Optimal and Learning Control for Robotics
Exercise Series 1

Bo Peng, New York University

Spring 2020

Exercise 1

Solution a) This problem is an open-loop optimal control problem. To solve this problem we need to do backward recursion:

- At step $N = 3$, calculate the terminal cost:

$$J_N = g_N(x_N) = x_3^2 \tag{1}$$

So, the terminal cost is:

$$J_3(x_3) = \begin{cases} 4, & x_3 = \pm 2 \\ 1, & x_3 = \pm 1 \\ 0, & x_3 = 0 \end{cases} \tag{2}$$

- From step $N - 1$ to step 0, calculate the cost to go with recursion:

$$\begin{aligned} J_k(x_k) &= \min_{u_k} g_k(x_k, u_k) + J_{k+1}(f(x_k, u_k)) \\ &= \min_{u_k} 2|x_k| + |u_k| + J_{k+1}(x_{k+1}) \end{aligned} \tag{3}$$

where x_{k+1} is determined by the environment dynamics:

$$x_{k+1} = \begin{cases} -x_k + 1 + u_k, & \text{if } -2 \leq -x_k + 1 + u_k \leq 2 \\ 2, & \text{if } -x_k + 1 + u_k > 2 \\ -2, & \text{else} \end{cases} \tag{4}$$

For example, in step 2 the cost to go with $x_2 = -2$ is:

$$\begin{aligned} J_2(x_2 = -2) &= \min_{u_2} g_k(x_2 = -2, u_2) + J_3(f(x_2 = -2, u_2)) \\ &= \min_{u_2} 4 + |u_2| + J_3(x_3) \end{aligned} \tag{5}$$

If $u_2 = -1$, the cost is:

$$\begin{aligned} J_2(x_2 = -2, u_2 = -1) &= 4 + 1 + J_3(2 + 1 - 1) \\ &= 5 + J_3(2) \\ &= 9 \end{aligned} \tag{6}$$

Similarly,

$$J_2(x_2 = -2, u_2 = 0) = 4 + J_3(2) \\ = 8 \tag{7}$$

$$J_2(x_2 = -2, u_2 = 1) = 5 + J_3(2) \\ = 9 \tag{8}$$

So the optimal control for $x_2 = -2$ is $u_2 = 0$ and the cost to go is 8.

- Following this step for all the states from step 2 to step 0, we could get the optimal control sequence for every time step.

The python code is shown here:

```python
def opt_ol(Xs, Us, N, env, terminal):
  """
  Find the optimal control sequence for open-loop cases.

  Args:
    Xs: the states;
    Us: the possible actions for each state;
    N: the number of time steps;
    env: the environment dynamics;
    terminal: the terminal step cost function.

  Returns:
    result: the optimal control sequence including the cost to go and
               the optimal control at every stage.
  """

  def recur(t, J, result):
    """
    A helper function to do dynamic programming recursively.
    """

    if t < 0:
      return result
    elif t == N:
      result["Time Step"] = []
      result["State"] = []
      result["Optimal Control"] = []
      result["Cost to Go"] = []

      J_new = {}
      for x in Xs:
        J_new[x] = terminal(x)

        result["Time Step"].append(t)
        result["State"].append(x)
        result["Optimal Control"].append(None)
        result["Cost to Go"].append(J_new[x])

    else:
      J_new = {}
      for x in Xs:
        j = {}
        for u in Us[x]:
          x_next, cost = env(x, u)
          j[u] = J[x_next] + cost

        u_opt, J_new[x] = min(j.items(), key=lambda x: x[1])

        result["Time Step"].append(t)
        result["State"].append(x)
        result["Optimal Control"] .append(u_opt)
        result["Cost to Go"].append(J_new[x])

    return recur(t-1, J_new, result)

  return recur(N, {}, {})


def env(x, u):
  x_next = -x + 1 + u
  if x_next > 2:
    x_next = 2
  elif x_next < -2:
    x_next = -2

  cost = 2*abs(x) + abs(u)

  return x_next, cost

def terminal(x):
  return x*x

Xs = [-2, -1, 0, 1, 2]
Us = {x: [-1, 0, 1] for x in Xs}
N = 3

```

Listing 1: Open-Loop Optimal Control Algorithm

The results in shown below:

Table 1: Optimal Control Sequence

Time Step	State	Optimal Control	Cost to Go
0	-2	0	10
0	-1	-1	6
0	0	-1	3
0	1	0	4
0	2	1	7
1	-2	0	9
1	-1	-1	5
1	0	-1	2
1	1	0	3
1	2	1	6
2	-2	0	8
2	-1	-1	4
2	0	-1	1
2	1	0	2
2	2	0	5
3	-2	-	4
3	-1	-	1
3	0	-	0
3	1	-	1
3	2	-	4

b) The sequence of control actions, states and the optimal cost are shown below:

