self,
                               x=self.width - 1,
                               y=i,
                               order=self.generate_order(UniqueItems, MaxStockPerOrder, numberOfCells),
                               displayMode=displayMode)

    self.grid.place_agent(DropOffCell, (DropOffCell.x, DropOffCell.y))

    if displayMode:
        # Adds in the labels on the right hand side of the grid based on the order in the dropoff.
        toLabel = DropOffCell.getOrder()

        for ix in range(UniqueItems):
            newFloor = Floor("Floor", self, x=self.width + ix, y=i)
            self.grid.place_agent(newFloor, (newFloor.x, newFloor.y))

        for ix in range(len(toLabel)):
            item, count = toLabel.popitem()
            labelAgent = Label(item, self, x=self.width + ix, y=i, item=item, count=count)
            self.grid.place_agent(labelAgent, (labelAgent.x, labelAgent.y))
```

Figure 4.3 Drop off, Label and Floor agent creation

The agents are created with the model and can be broken down into 3 steps, firstly shown in Figure 4.3 is where the drop off agents are created and placed into the grid object, then if display mode is activated then label agents and floor agents are created to fill in the area to the right of the drop off. Next the bin agents are created, this can be seen in Figure 4.4, this uses a function called `allocate_items_grid()` to create a list of grocery items from a sample list that will have one element for each bin agent to be added, as the bin agents are added to the grid, they are assigned an item to hold from this list.

```
# Adding a static agent to every cell, they allow mouseover information about what the cell is holding
GridContents = self.allocate_items_to_grid(((self.width * self.height) - self.height))
# Iterates over every cell in the grid
for Cellx in range(self.width - 1):
    for Celly in range(self.height):

        # The name of the cell it just the coordinates in the grid
        cellReference = (str(Cellx) + str(" ") + str(Celly))
        # Creates a new agent to sit in the cell as a marker
        newCell = Bin(cellReference, self, x=Cellx, y=Celly, contains=[GridContents.pop()])
        # Places the cell agent into their place in the grid
        self.grid.place_agent(newCell, (newCell.x, newCell.y))
```

Figure 4.4 Bin Agent creation

Finally, the robot agents are then added to the model, this can be seen in Figure 4.5. To do this, random coordinates in the grid are chosen as a potential start location, if it's confirmed that there is currently no agent located in that grid cell then the robot will be placed there and added to the schedule. The next part is for when the simulation is used in example mode. For this mode all robots except for one are used as static obstacles to show the agent path finding

through the grid, however because these robots will be static the cells that they spawn on can never be visited by the one working agent so all items that a robot is spawned on needs to be removed from any open orders. This is not a perfect solution as with a large number of agents in example mode there might be cells that are fully encapsulated by static robots and so can never be visited but equally have no robot on them, so their item was not removed.

```
# Creating the Robots
for i in range(self.num_agents):
    # Creates the robots starting at random points on the warehouse floor.
    # Loop to create the new robots and to add them into a grid cell that is empty
    while True:
        x = self.random.randint(0, self.width - 2)
        y = self.random.randint(0, self.height - 1)
        # When the random values find an empty grid cell break the loop and place the robot
        if len(self.grid.get_cell_list_contents((x, y))) == 1:
            break

    newRobot = Robot(i,
                    self,
                    y=y,
                    x=x,
                    gridInfo=[self.height, self.width],
                    pathFindingType=pathFindingType,
                    devMode=devMode)

    # Adds the new robot to the scheduler
    self.schedule.add(newRobot)

    # If in example mode then robots that don't move need to have items they start
    # on top of removed from all the orders.
    if devMode:
        cell = self.grid.get_cell_list_contents((newRobot.x, newRobot.y))[0]
        toRemove = cell.peekItem()
        for Celly in range(self.height):
            gridCell = self.grid.get_cell_list_contents((self.width - 1, Celly))[0]
            if toRemove in gridCell.order:
                gridCell.order.pop(toRemove)

    # Adds the robot to the grid according to its starting coordinates
    self.grid.place_agent(newRobot, (newRobot.x, newRobot.y))
```

Figure 4.5 Robot Agent creation and placement

4.1.4 Starting Data collector

```
#Starting the data collectors
self.datacollector = DataCollector({
    "% Ordes Filled": lambda m: self.countComplete(),
    "Items Delivered": lambda m: self.getItemsDelivered(),
    "Average Robot Moves": lambda m: self.getAvgRobotMoves()})
```

Figure 4.6 Data Collector creation

The data collector is setup in the initialisation of the model and so each time the model takes a step it will run the countComplete, getItemsDevlivered and getAvgRobotMoves functions to get the data it needs then it will add the data point to the charts, plotting the number of items delivered and the average steps taken on the same chart and the percentage of orders filled separately. All of these data points are plotted against the step the model is currently on.

4.2 Stepping the model

4.2.1 Blind Goal Step

The way the agent operates depends mostly on what path finding type they are using. When the agent is set to “Blind Goal” the step function will run the code seen in figure 4.7 it will use not use an intelligent path finding algorithm, it will instead just use the function `moveTowardsGoal` seen in Figure 4.8 to “blindly” head towards its goal. Before running that function however, it will check its deadlock counter.

This is used to detect if the robot has moved since its last step. If it hasn't that could mean that two robots are blocking each other which without deadlock detection would never end. Every turn the robot doesn't move its counter decreases by one, if the robot's counter reaches zero then instead of using the `moveTowardsGoal` function it will try to move randomly to break the deadlock. With this system the blind goal pathfinding will always be able to complete a simulation, but it can take an extremely long time due to robots all trying to move the same way causing large clumps of robots to form taking a long time for the robots to escape.

In the `moveTowardsGoal` function first the system checks if the goal has been found if so, depending on the agent at the goal location it will either pick up or drop off an item. If not at the goal it will run the `headRightDirection` function also seen in Figure 4.8, in this the agent will find out how far it needs to go along each axis to reach its goal and then it will randomly pick a direction to go in, the more movement required in a particular direction the more likely it is to choose that direction to go in.

```
if self.pathFindingType == 'Blind Goal':

    if self.deadLock <= 0:
        # print('deadlock detected')
        self.moveRandom()

    elif self.busy:
        self.moveTowardsGoal()

    else:
        self.getJob()
        # print('picking up a Job:', self.goal)
        self.moveTowardsGoal()

    self.checkValidCoords()
    self.checkCellEmpty()
    self.checkDeadLock()
    self.moveRobot()
```

Figure 4.7 Blind Goal Mode

```
def moveTowardsGoal(self):
    # print('trying to get to goal: ', self.goal, 'from ', self.x, self.y)
    if self.pos == self.goal:
        if self.model.grid.get_cell_list_contents(self.pos)[0].type == 'DropOff':
            self.dropOff()
            # print('item dropped off at goal, deleting goal')
            self.goal = None
            self.busy = False
        else:
            hovering_over = self.model.grid.get_cell_list_contents(self.pos)[0].peekItem()
            # print('Found current Job item', hovering_over)
            self.checkOpenOrders(hovering_over)
    else:
        self.headRightDirection()

def headRightDirection(self):
    directionVec = (abs(self.goal[0] - self.x), abs(self.goal[1] - self.y))
    # print(self.goal)
    # print(self.pos)
    prob_x = directionVec[0] / (directionVec[0] + directionVec[1])
    prob_y = directionVec[1] / (directionVec[0] + directionVec[1])
    # print(prob_x, prob_y)
    direction = self.model.random.choices([True, False], (prob_x, prob_y))[0]
    # print(direction)

    if direction:
        if self.goal[0] > self.x:
            self.x += 1
        elif self.goal[0] < self.x:
            self.x -= 1
    else:
        if self.goal[1] > self.y:
            self.y += 1
        elif self.goal[1] < self.y:
            self.y -= 1
```

Figure 4.8 How blind goal movement works.

4.2.2 Path Finding Step

A typical step for a robot with path finding starts with checking to see if the robot has marked itself as busy, if it not busy then it will get a job from a list kept by the model about items that are still missing from open orders, it will then plan a route for getting to its goal. If the robot is busy but doesn't have a route that means that something has gone wrong in the path finding process, this could be that a previous planned route would have led to a collision and so the move was cancelled and instead a new path needs to be planned, or it could be that the goal node is not found by the A* search algorithm. This can either be because there is currently a robot on or surrounding the goal cell. And so, because a full route cannot be created the robot just moves to a neighbour cell with the best heuristic not planning a full path. It then marks that route as false so next step it can try to plan a route again.

If the robot is marked busy and the route is empty that means that the robot has reached its goal. If the goal is a drop off point, then it will drop the item that it is holding into it and mark itself as no longer busy. If it is a bin then it will double check the item is still needed before picking it up, getting a new goal and planning a route.

