Autonomous Transport Agents:

Simulating warehouse operation on a bi-directional rail network

BSci
Final Year Individual Project
Final Report

Jean Moschetta
jm407@doc.ic.ac.uk

17th June 2010

Supervised by: Dr. Krysia Broda
Second Marker: Dr. Alessandra Russo

Contents

Abstract

Autonomous Transport Agents (ATAs) have been used to improve the efficiency and reliability of transportation systems in many real-life situations. However the cost of such systems is still very high. It is therefore unfeasible for the majority of businesses to implement such a system unless they can guarantee it will be more efficient than their current set up.

Therefore, the market for ATAs has a great need in having a reliable way of simulating warehouse operation before the systems are installed. Due to the fact that autonomous systems are extremely dependent on the environment's set up and the agents' own behavioural characteristics, not only should clients be able to simulate the autonomous systems, but they should also be able to record and analyse different sets of results in the aim of finding the optimal parameters to use with their warehouses.

This project therefore aims to bring an extensible ATA simulator so that warehouse operators interested in integrating ATAs can have a realistic estimate of the improvements in efficiency which can be gained through using intelligent agents. In this report we cover two main case studies, the European Combined Terminal of Rotterdam and KIVA Systems' robots used in many warehouses. We map these real-life examples to our simulation models and perform tests on the quality of agent protocols.

Acknowledgements

Firstly, I would like to thank my supervisor, Dr. Krysia Broda, for all the support, guidance and help she has brought to me throughout the duration of the project. I would also like to thank Dr. Alessandra Russo for agreeing to be my second marker and also for suggesting different approaches in which to tackle this project.

I am extremely thankful to my family and friends, who have given me constant support throughout my studies at university and beyond.

Chapter 1

Introduction

1.1 Context

An Automated Guided Vehicle (AGV) is a term used for a mobile robot which autonomously performs tasks in an industrial environment. AGVs have been used in warehouses since 1954 when Barrett Electronics (now Savant Automation) introduced them on the market [1]. These "Driver-less vehicles", as they were known at that time, were simple tow trucks guided by a wire installed on the ground of the warehouse, these transport agents were "autonomous" in the sense that they could operate by themselves, but could also do little apart from moving forwards and backwards. Since then the autonomous logistics industry has come a long way. There are now a wide range of AGVs available, some are capable of transporting 40 feet containers as well as navigating outside in different kinds of weather conditions, while others are able to transport small stacking shelves towards packing operators as well as reorganising the position of the shelves in the warehouse based on real-time customer demand for the items - the more an item is required by the customers, the closer the shelves will be to the packing operators to improve speed and efficiency. Intelligent agents are therefore the key in reaching the goal of optimal efficiency. AGVs therefore involve many areas of Computer Science, from robotics to AI and even operations research.

There are several reasons why it is advantageous for businesses which require transportation of goods to move towards automation of their warehouses. AGVs can operate 24 hours a day, 7 days a week with breaks only required to recharge batteries or replenish fuel. Declining population rates in developed countries like Japan have prompted the need for an alternative workforce and has in turn driven the development of autonomous systems, with great progress made in the AI of individual AGVs. The economic crisis along with rising labour costs in western Europe, as well as in many other countries throughout the world, has prompted warehouse managers to employ AGVs who, beyond maintenance, require little to no attention. Given a set of goals, AGVs will nowadays be completely autonomous and interact with the environment to accomplish these goals with no direction from human operators. Of course, the initial investment required to develop and install an AGV system is quite large and can be a

big obstacle for the majority of medium businesses, especially in this uncertain economic climate, however the gains in speed and efficiency of operations are great and can often start paying for themselves within 3 to 5 years [11].

Although there are many advantages to a fully automated warehouse, the complexity of designing efficient autonomous behaviour is a great challenge. Managing transportation in terms of planning and scheduling is a challenging task due to the complexity and dynamics of the processes involved. Hence recent developments show an increasing trend towards autonomous control of transportation processes with software agents acting on behalf of human operators. Software agents are the natural solution for transportation problems as they can be applied not only to the AGVs but also to other processes in the warehouse such as the refuelling or loading and unloading processes.

1.1.1 Motivation

Simulating the transport process will allow businesses interested in installing an AGV system to evaluate the gains in efficiency from developing various protocols of interaction. As we noted previously, costs associated with AGV systems are very high and business owners will want to simulate how well the agents are performing before embarking on automating their warehouses.

Instead of simply simulating traffic interactions between the AGVs we constrained the project to include bi-directional rails. This adds to the complexity of the project as autonomous behaviour will not simply reflect normal traffic, in our scenarios the AGVs will be able to navigate the warehouse freely but on single-track roads.

1.1.2 Project aim

In this project we set out to simulate the operation of a warehouse in the aim of finding the "best" autonomous behaviour using software agents. By simulating warehouse operation we were then able to observe the performance of a set of autonomous protocols and deduce the best protocols to use based on warehouse layout.

Therefore the main aims of this project can be summarised as follows:

1. Provide a warehouse simulation framework by extending the MASON Multiagent Simulator (see section 2.1.3).
2. Investigate how to improve the efficiency of the warehouse by experimenting with the simulation parameters.
3. Being able to visually observe the agents and improve their protocols.

We also set out to meet the following criteria:

- A simple system, we wanted the simulations to be physical implementation independent - we don't care so much about the physical aspects of AGVs apart from making sure the simulation is "legal". However we also wanted

to have enough parameters such as weight of packages and speed of agents which can be altered to reflect the real-life systems more accurately.

- We also wanted to implement a scalable routing protocol to attempt to find the highest efficiency based on different stimuli such as congestion or agent presence.
- Being able to analyse different network topologies - how to find out where bottlenecks are in a certain system.

1.2 Contributions

We now outline the contributions made in the design and development of this project:

- We have implemented a warehouse transportation system which is easily modifiable to fit the user's needs.
- We are able to record a number of statistics about each simulation - with the objective of being able to judge how "good" a protocol is based on different measures.
- We are able to change simulation parameters to reflect different real-life systems. In this project we have simulated the transportation aspect of a container terminal and a warehouse.
- Using these points we are able to test and report which particular parameters are most suitable for which situation.
- The system was designed to be able to add new agent protocols easily by extending the transport agent base class.
- We have implemented a series of different protocols and tested these protocols against different network layouts and different simulation parameters.
- The system also allows a future user to scale the A* routing protocol which was used extensively in the implemented protocols. Extending the heuristic function is all that is required to create new heuristics on which the test the agents with.

1.3 Report Structure

We begin this report with some background information concerning Agent Based Modelling and how we used this to model the real-life systems which were studied in the preliminary part of this project. We continue with case studies which were used as the motivation behind this project as well as detailing the MASON simulation framework which we used as the basis of our simulations. In chapter 3 we talk about the design aspects of the project and how all the simulation objects link together to form our models. This is followed by chapter 4 where we detail the implementation aspects of the project as well as giving concrete

examples of the situations in which we tested our agents. In this chapter we also detail each of the protocols of agent interaction which were implemented in tested. Chapters 5, 6 and 7 cover the experiments we carried out, the results which were obtained, and an evaluation of these results respectively. We also recommend a set of optimal parameters to use with each environment tested. We conclude this report in chapter 8 where we give detailed descriptions of possible extensions which could be added to this project in the future.

Chapter 2

Background

2.1 Agent Based Modelling

In Agent Based Modelling, or ABM, the real-life system which we are interested in is modelled using a collection of autonomous entities called agents. These agents usually represent actual objects from the system which interact with each other and are adaptive to their surrounding environment. By modelling a real-life system using ABM, one can model complex behavioural patterns and record meaningful information about the resulting effects on the system environment. Using this information we are able to infer theories of how the actual system would behave under the conditions in which we have modelled it.

In ABM we learn more about a system from the local interactions among the agents. The agents interact with the "environment", this is a space where all interactions occur. Agents can interact with the environment or interact with each other based on environmental conditions. The Object Oriented Programming paradigm is a very natural way of implementing Agent Based Models, where each agent is an object and interacts with other agent objects who themselves have specific sets of characteristics and rules.

This approach to modelling was also ideal since as we will see in section 2.2.2, one of the real-life systems which we studied used this very approach to operate their actual system.

2.1.1 Autonomous Agent

An agent is usually described as a discrete entity with its own set of behaviours and goals. In this project we also used the term "autonomous agent" to refer to an agent which has the capability to change its behaviour depending on the conditions of the local environment which it senses. These autonomous agents are usually mobile objects within the environment, however we also see that we can make any entity an autonomous agent - with the purpose of closely mapping the characteristics of the real-life system with our model.

An autonomous agent is characterised by the following general properties:

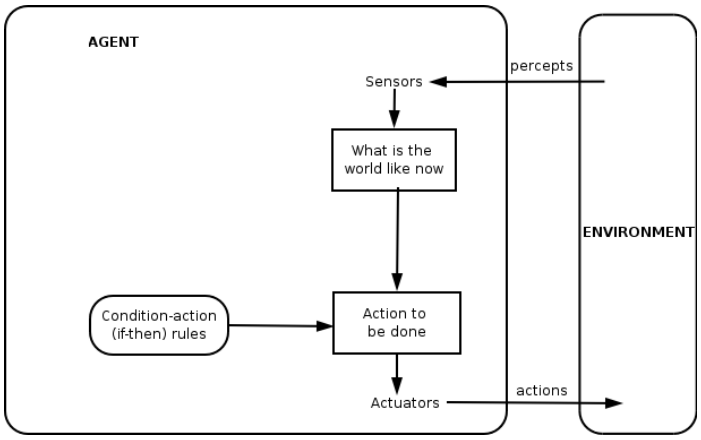


Figure 2.1: Autonomous Agent

- Interaction with the environment: the agent senses the (constantly changing) environment through its sensors and reacts accordingly.
- Goal driven: agents can have a wide range of goals, including satisfying local states or end goals.
- Intelligence: the agent has a decision-making capability by which it uses local information to select actions.

2.1.2 Key Steps of ABM

When using ABM, one usually takes a similar approach to [4] - from which the following list is adapted to fit the needs of this project.

1. Find a meaningful question to ask about the behaviour of a real-life system.
2. Take this problem and simplify it.
3. Simulate the system by programming the agents to follow simple rules and interact with the system.
4. Run the simulations many times and observe patterns.
5. Take the observations and propose theories about how the system behaves.
6. Change parameters of the simulation and identify sources of behavioural change.
7. Repeat steps 4 to 6.

Using this approach with the project proved very successful, especially since our simulations revolved around a lot of modifiable parameters and therefore step 6 was repeatedly performed and results were recorded in chapter 6.

2.1.3 MASON

This section describes the framework and libraries we will be using for the duration of the project.

MASON [10] is a fast discrete-event multiagent simulation library core written in Java, designed to be the foundation for large custom-purpose Java simulations, and also to provide more than enough functionality for many lightweight simulation needs. MASON contains both a model library and an optional suite of visualisation tools in 2D and 3D.

There are several simulation packages that would fit our needs, such as SWARM, which fully supports multiagent based simulations. We decided to go with MASON for several reasons: MASON was built with Java rather than Objective-C with SWARM, this author is more proficient in Java. The graphical component of MASON is also very nice and allows the user to build on top of the existing libraries, this was a big selling point for MASON due to the fact that in this project we wanted to spend more time designing heuristics for agent interaction rather than building a graphical library from scratch. MASON also has built-in grid representations, which are ideal for our project as we will explain below.

2.1.3.1 Grids

MASON uses a grid concept for representing the position of agents (although this can be done any way the user wants). There are different types of grids, such as int, double, etc. This allows us to represent all kinds of data, from agent position to static objects in the world. All the information which the grids contain can then be represented using MASON's internal graphics library. By superimposing the grids, more than one set of data can be represented at the same time.

The grids have other interesting uses. To prevent agents from bumping into each other during their journey (as they could be carrying fragile cargo), a method which we used in this project is to partition the world in small mutually exclusive zone. Essentially two agents cannot enter the same zone at the same time. Each zone must be cleared of agents before the next agent can enter it. This is like locks in multi-threaded programs and serves as a good way of making sure the agents are a safe distance away from each other. Using MASON's grid is one way of achieving this "zone" principle and can be easily implemented, each time an agent wants to move a grid square, it must first check there are no agents in the adjacent square and so on. In this project we implemented this scheme using the so-called "Moore neighbourhood", which signifies the 8 adjacent squares in a 2 dimensional grid. These squares are probed by the agent's sensors and the agent will be informed of the local environment in this way.

2.1.3.2 Networks

Also provided in the MASON's libraries is a network system consisting of edges and nodes. These edges and nodes can be built and manipulated in the world.

These constructs could be used by the agents to navigate the world. In this project we decided not to use MASON's edges and nodes so that agents could be visualised consistently with the grid representations that we used for the rest of the environment (rather than the abstract way in which networks are visualised in MASON). Although we did not use networks, we created our own node objects which we embedded inside the grid representations.

2.1.3.3 Visualisations

Since we aimed to visualise the simulations consistently but also collect results in a batch manner, MASON was the best choice since it is specifically designed to decouple the simulation from the visualisation. This enables us to run the simulations many times and record results for each of these runs, as well as observing the agents in action using the visualisation suites - enabling us to debug agent behaviour and notice any behavioural patterns as the simulation is in progress.

2.2 AGV

In this section we outline two AGV companies. The systems implemented by these companies are what our simulations are modelling closely.

2.2.1 Gottwald Automation

Gottwald Automation is a company which introduced the autonomous transport trucks at European Combined Terminals of Rotterdam in the Netherlands. The huge autonomous trucks carry 40 feet containers to and from ships that are docked in the harbour. Their navigation system works by following an electromagnetic wire which is embedded in the ground of the container terminal. This allows them to navigate through the terminal flawlessly in many different weather conditions (see figure 2.2).

2.2.2 KIVA Systems

The system described by the IEEE Spectrum article [6] served as an important source of inspiration for this project. Although there are many Warehouse Management Systems (WMSs) in the industry today, the one built by KIVA Systems uses a revolutionary approach (a robot of which is shown in figure 2.3). By transporting items to the packing centres rather than requiring packers to stand behind a conveyor belt, the system can optimise the time required for each TA to reach their destination.

2.2.2.1 Specification

The KIVA robot works by scanning the ground of the warehouse for barcodes which specify grid-like locations throughout the warehouse. By scanning these

Figure 2.2: Gottwald ECT trucks [2]

Figure 2.3: KIVA Systems' Autonomous Agent [13]

codes the robot knows exactly where it is in relation to the warehouse, the wares and other robots. This allows it to be very accurate in moving through the warehouse as well as avoiding other robots in advance, all while fulfilling the tasks that were assigned to it.

2.2.2.2 Traffic Management

What is important about this WMS to the project, is the way KIVA Systems modeled their warehouse. Figure 2.4 shows a colour diagram of KIVA's internal representation of the warehouse. In red are the items which are most needed by the warehouse, compared to green or purple which signifies less important items. These items are placed near the packing centres due to their high demand to increase efficiency.

In this project we will also use colour to represent different characteristics of the environment. Particularly congestion, which we cover in the visualisation section of this project in chapter 4.

2.2.2.3 Design Issues

Without giving the game away, the IEEE article mentioned some of the techniques which KIVA Systems used in building the algorithms needed for agent interaction. The engineers talked about how organising the robots in warehouses was a difficult task - because of having hundreds of robots and many packing centres - coordinating the whole system to run flawlessly is an NP-hard problem.

To solve this massive scale coordination problem they used software agents to run the robots, the packing centres and the central computer. Each agent would exchange information but act independently, each trying to optimise its own tasks. They also used heuristic methods such as *greedy algorithms*, which can make good (but not always optimal) decisions to perform the required tasks.

