Multiple Forklift Routing and Scheduling

Prakhar Bhargava, Anthony Kremor, Luke Tornilla, Timothy Wong

The University of Melbourne

Abstract

The project was aimed at developing an algorithm that determines an optimal path for forklift operations in a warehouse/industrial environment. The path would be optimised for the fastest operation while still adhering to safety requirements. This routing algorithm would find an application in the development of autonomous forklifts that can function in conventional factories not designed for autonomous operations and development of intelligent human-assist features in standard forklifts.

1. Introduction

The goal was to develop an algorithm that would automate the scheduling of forklift jobs within a warehouse with an eye on reducing transit costs on the vehicles. We assume that forklifts can be either loaded - moving an item from one place in the warehouse to another as described by a job, or unloaded - moving through the warehouse without transporting anything. The ideal vehicle path is the one that completes all queued jobs taking the shortest overall route and has the least amount of unloaded time. Therefore, picking jobs that chain through to the next job is key to achieving the goal.

Intelligent routing through the warehouse also helps in working towards this goal as the warehouse is divided into different speed zones and hence have different traversal costs (time penalties). The ideal route is a fine balance between shortest and fastest route by maximizing the time spent in the low-cost zones. Additionally, the best path would minimize the number of turns made to reach the destination. This is favorable for two reasons: (1) forklift's high center of gravity especially when loaded puts it at a risk of tipping over when performing a turn; (2) turns might not be possible in one go and hence may comprise the time optimality in actual operation.

2. Strategy and high-level solution

The system takes input data in the form of a job list and a warehouse layout. Each task has a starting and ending location but no prescribed route. The storage layout shows all possible forklift positions within the store and describes the cost of traversing through each. In our implementation, we have divided the warehouse into Outdoor, Indoor, and Human zones in order of decreasing speed limits (increasing transit cost), but it could be any number of divisions.

The Warehouse map module accepts a layout and converts it into a format that is usable by the Pathfinding module. The module solves paths from each vehicle starting position to the commencement of each job, and from the end position of each task to the start of every other job. These cost of each path is used in the optimisation module to choose the best overall route for each vehicle.

Every path is also converted into a series of positions at each timestep based on the transit costs of each location. Together with the finalised vehicle routes, we can detect collisions between vehicles at any particular time and then resolve them.

The final output is a particular route for each vehicle in the warehouse.

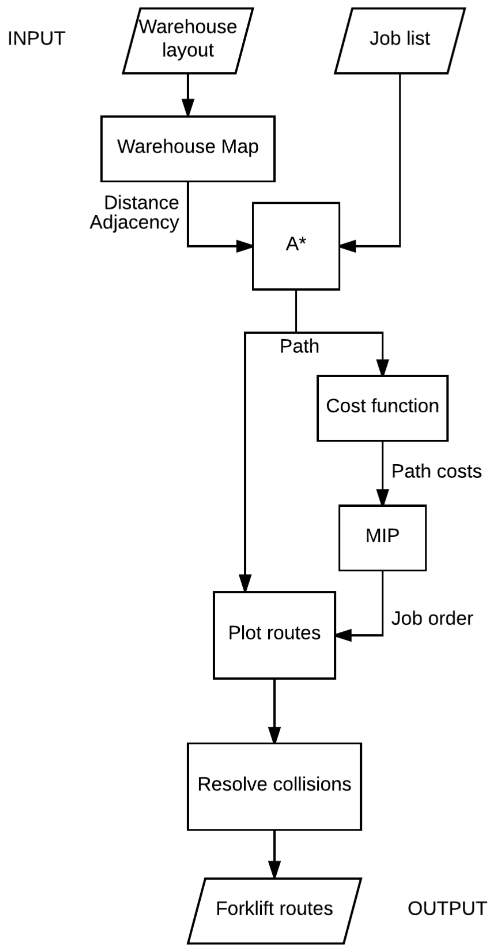
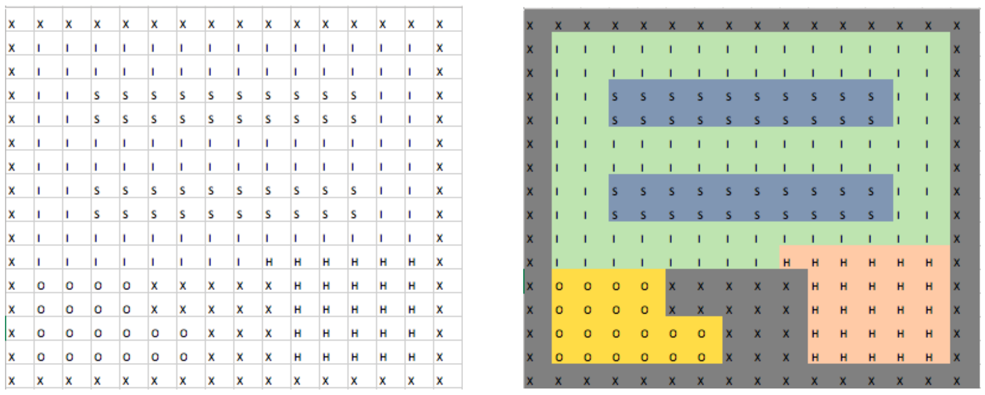


Figure 1: High level program structure and control flow

3. Solution

3.1. Warehouse

A grid based representation of a warehouse (an instance shown in Figure 2) was chosen as it provided an accurate representation of real world warehouses, which are highly planned and structured. The warehouse is made up of square grid cells of constant size, with the size of the warehouse written as $x \times y$. A warehouse is typically made up of the following zones:



X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
X	I	I	I	I	I	I	I	I	I	I	I	I	I	I	X
X	I	I	I	I	I	I	I	I	I	I	I	I	I	I	X
X	I	I	S	S	S	S	S	S	S	S	S	S	I	I	X
X	I	I	S	S	S	S	S	S	S	S	S	S	I	I	X
X	I	I	I	I	I	I	I	I	I	I	I	I	I	I	X
X	I	I	I	I	I	I	I	I	I	I	I	I	I	I	X
X	I	I	S	S	S	S	S	S	S	S	S	S	I	I	X
X	I	I	S	S	S	S	S	S	S	S	S	S	I	I	X
X	I	I	I	I	I	I	I	I	I	I	I	I	I	I	X
X	I	I	I	I	I	I	I	I	H	H	H	H	H	H	X
X	O	O	O	O	X	X	X	X	X	H	H	H	H	H	X
X	O	O	O	O	X	X	X	X	X	H	H	H	H	H	X
X	O	O	O	O	O	O	X	X	X	H	H	H	H	H	X
X	O	O	O	O	O	O	X	X	X	H	H	H	H	H	X
X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

Figure 2: CSV (left) and graphical (right) representation of warehouse map.