Table 2: Optimal Control Sequence for Different x_0

x_0	u_0	J_0	x_1	u_1	J_1	x_2	u_2	J_2	x_3	J_3
0	-1	3	0	-1	2	0	-1	1	0	0
-2	0	10	2	1	6	0	-1	1	0	0
2	1	7	0	-1	2	0	-1	1	0	0

c) This problem is an closed-loop optimal control problem. The difference between open-loop and closed-loop control is that we could not know the state in the next time step in advance because of the uncertainties. To solve this problem we need to do backward recursion:

- At step $N = 3$, calculate the terminal cost:

$$J_N = g_N(x_N) = x_3^2 \tag{9}$$

So, the terminal cost is:

$$J_3(x_3) = \begin{cases} 4, & x_3 = \pm 2 \\ 1, & x_3 = \pm 1 \\ 0, & x_3 = 0 \end{cases} \tag{10}$$

- From step $N-1$ to step 0, calculate the cost to go with recursion:

$$\begin{aligned} J_k(x_k) &= \min_{u_k} \mathbb{E}\Big[g_k(x_k,u_k,w_k) + J_{k+1}(f(x_k,u_k,w_k))\Big] \\ &= \min_{u_k} \mathbb{E}\Big[2|x_k| + |u_k| + J_{k+1}(x_{k+1})\Big] \end{aligned} \tag{11}$$

  where x_{k+1} is determined by the environment dynamics:

$$x_{k+1} = \begin{cases} -x_k + w_k + u_k, & \text{if } -2 \le -x_k + 1 + u_k \le 2 \\ 2, & \text{if } -x_k + 1 + u_k > 2 \\ -2, & \text{else} \end{cases} \tag{12}$$

  and w_k is the uncertainty:

$$w_k = \begin{cases} 0, & p(w_k = 0) = 0.3 \\ 1, & p(w_k = 1) = 0.7 \end{cases} \tag{13}$$

  For example, in step 2 the cost to go with $x_2 = -2$ is:

$$\begin{aligned} J_2(x_2 = -2) &= \min_{u_2} \mathbb{E}\Big[2|x_2| + |u_2| + J_3(x_3)\Big] \\ &= \min_{u_2} \mathbb{E}\Big[4 + |u_2| + J_3(x_3)\Big] \end{aligned} \tag{14}$$

  If $u_2 = -1$, the cost is:

$$\begin{aligned} J_2(x_2 = -2, u_2 = -1) &= P(w_2 = 0) \times [4 + 1 + J_3(2 + 0 - 1)] + P(w_2 = 1) \times [4 + 1 + J_3(2 + 1 - 1)] \\ &= 0.3 \times [5 + J_3(1)] + 0.7 \times [5 + J_3(2)] \\ &= 8.1 \end{aligned} \tag{15}$$

  Similarly,

$$\begin{aligned} J_2(x_2 = -2, u_2 = 0) &= 0.3 \times [4 + J_3(2)] + 0.7 \times [4 + J_3(2)] \\ &= 8 \end{aligned} \tag{16}$$

$$\begin{aligned} J_2(x_2 = -2, u_2 = 1) &= 0.3 \times [5 + J_3(2)] + 0.7 \times [5 + J_3(2)] \\ &= 9 \end{aligned} \tag{17}$$

  So the optimal control for $x_2 = -2$ is $u_2 = 0$ and the cost to go is 8.

- Following this step for all the states from step 2 to step 0, we could get the optimal control policy for every time step.

The python code is shown here:

```python
def opt_cl(Xs, Us, Ws, N, env, terminal):
  """
  Find the optimal control sequence for closed-loop cases.

  Args:
    Xs: the states;
    Us: the possible actions for each state;
```

```
      Ws: the probability function for noises;
      N: the number of time steps;
      env: the environment dynamics;
      terminal: the terminal step cost function.

    Returns:
      result: the optimal control sequence including the cost to go and
                the optimal control at every stage.
    """

    def recur(t, J, result):
      """
      A helper function to do dynamic programming recursively.
      """

      if t < 0:
        return result
      elif t == N:
        result["Time Step"] = []
        result["State"] = []
        result["Optimal Control"] = []
        result["Cost to Go"] = []

        J_new = {}
        for x in Xs:
          J_new[x] = terminal(x)

          result["Time Step"].append(t)
          result["State"].append(x)
          result["Optimal Control"].append(None)
          result["Cost to Go"].append(J_new[x])

      else:
        J_new = {}
        for x in Xs:
          j = {}
          for u in Us[x]:
            j[u] = 0
            for w, p in Ws.items():
              x_next, cost = env(x, u, w)
              j[u] += p*(J[x_next] + cost)

          u_opt, J_new[x] = min(j.items(), key=lambda x: x[1])

          result["Time Step"].append(t)
          result["State"].append(x)
          result["Optimal Control"] .append(u_opt)
          result["Cost to Go"].append(J_new[x])

      return recur(t-1, J_new, result)

    return recur(N, {}, {})


  def env(x, u, w):
    x_next = -x + w + u
    if x_next > 2:
      x_next = 2
    elif x_next < -2:
      x_next = -2
    cost = 2*abs(x) + abs(u)

    return x_next, cost

  def terminal(x):
    return x*x


  Xs = [-2, -1, 0, 1, 2]
  Us = {x: [-1, 0, 1] for x in Xs}
  Ws = {0: 0.3, 1: 0.7}
  N = 3
```

