Solutions for HW1

Yunhai Han
Department of Mechanical and Aerospace Engineering
University of California, San Diego
y8han@eng.ucsd.edu

April 16, 2020

Contents

1 Written part

1.1 Problem 1

1.1.1 Problem formulation

Construct a worst-case example for best-first search in a 2D world with obstacles, where a greedy move will result in large departure from the goal.

1.1.2 solution

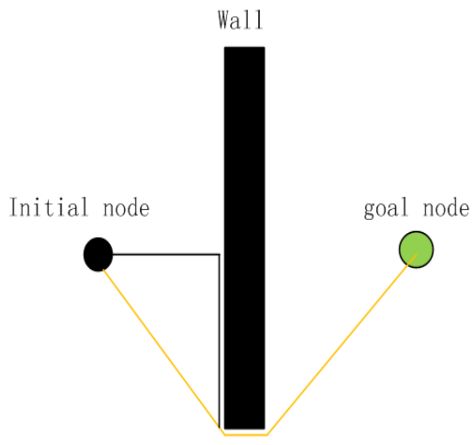


Figure 1: The worst-case example

From Fig 1, I illustrate the possible worst case for best-search in 2 2D world with a wall between the initial and goal nodes. At early stage, the agent would go straight towards the goal without the knowledge of the existence of the wall. However, after a few steps, when it detects the wall, it would move along the boundary of the wall. And this causes large departure from the goal.

The yellow lines show the optimal path, which is not available to obtain using best-first search. The black lines show a possible path for best-first search.

1.2 Problem 2

1.2.1 Problem formulation

The Dijkstra's algorithm was presented as a kind of forward search. Write Dijkstra's Thackward search algorithm version. Using an induction argument, prove that backward Dijkstra's algorithm provides optimal paths from the goal to the initial state.

1.2.2 solution

A general backward search template is(by LaValle's *Planning Algorithm*):

```
BACKWARD_SEARCH
1   $Q.Insert(x_G)$ and mark $x_G$ as visited
2   while $Q$ not empty do
3       $x' \leftarrow Q.GetFirst()$
4       if $x = x_I$
5           return SUCCESS
6       forall $u^{-1} \in U^{-1}(x)$
7           $x \leftarrow f^{-1}(x', u^{-1})$
8           if $x$ not visited
9               Mark $x$ as visited
10              $Q.Insert(x)$
11          else
12              Resolve duplicate $x$
13  return FAILURE
```

Figure 2: A general backward search template

As the same for the forward search version, we define a cost of sequence of actions applied in succession:

$$\ell(\pi) = \sum_i \ell\left(e_i\right), \quad \pi^{-1} = u_1^{-1} u_2^{-1} \dots u_n^{-1} \tag{1}$$

Then, define a cost-to-come from x_G to x as:

$$C^{\star}(x) = \min_{\pi^{-1}} \left\{ \ell(\pi^{-1}) | \pi^{-1} \text{ connects } x_G \text{ to } x \right\} \tag{2}$$

Note: This cost is actually computed incrementally as $C(x) = \min_{\text{known}} x\ell(\pi^{-1})$ and updated as we find more π^{-1}. Then, $Q.getFirst()$ in the Fig 2 picks an element with the lowest cost.

Also, back points need to be updated for states marked as visited.

- Suppose that $x \leftarrow f^{-1}(x', u^{-1})$ was marked as visited (step 7)
- Then, in step 12, do the following: If $C(x') + \ell(x', u^{-1}) < C\left(x\right)$, then

  1.update parent $(x) = x'$

  2.reorder x in Q according to the new value $C\left(x\right)$

The next step is to prove hte optimality of this algorithm. Here, if I could the optimality of the forward version, the optimality of the backward version is satisfied automatically based on the fact that if I switch the initial node and the goal node, the backward version could be turned into the forward one(because the action mapping U is a one-to-one mapping).

Hence, the left work is to prove the optimality of forward version. The cost function in D algorithm is denoted as $d(x)$, the length of edges is denoted as l and the shortest path distance from x_I to x is denoted as $\lambda(x)$. Here, we need to prove $d(x) = \lambda(x)$ for every node x at the end of the algorithm, showing the algorithm could find optimal paths from x_I to any nodes x. The proof by induction is shown as following:

- At the first step, only the initial node is contained in the queue Q, and $d(x_I) = \lambda(x_I) = 0$.
- Inductive hypothesis: Let u be the last node picked by this algorithm. Our Inductive hypothesis is: for each node x which has been picked $d(x) = \delta(x)$. I use R to denote the set of these nodes.

- By the Inductive hypothesis, for any nodes except u, we have the correct cost function $d(x) = \lambda(x)$. And we need to prove $d(u) = \lambda(u)$.

  Suppose for a contradiction that the shortest path from x_I to u is Q and has length

  $$l(Q) < d(u)$$

  Q starts in R and at some node x leaves R (to get to u which is not in R). Let xu be the first edge along Q that leaves R. Let Q_x be the x_I to x subpath of Q. Clearly, we have the following expression:

  $$l\left(Q_x\right) + l(xu) = l(Q)$$

  since $d(x)$ is the length of the shortest x_I to x path by the I.H., $d(x) \leq l\left(Q_x\right)$, giving us

  $$d(x) + l(xu) \leq l\left(Q\right)$$

  since u is adjacent to x, $d(u)$ must have been updated by the algorithm, so

  $$d(u) \leq d(x) + l(xu)$$

  Combining these inequalities gives us the contradiction that $d(x) < d(x)$. Therefore, no such shorter path Q must exist and so $d(u) = \delta(u)$.