- Walls (X): impassable areas by a vehicle.
- Shelves (S): impassable by a vehicle, but can contain either zero or one pallets.
- Outdoor vehicle zone (O): refer to areas outside of the warehouse or areas without shelves, where field of view is good. The vehicle can travel at maximum possible speeds in this zone.
- Indoor vehicle zone (I): refers to areas around shelves, has lower field of view and hence lower speed are permitted in these zone compared to outdoor zones.
- Indoor shared zone (H): Similar to an indoor vehicle zone, however may contain humans so a vehicles maximum speed is much lower

Each zone was associated with a character to provide a human friendly representation. Each zone letter can then be entered into the warehouse graphical interface to denote the representation of a particular grid cell. The representation of the warehouse is a character based comma-separated values (CSV) file, that can be edited in common programs such as Microsoft Excel or a text editor. Microsoft Excel allows for the use of cell highlighting based on the contents of a cell. The cell highlighting feature was used to make the visual representation easier to read, by colouring cells of the same type with a consistent colour. An example of this can be seen in Figure 2(right).

The CSV representation of the warehouse, while visually useful, is not in a usable format for programmatic operations. Consequently, the program includes an internal conversion tool that converts the CSV warehouse representation into a data format suitable for consumption by the remainder of the program. The full code is included in Appendix A.

3.2. Vehicles

A warehouse would be a static object store if it weren't equipped with an appropriate number of item retrieval vehicles to store, retrieve, and move objects around a warehouse. A key design requirement was to allow for the simultaneous operation of multiple vehicles, ensuring their usage was maximised according to a cost function, and that all safety requirements were observed. An example of a safety requirement was ensuring that no two vehicles were within a particular radius of each other, as a collision could otherwise occur. A more complex condition was minimizing the number of turns a vehicle was required to make. Turning an unstable vehicle such as a forklift (especially while carrying a load) is inherently dangerous, as their physical dynamics makes them prone to tipping while cornering. Other formulation constraints included a vehicle only being able to carry a single item at a time, and that each vehicle had a maximum continuous usage time before it was required to cease operation.

3.3. Jobs

The act of meaningfully moving an item from one location to another is referred to as a job. Note that this definition excludes the movement of a vehicle without cargo. The term 'meaningful' is of importance here, as it allows us to create an efficiency metric. Computing the total distance traveled by a vehicle completing meaningful jobs versus non-meaningful travel allows for the efficiency of the program to be evaluated in a secondary manner to the primary objective function.

3.4. Path planning

Initially a completely MIP based solution was considered a viable solution to determine what path a vehicle should take, given some information about the layout of the warehouse. This approach didn't seem feasible because of enormous number of variables and constraint equations that would be generated and in turn would require a lot of memory and computational power to solve.

Instead, we decided to generate optimal paths via another algorithm, and to use a MIP to allocate jobs to vehicles. Subsequently, the path planning algorithm was found in A*. The A* algorithm is an extension of Dijkstra's algorithm and is best-first based, meaning that the likely best paths are evaluated first. The A* star algorithm, after conversion of $x \times y$ grid map to nodes, to find the optimal path between a starting node and end node the A* algorithm relies on three inputs: (1) an adjacency matrix; (2) a distance matrix; and (3) a heuristic.

The adjacency matrix is a $x \cdot y \times x \cdot y$ matrix that describes which the nodes that can be reached from a given node. This is a boolean matrix, as one node is either directly connected to another node or it is not.

A supplementary distance matrix is introduced to enable nodes to be given integer traversal costs based on permitted speed limits in each zone, instead of the unit cost given by the adjacency matrix. For instance, an edge connecting two indoor nodes will have a smaller traversal cost whereas an edge between two human only nodes will have a much higher cost. This information is encoded in the distance matrix, that has a similar structure as the adjacency matrix.

Finally, a heuristic function is defined. The A* heuristic is an out of band function that provides an estimate of the distance between a given node and target node. Typically, the Manhattan or Euclidean distance are used as it is computationally cheap to compute them. The A* algorithm uses the heuristic to inform which nodes it should explore first, and provides significant savings in solver time and memory usage compared to our monolithic Integer program route-finder in Section 4.

3.4.1. Improving Routing Computation Speed

The solve time of A* algorithm was hugely dependent on the choice of the heuristic function. While choosing a heuristic function that underestimates the distance between the given node and end node gives us the shortest path, underestimating it too much can lead the function to fail it's purpose. Overestimating the heuristic distance gives us sub-optimal path but improves the solve time considerably. The ideal heuristic has to comparable to the actual distance of the optimal path. We tuned the heuristic using Manhattan distance estimation as such that it sacrifices optimality (on shortest distance) to some extent for gains in solve time.

$$h(x_n, y_n, x_e, y_e) = |x_n - x_e| + |y_n - y_e|$$

where, (x_n, y_n) are coordinated for any given node 'n' and (x_e, y_e) are the coordinated for end node.

The Julia implementation of the same is shown in Appendix A

3.4.2. Minimizing Turns

Next, to minimize the number of turns that are planned in any route the Manhattan distance heuristic is given weights to give preference to traversal in either x or y direction, depending on in which direction the distance to be traveled is more. This is done by defining a flag variable as follows:

$$w = \begin{cases} 1, & \text{if} |x_s - x_e| > |y_s - y_e| \\ 0, & \text{otherwise} \end{cases}$$

where, (x_s, y_s) are coordinated for the start node and (x_e, y_e) are the coordinated for end node.

And then re-writing the heuristic function as:

$$h(x_n, y_n, x_e, y_e) = (0.1 * w + 1) * |x_n - x_e| + (0.1 * (1 - w) + 1) * |y_n - y_e|$$

where, (x_n, y_n) are coordinated for any given node 'n' and (x_e, y_e) are the coordinated for end node.

The results of using this heuristic function can be seen in Figure 3.

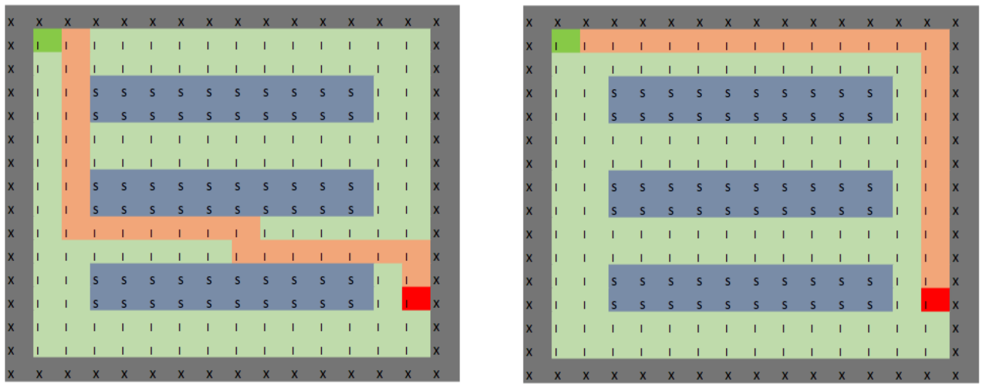


Figure 3: Calculated route using Manhattan distance(left) and weighted Manhattan distance(right).

This formulation of the heuristic function gives a quick way to minimize corners (if diagonal moves to neighbors aren't permitted) and still use the A* implementation for graph traversal from existing Julia packages.

