CS4100 Problem Set 1

Silas Nevstad

July 2024

1. **Assumptions**:
   - Agent has a complete map of the maze.
   - Agent knows its current location within the maze.
   - The maze has no obstacles and does not change.

   Agent's Percepts:
   - Its current position in the maze.
   - The complete layout of the maze, (walls, paths, the goal).

2. **Assumptions**:
   - Agent doesn't have a complete map of the maze.
   - Agent can only see the maze around its location (e.g. within a range).
   - Maze could contain hidden obstacles or make changes.

   Agent's Percepts:
   - Information about the neighboring cells.
   - The agent's estimated current position.

3. No, a known environment is not always fully observable. For example, a maze where the agent has a complete map of the maze (with no changes or obstacles), but its visibility is limited to its surroundings, the environment is known but only partially observable. Here the agent would know the entire layout of the maze but can't see the entire maze at any given time.

4. No, a partially observable environment is not always an unknown environment. For example, a maze (with no changes or obstacles) where the agent has an incomplete map (for example, only shows parts of maze) and can only see the surrounding cells. The environment is partially observable because the agent's vision is limited, but it is not completely unknown as the agent has some prior knowledge of the maze structure.

5. An unknown environment means the agent has no information about how the environment works (layout, rules, results of actions, etc) and has to gain knowledge to understand it, however, it could be deterministic, where identical actions produce the same results. For example, consider a robot, like a Roomba, entering a new building for the first time. It has no map or information about the building's layout and must therefore explore each room to build an internal map and understand the environment (each action could have the same result, no obstacle keep moving, hit obstacle change direction).

   On the other hand, a non-deterministic environment means the outcomes of an agent's actions are not always the same and can vary (meaning when the same action is made under identical conditions the results could differ). For example, imagine a robot navigating a factory where conveyor belts can change direction randomly, and machines can start or stop unpredictably. Even if the robot has a complete map of the factory, it can't predict these random changes, making the results of its actions unpredictable.

   Essentially, an unknown environment requires the agent to learn and gain knowledge about its environment (could still be deterministic), while a non-deterministic environment has unpredictable outcomes (even if the agent knows the environment).

6. Both BFS and DFS can be used to solve RJMs, however, only BFS guarantees the shortest path. BFS explores nodes level by level, starting from the start state and expanding all neighbors at the current depth before moving on to the next level. This means that BFS will always find the shortest path, as each move has a uniform cost of one. DFS, on the other hand, does not guarantee it finds the shortest path since it explores as far as possible along each branch before backtracking, meaning it can potentially get stuck in a long, winding path, finding a non-optimal solution.

7. A heuristic h is consistent if $h(n) <= c(n, a) + h(a)$ for every node n and every successor a of n, where c(n, a) is the cost of getting from n to a. In this case, since each jump costs 1, $c(n, a) = 1$ for any move. The Manhattan distance between n and the goal can change by at most (n-1) in a single move (its the max possible change in x or y). Let's consider a move from node n to node a:

   So, $|H(n) - H(a)| <= (n-1)/(n-1) = 1$

   This means $H(n) - H(a) <= 1 = H(n) <= 1 + H(a)$

   And since $c(n, a) = 1$, we have, $H(n) <= c(n, a) + H(a)$

   This satisfies the consistency condition for any nodes n and a. Therefore, $H(node, goal, n) = (|nodex - goalx| + |nodey - goaly|)/(n-1)$ is a consistent heuristic for this problem.

8. **Step 2**

- Pop the node with the lowest priority, D2 (FIFO).
- Priority Queue, PQ = (B4, 5/3, 1, [B2])
- cost = 1
- v = D2
- path = [B2, D2]
- Since D2 is not the goal state, continue.
- Neighbors of D2 = C2, D1, D3
- Priority for C2 = cost + wt(D2, C2) + H(C2) = 1 + 1 + 3/3 = 3
- Priority for D1 = cost + wt(D2, D1) + H(D1) = 1 + 1 + 3/3 = 3
- Priority for D3 = cost + wt(D2, D3) + H(D3) = 1 + 1 + 1/3 = 2.33
- New cost for C2 = cost at D2 + wt(D2, C2) = 1 + 1 = 2
- New cost for D3 = cost at D2 + wt(D2, D3) = 1 + 1 = 2
- New cost for D1 = cost at D2 + wt(D2, D1) = 1 + 1 = 2
- Push (v = C2, priority = 3, cost = 2, path = [B2, D2])
- Push (v = D1, priority = 3, cost = 2, path = [B2, D2])
- Push (v = D3, priority = 2.33, cost = 2, path = [B2, D2])
- PQ = (B4, 5/3, 1, [B2]), (C2, 3, 2, [B2, D2]), (D1, 3, 2, [B2, D2]), (D3, 2.33, 2, [B2, D2])