2 programming part

(Programming problem) In this exercise, you will implement and demonstrate the performance of various deterministic search algorithms in different maze environments. To do this, please download the folder "search" from canvas. In it you will find several files. The file mazemods.py contains main functions to plot a maze, check for collisions, calculate costs, and draw paths in the maze environment. The files smallMaze.py, mediumMaze.py, and bigMaze.py are mazes of several sizes. A maze is a 2D grid world of $n \times m$ states given by $x = (a, b)$, with $0 \leq a \leq n-1, 0 \leq b \leq m-1$, and a list of forbidden cells in it described by O. Transitions at each state are given by legal controls $u \in U = \{(1,0),(-1,0),(0,1),(0,-1)\}$, which do not incur into collision, and which can be obtained by simple addition $x+u = x'$. The illegal or forbidden transitions are those resulting into a wall collision, and this is evaluated by the function collisioncheck inside mazemods.

2.1 a

2.1.1 Problem formulation

Define a function depthFirstSearch(x_I, x_G, n, m, O) implementing a DFS algorithm on a maze. The inputs of the function are an initial state, x_I, a goal state, x_G, and problem environment parameters, n, m, O. The outputs of the function should be a list of legal control actions following DFS, the cost of this sequence of actions, the number of nodes explored in the search, and the plot of a final path from start to goal. To compute the cost of the action sequence use the function getcostofActions(x_I, x_G, n, m, O) in mazemods.py. Note that any illegal action will result in a very large cost. Is the exploration order what you would have expected? Is this a least cost solution? If not, think about what depth-first search is doing wrong.

2.1.2 Solution

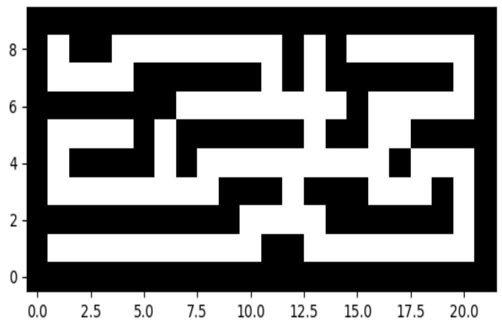


Figure 3: The configuration of small maze

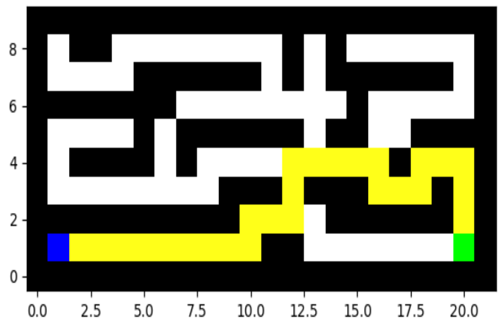
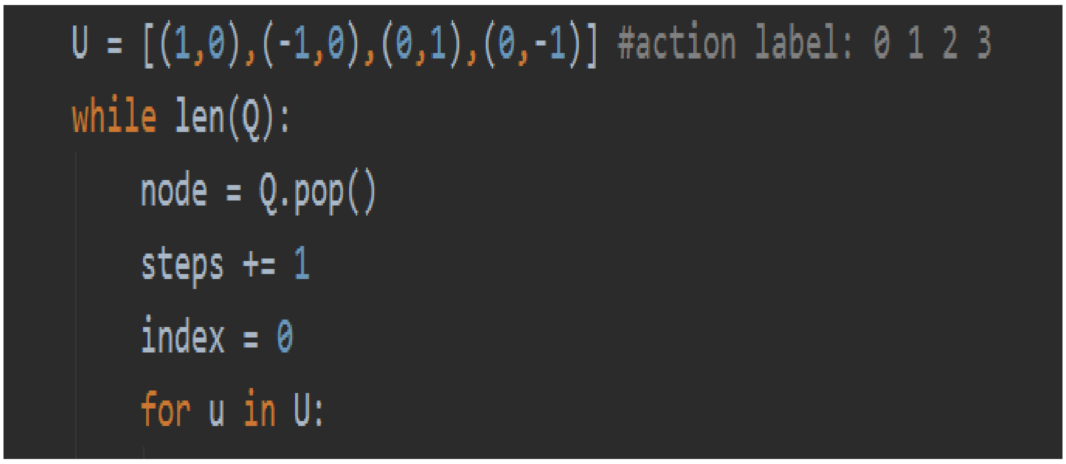


Figure 4: The extracted path($x_I = (1,1), x_G = (20,1)$)

Obviously, the generated path is not what we have expected because these is definitely a shorter path. This means DFS algorithms could find a feasible path but not an optimal one. The reason is simple: it prefers to exploring along one direction and stops until the end.

From Fig 4, the exploring direction goes up at the node(12,2) and indeed we expect it to keep forward. This could be explained by the selected action order in my codes when there are several feasible active neighbour nodes.



```
U = [(1,0),(-1,0),(0,1),(0,-1)] #action label: 0 1 2 3
while len(Q):
    node = Q.pop()
    steps += 1
    index = 0
    for u in U:
```

Figure 5: Selected action order shown in U

From the above figure, you could see the action(1,0) is ahead of the action(0,1), which means the neighbour node(13,2) could be added into the queue$Q$ earlier then the node(12,3). We know DFS use the LIFO(Last-in-First-out) criterion to select the node in the next iteration, so this algorithm tends to explore the areas along the vertical directions($u = [(1,0),(-1,0),(0,1),(0,-1)]$). Also, in such conditions, it would cost in total 71 steps until it finds the goal node.