Similarly for this project, agents will be the most important entity in the simulation. We also attempted to use the same sort of methods which KIVA

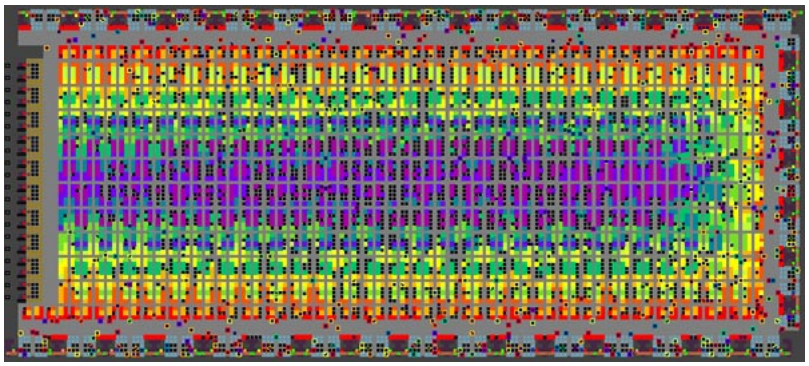


Figure 2.4: KIVA System's demand representation

Systems have used in their design.

2.3 Further Notes

- We refer to a "Moore neighbourhood" in several places in this report. This neighbourhood consists of the 8 squares which are adjacent to any chosen square in a 2D grid.
- The network topologies implemented in this project are "planar graphs", i.e. no edge in the network will overlap other edges. This was done to realistically represent roads in a 2D environment, although bridges can exist in a 3D environment, we restricted the simulation to operate only in 2D.
- We will also refer to the Time To Live, or TTL, of a package - this term is used when talking about packets travelling through networks. It is used here to refer to the perishability of a package, this TTL must not be exceeded or else the package "dies", i.e. it failed to be transported in due time.
- Lastly but most importantly, when we refer to the waiting time of a package we mean the time it took from package generation (at a station) until it was safely transported to its destination station and exist the system. This is the main measure of quality of the protocols of agent interaction, as the smaller the waiting time for a package, the better the protocol is at navigating through the system and accomplishing its goals.

2.3.1 A* Algorithm

The A* algorithm was used extensively in this project. A pseudocode version can be found at algorithm 2.1, a full listing of the Java implementation of the A* algorithm for this project can be found in appendix A.1. The A* algorithm works by using three functions, G,H and F. G is a function which calculates the distance between two adjacent nodes. H is the heuristic function which tries to

estimate the total remaining distance until the goal and F is the sum of the G and H functions. Each A* node keeps these values to allow the algorithm to choose the best path.

Algorithm 2.1 A* Search Algorithm [3]

```
function A*(start,goal)
    // The set of nodes already evaluated.
    closedset := the empty set

    // The set of tentative nodes to be evaluated.
    openset := set containing the initial node

    // Distance from start along optimal path.
    g_score[start] := 0
    // Estimated total distance from start to goal through y.
    h_score[start] := heuristic_estimate_of_distance(start, goal)

    f_score[start] := h_score[start]
    while openset is not empty
        x := the node in openset having the lowest f_score[] value
        if x = goal
            return reconstruct_path(came_from[goal])
        remove x from openset
        add x to closedset
        foreach y in neighbor_nodes(x)
            if y in closedset
                continue
            tentative_g_score := g_score[x] + dist_between(x,y)

            if y not in openset
                add y to openset
                tentative_is_better := true
            elseif tentative_g_score < g_score[y]
                tentative_is_better := true
            else
                tentative_is_better := false
            if tentative_is_better = true
                came_from[y] := x
                g_score[y] := tentative_g_score
                h_score[y] := heuristic_estimate_of_distance(y, goal)
                f_score[y] := g_score[y] + h_score[y]
    return failure

function reconstruct_path(current_node)
    if came_from[current_node] is set
        p = reconstruct_path(came_from[current_node])
        return (p + current_node)
    else
        return current_node
```

Chapter 3

Specification and Design

Bearing in mind that the main goal of the project was to simulate warehouse operation with the intention to implement and compare different protocols of transport agent interaction. We now describe the specification of the simulation as a whole, along with the design decisions that were made to achieve the goals that we set out to complete.

We begin this chapter by explaining several important points about the assumptions that were made in designing the project, we then show how we used the MASON scheduler in order to give life to the simulation objects. We then move on to section 3.7 where we outline the design of the crucial package objects, without which the simulation would not have much meaning. The internal design of the maps, nodes and details of a basic agent protocol are covered in sections 3.5, 3.6 and 3.8 respectively. We conclude this chapter with section 3.10 which covers another crucial aspect of the project, its visual design.

3.1 Abstraction

Although we aimed to reach a realistic representation of warehouse operation, we also did not want the simulation to be too dependent on the physical implementation of the Warehouse Management System (WMS). As seen in section 2.2 there are many different types of AGVs and even more ways of making them interact with their environments, whether they navigate the world using electromagnetic wires or bar codes specifying warehouse locations.

Because of this we abstracted our simulation to be largely "physical implementation independent". By this we mean that we do not concentrate on the physical implementation of the agents, but instead we focus on their interaction in the world. The transport agents could be big or small, fast or relatively slow, this simply does not matter as far as our simulation is concerned. The only physical aspects that was taken into account were how we perceived transport agents to operate in a warehouse environment - which we designed through careful observation of real life WMSs - as well as how the agents work their way through the warehouse on the rail network which we devised, this is further explained in the next section.

3.2 Bi-directional Rail Network

One of the main design decision before starting the implementation of the simulation was to restrict agents to operate on bi-directional rails. Unlike conventional railways where trains usually travel in one direction, our simulation allows the transport agents to go back and forth on the same piece of track. We decided to use this model to allow more interesting levels of interaction from the agents in their environment, if we had used uni-directional tracks our simulation would amount to modelling conventional traffic interactions.

Allowing agents to go in both directions increases the flexibility with which they can travel through the network. It will also cause agents to travel towards each other, causing one or both agents to have to abandon the path once they have realised that it is blocked. Because of this "problem", the project's emphasis therefore resided in making sure the agents could recover from situations like these and attempt to find a better path than the one they had just tried.

3.3 General Specification

In this section we outline the general points of the specification which our simulation complied with. To achieve a reasonable degree of realism in the simulation several points were taken into account during the design process of this project.

- Although the MASON data structures for keeping track of the transport agents allow multiple agents at the same grid location, we restricted our simulation to one agent per location. Transport agents travelling through each other would not be very interesting and would not satisfy our goal of making the simulation as realistic as possible.
- There is therefore no possibility of overtaking occurring on the same rail. Any form of overtaking in our simulation would amount to routing through an adjacent path and continuing towards the goal at a faster rate than other agents.
- The speed of the transport agents was a factor which was identified as being crucial to the "realism" of the simulations. Early in the project's inception we decided that speed should be inversely proportional to the weight carried. It would make little sense for heavily loaded agents to travel at the same speed as agents who are not carrying anything. We have however provided functionality so that speed can be modelled to user preference, more on this in chapter 4.
- Naturally, a maximum weight was imposed on the agents' cargo space.
- Since it would be unrealistic for the transport agents to load and unload packages instantly, the simulation has parameters which simulate the loading times dependent on the amount of packages picked up or dropped off. These parameters can be altered to user preference as well.
- Packages record their Time To Live (TTL) to simulate perishable packages, this parameter is completely user dependent and serves only to test

the performance of the agents if high priority packages enter the environment. The TTL function which defines how much time until a package expires is based on the priority of the package. The higher the priority the less time the package will "live" in the simulation. Failure to deliver these packages is recorded and displayed at the end of the simulation.

3.4 Scheduling

The MASON simulation framework works like any discrete-event simulator. By scheduling events to take place in the future on the schedule we can make the warehouse world come alive. In this section we will use the term timestep to specify a unit of time.

Although events in MASON can be scheduled to happen at very small intervals within a timestep, for simplicity we made events occur only at whole timesteps. This simplification does not reduce the scope of the simulation, events that are scheduled to happen immediately are scheduled for the next timestep and events which require more time are simply scheduled multiple timesteps in the future. One can think of a timestep as the smallest unit of time in our discrete-event simulator.

3.4.1 Stepping the Schedule

Before we dive into further detail of how the schedule was used with all the elements of the simulation, we note that the MASON framework specifies a "steppable" interface. This interface must be implemented by all agents which are to be scheduled in the simulation. The steppable interface requires the implementation of a single method called the "step" method. This method is called whenever the schedule reaches a point in time where an agent is to be scheduled, it can be thought of as the agent performing an action - hence all interactions with the world occur within this step method.

We now continue this chapter with detailed explanations of how the schedule was used to model different aspects of the warehouse and its transport agents.

3.4.2 Speed

As noted earlier in this chapter, speed is a crucial variable which we wanted to model accurately, namely that more heavily loaded agents travel at a slower rate through the rail network.

To achieve this we decided to schedule empty agents immediately to the next timestep, whereas heavily loaded agents will be scheduled to move only several timesteps in the future - all depending on how many packages they are carrying. This method of modelling speed proved very successful considering the alternative, to calculate how many grid locations an agent would traverse in a single timestep, would cause headaches in dealing with agents overlapping each other.

3.4.3 Stations and Packages

Since stations are themselves "software agents", they also have a place on the schedule. Stations are scheduled every timestep and for every timestep stations will generate package arrivals.

The package arrivals are generated at random intervals, to achieve this we used an exponential distribution along with a threshold parameter. The threshold parameter being user defined, if a sample from the exponential distribution exceeds this threshold then a package is generated, if on the other hand the threshold is not reached the station object simply does not generate a new package.

To sample from the exponential distribution we used the inverse transform method which is appropriate since the cumulative distribution function of the exponential distribution has an inverse.

If we take random variable $U \sim Uniform(0,1)$ which is a uniformly distributed random variable between 0 and 1, the variate

$$T = F^{-1},$$

has an exponential distribution, where F is the cumulative distribution function of $Exp(\lambda)$. In detail:

$$X \sim Exp(\lambda)$$
$$F(x) = 1 - \exp\{-\lambda x\}, \qquad x \geq 0$$

setting

$$\begin{aligned} U &= F(X) \\ U &= 1 - \exp\{-\lambda X\} \\ 1 - U &= \exp\{-\lambda X\} \\ \ln(1-U) &= -\lambda X \\ -\frac{\ln(1-U)}{\lambda} &= X \\ F^{-1}(U) &= X, \end{aligned}$$

so since $U \sim Uniform(0,1)$ then

$$F^{-1}(U) \sim Exp(\lambda).$$

Hence we used the exponential distribution sample as

$$F^{-1}(U) = -\frac{\ln(1-U)}{\lambda} \sim -\frac{\ln(U)}{\lambda} \sim Exp(\lambda),$$

since $1 - U$ is also $Uniform(0,1)$.

3.4.4 Graphics

For the project's visualisation which is specified in chapter 4, we used several 2D grids to represent different effects such as trails of the TAs previous location

or other visual aids. These visual effects also took place on the schedule, although the effects were not agents themselves, they were created by the agents interacting with the environment. The visual effects are coordinated by a separate software agent, which like other objects on the schedule, operates at each timestep to regulate the intensity of the visual effects. The visual effects were created for easier analysis of the world when the simulation is paused, but more on this in the next chapter.

3.5 Maps

The warehouse layouts were represented using a combination of several data structures.

- For the model of the rail layout, a single 2D grid was used representing areas with the rail tracks and areas without. The 2D grid representations provided by MASON can hold any floating point value at each location, where any location that is non-zero specifies a location in the world which has a rail track. This allows the transport agents to query their surroundings by checking this 2D array and making sure they are still travelling along the rail track.
- Maps contain four different types of objects: nodes, stations, TAs and usually a single dispatcher. These objects are stored within a sparse object field, where each field has locations mapping one-to-one onto the 2D grid representation. This allows locations for each field to map directly onto a location in the 2D grid on which the rail tracks are specified. Further detail on each object which the maps contain are detailed in the following sections.
- Rail tracks are connected using node objects, a similar concept to a train set - where uniform pieces of rail track can be connected in different ways to form the track.
- We note that the 2D grid representing all possible locations within the world is a discrete grid, by this we mean that agents move from their current location to the adjacent location in one timestep, i.e. there is nothing in between. One can think of a 1-dimensional array, with an agent occupying a place within the array and when the agent moves it simply changes location from one array index to the one next to it.

Several other 2D arrays were used to specify the visual aspects of the simulation, more on this in section 4.3.

3.6 Nodes

Nodes are the base class for station and dispatcher objects, more importantly they act as the single entry points where agents retrieve or calculate their next path.

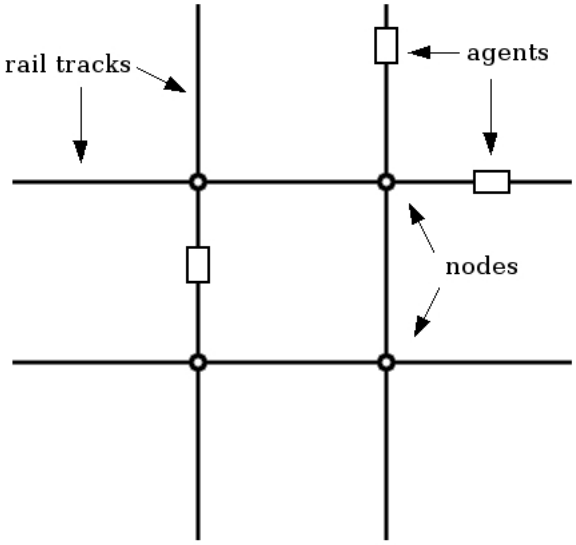


Figure 3.1: Rail tracks, nodes and agents

Agents essentially make decisions about where to go next when they arrive at a node. Of course if an agent were to detect another agent in its path it would make a decision wherever it finds itself, but network routing only occurs at node objects.

Not only does this minimise the amount of computation required per timestep, but this prevents agents from recalculating a better path at each timestep and suddenly deciding to abandon a path. Although it would seem advantageous to recalculate a better path at each timestep, depending on the routing algorithm, worst case scenarios could arise.

For example, if dynamic congestion is taken into account, the agent could calculate a path and go along a rail track. If by the time the agent is in the middle of the track the congestion changes, the agent could decide to reverse and try an alternative track which is all good so far, however if the dynamic congestion suddenly changes again the agent could decide to go down this track again, essentially live-locking itself and not making any real progress. It was therefore decided that agents should choose a piece of track until the next node. Since nodes are evenly spaced within the maps the cost of choosing a track and not abandoning it until the next node is not huge and prevents the live-lock condition.

3.6.1 Stations

Stations are a subclass of node objects. Their sole purpose is to generate packages into the world, when a package is generated it is assigned a destination at random. TAs load and unload packages on these stations, therefore agents will remain inside the station for the specified loading/unloading period.

3.6.2 Dispatcher

The dispatcher is a convenience object with two objectives.

1. It has the responsibility of spawning the TAs into the world, it can be thought of as the "source" of the agents. The number of TAs to be generated is obtained from a parameter class and can be altered if multiple simulations are executed with the goal of testing performance with different numbers of TAs.

2. It can also be set to be the "sink" once the simulation is nearing the end, by this we mean that when all packages have been generated and picked up by the TAs, any TA which does not have a job will set its next path to the dispatcher and be removed from the simulation once it reaches it - this is to allow the simulation to end without one agent taking a very long time to find its final destination due to other idle agents being in the way.