Step 3

- Pop the node with the lowest priority, B4.
- Priority Queue, PQ = (C2, 3, 2, [B2, D2]), (D1, 3, 2, [B2, D2]), (D3, 2.33, 2, [B2, D2])
- cost = 1
- v = B4
- path = [B2, B4]
- Since B4 is not the goal state, continue.
- Neighbors of B4 = A4, B3, C4
- Priority for A4 = cost + wt(B4, A4) + H(A4) = 1 + 1 + 3/3 = 3
- Priority for B3 = cost + wt(B4, B3) + H(B3) = 1 + 1 + 3/3 = 3
- Priority for C4 = cost + wt(B4, C4) + H(C4) = 1 + 1 + 1/3 = 2.33
- New cost for A4 = cost at B4 + wt(B4, A4) = 1 + 1 = 2
- New cost for B3 = cost at B4 + wt(B4, B3) = 1 + 1 = 2
- New cost for C4 = cost at B4 + wt(B4, C4) = 1 + 1 = 2
- Push (v = A4, priority = 3, cost = 2, path = [B2, B4])
- Push (v = B3, priority = 3, cost = 2, path = [B2, B4])

- Push (v = C4, priority = 2.33, cost = 2, path = [B2, B4])
- PQ = (C2, 3, 2, [B2, D2]), (D1, 3, 2, [B2, D2]), (D3, 2.33, 2, [B2, D2]), (A4, 3, 2, [B2, B4]), (B3, 3, 2, [B2, B4]), (C4, 2.33, 2, [B2, B4])

Step 4

- Two nodes have the same priority, both D3 and C4 have 2.33 (the lowest), but following FIFO, pop D3.
- Priority Queue, PQ = (C2, 3, 2, [B2, D2]), (D1, 3, 2, [B2, D2]), (A4, 3, 2, [B2, B4]), (B3, 3, 2, [B2, B4]), (C4, 2.33, 2, [B2, B4])
- cost = 2
- v = D3
- path = [B2, D2, D3]
- Since D3 is not the goal state, continue.
- Neighbors of D3 = A3
- Priority for A3 = cost + wt(D3, A3) + H(A3) = 2 + 1 + 4/3 = 4.33
- New cost for A3 = cost at D3 + wt(D3, A3) = 2 + 1 = 3
- Push (v = A3, priority = 4.33, cost = 3, path = [B2, D2, D3])
- PQ = (C2, 3, 2, [B2, D2]), (D1, 3, 2, [B2, D2]), (A4, 3, 2, [B2, B4]), (B3, 3, 2, [B2, B4]), (C4, 2.33, 2, [B2, B4]), (A3, 4.33, 3, [B2, D2, D3])

Step 5

- Pop the node with the lowest priority, C4.
- Priority Queue, PQ = (C2, 3, 2, [B2, D2]), (D1, 3, 2, [B2, D2]), (A4, 3, 2, [B2, B4]), (B3, 3, 2, [B2, B4]), (A3, 4.33, 3, [B2, D2, D3])
- cost = 2
- v = C4
- path = [B2, B4, C4]
- Since C4 is not the goal state, continue.
- Neighbors of C4 = C1
- Priority for C1 = cost + wt(C4, C1) + H(C1) = 2 + 1 + 4/3 = 4.33
- New cost for C1 = cost at C4 + wt(C4, C1) = 2 + 1 = 3
- Push (v = C1, priority = 4.33, cost = 3, path = [B2, B4, C4])
- PQ = (C2, 3, 2, [B2, D2]), (D1, 3, 2, [B2, D2]), (A4, 3, 2, [B2, B4]), (B3, 3, 2, [B2, B4]), (A3, 4.33, 3, [B2, D2, D3]), (C1, 4.33, 3, [B2, B4, C4])

