Nimra Anjum, Chris Westerhoff, Jason Zhang
ECE 479/579
Dr. Jerzy Rozenblit
5 May 2023

Final Project

Youtube Link: https://youtu.be/x0AJUa9dErk

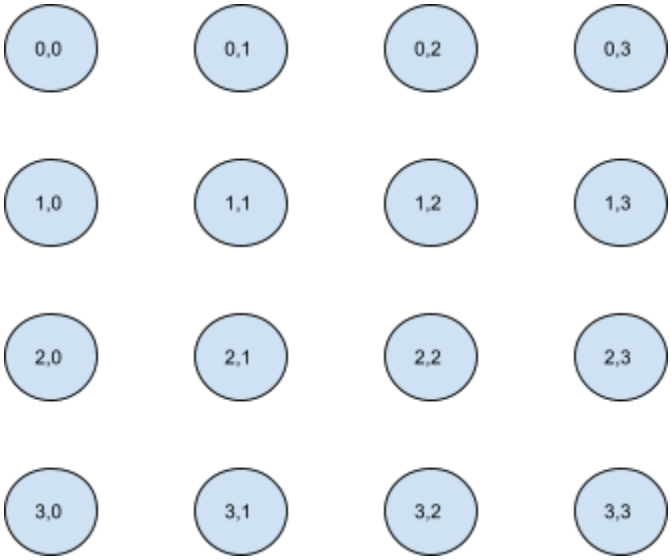
Part A:

We are assuming that delivery time and energy consumption are directly related to the distance traveled by the robot. We are defining our simulation parameters to be 3 robots that can each carry a maximum of 2 orders. Assuming there are multiple robots and there are multiple goal states on our map, we want to determine whether it is efficient for the robot to carry two orders. Our algorithm will work in a way that whenever we will receive a new order we will check for the robots available to deliver that order based on certain constraints. In addition, we need to consider certain constraints while we are checking for available robots including (time constraint, fuel/energy constraint). The fuel/energy constraint mainly looks into whether the robot is able to take the order. The time constraint mainly looks into which order to deliver first.

As a visual layout for the grid of our program, it can be shown below. Here we are assuming that the individual nodes have a distance of 1m apart vertically and horizontally. In addition to this, the robot travels 1m/s. As a result, this means that when a robot travels to one node, it will take 1 second. For the energy, we assumed that it is directly affected by the distance traveled. We have set the total energy charge for a robot to be 100%. When the robot travels to one node, the robot will use up 1% of its energy.

The heuristic values of each node are calculated in a way of how far away they are from the goal state. We are assuming that the robot can only travel horizontally and vertically. An example could be having the goal state at (2,2). Node (0,1) would have a heuristic value of 3.

In the case of an obstacle, the heuristic value will be a very large value, so the search will automatically avoid the obstacles. Prior to the robot highlighting a path to travel, the robot will recalculate the heuristic of the grid.



Shown below is a high-level overview of pseudocode for our program.

Order assignment

New_orders_list={N1(goal state),N2(goal state),N1(goal state)...}
if(New_order_list == not empty)
{
Robots_available=Check for empty compartment in robots (R1,R2,R3) // lets say R1,R2 are available
R1_distance,R2_distance= Calculate_minimum_Robots_distance for available robots(R1.R2);
Selected robot= Check min(R1_distance,R2_distance) // lets say R1 has minimum distance
Add order into orders list of selected robot R1
}

Robot R Module

Whenever initial position recalculate path based on time and fuel constraints;
Traverse path
current node=goal node make a stop;

Max_order=2

Functions

Robot path
R_full(R)
R_empty(R)
R_current_pos(R)
Current_num_orders(R)

Cal_path(initial node,goal node)
Calculate heuristic and generate path

Calculate_minimum_Robots_distance
n=Current_num_orders(R)

S1=first complete previous orders then take next one
For n orders:
[previous_path+cal_path(final goal of previous path,initial)+cal_path(initial,goal_new)]
Our example:For n=2
[cal_path(current_node,goal1)+cal_path(goal1,initial)+cal_path(initial,goal_new)]

S2=grab the next order first and then deliver all orders at once
For n orders:
[cal_path(current_node,initial)+{cal_path(initial,goal1_prev)cal_path(goal1_prev,goal2_prev)+...}++cal_path(goal1,goalN)]
Our example:For n=2
[cal_path(current_node,initial)+cal_path(initial,goal1)+cal_path(goal1,goalN)]