Listing 2: Closed-Loop Optimal Control Algorithm

The results in shown below:

Table 3: Optimal Control Sequence

Time Step	State	Optimal Control	Cost to Go
0	-2	0	10.60
0	-1	-1	6.07
0	0	0	3.07
0	1	0	4.72
0	2	1	7.72
1	-2	0	9.30
1	-1	-1	4.82
1	0	0	1.82
1	1	0	3.60
1	2	1	6.60
2	-2	0	8.00
2	-1	-1	3.70
2	0	0	0.70
2	1	0	2.30
2	2	1	5.30
3	-2	-	4.00
3	-1	-	1.00
3	0	-	0.00
3	1	-	1.00
3	2	-	4.00

d) The costs in probabilistic models are fluctuating from the costs in the deterministic model, and some optimal controls are different in the two kind of models. I think it's because of the uncertainty w_k in the probabilistic model. When doing the backward recursion, the state could transfer to 2 possible states in each step and we need to take (weighted) average of the cost of the 2 states in the probabilistic case; while there's only one possible state in the deterministic case. So the optimal control in the deterministic model may not be optimal in the probabilistic model and the optimal cost changes as well.

Exercise 2

Solution a) This problem is almost the same as Exercise 1, with the only difference on environment dynamics, cost function and some parameters. To solve this problem, we only need to change these. The python codes for this problem are changed to below:

```python
def env(x, u):
  x_next = x + u
  cost = (x+4)**2 + u*u

  return x_next, cost

def terminal(x):
  if x == 0:
    return 0
  return float("inf")

Xs = [-4, -3, -2, -1, 0, 1, 2, 3, 4]
Us = {-4: [0, 1, 2],
      -3: [-1, 0, 1, 2],
      -2: [-2, -1, 0, 1, 2],
      -1: [-2, -1, 0, 1, 2],
       0: [-2, -1, 0, 1, 2],
       1: [-2, -1, 0, 1, 2],
       2: [-2, -1, 0, 1, 2],
       3: [-2, -1, 0, 1],
       4: [-2, -1, 0],
        }
N = 15

```

Listing 3: Parameter Changes

Then, we can use the open-loop optimal control algorithm function defined in Exercise 1 to solve this problem.

b) The cost-to-go for $x_4 = 4$ is:

$$J_4(x_4 = 4) = 146 \tag{18}$$

c) The states and controls plot for $x_0 = 0, 1, -4$ are shown below:

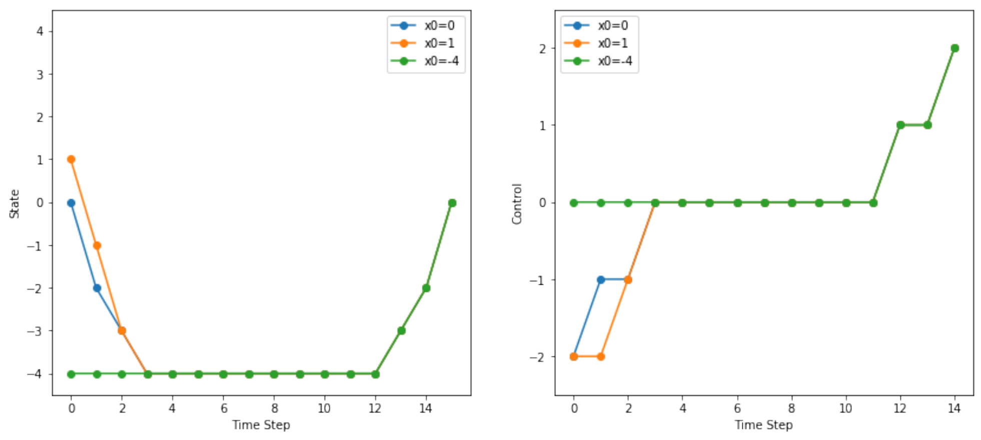

Figure 1: States and Controls Plot

And the cost-to-go are:

$$J_0(x_0) = \begin{cases} 38, & x_0 = 0 \\ 55, & x_0 = 1 \\ 11, & x_0 = -4 \end{cases} \tag{19}$$

d) In this case, we only need to change the terminal cost function:

```python
def terminal(s):
  return 0

```

Listing 4: Terminal Cost Function Changes

The plots are:

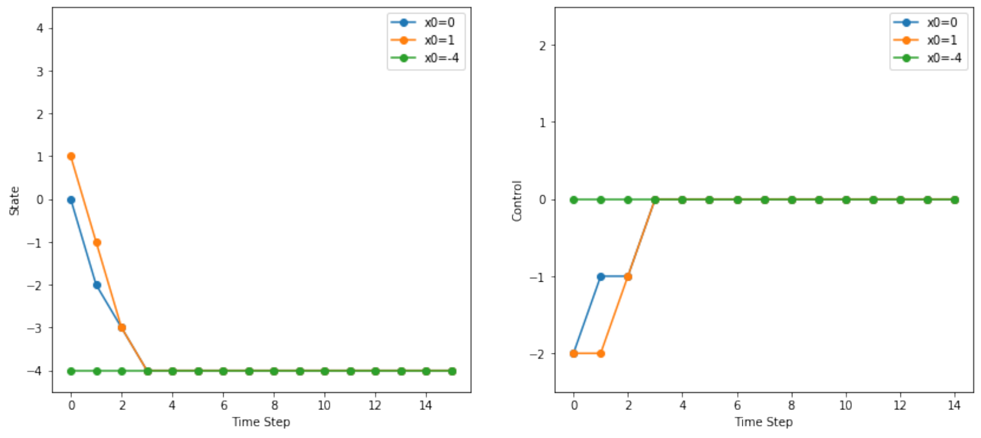

Figure 2: States and Controls Plot

And the cost-to-go are:

$$J_0(x_0) = \begin{cases} 27, & x_0 = 0 \\ 44, & x_0 = 1 \\ 0, & x_0 = -4 \end{cases} \tag{20}$$

In this problem the optimal cost decreases, and the optimal controls remain the same at the beginning but all stick to $u_k = 0$ in the later steps for each initial state. The reason for this phenomenon is that there's no terminal cost function any more. Since all the states has the same terminal cost, the optimal control would try to minimize the stage cost function $g_k = (x_k + 4)^2 + u_k^2$ for each step. So the optimal state is $x_k = -4$ with the optimal control $u_k = 0$ (the stage cost is 0), and the control sequence will try to transfer to this state from any other states and then stay in this state forever.

e) In this case, we need to change the environment dynamics:

```
  def env(x, u):
    x_next = x + u
    cost = (x+4)**2

    return x_next, cost

```

Listing 5: Terminal Cost Function Changes

The plots are:

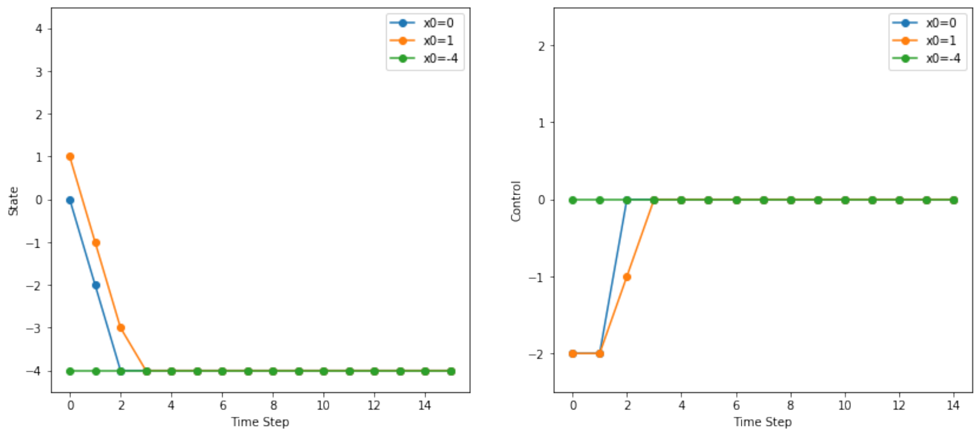

Figure 3: States and Controls Plot

And the cost-to-go are:

$$J_0(x_0) = \begin{cases} 20, & x_0 = 0 \\ 35, & x_0 = 1 \\ 0, & x_0 = -4 \end{cases} \tag{21}$$

In this problem, the optimal cost decreases again for each initial state, and the optimal control becomes more aggressive for the initial state $x_0 = 0$. The reason for this is that there's no cost on the control itself in the stage cost function. So the control sequence will try to get to the optimal state $x_k = -4$ as quick as possible without caring the control cost.