If I switch the sequence of (1,0) and (0,-1) in U, it would still give me the same path as shown in Fig 4, but it would cost in total 85 steps. The difference comes from the fact that the algorithm tends to explore the space upward instead of downward. Hence, in the right side of the map, it would explore the right-upper region first and then explore the right-bottom region with goal node inside, which would cost 14 more steps.

For comparison, I reverse U and implement this algorithm again.

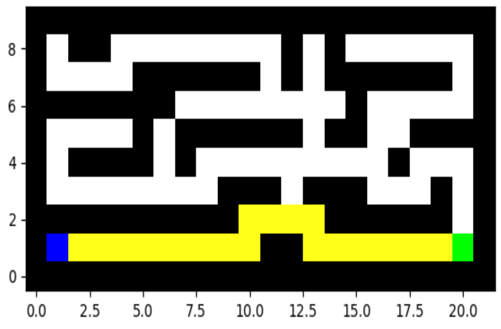
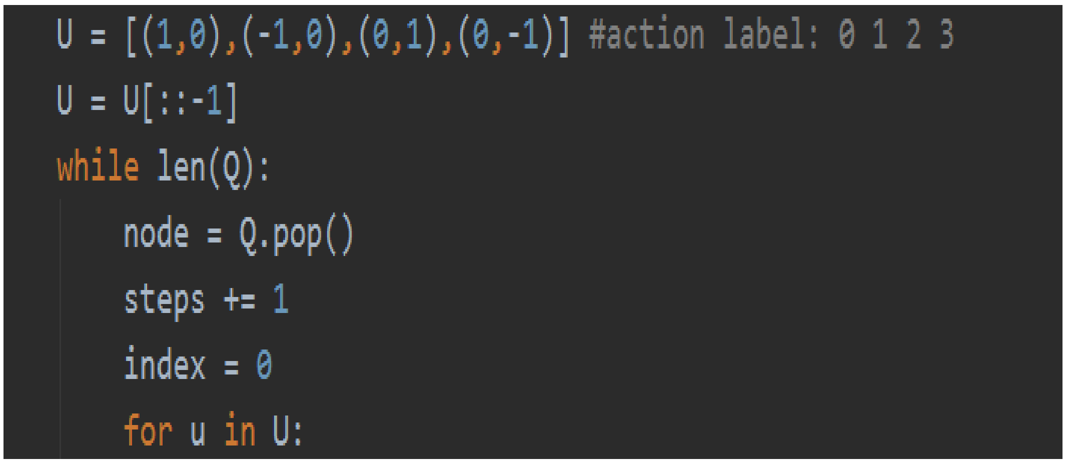


Figure 6: The extracted path($x_I = (1,1), x_G = (20,1)$)



```
U = [(1,0),(-1,0),(0,1),(0,-1)] #action label: 0 1 2 3
U = U[::-1]
while len(Q):
    node = Q.pop()
    steps += 1
    index = 0
    for u in U:
```

Figure 7: Selected action order shown in U

From Fig 6, you could see this time, the extracted path is really what we expect. Because, after I reverse U(by Python command $U = U[:: -1]$), the algorithm prefers to explore the space along the horizontal direction and it would only cost 21 steps to find the goal node with respect to the certain map configuration.

Here, I also use other map configuration and tune the coordinates of the initial and goal nodes. I put some results by implementing DFS algorithm.

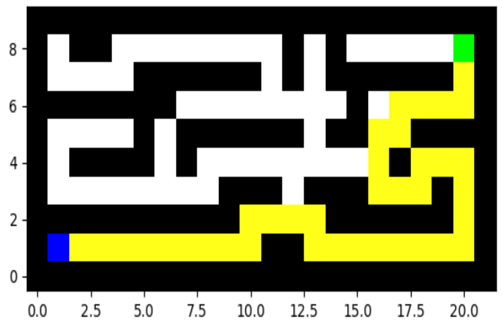


Figure 8: The extracted path

The conditions are:Map=smallmaze, $x_I = (1,1), x_G = (20,8), U = (1,0),(-1,0),(0,1),(0,-1), U = U[:: -1]$.

No such path found. The conditions are:Map=smallmaze, $x_I = (1,1), x_G = (14,5)$, $U = (1,0),(-1,0),(0,1),(0,-1)$. The node x_G is inside the obstacles.

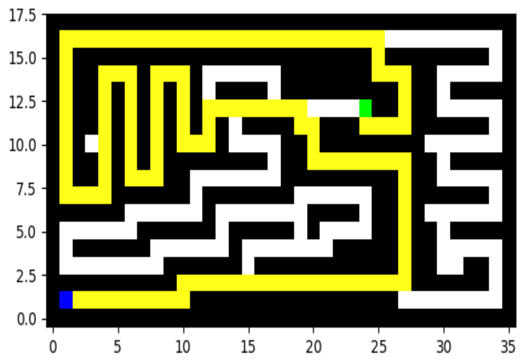


Figure 9: The extracted path

The conditions are:Map=mediummaze, $x_I = (1,1), x_G = (24,12), U = (1,0),(-1,0),(0,1),(0,-1), U = U[:: -1]$.