Return Min(S1,S2)

Pseudo code Description:

Order assignment:
So When the user will assign a new order it will go to new_order_list. A thread is running continuously to check whether there are any orders in that list. If the list is not empty then we will decide which robots should be assigned the new order by comparing the results of the function Calculate_minimum_Robots_distance for each robot. That function gives the most optimal path(shortest path) for each robot. Then the optimal paths obtained for all the robots are compared and the one robot with the shortest optimal path is assigned the next order.
Robot R:
There are two main functions in the robot module.
Cal_path: it takes two nodes as inputs and calculates the minimum distance between two nodes based on the heuristic values defined above.
Calculate_minimum _Robot_distance: The function gives the shortest distance for a robot to deliver all the orders (all the previous orders plus the new order).In our case it does that by calculating two paths S1 and S2 and then return the shortest of them back to the order assigning module.
Where in **S1**: the path robot will take is to complete the already assigned orders and then taking the next order.
While in **S2**: the robot would first take the new orders and then will deliver all orders at once.
These two distances are calculated using the cal_path module.

The other functions in the module Robot R are pretty much self explanatory.

Part B:

In this part we will be proposing a rule system that will bag the foot items in a robot's compartment. We propose that there will be three classes of items: large, medium and small, with each class having labels attached to each item such as if the item is a freezer item or if the item is fragile. We will be designing a simulation in which the robot will start with the large items and work their way up to small items. The robot will keep track if the item requires a freezer bag or a normal paper bag, with each bag containing a limited amount of items. In our simulation each bag can not contain more than ten items in a single bag. For fragile items we will place them into the bag last, an example of a fragile item would be a loaf of bread as if we put it on the bottom of the bag it would get crushed by the items on top of it. The simulation designed for these rules will have an input of a list of items with characteristics assigned to each item, the size, if it is fragile, or is a freezer item. The simulation will then go through from size of items to the items into bags, while doing it efficiently, i.e. least number of bags, and bagging each item into the corresponding bag corresponding to its characteristics.

The rules that we will be defining for the bagging simulation are:

R1: Bag large items first
R2: Bag medium items second
R3: Bag small items third
R4: Freezer items must go in a freezer bag
R5: Fragile items will be put together in a bag with weight limit half of normal bags
R6: Normal bags can contain less than 10 weight (large: 5, medium: 2, small: 1) create a new bag when items in the bag meet the weight limit
R7: For fragile items bag large items first
R8: For fragile items bag medium items second
R9: For fragile items bag small items third
R10: When fragile items bag meets the weight limit create a new bag
R11: For frozen items bag large items first
R12: For frozen items bag medium items second
R13: For frozen items bag small items third
R14: When frozen items bag meets the weight limit create a new bag

A simulation was run with the following rules, it was designed to load in a list of items that were purchased by a user. The list would be read in with the following characteristics: name, weight, freezer, and fragile. Using these characteristics we would go rule by rules for each item in the list to determine where the item should be bagged and in what order. The items would be