Exercise 3

Solution a) The number of tested nodes and length of these shortest paths are listed below:

Table 4: Shortest Path with DFS

	Example Maze	Maze1	Maze2	Maze3
Number of Tested Nodes	17	223241	176721	332734
Length of Path	9	99	131	141

The shortest path for each maze:

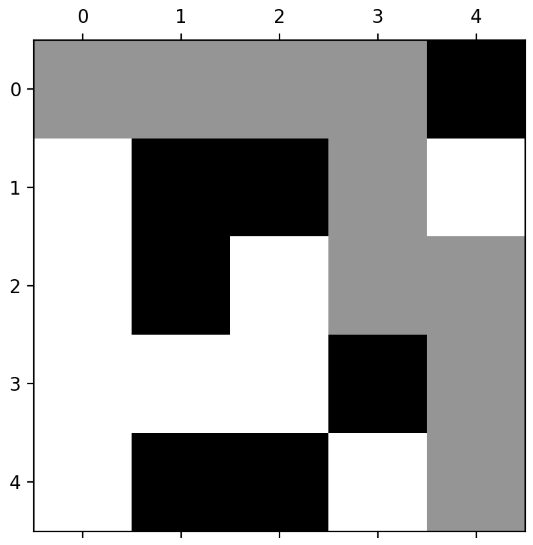

Figure 4: Shortest Path for Example Maze (DFS)

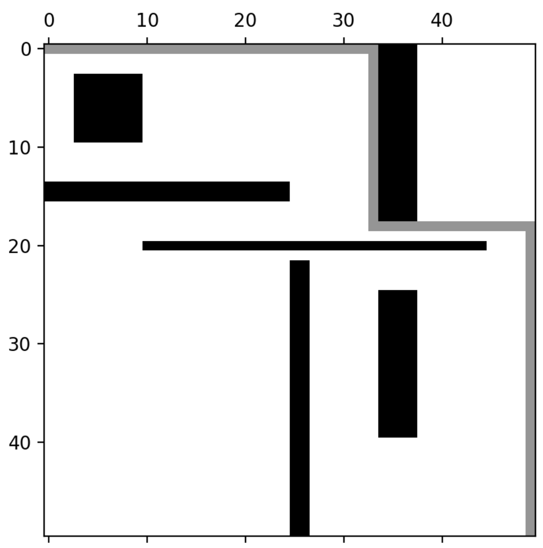

Figure 5: Shortest Path for Maze1 (DFS)

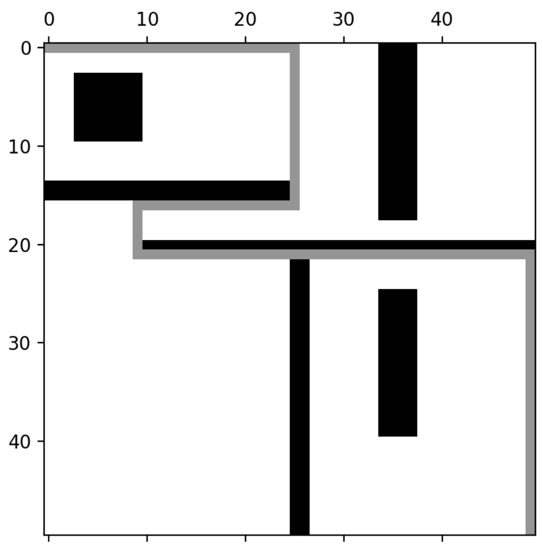

Figure 6: Shortest Path for Maze2 (DFS)

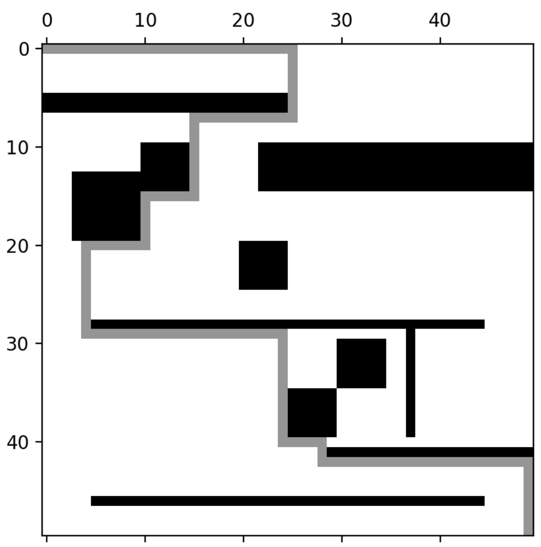

Figure 7: Shortest Path for Maze3 (DFS)

b) The number of tested nodes and length of these shortest paths are listed below:

Table 5: Shortest Path with BFS

	Example Maze	Maze1	Maze2	Maze3
Number of Tested Nodes	16	2024	2172	2048
Length of Path	9	99	131	141

The shortest path for each maze:

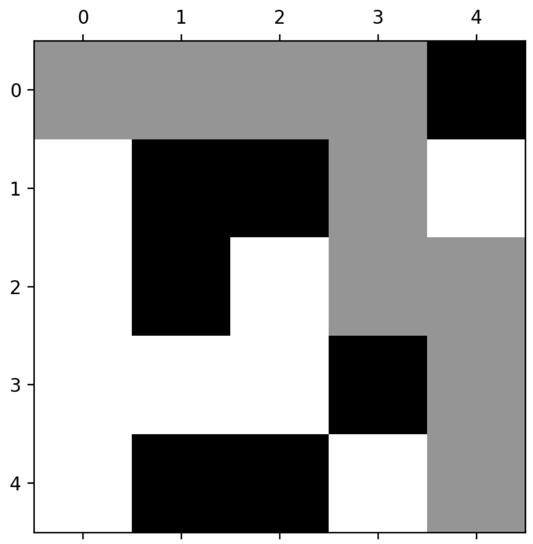

Figure 8: Shortest Path for Example Maze (BFS)