```
elif self.pathFindingType == 'Path Finding':  
    # print('----- Robot Info -----')  
    # print(self.pos, self.goal, self.unique_id, self.holding, self.route, self.busy)  
    # print('-----')  
  
    if not self.busy:  
        self.getJob()  
        self.planAndBid()  
    else:  
        if self.route is False:  
            self.route = []  
            self.planAndBid()  
        else:  
            # print('Following Route:', self.route)  
  
            if self.route == []:  
                # print('goal found', self.goal)  
                # This section should be made into a function as it's universal to both  
                cell = self.getBin(self.pos)  
                if cell.type == 'DropOff':  
                    if self.holding == []:  
                        # print('trapped in drop off')  
                        None  
                    else:  
                        self.dropOff()  
                        self.goal = None  
                        self.busy = False  
                else:  
                    hovering_over = cell.peekItem()  
  
                    if self.checkOpenOrders(hovering_over):  
                        self.planAndBid()  
                    else:  
                        self.busy = False  
                # Until here  
            else:  
                # print('route:', self.route)  
                next_x, next_y = self.route.pop(0)  
                self.checkNextCellInRoute(next_x, next_y)  
                self.moveRobot()  
                # self.clearBooking()
```

Figure 4.9 Path Finding Control

If none of these apply, then it will simply pop the next node off of its planned route, check if it is clear then move to that next cell. If the next cell is not clear however it will cancel the route and either wait a step where it is or move the robot to the best next available cell based on the heuristics of the cell it's on as well as the surrounding cells.

4.2.2 Planning a route

The path finding algorithm uses A* to find a route from the robot's current position to the goal cell. It ignores nodes that contain a robot and marks them as non-traversable. It uses Manhattan distance as the heuristic for exploring child nodes and will find the optimal path to the goal node.

```
def pathFind(self):
    # print('pathfinding from %s to %s' %(self.pos, self.goal))
    openList = {}
    closedList = {}
    parents = {}

    openList.update({self.pos: 0})
    while len(openList) > 0:
        node = self.getLowestCell(openList)
        cost = openList.pop(node)
        if node == self.goal:
            # print('Goal node found')
            return parents
        childNodes = self.removeBots(self.model.grid.get_neighbors(pos=node, moore=False))
        # childNodes = self.filterAvaliable(childNodes)
        # print('Looking at node', node)
        for childNode in childNodes:
            # print('Looking at child node', childNode)
            if childNode == self.goal:
                # print("Goal node found in the loop")
                parents.update({self.goal: node})
                return parents
            g = cost + 1
            h = self.getManhattanDistance(childNode)
            f_cost = g + h
            # print('child cost', f_cost)
            if childNode in openList:
                if cost < f_cost:
                    continue
            elif childNode in closedList:
                if cost < f_cost:
                    continue
            else:
                openList.update({childNode: f_cost})
                parents.update({childNode: node})
            closedList.update({node: cost})
    return parents
```

Figure 4.91 A* Path finding Algorithm

This one part of the project took by far the longest to implement and right up until the end was where most of my focus was placed on. I had multiple different plans or solutions to try and make this non optimal solution as efficient as possible. I had one idea of using a booking where a robot would find their optimal route and then place a booking into each cell, they would use along with the turn they would use it on. Then when the next robot started to path find it would not just avoid cells with robots currently on it would instead test to see what turn an unexplored node would be reached on and then see another agent had booked it for that turn in which case it would mark it as unavailable. This however proved to be much more difficult than I had planned and despite a large amount of time working on it I could not get it to function as intended. There are a lot of left-over features in the code from working on these implementations. Figure 4.92 shows some of my plans for how this implementation would work. The mouse over information shows details about the cells, in the figure you can see that two different robots have made booking for cell (2, 4) for different turns. The way I had planned for this system to work was to use A* search to find the optimal path from the robot's start to

Current Step: 23

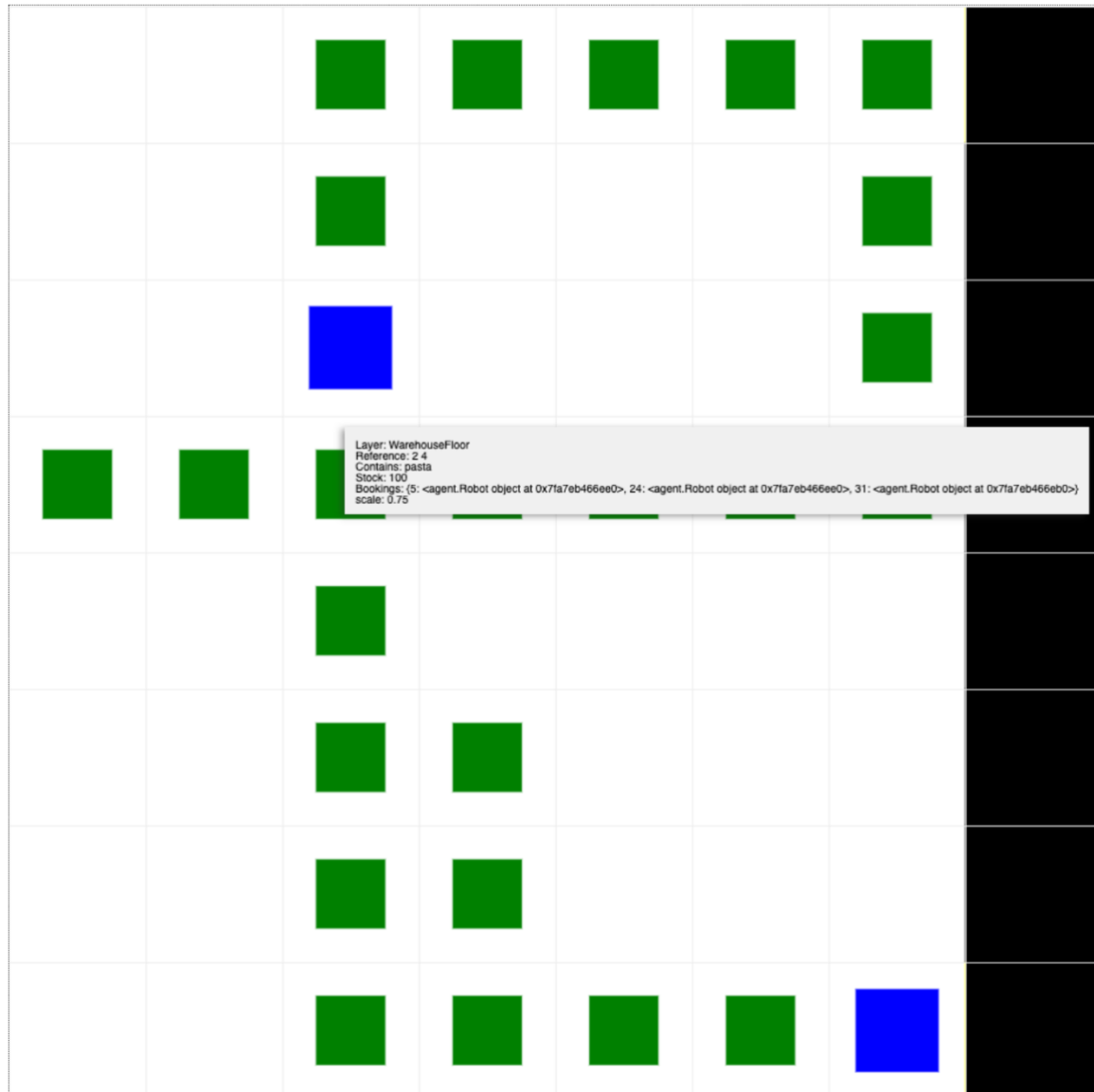


Figure 4.92 Plans for cell booking system

its goal then when it had this route place a booking on the cell for the turn it would arrive on that cell, on the turn (Model's current step + the cells position in the route). In the figure you can see the model's current step is at 23 and the robot is at cell (2,5) and so it placed a booking on (2,4) for turn 24. As you can see from the figure this was working however the issues arose in trying to modify the A* algorithm to work out the route cost for each node that was explored. Every time the lowest code node was taken from the open list and its children looked at the algorithm would work out the steps taken to get there if this node was used in a route and then check for current open bookings for that cell.

While I do think that this algorithm would have worked it proved to be too much for me to implement in the time scale remaining for the project. In the end the solution that I ended up using is not ideal. When there are clashes in the paths then the route is not followed instead the fixPath function is run, this function as seen in Figure 4.93 takes all of the cells around the robot that are valid cells to move to and that are unoccupied, it will then work out the heuristic value for each cell and then move to the one with the lowest value.