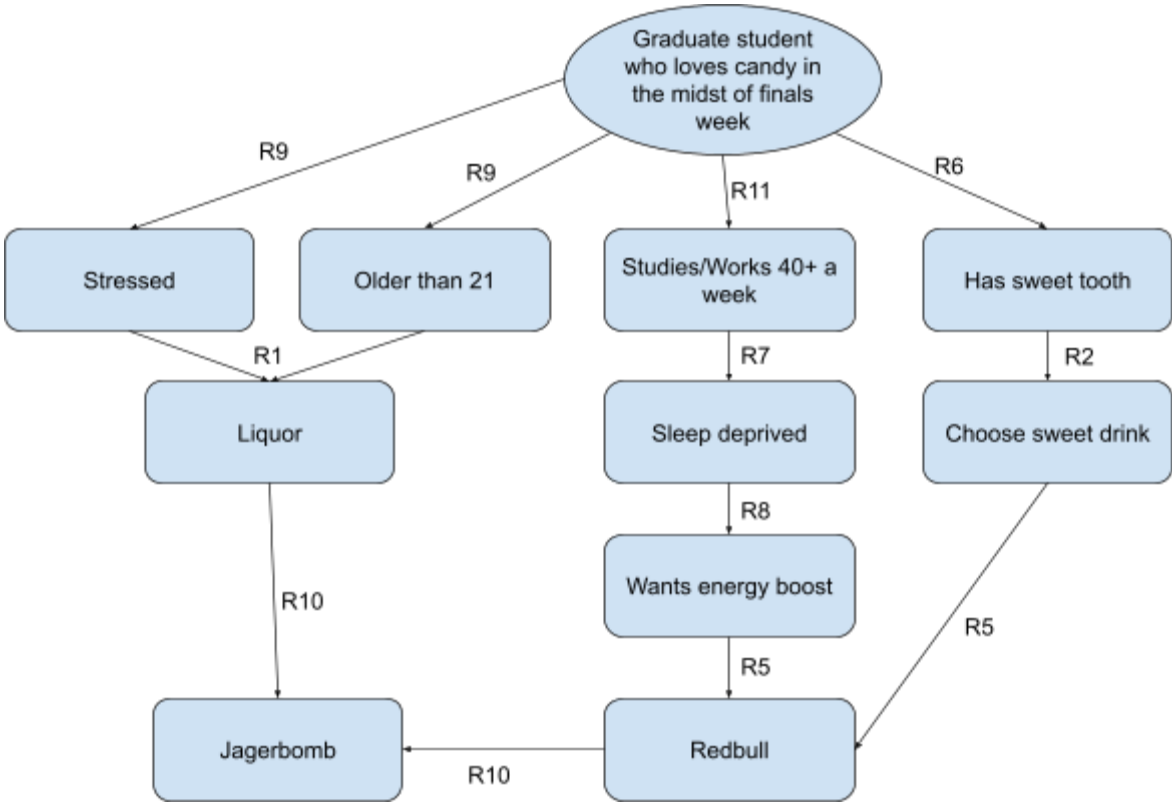
bagged in a way to try and fill the most items in the least amount of bags and comply with all the rules set in the simulation. An example of the code running can be found within the youtube link where we demonstrate this part in action.

Part C:

With the backward chaining system, we essentially want to scan the conclusions of each rule and check to see which rules support it.

Beverages: water, sparkling water, milk, soda, juice, wine, beer, liquor, with specific brands
Rules:
R1: If you can choose liquor, then the guest is stressed and older than 21.
R2: If you choose a drink that is sweet (soda, juice, energy drink), then guest has a sweet tooth
R3: If guest can only have non-dairy drinks, then guest is lactose intolerant
R4: If you choose sparkling water or soda then guest enjoys carbonated drinks
R5: If you choose a redbull, then guest wants an energy boost and has a sweet tooth
R6: If a guest has a sweet tooth, then the guest likes candy
R7: If guest is sleep deprived, then guest is an individual who studies 40+ hours a week
R8: If guest wanted an energy boost, then guest is sleep deprived
R9: If guest is stressed and older than 21, then they are a graduate student
R10: If Jagerbomb, then liquor + red bull
R11: If one studies 40+ hours a week, then it is finals week

What kind of drink would you give a graduate student in the midst of finals week?



Part D:

For our STRIPS rules, we are assuming that the robot can hold 3 bags. In order to fit all of the bags into the robot, they have to be stacked. In order to determine the order of bags from bottom to top, we are going to use weight. So, ideally, the heaviest bag will be on the bottom and the lightest bag will be on top.