The dispatcher was made a subclass of station so that agents could easily path find it without any changes to their protocols. Although manually placing the TAs within the simulation is possible (and was done in the early stages of the project), by using a scheduler we can specify the number of agents at run time using the parameters file and therefore run many simulations with varying amounts of TAs.

3.7 Packages

Packages, along with the agents themselves, are the core of our simulation. Transport agents pick up and transport these packages to their respective destinations, how fast the agents perform this task is a measure of how well the protocols are performing in the specified environment.

We now list the fields which each package contains:

destination as the name suggests, this keeps a reference to the station through which the package will exit the system.

timeIn records the time from the schedule at which the package first entered the system.

timeOut records the time from the schedule at which the package exited the system, this is used to calculate total waiting time.

priority — a number from 0 to 9 specifying the priority of the package, 0 is the highest priority. This variable is used in calculating the TTL. The simulation generates packages with priority uniformly distributed within this interval.

TTL — time to live, specifies the upper bound of the amount of time the package is allowed to stay on the system. If timeOut - timeIn is greater than this TTL then the package has failed to be delivered in time. A low priority package will have a large TTL - usually larger than the average waiting time for a package, this TTL variable is used to simulate perishable packages.

weight — a variable describing the weight of the package, the simulation generates packages with weight uniformly distributed between a user defined parameter.

Packages are initially contained by stations - where they are generated - they are then picked up and transferred into the transport agents. They finally exit the system on their destination stations after which they are removed entirely.

A last note on the design of the priority variable, to prevent packages with a low priority from staying inside an agent for too long, the priority of each package decreases as the transport agents progress through their goals - each time an agent deposits packages, all packages remaining to be transported increase in priority (decreasing the priority variable towards 0). This will bring packages at the front of the priority queue so that they are eventually removed. The TTL is not affected by this change in priority, this will simply prevent low priority packages from staying on a transport agent for too long and skew the average waiting time in our results.

3.8 Transport Agent Protocol

In this section we describe the basic protocol for a transport agent. We note that regardless of the decisions the agent takes, it is crucial for the agent to remain "in motion". Permanent deadlocks are not acceptable in our simulation since the purpose of the simulation is to transport packages to and from stations, if TAs remain in a deadlock state permanently this prevents any packages from reaching their destinations and the simulation would therefore never end.

We now list the most "basic instincts" which TAs will have, this behaviour will be observable for each of the protocols that were implemented in chapter 4, regardless of their level of intelligence.

1. Recognise the environment
2. Acquire goal
3. Pathfind goal
4. Pick up / unload packages

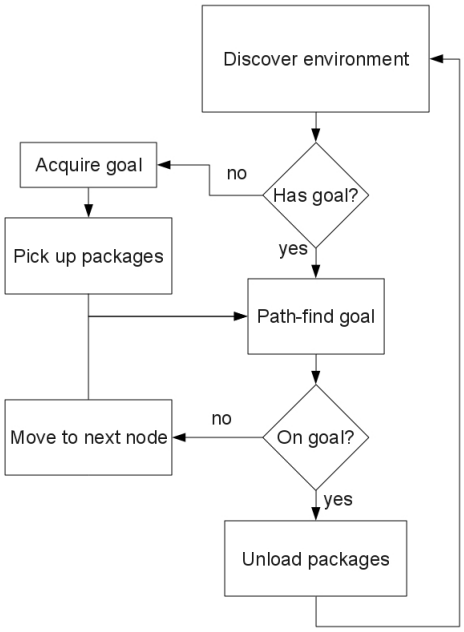


Figure 3.2: Flowchart of protocol basics

See figure 3.2 for a more detailed breakdown of basic agent behaviour.

The TA class is the base class which all protocols extend. This class holds all core functionality which we list below:

- Locating the agent's goal.
- Finding the nearest node.
- Finding the next forward direction.
- Environment discovery:
  - Road detection.
  - Node detection.
  - Collision detection.
- Speed calculations based on packages carried.
- Picking up and unloading packages.

By providing core agent functionality in the base TA class, we enabled protocol creation to be a lot faster to implement. Protocols only specify their step method, which is essentially a different version of the flow chart in figure 3.2.

3.9 Analysis

By recording the occurrence of events we were able to collect meaningful information from our simulation runs. This is a crucial aspect of the project as we initially wanted to be able to compare the protocols and how well they perform on certain types of layouts.

With this in mind we created a global statistics class which is used by all objects and specifies the occurrence of events. We note that events are completely user defined and the simulation has no way of automatically recording when such events occur. Because of this, when a protocol is designed, the statistics class must be used whenever an event of interest is noticed to happen and should be recorded.

To explain further, if the user wanted to record how many times an agent turns right compared to its direction of travel, the user would use the statistics class to notify that this type of event has occurred. The results would then be calculated and displayed at the end of the simulation.

It is possible to record any kind of event occurring within the simulation environment, the two most important aspects of a simulation are as follows:

1. Total simulation time for agents to transport a user specified amount of packages.
2. Average waiting time for a package (from arrival on the system to its exit).

These two factors allow us to judge how well a protocol is performing, the worst the protocol, the more time a package will spend being transported. With this measure of quality, we were then able to design experiments and test the protocols in chapter 5.

3.10 Visual Design

As one of the main goals of the project was to be able to visualise the simulations, a bulk of the work involved creating the visual representation of each element of the environment and how the objects within the world are visualised. Details of each visual representation is specified in chapter 4, in this section we outline the visual aspects which we have covered in this project and leave the implementation details and examples of each visualisation for the next chapter.

The basic visualisation of the simulations is a 2D representation with rail tracks, nodes, stations and agents as different coloured objects on screen. By visualising the simulation in real time we are able to observe how well the agents interact with the world. Not only does it allow us to see the agents interact as we have programmed them to, but it is also crucial for debugging of the protocols. Since creating autonomous behaviour involves taking into account all possible situations in which an agent will find itself, it is sometimes difficult to think about all possible scenarios on the first iteration of a particular protocol.

By observing the agents in an informative 2D representation, we are able to see when the agents do not behave as required because a particular scenario

was not taken into account. The 2D visualisation also allowed us to represent different events happening in the environment, such as packages arriving at a node, or congestion building up on certain bottle neck roads.

By using Java3D's libraries, the MASON framework also allows us to visualise the simulation in a 3D environment. Although the world is still flat, by changing certain aspects of the simulation, such as the congestion, into a height map, we are able to observe in real time where most of the congestion is focused. In a 2D visualisation, although congestion can still be observed, we only have a flat view of it and it is therefore not as informative as having a 3D representation of it. The 3D visualisation can also enable us to run the simulations until they have finished and observe on which roads the agents were stuck on the most. This in turn can let us change the layout of the world and observe the changes that it has on overall congestion.

3.11 Summary

In this chapter we specified the design decisions which were taken to reflect the original goals of the project. We also go into detail about certain aspects of discrete-event simulation as well as describing the most basic elements of the behaviour of transport agents.

In the next chapter we concentrate on the implementation details of the project, giving concrete examples as well as specifying each protocol of interaction for the transport agents. We give screen-shots of the simulations in different stages of progress as well as a breakdown of all the visual aspects which were implemented through the duration of the project.

Chapter 4

Implementation

4.1 Extending the SimState

MASON provides a base simulation class called SimState. Every simulation that is modelled using MASON must extend this class and implement the relevant methods to set up the world and start the schedule. As specified in section 3.4, objects which wish to perform actions must implement the steppable interface before being able to be scheduled.

Figure 4.1 shows in detail how MASON operates.

4.2 Protocols

In this section we specify the different protocols which we implemented. All protocols have the same collision detection mechanism. At each timestep the agents recognise their environment and discover any possible collisions with nearby agents - at which point they will try another path if they are blocked. If all paths are blocked agents will simply schedule themselves immediately on the next timestep and try to move again, at which point they will resume scheduling themselves further in the future to simulate the different speeds of travel.

Deadlocks will therefore not occur with this collision detection mechanism. Agents which are blocked will "wait" until other agents have left the immediate area - which in practice will not result in a deadlock but a wait of a few timesteps for the agent. The only way a full, simulation-wide deadlock can occur is if the number of agents exceeds the grid locations within a simulation. In this situation each and every track would be saturated with TAs, however since the simulation environments are generally quite large, a huge number of agents would be required for this situation to occur. Most simulations will handle hundreds of agents without any such problem, this issue would only arise in smaller maps and as long as the user is sensible with the number of agents per simulation there is nothing to worry about.

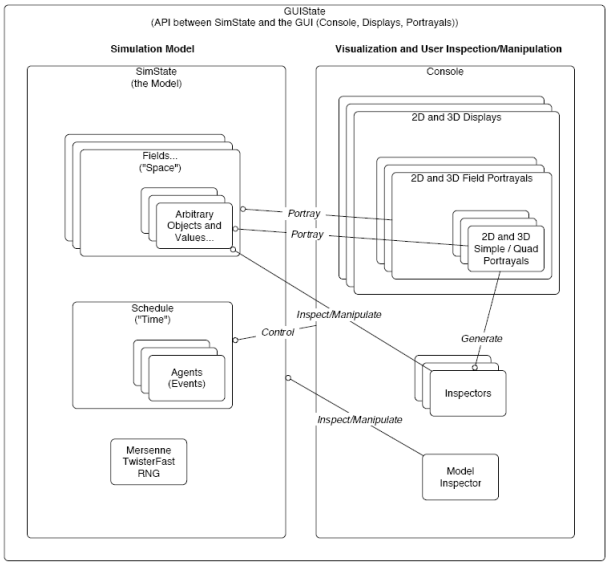


Figure 4.1: MASON

4.2.1 Initial Behaviour

The initial behaviour when the agents first enter the system is random. A simple job requesting mechanism was implemented in which nodes request agents to come pick up their packages. However this system did not prove to bear a significant impact on the results and was abandoned.

The initial random behaviour which is adopted by all the following protocols proves to be very efficient in getting agents to navigate towards nodes and start transporting packages. For this reason we decided to spend more time on developing efficient routing protocols than worrying about the agents' initial behaviour. However, a more complex job requesting system is discussed in chapter 8, to be implemented if more time was given for the project.

4.2.2 Stochastic Greedy Geographic Routing Protocol

This is the first protocol to be successfully implemented. Before deciding on using the A* routing algorithm, a simple geographic scheme was adopted so that the agents could navigate through the world. By geographic, we mean that whenever the agent makes a decision it will first locate the goal and depending on its own current geographic location within the world, make a decision about which path is best to follow. For example, if the agent realises that the goal lies south of its own location, it will attempt to move south, this can be seen from figure 4.2 - the blue nodes are the goals and the arrows specify the direction which the agent will take.

If there is a choice between two directions, such as when the goal is located in the north western part of the map - the agent will pick one of the two paths at random. Since this is a "greedy" protocol, it will attempt to find the shortest

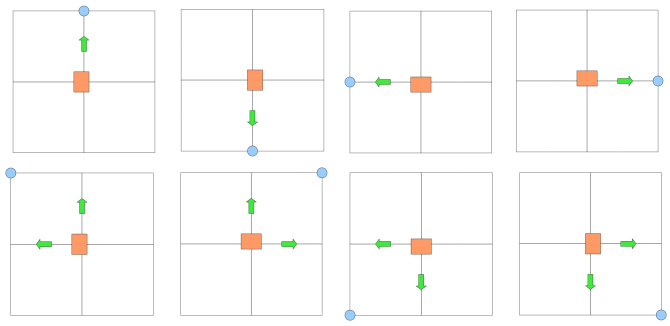


Figure 4.2: Geographic Routing

path towards its goal, disregarding any other paths through the network.

Because of the nature of this routing scheme, it is possible for the agent to navigate towards its goal but get stuck if the map is designed in such a way that being "close" to the goal does not necessarily result in an agent ever reaching that goal. This can be seen in figure 4.3 where the agent continues to get closer to the goal, but the actual optimal path requires the agent to get further from the goal before finally reaching it in the end.

This is why the stochastic factor was introduced in the second iteration of this protocol. It would prevent agents from always choosing the shortest path and sometimes go down a random path (with the default behaviour being set to 10% of path decisions being random), causing the agent to eventually reach its destination even if the map is too complex to achieve reasonable performance with the geographic routing scheme. We note that on the warehouse grid map which is detailed in section 4.4.3, this protocol performed in a very similar fashion to the more complex protocols which follow.

4.2.2.1 Machine learning

Even though the protocol will detect collisions and reverse its direction as a result, it is often the case that the agent will try the same path again after reversing from a collision with another agent even though the other agent could potentially still be in the way. This behaviour is greedy and can sometimes be the ideal situation if the nearby agent that was blocking the way has now left the area (a common occurrence near stations). Although the stochastic element of the protocol will often force the agent to try a different path before converging on the goal once more, the protocol will have tried the same path several times before the stochastic factor comes into effect. For this reason we devised a machine learning element to this protocol.

- The protocol will record how many times it detects a collision with nearby agents.
- If the agent has been blocked by other agents too many times per schedule steps, its stochastic factor will increase by 10%.

Algorithm 4.1 Detailed pseudocode of the SGGR protocol

```
find unobstructed directions
recalculate speed

if unobstructed directions = 0
   increment blocked count
   schedule self for the next timestep

// unobstructed directions > 0
// start random behaviour

else if has job = false
   if we found a station
      pick up packages
      schedule self with loading time
   if we are on a node
      pick a random direction
      schedule self according to speed
   else we are on a track
      attemp to travel forward
      reverse if an agent is in the way
      schedule self according to speed

// end random behaviour

else we have a job
   if we are on our goal
      unload packages
      schedule self with unloading time

// begin find goal

   else if we are on a node

      find best direction according
      to geographic location of goal

      //
      // machine learning
      //

      if blocked count too high
         choose random direction
         schedule self according to speed
      if able to go best direction
         go best direction
         schedule self according to speed
      else
         choose random direction
         schedule self according to speed

   else we are not on a node
      if able to go forward
         schedule self according to speed
      else
         reverse
         schedule self according to speed

// end find goal
```

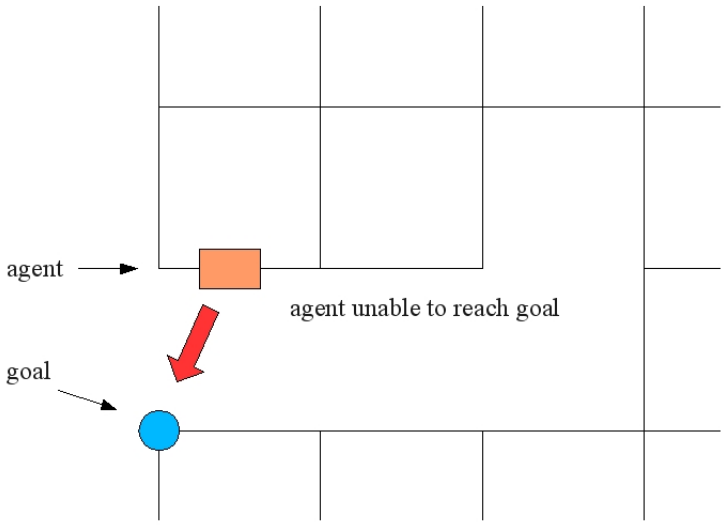


Figure 4.3: Disadvantage of the Greedy Geographic protocol

- This will increase the likelihood of the agent trying a path at random and therefore moving away from a problematic path. If the agent continues to be blocked, the stochastic factor will continue to increase until a user specified threshold.
- If the agent is now clear of other agents, it must now reduce its random decisions so as to converge towards the goal once more. The stochastic factor will therefore decrease if collisions with other agents have been reduced per unit time.

