The BRITISH UNIVERSITY IN EGYPT

Café Robot

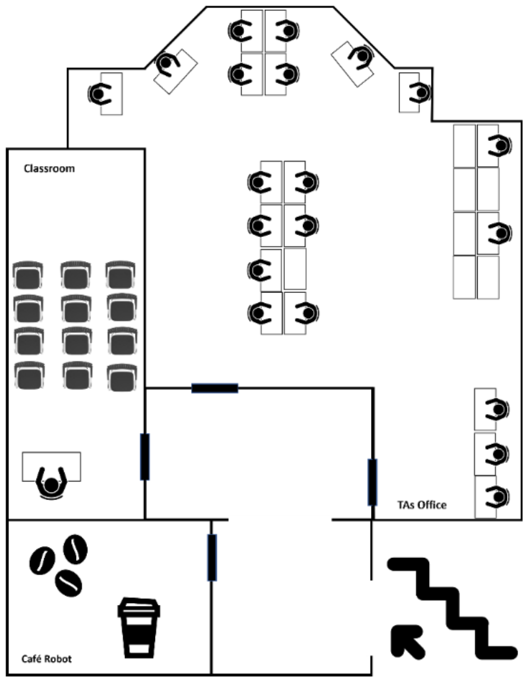
Delivering Coffee to Teaching Staff

ID	Student Name	E-mail
196280	Ashraf Adel	ashraf196280@bue.edu.eg

Table of Contents

Introduction: Scenario Setup

The Café Robot scenario is as following:
Given environment "E":



We want the robot "R":

To deliver "Y" number of coffee to "X" number of staff members where each member requests "Z" number of coffee. Succinctly:

$$Y = \sum_{i=1}^{X} Z_i$$

From now on, we'll call the operation of the robot moving from its current position to the next staff member (or back to the cafe room) as "op".
Moreover, it is required that the robot moves as efficiently as possible. In other words, to always take the shortest path per each "op".
When the robot delivers coffee to the last staff member, it should return to the cafe room "cfr" (another operation).
To summarise, if we refer to the number of times "op" occur as "ops", then:

$$ops = X + 1$$

Problem Assumptions

Since we need to represent “E” in a succinct way for the a.i program to properly accomplish its task, we have to recognize and address all of the environment, staff, and robot details that were not explicitly stated in the aforementioned introduction.

Assumptions Regarding The Environment

1. Doors: The doors don’t have knobs, and automatically close if someone opens them. Therefore, “R” can just push the door with its hand to open it. After passing through the door, it will close, as the staff don’t want any open doors left by “R”.

2. Coffee types: Since training “R” to recognize different coffee types is a time-consuming task for the IT, it’s still being developed. Therefore, “R” will be currently programmed to recognize a certain type of coffee, so the staff members can only choose that type of coffee.

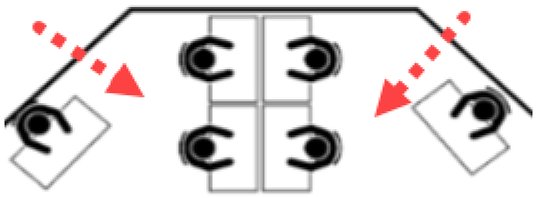
3. Method of ordering coffee: each staff member can log on the mobile application connected to “R” and order “Z” amount of coffee. The relative time that the order was made will be also logged. “Relative time” means in relation to other orders. For example, if you have 2 orders “o1” and “o2”, where “o1” was made before “o2”, then that information will be stored in the application and is known to “R”.

4. Initial state: From the scenario’s diagram, one can notice that the coffee cups and beans are placed in “cfr”. This means that “R” has to first prepare “Y” coffee (by going to the beans and preparing them in the cups) before the first “op”. Since this process will be done no matter how many staff members want coffee, we could therefore start our initial state with the following: “R” will have seen the orders of all “X” staff members, and will have prepared and carried all “Y” coffee, such that the next state will be an “op” to the first staff member that made the order (First-Come-First-Served (FCFS) policy).

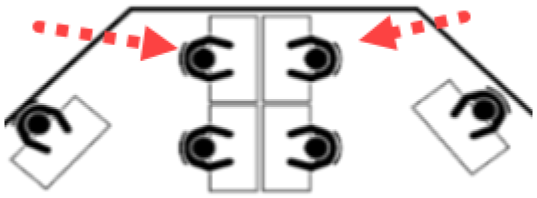
Assumptions Regarding Staff’s Needs

1. Coffee delivery policy: Since no one wants to wait a long time after they’ve made the order, the staff members have decided that the robot should serve the orders in a First-Come-First-Served (FCFS) fashion.

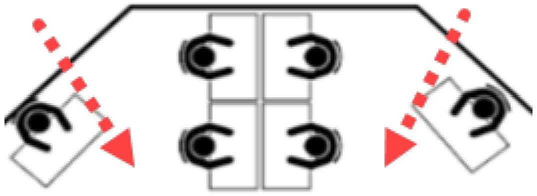
2. Coffee placement concerns: The staff members are concerned about the off-chance that “R” accidentally spills the coffee on their desk. Therefore, they want “R” to come and deliver the coffee directly to the staff member, so that they may put it on the desk themselves. However, in some cases, such as these ones:



The staff members are afraid that “R” will spill the coffee due to the narrow space. Therefore, these staff members:



have agreed to personally go to “R” here:



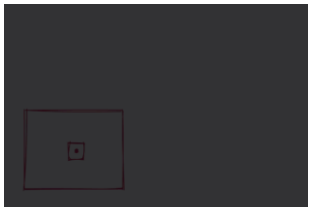
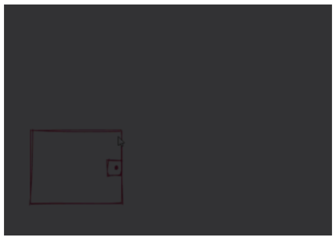
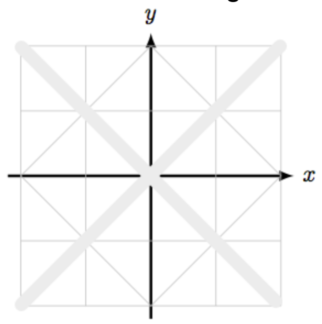
in order to take the coffee themselves. An example for this is the third scenario presented in the “illustrations” section.

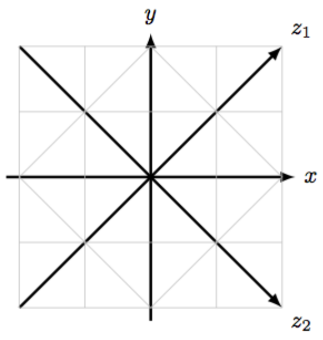
3. Number of coffee ordered (“Z”) by a staff member: Since it’s previously stated that there is only one type of coffee, it doesn’t make sense for a staff member to order 2 cups of coffee of the same type. Even if said staff member wants to drink more than one cup, they logically won’t order the cups at the same time, as one will get cold as they’re drinking the other. Therefore, the app will restrict “Z” to be equal to 1, where said staff member can make another order if they want to drink another coffee.

4. Disturbance policy: The staff members need to focus on their work, so they do not want R to go near them while traversing the floor except when R is supposed to give them coffee. An example for this is the second scenario presented in the “illustrations" section.

Assumptions Regarding The Robot

1. Movement method: The robot can move in an omnidirectional and orthogonal fashion. (due to how its wheels are built).
   Side note: omnidirectional movement is like this:


While unidirectional movement is like this:

Meanwhile, orthogonal movement is illustrated as:


While the octilinear movement [1] is illustrated as:


2. How much coffee “R” can carry: since R’s hands are so steady. It can carry a very large tray on its hand where the tray can be multi-leveled (since the faculty could provide a lot of trays):

This pose is adapted from the African culture [2], where you can find Africans carrying

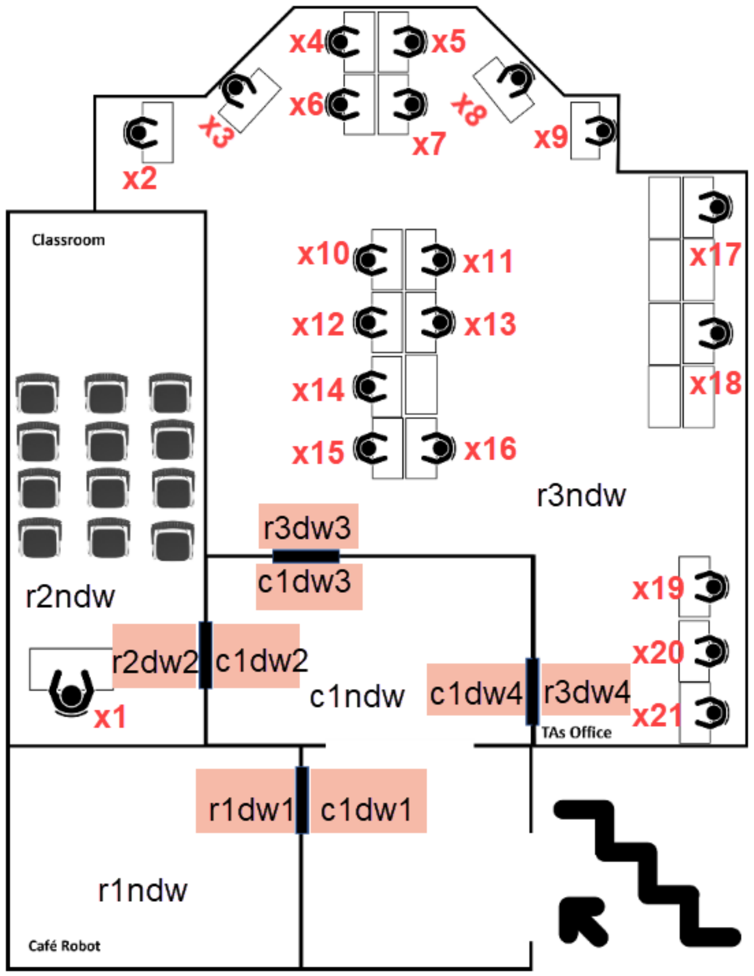
various objects on their heads [3]. Therefore in the initial state, “R” can carry all coffee no matter how many staff members have ordered. Which means that “R” will go to “cfr” only once; when it is done with all the orders.