Step 6

- Pop the node with the lowest priority, C2 (FIFO).
- Priority Queue, PQ = (D1, 3, 2, [B2, D2]), (A4, 3, 2, [B2, B4]), (B3, 3, 2, [B2, B4]), (A3, 4.33, 3, [B2, D2, D3]), (C1, 4.33, 3, [B2, B4, C4])
- cost = 2
- v = C2
- path = [B2, D2]
- Since C2 is not the goal state, continue.
- Neighbors of C2 = A2, C4
- Priority for A2 = cost + wt(C2, A2) + H(A2) = 2 + 1 + 5/3 = 4.66
- Priority for C4 = cost + wt(C2, C4) + H(C4) = 2 + 1 + 1/3 = 3.33
- New cost for A2 = cost at C2 + wt(C2, A2) = 2 + 1 = 3
- New cost for C4 = cost at C2 + wt(C2, C4) = 2 + 1 = 3
- Push (v = A2, priority = 4.66, cost = 3, path = [B2, D2, C2])
- Push (v = C4, priority = 3.33, cost = 3, path = [B2, D2, C2])
- PQ = (D1, 3, 2, [B2, D2]), (A4, 3, 2, [B2, B4]), (B3, 3, 2, [B2, B4]), (A3, 4.33, 3, [B2, D2, D3]), (C1, 4.33, 3, [B2, B4, C4]), (A2, 4.66, 3, [B2, D2, C2]), (C4, 3.33, 3, [B2, D2, C2])

Step 7

- Pop the node with the lowest priority, D1 (FIFO).
- Priority Queue, PQ = (A4, 3, 2, [B2, B4]), (B3, 3, 2, [B2, B4]), (A3, 4.33, 3, [B2, D2, D3]), (C1, 4.33, 3, [B2, B4, C4]), (A2, 4.66, 3, [B2, D2, C2]), (C4, 3.33, 3, [B2, D2, C2])
- cost = 2
- v = D1
- path = [B2, D2]
- Since D1 is not the goal state, continue.
- Neighbors of D1 = C1, D2
- Priority for C1 = cost + wt(D1, C1) + H(C1) = 2 + 1 + 4/3 = 4.33
- Priority for D2 = cost + wt(D1, D2) + H(D2) = 2 + 1 + 2/3 = 3.66
- New cost for C1 = cost at D1 + wt(D1, C1) = 2 + 1 = 3
- New cost for D2 = cost at D1 + wt(D1, D2) = 2 + 1 = 3
- Push (v = C1, priority = 4.33, cost = 3, path = [B2, D2, D1])
- Push (v = D2, priority = 3.66, cost = 3, path = [B2, D2, D1])
- PQ = (A4, 3, 2, [B2, B4]), (B3, 3, 2, [B2, B4]), (A3, 4.33, 3, [B2, D2, D3]), (C1, 4.33, 3, [B2, B4, C4]), (A2, 4.66, 3, [B2, D2, C2]), (C4, 3.33, 3, [B2, D2, C2]), (C1, 4.33, 3, [B2, D2, D1]), (D2, 3.66, 3, [B2, D2, D1])

Step 8

- Pop the node with the lowest priority, A4 (FIFO).
- Priority Queue, PQ = (B3, 3, 2, [B2, B4]), (A3, 4.33, 3, [B2, D2, D3]), (C1, 4.33, 3, [B2, B4, C4]), (A2, 4.66, 3, [B2, D2, C2]), (C4, 3.33, 3, [B2, D2, C2])
- cost = 2
- v = A4
- path = [B2, B4]
- Since A4 is not the goal state, continue.
- Neighbors of A4 = A1, D4
- Priority for A1 = cost + wt(A4, A1) + H(A1) = 2 + 1 + 6/3 = 5
- Priority for D4 = cost + wt(A4, D4) + H(D4) = 2 + 1 + 0 = 3
- New cost for A1 = cost at A4 + wt(A4, A1) = 2 + 1 = 3
- New cost for D4 = cost at A4 + wt(A4, D4) = 2 + 1 = 3
- Push (v = A1, priority = 5, cost = 3, path = [B2, B4, A4])
- Push (v = D4, priority = 3, cost = 3, path = [B2, B4, A4])
- PQ = (B3, 3, 2, [B2, B4]), (A3, 4.33, 3, [B2, D2, D3]), (C1, 4.33, 3, [B2, B4, C4]), (A2, 4.66, 3, [B2, D2, C2]), (C4, 3.33, 3, [B2, D2, C2]), (C1, 4.33, 3, [B2, D2, D1]), (D2, 3.66, 3, [B2, D2, D1]), (A1, 5, 3, [B2, B4, A4]), (D4, 3, 3, [B2, B4, A4])

