Tingzhou Wan

Artificial Intelligence Assignment 1 Report

2023-03-10

Abstract

Maze solving is a well-known computer science topic that has been thoroughly investigated using a variety of algorithms. The performance of five distinct maze-solving algorithms—Depth-First Search (DFS), Breadth-First Search (BFS), A* search, Value Iteration, and Policy Iteration—is examined in this research.

Uninformed search algorithms DFS and BFS both explore as far as they can down each branch before turning around, while BFS first investigates all of the nodes nearby before proceeding to the next level. A* search is a methodical search algorithm that employs a heuristic function to direct the search in the right direction.

Value Iteration and Policy Iteration are two model-based techniques for solving maze utilizing Markov Decision Processes (MDP). In contrast to Policy Iteration, which alternates between steps of policy evaluation and policy improvement until the best policy is identified, Value Iteration computes the optimal value function by repeatedly using the Bellman optimality equation.

In this study, we compare how well these algorithms performed when used to solve mazes created by the Pyamaze library, which ranged in size and complexity. We assess each team's

performance using a variety of criteria, such as the number of nodes they have examined, how long it took them to figure out the maze, and how long the best path was. We also go through the advantages and disadvantages of each algorithm and offer suggestions for selecting the best algorithm for a certain situation when solving a maze.

Performance Analysis:

To evaluate the performance of each algorithm implemented in this assignment, I used 3 metrics for evaluation of the results: time cost, cells searched and length of the shortest path.
Time costs refer to the amount of time it takes for an algorithm to find the shortest path from the start to the goal. This metric can be useful in assessing the efficiency of an algorithm in terms of speed, and is especially important in real-time applications where quick response times are necessary. Algorithms that can find the shortest path in less time are generally considered more efficient.

Cells searched refers to the number of cells that an algorithm explores during the search process. This metric is closely related to time costs, as algorithms that search fewer cells generally take less time. However, cells searched can provide additional insights into the behavior of an algorithm, such as whether it explores dead-ends or backtracks frequently. Algorithms that search fewer cells while still finding the shortest path are generally considered more effective.

The length of the shortest path refers to the number of cells that make up the shortest path from the start to the goal. This metric is important because it directly measures the quality of the

solution provided by the algorithm. Algorithms that find shorter paths are generally considered more effective.

Results:

5x5 maze:

5x5	A*	DFS	BFS	MDP-Policy Iteration	MDP-Value Iteration
time cost	0.005392874999	0.001360791008	0.003303375008	0.001504375008	0.001303375008
shortest path length	9	9	9	9	9
cells searched	11	10	25	11	13

10x10 maze:

10x10	A*	DFS	BFS	MDP-Policy Iteration	MDP-Value Iteration
time cost	0.01668291699	0.004736833012	0.02486775001	0.01668291699	0.01668291699
shortest path length	19	23	19	19	19
cells searched	30	24	100	27	29

20x20 maze:

20x20	A*	DFS	BFS	MDP-Policy Iteration	MDP-Value Iteration
time cost	0.05196175	0.011703667	0.2903635	0.07794262499	0.1613130556
shortest path length	39	43	39	39	43
cells searched	92	44	400	80	95

Evaluation:

As we can see from the results tables, BFS normally holds the longest running time and largest number of expanded cells among 5 algorithms and finds the optimal way from the start to the goal. While DFS normally holds the least running time and least number of expanded cells among 5 different algorithms, it does not return an optimal path to the goal each time. Meanwhile, for the 2 MDP algorithms, policy iteration usually performs better in the time cost and both algorithms return the optimal path to the goal. When making a comparison between MDP and search algorithms, MDP usually performs a high accuracy in finding the optimal path while DFS still holds a less time cost in searching. DFS searches deep into the maze by exploring each path as far as possible before backtracking. This approach can be faster than BFS, A*, and MDP algorithms in certain cases where the optimal solution is not required, but instead, a solution that reaches the goal state as quickly as possible is sufficient. DFS can often quickly identify a solution path when the goal is located close to the start.