Problem Approaches

The Café problem can be represented in multiple ways. The following section explains some of these representations.

Space Representation

One possible representation is to break down each room as a set of spaces; where there are spaces for both the floor and objects in each room. Illustration:

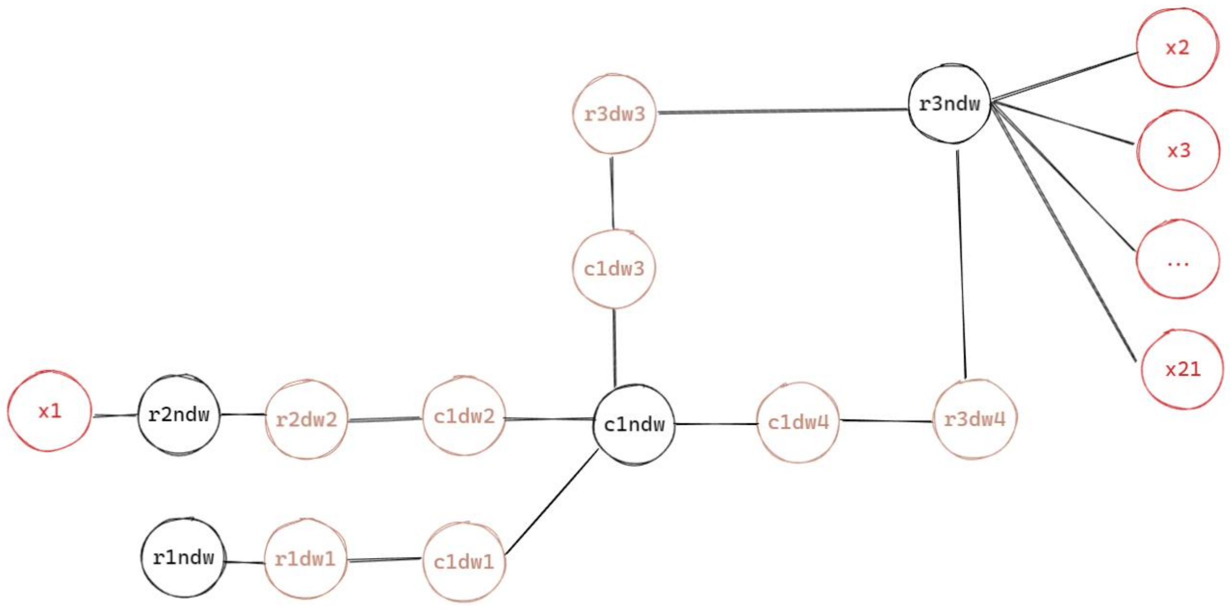


Such that:
“r”: “room”
“c”: “corridor”
“ndw”: “no door way”, which represents the empty floor space in a room

"dw" : "door way", which represents the space near a door
X = [x_1 , x_2 , … , x_W]: "X" here represents a staff member, "W" represents the number of staff who ordered coffee (so there's a space at each X).

Graph Representation

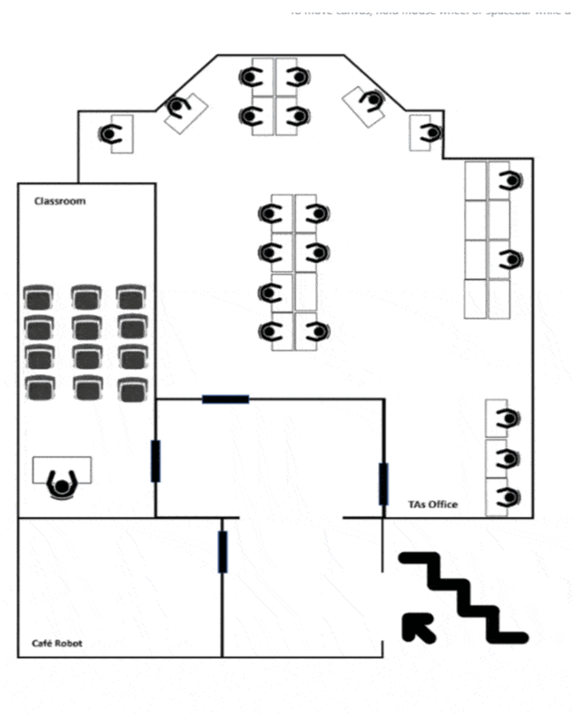
Another possible representation is to consider each "object/space" of interest in each room as nodes, where each node n_i is connected to its neighbouring "objects" (via edges) that R can go to, given that its currently at node n_r. Illustration:



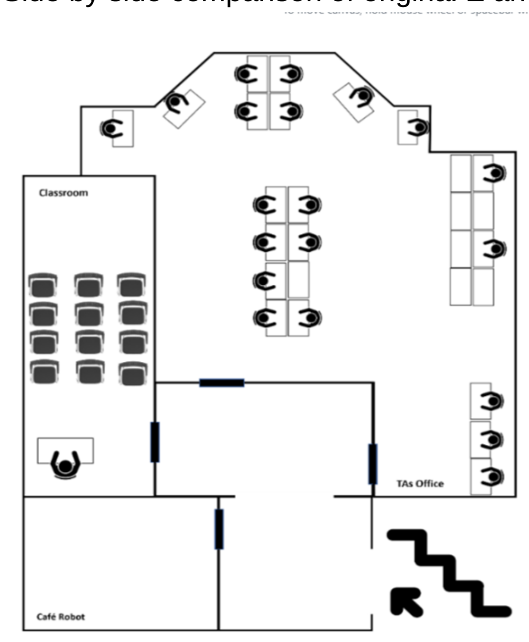
Such that the terminology here is the same as the terminology mentioned in the space representation.

Grid Representation

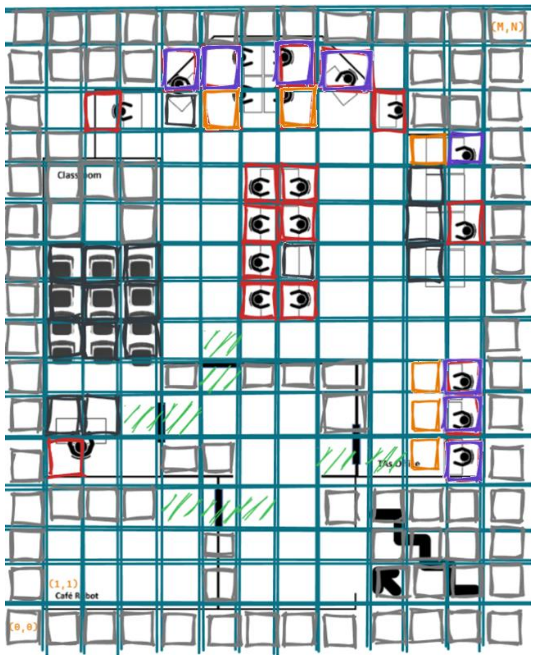
The last approach proposed is to consider each space, object, or part of an object as a square (block) on a grid, where the size of each block is based on the smallest "object" in E. In our case, it's the robot and staff member, not an object! Therefore, E can be represented in the following manner:



Side by side comparison of original E and its grid representation:



→



Erratum:
There are supposed to black cells here, as there are desks present:

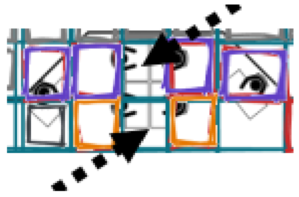


Diagram notation:

(1,1) → The location where R will start and end at

(M,N) → "M" represents the number of columns and "N" represents the number of rows

⬛ → A cell that contains an object (R can't go to these cells)

⬜ → A cell that represents part of the building's architecture (R can't go to these cells)