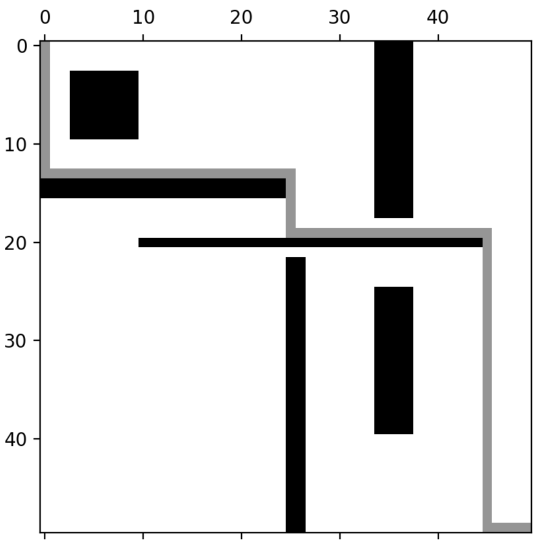

Figure 9: Shortest Path for Maze1 (BFS)

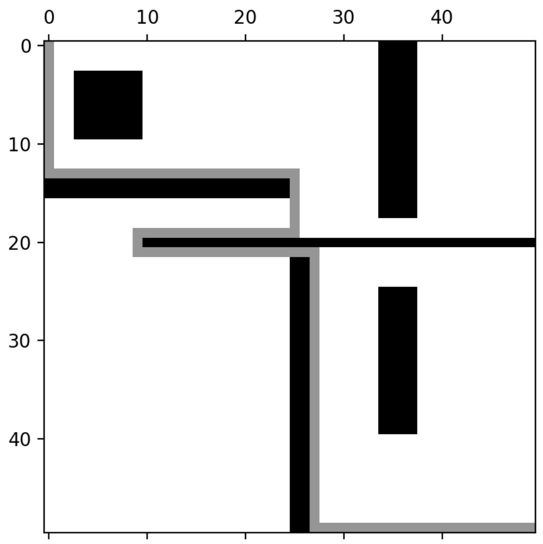

Figure 10: Shortest Path for Maze2 (BFS)

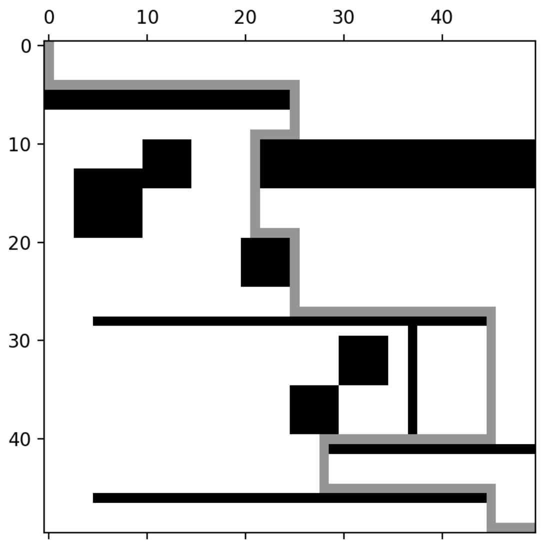

Figure 11: Shortest Path for Maze3 (BFS)

c) The number of tested nodes and length of these shortest paths are listed below:

Table 6: Shortest Path with A* ($h_{i,j} = 0$)

	Example Maze	Maze1	Maze2	Maze3
Number of Tested Nodes	17	2037	2137	2049
Length of Path	9	99	131	141

The heuristic function used here is a under-estimator of cost-to-go. For the $h_{i,j} = 0$ case, since the cost-to-go is always larger or equal to 0, this heuristic is always less or equal to the cost-to-go. So, it's a under-estimator.