```

def fixPath(self):
    neighborCells = self.model.grid.get_neighborhood(self.pos, False, True)
    validCells = []
    for cell in neighborCells:
        gridCellInfo = self.model.grid.get_cell_list_contents(cell)
        if len(gridCellInfo) == 1:
            if gridCellInfo[0].type in ['DropOff', 'Bin']:
                validCells.append(gridCellInfo[0])

    cellCosts = {}
    for i in validCells:
        cellCosts.update({i.pos: self.getManhattanDistance(i.pos)})
    if len(cellCosts) > 0:
        bestOption = min(cellCosts, key=cellCosts.get)
    else:
        bestOption = self.pos
    self.x, self.y = bestOption
    self.moveRobot()

```

Figure 4.93 fixPath Function

4.3.3 Picking up and dropping items

Picking up items and dropping them off is very similar with both path finding types, Blind Goal's approach can be seen in Figure 4.8 with the moveTowards function. When it takes a step, before moving it checks to see if it is currently at its goal, if the goal is a bin, then it will pick an item up out of it and then if it is a drop off cell it will drop whatever it is holding down into it. The path finding approach can be seen in Figure 4.9 and acts similarly in that it will check what type of agent it is in a cell with and drop its item off, then mark itself as not busy and clear its current route. If the cell is a bin cell, then it will peek at the item in the cell, double check that the item is still needed by an order then pick it up, then it will update its route to head towards its new goal.

4.3.4 Collecting Data

The data that is collected for the graphs is done using in DataCollector object from the MESA Library, Figure 4.6 shows the data collector being initialised using the three functions shown in Figure 4.94. The countComplete function iterates over the drop-off agents in the system and then works out the percentage of them that have all the items they need. The other two functions are based on a model variable that is incremented every time a robot drops an item off or moves respectively. The number of moves is then averages between all robots before being returned.

```

def countComplete(self):
    count = 0
    for i in range(self.height):
        if self.grid.get_cell_list_contents((self.width - 1, i))[0].checkComplete():
            count += 1
    return (count / self.height) * 100

def getItemsDelivered(self):
    return self.itemsDelivered

def getAvgRobotMoves(self):
    averageStepsPerRobot = self.robotMoves // self.num_agents
    # print(averageStepsPerRobot)
    return averageStepsPerRobot

```

Figure 4.94 the functions used by the data collector for the graph visualisation

4.3.5 Building the Visualisation

When building the visualisation part of this project there is an established way off setting up the appearance of the agents within the MESA library, when the CanvasGrid object is created it is passed a `portrayal_method` parameter, this is a function that takes an agent object and then returns a dictionary containing information about how the agent should look in the visualisation. This dictionary is then converted to a JSON file and given to the MESA visualisation browser page to be displayed.

The `portrayal_method` in this project is a function called `Appearance()`. With this project there are a large number of agents to consider on the grid and each one needs to look different. The visualisation browser has a number of options for how the agents can appear, from the shape and size of the agents, colour, text overlay, whether to use an image and some others not used in this project. The `Appearance` function checks to see what type of agent is being looked at and then will call another appearance function based on the agent type, this is shown in Figure 4.95. Once the correct appearance function is called then it will evaluate the current model settings like if the display mode is enabled or if the agent is holding an item. In Figure 4.95 the `robotAppearance` function is shown and you can see how based on if the model is in `displayMode` it will either set the robot shape to a rectangle or will load the robot avatar image to be displayed.

```
def Appearance(agent):
    if agent.type == "Robot":
        return robotAppearance(agent)

    elif agent.type == "Bin":
        return binAppearance(agent)

    elif agent.type == "Floor":
        return floorAppearance(agent)

    elif agent.type == "Label":
        return labelAppearance(agent)

    elif agent.type == "DropOff":
        return dropOffAppearance(agent)

def robotAppearance(agent):
    if model_params["displayMode"].value:
        if agent.holding == []:
            robotImage = 'resources/Robot.png'
        else:
            robotImage = 'resources/Robot Busy.png'
            robotLayer = 'Interaction'
        else:
            robotImage = 'rect'
            robotLayer = 'WarehouseFloor'

    if agent.holding == []:
        robotColor = "Blue"
    else:
        robotColor = "Red"

    if model_params["devMode"].value and agent.unique_id == 0:
        robotColor = "Orange"

    portrayal = {"Shape": robotImage,
                 "Filled": "true",
                 "Color": robotColor,
                 "Layer": robotLayer,
                 "Number": agent.unique_id,
                 "Carrying": agent.holding,
                 "w": 0.6,
                 "h": 0.6}