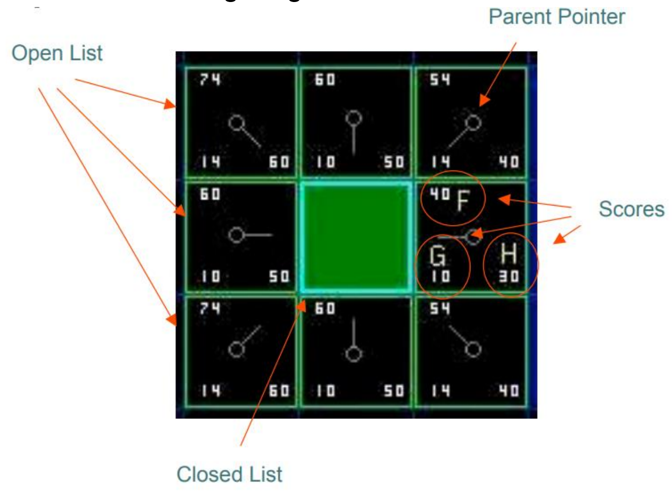
🟥 → A cell that represents a staff member (R can't pass by these cell except the cell where a staff member, specifically the next in the queue, has ordered coffee, noting that if that happens, R has to go back from that cell once it hands the coffee to that member)

🟪 → A cell that represents the actual location of a staff member (present only for illustration, will be assumed as a regular unpassable object when implementing the grid representation)

🟧 → A cell that represents where R will go to, to deliver coffee to staff members located at that gold cell's violet cell(s) (🟪) (Note that the same rules of 🟥 also applies on 🟧)

▨ → A cell that represents a possible location from which R can go through a door

Note: the following diagram is to illustrate certain terminologies regarding this representation [4]:



Green tile: the current position of "R"

Scores (explained in depth in the "planning strategies" section):

F = G + H

G = the movement cost to move from the starting point to the given square following the generated path.

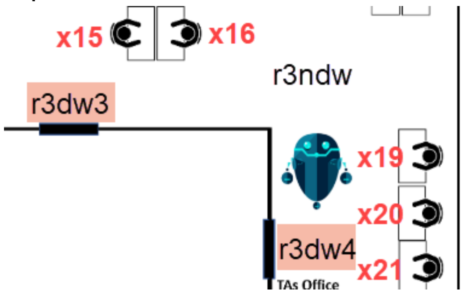
H = the estimated movement cost to move from the given square to the target.

Chosen Representation

The approach that will be used to solve the café problem is to map the given E into the grid representation. The reason for this is divided into two sections:

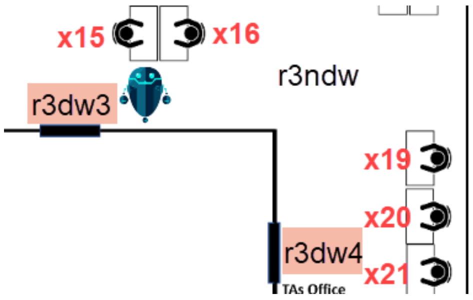
Limitations of Other Representations

Both space and graph representations are not as explicit as the grid representation, which will possibly ease their code implementation. However, this level of implicitness will also cause vagueness in some details about the implementation, specifically in how the searching algorithm will find the shortest path for any “op”. Consider the following example for the space representation:



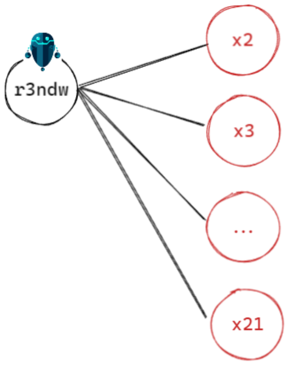
(Scenario 1)

In this situation, “R” is closer to x_{21} than to x_{15}. However, if he’s here:



(Scenario 2)

The reverse is true. The problem is that one cannot pin-point R’s location, as in both situations, he’s in “r3ndw”. The same problem occurs in the graph representation:



One possible solution is to assign costs on each edge to indicate the distance from “r3ndw” to each respective x staff member. However, this will have the same issue of ambiguity mentioned previously; the exact location of R in “r3ndw” is not known. If we’re in scenario 1, then the cost from “r3ndw” to x_{21} should be less than that of x_{15}, and the reverse is true for scenario 2. To solve this issue, extra nodes could be added representing the specific spaces at “r3ndw”, but even then, we have to represent the specific spaces that we think are relevant along with costs to the staff from there, and all of these assumptions are subjective.

Benefits of Grid Representation

Even though the grid representation is more explicit and thus could have a bigger time and space complexity when implementing its code, it will bypass the ambiguity issues that were found in space and graph representations, as the room is not just broken down to its important (relevant) parts, but rather all the small details are mentioned as well.

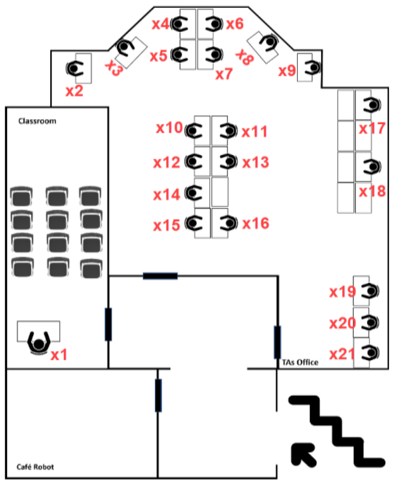
Computer Representation of The Problem

The following section describes detailed operations (functions) that R can do in E coupled with examples.

Notations

Below are re-statements of previous notations mentioned at the “scenario setup” section, and the introduction of new notations:

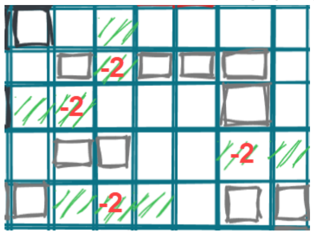
- R → The robot
- E → The environment
- X_i → the ith staff member who ordered coffee:



Noting that due to the previously made assumptions, X_3, X_4, and X_5 will be considered by

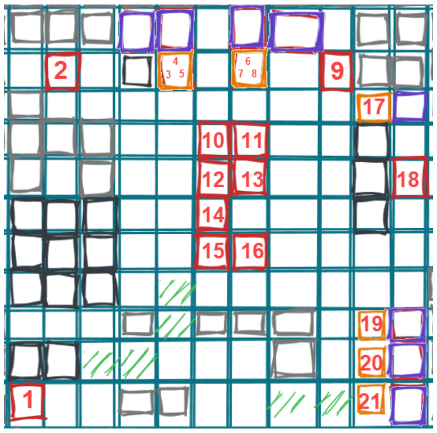
R to be the location of X_5, while X_6, X_7, and X_8 will be considered by R to be the location of X_8

- W → the total number of staff members who ordered coffee
- op_i → the ith operation of the robot moving from its current position to X_i (such that X_{W+1} is the operation of the robot going back from its current position to the starting location)
- Z_i → The number of coffees ordered by X_i, noting that from the assumptions mentioned previously, it is always assumed that $Z_i = 1 \quad \forall\, i \in \{1,2,\ldots,W\}$
- (14,1) (0-based index) → The location where R will start and end at (note that it is different than (1,1) mentioned in the "grid representation" section, as we're dealing with (rows, cols) in python instead of (x, y) coordinates, and starting from top left, not bottom left, check "env" list of lists below for clarification)
- (N, M) → "N" represents the number of rows and "M" represents the number of columns (note that this differs than the notation introduced in the "grid representation" section, as we're dealing with (rows, cols) in python instead of (x, y) coordinates).
- -1 → cells that R can't go through (these cells represent objects, building architecture, staff members that aren't the next in the order queue (or have already received coffee))
- -2 → closed door (for simplicity, the door itself will be represented, while the area around it is considered empty space (0)):



- 0 → cells that R can go through (empty space in the environment, or opened door)
- wentThroughDoor → Boolean variable that indicates if the parent of the current node (cell) was a door; that door is closed (0 to -2) when this variable is true

- From 1 to 21: the staff member locations (according to R):



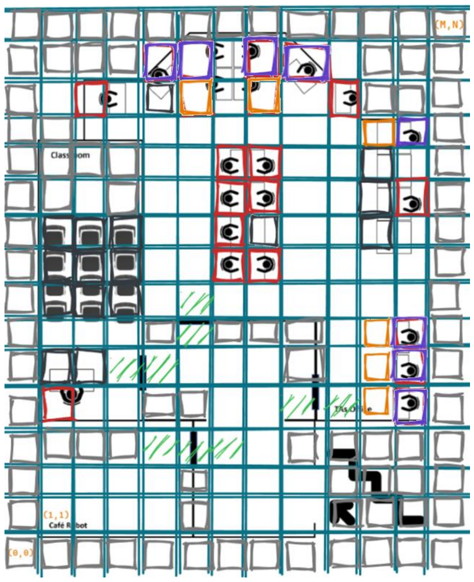
- directions → list of possible orthogonal movements:
  [⬆ , ➡ , ⬇ , ⬅] = [(-1,0), (0,1), (1,0), (0,-1)]
  (such that the first value of tuple is the row number, and the second argument is the column number)

- Given the notations above, the initial setup of E will be the following:

  - orders → a queue of orders that R will respond to. Example: [1, 21, 2] which means coffee should be delivered to the following staff members in order: Dr Gerard, T.A Toka, T.A Sara (pre-processing step: if X_3 or X_4 order something, they will both be added to the "orders" list as "5". Same goes for X_6 and X_7; they will be stored as "8"

