PS3

Graded

Student

Chetan Hiremath

Total Points

90 / 100 pts

Question 1

(no title) **15** / 15 pts

1.1 **(a)** **7** / 7 pts

✔ **+ 7 pts** Correct

1.2 **(b)** **8** / 8 pts

✔ **+ 8 pts** Correct

Question 2

(no title) **15** / 15 pts

✔ **+ 15 pts** Correct

Question 3

(no title) **Resolved** **10** / 20 pts

✔ **+ 20 pts** Correct

– 10 pts not readable

Regrade Request **Submitted on: Feb 15**

Will you regrade Question 3? My answer is correct, and some parts are readable. How is this answer wrong? Will you let me know?

your answer may not be wrong, but it is not clear and readable.

Reviewed on: Feb 15

Question 4

(no title) **50** / 50 pts

✔ **+ 50 pts** Correct

Question assigned to the following page: 1.1

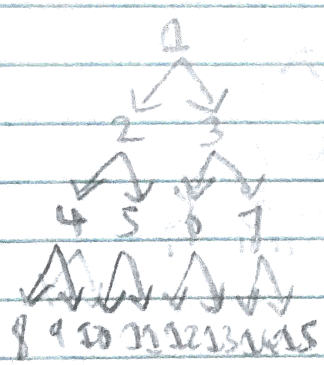
1a. States - There are 4 colors for a planar map.
Initial State - No regions are colored.
Actions - Assign a color to an uncolored region of the map.
Transition Model - The previous region that is not colored will have an assigned color.
Goal Test - It checks if the regions are colored without 2 adjacent regions with the same color.
Path Cost - Number of assignments.
Entire State Space X - Regions of the planar map.

Question assigned to the following page: 1.2

b. States - There are 3 jugs.
Initial State - 3 jugs have values [0, 0, 0].
Actions - Generate [12, y, z], [x, 8, z], and [x, y, 3] by filling. Generate [0, y, z], [x, 0, z], and [x, y, 0] by emptying.
Transition Model - Pour y int x, which changes to the minimum capacity of (x+y). The jug with y will decrease its capacity.
Goal test - It checks if one gallon is measured out in the jugs after filling or emptying.
Path Cost - Number of actions.
Entire State Space X - Smallest capacity of 3 jugs.

Question assigned to the following page: 2

2a.



Goal State - 11

b. Breadth-First Search -
1 2 3 4 5 6 7 8 9 10 11

Depth Limited Search with limit 3 -
1 2 4 8 9 5 10 11,

Iterative Deepening Search -
1; 1 2 3; 1 2 4 5 3 6 7; 1 2 4 8
9 5 10 11.

c. Bidirectional search works since the only successor of n in backward order is $[(\frac{n}{2})]$, and it helps the search focus. The branching factors are 2 and 1 in forward and backward directions respectively

Question assigned to the following page: 2

d. The previous answer suggests a reformulation that solves the problem by using single reverse successor action until state 1.

e. Let f be a function of input n. If n = 1, then the state is constant. If n is even, then the state k goes to state 2k Left. If the 2 conditions are false, then the state k goes to state (2k+1) Right.

Question assigned to the following page: 3

3a.

Lugoj 244 (244 = 0+244)

329 Timisoara mehadia 241 (311 = 70+241)

Lugoj 244 Drobeta 242 (387 = 145+242)

241 mehadia Craiova 160 (425 = 265+160)

Drobeta 242 Rimnicu Vilcea 193 Pitesti 100 (503 = 403+100) — Bucharest 0

Craiova 160 Rimnicu Vilcea 193 504 = 504+0

Straight-Line Distance Heuristic

b.

Lugoj 244 = 0+244

Timisoara 440 = 111+329 mehadia 311 = 70+241

Lugoj 384 = 140+244 Drobeta 387 = 145+242

550 = 222+329 Timisoara mehadia 451 = 210+241 mehadia 460 = 220+242

Lugoj 524 = 280+244 Drobeta 527 = 285+242

Craiova 425 = 265+160 Pitesti 503 = 403+100

Drobeta 627 = 385+242 Rimnicu Vilcea 606 = 413+193

Craiova 702 = 542+160 Rimnicu Vilcea 693 = 500+193 Bucharest 504 = 504+0

One-Step Look-Ahead

Question assigned to the following page: 4

4a.

Sources- The code of the 3 algorithms is borrowed, used, and modified from SwappingCounters.ipynb, BUG TRAP.ipynb, and PS3.4 Notes: Planning in a Grid World.