    return portrayal
```

Figure 4.95 Appearance Function and `robotAppearance` function showing how the visualisation of agents works.

There are some other agents that have further functions to determine their appearance. The colour value in the dictionary can display gradient colours and so the agent has a routine to work out how many items have been delivered to it and how many it is waiting to receive; it will then use this to make a gradient colour array for the portrayal to show the order progress.

When gathering the images for the bin and label agents I started by gathering black and white images for the icons, then when realising this was not clear enough, I then got colour images. However, there was a further issue with this as many of the images had full colour backgrounds that when scaled down made it hard to see both the item and also the grid dividing lines. Once again, I then gathered images with a white background this was definitely the clearest version of the visualisation however I then realised to be used in the project the images would need to be from an open-source image repository. Due to the nature of using products as icon I found it very hard to find any open-source images that were what I was looking for. Because of this I had to revert from using photos to using fair use clipart from[16]. After I had gathered the images that I needed I used a batch resizer to change the resolution of all of the images to a 30x30 pixel size to make it easier for the visualisation to load the images, for this I used the website[17]. However, there were still some issues with this implementation and can cause the visualisation to flicker when the display mode is enabled, especially with larger grid sizes and frame rates. I tried to further reduce the image quality to reduce this, but this didn't seem to improve the issue, so I reverted this leaving the images at their original size. This screen flicker is something I mention in the further work section of the report.

5. Results and Evaluation

5.1 Evaluating the import outcomes of the project.

In the introduction of this report, I outlined several key outcomes that the project needed to be able to demonstrate.

- Create a working visualisation of a robotic warehouse floor
 - The visualisation should be clear and easy to understand even to people without a technical background.
 - The visualisation needs to clearly show how the robots are operating in the grid.

I feel that the project that I have produced meets these required outcomes. The simulation works and clearly shows how robot workers can be utilised to pack customer orders. The visualisation is easy to follow with appropriate images used to show the items held at each location and also what items are needed at each drop off. The added example mode further serves to help users understand how the robots work at an individual level managing to move around the environment finding routes through obstacles. I think that someone without a technical background would be able to understand the visualisation and also be able to understand the user parameters to change some of the settings.

To show this I included the example mode on the project to demonstrate how the robot is able to path find around the obstacles in the grid. Using an example 8x8 grid with random seed of 1, with the settings of 16 agents, 8 items per order with a stock of 1, I generated this example map of a warehouse floor with the static robots acting as obstacles. This is a chosen example due to problems with the example mode map generation, if there are any inaccessible nodes without a robot on them then this can result in the map being no completable by the robot. This is tested and discussed further in this evaluation section.

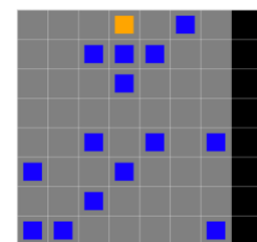


Figure 5.1 Example mode, randomly generated grid.

From this starting state I then run the visualisation and in Figure 5.2 you can see screenshots of the map's appearance at 10 step intervals to show the progress the robot is making. Having the example mode really shows how the robot is able to plan routes to the users and the green line showing this route also gives the user a way of tracking the current objectives of the robot.

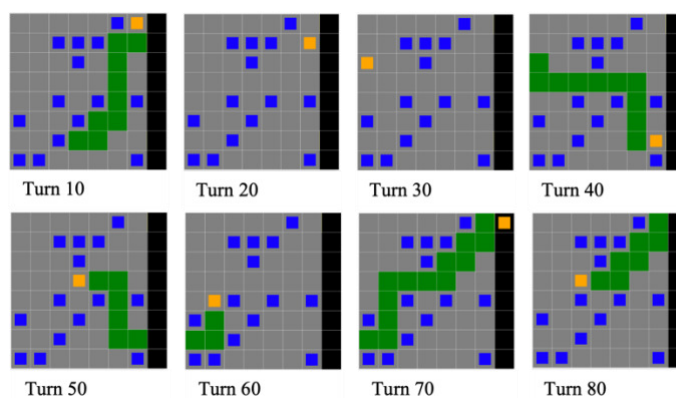


Figure 5.2 Steps of the Example Mode Path finding mode

The example mode also works with the other path finding mode, Blind Goal however since that mode doesn't have any path finding capabilities it will very rarely be able to traverse the map with obstacles. In the example shown above, it will be stuck forever in the position shown in Figure 5.3.

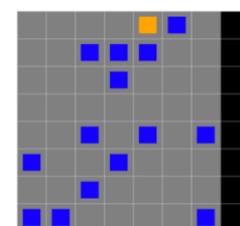


Figure 5.3 Stuck position

- Have robots that are able to process customer orders
 - The Robots should be able to fulfil any customer orders generated by the system unless the setup parameters make this impossible.
 - There should be at least two modes of operation for the robots with one utilising artificial intelligence to better move the robots through the warehouse.

The robots in the simulation are able to process customer orders however they are not able to fulfil every single variation generated by the system. There are several factors in the setup process that can mean that the grid generated by the system is not solvable. This is mostly generating a map in the example mode. Example mode works by making all robot agents apart from one static, they only exist in the grid to block the other robot's path. In doing so the bins that they are generated on top of can never be reached by the robot. I have made attempts to reduce this problem, in example mode when a robot is placed into the grid the item that it is placed on is removed from all outstanding orders. This can sometimes mean that the simulation starts with some orders already filled. The problem however arises with cells that while not having a robot on top of them are unreachable for the robot that is moving. This happens when the cell is surrounded by spawned-in robots, either closely or sometimes at a distance. When this happens the item that it holds is not removed from the orders and so in this case that order is impossible to fill and so the whole grid becomes unsolvable. Figure 5.4 shows an example of an unsolvable grid. With the problem cell highlighted with a red box. This square is empty, but it can never be reached by the robot.