3.5. Optimizer Formulation

The Job Scheduling Module takes the array of costs generated by the Pathfinding module and minimizes the overall travel cost across the fleet of vehicles. In this way it is similar to a Traveling Salesman Problem with multiple salesmen and where the edges between some locations - job start and job end points - are necessarily set to TRUE.

To do this we represent the model with two matrices of binary variables. A trip flag represents the fact that we take a particular trip from a start location to a job location, and an enable flag for each vehicle representing that the forklift is in use. A zero value trip flag means that the trip is not taken, and a zero value enable flag means that the vehicle is not in use and is kept out of the job pool.

```
@variable(m, flag[ui,si,vi], Bin)
# indicates a trip from u to s by forklift v
@variable(m, venable[vi], Bin)
# indicates that the forklift v is in use
```

We can see that trip flags is a 3-dimensional matrix of $ui \times si \times vi$. ui is the number of valid origins and si is the number of valid destinations, vi is the number of vehicles. The starting locations ui are the vehicle's home locations and the end point of each job to be done. The destination locations si are the vehicle's home locations and the starting point of each job. Therefore a variable set so

```
flag[12,15,3] = 1
```

would indicate that the trip starting at the end-point of job 12 and going to the start-point of job 15 would be undertaken by forklift number 3.

Aside from the model variables it also takes data in the form of the path cost matrix, a 2-dimensional matrix of $ui \times si$.

The model's objective function minimizes the total cost incurred by the fleet of vehicles for all travel across jobs.

Notice that the Scheduler does not deal with finding the optimal route - such pathfinding is left to the A* algorithm, meaning that the size of the model variables does not rely at all on the size of the warehouse, only the number of jobs and vehicles.

```
@objective(m, Min, sum(cost[i,j]*flag[i,j,k]
        for i in ui, j in si, k in vi))
```

In order to plot a complete and valid route, we need to enforce the following constraints:

```
@constraint(m, [j in si], sum(flag[i,j,k] for i in ui, k in vi) == 1)
# travel to a job only once

@constraint(m, [j in xi], sum(flag[i,j,k] for i in ui, k in vi) <= 1)
# travel to a starting position up to one time

@constraint(m, [k in vi, j in ui], sum(flag[i,j,k] for i in ui)
        == sum(flag[j,i,k] for i in ui))
# for all destinations the vehicles in must equal the vehicles out

@constraint(m, [k in vi], sum(flag[i,j,k] for j in ui, i=k)
        == venable[k])
# for all vehicle origins (U - S)
# all vehicles must respect their starting position

@constraint(m, [k in vi], sum(c[i,j]*flag[i,j,k] for i in ui, j in ui)
        <= venable[k]*vmax[k])
# each vehicle has a maximum distance/time to travel

@constraint(m, [i in ui], sum(flag[i,j,k] for k in vi, j=i) == 0)
# trips cannot start and end in the same node

@constraint(m, [i in vi, j in vi, k in vi], flag[i,j,k] == 0)
# vehicles cannot directly go to a pitstop
```

Finally, a callback to check for and eliminate subroutes is required. However, because of the complication arising from multiple vehicles being able to carry out the jobs the standard TSP subroute elimination constraint was not used. We opted instead to cut out one unique solution at a time in order to avoid overstepping and cutting potentially valid solutions. The full code for routeIsValid() is included in Appendix B.

```
function cutInvalidRoutes(cb)
    floatArr = floatArrayfromJuMPArray(getvalue(flag))
    valid = routeIsValid(getroute(floatArr))

    if valid == false
        println("Found invalid subtours, cutting invalid solution")
        @lazyconstraint(cb, sum(floatArr[i,j,k]*flag[i,j,k]
                for i in ui, j in ui, k in vi) <= u-1)
        # the new solution must differ from the old one
        # in at least one column
    end
end

addlazycallback(m, cutInvalidRoutes)
```

At the end of all this, the matrix of trip flags is converted into a readable job schedule for each vehicle and passed to the collision checking subsystem.

3.6. Collision detection

Collision checking is done by converting the job schedule created by the Scheduler into an ordered list of positions in the warehouse, correcting for the time cost (e.g. the "speed limit") of each node and then checking that each forklift occupies a unique position in the warehouse at each time. The full code of the Collision resolver module can be found in Appendix C. For example, the program flow might be:

```
MIP_output =  [[6,1], [4,2], [5,3]]
# indicates that vehicle 1 performs job 6 and then returns to start
# vehicle 2 performs job 4 and returns to start
# vehicle 3 performs job 5 and returns to start
```

The function animateroutes() converts a job schedule into a list of positions for each vehicle:

```
[
[ 1, 2, 3, 8,13,18,17,17,16,... 8, 9, 9,10,10,10,10, 9, 9, 8, 3, 2, 1],
[ 5, 4, 3, 8,13,18,19,19,20,... 8, 7, 7, 6, 6, 6, 6, 7, 7, 8, 3, 4, 5],
[ 3, 8,13,12,12,11,11,11,11,12,12,13,14,14,15,15,15,15,14,14,13, 8, 3]
]
```

We can see a collision between vehicles 1 and 2 at node 3 at time = 3.

The function resolvecollisions() takes the result of animateroutes() and performs the following operations:

1. At each time starting from the beginning, compare each vehicle's location.
2. If two vehicles share the same location, select the vehicle which arrived in that location later.

3. If both vehicles arrived at the location at the same time, select the vehicle with the shorter overall route.
4. Add a pause step to the selected vehicle - insert a position into its route to make it remain in its previous location for one additional unit of time
5. Go back one timestep to check for collisions created by the change
6. Repeat until the end of the routes

This formulation does not resolve head on collision. In fact a head on collision will result in two ve

4. Alternative MIP formulation

Our initial formulation was very monolithic in its approach, with the job selection being the same variant TSP as previously described and the routing problem being modeled as some type of commodity flow forward in time. Combining both pathfinding and job selection across multiple vehicles and across time would result in a massively unfeasible solution space - something like $M \times N \times t \times v$ where M and N are the dimensions of the warehouse, t is the length of time available to the vehicle, and v is the number of vehicles performing the jobs.

This approach may have made collision checking trivial - incorporated directly into the model constraints - but without an intelligent strategy to reduce the solution space it is computationally difficult.