  - env → a list of list (2D grid) which represents the environment:

```
env = [[-1, -1, -1, -1, -1, -1, -1, -1,  -1, -1, -1, -1, -1],
       [-1, -1, -1, -1, -1, -1, -1, -1,  -1,  0, -1, -1, -1],
       [-1,  0,  2,  0,  -1,  5, -1,  8,   0,  9, -1, -1, -1],
       [-1, -1,  0,  0,   0,  0,  0,  0,   0,  0, 17, -1, -1],
       [-1, -1, -1, -1,   0,  0, 10, 11, 0,  0, -1,  0,  -1],
       [-1, -1,  0, -1,   0,  0, 12, 13, 0,  0, -1, 18, -1],
       [-1, -1, -1, -1,   0,  0, 14,  0,  0,  0, -1,  0,  -1],
       [-1, -1, -1, -1,   0,  0, 15, 16, 0,  0,  0,  0,  -1],
       [-1, -1, -1, -1,   0,  0,  0,   0,  0,  0,  0,  0,  -1],
       [-1,  0,  0,  0,  -1, -2, -1,  -1, -1,  0, 19, -1, -1],
       [-1, -1, -1,  0,  -2,  0,  0,  0,  -1,  0,  20, -1, -1],
       [-1,  1,  0,  0,  -1, -1,  0,  0,  -2,  0,  21, -1, -1],
       [-1, -1, -1, -1,  0, -2,   0,  0,  -1, -1, -1, -1,  -1],
       [-1,  0,  0,  0,  0,  -1,   0,  0,   0,  -1, -1, -1, -1],
       [-1,  0,  0,  0,  0,  -1,   0,  0,   0,  -1, -1, -1, -1],
       [-1, -1, -1, -1, -1, -1,  -1, -1,  -1,  -1, -1, -1, -1]]
```



  - paths → a list of lists of tuples (directions) that R will make to succeed in all W+1 ops. An example is provided in the "example" section of "operations".

Operations

The following are the operations that will be responsible for R’s traversal and actions in E (note that the parameters/return values mentioned below may not be included in the final implementation if the code is implemented as a class, as the parameters/return values will be attributes in that class which could be accessed by the class’s methods)

chooseNextLocation(openList)

Makes R identify the location with the least cost (based on A* algorithm), so that it knows which location it should go to next from its current location. Pseudo-code:

1. If openList is empty, then the goal state is unattainable
   (Side note: this will never happen given the predefined environment configuration)
2. Remove the node (location) with the least cost in openList and check:
   a. If It’s a goal node, then call executeOp() and terminate the function
   b. Else, add it to closedList, and add it to “path” list
3. return the path, closedList and currentNode (chosen location)

- **Preconditions**: State-wise: The location of the robot to be in a position adjacent to the new chosen location. Computer-wise: The openList is not empty and at least one node in that list is walkable
  (side note: the second condition is guaranteed to be true, as generateLocations() checks for validity of nodes before pushing them to openList)
- **Delete**: The current location that R is at (currentNode)
- **Add**: The new location that R will go to (computer-wise: into the closed list)

executeOp(path)

Makes R finish an “op”; from going and delivering coffee to updating its perception about the environment (note that this function doesn’t affect the state, that’s why “delete” and “add” don’t affect the state; the function is only relevant code-wise). Pseudo-code:

1. Call traversePath()
2. Call changeEnv()

- **Preconditions**: The current node is the goal node (i.e. R has searched and found a path that will lead it to the next staff member)
- **Delete**: Nothing
- **Add**: The path that R will traverse

traversePath(path)

Makes R actually go from its current location to the goal location; opening (i.e. -2 to 0) and closing (i.e. 0 to -2) doors whenever they appear. Pseudo-code:

1. Reverse "path" list
2. For each tuple t_i in "path":
    a. If it's corresponding place in "env" (env[t_i]) is "-2", then change that location to "0", and assign "wentThroughDoor" to true to signify that R has opened the door
    b. Else, if "wentThroughDoor" is true, then assign the previous location (env[t_{i-1}]) to "-2" instead of "0", to signify that the door is closed again
    c. Else, continue
3. Signify that the coffee has been delivered!

- **Preconditions**: Same preconditions of executeOp()
- **Delete**: State-wise: The location that R was previously at
- **Add**: State-wise: The location that R will go to next

updateEnv(path)

Makes R perceive the environment differently in such a way that people whom R delivered coffee to are no longer accessible (R can't go to them in order not to disturb them). Pseudo-code:

1. Add the reverse of the backtracked path to the list of paths that R have traversed
2. If R is back at the starting location, then terminate the function, as all W+1 ops are finished!
3. Re-initialize "path" list with currentNode
4. Change currentNode to parentNode (path[1]) (to make R step away from that person after delivering coffee in order not to disturb them any further)
5. Make R perceive that person as a non-walkable location (by changing the staff member's representation from "i" to -1 in the environment)
6. Re-initialize openList with currentNode (for R to start traversing to the next goal (staff member)) and re-initialize closedList with parentNode (for R to not go to the previous staff member again)
7. Add the currentNode to openList with f = h, g = 0

- **Preconditions**: State-wise: R needs to be at the location where it has delivered the order (goal node). Code-wise: “orders” queue is not empty. Otherwise, only step 1. will be executed

- **Delete**: The previous state of the staff member R has just delivered coffee to (previous state: staff member). Code-wise: the elements of openList and closedList (re-initialized)

- **Add**: The new state of that staff member (new state: obstacle; as R will consider this place as an obstacle that it shouldn’t go through). Code-wise: the new currentNode to openList

generateLocations(currentNode)

Makes R perceive the new locations that it can go to from its current location (in an orthogonal fashion). Pseudo-code

1. Get the 4 locations adjacent to currentNode using “directions” list
2. For each location:
    a. If the current location (curLoc) is not walkable (i.e. -1), then continue to next location.
    b. If curLoc is in the closedList (previously visited), then the optimal path for it was already found and we don’t need to traverse it again. Therefore, continue to next location
    c. Else, calculate g(curLoc), h(curLoc), and f(curLoc)
    d. If curLoc is in the openList, then:
        i. If f(curLoc) > f(curLoc in openList), then we’ve already found a better (cheaper) path to curLoc. Therefore, continue to next location
        ii. Else, go to e.
        (side note: why not delete curLoc in openList? Because there’s a possibility that f(curLoc) == f(curLoc in openList), and in that case, we should keep both paths in the priority queue)
    e. Else, add curLoc to the openList

- **Preconditions**: State-wise: the current location of R in the environment. Code-wise: Nothing; the function will always generate 4 locations wherever R is placed.

- **Delete**: Nothing, as the reason for keeping curLoc in openList was explained in d. ii.

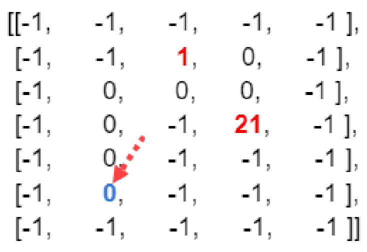
- **Add**: State-wise: the new locations that R can go to in the environment. Code-wise: each valid curLoc.
(side note: “valid” here is refers to openList and closedList conditions, not walkable/non-walkable conditions)

Example

This example is the same as “scenario 1” in the “illustrations” section, but focuses on the actual computer representation, instead of just presenting a high-level abstraction, and is done in a much smaller environment. The example is also limited to 2 ops, instead of 3 (R will go to “1”, then back to its initial location):

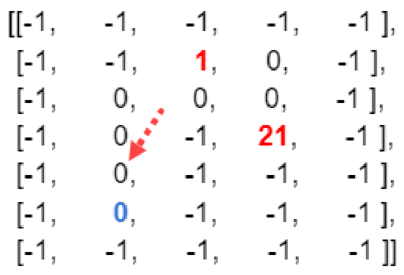
1. Initial environment:
   Orders = [1, 21]
   ```
   env = [[-1,    -1,    -1,    -1,    -1 ],
          [-1,    -1,     1,     0,    -1 ],
          [-1,     0,     0,     0,    -1 ],
          [-1,     0,    -1,    21,    -1 ],
          [-1,     0,    -1,    -1,    -1 ],
          [-1,     0,    -1,    -1,    -1 ],
          [-1,    -1,    -1,    -1,    -1 ]]
   ```
   curNode = {r:5, c:1, g:0, h:5, f:5}
   (Note 1: in implementation, curNode won’t be a dictionary, this notation here is just a simplification)
   (Note 2: h = Manhattan distance = | curNode.r-goalNode.r | + | curNode.c-goalNode.c | = | 5 - 1 | + | 1 - 2| = 4 + 1 = 5)

2. chooseNextLocation() will choose this location (which is expected, as it’s currently the only one in the openList):



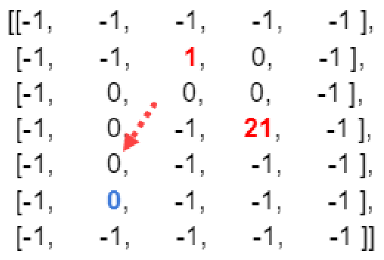
```
[[-1,   -1,   -1,   -1,   -1 ],
 [-1,   -1,    1,    0,   -1 ],
 [-1,    0,    0,    0,   -1 ],
 [-1,    0,   -1,   21,   -1 ],
 [-1,    0,   -1,   -1,   -1 ],
 [-1,    0,   -1,   -1,   -1 ],
 [-1,   -1,   -1,   -1,   -1 ]]
```