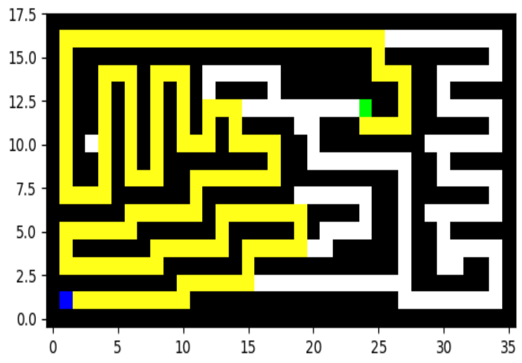


Figure 10: The extracted path

The conditions are:Map=mediummaze, $x_I = (1,1), x_G = (24,12), U = (1,0),(-1,0),(0,1),(0,-1)$.

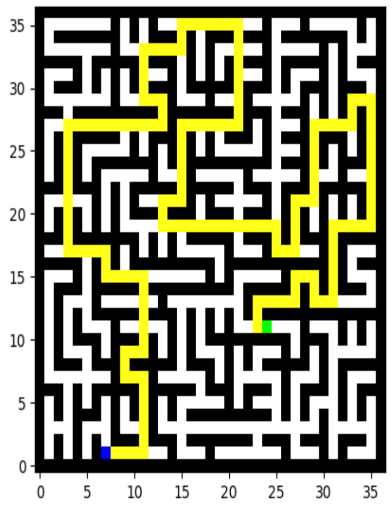


Figure 11: The extracted path

The conditions are:Map=bigmaze, $x_I = (7, 1), x_G = (24, 11), U = (1, 0), (-1, 0), (0, 1), (0, -1)$.

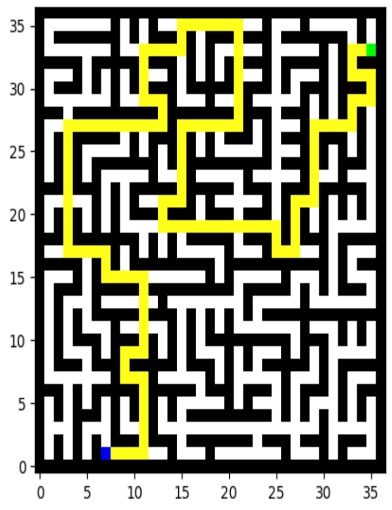


Figure 12: The extracted path

The conditions are:Map=bigmaze, $x_I = (7, 1), x_G = (35, 33), U = (1, 0), (-1, 0), (0, 1), (0, -1), U = U[:: -1]$.

2.2 b

2.2.1 Problem formulation

Define a function breadthFirstSearch (x_I, x_G, n, m, O) implementing a BFS algorithm on a maze. The inputs of the function are an initial state, x_I, a goal state, x_G, and problem environment parameters, n, m, O (maze size and obstacles). The outputs of the function should be a list of legal control actions following BFS, the cost of this sequence of actions, the number of nodes explored in the search, and a plot of the final path from start to goal. To compute the cost of the action sequence use the function getcostofactions(x_I, x_G, n, m, O) in mazemods.py.

2.2.2 Solution

The algorithm BFS is similar to DFS except to one difference:BFS uses FIFO(First-in-First-out) instead of LIFO. In other sense, this algorithm tends to explore the whole feasible space along all direction starting from the initial nodes and that's why it is named as **breadthFirst**. With this property, the extracted path is the shortest one(optimal). For programming, we don't have to change a lot from the codes for DFS. We only have to change how to pick node from queue$Q$ and this could be done easily by python command$node = Q.pop(0)$.

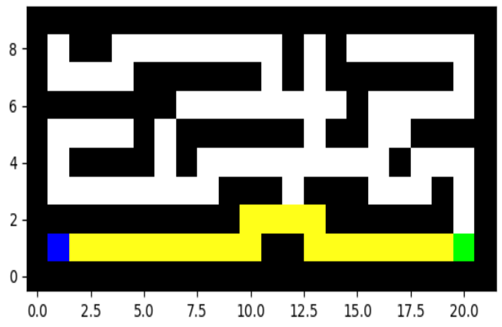


Figure 13: The extracted path

The conditions are:Map=smallmaze, $x_I = (1, 1), x_G = (20, 1)$. Because we are using BFS, the action sequence in U has little effect on the total steps and final path. And no matter what sequence I set in U, the algorithm could always give me the shortest path.

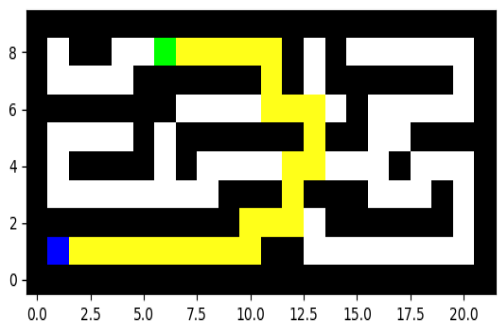


Figure 14: The extracted path

The conditions are:Map=smallmaze, $x_I = (1, 1), x_G = (6, 8)$.