4.2.3 Precomputed A* Pathfinding Protocol

The A* pathfinding algorithm proved to be extremely efficient at navigating through the warehouse. In the initial stages of this protocol, a very naive implementation of a recursive pathfinding algorithm was used. This recursive algorithm turned out to be too inefficient and was later dropped in favour of the A* algorithm.

The full listing of the A* algorithm that was implemented for this project can be found in appendix A.1 and the pseudo code that was used for the implementation can be found in chapter 2.

Because we had already designed the bulk of the project by the time this protocol was implemented, it was impractical to reuse our node objects to implement the A* routing algorithm since each node requires to hold G, H and F values which specifies the quality of the chosen path. For this reason we created an overlay network consisting of A* nodes, which themselves map one-to-one to

Algorithm 4.2 Brief pseudo code of the precomputed A* protocol

```
find unobstructed directions
recalculate speed

if unobstructed directions = 0
   wait

// unobstructed directions > 0
else if has job = false
   behave randomly until find a node

else we have a job
   if we are on our goal we unload packages

// begin find goal

   else if we are on a node

      //
      // query path matrix for next decision
      //

      if able to go best direction
         go best direction
      else
         choose random direction

   else we are not on a node
      if able to go forward
         go forward
      else
         reverse

// end find goal
```

actual nodes within our simulation. This was possible due to the fact that we assigned unique identification numbers to each node in the simulation network, these node IDs can be seen on the screenshots of the 2D visualisations of the simulation runs.

4.2.3.1 Path-Matrix

In this protocol, before the agents are spawned into the world the A* search algorithm is run from each node pair. Using the optimal paths created by the algorithm we then build a "path-matrix" which stores these optimal paths for later retrieval. Once this process is over the simulation can now start and the agents query this matrix to select the ideal path from node to node. This approach to pathfinding is advantageous because the A* algorithm need not be run again once the simulation is in motion, saving computation time. The drawback is that new paths are not calculated in real time hence the heuristic element of the A* algorithm cannot be used to find better paths due to changes in the environment during a simulation.

Much like the previous protocol, this protocol is greedy in that it tries the

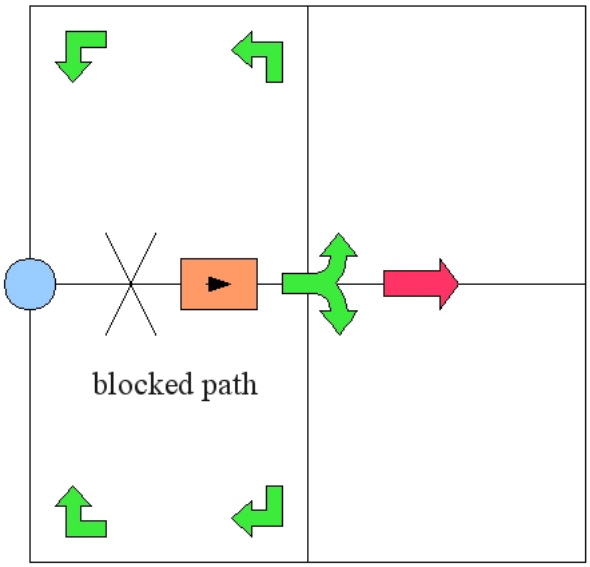


Figure 4.4: Agent Avoidance

optimal path even if it is blocked by another agent, reversing and trying again. The next section specifies the measures we took in implementing protocols which avoid agents.

4.2.4 Agent Avoidance

It is often the case that agent will detect each other going through the same path, this is especially the case in simulations with a large amount of agents. For this reason agents will often have to reverse and try another path. The two previous protocols greedily tried the same path until they were either unable to enter the path (stuck at a node), or the agent that was blocking the path moved away as itself could not enter the path.

Agent avoidance is a scheme by which agents will now almost never try the same path again if it is blocked.

- If an agent has had to reverse, once it reaches the node it entered the path from, it will attempt to go any other path than the one it just tried.
- The agent will try a path on its left or right - which is the next ideal path as it will lead the agent to navigate around the blocked path, and if those paths are unavailable it will continue forward - essentially going further away from its objective. This is illustrated on figure 4.4.

4.2.5 Real-time A * pathfinding

The A* algorithm proved to be very efficient and even for a large amount of agents, the algorithm was fast enough to implement real time pathfinding. Not

Algorithm 4.3 Brief pseudo code of the real-time A* protocol with agent avoidance

```
find unobstructed directions
recalculate speed

if unobstructed directions = 0
   wait

// unobstructed directions > 0
else if has job = false
   behave randomly until find a node

else we have a job
   if we are on our goal we unload packages

// begin find goal

   else if we are on a node

      //
      // perform A* search for next best decision
      //

      if unable to go best direction
         choose random direction
      else if that random direction means reversing

         //
         // agent avoidance
         //

         attempt to turn away from path
         move away from goal as last resort

   else we are not on a node
      if able to go forward
         go forward
      else
         reverse

// end find goal
```

only could agents recompute paths in case they found themselves unable to proceed through a path that was previously computed, but the heuristic function of the A* algorithm could now be used to find the most optimal paths based on different characteristics.

4.2.5.1 Heuristics

By the time this protocol was implemented in the project, we were able to include a congestion characteristic within the heuristic function of the A* search.

- The congestion in our simulations is modelled on the speed that agents travel through the network. A slower agent produced more congestion for the path on which it currently is.

- Congestion for a path decreases at each timestep of the simulation, preventing paths which were previously heavily congested to stay in this state.

We note that in reference to the pseudocode in algorithm 4.3, all A* heuristics are calculated when the agent performs the A* search. The agent itself does not see any of the implementation of the heuristics as it need not have this information. We tested this congestion heuristic in chapter 5.

4.2.6 Overriding

As a final detail about protocols, we note that almost all types of behaviour specified in this section can be overridden to suit the user's need. For example, the way we modelled speed calculation - as specified in section 3.8 - can be changed so that the relationship between speed and cargo carried is not linear. Since we designed the protocols to be extensions of the base TA class, all the methods in this base class can be overridden in usual object-oriented fashion.

4.3 Visualisation

MASON provides both 2D and 3D visualisation capabilities. As stated in chapter 3, the visualisation of the simulation was one of the main goals of this project. Not only was visualisation useful for creating the environment in which the agents would operate, but it was also crucial in checking agent behaviour and finding optimisations in their behaviour. We note that MASON decouples a simulation from its visualisation. This is a very important features of MASON as it allows us to run simulations without worrying of their graphical appearance. We can then implement a graphical view of the simulation and modify it as needed without changing any of the implementation details of the simulation.

4.3.1 Console Runs

At the most basic level, the visualisation of the simulation is simply the results that it returns. By using the statistics class from section 3.9 we can record the quality of the protocols by how quickly they transport packages to and from stations. We can also display messages on the console when certain events happen, allowing us to track different characteristics such as how the stochastic element of the first protocol changed as the simulation was executed. This type of simulation is the fastest because the visual elements are restricted to the messages that it returns, there is no fancy graphical element - hence this was ideal to record results or do some initial testing.

4.3.2 2D Visualisation

The 2D visualisation of the simulations is the most useful type of visualisation when designing the maps or agent behaviour. In this section we refer to the term "portrayal" as an entity which displays information about its underlying data structure on screen.

Several 2D grids are used to represent information from the simulation. The MASON framework then superimposes these grids on top of each other and allows their values to be represented on screen. We now list the different types of portrayals that we used for the 2D visualisation.

Roads — The roads portrayal simply displays the paths which the agents can take between each node. This portrayal uses the information stored in the 2D grid specifying the network.

Effects — The effects portrayal is used to show the trails of the agents. This shows the previous location of the agents, making it easy to distinguish how agents are behaving when deciding on a new path.

Congestion — This portrayal displays the congestion element of the simulation. This is particularly useful for observing the A* heuristic which takes into account the congestion of the network.

Paths — The paths portrayal displays the chosen path by the A* algorithm. This was created to help development of the algorithm and debug any issues.

Text — The portrayal which displays text in certain areas of the grid, as can be seen in section 4.4.

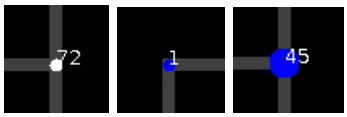
Trucks — This portrayal showed the TAs on screen. As we explained in the design chapter of this report, the TAs are stored on a different data structure than other objects but map one-to-one to a grid location. This made it easy to display the agents interacting with the world.

Stations — Likewise for node, station and the dispatcher objects - although they are stored on a separate field from other objects, they also map one-to-one to the 2D grid and are represented using their own custom portrayals.

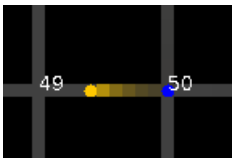
We now introduce visual detail of how each object was represented.

4.3.2.1 Visual Guide

Stations — Here are the graphical representation of stations with different amount of packages ready for transport. White stations are empty stations whereas blue stations have generated at least one package. Stations which have generated many packages that have not yet been transported grow in size as the simulation progresses.

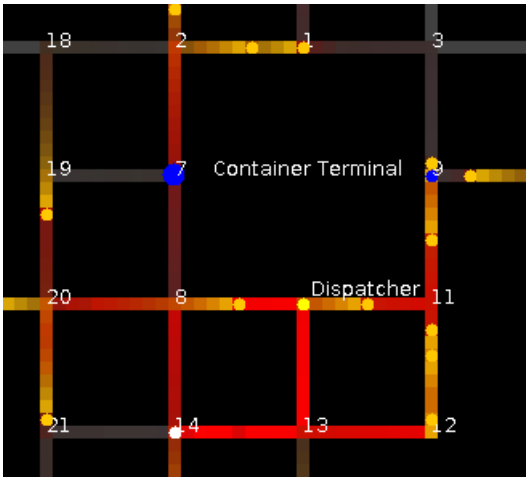


Agents — We represented agents as oval orange objects with a trail showing the agent's previous location.

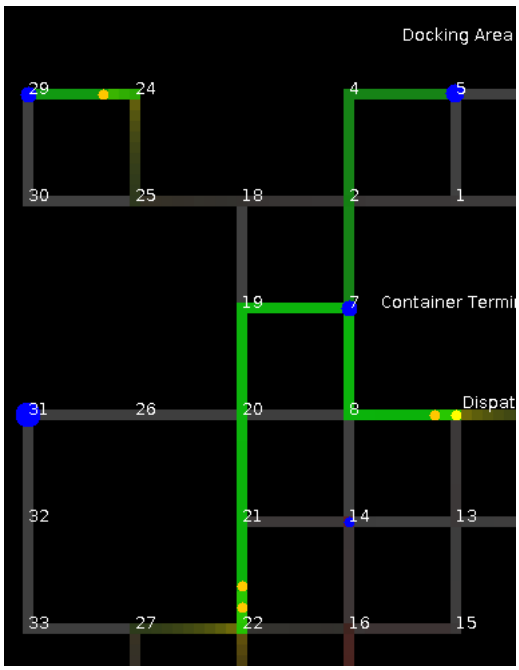


Dispatcher Since the dispatcher is also a node in the network we represented these objects just like stations but coloured yellow. We also inserted a text object to make identification clearer.

Congestion We modelled this visual effect in red, where a brighter red signifies higher levels of congestion.



Paths Green outlines were implemented whenever an agent calculates a new path for protocols which use the A* algorithm.



4.3.3 3D Visualisation

For the 3D visualisation of the simulation, MASON uses the Java3D libraries. Using this functionality we implemented a 3D representation as can be seen in figure 4.5. Much like in the 2D representation, agents are coloured orange. Stations are the blue spheres and the white spheres represent the nodes connecting each path.

In figure 4.6 we can observe the congestion as a height map as was detailed in chapter 3. This makes it easier to notice bottle-necks within the network. Slower agents produce the much larger peaks in congestion that can be seen from the figure. Faster agents will produce negligible congestion unless they are stopped at a station, at which point the congestion will often peak only at this location.

The final 3D visualisation that we implemented was a "lifetime" height map. This height map records each location that agents visit during the simulation run. At the end of the simulation run it is possible to view this height map and locate any bottle-necks within the network layout. Figure 4.7 shows the height map at the end of the simulation. The higher the peaks, the more times agents visited the location. The highest peaks will show the location of stations, since agents must spend time on these locations to load and unload packages. The smaller peaks however, show where agents spent the most time while they were navigating the world - indicating potential bottle-necks.