3. generateLocations() will add the following location to openList after calculating its f value:



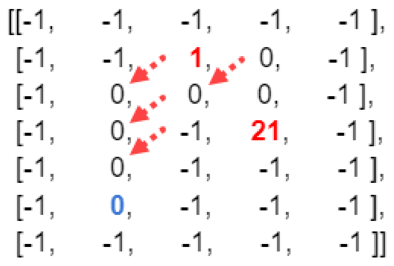
```
[[-1,   -1,   -1,   -1,   -1 ],
 [-1,   -1,    1,    0,   -1 ],
 [-1,    0,    0,    0,   -1 ],
 [-1,    0,   -1,   21,   -1 ],
 [-1,    0,   -1,   -1,   -1 ],
 [-1,    0,   -1,   -1,   -1 ],
 [-1,   -1,   -1,   -1,   -1 ]]
```

4. chooseNextLocation() will choose this location, as it has the lowest f cost = 4 (and because it’s the only one in the openList):



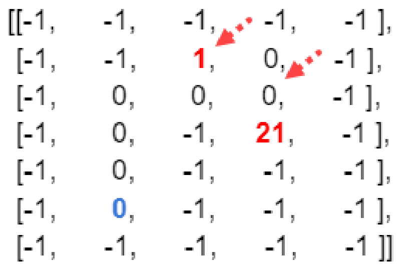
```
[[-1,   -1,   -1,   -1,    -1 ],
 [-1,   -1,    1,    0,    -1 ],
 [-1,    0,    0,    0,    -1 ],
 [-1,    0,   -1,   21,    -1 ],
 [-1,    0,   -1,   -1,    -1 ],
 [-1,    0,   -1,   -1,    -1 ],
 [-1,   -1,   -1,   -1,    -1 ]]
```

5. Repeat generateLocations() and chooseNextLocation() until:



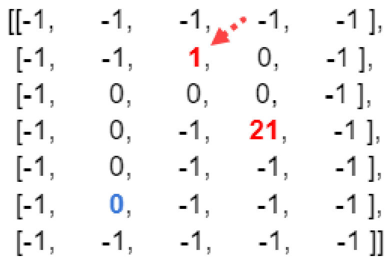
```
[[-1,   -1,   -1,   -1,    -1 ],
 [-1,   -1,    1,    0,    -1 ],
 [-1,    0,    0,    0,    -1 ],
 [-1,    0,   -1,   21,    -1 ],
 [-1,    0,   -1,   -1,    -1 ],
 [-1,    0,   -1,   -1,    -1 ],
 [-1,   -1,   -1,   -1,    -1 ]]
```

6. Then, generateLocations() will give the following locations:



```
[[-1,   -1,   -1,   -1,    -1 ],
 [-1,   -1,    1,    0,    -1 ],
 [-1,    0,    0,    0,    -1 ],
 [-1,    0,   -1,   21,    -1 ],
 [-1,    0,   -1,   -1,    -1 ],
 [-1,    0,   -1,   -1,    -1 ],
 [-1,   -1,   -1,   -1,    -1 ]]
```

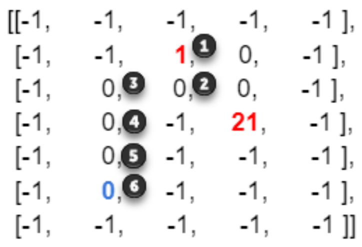
7. chooseLocation() will then choose the goal node:



```
[[-1,   -1,   -1,   -1,    -1 ],
 [-1,   -1,    1,    0,    -1 ],
 [-1,    0,    0,    0,    -1 ],
 [-1,    0,   -1,   21,    -1 ],
 [-1,    0,   -1,   -1,    -1 ],
 [-1,    0,   -1,   -1,    -1 ],
 [-1,   -1,   -1,   -1,    -1 ]]
```

And call executeOp(), which will call traversePath(), then updateEnv()

8. traversePath() will reverse the "path" from this:

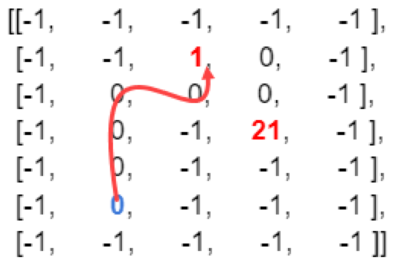


```
[[-1,   -1,   -1,   -1,    -1 ],
 [-1,   -1,    1,❶  0,    -1 ],
 [-1,    0,❸  0,❷  0,    -1 ],
 [-1,    0,❹ -1,   21,    -1 ],
 [-1,    0,❺ -1,   -1,    -1 ],
 [-1,    0,❻ -1,   -1,    -1 ],
 [-1,   -1,   -1,   -1,    -1 ]]
```

To this:

```
[[-1,   -1,   -1,    -1,   -1 ],
 [-1,   -1,    1,❻   0,   -1 ],
 [-1,    0,❹   0,❺   0,   -1 ],
 [-1,    0,❸  -1,    21,   -1 ],
 [-1,    0,❷  -1,    -1,   -1 ],
 [-1,    0,❶  -1,    -1,   -1 ],
 [-1,   -1,   -1,    -1,   -1 ]]
```

And will signify whenever R passes by a door while traversing the path:

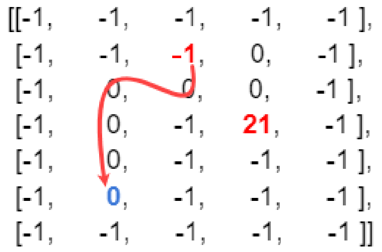


```
[[-1,   -1,   -1,   -1,   -1 ],
 [-1,   -1,    1,    0,   -1 ],
 [-1,    0,    0,    0,   -1 ],
 [-1,    0,   -1,   21,   -1 ],
 [-1,    0,   -1,   -1,   -1 ],
 [-1,    0,   -1,   -1,   -1 ],
 [-1,   -1,   -1,   -1,   -1 ]]
```

9. updateEnv() will add "path" to list of paths:
   paths = [[(5,1), (4,1), (3,1), (2,1), (2,2), (1,2)]], then it will reset path = [], then updates the environment such that R can't go to "1" now:

```
[[-1,   -1,   -1,   -1,   -1 ],
 [-1,   -1,   -1,    0,   -1 ],
 [-1,    0,    0,    0,   -1 ],
 [-1,    0,   -1,   21,   -1 ],
 [-1,    0,   -1,   -1,   -1 ],
 [-1,    0,   -1,   -1,   -1 ],
 [-1,   -1,   -1,   -1,   -1 ]]
```

10. The previous steps are repeated until R reaches its initial location:



```
[[-1,   -1,   -1,   -1,   -1 ],
 [-1,   -1,   -1,    0,   -1 ],
 [-1,    0,    0,    0,   -1 ],
 [-1,    0,   -1,   21,   -1 ],
 [-1,    0,   -1,   -1,   -1 ],
 [-1,    0,   -1,   -1,   -1 ],
 [-1,   -1,   -1,   -1,   -1 ]]
```

   Noting that updateEnv() will make
   paths = [[(5,1), (4,1), (3,1), (2,1), (2,2), (1,2)], [(1,2), (2,2), (2,1), (3,1), (4,1), (5,1)]]

Planning (Search) Strategies

There are many search strategies available for this problem, discussed below are some of them:

Breadth First Search (BFS)

- It is a searching algorithm which explores all the nodes at the present depth before moving on to the nodes at the next depth level [5]

- It uses queue data structure for its implementation

- Can get into an infinite loop if you don't store previously visited nodes

- It is one of the uninformed search techniques

    - Uninformed search is an algorithm that blindly follows predefined steps regardless of whether they're efficient or in-efficient [6]. In other words, it can't determine in advance which is the next suitable step to take in order to reach the goal node.

- Algorithm steps:

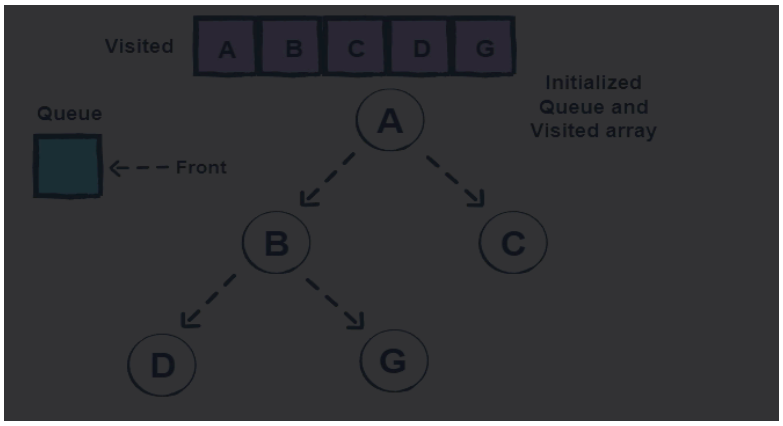
    - Pick a node (initially the root) and enqueue all its adjacent nodes (children) into a queue.