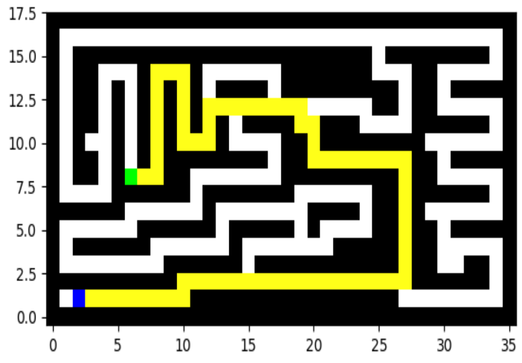


Figure 15: The extracted path

The conditions are:Map=mediummaze, $x_I = (2, 1), x_G = (6, 8)$.

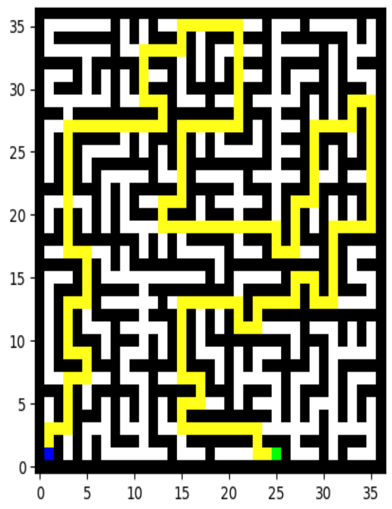


Figure 16: The extracted path

The conditions are:Map=bigmaze, $x_I = (1,1), x_G = (25,1)$.

The choice between DFS and BFS algorithms depend on the prior knowledge of the structure of the optimal path. If the structure of the optimal path could be inferred from the known path of the map and the coordinates of the initial and goal nodes, we may have the chance to switch the action sequence in U to accelerate the searching process using DFS. However, in general cases, the p the erformances of BFS are better and the worse performances of DFS is much worse than BFS. Besides, there is no guarantee that the feasible path obtained from DFS is optimal.

2.3 c

2.3.1 Problem formulation

While BFS will find a fewest-actions path to the goal, we might want to find paths that are "best" in other senses. Implement a Di jkstraSearch(x_I, x_G, n, m, O, cost =westcost) implementing Dijkstra's algorithm over a maze. The inputs of this function are x_I, x_G, n, m, O as before as well as a cost function. The outputs should be a list of legal control actions following Dijkstra's, the cost of the sequence of actions, the number of nodes explored in the search, and a plot of the final path from start to goal. Provide example plots and results based on the cost function that favors that the agent stays West; $\ell((a,b),u) = \ell(a',b') = a'^2$, and the cost function that favors the agent staying East $\ell((a,b),u) = \ell(a',b') = (x_{\max} - a')^2$, where $x_{\max}$ is the furthest east coordinate of the environment. These functions can be found in mazemods.py.

2.3.2 Solution

The following results are obtained with the conditions set as:Map=mediummaze, $x_I = (1,1), x_G = (34,16)$.

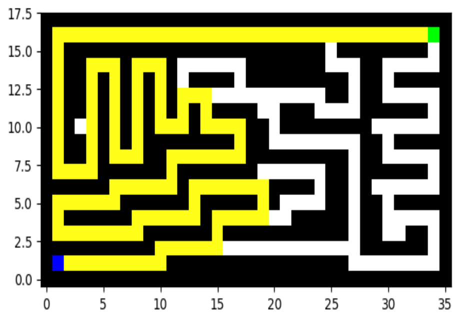


Figure 17: The extracted path(West)

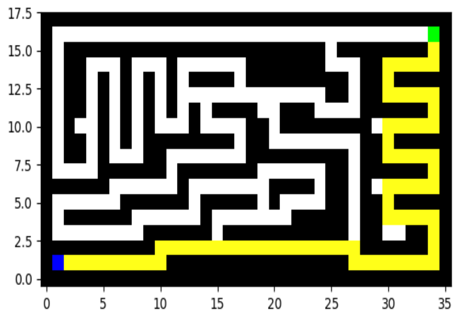


Figure 18: The extracted path(East)

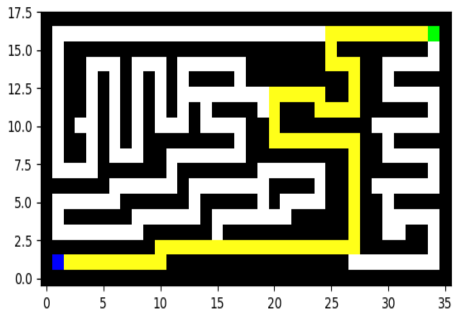
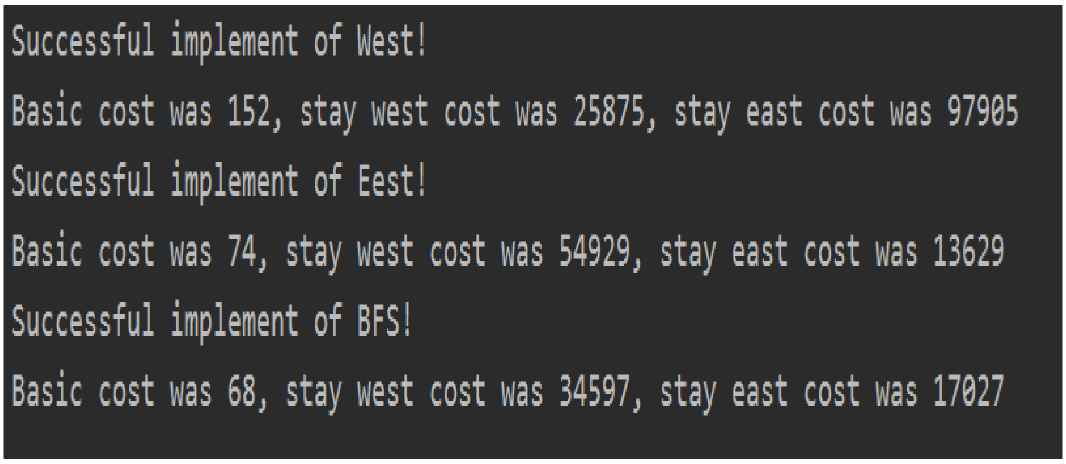