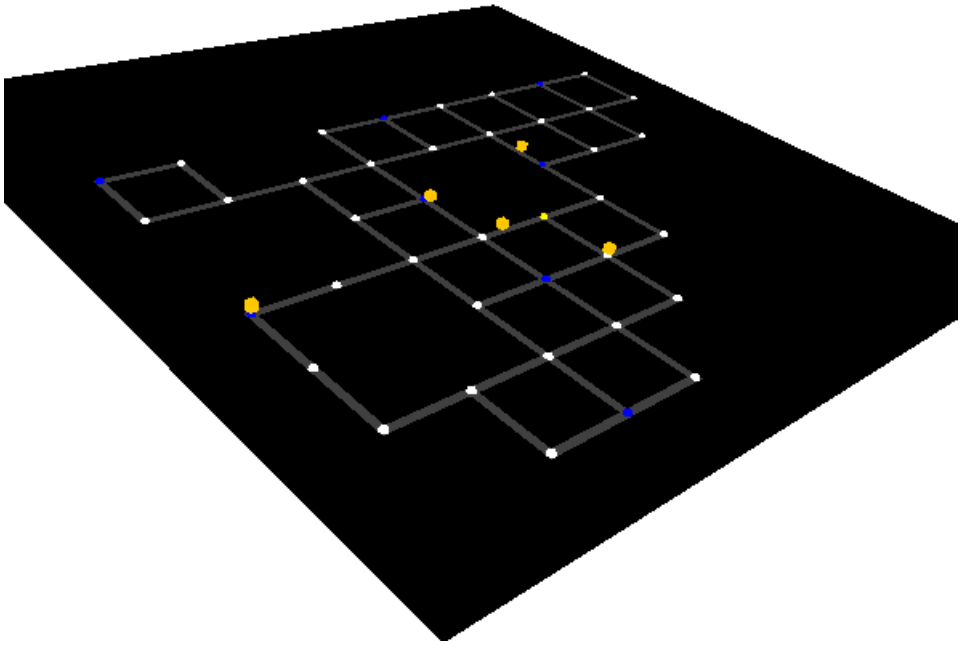


Figure 4.5: 3D Visualisation

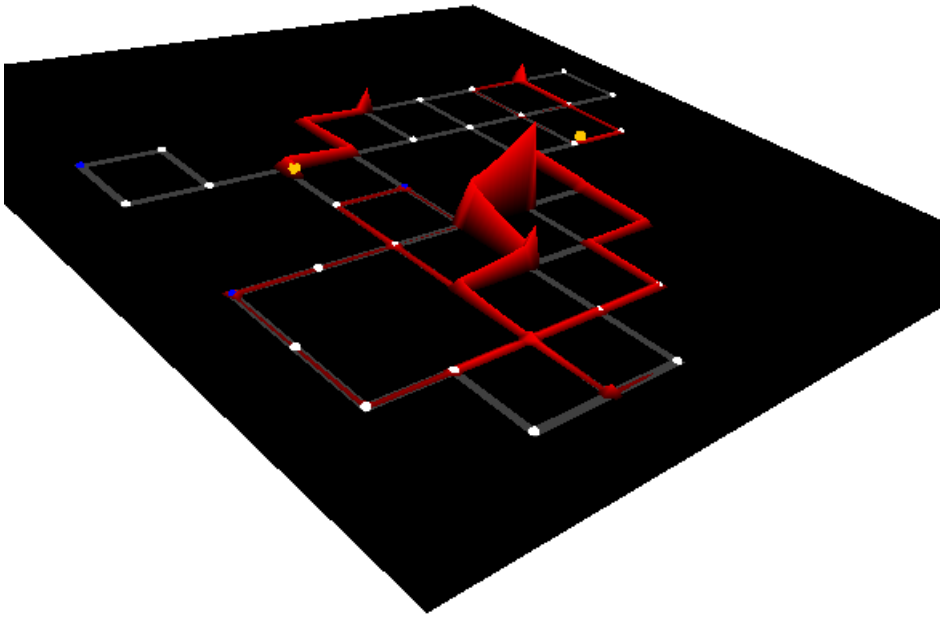


Figure 4.6: Congestion height map

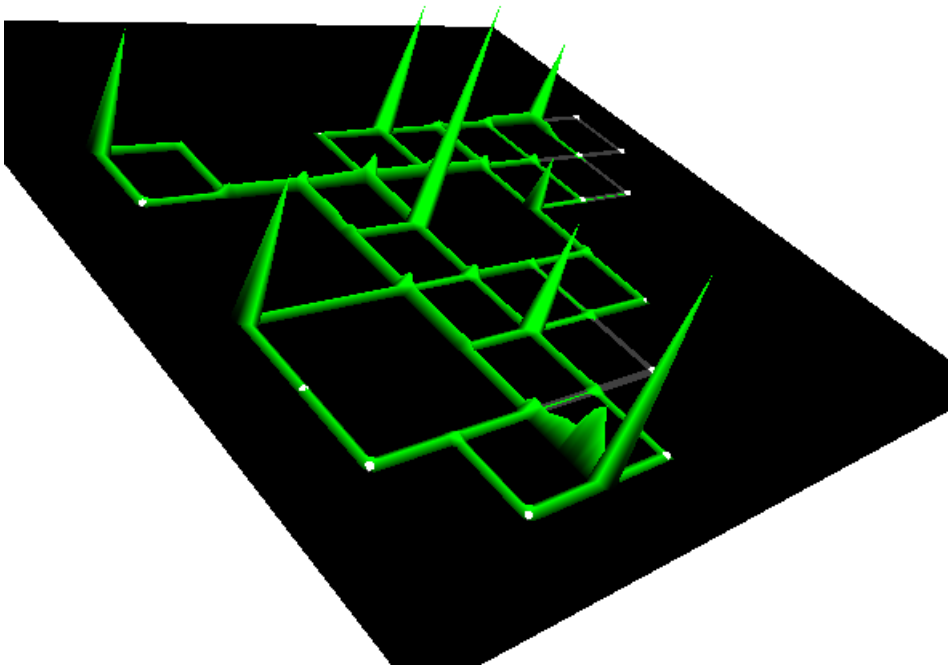


Figure 4.7: Lifetime height map

4.3.4 Speed of the Simulation

The speed of the simulation can either be slowed down manually (to allow a greater amount of accuracy) or modified using the MASON console. The MASON console will slow down the simulation and let you specify how many seconds (or fractions of a second) to wait between each timestep, resulting in a slower simulation. However slowing down to even a fraction of a second per timestep can sometimes be too much and result in a very jerky simulation.

By slowing down the simulation manually by using Java thread functionality, an even smaller delay can be imposed, resulting in fluid simulations which are slow enough to observe the interactions between the agents. This is very much appropriate for small simulations (especially in the debugging stage of a protocol), where most computers are powerful enough and will therefore zoom through simple simulations. When the number of agents increases however, the simulation will often slow down simply because of the expensive cost of simulating so many agents on screen - at which point a better computer is required if one wants to visualise the simulation fluidly.

4.4 Map Examples

In this section we introduce the most important rail network topologies that were used to record results for our experiments. We note that the implementation of

a more convenient GUI editor would be an applicable extension to this project, which is outlined in chapter 8.

4.4.1 Debugging Maps

Two smaller maps than the ones following were created in order to debug agent behaviour, these can be found in appendix B. It was important to test the initial protocols on a smaller environment, especially when designing the routing algorithms. We found that if the routing algorithms worked on small maps, bigger maps were not a problem. It also made it easier to debug potential problems whether with visualisation, agent behaviour or the routing algorithms.

4.4.2 Container Terminal

This map is based on the container terminal layout as can be seen in certain ports like the ECTerminal in Rotterdam, Netherlands. This was an important map to implement as like the following warehouse map, it is based on a real life example. By introducing the TAs in a realistic situation we increased the relevance of our results with the real world.

Figure 4.9 shows the full map using 2D visualisation. Labels were introduced to reflect the actual location of certain elements in a real container terminal such as the docking area, the centre of the terminal which holds all the containers and the port exit from which the containers enter and leave the terminal. The TA dispatcher was placed in the centre of the map to allow the TAs to quickly flood the system and start moving packages to and from stations.

This layout is fairly complex and involves bottle-necks, this makes it difficult for the TAs to easily navigate the world. This type of sparse topology is ideal to distinguish the quality of the agent protocols. Simpler protocols will obviously have more difficulty navigating through this map than other more intelligent protocols. This map is also useful to see bottle-necks using 3D visualisation.

4.4.3 Warehouse Grid

The warehouse network topology is taken directly from real warehouse layouts. To make maximum use of space real warehouses have a grid-like layout with wares being stored in all locations within the warehouse. This network topology is exceptionally easy for agents to traverse since it is so densely populated with nodes. At each node agents will usually have 4 different paths that they can choose from. As we will see later, all protocols scale very well on this kind of map.

Like with the container terminal map, the dispatcher was placed right at the centre to allow agents to quickly traverse the network and start transporting packages. This can be seen on figure 4.11.

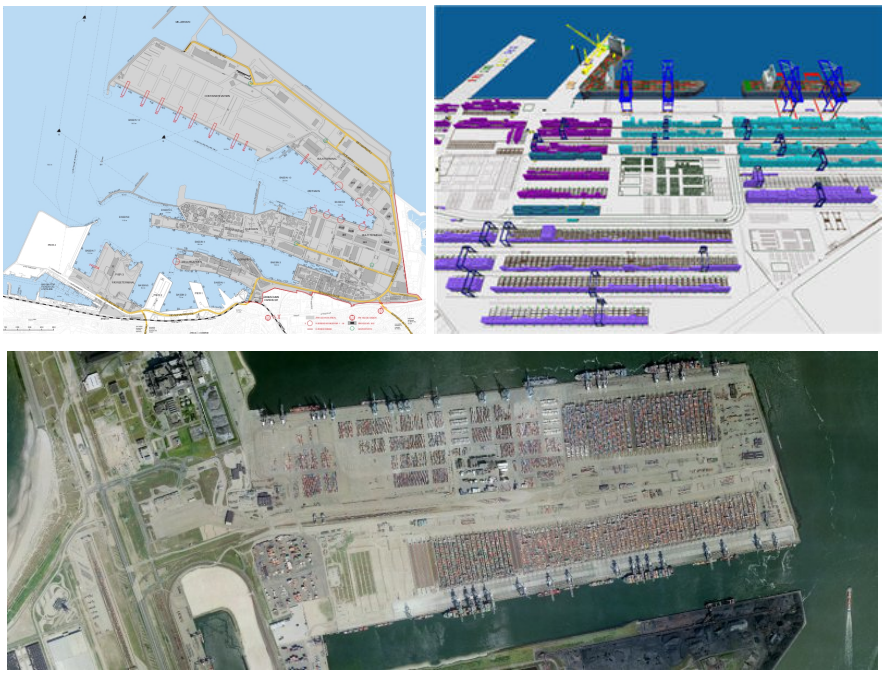


Figure 4.8: Container Terminal Layouts

4.4.4 Cross World

This map is not based on any real life example but was simply created to test the performance of the agents on a different layout. By having stations in the centre of the cross as well as at its tips, the TAs had a lot of work to do to traverse each side of the map as there is no other choice but to return to the centre of the cross before continuing to another tip. The map is shown on figure 4.12.

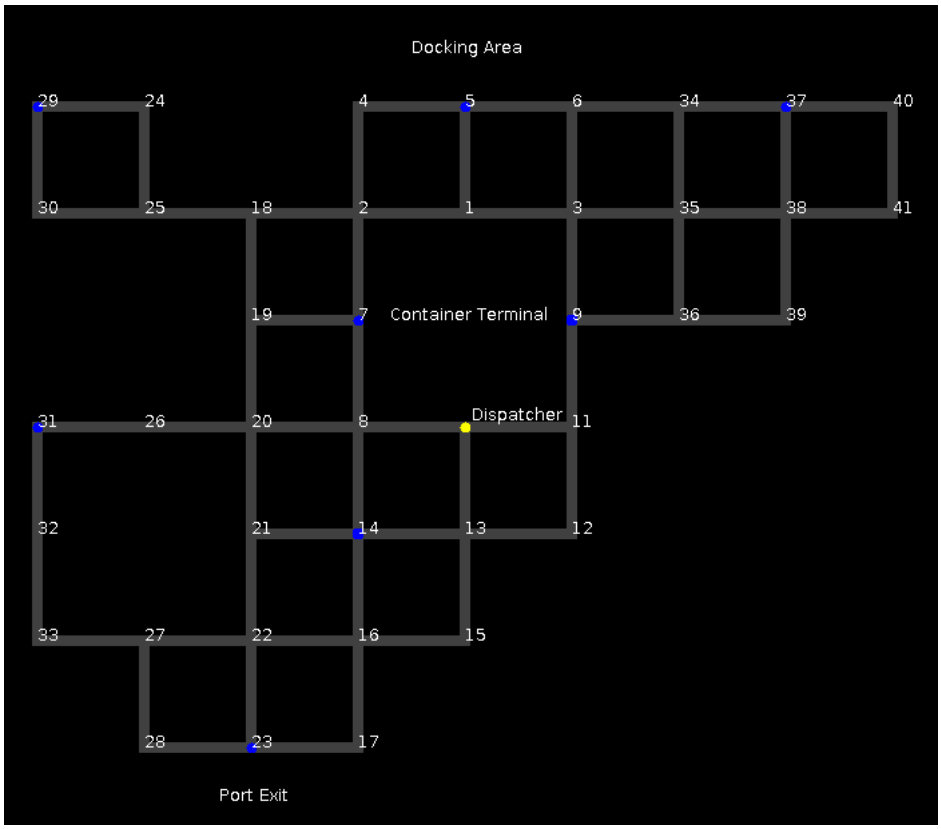


Figure 4.9: Container Terminal

Figure 4.10: Warehouse layout

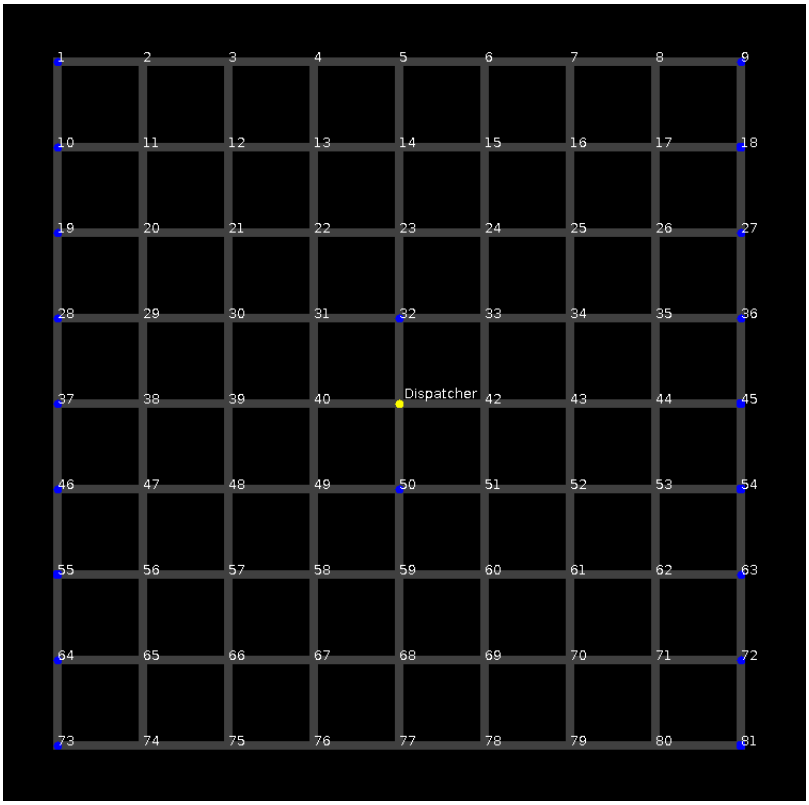


Figure 4.11: Warehouse grid

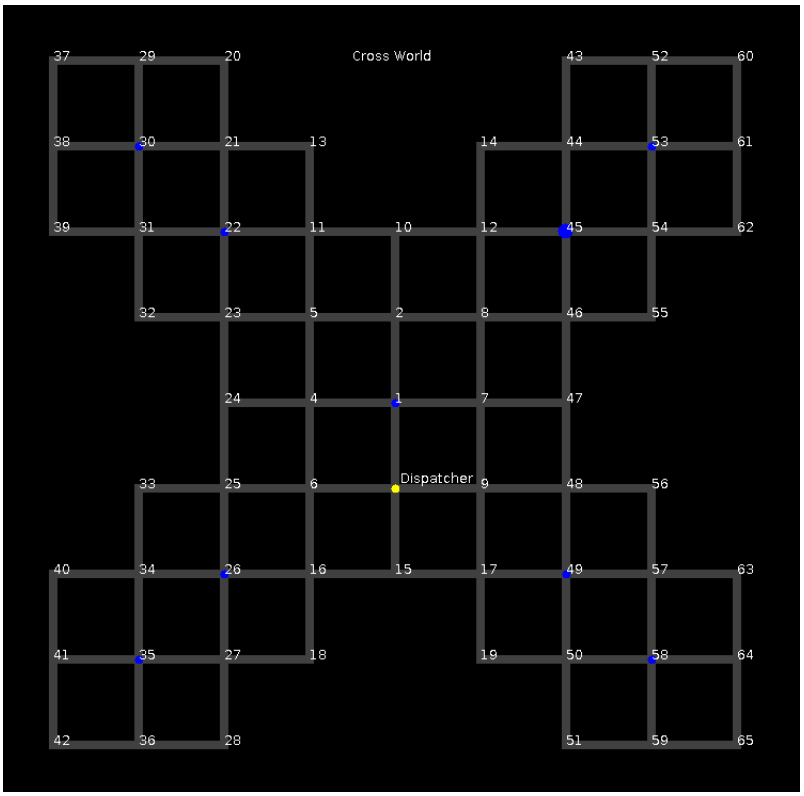


Figure 4.12: Cross world

Chapter 5

Experimental Design

5.1 Parameters

In the following experiments we altered parameters which control either agent or environment behaviour. There are many possible parameters to tweak, we list a sample of these below:

- Number of TAs.
- Number of packages per simulation.
- Arrival rates of packages.
- Maximum weight which the agents can carry.
- Stochastic parameters for the relevant protocols.
- Heuristic parameters for the A* routing algorithm.

Each of these parameters will affect the environment and in turn the performance of the protocols. We will select several of the above parameters and modify them while measuring the quality of the protocols using the criteria outlined in section 3.9. We will outline recommendations about which protocol to use for which warehouse layout and with what parameters. This can be found in chapter 7.

5.2 Experiments

Each experiment was performed on the three main maps of the project:

- Container Terminal
- Warehouse Grid
- Cross World

5.2.1 Agent Population

In this first experiment we altered the number of agents operating in one simulation run. We kept the number of packages constant - likewise for other simulation parameters. Changing too many parameters per experiment would make it difficult to extract meaningful information from the results.