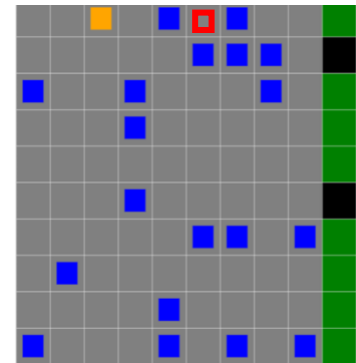


Figure 5.4 Unsolvable Example mode grid.

I tested how often the example mode would generate a grid that was unsolvable. Testing first on a 10x10 grid with incrementing random seeds and with different values for the unique items per order. The data for this test is in table 6 in the appendix and the number of agents at which the grid was no longer solvable for each of the random seeds that were used.

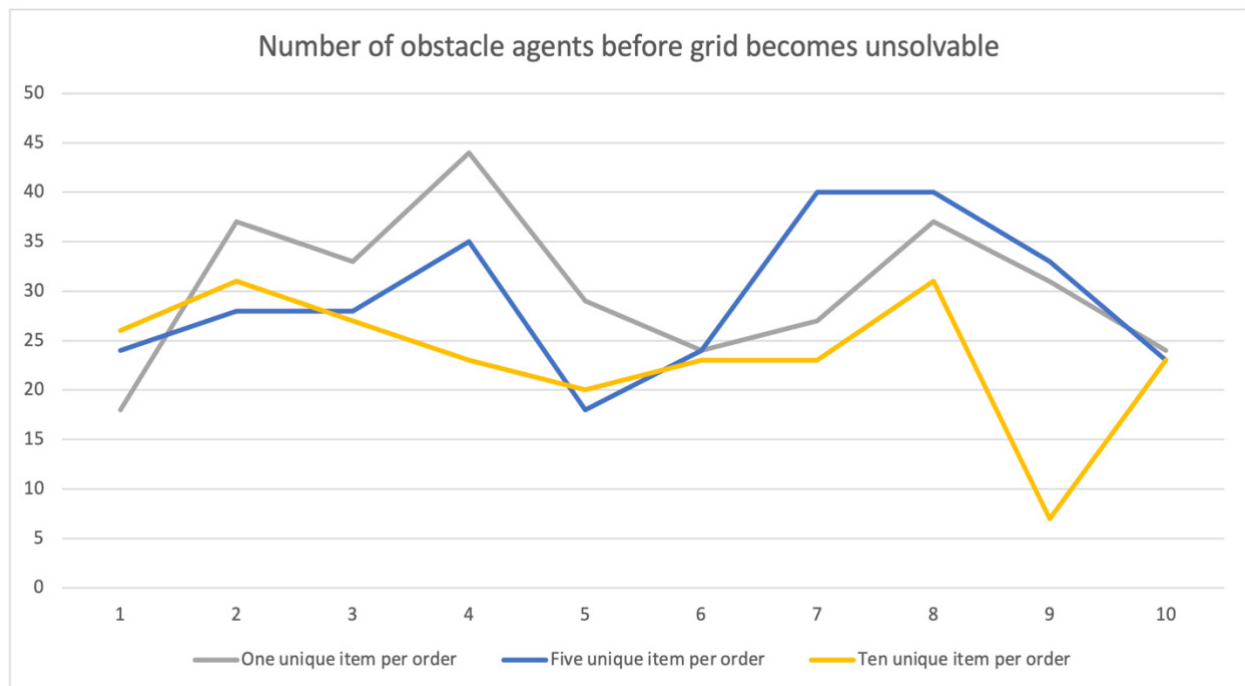


Figure 5.5 Graph showing the data from table 6, plotting the number of robots that made a grid unsolvable.

From this graph we can see that for the most part the more unique items per order the less agents it takes to make the grid unsolvable. This makes sense as the more unique items that are selected across all of the orders, the more likely that it is that one of those items will be in a trapped cell. In the process there were many times when using few unique items per order trapped cells would appear in the grid but as the item didn't appear in any order the grid remained solvable.

There are other issues that can also affect whether the grid is feasible to be solved. For example, grid size and agent number. For larger grid sizes the runtime it takes to solve the grid increases quickly and while the grid remains solvable it becomes unreasonable for a user to run the simulation for the length of time it would take to solve. In table 6 of the appendix, you can see the results for loading up grids of various sizes and testing to see how many steps it took for a single agent to solve. This is also shown as a graph in Figure 5.6

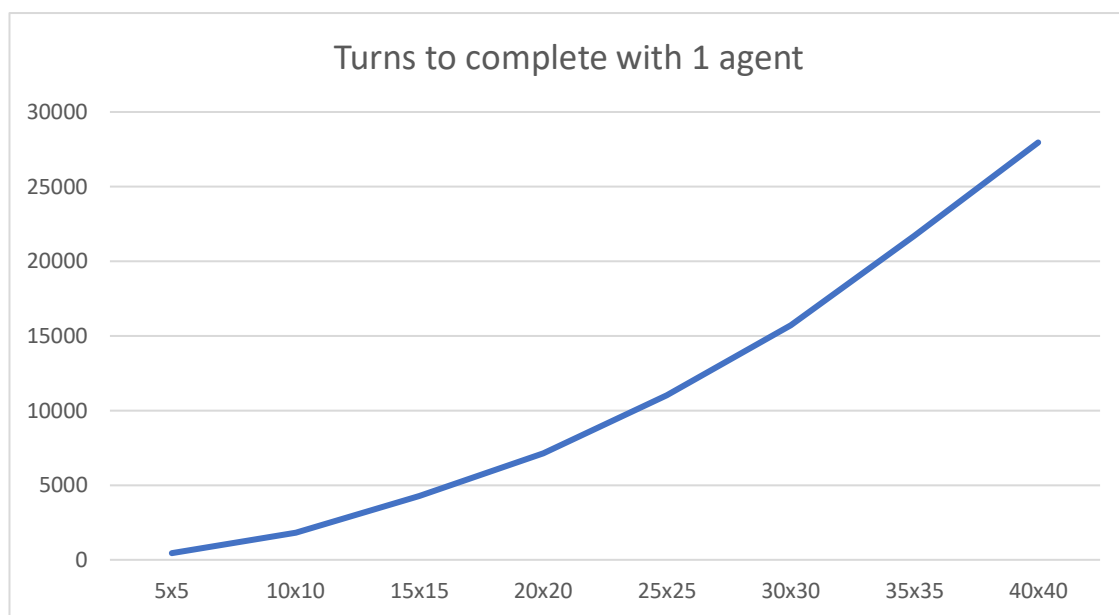


Figure 5.6 Graph showing the turns taken for a single agent to

The performance of the system is also affected by how many agents are loaded onto the grid; the MESA library has a built-in step limit that large numbers of agents can exceed on a large grid. I have included a point about this in the future work section of the report about a warning to users that a particular setup may take a long time to finish.

The other important outcome here was to have two modes of operation for the robots to traverse the map using. I have implemented two different techniques for this Path finding and Blind goal, and so to test them against each other and demonstrate their capabilities, robots using each type of path finding will fill out the same set of orders, once using the path finding algorithm and once using the blind goal approach. This test would then be repeated with increasing size grids and increasing number of robots. In the appendix tables 1 to 5 show the results for the number of turns taken for the simulation to finish running when packing orders on five different grid sizes. These sizes were a 3x3, 4x4, 5x5, 10x10 and 20x20 grid. This is to show how the different algorithms can affect the performance of the system as highlights some strengths and weaknesses in both. All of these tests were done using the random seed of one and all with the same order constraints where the unique items per order was set to ten and but the stock per item limit was set to one.

Figure 5.1 shows the performance of increasing number of robots to fulfil outstanding orders, the first three graphs have all the values of the number of robots used due to the small map size however as the size of the grids increased instead increments of five was used for the number of robots. The first key takeaway from the graphs is how the relationship between the two techniques acts with a low number of robots, there is a very close relationship between the results for both the blind goal and path finding robots. With the low number of robots there are fewer “crushes” this is where robots are trying to move but are surrounding on all sides by other robots. Another interesting thing shown in these graphs is these crushes take longer to form with large sized graphs but also that there is a very immediate change. On the smaller grids the two types of robots remain comparable for most numbers of robots however in the large sizes the 10x10 and the 20x20 the blind goal implementation becomes much worse compared to its path finding counterpart. This really emphasises how much the different types of algorithms used to move the robots can affect the performance of the whole system.

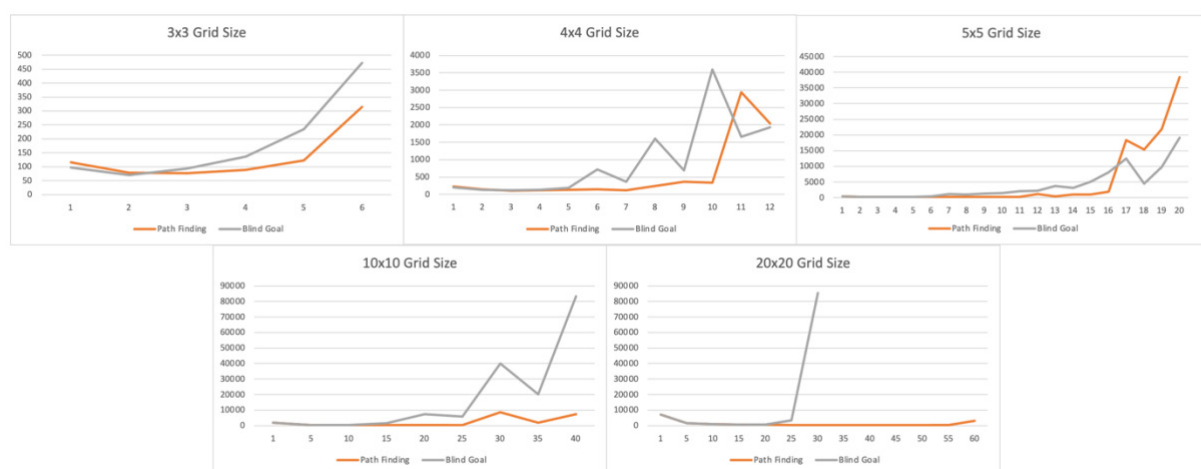


Figure 5.7 Charts plotting the amount of turns it took the system to finish against the number of agents used on different grid sizes.

Another insight from this was how many steps it took the system to approach 100% complete. The way that the system is currently setup there are no new orders coming in and so as the simulation begins to clear the orders this leaves more and more robots without jobs to be completing which in turn can cause crushes as they are not moving around the whole grid. This definitely has an impact on performance of the system and can be clearly seen in figure 5.2 which is a screenshot of the system completing the 20x20 grid test with 60 agents. In the graphs you can see that it reached 95% of orders complete by turn 380 and yet left running this task will take an additional 2647 turns to complete.

In the lower graph in the figure, you can see that the items delivered graph and average robot moves increases at about the same rate when the simulation starts however at the same time as the system reaches 95% complete the number of items delivered stops increasing at the same rate and the values begin to diverge. In the screenshot you can see a crush forming in the simulation as robots trying to reach the goal end up trapping robots that no longer have a goal to move towards. I have several ideas for how to tackle this problem with the system and have outlined some of them in the future work section of this report. These

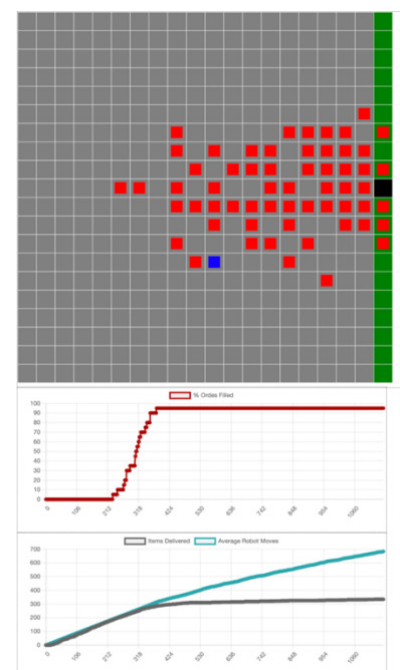