Figure 19: The extracted path(BFS)



```
Successful implement of West!
Basic cost was 152, stay west cost was 25875, stay east cost was 97905
Successful implement of Eest!
Basic cost was 74, stay west cost was 54929, stay east cost was 13629
Successful implement of BFS!
Basic cost was 68, stay west cost was 34597, stay east cost was 17027
```

Figure 20: Programming outputs

From the above figures, you could see the difference for the results obtained from different algorithms. More specifically, from Fig 20, the total west stay west cost for Dijkstra algorithm with function that favors that agent stays west are much smaller than the one with east function and vice versa. For BFS algorithm, it dosen't show any priority for staying east or west, so the difference between east cost and west cost is smaller. And in this case, the final goal is on the right-upper region, so that's why the stay east cost is smaller.

The following results are obtained with the conditions set as:Map=bigmaze, $x_I = (1, 1), x_G = (35, 1)$.

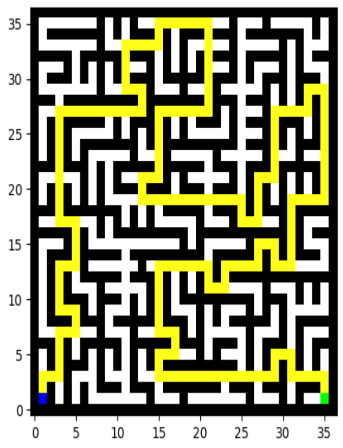


Figure 21: The extracted path(West)

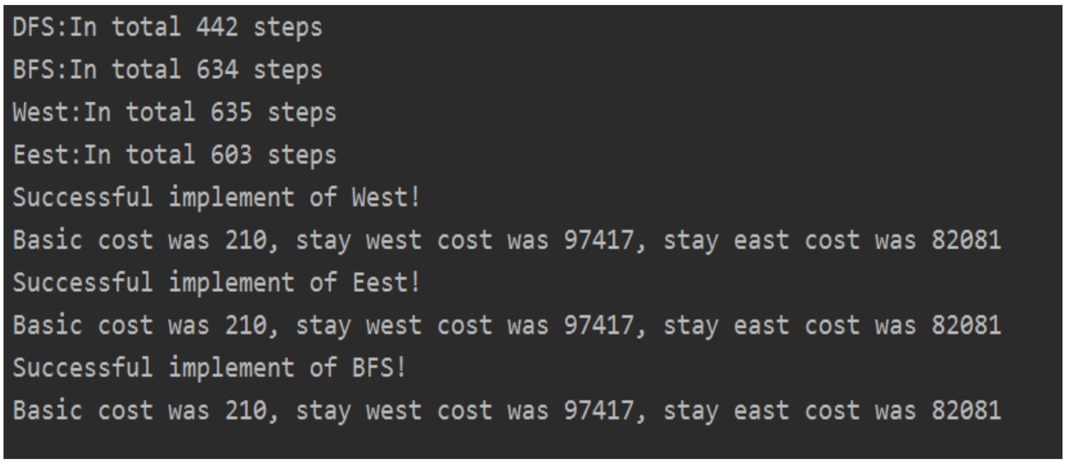

```
DFS:In total 442 steps
BFS:In total 634 steps
West:In total 635 steps
Eest:In total 603 steps
Successful implement of West!
Basic cost was 210, stay west cost was 97417, stay east cost was 82081
Successful implement of Eest!
Basic cost was 210, stay west cost was 97417, stay east cost was 82081
Successful implement of BFS!
Basic cost was 210, stay west cost was 97417, stay east cost was 82081
```

Figure 22: Programming outputs

In this case, all the plots are the same, which means there is only one feasible path and it is the optimal one. From Fig 22, you could see the total steps for each algorithm are not the same.

2.4 d

2.4.1 Problem formulation

A^* implementation. A^* expands Dijkstra's algorithm by making use of a heuristic function as an argument. Heuristics take two arguments: a state in the search problem (the main argument), and the goal state in the problem. Implement a function astarSearch $(x_I, x_G, n, m, O,$ heuristic=givenHeuris that implements the A^* search. To do this, use the regular cummulative cost function getcostofActions. You also have to define two heuristic functions to replace givenHeuristic as follows:

- Using $|a' - a| + |b' - b|$ as a heuristic (manhattanHeuristic)
- Using $\sqrt{(a' - a)^2 + (b' - b)^2}$ as a heuristic (euclideanHeuristik)

As an example, the trivial null Heuristic is given for you in search.py. Evaluate A^* on various grid-based problems of your choice. Compare the performance for the two different cases: Which heuristic is superior? Explain your answer.

2.4.2 solution

For A star algorithm, the only thing I need to change in codes are the cost-to-go functions. Here the cost-to-come function is simple:we assume the length of each edge is 1. The other parts of this algorithm are the same as in the previous section.