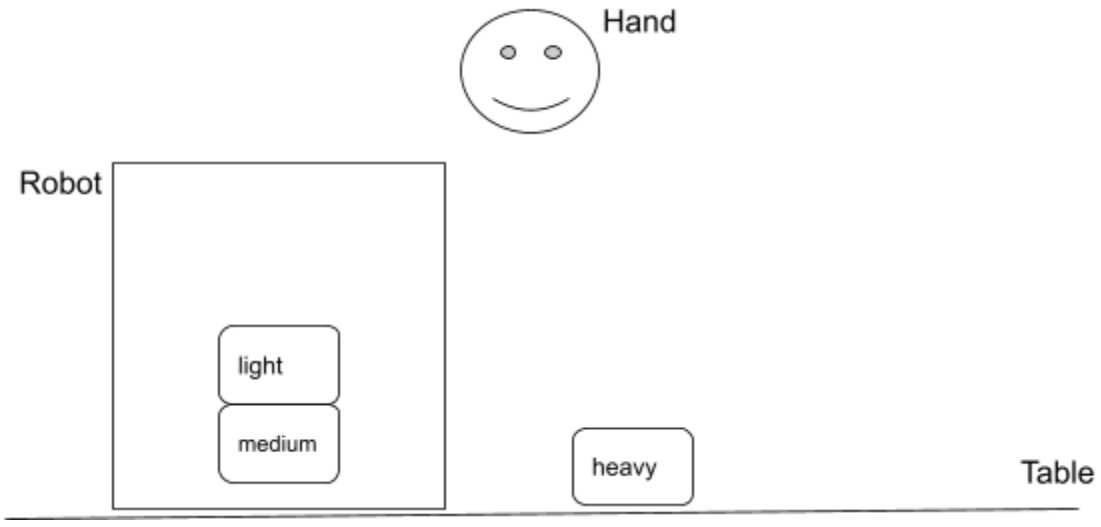
We will assume that the arm itself will perform the weighing. Since we need to compare the weight of the bags individually to each other, a bag that has been weighed is added and sorted to a priority list based on weight. This means that the heaviest bag will be first on the priority list since it will be first placed on the bottom of the robot.

STRIPS actions:
Definition of actions:

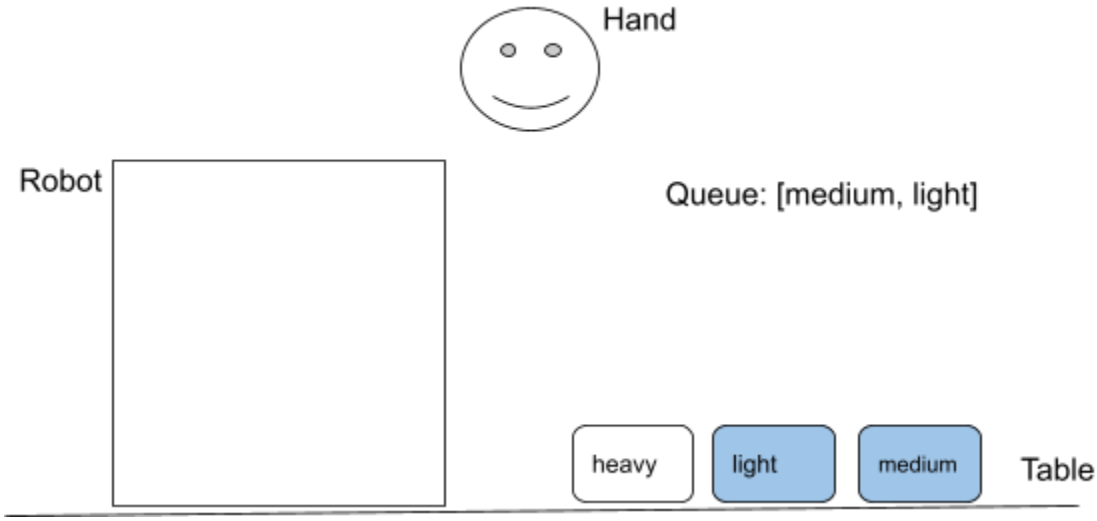
- handempty: the hand is not holding a bag
- clear(x): there is no bag on top of x
- holding(x): the hand is holding x
- notweighed(x): x has not been weighed and not in priority queue
- weighed(x): x has been weighed and sorted in priority queue
- ontable(x): x is on table

putdown(x)
P&D: holding(x)
A: handempty, clear(x), ontable(x)
pickup(x)
P&D: handempty, clear(x), ontable(x)
A: holding(x)
pack(x)
P&D: holding(x), clear(x), weighed(x)
A: inrobot(x), notweighed(x)
unpack(x)
P&D: inrobot(x), handempty, clear(x)
A: holding(x)
weigh(x)
P&D: holding(x), notweighed(x)
A: weighed(x)