MDP value iteration and policy iteration are reinforcement learning algorithms that can be used to find the optimal policy for maze solving. These algorithms can handle more complex mazes than BFS, DFS, and A* and can find an optimal policy that maximizes the expected cumulative reward. However, MDP algorithms are computationally expensive and require a complete model of the environment. Therefore, they may not be suitable for real-time maze solving applications or large-scale mazes.

In conclusion, the selection of an algorithm for solving a maze depends on the particulars of the task, including the size and complexity of the maze, the desired level of solution quality, and the available computer resources. Finding the shortest route to the objective is best handled by BFS

and A*, however DFS can be quicker when an approximation would do. When the objective is to identify the best possible policy that maximizes the expected reward, MDP value iteration and policy iteration are more suitable.

Appendix:

BFS:

```
from collections import deque
def BFS(maze):
    start=(maze.rows,maze.cols)
    frontier = deque()
    frontier.append(start)
    bfsPath = {}
    explored = [start]
    searchPath=[]
    while len(frontier)>0:
        tempCell=frontier.popleft()
        if tempCell==maze._goal:
            break
        for d in 'ESNW':
            if maze.maze_map[tempCell][d]==True:
                if d=='E':
                    childCell=(tempCell[0],tempCell[1]+1)
                elif d=='W':
```

```
                childCell=(tempCell[0],tempCell[1]-1)
            elif d=='S':
                childCell=(tempCell[0]+1,tempCell[1])
            elif d=='N':
                childCell=(tempCell[0]-1,tempCell[1])
            if childCell in explored:
                continue
             frontier.append(childCell)
            explored.append(childCell)
            bfsPath[childCell] = tempCell
            searchPath.append(childCell)
    shortestPath={}
    cell=maze._goal
    while cell!=(maze.rows,maze.cols):
        shortestPath[bfsPath[cell]]=cell
        cell=bfsPath[cell]
    return searchPath,shortestPath
```

DFS:

```
def DFS(m):
    start=(m.rows,m.cols)
    explored=[start]
    frontier=[start]
    dfsPath={}
    searchPath=[]
    while len(frontier)>0:
```

```
    tempCell=frontier.pop()
    searchPath.append(tempCell)
    if tempCell==m._goal:
        break
    poss=0
    for d in 'ESNW':
        if m.maze_map[tempCell][d]==True:
            if d =='E':
                child=(tempCell[0],tempCell[1]+1)
            if d =='W':
                child=(tempCell[0],tempCell[1]-1)
            if d =='N':
                child=(tempCell[0]-1,tempCell[1])
            if d =='S':
                child=(tempCell[0]+1,tempCell[1])
            if child in explored:
                continue
            poss+=1
            explored.append(child)
            frontier.append(child)
            dfsPath[child]=tempCell
    if poss>1:
        m.markCells.append(tempCell)
shortestPath={}
cell=m._goal
while cell!=start:
    shortestPath[dfsPath[cell]]=cell
```

```
        cell=dfsPath[cell]
    return searchPath,shortestPath
```

Astar

```
from queue import PriorityQueue



def h(cell1, cell2):
    x1, y1 = cell1
    x2, y2 = cell2
    return (abs(x1 - x2) + abs(y1 - y2))


def aStar(m):
    start = (m.rows, m.cols)
    frontier = PriorityQueue()
    frontier.put((h(start, m._goal), h(start, m._goal), start))
    astarPath = {}
    g_score = {row: float("inf") for row in m.grid}
    g_score[start] = 0
    f_score = {row: float("inf") for row in m.grid}
    f_score[start] = h(start, m._goal)
    searchPath = [start]
    while not frontier.empty():
        tempCell = frontier.get()[2]
        searchPath.append(tempCell)
        if tempCell == m._goal:
```

```
            break
        for d in 'ESNW':
            if m.maze_map[tempCell][d] == True:
                if d == 'E':
                    childCell = (tempCell[0], tempCell[1] + 1)
                elif d == 'W':
                    childCell = (tempCell[0], tempCell[1] - 1)
                elif d == 'N':
                    childCell = (tempCell[0] - 1, tempCell[1])
                elif d == 'S':
                    childCell = (tempCell[0] + 1, tempCell[1])

                temp_g_score = g_score[tempCell] + 1
                temp_f_score = temp_g_score + h(childCell, m._goal)

                if temp_f_score < f_score[childCell]:
                    astarPath[childCell] = tempCell
                    g_score[childCell] = temp_g_score
                    f_score[childCell] = temp_g_score + h(childCell, m._goal)
                    frontier.put((f_score[childCell], h(childCell, m._goal),
childCell))

    shortestPath = {}
    cell = m._goal
    while cell != start:
        shortestPath[astarPath[cell]] = cell
        cell = astarPath[cell]
```

```
    return searchPath, shortestPath
```