The following results are obtained with the conditions set as:Map=smallmaze, $x_I = (1, 1), x_G = (20, 8)$.

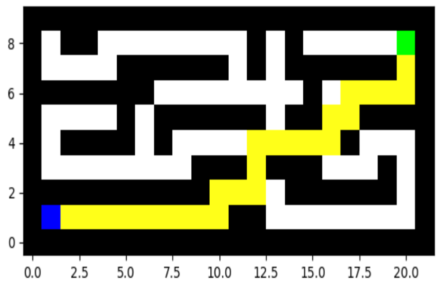
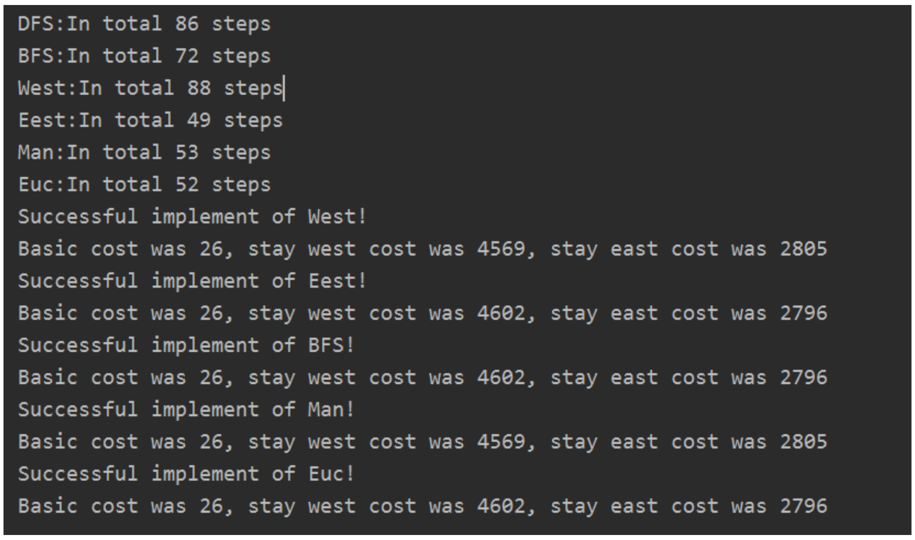


Figure 23: The extracted path



```
DFS:In total 86 steps
BFS:In total 72 steps
West:In total 88 steps
Eest:In total 49 steps
Man:In total 53 steps
Euc:In total 52 steps
Successful implement of West!
Basic cost was 26, stay west cost was 4569, stay east cost was 2805
Successful implement of Eest!
Basic cost was 26, stay west cost was 4602, stay east cost was 2796
Successful implement of BFS!
Basic cost was 26, stay west cost was 4602, stay east cost was 2796
Successful implement of Man!
Basic cost was 26, stay west cost was 4569, stay east cost was 2805
Successful implement of Euc!
Basic cost was 26, stay west cost was 4602, stay east cost was 2796
```

Figure 24: Programming outputs

From the above figures, A* algorithm with both heuristic function could successfully find the optimal path as soon as possible. It takes 53 and 52 steps for A* algorithm with *manhattanHeuristic* and *euclideanHeuristic* functions, respectively. In this case, since the goal node is in the right-upper region, Dijskra algorithm with stay east cost function could also find the optimal path soon(51 steps).

The following results are obtained with the conditions set as:Map=mediummaze, $x_I = (1, 1), x_G = (25, 14)$.

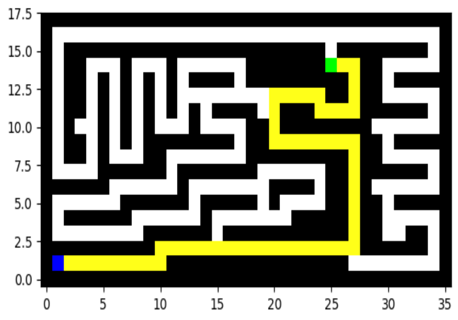


Figure 25: The extracted path

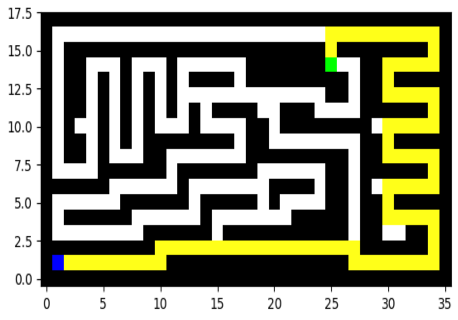
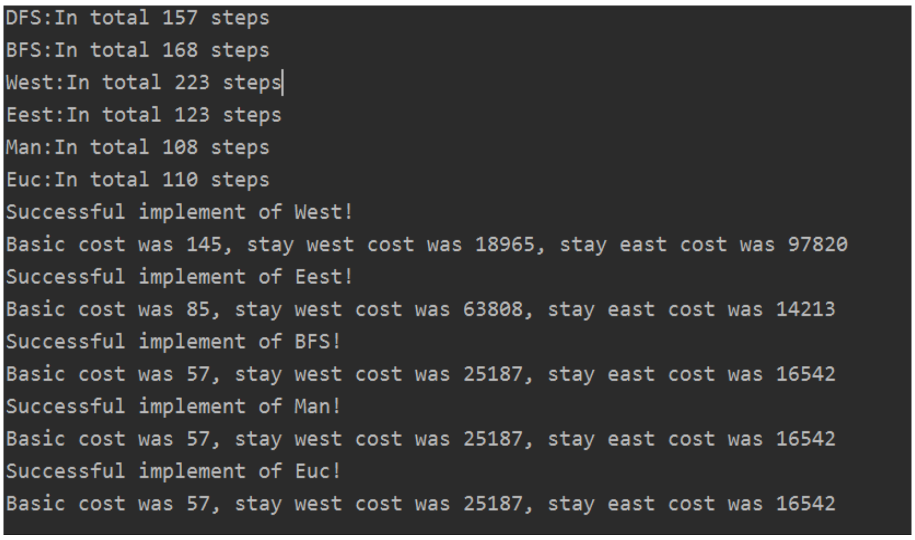