    - Dequeue a node from the queue, mark it as visited and enqueue all its adjacent nodes into a queue.

    - Repeat this process until the queue is empty or you meet a goal.

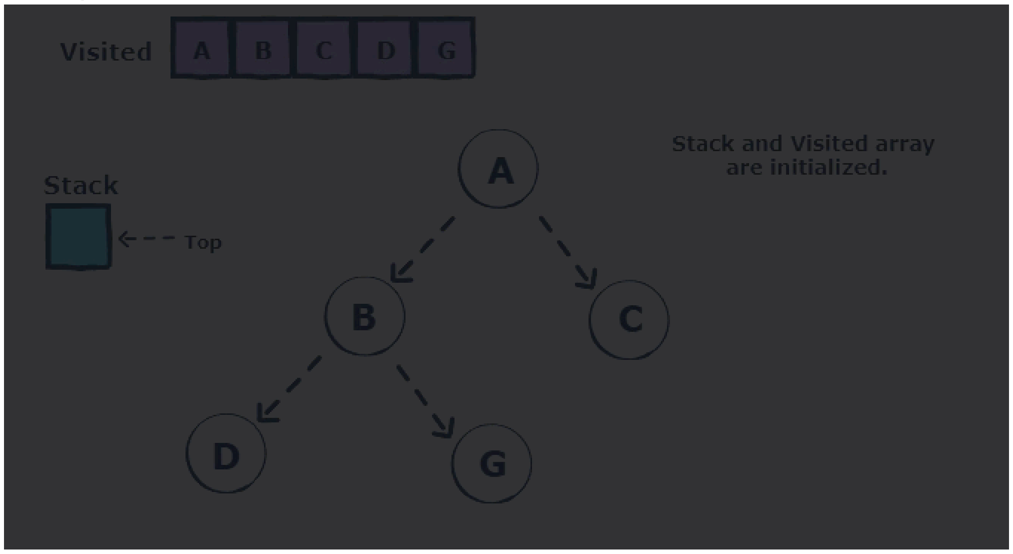
- Time complexity: O(n) where n is the number of nodes.

- Example:



Depth First Search (DFS)

- It is a searching algorithm which explores all the nodes by going forward if possible or uses backtracking
    - Backtracking is the action of moving backwards from a current node to the previous one (its parent) and exploring nodes on the same path. Moving backwards occurs whenever this current node has no children (i.e. a leaf node), when it is already visited, or when it is the goal node [7]
- It uses stack data structure for its implementation
- Can get into an infinite loop if you don't store previously visited nodes
- It is one of the uninformed search techniques
- Algorithm steps:
    - Pick a node (initially the root) and push all its adjacent nodes (children) into a stack [8]
    - Pop a node from the stack and push all its adjacent nodes into a stack
    - Repeat this process until the stack is empty or you meet a goal
- Time Complexity: O(n) where n is the number of nodes
- Example:

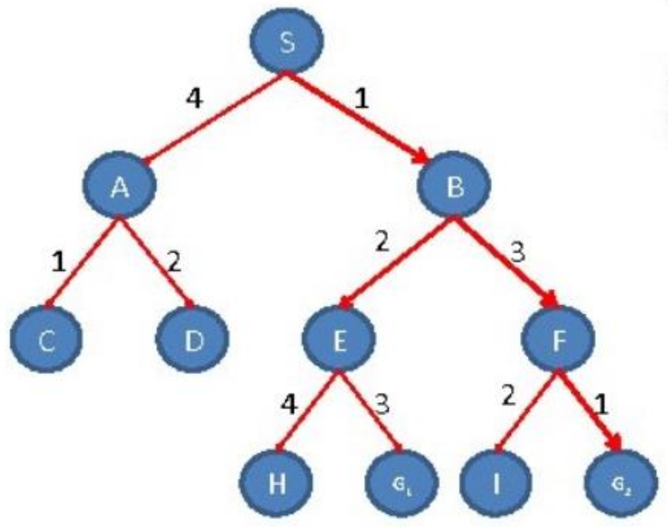


Best First Search

- Considered an informed method of searching
    - Informed method is a way of searching using additional information (heuristics) to find out the next suitable step to take [9]
- It uses an evaluation function to determine which neighbour node would be most appropriate to move to
- It uses priority queue data structure for its implementation
- Algorithm steps:
    - Create two empty lists
    - Start from the initial node and add it to the ordered open list
    - Then, the below steps are repeated until the final/goal node (endpoint) is reached
        - If the open list is empty exit the loop and return a False statement which says that the final node cannot be reached
        - Select the top node in the open list and move it to the closed list while keeping track of the parent node
        - If the node removed is the endpoint node return a True statement meaning a path has been found then move the node to the closed list
        - However if it is not the endpoint node then list down all of the current node's neighbouring nodes (children) and add them to the open list
        - Re-order the nodes according to the evaluation function (this is the task of the priority queue data structure)
- Time Complexity: $O(n*\log_2(n))$ where n is the number of nodes

- Example:

open=[S_0]; closed=[ ]
open=[B_1 , A_4]; closed=[S_0]
open=[E_3 , A_4, F_4]; closed=[S_0, B_1]
open=[A_4 , F_4 , $G1_6$, , H_7]; closed=[S_0, B_1, E_3]
open=[F_4, C_5 , $G1_6$, D_6 , H_7]; closed=[S_0 , B_1, E_3, A_4]
open=[C_5 , $G2_5$, $G1_6$, I_6, D_6 , H_7]; closed=[S_0 , B_1, E_3, A_4 , F_4]
open=[$G2_5$, $G1_6$, I_6, D_6 , H_7]; closed=[S_0 , B_1, E_3, A_4 , F_4 , C_5]



SEARCH PATH = [S_0 , B_1, E_3, A_4 , F_4 , C_5 , $G2_5$]

Cost = 1+3+1=5

- Variations: Best first search has variations that include greedy best first search, and A* best first search. The only difference between the two is that the evaluation function “f(n)” only consists of the heuristic function “h(n)” in the greedy version, but consists of both the heuristic function and path function “g(n)” in the A* version. Therefore A* is the more optimal of the two approaches as it also takes into consideration the total distance travelled so far i.e. g(n). Therefore, only A* will be discussed next. Side note: It’s important to note that:

  - g(n): path distance from root node to node “n”

  - h(n): Estimate distance from node “n” to goal node

  - f(n): evaluation function (g(n)+h(n) in case of A*)

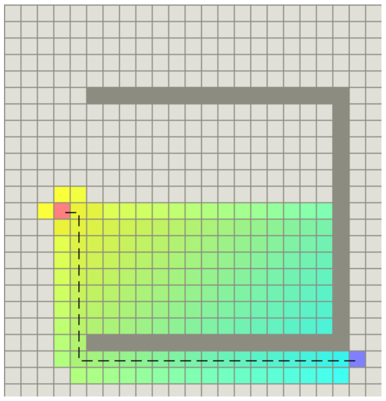
A* Algorithm

- Already discussed in the last part of the best first search section

- In general, the algorithm makes its decisions by taking the f-value into account. The algorithm selects the smallest f-valued cell and moves to that cell. This process continues until the algorithm reaches its goal cell [10]

- The heuristic function can be calculated in many ways. Here, only the Manhattan distance method is considered

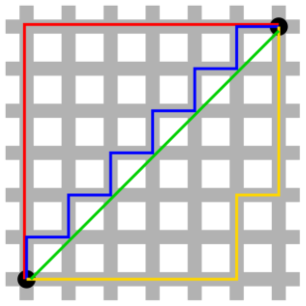
  - Supposing we're in a grid like environment where the red and blue cells are the start and end locations respectively:

    Manhattan distance is then defined as the total number of squares moved horizontally and vertically to reach the target square from the current square. Noting that we ignore diagonal movement and any obstacles that might be in the way [11]. Succinctly:

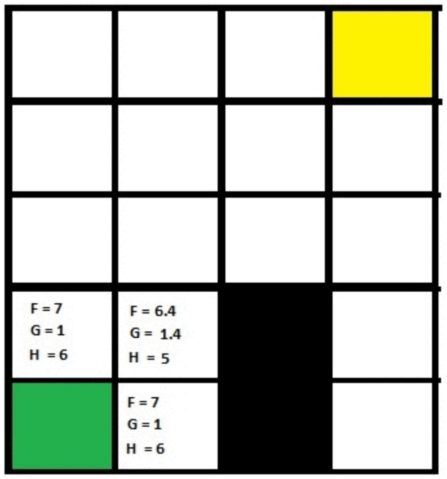
    $$h = |x_{start} - x_{destination}| + |y_{start} - y_{destination}|$$

    Visualisation of different paths:

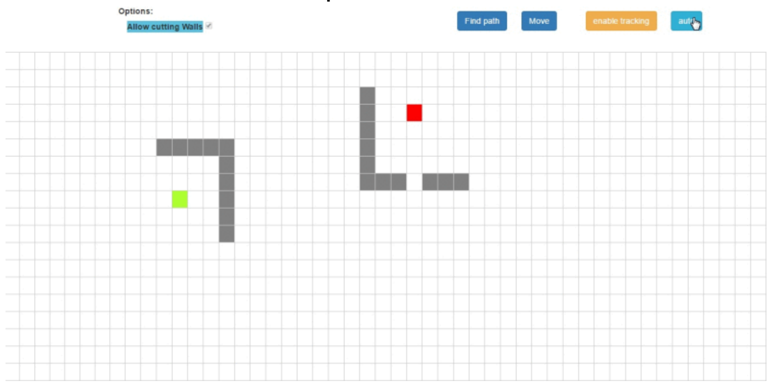
    

- Time complexity: $O(b^d)$ [12], where b = branching factor (in case of orthogonal movement, 4), and d is the depth of the goal node (depends on which staff member requests coffee). However, following the same logic of previous algorithms, we'll consider the time complexity with regards to programming. In that case, the time complexity is O(E), where E = number of edges in the graph [13].