Figure 5.8 Screenshot of an existing problem with the implementation.

include introducing bias to non-busy robots to avoid busy areas and also changing the location of drop off stations. However, at this stage of the project it is definitely something that could make a grid take too long to solve.

The next important outcome was:

- Have a way of seeing how the model is running compared to other models
- The systems should have graphs and logs to be able to easily compare how changing settings impact performance of the simulation.

Both of these points are present in the project, there are a number of metrics that are used in the project to be able to compare the performance of the model when using different start-up settings. These include the percentage of orders complete at a given step count, the average number of robots moves that have been made and the number of items delivered over the course of the simulation. Here are some example graphs generated by running the system with set input parameters that can be seen in Table 8 in the appendix.

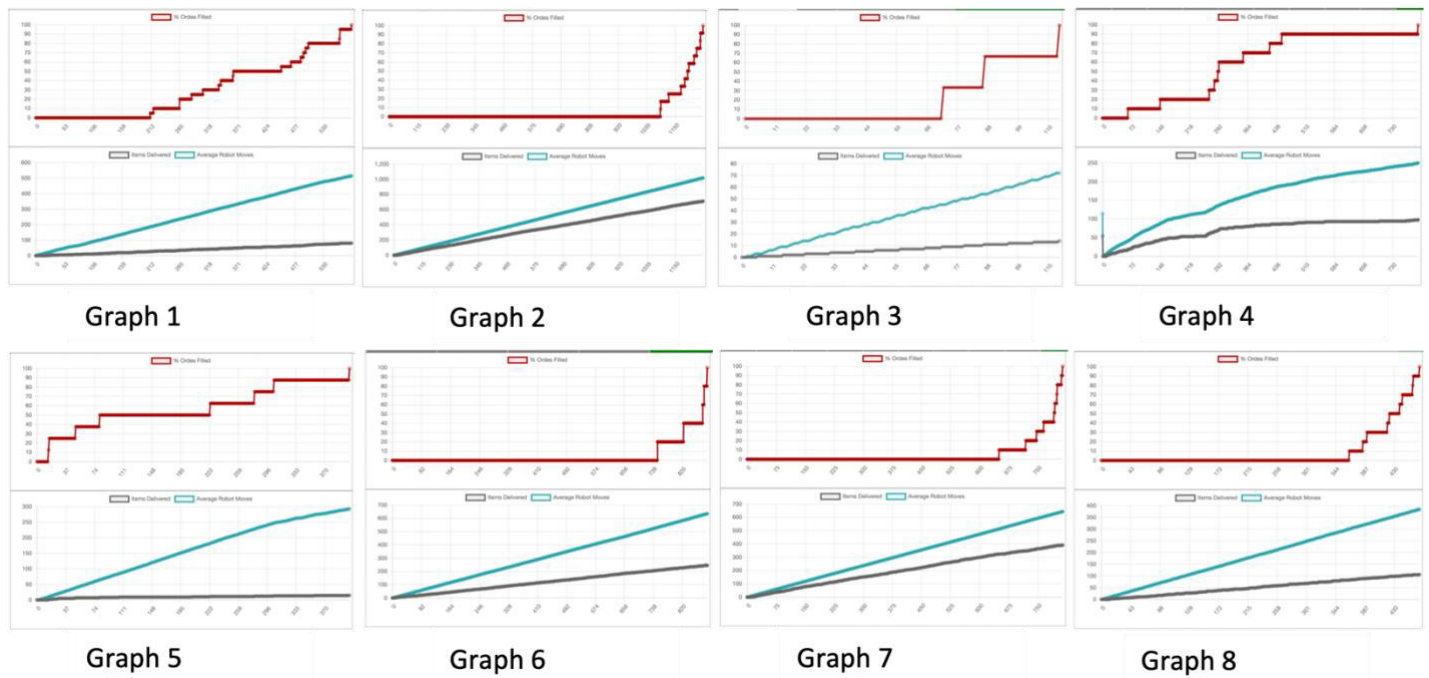


Figure 5.9 Graphs from random models, settings shown in Table 8 in the appendix.

Using these example model despite the warehouse models having different size, agent numbers, order limitations we can use these graphs to see which models perform the best. For example, when looking at Graph 5 we can see that there was a large discrepancy between the number of steps the robots took compared to the number or items delivered this means that there was a lot of inefficient movement from the robots. This in comparison to example Graph 2, in this graph despite taking a lot longer to finish the simulation we can see that the robots acting more efficiently to pack the orders needed. We can also look out for other inefficiencies, in Graph 4 there is a large plateau in the items delivered graph from about 550 to 700. This is possibly caused by a deadlock in the simulation resulting in robots not able to get where they need to and moving random not delivering items.

Evaluating the user interface of the system.

I think that the user interface of the project is an area that is good but still needs work. I had to deal with a number of setbacks regarding the types of images to use in the visualisation of the project and so it is not as clear as I would have liked with some of the icons being ambiguous as to what they actually represent. However, with the label system it is still possible to match these images quickly to see where the robot needs to get to without having to know exactly what the image is. There is also another larger issue with the visualisation, when the display mode is enabled, there can be flickering on the screen in the time it takes the images used in the visualisation to be loaded each time the grid is redrawn.

Another issue with the visualisation is with the chart module, is that it can have severe impacts on the runtime of the simulation. Table 9 in the Appendix has the run time information about several grids that were generated and tested. It shows how the two settings, display mode and charts affect the performance of the system. Figure 5.91 shows the graph of that data showing the time taken for each of the grid to complete all of their orders. All of these tests were done using the same random seed for each test, with the same unique items per order and stock per item limit.

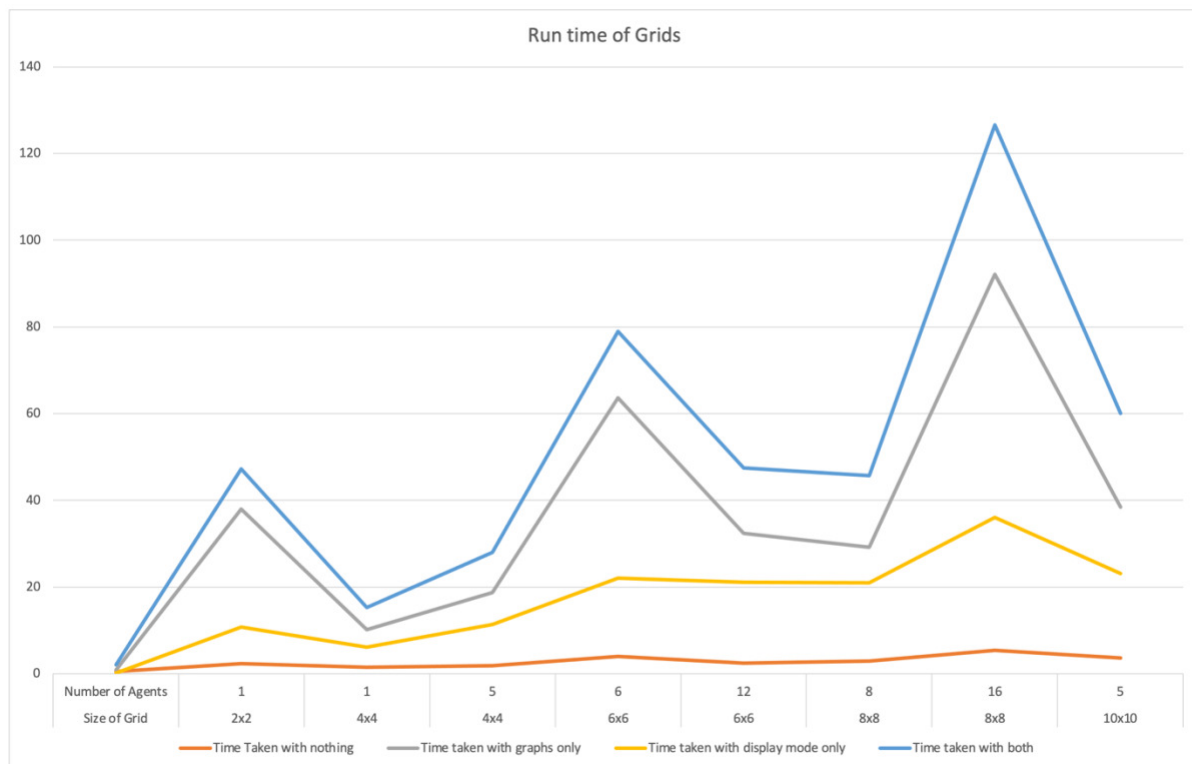


Figure 5.91 Graph of Table 9 showing the runtime of the simulation and how different modes affect performance. Time in seconds.

This graph shows that whilst both the display mode and the chart mode have impacts on the performance of the system. Having the chart mode enabled has the largest effect. What is also interesting is how closely the line graphs for the chart mode follow Figure 5.92 which shows the steps taken to solve the grid. This was a result that I was expecting and when trialling the chart mode, I noticed how because a data point is added for every step taken the charts don't have much of an impact with small grid sizes as not many steps are taken, but the more steps that are taken the more dramatic the slowdown is. This is clear to see as for the two graphs not affected by the chart mode, they resemble the steps taken graph much less.

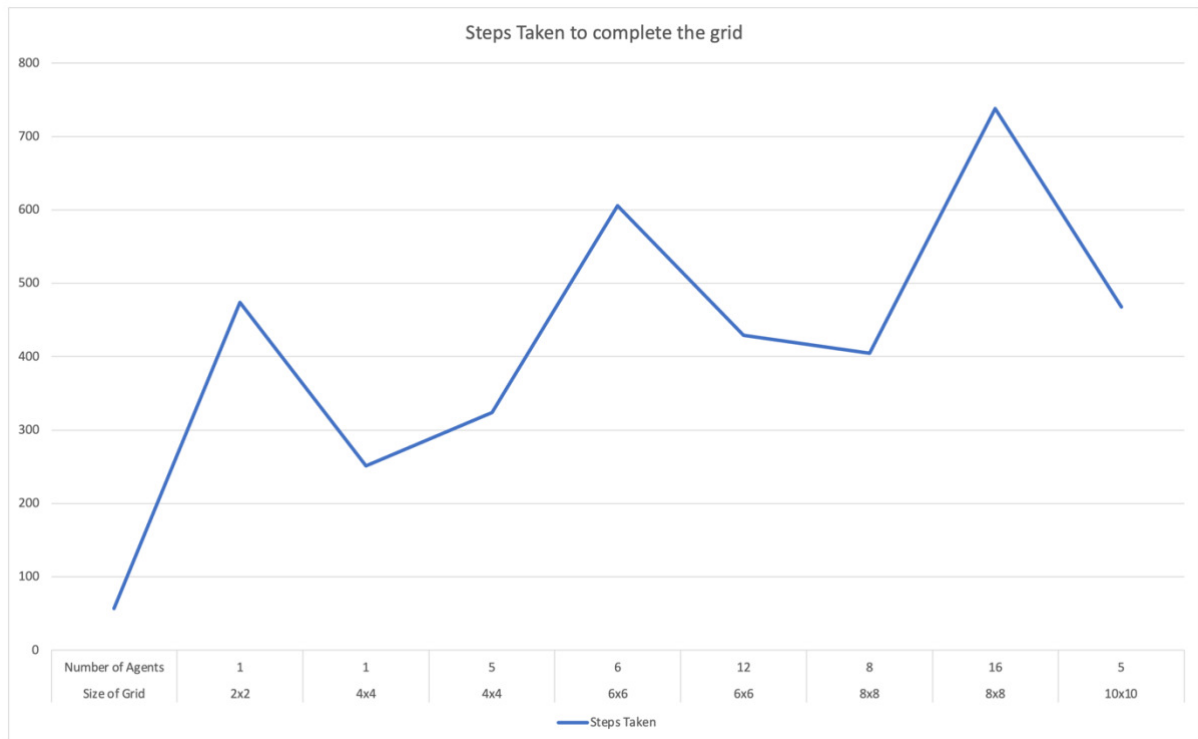


Figure 5.92 Graph showing the number of steps taken to complete the grid also from data in Table 9.

6. Future Work

There are many other features that I wanted to include in this project, some to give the users greater control over the simulation and other to improve the performance of the system.

To start with I would have really liked to get a version of the grid cell booking system to work to be able to compare it against the existing path finding methods in the system. I put a lot of time into trying to get this working and so having to drop the development of it due to time constraints was difficult, but it was not feasible to get working in the time frame I had left for the project. If I was working more on this project, I would focus on changing the A* path finding algorithm to be able to find the distance to a given node when explored and use that for the bookings.

Another algorithm if given more time I would have liked to have implemented would be the Conflict based search that I discussed in the background section of the project. I mentioned that I didn't think that it was the right solution for this project due to it only being implemented in problems that had each robot only needing to find one path in all papers I could find on it. I would have liked to have been able to implement this algorithm to test how it performed within this environment of having to find multiple paths. I would have tried to implement it in one of two ways, either treating the robot going between multiple goal cells as one continuous path to find, or more likely to reset the paths of each robot when a goal state to was reached, this implementation would have had a very slow run time however the average number of steps taken to complete the simulation would have been interesting to see.

