Planning a delivery route

## Show the data structure that holds the plan of the floor.

The floor plan consists of a map which contains all locations as the keys and then the possible moves from that location as the value, held in a vector. This results in a map that looks like

{ “r101” [“o101”]

“o101” [“r101” “o103” “ts” “a3”]}

## How is this data structure built?

This structure is built manually from a map. To extend this data structure would be as simple as a user just appending another record to the end of the map.

## How are starting and finishing locations identified?

There are two ways to run the program. The first way requires giving the parcels to collect, parcels to deliver and a start location.

(run {parcelsToCollect} {parcelsToDeliver} {start})

If the start location is “office” as in most cases, then the start can be omitted and just provide the parcels to collect and parcels to deliver.

(run {parcelsToCollect} {parcelsToDeliver})

In all cases, it is assumed that the robot will be asked to return to the mail room when finished.

## Do you validate start and end points before planning a route?

The user can change the start and the rooms that packages needs to be collected from and delivered to. These are validated as existing as a key in the map.

## Show the data structure that holds the robot’s route.

The route is held using a vector of sequences. Each sequence represents one leg of the journey. To perform task 8 (office -> r111 -> r121 -> r131 -> office), 4 legs would be needed and would look like.

[("office" "mail" "ts" "o101" "o103" "o105" "o107" "o109" "o111" "r111")

("r111" "o111" "o109" "o105" "o103" "b3" "b1" "c2" "c1" "o123" "o121" "r121")

("r121" "o121" "o123" "o125" "o127" "o129" "o131" "r131")

("r131" "o131" "mail" "office")]

## How is this data structure built?

The data structure is built while the robot proceeds. Every time a robot has completed one leg, it appends the route it took to a vector containing the current path. See next section for more information

## Discuss the approach that the robot uses to planning a route.

To plan a route, the robot uses a breadth first search. To do this, from the starting point a thread was

spawned for each possible move. This was repeated for each one of those threads, effectively spreading out one level at a time from the starting point. There were two possible outcomes from each route, either it would find the target and be added to a list of possible routes or it would get ‘stuck’ with no possible moves and then be classed as not a route and dismissed. The weights for these routes were then calculated and the shortest selected.

This method was chosen over doing a depth first search as there is no backtracking and the use of threading, both providing performance gains.

These steps were repeated for each stage, with every permutation of targets been tested to try and find the overall shortest route. This is very similar to how Dijkstra’s algorithm works. With all stages and permutations done. The shortest route between start and end, visiting all the

targets were given.

## How does the robot handle multi-stage routes?

Once the routes have been built, as described above, each stage would still be held separate. Once the first stage has happened (eg. If the pass is office -> r101 -> b2 -> office then the first stage is office ->r101), the calculations as described in section 7 happen again, using the target (r101) as the starting point. This means that if any targets are added here then they will also be calculated into the route.

## Show the code that is used to dynamically add stages to routes. Can your program also remove stages from routes?

Dynamically adding stages to routes can only be done when a package is collected. When collecting a package, the destination of that package is added to the targets and the collection point is removed. It does not show much sense to have just this single piece of code here but can be seen where the run function utilises the collectPackages function in the whole code.

## Show the code that you use to display a route. How do you manage output as a side-effect?

The path is displayed in the summary map. This also contains the final destination and the distance it took.

(prn {

:finalDestination (pretty/blue s)

:path r

:distance (pretty/magenta l) })

The path was output using a print statement, with the other useful parts been coloured.