5. Future Works

5.1. Collision detection

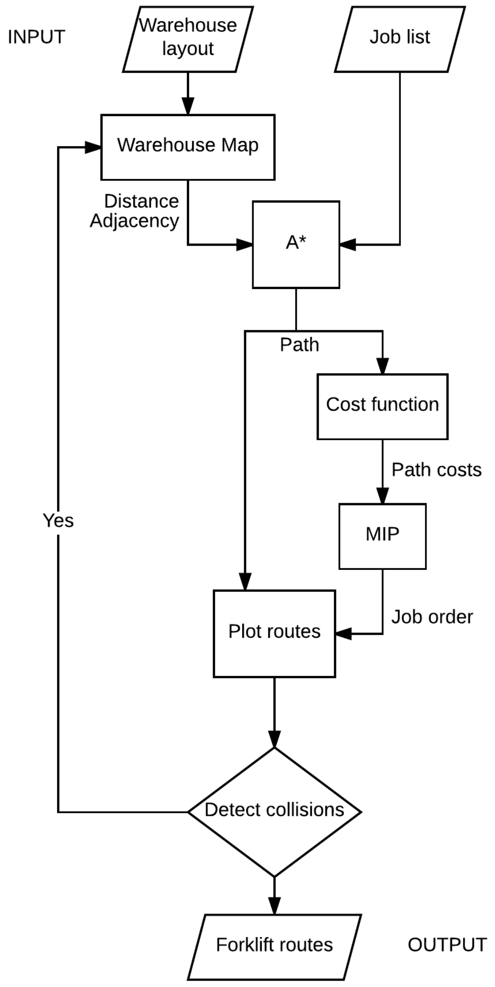


Figure 4: High level program structure and control flow

This alternative strategy for resolving collisions is potentially more efficient than the one we implemented while still being reasonably feasible. Rather than simply adding pause nodes to iron out vehicle collisions this system would detect the vehicles involved in each collision and their location. That information would be passed back to the warehouse map to alter the transit costs for the offending node and its adjacent nodes, effectively creating a "speed bump" or "congestion penalty" for using that route. The altered map would be sent to A* and new routes for each of the conflicting jobs would be found. The new route with the lowest cost would be passed to the Scheduler and a new job schedule would be found that takes into account the adjusted path.

In the group's view this would be a viable path to take to improve the job scheduling program if we had more time.

5.2. Calculation of route costs

Profiling in section 6 and specifically in subsection 6.1 shows that the computation time scales with the number of jobs and warehouse dimension.

The program's overall performance could be improved by decreasing the number of times the path finding module is run. One improvement would be to use the path finding heuristic described in section 3.4.1 to also generate the route costs array between all vehicles/jobs. The path finding heuristic takes in start and target nodes and produces an underestimate for the distance between two nodes. The computation of this heuristic is very fast, as it is a simple Manhattan/Euclidean distance. This would allow for the generation of route costs in a much shorter time, although the correctness of its contents is sacrificed. To move back to an optimal solution, the estimated route costs could be passed to the Scheduler, which would allow it to identify a set of likely routes. Then, the full path finding module could be run on this smaller subset of routes. The length of the actual routes would then be substituted into the route costs matrix, and the Scheduler run again.

As the heuristic function always provides an underestimate of the route cost, the Scheduler will either create a new subset of what it thinks are the optimal paths, or it will return the same subset. If the same subset is returned, then we have identified the optimal paths while using the expensive path finding module sparingly. If the subset returned is different, we iterate on the above solution. In the worst case, the path finding module may be required to evaluate all routes. As this is the current operating state, this proposed optimisation will either be slightly slower in speed as the current implementation, or faster, with no loss of optimality.

6. Analysis

Two different machines were used to generate the runtime analysis seen below. Consequently, there is some variability in the timing results reported, as the two computer systems had different specifications.

Compared to our alternative formulation, the decomposed solution is much more feasible to solve and ignores one of the largest scaling problems - warehouse size - in terms of the Scheduler module.

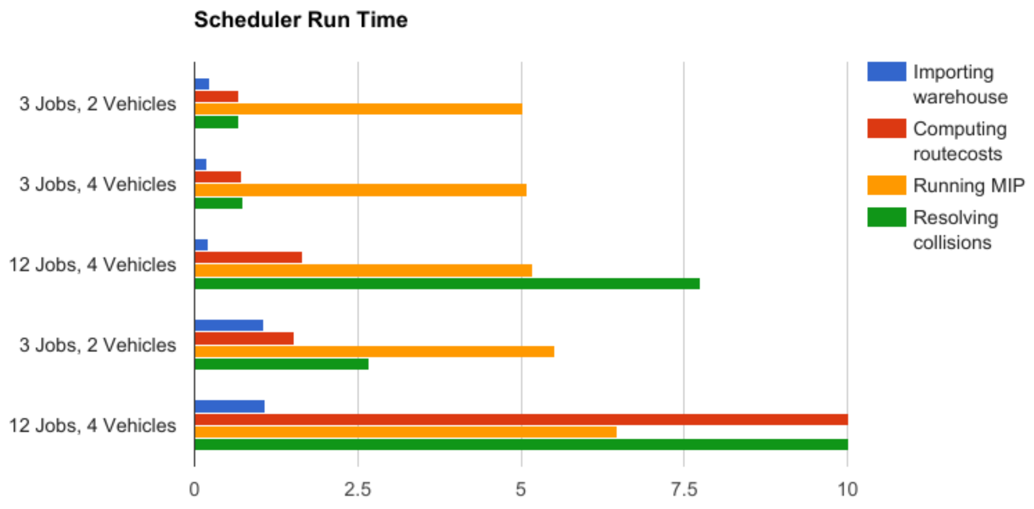


Figure 5: Running time for different modules

In Figure 5 we can see how the running time of each module varies with changes in the number of jobs, vehicles, and size of the warehouse. The first three schedules are for a warehouse of 16x16 nodes, the last two are for a warehouse of 120x120 nodes. In particular, we can see that the runtime of the Pathfinding module in red scales very harshly with the increase in jobs and warehouse sizes while the Scheduling module in yellow remains stable as across varying warehouse dimensions as shown in Figure 6.