MDP

```
import numpy as np

class MDP:

    def __init__(self, maze):

        self.maze = maze

        self.states = [(x, y) for x in range(maze.width) for y in
range(maze.height)]

        self.actions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

        self.gamma = 0.99


    def reward(self, state):

        if state == self.maze.goal:

            return 1.0

        else:

            return 0.0


    def transition_probs(self, state, action):

        next_state = tuple(np.array(state) + np.array(action))

        if next_state not in self.states or self.maze[next_state] == '#':

            return [(1.0, state)]

        else:

            return [(1.0, next_state)]
```

```
    def policy_iteration(maze):
        mdp = MDP(maze)

        pi = {s: mdp.actions[0] for s in mdp.states}
        while True:
            # Policy evaluation
            V = {s: 0.0 for s in mdp.states}
            eps = 1e-10
            while True:
                delta = 0
                for s in mdp.states:
                    v = V[s]
                    V[s] = sum(prob * (mdp.reward(next_state) + mdp.gamma *
V[next_state]) for prob, next_state in
                               mdp.transition_probs(s, pi[s]))
                    delta = max(delta, abs(v - V[s]))
                if delta < eps:
                    break

            # Policy improvement
            policy_stable = True
            for s in mdp.states:
                old_action = pi[s]
                pi[s] = max(mdp.actions, key=lambda a: sum(
                    prob * (mdp.reward(next_state) + mdp.gamma * V[next_state])
for prob, next_state in
                     mdp.transition_probs(s, a)))
```

```
                if pi[s] != old_action:
                    policy_stable = False
            if policy_stable:
                break

        # Find shortest path and search path using pi
        start = maze.start
        goal = maze.goal
        path = [start]
        search_path = [start]
        while path[-1] != goal:
            next_state = tuple(np.array(path[-1]) + np.array(pi[path[-1]]))
            path.append(next_state)
            search_path.append(next_state)
        return search_path,path

    def value_iteration(maze):
        mdp = MDP(maze)

        V = {s: 0.0 for s in mdp.states}
        eps = 1e-10
        while True:
            delta = 0
            for s in mdp.states:
                v = V[s]
                V[s] = max(sum(prob * (mdp.reward(next_state) + mdp.gamma *
V[next_state]) for prob, next_state in
```

```
                                mdp.transition_probs(s, a)) for a in mdp.actions)
                delta = max(delta, abs(v - V[s]))
            if delta < eps:
                break


        # Find shortest path and search path using V
        start = maze.start
        goal = maze.goal
        path = [start]
        search_path = [start]
        while path[-1] != goal:
            next_state = tuple(np.array(path[-1]) + np.array(max(mdp.actions,
key=lambda a: sum(
                prob * (mdp.reward(next_state) + mdp.gamma * V[next_state]) for
prob, next_state in
                 mdp.transition_probs(path[-1], a)))))
            path.append(next_state)
            search_path.append(next_state)
        return search_path,path
```

Reference

Naeem, Muhammad Ahsan. “A Python Module for Maze Search Algorithms.” *Medium*, 24 Sept. 2021, towardsdatascience.com/a-python-module-for-maze-search-algorithms-64e7d1297c96.

Tokuç, A. Aylin. “Value Iteration vs. Policy Iteration in Reinforcement Learning.” *Baeldung on Computer Science*, 9 Nov. 2022, https://www.baeldung.com/cs/ml-value-iteration-vs-policy-iteration.