We repeated the simulations 10 times and took the average of all these runs so as to minimise the effect of outliers on our results. For each batch of simulations, we recorded the average waiting time for a package (the time the package spent on the system) and plotted these results against a growing number of agents. We then tested the same set up against different network layouts to witness how well protocols performed when faced with different environments.

- We predict that with a low number of agents interacting with the world, the average waiting time will be very big. This is due to the fact that packages will be generated faster than agents can realistically transport them from station to station.
- As we increase the agent population, we should see a great improvement in the average waiting time - this is simply because the agents can now transport packages faster than they are generated.
- However as we keep increasing agent population we should see a drop in performance, especially on maps which a sparser node layout, since agents will now impede each other due to there being so many agents in the environment. They will block each other's paths and increase the average waiting time for a package.

5.2.2 System Load

For this experiment we altered parameters which would simulate an increase in system load. By this we mean the rate of package arrivals. The more often packages arrive, the more they will wait at their respective stations before enough agents come and transport them.

- The natural prediction is that as the rate of arrivals increases, the average waiting time obviously increase as well. We will see how this scales when there are so many more packages than the agents can carry.
- In this situation we will expect protocols which are better at navigating the maps to scale better as packages keep on arriving on the system at faster rates.

5.2.3 Optimal Parameters

As many of the simulation characteristics can be altered by changing the parameters, it also makes sense to experiment with different values in the parameter space and see how these affect the overall performance of the agents or how the environment responds to the changes.

We will test both the stochastic parameters for the SGGRP agent from subsection 4.2.2 and the heuristic parameters for the real-time A* agent.

- For the stochastic parameters, we predict that as the randomness of the agent is increased, the better the agent will perform in sparse network layouts like the container terminal map. However we also predict that if the parameter which specifies the upper bound for the randomness of the agent is too high, then the agent will then perform poorly - as if the agent is too random, although it will be able to move away from blocked areas, it will also not navigate towards its goal efficiently.
- We will also test the congestion heuristic parameters for the A* algorithm, by modifying how much the congestion heuristic is "worth" - we will see whether having the congestion heuristic improves are impedes the performance of the agents.

Chapter 6

Results

For each parameter, we tested the agents on the Container Terminal, Warehouse Grid and Cross World maps. We now give graphical representation of these results. Please refer to chapter 7 for a detailed analysis of these graphs and the recommendations of which parameters are best to use for which map tested.

6.1 Agent Population

We begin by testing each protocol with a varying amount of agents on each of our three example maps, while keeping other simulation parameters constant. Since we are showing average waiting time in these graphs, the higher the curve, the worst the performance.

In the following graphs we ran simulations until the agents had transported 100 packages. We start with a low number of agents, increasing the agent population by 5 agents each time. Each simulation was run for a total of 10 times and the average of the simulations was plotted. As is usual, we plotted the standard error directly on the graphs to give the viewer an idea of the significance of the curves in relation to each other.

We use the following abbreviations when referring to the different protocols which we tested:

SGGRP Stochastic Greedy Geographic Routing Protocol

PA*RP Precomputed A* Routing Protocol

PA*RPAA Precomputed A* Routing Protocol with Agent Avoidance

RA*RPCH Real-time A* Routing Protocol with Congestion Heuristic

In the following tests, agents with the RA*RPCH used a weighting of 1.0 for the congestion heuristic unless otherwise specified. Agents with SGGRP used stochastic parameters of 10% minimum random behaviour and 50% maximum for when the machine learning is activated.

	Best Average	Waiting Time	
	Container Terminal	Warehouse Grid	Cross World
SGGRP	926.40	556.20	590.16
PA*RP	781.77	556.04	482.17
PA*RPAA	720.55	519.48	473.31
RA*RPCH	623.12	507.71	500.00

	Optimal number	of TAs	
	Container Terminal	Warehouse Grid	Cross World
SGGRP	25	45	40
PA*RP	20	45	35
PA*RPAA	40	55	30
RA*RPCH	35	30	35

Table 6.1: Best Performance

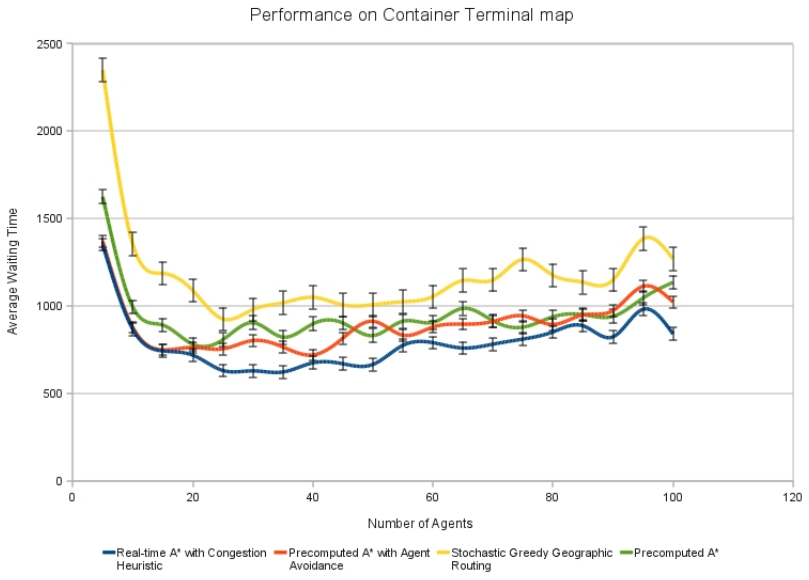


Figure 6.1: Number of agents increasing on Container Terminal map

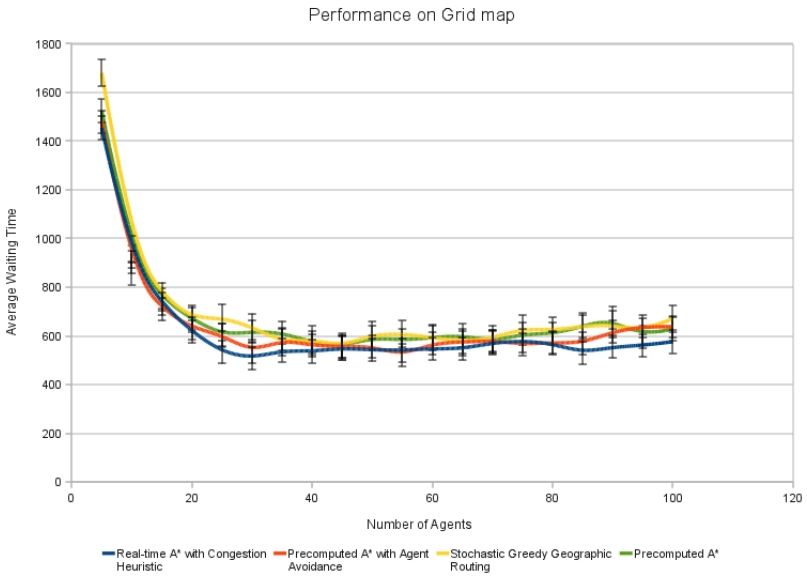


Figure 6.2: Number of agents increasing on Warehouse Grid map

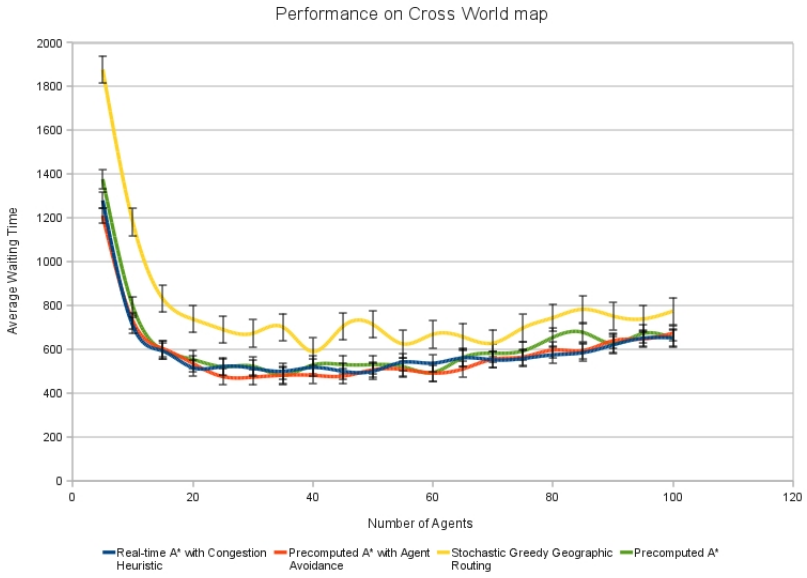


Figure 6.3: Number of agents increasing on Cross World map

One can clearly see that the performance with a low number of agents (each simulation started with 5 agents initially) is very poor. This makes sense as packages are generated a lot faster than the few agents can deal with them. This is indeed in line with our prediction in chapter 5 - we predicted that initial performance would be very poor and show a sharp increase as we increased agent population. We also predicted that performance would get worst as more and more agents are added to the simulation. For the Container Terminal map on figure 6.1, we do indeed notice this trend, with an initial drop in average waiting time, followed by a gradual increase.

We observe different behaviour on the different maps, with the Warehouse Grid map on figure 6.2, showing very similar performance for each of the protocols even though the number of agents is increasing. Notice that the Container Terminal map has a more significant increase as the number of agents increase, this is similar but to a lesser extent on the Cross World map. The Warehouse Grid map shows almost no increase, even when faced with 100 agents in the environment, we suggest reasons for this behaviour in chapter 7.

6.2 System Load

To test how system load affects different protocols we used an arrival rate from 0 (extremely fast) to 6 (fairly slow) for the generation of packages. More detail on how we implemented arrival rate using the exponential distribution can be found in subsection 3.4.3. We tested two protocols for this experiment, RA*RPCH (real-time A*) and SGGRP (geographic) - on the two case-study maps, Warehouse Grid and Container Terminal.

Each simulation was made to generate 100 packages with a population of 30 agents for each protocol. We repeated the simulations 100 times to get a more accurate view of the results. Since we did not compare all protocols it was feasible to repeat the simulations for so many iterations.

6.3 Optimal Parameters

6.3.1 Stochastic Thresholds

Figure 6.6 shows how agents with the SGGRP performed when the stochastic parameters were altered. We changed the maximum percentage of allowable random behaviour from 0% (no learning) to 90%. We observe a sharp dip from 0% random behaviour to around 20%, when the performance of the agents start to worsen.

We plotted 3 different agent populations: 25 agents, 65 agents and 85 agents. This was to check that with a lower number of agents the machine learning capabilities of the SGGR protocol would not often be activated. This is indeed reflected in the curves, with the 25 agent population (green) curve staying fairly level regardless of the maximum percentage of random behaviour allowed.

We also note that although the maximum allowable percentage of random behaviour might theoretically reach 90%, resulting in a very random agent, this

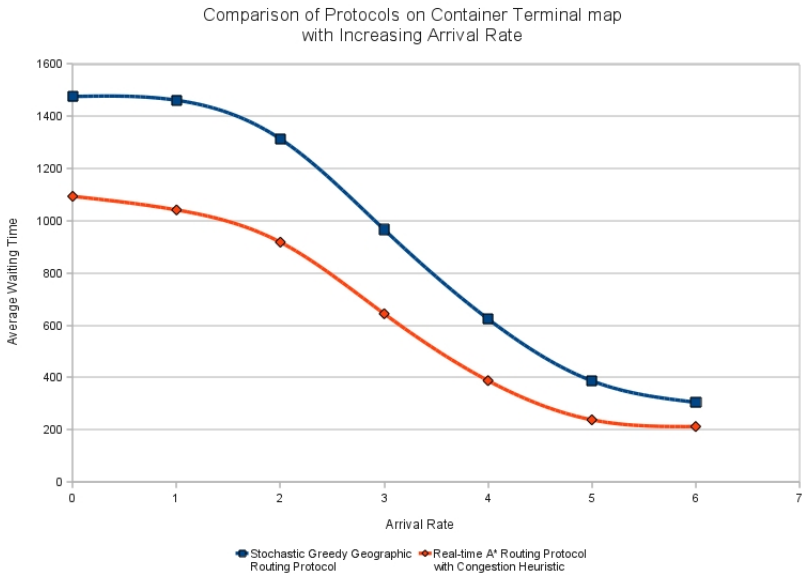


Figure 6.4: Increasing System Load on the Container Terminal

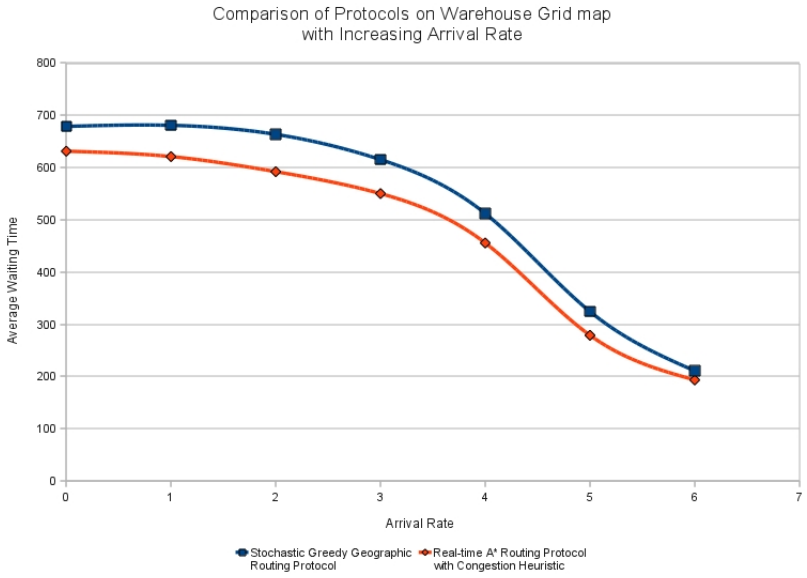


Figure 6.5: Increasing System Load on the Warehouse Grid

	Number of Agents		
	85	65	25
Best Average Waiting Time	1080.45	975.54	952.53
Best Parameter to use	20%	20%	10%

Table 6.2: Best Stochastic Parameters

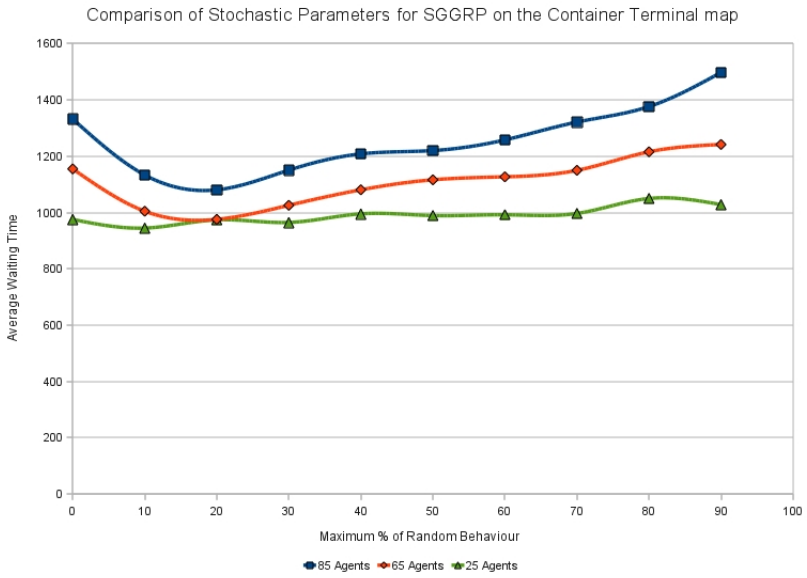


Figure 6.6: Comparison of Stochastic Parameters

behaviour would only occur if the agent has difficulty navigating the world and is very blocked often. Once the agent is unblocked the percentage of allowable random behaviour then decreases towards 10%, making the agent a lot more deterministic in its choice of paths.