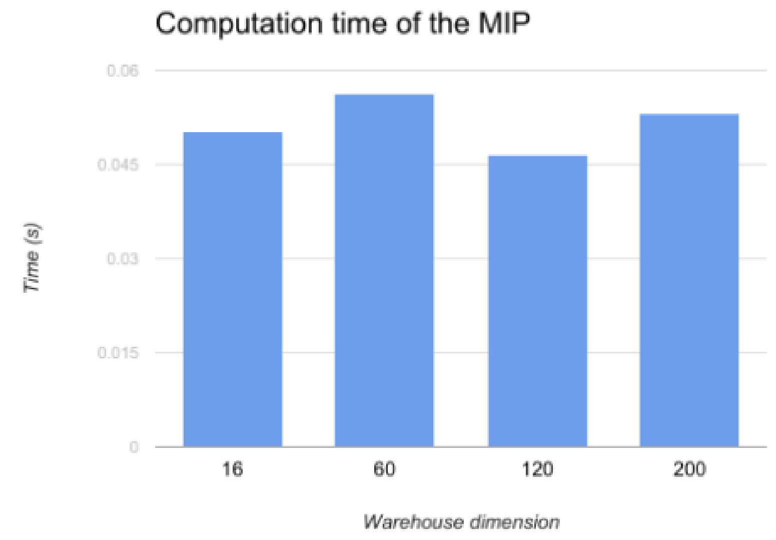


Figure 6: Running time of the MIP for various warehouse sizes

6.1. Scalability of the Pathfinding Module

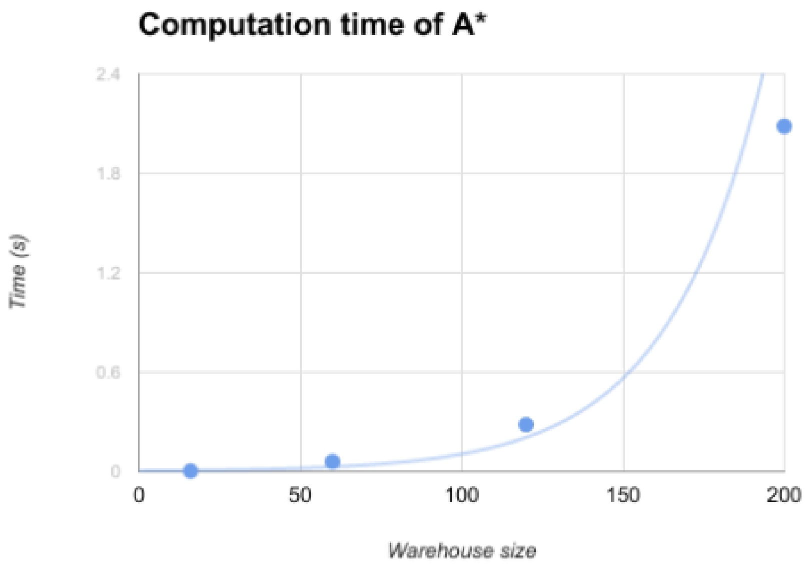


Figure 7: Running time of the path finding module for various warehouse sizes

The Pathfinding module, running on the A* algorithm, intelligently searches a map without exploring all the locations. It scales with warehouse dimensions but only indirectly. More accurately, it scales with the shortest path distance which is closely related to the size of the warehouse. Experimentally, we found that our implementation of A* scaled with the square of the warehouse area. The trendline shows $y = x^2$ next to our experimental data, with a fit of $r = 0.94$.

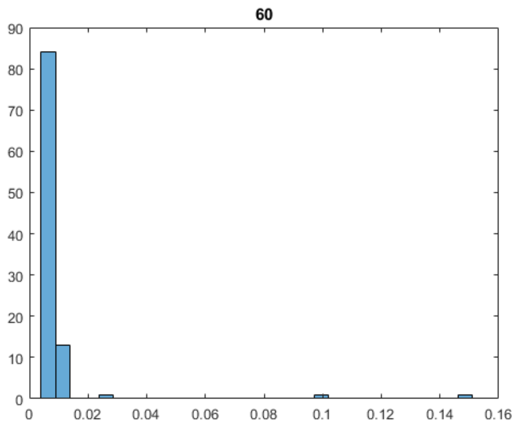


Figure 8: A* Solve time for a path in 60x60 warehouse

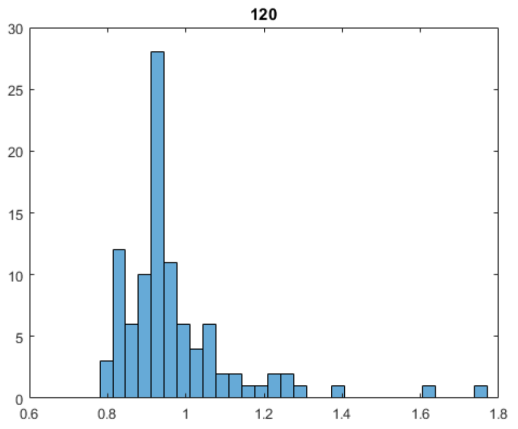


Figure 9: A* Solve time for a path in 120x120 warehouse

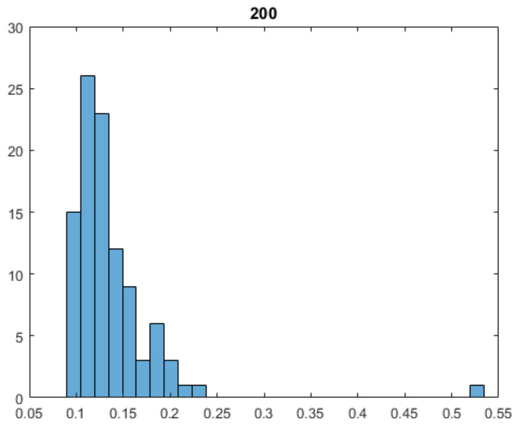


Figure 10: A* Solve time for a path in 200x200 warehouse

6.2. Scalability of the Schedule Optimizer

The Scheduler uses a matrix of flag variables of dimension $(v+j),(v+j),v$ where v is the number of vehicles and j is the number of jobs. Additionally, it has $v^3 + v^2 + (4 \times v) + (2 \times j) + (v \times j)$ constraints. In other words, the Scheduler scales polynomially with respect to the variables v and j.

6.3. Scalability of the Collision Resolver

The collision solver relies on a matrix of dimension $v \times t$ where v is the number of vehicles and t is the time required to complete the scheduled route. t is based on the warehouse nodes in the route and the traversal cost of each node. The runtime of the Collision module scales polynomially with $(v \times j) + (v \times t) + v + [(3/2)v \times t]$.

7. Conclusion

An effective method for warehouse job planning, path finding, and allocation is presented. This formulation uses a combination of integer programming techniques and path finding algorithms to converge on an efficient solution, as integer programming is not well suited to path finding. The A* path finding algorithm is used alongside a binary integer program to work out which jobs should be done by which vehicle, for a given time period. As this is a multi-vehicle warehouse, a method for avoiding vehicular collisions is also presented. Further optimisations and improvements are also detailed that could reduce the runtime of this program. Currently, the program is capable of solving large warehouses (200x200 grid), with a medium number of vehicles (10) and jobs (10).

The solution time is heavily dependent on the job requirements, with long or complex jobs taking longer to solve than shorter jobs. The inclusion of a heuristic function is detailed above that could create a program with near constant solution time. This would be ideal if the program was to be deployed in industry, as providing a strong estimate for solution time is critical to integrating an optimisation module into a workforce so that people and resources can be planned optimally.