- Example (notice how the next value selected is the one with the least f(n) value):

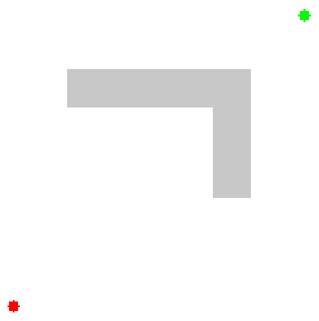


- Another visualised example:



Source: [14]

- A third visualisation to compare it with the visualisation of Dijkstra's algorithm shown below:

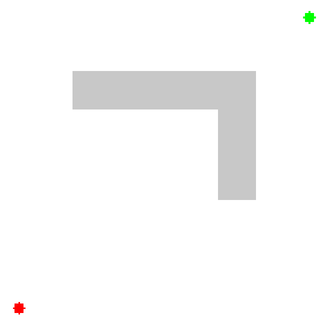


Source: [15]

- Algorithm steps [4]:
    - Add the starting square to the open list.
    - Repeat the following:
        - Look for the lowest F cost square on the open list. We refer to this as the current square.
        - Switch this current square to the closed list.
        - For each square orthogonally adjacent to this current square (we will refer to this as the future square):
            - If it is not walkable or if it is on the closed list, ignore it.
            - If it is not on the open list, add it to the open list. Make the current square the parent of this future square. Record the F, G, and H costs of this future square.
            - If it is on the open list already, check to see if this path to that square is better, using G cost as the measure. If so, change the parent of this future square to the current square, and recalculate the G and F scores of this future square.
    - Save the path. Working backwards from the target square, go from each square to its parent square until you reach the starting square.
        - Stop when you:
            - Add the target square to the closed list, in which case the path has been found or
            - Fail to find the target square, and the open list is empty. In this case, there is no path

Dijkstra’s Algorithm

- It is similar to A* algorithm, but without the heuristic function [16]. In other words, f(n) = g(n) (h(n) = 0)
- Visualisation:



Source: [17]

- Even though this algorithm is also guaranteed to find the shortest path, it exhaustively searches for it (as it has no additional knowledge (i.e. heuristics) to guide it through its search). Therefore its complexity is guaranteed to be larger than A*, and therefore will not be further discussed.

Chosen Strategy

Since informed strategies perform better than the uniformed ones, and since A* takes less computational time than Dijkstra’s algorithm. Therefore, the A* algorithm will be how the “R” navigates through E.

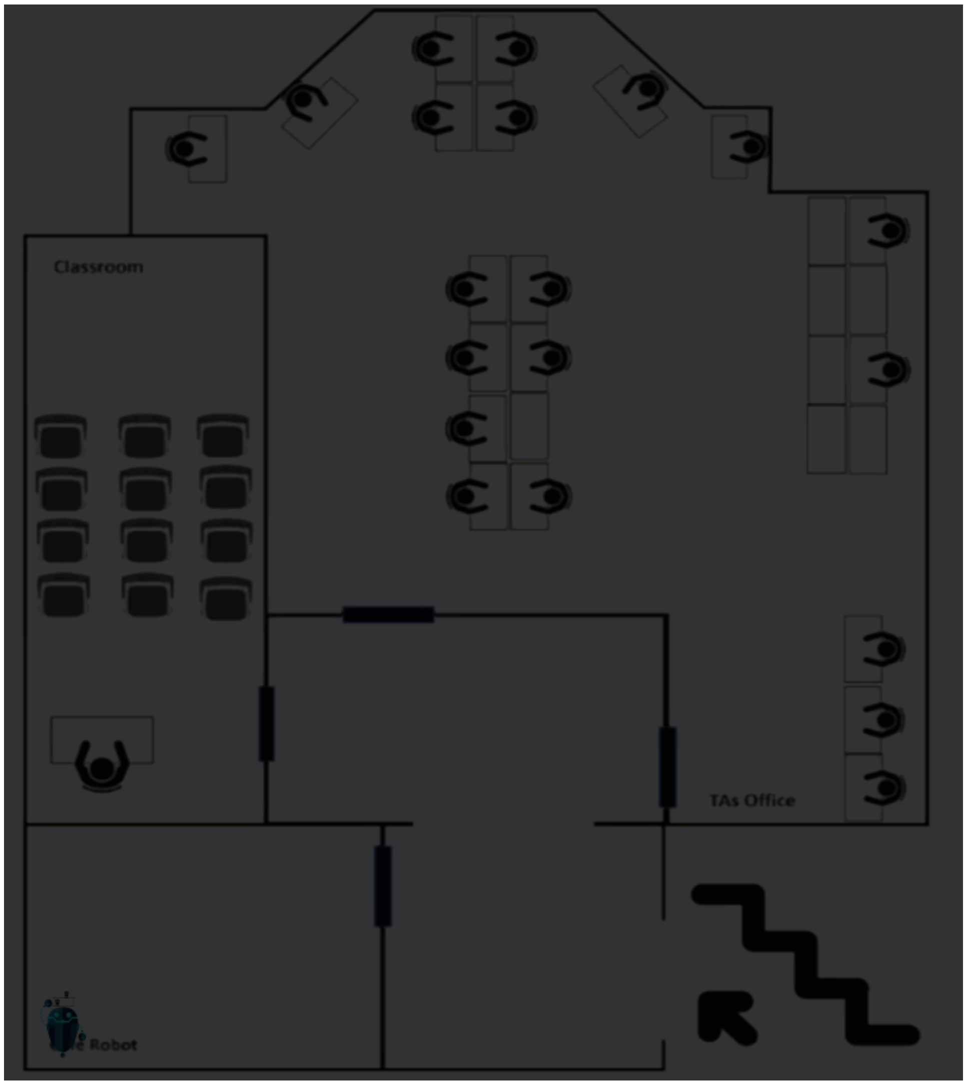
Illustrations of The “Café Robot Problem”

The following section is included in the powerpoint accompanied with this document.
(Note, when opening the powerpoint, it is preferable not to manually skip slides, as the powerpoint automatically proceeds to the next slide. Moreover, a newer version of powerpoint that supports the “morph” transition should be available to properly display the content of the powerpoint)

Scenario 1

DELIVER COFFEE TO DR. GERARD & T.A TOKA

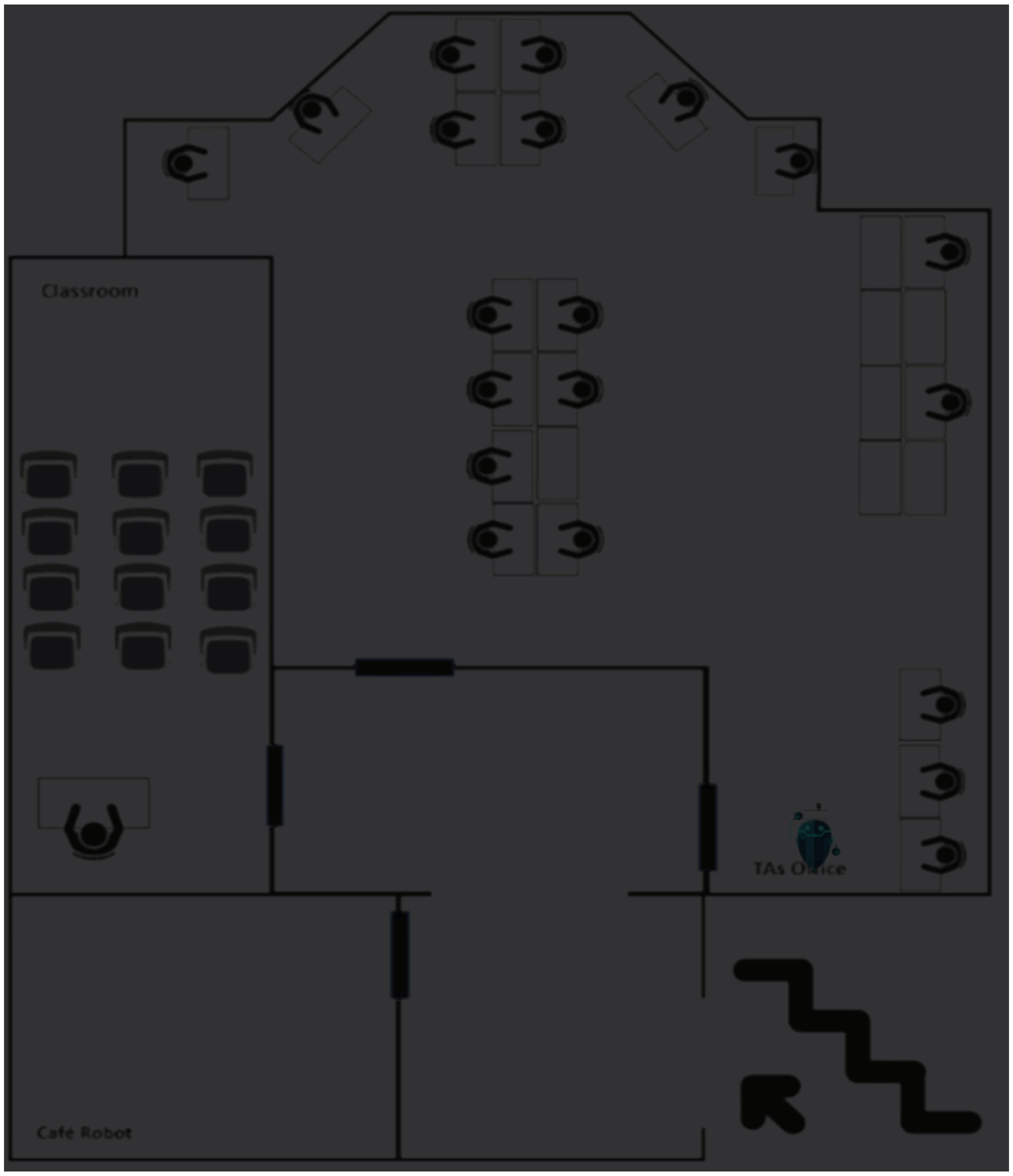
Using A* to make the robot "R" traverse the shortest path