I also would have liked to have solved the issue with crushes causing the late stages of the program to run much slower than the initial stages. The idea I had for trying to solve this was

to introduce bias to the robot's movement when they had no job to promote bots to move towards the left-hand side of the grid thereby reducing the number of agents to crush on the right side near the drop off points. I actually did do a test with this theory and had some positive results for example against the setup used for Figure 5.2 this new method completed in 434 turns instead of 3027 however for smaller numbers of agents or smaller grids the difference was negligible and bears further testing to see if it is in fact an all-around better solution.

Another idea I had for trying to reduce the number of crushes present in the system was to change the position of the drop off cells to be spread out all around the perimeter of the grid. This would have reduced the number of robots crushing on the right-hand side of the grid and could have led to the simulation completing in fewer steps. This actually feeds into another feature that I would have liked to implement, giving the user of the program the ability to drag and drop features onto the grid, so they could for example, change the starting position of the robots, select the exact cells that they wish to use as drop offs. Also giving the users the ability to manually select the contents of an order. Another idea that could have been useful for users of the system would be included and non-traversable tile, this would allow for users to test different column layouts for planning warehouses.

Another way of improving the performance of the system would be to improve the way in which jobs are allocated to robots around the grid. In the current implementation the robots are just given a task from a random drop off cell with no regard for their current position, this could really affect the performance of the system. Instead of this my idea would be to have the robot look at items near it until it found one that was needed by an order.

With thinking about the performance of the system another feature that could be added to the system would be a dialogue box to warn users about having a setup that is likely to take a long time to run, for example in table 5 you can see that using blind goal pathfinding on the 20x20 grid size with 30 agents took over 80,000 steps to complete. Giving the user about their start up settings would be helpful.

There were also issues I would have liked to have solved with the screen flickering when loading many images for the display mode visualisation. This was particularly prevalent with the larger sizes of grids. In order to have fixed this issue I was planning on trying to change some of the existing code within the mesa library to allow agents to be excluded from being updated each frame as the bin agent's images would only need to be loaded once at the start of the visualisation instead of at each step. I think this would have solved the screen flickering issue.

Finally I would have liked to have worked on a method to have the simulation run continually, so instead of simply completing all of the outstanding orders and then stopping, as the orders were completed new orders would replace them, this would help to stop crushes by reducing the amount of robots that were not busy working on filling an order and it also would have been a more realistic simulation of a real warehouse environment.

7. Conclusions

The aim of this project was to produce a simulator for warehouses utilising robotic workers. I feel that I achieved this over the course of the project, my simulator runs well, is clear, easy to understand and gives users insight into how decisions made in design can affect the efficiency

of the warehouse's operation. I explored some of the existing algorithms used for problems in the multi agent path finding space before realising that they were not appropriate for the specific implementation I needed.

I worked to develop an algorithm of my own but realised that I wasn't going to be able to in the time scale of the project and so I instead implemented a non-optimal artificial intelligence technique that still gave improved results over the non-intelligent method. I am very happy with my solution for this and while it definitely has room for improvement it forms a really strong foundation for that improvement. With this the agents are able to fulfil customer orders and give valuable data about the efficiency of the simulated warehouse, I think that the data gathered when running the simulation is reliable and useful, also because of how the robot steps are controlled this project also forms a good starting point for someone looking to try and implement their own multi agent path finding algorithm as the data gathering and MESA integration has been done very flexibly.

Overall, I am happy with the work that I have done over the course of this project but recognise that there is a lot more work that could improve this project and I have outlined some of those points in the future work section of the report.

8. Reflection on Learning

When reflecting back on the course of the project I can think of many valuable skills that I have been able to develop. Planning this kind of project was very daunting, coming up with an initial outline for how the system would work took a lot of time, scoping through the Mesa documentation and included example projects. This was a skill that I hadn't ever need to use before having previously at more needed to read documentation about how a select number of functions worked when using the MESA library because of how the different module work together I needed to be able to fully understand how the framework operated as a whole.