Appendix A. Warehouse Map Generator and Pathfinding

```
using GraphPlot
using LightGraphs
using Colors
using Compose


function computeCosts(start_node, target_node, costs, x_size, y_size)

    target_x, target_y = nodeToXY(target_node, x_size, y_size)
    start_x, start_y = nodeToXY(start_node, x_size, y_size)

    # Choosing to give preference to x or y direction
    dx = abs(start_x - target_x)
    dy = abs(start_y - target_y)

    if(dx>dy)
        w = 1
    else
        w = 0
    end

    # Dependent on average traversal cost between nodes
    # Also tuning factor for optimality to solve time
    s_f = 50

    for i in 1:x_size, j in 1:y_size
        # Convert to node number
        node_number = xyToNode(i,j,x_size);

        dx = abs(i - target_x)
        dy = abs(j - target_y)

        costs[node_number] = round(s_f*
         (0.1*w+1)*dx + (0.1*(1-w)+1)*dy )

    end
end
```

```
function nodeToXY(node_number::Int64, x_size::Int64, y_size::Int64)
    x = Int64(floor(node_number / x_size) + 1);
    y = rem(node_number, y_size);
    return x,y ;
end

function xyToNode(x::Int64, y::Int64, x_size::Int64)
    return (x-1)*x_size + y;
end

function nodeValue(char)
    if char == "O"
        return 10;
    elseif char == "I"
        return 50;
    elseif char == "H"
        return 100;
    elseif char == "X"
        return 10000000;
    else
        return 10000000;
    end
end


function isValidMove(element)
    if((element=="I")||(element=="O")||(element=="H"))
        return true
    else
        return false
    end
end


function checkNeighbour(map, adj_mat, dist_mat, i, j)

    # Offsets relative to the cell of interest
    for x_offset in [-1, 0, 1], y_offset in [-1, 0, 1]
        # Check it isn't entering node [i,j] itself
```

```
                if (!(x_offset == 0 && y_offset == 0))
                    # Set the coordinate of the neighbour we are examining
                    x = i + x_offset;
                    y = j + y_offset;
                    if ((x > 0) &&
                        (x <= size(map)[1]) &&
                        (y > 0) &&
                        (y <= size(map)[2]))
                        if isValidMove(map[x,y])
                            # We increase the cost of a diagonal path by 1.5
                            # as it requires 'driving' a further distance

                            if (x_offset == 0 || y_offset == 0)
                                # If at least one offset is 0,
                                # then it is a straight move
                                            adj_mat[(i-1)*size(map)[1] + j,
                                                (x-1)*size(map)[1] + y] = true
                                dist_mat[(i-1)*size(map)[1] + j,
                                                 (x-1)*size(map)[1] + y]
                                                = nodeValue(map[x,y])

                            else
                                # Diagonal moves only possible in outdoor zone.
                                if(map[i,j]=="O")
                                    adj_mat[(i-1)*size(map)[1] + j,
                                                (x-1)*size(map)[1] + y] = true
                                    diag_mult = 1.5;
                                    dist_mat[(i-1)*size(map)[1] + j,
                                                 (x-1)*size(map)[1] + y]
                                             = diag_mult * nodeValue(map[x,y])
                                end
                            end
                        end
                    end
                end
            end
        end


        function readMapCSV(filename, MAP_SIZE_X, MAP_SIZE_Y)
```

```
    fp = open(filename, "r")
    map = Array(String,MAP_SIZE_X,MAP_SIZE_Y)


    for i in 1:MAP_SIZE_X
        line = readline(fp)
        # Ensure we don't read more Y values than we intended
        map[i,:] = [strip(string(s))
                                    for s in split(line,",")[1:MAP_SIZE_Y]]
        # println(map[i,:]);
    end

    close(fp)

    return map
end


function generateMapGraph(map)

    x, y = size(map);

    # Set up adjacency and distance matrices, prefill with 'empty'
    adj_mat = fill(false, x * y, x * y);
    dist_mat = fill(0, x * y, x * y);

    for i in 1:x
        for j in 1:y
            if((map[i,j]=="I")||(map[i,j]=="O")||(map[i,j]=="H"))
                checkNeighbour(map, adj_mat, dist_mat, i, j)
            end
        end
    end

    # Generating a Directed Graph
    g = DiGraph(adj_mat)
    println(g)
    return g, dist_mat
end
```

```
function path_init(filename, x_size, y_size)

    w_map = readMapCSV(filename, x_size, y_size)
    g, dist_mat = generateMapGraph(w_map)

    return g, dist_mat, w_map
end


function draw_path(g, x_size, y_size, path, draw_node_labels)
    # Build the 2d layout for the visualisation of nodes
    locs_x = Array(Float64, 1, nv(g))
    locs_y = Array(Float64, 1, nv(g))

    for i in 1:x_size, j in 1:y_size
        locs_y[(i-1) * x_size + j] = i;
        locs_x[(i-1) * x_size + j] = j;
    end

    locs_x = vec(locs_x);
    locs_y = vec(locs_y);

    nodefillc = fill(colorant"lightseagreen", nv(g))
    edgestrokec = fill(colorant"white", ne(g))
    nodesize = fill(1.0, nv(g))

    # Colour the nodes we travelled through
    for node in path
        nodefillc[node] = colorant"orange";
        # Color the from node orange
        nodesize[node] = 100000.0
    end

    if draw_node_labels
        draw(PDF("nodes.pdf", 160cm, 160cm),
                 gplot(g, locs_x, locs_y, nodelabel=1:nv(g),
             nodefillc=nodefillc,
             edgestrokec=edgestrokec))
    else
```

```
            draw(PDF("nodes.pdf", 160cm, 160cm),
                     gplot(g, locs_x, locs_y, nodefillc=nodefillc,
                       edgestrokec=edgestrokec, nodesize=nodesize))
        end

    end

    # TODO: Not sure how this works.
    function animate_path(g, x, y, path)
        # Build the 2d layout for the visualisation of nodes
        locs_x = Array(Float64, 1, x*y )#+ 59); # TODO Why + 59?
        locs_y = Array(Float64, 1, x*y )#+ 59);

        for i in 1:x, j in 1:y
            locs_y[(i-1) * x + j] = i;
            locs_x[(i-1) * x + j] = j;
        end

        locs_x = vec(locs_x);
        locs_y = vec(locs_y);

        nodefillc = fill(colorant"lightseagreen", x * y + 59);
        # TODO WHY 59

        # Colour the nodes we travelled through
        for (index, node) in enumerate(path)
            nodefillc[node[1]] = colorant"orange";
            # Color the from node orange
            nodefillc[node[2]] = colorant"orange";
            # Color the to node orange
            # Not the most efficient way of doing this
            # but it increases readability for a tiny performance penalty.
            # It might even be optimised out...

        end
        draw(PDF("nodes.pdf", 160cm, 160cm),
             gplot(g, locs_x, locs_y, nodelabel=1:x * y, nodefillc=nodefillc))
    end
```

```
function calc_path_cost(path, dist_mat)

    path_cost = 0;

    # TODO So this isn't great...
    try
        for move in path
            path_cost += dist_mat[move[1], move[2]];
        end
    catch
        path_cost = 0;
    end

    return path_cost;
end


function path_main(start_node, end_node, g, dist_mat, x_size, y_size)

    # The first call to both @time and a_star
    # will give inaccurate results, so we run it again
    #println("Compiling @time and a_star functions. Ignore two results below.
    #@time path = a_star(g, start_node, end_node, dist_mat);

    costs = Array(Int64, 1, nv(g))

    heuristic = generateHeuristic(start_node,
    end_node,
                                  costs,
                                  x_size,
                                  y_size)
    println("Planning path from ", start_node, " to ", end_node);
    @time path = a_star(g, start_node, end_node, dist_mat, heuristic);

    return path

end


function generateHeuristic(start_node, end_node, costs, x_size, y_size)
```

```
    computeCosts(start_node, end_node, costs , x_size, y_size);

    function heuristic(v)
        return costs[v];
    end

    return heuristic
end


function main(s,t,filename = "WarehouseMap.csv",size=60)

    # Functions run once
    g, dist_mat, map = path_init(filename,size,size)
    costs = Array(Int64, 1, nv(g))

    # Functions for each job
    heuristic = generateHeuristic(s, t, costs, size, size)
    @time path = a_star(g, s, t, dist_mat, heuristic);
    # TODO: Probably the dist_max being passed here would have to be scaled.
    println(calc_path_cost(path, dist_mat))

    draw_path(g, size, size, path)

end
```