Figure 26: The extracted path



```
DFS:In total 157 steps
BFS:In total 168 steps
West:In total 223 steps
Eest:In total 123 steps
Man:In total 108 steps
Euc:In total 110 steps
Successful implement of West!
Basic cost was 145, stay west cost was 18965, stay east cost was 97820
Successful implement of Eest!
Basic cost was 85, stay west cost was 63808, stay east cost was 14213
Successful implement of BFS!
Basic cost was 57, stay west cost was 25187, stay east cost was 16542
Successful implement of Man!
Basic cost was 57, stay west cost was 25187, stay east cost was 16542
Successful implement of Euc!
Basic cost was 57, stay west cost was 25187, stay east cost was 16542
```

Figure 27: Programming outputs

From the Fig 25, the A* algorithm could find the optimal path quickly(108steps and 110steps, respectively). Here, because the path are the same path as we obtain using BFS, so I could safely tell it is the optimal path.

For comparison, Dijskra algorithm with stay east cost function could also find a feasible path(shown in Fig 26) within 123 steps. However, this path is not optimal.

The following results are obtained with the conditions set as:Map=bigmaze, $x_I = (1, 1), x_G = (20, 13)$.

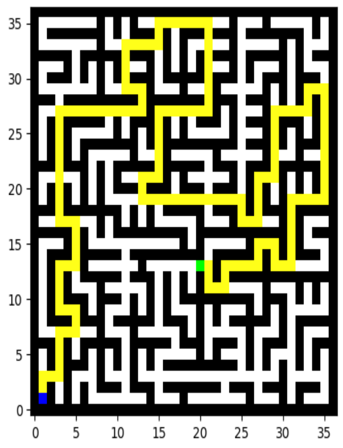


Figure 28: The extracted path

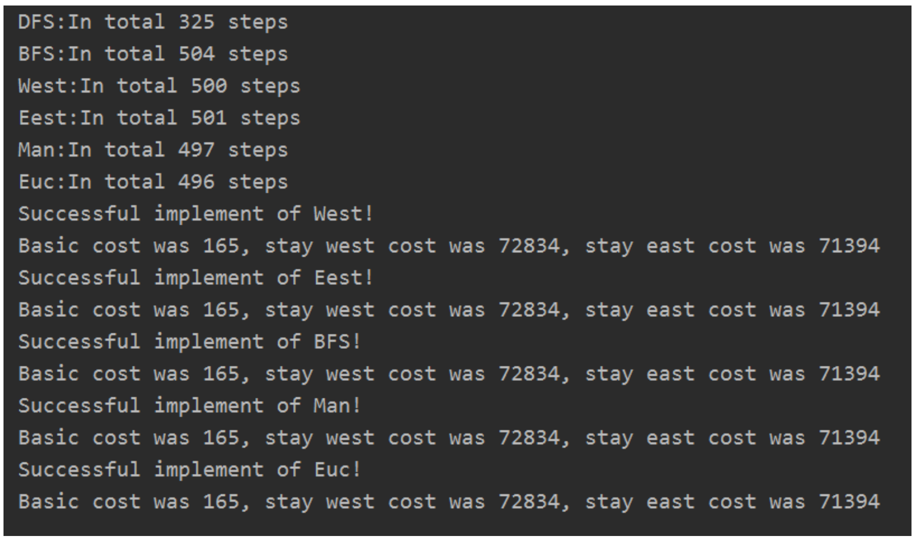

```
DFS:In total 325 steps
BFS:In total 504 steps
West:In total 500 steps
Eest:In total 501 steps
Man:In total 497 steps
Euc:In total 496 steps
Successful implement of West!
Basic cost was 165, stay west cost was 72834, stay east cost was 71394
Successful implement of Eest!
Basic cost was 165, stay west cost was 72834, stay east cost was 71394
Successful implement of BFS!
Basic cost was 165, stay west cost was 72834, stay east cost was 71394
Successful implement of Man!
Basic cost was 165, stay west cost was 72834, stay east cost was 71394
Successful implement of Euc!
Basic cost was 165, stay west cost was 72834, stay east cost was 71394
```

Figure 29: Programming outputs

From Fig 29, the steps for each are 497 and 496.

Compare the performance for the two different heuristic functions, I find it hard to tell which one is superior. However, since both of them are underestimates of the real optimal cost-to-go function which we do not know(because they ignore the obstacles), the path should be optimal using any of them. However, the manhattan distance is always lager than euclidean distance, it would focus more on the right path(we have no idea of what it is). Just think of Dijksta algorithm, in which case the cost-to-go function is zero, it would explore the whole space and finally could find the optimal path. In this sense, the manhattan distance is closer to the real optimal cost, so I think it would bring in better performance(fewer search steps). And both of them could find the optimal path.