The shortest path for each maze:

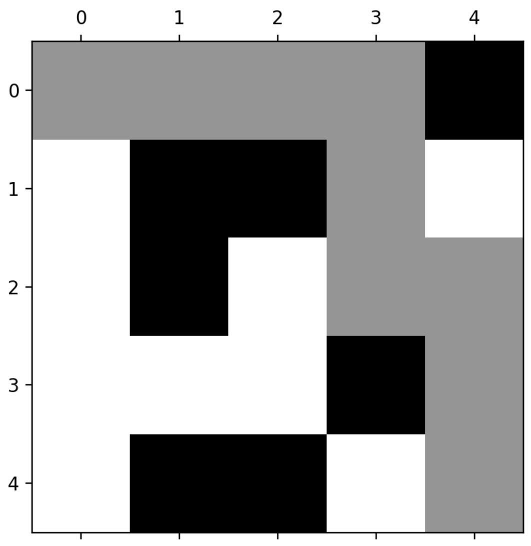

Figure 12: Shortest Path for Example Maze (A*, $h_{i,j} = 0$)

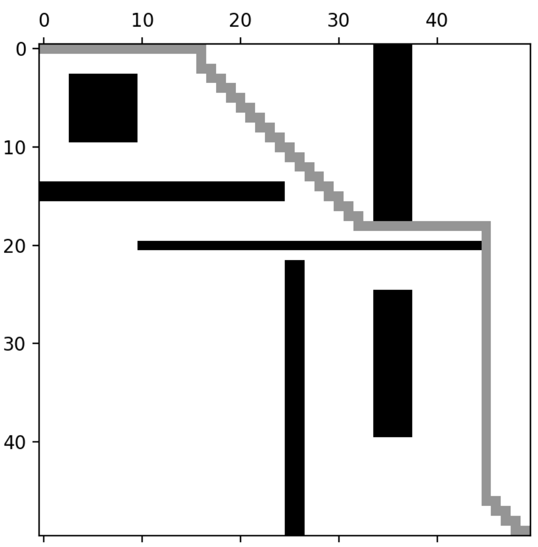

Figure 13: Shortest Path for Example Maze1 (A*, $h_{i,j} = 0$))

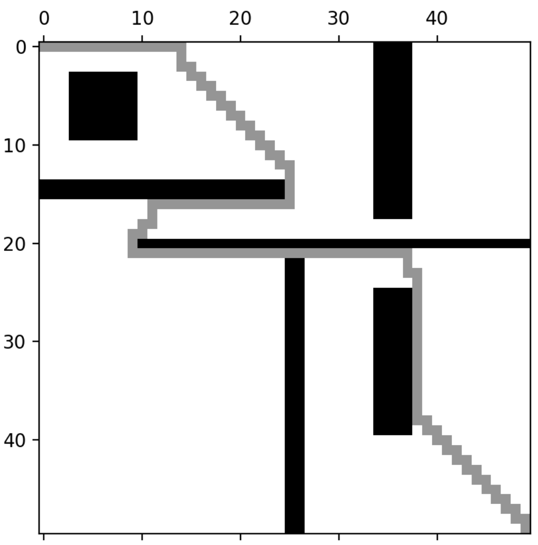

Figure 14: Shortest Path for Example Maze2 (A*, $h_{i,j} = 0$)

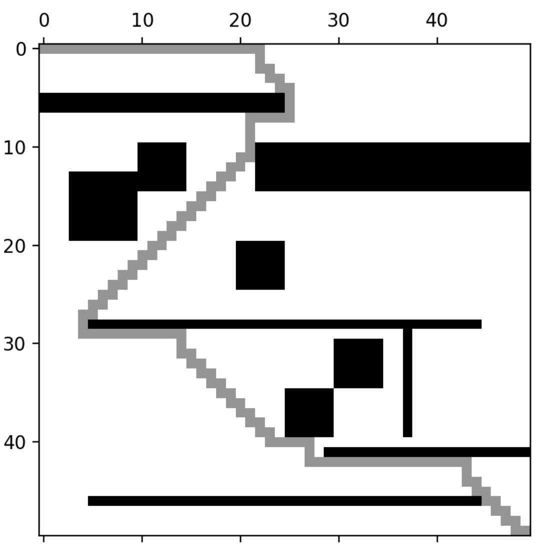

Figure 15: Shortest Path for Maze3 (A*, $h_{i,j} = 0$)

Table 7: Shortest Path with A* ($h_{i,j} = |i - \text{goal}_i| + |j - \text{goal}_j|$)

	Example Maze	Maze1	Maze2	Maze3
Number of Tested Nodes	9	99	1213	1917
Length of Path	9	99	131	141

For the $h_{i,j} = |i - \text{goal}_i| + |j - \text{goal}_j|$ case, suppose there are no obstacles in the maze, then the cost-to-go equals to the heuristic. We can choose a path from the optimal solutions. After adding obstacles in the maze, if there exist obstacles in this path, we need to bypass the obstacles, which leads to additional cost but the heuristic remains the same. So the heuristic is less than the cost-to-go now. In conclusion, this heuristic function is always less or equal to the cost-to-go, so it's a under-estimator.