Appendix B. Scheduler Optimization program

```
using JuMP
using Gurobi

function planroutes(costmatrix::Array,jobs::Int,vehicles::Int)

    u = vehicles+jobs
    s = jobs
    v = vehicles
    # u = vehicle origins {x,y,...A,B,C,..}
    # s = vehicle target positions {A’,B’,C’}
    vmax = [40000000, 40000000, 40000000, 40000000, 40000000]
    # vehicle maximum distance/time
    # set for each vehicle

    ui = 1:u
    si = vehicles+1:u
    vi = 1:v
    xi = 1:v

    # some functions to parse the output of the MIP

    function floatArrayfromJuMPArray(A::JuMP.JuMPArray)
    # converts JuMP variables to julia primitives
        B = zeros(JuMP.size(A))

        for k in 1:JuMP.size(A,3)
            for j in 1:JuMP.size(A,2)
                for i in 1:JuMP.size(A,1)
                    B[i,j,k] = A[i,j,k]
                end
            end
        end

        return B
    end

    function getroute(F::Array)
    # expresses the route of the forklift as a list of jobs
        vex = size(F,3)
        route = Array{Any}(vex)
        for k in 1:vex
            println("vehicle ", k)
            row = k
            route[k] = Array{Int}(0)
            #println("start row = ", row)
            while size(find(F[row,:,k]),1) >= 1
                row = find(F[row,:,k])[1]
                #println("next row = ", row)
                append!(route[k],row)
                if row==k
                    break
                end
            end
        end

        return route
    end

    function routeIsValid(R::Array)
        # check that the combined route covers all jobs
        # target value is at least jobs + 1 vehicle
        # this can break down if
        # vehicles>jobs but that seems pretty unlikely
        # maybe change this out at some point / write a more robust one
        println(R)
        return sum(size(R[i])[1] for i in eachindex(R)) >= u
    end

    m = Model(solver = GurobiSolver(Gurobi.Env()))

    @variable(m, flag[ui,ui,vi], Bin)
    # indicates a trip from i to j
    @variable(m, venable[vi], Bin)
    # indicates that the forklift is in use

    @objective(m, Min, sum(costmatrix[i,j]*flag[i,j,k]
        for i in ui, j in si, k in vi))

    function cutInvalidRoutes(cb)
        floatArr = floatArrayfromJuMPArray(getvalue(flag))
        valid = routeIsValid(getroute(floatArr))

        if valid == false
            println("Found invalid subtours, cutting invalid solution")

            @lazyconstraint(cb, sum(floatArr[i,j,k]*flag[i,j,k]
                for i in ui, j in ui, k in vi) <= u-1)
            # the new solution must differ from the old one
            # in at least one column
        end
    end
```

```
addlazycallback(m, cutInvalidRoutes)

@constraint(m, [j in si], sum(flag[i,j,k] for i in ui, k in vi) == 1)
# travel to a job only once

@constraint(m, [j in xi], sum(flag[i,j,k] for i in ui, k in vi) <= 1)
# travel to a starting position up to one time

@constraint(m, [k in vi, j in ui],
    sum(flag[i,j,k] for i in ui) == sum(flag[j,i,k] for i in ui))
# for all destinations the vehicles in must equal the vehicles out

@constraint(m, [k in vi],
    sum(flag[i,j,k] for j in ui, i=k) == venable[k])
# all vehicles must respect their starting position

@constraint(m, [i in ui], sum(flag[i,j,k] for k in vi, j=i) == 0)
# trips cannot start and end in the same node

@constraint(m, [i in vi, j in vi, k in vi], flag[i,j,k] == 0)
# vehicles cannot directly go to a pitstop

@constraint(m, [k in vi], sum(costmatrix[i,j]*flag[i,j,k]
    for i in ui, j in ui) <= venable[k]*vmax[k])
# each vehicle has a maximum distance/time to travel


@time begin
solve(m)
end

for k in vi
    println("vehicle ", k)
    for i in ui
        print(getvalue(flag[i,:,k]))
    end
end
println(getobjectivevalue(m))

for k in vi
```

```
        println("vehicle ", k,
                        " distance: ",
                        sum(costmatrix[i,j]*getvalue(flag[i,j,k])
                      for i in ui, j in si))
    end

    return getroute(floatArrayfromJuMPArray(getvalue(flag)))

end
```

Appendix C. Collision Resolver Module

```
function plotroutes(routeplan::Array,routes::Array)

    vehicles = size(routeplan,1)
    println("plotting routes for ",vehicles," vehicles")
    actualroute = Array{Any}(vehicles)

    for v in eachindex(routeplan)
        println("vehicle ",v)
        startrow = v
        nextrow = v
        jobs = routeplan[v]
        vehicleroute = []

        for i in eachindex(jobs)
            startrow = nextrow
            nextrow = jobs[i]
            println("nextrow ", nextrow)

            println(vehicleroute);
            println(routes);

            append!(vehicleroute,routes[startrow,nextrow])
            println("updated route ",vehicleroute)

            if nextrow > vehicles
                # i.e. it is a job, not a start position
                #println("nextrow ", nextrow)
                append!(vehicleroute,routes[nextrow,nextrow])
                #println(vehicleroute)
            end
        end

        actualroute[v] = vehicleroute
    end

    #println("vehicle1 route", actualroute[1])
    #println("vehicle2 route", actualroute[2])
    #println(actualroute)
    return actualroute
end

function animateroutes(AR::Array,nodecost::Array)
# takes an array of proposed routes
# and an array relating node number to node cost
# returns an animated array showing vehicle locations per timestep
    animatedroutes = Array{Any}(size(AR,1))

    for vehicle in eachindex(AR)
        #println("vehicle",vehicle)
        animatedroute = []
        for step in eachindex(AR[vehicle])
            node = AR[vehicle][step]
            squarecost = nodecost[node]
            for x in 1:(squarecost)
                append!(animatedroute,node)
            end
        end
        animatedroutes[vehicle] = animatedroute
    end
    return animatedroutes
end


#given job routes with conflicting timespaces
#find vehicle with greater slack
#-> add a "pause" node to that vehicle's route before the collision
#-> assumes that all vehicles have unique starting positions
#resolve collisions
```

```
function resolvecollisions(AR::Array)
    vehiclecount = size(AR,1)

    routelength = Array{Int}(vehiclecount)
    # find the length of each vehicles route
    for i in eachindex(AR)
    #println(i)
    routelength[i] = size(AR[i],1)
    end

    endtime = maximum(routelength)

    timeslider = 1

    while timeslider<=endtime

        #check if the cell exists, then compare to other rows if they exist
        # if a collision is detected,
        # find the vehicle with the shorter path
        # add a pause node to that path BEFORE the collision location
        for i in 1:vehiclecount
          if timeslider<=routelength[i]
            p1 = AR[i][timeslider]
            for j in i+1:vehiclecount
              #println("vehicle",i," vs vehicle",j," at time",timeslider,"/",
              if timeslider<=routelength[j]
                p2 = AR[j][timeslider]
                if p1==p2
                  # collision detected at node p1=p2
                  # the pause node must be added to the vehicle that arrives
                  # should only ever have to look back 1 timestep
                  if routelength[i]<=routelength[j]
                    # add pause in vehicle i
                    # default q value
                    q = i
                  else
                    # add pause in vehicle j
                    q = j
                  end
                  # then check if q value is consistent with arrival times
                  # if unequal arrival time then use that q instead
                  if AR[i][timeslider-1]==p1 && AR[j][timeslider-1]!=p1
                    q = j
                  elseif AR[i][timeslider-1]!=p1 && AR[j][timeslider-1]==p1
                    q = i
                  end

                  println("adding pause to vehicle",q," at time",timeslider,"
                  insert!(AR[q],timeslider,AR[q][timeslider-1])
                  routelength[q] = size(AR[q],1)
                  endtime = maximum(routelength)
                end
              end
            end
          end
        end
        timeslider = timeslider+1
    end

    return AR
end
```