Scenario 2

SCENARIO 2: "R" CAN'T DELIVER COFFEE TO T.A MAHA

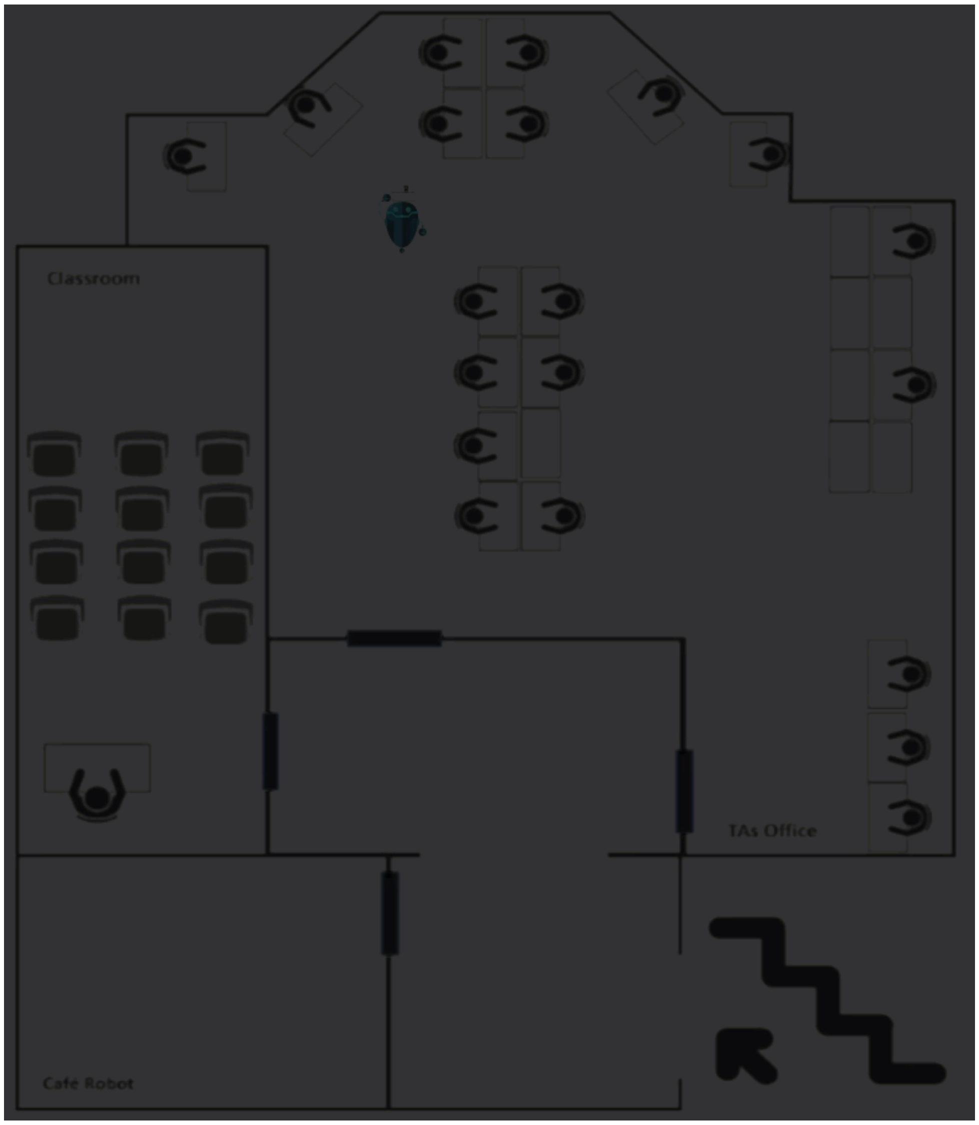
Therefore, for specific staff members, it is allowed for "R" to deliver coffee on the table



Scenario 3

STAFF CONCERNS REGARDING "R" MOVING THROUGH A NARROW SPACE

Therefore, for specific staff members, they will have to get the coffee themselves from "R"



References

[1] J. Oke, “Answer to ‘What word describes something that can move orthogonally and diagonally?,’” *English Language & Usage Stack Exchange*, Feb. 03, 2015. https://english.stackexchange.com/a/225389 (accessed Oct. 27, 2022).

[2] G. M. O. Maloiy, N. C. Heglund, L. M. Prager, G. A. Cavagna, and C. R. Taylor, “Energetic cost of carrying loads: have African women discovered an economic way?,” *Nature*, vol. 319, no. 6055, Art. no. 6055, Feb. 1986, doi: 10.1038/319668a0.

[3] “How do African women balance and carry such heavy loads on their heads?,” *Quora*. https://www.quora.com/How-do-African-women-balance-and-carry-such-heavy-loads-on-their-heads (accessed Oct. 25, 2022).

[4] The University of Rhode Island, “A* Pathfinding.” https://homepage.cs.uri.edu/faculty/hamel/courses/2010/spring2010/csc481/lecture-notes/ln015.pdf (accessed Oct. 28, 2022).

[5] “What is Breadth First Search?,” *Educative: Interactive Courses for Software Developers*. https://www.educative.io/answers/what-is-breadth-first-search (accessed Oct. 27, 2022).

[6] “Uninformed Search Algorithms in AI | Search Algorithms in AI.” https://www.analyticsvidhya.com/blog/2021/02/uninformed-search-algorithms-in-ai/ (accessed Oct. 27, 2022).

[7] “Depth First Search Tutorials & Notes | Algorithms | HackerEarth.” https://www.hackerearth.com/practice/algorithms/graphs/depth-first-search/tutorial/ (accessed Oct. 27, 2022).

[8] “What is Depth First Search?,” *Educative: Interactive Courses for Software Developers*. https://www.educative.io/answers/what-is-depth-first-search (accessed Oct. 27, 2022).

[9] “Best First Search algorithm,” *OpenGenus IQ: Computing Expertise & Legacy*, Nov. 09, 2021. https://iq.opengenus.org/best-first-search/ (accessed Oct. 27, 2022).

[10] “What is the A* algorithm?,” *Educative: Interactive Courses for Software Developers*. https://www.educative.io/answers/what-is-the-a-star-algorithm (accessed Oct. 27, 2022).

[11] “A* Search | Brilliant Math & Science Wiki.” https://brilliant.org/wiki/a-star-search/ (accessed Oct. 27, 2022).

[12] D.W, “Answer to ‘A* graph search time-complexity,’” *Computer Science Stack Exchange*, Apr. 19, 2016. https://cs.stackexchange.com/a/56184 (accessed Oct. 27, 2022).

[13] “A* Search Algorithm,” *GeeksforGeeks*, Jun. 16, 2016. https://www.geeksforgeeks.org/a-search-algorithm/ (accessed Nov. 02, 2022).

[14] vittin, “A-Star.” Sep. 26, 2022. Accessed: Oct. 27, 2022. [Online]. Available: https://github.com/vittin/A-Star

[15] “A* search algorithm,” *Wikipedia*. Oct. 02, 2022. Accessed: Oct. 28, 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=A*_search_algorithm&oldid=1113605932

[16] C. Kalansuriya, “Answer to ‘Difference and advantages between dijkstra & A star,’” *Stack Overflow*, Apr. 23, 2017. https://stackoverflow.com/a/43567824 (accessed Oct. 25, 2022).

[17] “Dijkstra’s algorithm,” *Wikipedia*. Oct. 22, 2022. Accessed: Oct. 25, 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Dijkstra%27s_algorithm&oldid=1117533081