The shortest path for each maze:

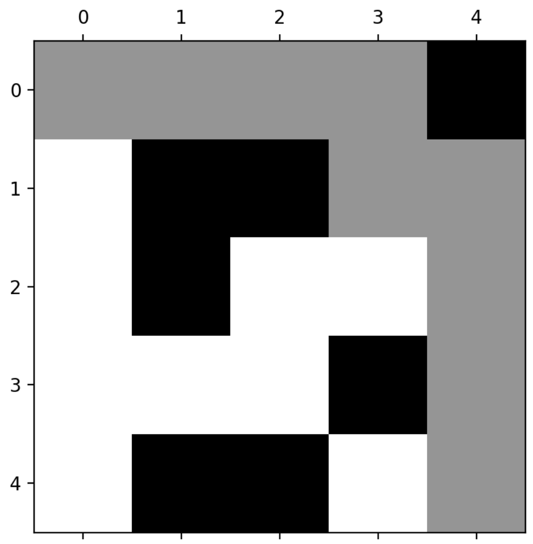

Figure 16: Shortest Path for Example Maze (A*, $h_{i,j} = |i - \text{goal}_i| + \left|j - \text{goal}_j\right|$)

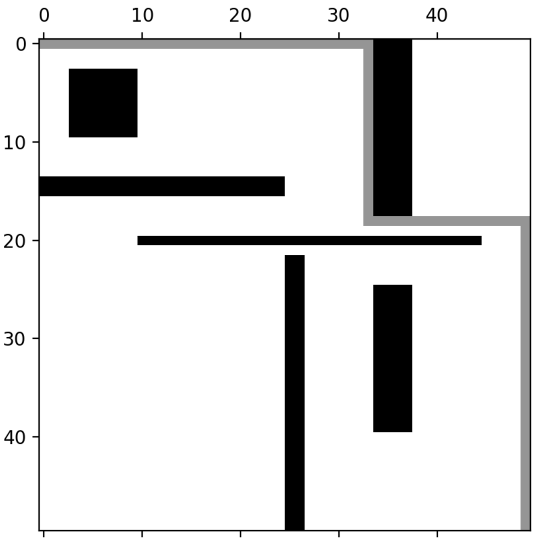

Figure 17: Shortest Path for Example Maze1 (A*, $h_{i,j} = |i - \text{goal}_i| + \left|j - \text{goal}_j\right|$))

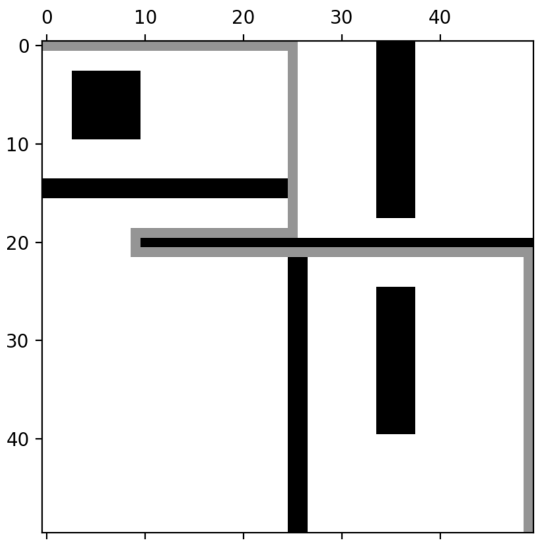

Figure 18: Shortest Path for Example Maze2 (A*, $h_{i,j} = |i - \text{goal}_i| + |j - \text{goal}_j|$)

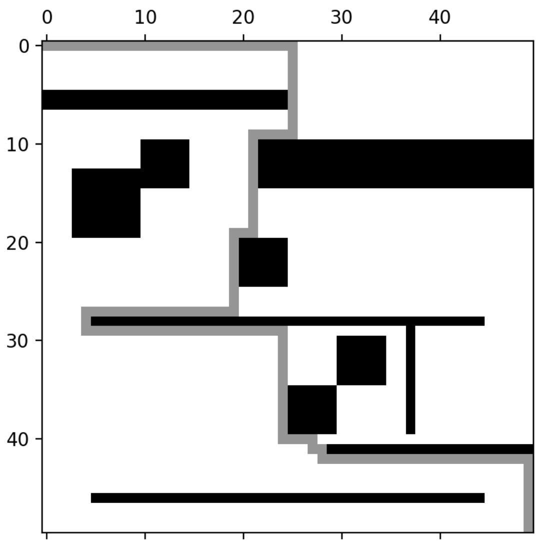

Figure 19: Shortest Path for Maze3 (A*, $h_{i,j} = |i - \text{goal}_i| + |j - \text{goal}_j|$)

d) The pros and cons of the above algorithms are listed below:

Table 8: Pros and Cons of the Algorithms

	Pros	Cons
DFS	Easy to implement; Usually memory efficient.	Can be less efficient in some case;
BFS	Easy to implement. Can be more efficient than DFS in some cases.	Not memory efficient compared to DFS;
A*	Much more efficient than DFS and BFS with proper heuristic function.	Harder to implement; Extracting the node with minimal cost could lead to extra time consumption; Picking heuristic function can be tricky; Improper heuristic function will lead to sub-optimal solution.