Appendix D. Sample Main Function

```
include("./lazywarehouse.jl")
include("./collisionchecker.jl")
include("./WarehouseGrapher.jl")

# define global constants for organizing the route array
jobcount = 12
vehiclecount = 4

u = jobcount + vehiclecount

# Initialise A* for a given map
x = 120;
y = 120;
x_size = x;
y_size = y;
@time begin
```

```
g, dist_mat, w_map = path_init("120x120WarehouseMap.csv", x, y);
println("==================== PATH INIT =======================")
end

@time begin
jobs = [Dict("job_id" => 1,
             "start_node" => 1084,
             "end_node" => 14147)
        ,Dict("job_id" => 2,
             "start_node" => 6731,
             "end_node" => 3202),
        Dict("job_id" => 3,
             "start_node" => 9488,
             "end_node" => 4975),
         Dict("start_node"=>13726,
            "end_node"=>1598,
            "job_id"=>4),
        Dict("start_node"=>3144,
            "end_node"=>5527,
            "job_id"=>5),
        Dict("start_node"=>6368,
            "end_node"=>7847,
            "job_id"=>6),
        Dict("start_node"=>4335,
            "end_node"=>3881,
            "job_id"=>7),
        Dict("start_node"=>8724,
            "end_node"=>2233,
            "job_id"=>8),
        Dict("start_node"=>1949,
            "end_node"=>12996,
            "job_id"=>9),
        Dict("start_node"=>14100,
            "end_node"=>175,
            "job_id"=>10),
        Dict("start_node"=>3450,
            "end_node"=>6248,
            "job_id"=>11),
        Dict("start_node"=>6385,
            "end_node"=>13570,
            "job_id"=>12)
             ];

vehicles = [Dict("vehicle_id" => 1,
             "start_node" => 5093),
        Dict("vehicle_id" => 2,
             "start_node" => 168),
        Dict("start_node"=>11145,
            "vehicle_id"=>3),
        Dict("start_node"=>6818,
            "vehicle_id"=>4)];

for job in jobs
    i, j = nodeToXY(job["start_node"], x_size, y_size);
    println("Node: ", job["start_node"], " is ", w_map[i, j]);
    i, j = nodeToXY(job["end_node"], x_size, y_size);
    println("Node: ", job["end_node"], " is ", w_map[i, j]);
end

for v in vehicles
    i, j = nodeToXY(v["start_node"], x_size, y_size);
    println("Node: ", v["start_node"], " is ", w_map[i, j]);
end


routecosts = Array(Int,u,u)
# generated by pathcost module

routedefinitions = Array(Any,u,u)
# generated by A*

nodecosts = Array(Int,x*y)
# generated from map
# 1 dim array nodecosts[1] = cost to traverse node1

for (v_index, vehicle) in enumerate(vehicles)

    # Calculate vehicle to vehicle costs
    current_index = 1;
    for (v_other_index, vehicle_other) in enumerate(vehicles)
```

```
            route = path_main(vehicle["start_node"],
                                             vehicle_other["start_node"],
                              g,
                              dist_mat,
                              x_size,
                              y_size);
            cost_vehicle_start_to_job_start =
                            calc_path_cost(route, dist_mat);
            routecosts[v_index, current_index] =
                            cost_vehicle_start_to_job_start;
            routedefinitions[v_index, current_index] = route;
            current_index += 1;
        end

        current_index -= 1; # TODO Comment

        for (j_index, job) in enumerate(jobs)
            route = path_main(vehicle["start_node"],
                                             job["start_node"],
                              g,
                              dist_mat,
                              x_size,
                              y_size);

            cost_vehicle_start_to_job_start =
                            calc_path_cost(route, dist_mat);
            routecosts[v_index, current_index + j_index] =
                            cost_vehicle_start_to_job_start;
            routedefinitions[v_index, current_index + j_index] =
                            route;
        end
    end

    for (j_index, job) in enumerate(jobs)
        # Calculate job to vehicle costs
        current_index = 1;
        for (v_index, vehicle) in enumerate(vehicles)
            route = path_main(job["end_node"],
                                             vehicle["start_node"],
                              g,
                          dist_mat,
                          x_size,
                          y_size);
            cost_vehicle_start_to_job_start =
                        calc_path_cost(route, dist_mat);
            routecosts[j_index + size(vehicles)[1], current_index] =
                        cost_vehicle_start_to_job_start;
            routedefinitions[j_index + size(vehicles)[1],
                                         current_index] =
                                route;
            current_index += 1;
        end

        current_index -= 1; # TODO Comment

        for (j_other_index, job_other) in enumerate(jobs)
            route = path_main(job["end_node"],
                                          job_other["start_node"],
                          g,
                          dist_mat,
                          x_size,
                          y_size);

            cost_vehicle_start_to_job_start =
                        calc_path_cost(route, dist_mat);
            routedefinitions[j_index + size(vehicles)[1],
                                         current_index + j_other_index] =
                                route;
            routecosts[j_index + size(vehicles)[1],
                           current_index + j_other_index] =
                                cost_vehicle_start_to_job_start;
        end
    end

    # Flatten route definitions
    for i in 1:size(routedefinitions)[1], j in 1:size(routedefinitions)[2]
        flattened = [];

        # TODO Not ideal but gets the job done.
        # This thing gets upset when it receives an empty (or undefined) route.
        try
            for (index, edge) in enumerate(routedefinitions[i,j])
                if index == 1
                    append!(flattened, edge[1]);
                end
                append!(flattened, edge[2]);
            end
            routedefinitions[i, j] = flattened;
        catch
            routedefinitions[i, j] = [];
        end
    end
    println("==================== COMPUTE ARRAYS =======================")
    end

    @time begin
        routeplan = planroutes(routecosts, jobcount, vehiclecount);

        println("==================== MIP =======================")
    end
    #=

    @time begin
    routeplots = plotroutes(routeplan, routedefinitions);

    for i in 1:x, j in 1:y
        index = (i - 1) * x + j;
        nodecosts[index] = nodeValue(w_map[i, j]);
    end

    animatedroutes = animateroutes(routeplots, nodecosts);
    resolvedroutes = resolvecollisions(animatedroutes);
    println("==================== COLLISIONS =======================")
    end

    #draw_path(g , x_size, y_size, resolvedroutes[2], false);

    #println("Resolved routes");
    #println(resolvedroutes)
```