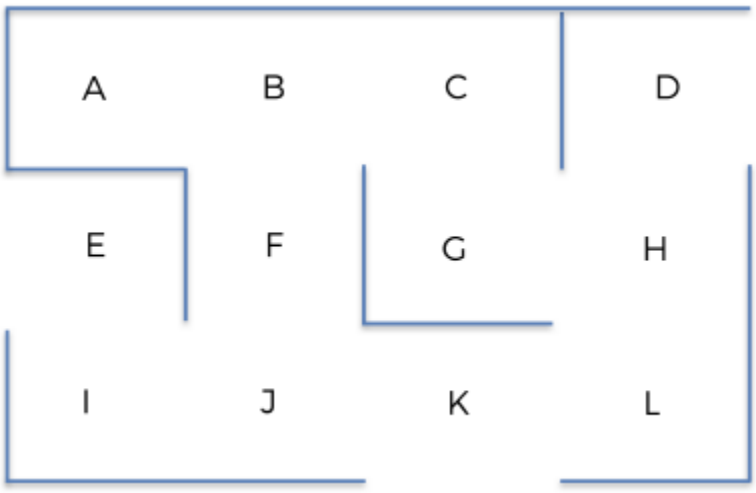
Using Q-learning to optimize warehouse flows:

This is the layout based on which the prototype has been developed.



Environment to define:

- Define the states
- Define the actions
- Define the rewards

States:

For states we will encode the states with index number

A->0

B->1

C->2

D->3

E->4

F->5

G->6

H->7

I->8

J->9

K->10

L->11

Defining the actions:

Actions are being defined by using the current state the agent is in.

So, we will deploy the actions based on the state as all actions cannot be played at every state

So, as states actions are also an array actions=[0,1,2,3,4,5,6,7,8,9,10,11]

Defining the rewards:

So, the last thing is we must build a reward system.

So, it will take the state and action as input and compute the reward.

This is made by using an adjacency matrix.

So, a zero reward to actions it cannot play and 1 reward to actions it can play.

And then the place we want to reach will have the highest of all rewards.

So that the training converges to the final highest reward i.e., our destination.

Algorithm brief overview:

1.We select a random state S_t from our 12 possible states:

Random(0,1,2,3,4,5,6,7,8,9,10,11)

2. we will play a random action that can lead to a next possible state such that the reward function is >0.

3.We reach the next state and we get the reward.

4.We then compute the temporal difference:

Which is given by:

$$TD_t(s_t, a_t) = R(s_t, a_t) + \gamma \max_a (Q(s_{t+1}, a)) - Q(s_t, a_t)$$

5. We update the Q-value by applying the bellman equation

$$Q_t(s_t, a_t) = Q_{t-1}(s_t, a_t) + \alpha TD_t(s_t, a_t)$$

Code:

```
#importing the libraries

import numpy as np



#Setting the parameters for gamma and alpha for the Q-learning
gamma = 0.75
alpha = 0.9

#Part-1 DEFINING the environment

#defing the states

location_to_state = {"A": 0,
                     "B" : 1,
                     "C" : 2,
                     "D" : 3,
                     "E" : 4,
                     "F" : 5,
                     "G" : 6,
                     "H" : 7,
                     "I" : 8,
                     "J" : 9,
```

```
        "K" : 10,
        "L" : 11}

#defining actions
actions = [0,1,2,3,4,5,6,7,8,9,10,11]

#defining the rewards
#it can be considered as a adjacency matrix with rewards.
R = np.array([[0,1,0,0,0,0,0,0,0,0,0,0],
        [1,0,1,0,0,1,0,0,0,0,0,0],
        [0,1,0,0,0,0,1,0,0,0,0,0],
        [0,0,0,0,0,0,0,1,0,0,0,0],
        [0,0,0,0,0,0,0,0,1,0,0,0],
        [0,1,0,0,0,0,0,0,0,1,0,0],
        [0,0,1,0,0,0,1,1,0,0,0,0],
        [0,0,0,1,0,0,1,0,0,0,0,1],
        [0,0,0,0,1,0,0,0,0,1,0,0],
        [0,0,0,0,0,1,0,0,1,0,1,0],
        [0,0,0,0,0,0,0,0,0,1,0,1],
        [0,0,0,0,0,0,0,1,0,0,1,0]])

# AI-solution

#making a mapping from state to location
```

```
state_to_location = {state : location for location , state in location_to_state.items()}
#makng a function to return the optimal route


def route(starting_location,ending_location):
    R_new = np.copy(R)
    ending_state = location_to_state[ending_location]
    R_new[ending_state,ending_state] = 1000
    Q = np.array(np.zeros([12,12]))
    for i in range(1000):
        current_state = np.random.randint(0,12)
        playable_actions = []
        for j in range(12):
            if R_new[current_state, j] > 0:
                playable_actions.append(j)
        next_state = np.random.choice(playable_actions)
        TD = R_new[current_state, next_state] + gamma * Q[next_state, np.argmax(Q[next_state,])] -
Q[current_state, next_state]
        Q[current_state, next_state] += alpha * TD
    route = [starting_location]
    next_location = starting_location
    while(next_location != ending_location):
        starting_state = location_to_state[starting_location]
        next_state = np.argmax(Q[starting_state,])
        next_location = state_to_location[next_state]
        route.append(next_location)
        starting_location = next_location
    return route
#going into production
```

```
def best_route(starting_location,intermediary_location,ending_location):
    return route(starting_location,intermediary_location) +
route(intermediary_location,ending_location)[1:]
```

```
#printing the final route
print('Route:')
best_route('E' ,'F', 'G')
```

Output:

```
    ....
    ...: #printing the final route
    ...: print('Route:')
    ...: best_route('E' ,'F', 'G')
Route:
Out[1]: ['E', 'I', 'J', 'F', 'B', 'C', 'G']
```

So, when we call this the agent will find the best route to move to from ‘E’ to ‘G’ through ‘F’.

Refer the best_route() function to understand implementation and then refer to the graph picture in the beginning.

Now, the same route of ‘E’ to ‘G’ through ‘K’.

```
    ...: print('Route:')
    ...: best_route('E' ,'K', 'G')
Route:
Out[2]: ['E', 'I', 'J', 'K', 'L', 'H', 'G']
```

Now from A to D through G:

```
    ...: print('Route:')
    ...: best_route('A' ,'G', 'D')
Route:
Out[3]: ['A', 'B', 'C', 'G', 'H', 'D']
```

Business application implications:

Any warehouse based on the robot or vehicle dimensions can be modelled as a connected graph.

The simple implementation of the model based on only python can be easy to implement this in any robot which can be easily prototyped using Raspberry Pi.

Automation can be extensive here as it can be also created as an web API which can use GET queries from the user and navigate the agent through the warehouse reducing human intervention.