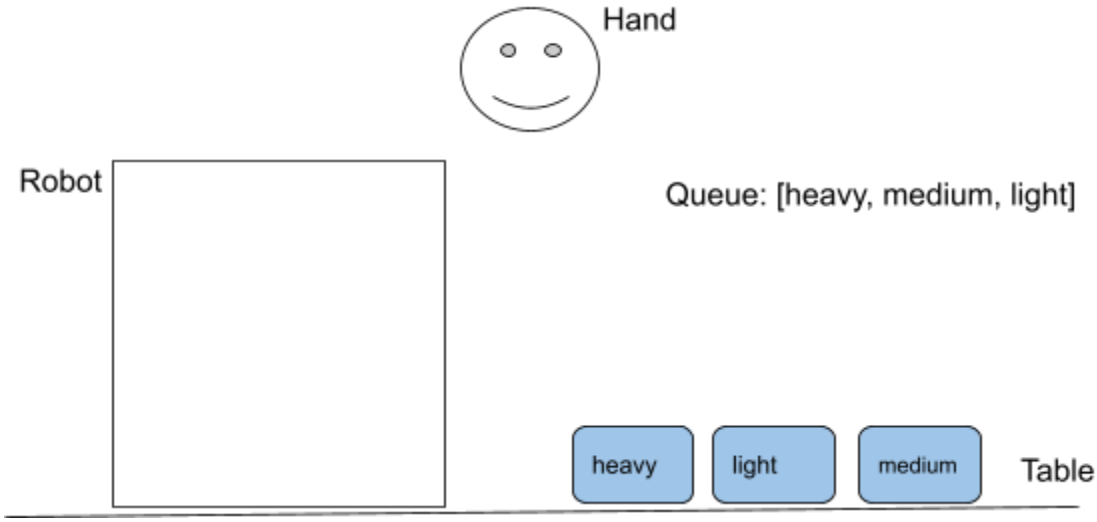
An example of these actions being in use can be shown below where a robot already has two orders in place, but an unexpected heavy order comes in.



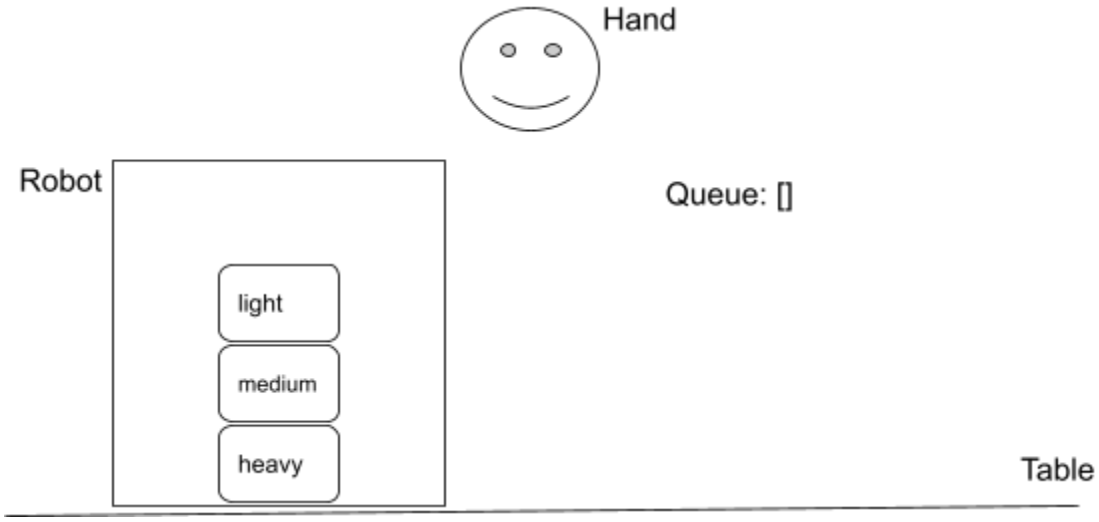
The hand would first need to make sure that the heavy bag is actually heavier than the bag that is on the bottom in the robot. In order to do this, we would need to have the hand unpack all the bags from the robot and weigh them individually. Light blue below shows that the bags have been weighed. The bags that have been weighed will be sorted into the priority queue from heaviest to lightest.



The robot would not be able to pack until all of the bags have been weighed. This means we would need to weigh the heavy bag and place it back onto the table.



After all bags have been weighed, the hand will begin packing the bags from heaviest to lightest.



The ultimate sequence will be this: unpack(light), weigh(light), putdown(light), unpack(medium), weigh(medium), putdown(medium), pickup(heavy), weigh(heavy), putdown(heavy), pickup(heavy), pack(heavy), pickup(medium), pack(medium), pickup(light), pack(light)

Remember that there is a priority queue in the background that determines the order of placement. As we can see, even if there is an unexpected bag/order, there is always a combination of actions that can be placed to ultimately achieve the heaviest bag on the bottom and the lightest on the top.