After the scoping phase having to put a plan in place for how I was going to go about developing this project was another new thing to learn how to do, setting milestones to have completed and breaking down what needed to be worked on for the project to proceed was a new skill, and so was sticking to the plan. In group work there is shared sense of accountability but with this project it was up to me to stick to the plan I had made. Equally as important as planning for how the project would go was it was also important at points to change that plan and pivot to either a slightly different solution or just realised a section wasn't working and decide to rework it completely.

When looking back at the project and thinking about what I would change if I had to do it again, I would allocate more time within the development process to writing the report. As I started to write the report after I had completed the project it had been a long time since I had completed a lot of the research I did before started and so had to redo a lot of that research from my notes and sources I used. If instead I had started to write the problem background while researching it, that would have been easier. Equally I would include tests that the system needed to pass in my plan for the project to help make the results and evaluation of the report easier to write and also to be able to confirm that changes made to the project didn't affect the function of other sections. To finalise, completing this project was really good learning experience, it taught me many things that I will carry forward with me into all future projects.

9. Reference List

- [1] Building a multichannel Tesco, Tesco News YouTube channel, available at <https://www.youtube.com/watch?v=QONyKR0KdYs>
- [2] Inside A Warehouse Where Thousands of Robots Pack Groceries, Tech Insider YouTube channel, available at https://youtu.be/4DKrcpa8Z_E
- [3] Mesa Overview, available at: <https://mesa.readthedocs.io/en/latest/overview.html>
- [4] Mesa modelling module overview, available at <https://mesa.readthedocs.io/en/latest/overview.html#modeling-modules>
- [5] Mesa visualisation module overview, available at <https://mesa.readthedocs.io/en/latest/overview.html#visualization-modules>
- [6] Conflict-based search for optimal multi-agent pathfinding by Guni Sharon, Roni Stern, Ariel Felner, and Nathan R. Sturtevant available at: <https://www.sciencedirect.com/science/article/pii/S0004370214001386>
- [7] The Increasing Cost Tree Search for Optimal Multi-Agent Pathfinding by Guni Sharon, Roni Stern, Meir Goldenberg, and Ariel Felner available at: <https://www.ijcai.org/Proceedings/11/Papers/117.pdf>
- [8] Mesa: An Agent-Based Modelling Framework by Masad, David & Kazil, Jacqueline. (2015). 51-58. 10.25080/Majora-7b98e3ed-009. Available at: https://www.researchgate.net/publication/328774079_Mesa_An_Agent-Based_Modeling_Framework
- [9] An Explanation on A* Algorithm by Thaddeus Abiy, Hannah Pang, Beakal Tiliksew, Karleigh Moore and Jimin Khim. Available at: <https://brilliant.org/wiki/a-star-search/>
- [10] Algorithms - A* available at: https://cs.stanford.edu/people/eroberts/courses/soco/projects/2003-04/intelligent-search/astar.html#:~:text=A*%20is%20complete%20and%20optimal,heuristics%20are%20admissible%20and%20monotonic
- [11] Tool used to generate the image available at: <https://qiao.github.io/PathFinding.js/visual/>
- [12] Figure available at: <https://www.mathscareers.org.uk/taxicab-geometry/>
- [13] Figure available at: <https://www.checkout.ie/technology/lighter-robots-and-hi-tech-routing-ocado-innovates-to-deliver-growth-161271>
- [14] Building mobile robots using automated solutions, Ocado website, available at: <https://www.ocadogroup.com/technology/blog/life-bot-building-mobile-robot-using-automated-solutions>
- [15] Padeloup, Bastien & Gripon, Vincent & Grelier, Nicolas & Vialatte, Jean-Charles & Pastor, Dominique. (2017). Translations on graphs with neighbourhood preservation

available at:

https://www.researchgate.net/publication/319662543_Translations_on_graphs_with_neighborhood_preservation

[16] Images sourced from clip art library where images are under a personal use license available at: <http://clipart-library.com/>

[17] Batch image resizer, available at: <https://bulkresizephotos.com/en>

10. Appendix

Table 1

3x3	Path Finding Type	
Number of Agents	Path Finding	Blind Goal
1	115	97
2	79	70
3	76	94
4	88	136
5	122	235
6	314	472

Table 2

4x4	Path Finding Type	
Number of Agents	Path Finding	Blind Goal
1	230	204
2	140	138
3	109	117
4	125	131
5	139	181
6	143	718
7	113	366
8	240	1599
9	366	694
10	335	3597
11	2937	1655
12	2036	1936

Table 3

5x5	Path Finding Type	
Number of Agents	Path Finding	Blind Goal
1	447	416
2	245	261
3	206	238
4	228	191
5	161	285
6	152	413
7	178	1182
8	230	973
9	226	1298
10	233	1440
11	228	2133
12	1119	2187
13	391	3697
14	1046	3042
15	941	5076
16	1965	8044
17	18395	12451
18	15310	4519
19	21892	9786
20	38521	19158

Table 4

10x10	Path Finding Type	
Number of Agents	Path Finding	Blind Goal
1	1809	1766
5	442	475
10	308	458
15	273	1455
20	266	7274
25	310	5734
30	8556	39892

Table 5

20x20	Path Finding Type	
Number of Agents	Path Finding	Blind Goal
1	7152	7049
5	1610	1643
10	910	900
15	669	700
20	572	576
25	453	3299
30	455	85545
35	401	
40	371	
45	437	
50	384	
55	350	
60	3027	

Table 6

Random Seed Used	One unique item per order	Five unique item per order	Ten unique item per order
1	18	24	26
2	37	28	31
3	33	28	27
4	44	35	23
5	29	18	20
6	24	24	23
7	27	40	23
8	37	40	31
9	31	33	7
10	24	23	23
Average	30.4	29.3	23.4

Table 7

Grid Size	Turns to complete with 1 agent
5x5	447
10x10	1810
15x15	4267
20x20	7152
25x25	11012
30x30	15704
35x35	21716
40x40	27958

Table 8

Graph Number	Grid Size	Number of Agents	Unique items	Stock per order limit	Path Finding Type
1	10x10	5	10	1	Path Finding
2	10x10	20	5	14	Path Finding
3	5x5	4	10	10	Blind Goal
4	8x8	30	1	1	Path Finding
5	10x10	9	3	5	Blind Goal
6	3x3	1	2	2	Blind Goal
7	12x12	20	10	10	Path Finding
8	20x20	5	1	7	Blind Goal

Table 9

Size of Grid	Number of Agents	Time Taken with nothing	Time taken with graphs only	Time taken with display mode only	Time taken with both	Steps Taken
2x2	1	0.53	0.87	0.094	2.08	57
4x4	1	2.28	37.92	10.75	47.17	474
4x4	5	1.43	10.16	6.09	15.25	251
6x6	6	1.85	18.72	11.29	27.94	324
6x6	12	3.93	63.64	22.07	78.91	606
8x8	8	2.42	32.4	21.07	47.45	429
8x8	16	2.92	29.17	20.93	45.73	405
10x10	5	5.38	92.19	36.03	126.6	738
10x10	10	3.66	38.38	23.16	60.12	468