6.3.2 Congestion Heuristic

In figure 6.7 we plotted a comparison of performance of the A* congestion heuristic as agent population increased. These simulations were also performed 10 times but to a lower resolution. We did not deem it necessary to perform the simulation in increments of 5 agents instead opting for 20. See chapter 7 for a detailed explanation of the results.

6.4 Example of Bottleneck Detection

We now give an example of the extra functionality which our implementation provides. By using a "lifetime" height map, we can record the locations which the TAs have visited throughout the simulation. We can inspect these lifetime maps after the simulation is over or at a late stage in the simulation and see

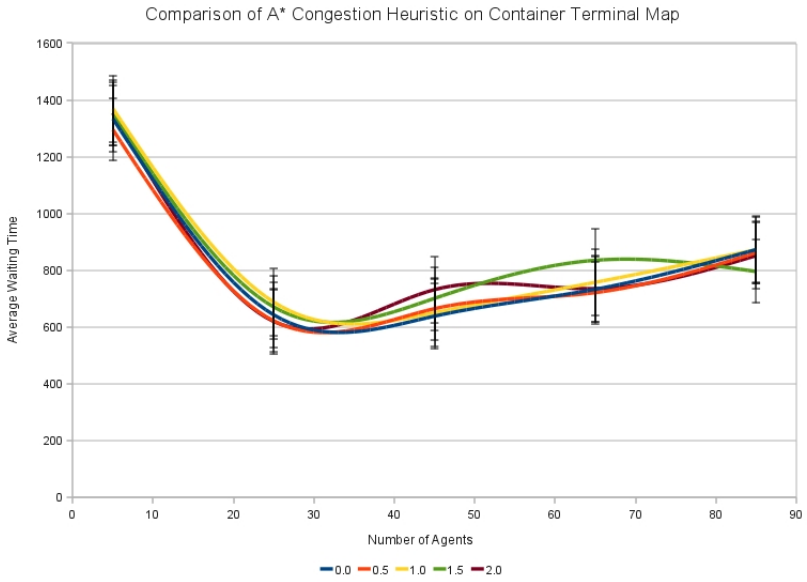


Figure 6.7: Comparison of congestion heuristic

where the agents have been the most. Peaks in the height map represent areas which agents were most concentrated around, these are usually stations where agents remained to load and unload packages. However if there are huge peaks compared to the rest of the map, this usually signifies that that particular area is subject to a bottleneck during the simulations.

In figure 6.8 we give an example of how this functionality is used in an obvious case. By taking the Warehouse Grid map and modifying it so as to cause a bottleneck between the two halves, we can see that where agents were most congested was the middle bridge of the two halves - where stations are on both sides.

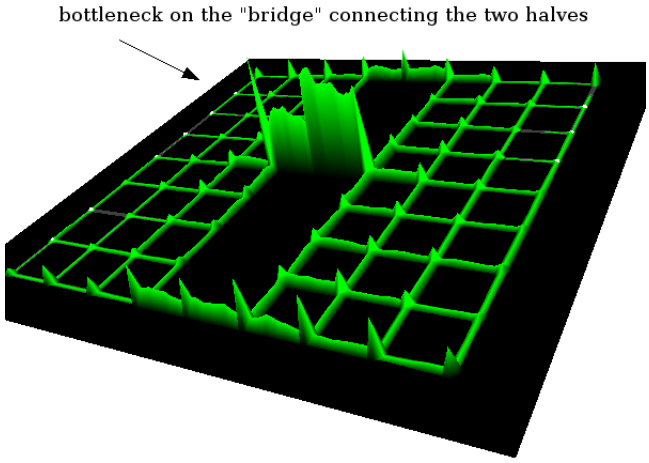


Figure 6.8: Bottleneck on a modified warehouse grid layout

Chapter 7

Evaluation

In this chapter we give a detailed analysis of the results obtained from chapter 6. We give recommendations of which protocols and what parameters to use to obtain the best performance for the three maps which we tested as well as insight into the behaviour of the protocols when faced with different environments.

7.1 Recommendations

7.1.1 Container Terminal

Container terminals generally have more complex layouts than warehouses (figure 4.9). For this reason the A* protocols performed a lot better than the SGGRP - the A* search guarantees an optimal path towards the goal whereas SGGRP only attempts to get closer to the goal if it can, deviating every once in a while due to its stochastic factor. The SGGRP therefore scales quite badly compared to the other protocols on this map.

- Like with other potentially complex layouts, we recommend the A* protocols to be used with this map.

The best protocol was RA*RPCH, we believe this is due to it being able to recalculate paths in real time rather than use precomputed paths like the other A* protocols do. By recomputing their optimal path at each node, agents which are blocked can easily find ways around these blocked paths.

Indeed, both precomputed A* protocols performed fairly similarly to each other (see figure 6.1). With the agent avoidance protocol performing slightly better than its simpler cousin. We note however that SGGRP required 25 agents to achieve its best performance, and the simplest A* protocol PA*RP only required 20 agents. In comparison RA*RPCH which achieved the best results required 35 agents.

- We therefore recommend the use of RA*RPCH if the map is used with perishable packages, as this protocol achieved the best performance but

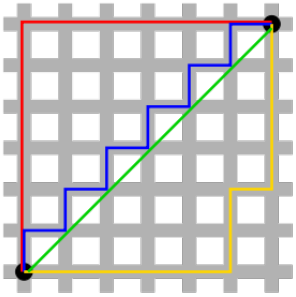


Figure 7.1: Taxicab Geometry

with a slightly increased number of agents. If the cost of each extra agent is too high, we also recommend the use of PA*RP (with only 20 agents for best performance) if the user is willing to trade a greater average waiting time for a reduced number of agents.

7.1.2 Warehouse Grid

The Warehouse Grid layout is simplest of all the maps that were tested in this report (figure 4.11). It is very densely populated with nodes and allows maximum freedom for the agents to move around. In fact it is a kind of "taxicab" geometry, as shown in figure 7.1 where the red, blue and yellow paths each have the same length and are each a shortest path.

This therefore means that the A* algorithm will usually choose the blue path due to the diagonal distance being the shortest. On the other hand, the geographic protocol will uniformly choose between the red, blue and yellow paths. This will often result in the geographic protocol choosing different (but equally good) paths compared to the A* algorithm. In fact, it turns out that SGGRP performs very similarly to PA*RP. We can see this in figure 7.2 - apart from the beginning of the curve where SGGRP performs marginally worst than PA*RP, the other points are well within the error margin.

- For this map, it seems that using a basic protocol (SGGRP) does not seriously impact the overall performance of the agents. SGGRP therefore becomes a viable alternative in this sort of map if the designer does not wish to spend too long developing complex interactions.

7.1.3 Cross World

Although this network topology (figure 4.12) would be rarely used in practice, it does show the performance of the agents when faced with different kinds of networks.

From figure 6.3 we can see that the A* protocols once again perform the best in this more complex map, however it is unclear from the results which of the A* protocols scales better on this map due to the errors overlapping over each

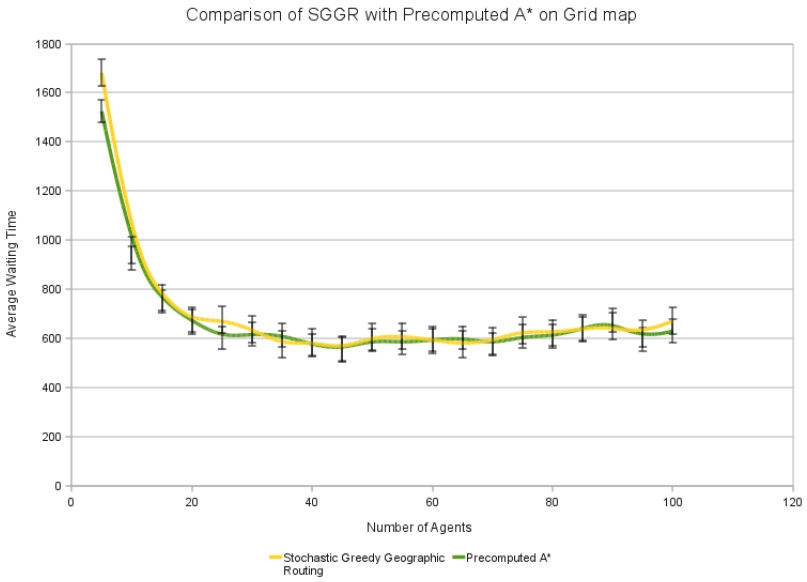


Figure 7.2: Comparison of SGGR with precomputed A* routing

curve. There is a hint that agent avoidance does increase the performance of the protocols as both protocols that used it seemed to perform better overall than the simple precomputed A* search. However the differences are not so important so the overall recommendation for this network is indeed to use agents with A* search, whether with complex agent avoidance behaviour or not.

- In this map we would use a total of around 30 to 40 agents to achieve the best performance - in fact the best performance was achieved by RA*RPAA with 30 agents (see table 6.1).

7.1.4 System Load

We had predicted that increasing system load would of course increase average waiting time and this was clearly shown in figures 6.4 and 6.5. The higher the arrival rate (0), the higher the average waiting time. As load decreases (towards value of 6) so does average waiting time. This is because packages are not generated as often and while the ones that were previously generated are being transported, the arrival rate is such that not many packages are generated and left waiting.

Since we kept the number of packages to be generated constant, the highest average waiting time is the upper bound when using 100 packages. If we had used a different amount of packages we would see the same trend (the same shape of the curve), but with a higher upper bound on the average waiting time.

Comparing both the SGGRP and RA*RPCH allowed us to see how much better A* protocols are at transporting packages in general. We note that like

previous results, the Warehouse Grid map shows a smaller gap between the two protocols, whereas the more complex Container Terminal map has a wide gap between the performance of the protocols.

We also note that on the Container Terminal map, as the arrivals are made to be less frequent, the average waiting times for both protocols start to become more and more similar (see figure 6.5). We believe this is because if packages are not generated rapidly, the SGGRP still manages to carry out the transportation in a reasonable amount of time. However this points that the A* protocol is much faster than SGGRP at transporting packages when the system load is very high.

- We therefore recommend the use of RA*RPCH for all simulations in which package arrivals are very frequent.
- As noted previously, SGGRP does scale pretty well on the Warehouse Grid since the layout is very dense - however its performance on complex maps with a high arrival rate is very poor.

7.1.5 Stochastic Parameters

From our predictions in chapter 5, we can indeed see that figure 6.6 shows that allowing the agents to behave randomly improves their overall performance. This is due to the fact that their randomness allows them to behave like if they had implemented the agent avoidance mechanisms. By abandoning the blocked paths and choosing a random path instead, the agents can unblock their way out of tricky situations.

However we also verified that if the maximum percentage of random behaviour is too high (above 30 or 40%) then the agents starts to perform less well due to the fact that they now behave *too* randomly to be efficient at transporting the packages. As mentioned in the previous chapter, a low population of agents will not activate the machine learning capabilities of the SGGRP and therefore increasing the stochastic parameters will not affect the performance of the agents significantly.

7.1.6 Congestion Heuristic

We can see from figure 6.7 that our congestion heuristic for the A* search did not significantly affect the results, with parameter 0.0 (when congestion was ignored) often performing better compared to when the heuristic was given more weight.

We conclude that the way we implemented congestion, by recording where agents were in the past and how fast the agents were going at that time (increasing congestion), is not a realistic estimation of agent congestion. Congestion should be calculated in the future as well as in the past, by this we mean that our heuristic should take into account where the agents are going and plot congestion accordingly, as it could be the case that a path does not have much congestion, but many agents are concurrently converging on this path to make it completely saturated at some later point.

Chapter 8

Conclusion

8.1 Limitations & Future Work

We have shown that using Agent Based Modelling is very much appropriate for simulating autonomous transportation agents. The next step is a formal justification that all the components that were implemented during the length of this project do indeed work as specified. With this in mind we propose several improvements to the current implementation as well as improved ways of collecting results for analysis.

8.1.1 Further Realism - Fuel Constraints

Although this project took into account a range of realistic parameters (such as speed being affected by weight carried), further parameters may be implemented to reach a higher level of fidelity with real world mobile transportation systems. In this project we simplified the simulation to ignore fuel constraints on the transport agents. The agents would of course need some sort of fuel (electric or otherwise) hence a viable way of recharging the agents during operation of the warehouse must be found to further enhance the realism of the simulation.

Because of the way this project was designed, fuel costs and recharging stations can be implemented by extending the current transport agent and node objects in the Java code. A protocol taking into account fuel capacity as well as fuel consumption parameters based on speed can be created, recharging nodes can themselves be placed on the warehouse layout and will act like other station nodes where the agent spends a certain amount of time on the node while it recharges its fuel capacity.

Not only would this constraint add to the realism of the simulation, but it would also allow the operators to find out the costs associated with agents recharging periodically, as well as finding the optimal location of these recharging stations to minimise congestion (which is already doable with the current implementation), other statistics could also be collected such as how much fuel is required, or how many times agents need to recharge when they are loaded with a certain capacity. All these parameters would be of interest to increase

efficiency of the simulation, hence find out a more accurate estimate of the total costs of installing an autonomous transportation system.

8.1.2 Agent Failure

Increasing the amount of realism by taking into account fuel constraints would also lead to the question of what an agent should do when its fuel runs out? When designing protocols which take fuel into account we would of course expect the agent to stop its current task and head for a recharging station before it completely runs out of fuel. However, due to congestion or otherwise it will be possible for the agent to fail to reach the recharging station before it runs out of fuel.

The agent will therefore be stopped in the middle of a track. There are two obvious ways of rescuing the failed agent. Firstly other nearby agents may be given the task of pushing the immobile agent to a recharging station - they would of course choose to help the agent or not depending on their current job, speed and location. Another approach would be to create a special agent that would be dispatched when a failure occurs, one or more of these special repairing agents could enter the simulation depending on the number of failures and refuel or repair the broken down agents.

8.1.3 Job System

Each of the protocols that were implemented start with random behaviour when they are first spawned into the world. As mentioned previously, attempts at creating a simple job requesting system were not met with satisfying results - however, a more complex system could be implemented and its performance measured against the current behaviour.

This more complex job system could include jobs which the agents give to each other, by this we mean things like commanding other agents to help them with a certain task. For example, agents could cooperate together if one agent has discovered that a node has many packages waiting to be transported and these packages are nearing the end of their TTL. Nearby agents responding to the call could then converge on this node in an intelligent manner (so as not to create so much congestion for outgoing agents) and attempt to transport the packages before their TTL expires.

8.1.4 Hill-climbing Algorithms for Parameter Search

As we found out from our results, the protocols of interaction varied a lot when the parameters of the simulation were altered. Although we performed a more "brute-force" approach (testing parameters incrementally) to choosing which parameters to use, given more time for the project, we would try to use hill-climbing algorithms [8] to obtain more accurate results for our choice of parameters. These algorithms do not randomly search through the parameter space, but instead they pin-point areas where there is a hill (usually indicating

optimal parameters) and "climb" this hill with small changes to the parameters, with the goal of obtaining an accurate result of the best parameter to use.