BFS General Implementation's Python Code-

```
success = False
closed = []
fringe = []
Q = Queue()
fringe.append(x_i)
Q.put(item=[x_i])
visited = np.zeros(shape=(8,4)) #Make a grid.
visited[x_i] = 1
while not Q.empty():
    x = Q.get()
    closed.append(x) #Include x in closed cells.
    fringe.remove(x[0]) #Remove x from fringe or frontier cells.
    x = x[0]
    #Check if the node reaches to the goal state.
    if x == X_g:
        success = True
        break
    for u in U(x, n, m, obs):
        _x = add_tuple(x,u)
        #Add the child node to the queue if it isn't visited.
        if not visited[_x]:
            visited[_x] = 1
            Q.put(item=[_x, x])
            fringe.append(_x)
        else: #Continue if the node is visited.
            Continue
```

Question assigned to the following page: 4

GBFS General Implementation's Python Code-

```
    success = False
    closed = []
    fringe = []
    #Track parent node of the queue.
    Q = [[abs(x_i[0] - X_g[0]) + abs(x_i[1]-X_g[1]), x_i, x_i]]
    fringe.append(x_i)
    heapq.heapify(Q)
    visited = np.zeros(shape=(8,4)) #Make a grid.
    visited[x_i] = 1
    while not len(Q) == 0:
        x = heapq.heappop(Q)
        closed.append([x[1], x[2]]) #Include x in closed cells.
        fringe.remove(x[1]) #Remove x from fringe or frontier cells.
        x = x[1]
        #Check if the node reaches to the goal state.
        if x == X_g:
            success = True
            break
        for u in U(x, n, m, obs):
            _x = add_tuple(x,u)
            #Add the child node to the queue if it isn't visited.
            if not visited[_x]:
                visited[_x] = 1
                heapq.heappush(Q, [abs(_x[0] - X_g[0]) + abs(_x[1]-X_g[1]),
_x, x])
                fringe.append(_x)
            else: #Continue if the node is visited.
                continue
```

Question assigned to the following page: 4

A* General Implementation's Python Code-

```
success = False
closed = []
fringe = []
#Track parent node of the queue.
Q = [[abs(x_i[0] - X_g[0]) + abs(x_i[1]-X_g[1]), x_i, x_i]]
fringe.append(x_i)
heapq.heapify(Q)
visited = np.zeros(shape=(8,4)) #Make a grid.
cost = np.full(shape=(8,4), fill_value=np.Infinity)
visited[x_i] = 1
cost[x_i] = 0
while not len(Q) == 0:
    x = heapq.heappop(Q)
    closed.append([x[1],x[2]]) #Include x in closed cells.
    fringe.remove(x[1]) #Remove x from fringe or frontier cells.
    x=x[1]
    #Check if the node reaches to the goal state.
    if x == X_g:
        success = True
        break
    for u in U(x, n, m, obs):
        _x = add_tuple(x,u)
        if cost[x] + 1 < cost[_x]: #Update the cost.
            cost[_x] = cost[x] + 1
        #Add the child node to the queue if it isn't visited.
        if not visited[_x]:
            visited[_x] = 1
            heapq.heappush(Q, [cost[_x] + abs(_x[0] - X_g[0]) +
abs(_x[1]-X_g[1]), _x, x])
            fringe.append(_x)
        else: #Continue if the node is visited.
            continue
```

Question assigned to the following page: 4

b. I have used the integer pairs that represent the initial and the goal states and the general implementations of the 3 algorithms on the example grid that is discussed during the lecture, and the results shows that the algorithms have enabled to reach to the goal state of this example grid successfully since I see the solved example grid on the terminal.

Question assigned to the following page: 4

c. I have changed the sizes of the grid, which is 100 X 100 grid and used and modified the code from BUG TRAP.ipynb for this part. Then, I have used the implementations of the 3 algorithms on this new example grid to get the results on the terminal. Here are the results of the new example grid.

Number of Closed Cells of BFS- 4368.

Number of Frontier Cells of BFS- 116.

Length of the Path of BFS- 53.

Number of Closed Cells of GBFS- 1042.

Number of Frontier Cells of GBFS- 75.

Length of the Path of GBFS- 53.

Number of Closed Cells of A*- 785.

Number of Frontier Cells of A*- 99.

Length of the Path of A*- 53.

Question assigned to the following page: 4

d. I have repeated the procedure of the previous part by reversing the initial and the goal states. Then, I have used the implementations of the 3 algorithms on this new example grid and the reversed states to get new results on the terminal. Here are the results of the new example grid with reversed states.

Number of Closed Cells of BFS- 2727.

Number of Frontier Cells of BFS- 47.

Length of the Path of BFS- 53.

Number of Closed Cells of GBFS- 53.

Number of Frontier Cells of GBFS- 42.

Length of the Path of GBFS- 53.

Number of Closed Cells of A*- 400.

Number of Frontier Cells of A*- 57.

Length of the Path of A*- 53.

Question assigned to the following page: 4

e. The results of the 3 algorithms' implementations show that the lengths of the path are same even though I have used different algorithms. But the other values liked closed cells and frontier cells' counts are different. They have managed to allow the initial state, which goes to the goal state successfully. Bidirectional search is advantageous for this class of problems since it has a direct solution and no large search space. It can reduce the number of nodes for exploration, so the forward search from the initial state and the backward state from the goal state are computed until the search frontiers meet at the same state. The benefits of this search are reduced search space, efficiency, and optimality for solutions that can't be computed by other algorithms. It can reduce the search space, converge to a solution, and lead to optimal solutions. So, this search is very helpful and provides an advantage of finding solutions fast.