8.1.5 Extending GUI for easy map creation

We propose a final extension for this project, which would allow easier creation of maps. A simple GUI could be created where objects such as stations, nodes and roads/tracks could be inserted easily. The maps could then be read from an external file type such as XML and recreated for MASON to interpret and simulate.

8.2 Closing Remarks

Overall, we believe that this project has been successful in reaching the goals that we initially set out to do. The use of MASON was a very good decision early in the project and proved very flexible to allow us to implement a wide range of characteristics. We recommend that future projects involving autonomous agents try the MASON framework, given how easy to use it proved to create simple simulations and visualise them.

Bibliography

[1] Savant Automation. FAQ. *http://www.agvsystems.com/faqs/q1.htm*, 2010.

[2] Gottwald. Autonomous Agent Photo Archive. *http://www.gottwald.com/gottwald/site/gottwald/en/news/gallery/agv.html*.

[3] N. J.; Raphael B. Hart, P. E.; Nilsson. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics SSC4*, 1968.

[4] Andrew Ilachinski. *Artificial War - Multiagent-Based Simulation of Combat*. World Scientific, 2004.

[5] Iris F. A. Vis and Ismael Harika. Comparison of vehicle types at an automated container terminal. *OR Spectrum*, 26:117–143, 2004.

[6] KIVA Systems. *http://spectrum.ieee.org/robotics/robotics-software/three-engineers-hundreds-of-robots-one-warehouse/3*.

[7] Ling Qiu and Wen-Jing Hsu. Algorithms for Routing AGVs on a Mesh Topology. *Centre for Adv. Info. Sys., Schl. of Applied Science., Nanyang Tech. Univ., Singapore*, Oct 1999.

[8] Sean Luke. *Essentials of Metaheuristics*. 2009. available at http://cs.gmu.edu/∼sean/book/metaheuristics/.

[9] Rajeeva Lochana Moorthy. Cyclic deadlock prediction and avoidance for zone-controlled agv system. 2001.

[10] Keith Sullivan Liviu Panait Sean Luke, Gabriel Catalin Balan. MASON. *http://cs.gmu.edu/ eclab/projects/mason/*.

[11] Port Strategy. AGVs, what will it take? *http://www.portstrategy.com/features101/port-operations/planning-and-design/automation/what-will-it-take*, October 2005.

[12] Swisslog. Logistics Solution and Pharmacy Automation for Healthcare. *http://www.swisslog.com/index/hcs-index.htm*.

[13] Christopher M. Wright. Dance of the Bots. *APICS Magazine*, October 2007.

Appendix A

Listing

A.1 A* Algorithm

We now list the full code in algorithms A.1, A.2, A.3 and A.4 for the Java implementation used in this project. We mapped each AStarNode to a single Node object in our simulations. Although this increases complexity, it makes it easy for the designer to alter details in the A* algorithm without affecting the simulation objects at all.

Algorithm A.1 A* Node

```
package warehouse;

public class AStarNode
{
   private int ID;
   private AStarNode parent;

   public double fscore = 0;
   public double gscore = 0;
   public double hscore = 0;

   public AStarNode( int ID )
   {
      this.parent = null;
      this.ID = ID;
   }

   public AStarNode( int ID, AStarNode parent )
   {
      this.parent = parent;
      this.ID = ID;
   }

   public int getID()
   {
      return ID;
   }

   public AStarNode getParent()
   {
      return parent;
   }

   public void setParent( AStarNode parent )
   {
      this.parent = parent;
   }

   @Override public final boolean equals( Object obj )
  {
      if( this == obj ) return true;
      if( !( obj instanceof AStarNode ) ) return false;
      return ( ( AStarNode ) obj ).getID() == this.ID;
  }

   @Override public int hashCode()
   {
      return ID;
   }
}
```

Algorithm A.2 A* Declaration

```
package warehouse;

public class AStar
{
   // references for use in the methods
   Warehouse world;

   AStarNode start;
   AStarNode goal;

   public AStar( Warehouse world )
   {
      this.world = world;
   }

   public AStar( Warehouse world, AStarNode start, AStarNode goal )
   {
      this.world = world;
      this.start = start;
      this.goal = goal;
   }

   public void set( AStarNode start, AStarNode goal )
   {
      this.start = start;
      this.goal = goal;
   }

   // work backwards from goal and create path
   public LinkedList< AStarNode > reconstruct( AStarNode goal )
   {
      LinkedList< AStarNode > retList = new LinkedList< AStarNode >();

      AStarNode current = goal;

      while( current != null )
      {
         // add first so that we have an ordered list from start -> goal
         retList.addFirst( current );
         if( current.getParent() != null )
         {
            drawPath( current.getID(), current.getParent().getID() );
         }
         current = current.getParent();
      }

      return retList;
   }

   public LinkedList< AStarNode > neighbours( Warehouse world,
         AStarNode current )
   {
      Node node = Node.getNode( current.getID(), world );

      LinkedList< AStarNode > list = new LinkedList< AStarNode >();

      if( node.Nneighbour != null ) { list.add( new AStarNode(
            node.Nneighbour.getID(), current ) ); }
      if( node.Sneighbour != null ) { list.add( new AStarNode(
            node.Sneighbour.getID(), current ) ); }
      if( node.Eneighbour != null ) { list.add( new AStarNode(
            node.Eneighbour.getID(), current ) ); }
      if( node.Wneighbour != null ) { list.add( new AStarNode(
            node.Wneighbour.getID(), current ) ); }

      return list;
   }
}
```

Algorithm A.3 A* Search Implementation

```
// we'll use a LinkedList instead of ArrayList since we don't need positional
// access to the list (i.e. we only want the beginning/end)

// since we'll be adding a lot of elements to the list and iterating through
// it - LinkedList takes constant time for these operations whereas ArrayList
// would take linear time

public LinkedList< AStarNode > search()
{
   LinkedHashSet< AStarNode > openSet = new LinkedHashSet< AStarNode >();
   LinkedHashSet< AStarNode > closedSet = new LinkedHashSet< AStarNode >();

   // add start node to open-set
   openSet.add( start );

   start.gscore = 0;
   start.hscore = h( start, goal );
   start.fscore = start.hscore;

   // while there are still nodes to check
   while( openSet.size() > 0 )
   {
      AStarNode current = findBest( openSet );

      // stop if we have found the goal
      if( current.equals( goal ) ) { return reconstruct( current ); }

      // remove the next choice from open set, add it to closed set
      if( !openSet.remove( current ) ) { System.exit( -1 ); }
      closedSet.add( current );

      Iterator< AStarNode > it = neighbours( world, current ).iterator();

      boolean tentative_is_better = false;

      while( it.hasNext() )
      {
         AStarNode neighbour = it.next();
         if( closedSet.contains( neighbour ) )
         {
            continue;
         }

         double tentative_g_score = g( current ) + 1;

         if( !openSet.contains( neighbour ) )
         {
            openSet.add( neighbour );
            tentative_is_better = true;
         }
         else if( tentative_g_score < g( neighbour ) )
         {
            tentative_is_better = true;
         }
         else tentative_is_better = false;

         if( tentative_is_better )
         {
            neighbour.setParent( current );
            neighbour.gscore = tentative_g_score;
            neighbour.hscore = h( neighbour, goal );
            neighbour.fscore = neighbour.gscore + neighbour.hscore;
         }
      }
   }

   // failed to find goal
   return null;
}
```

Algorithm A.4 A* Methods

```
// convenience method
   public LinkedList< Node > getNodePath()
   {
      LinkedList< Node > path = new LinkedList< Node >();

      Iterator< AStarNode > it = search().iterator();

      while( it.hasNext() )
      {
         path.add( Node.getNode( it.next().getID(), world ) );
      }

      return path;
   }

   // convenience method
   public LinkedList< Node > search( Node current, Node goal )
   {
      set( new AStarNode( current.getID() ), new AStarNode( goal.getID() ) );

      return getNodePath();
   }

   public AStarNode findBest( LinkedHashSet< AStarNode > set )
   {
      Iterator< AStarNode > it = set.iterator();

      AStarNode min = it.next();
      AStarNode current = null;

      while( it.hasNext() )
      {
         current = it.next();
         if( current.fscore < min.fscore )
         {
            min = current;
         }
      }

      return min;
   }

   // distance from start node to current node along generated path, there is
   // always a distance of 1 unit between child and parent node
   public double g( AStarNode curr )
   {
      AStarNode current = curr;

      int count = 0;

      while( current != null )
      {
         count++;
         current = current.getParent();
      }

      // pre-decrement count as distance between node and itself is 0
      return --count;
   }

   // distance between two nodes
   public double h( AStarNode start, AStarNode goal )
   {
      return Node.getNodePosition( start.getID(), world ).distance(
            Node.getNodePosition( goal.getID(), world ) );
   }
```

Appendix B

Extra Material

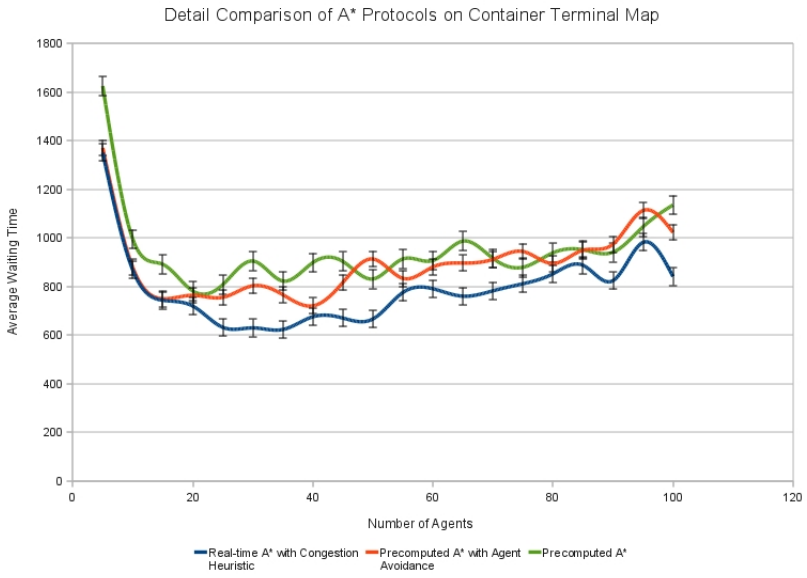
B.1 Square Map

The map was used with the first protocol that was implemented as a debugging environment (figure B.1).

B.2 Small Grid Map

This map was used for testing the A* algorithm since it is more complex than the square map. It was also first used to test the dispatcher agent (figure B.2).

B.3 Detailed Comparison of A* Protocols



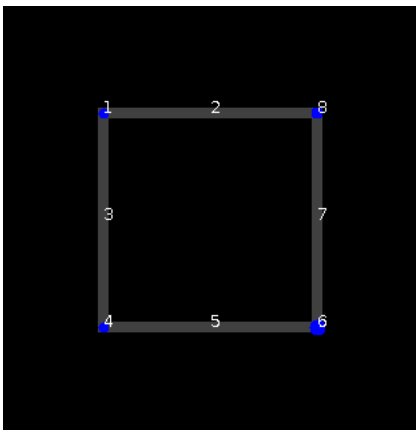


Figure B.1: Square

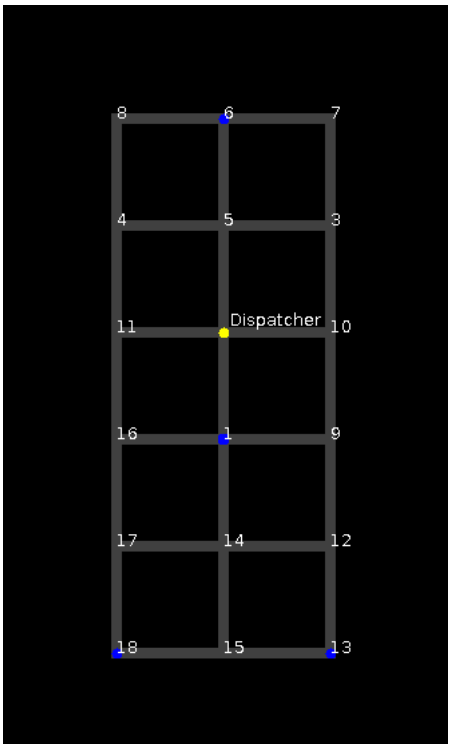
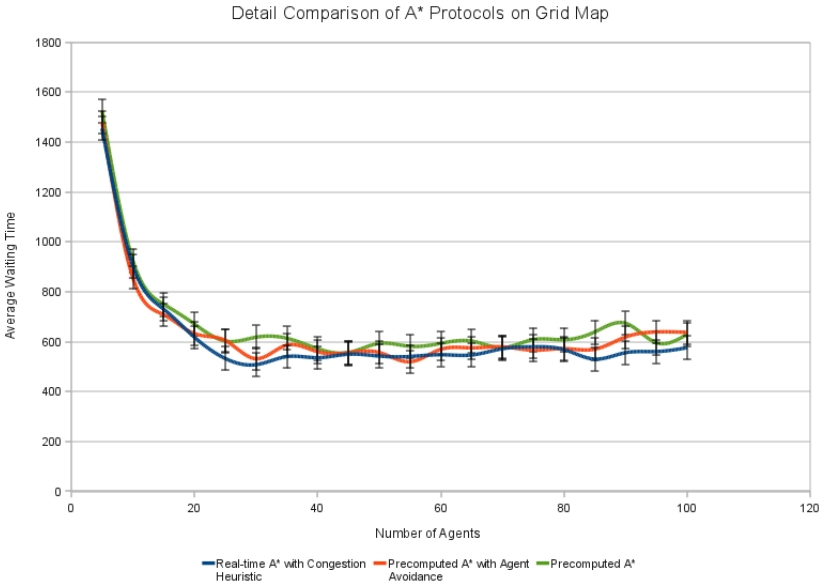
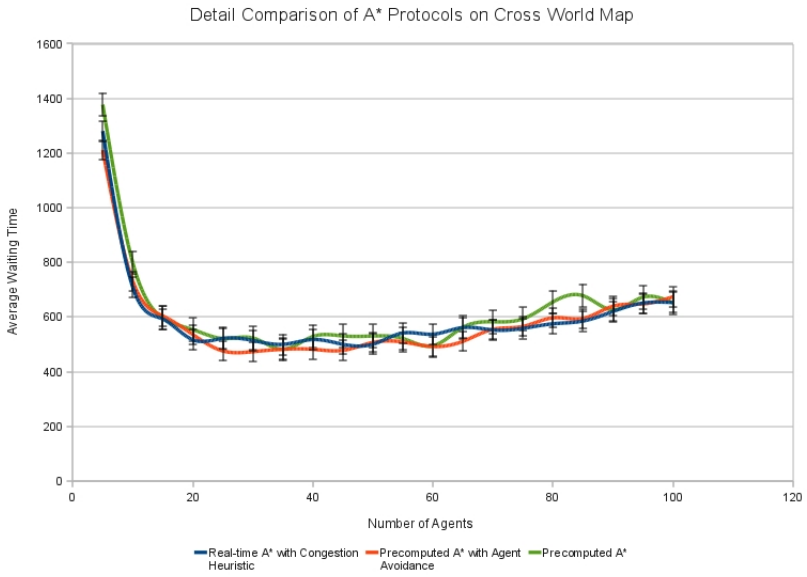


Figure B.2: Small Grid


Detail Comparison of A* Protocols on Grid Map
Average Waiting Time
Number of Agents
Real-time A* with Congestion Heuristic
Precomputed A* with Agent Avoidance
Precomputed A*


Detail Comparison of A* Protocols on Cross World Map
Average Waiting Time
Number of Agents
Real-time A* with Congestion Heuristic
Precomputed A* with Agent Avoidance
Precomputed A*