Step 9

- Pop the node with the lowest priority, B3 (FIFO).
- Priority Queue, PQ = (A3, 4.33, 3, [B2, D2, D3]), (C1, 4.33, 3, [B2, B4, C4]), (A2, 4.66, 3, [B2, D2, C2]), (C4, 3.33, 3, [B2, D2, C2]), (C1, 4.33, 3, [B2, D2, D1]), (D2, 3.66, 3, [B2, D2, D1]), (A1, 5, 3, [B2, B4, A4]), (D4, 3, 3, [B2, B4, A4])
- cost = 2
- v = B3
- path = [B2, B4]
- Since B3 is not the goal state, continue.
- Neighbors of B3 = B2, A3, B4, C3
- Priority for B2 = cost + wt(B3, B2) + H(B2) = 2 + 1 + 4/3 = 4.33
- Priority for A3 = cost + wt(B3, A3) + H(A3) = 2 + 1 + 4/3 = 4.33
- Priority for B4 = cost + wt(B3, A3) + H(A3) = 2 + 1 + 2/3 = 3.66
- Priority for C3 = cost + wt(B3, A3) + H(A3) = 2 + 1 + 2/3 = 3.66
- New cost for B2 = cost at B3 + wt(B3, B2) = 2 + 1 = 3

- New cost for A3 = cost at B3 + wt(B3, A3) = 2 + 1 = 3
- New cost for B4 = cost at B3 + wt(B3, B4) = 2 + 1 = 3
- New cost for C3 = cost at B3 + wt(B3, C3) = 2 + 1 = 3
- Push (v = B2, priority = 4.33, cost = 3, path = [B2, B4, B3])
- Push (v = A3, priority = 4.33, cost = 3, path = [B2, B4, B3])
- Push (v = B4, priority = 3.66, cost = 3, path = [B2, B4, B3])
- Push (v = C3, priority = 3.66, cost = 3, path = [B2, B4, B3])
- PQ = (A3, 4.33, 3, [B2, D2, D3]), (C1, 4.33, 3, [B2, B4, C4]), (A2, 4.66, 3, [B2, D2, C2]), (C4, 3.33, 3, [B2, D2, C2]), (C1, 4.33, 3, [B2, D2, D1]), (D2, 3.66, 3, [B2, D2, D1]), (A1, 5, 3, [B2, B4, A4]), (D4, 3, 3, [B2, B4, A4]), (B2, 4.33, 3, [B2, B4, B3]), (A3, 4.33, 3, [B2, B4, B3]), (B4, 3.66, 3, [B2, B4, B3]), (C3, 3.66, 3, [B2, B4, B3])

Step 10

- Pop the node with the lowest priority, D4.
- Priority Queue, PQ = (A3, 4.33, 3, [B2, D2, D3]), (C1, 4.33, 3, [B2, B4, C4]), (A2, 4.66, 3, [B2, D2, C2]), (C4, 3.33, 3, [B2, D2, C2]), (C1, 4.33, 3, [B2, D2, D1]), (D2, 3.66, 3, [B2, D2, D1]), (A1, 5, 3, [B2, B4, A4]), (B2, 4.33, 3, [B2, B4, B3]), (A3, 4.33, 3, [B2, B4, B3]), (B4, 3.66, 3, [B2, B4, B3]), (C3, 3.66, 3, [B2, B4, B3])
- cost = 3
- v = D4
- path = [B2, B4, A4]
- Since D4 is the goal state, terminate search

9. Yes, based on our class discussions/material, this problem is well-suited to local search algorithms. Local search is used for optimization problems and here we are trying to optimize the configuration of several variables to produce the best espresso. So rather than finding a specific optimal path, we are trying to optimize continuous variables (weight, size, pressure, time), creating a continuous state space, which local search handles particularly well. For example, one could use a hill climbing or gradient descent approach to iteratively improve the variables, moving towards better and better espresso over time (additionally one might use random restarts or simulated annealing to overcome local optima).

10. To implement such a program, I would try to iteratively improve the parameters (size, weight, pressure, time) using feedback, in order to move towards better espressos. For example, I could use a hill climbing approach where you start with random values for each parameter with an objective function based on something like user feedback (a 1-100 rating, for example). This algorithm could then generate neighboring states by

slightly adjusting the parameters, evaluate them using the objective function and move to the best neighbor, repeating this process until no better neighbor is found. The objective function could simply use the user rating or perhaps a weighted sum of factors like the user rating, the consistency of shots, how close parameters are to their most popular ranges, etc. To avoid the local optima problem, I could use random restarts or simulated annealing. Another approach could be using a genetic algorithm. We could start with a population full of random parameter configurations, evaluate them and select the best ones to create a new generation (breeding/combining the configurations).

11. Random restarts would be useful because the landscape, like most spaces, most likely has multiple local optima. Therefore, by randomly restarting the search from different starting points, we can increase our changes of find the real global optimum.

    Varying step size could be helpful because it allows for both broader exploration of the state space and fine tuning. For example, using larger steps initially to explore the space more broadly (finding optimal regions), then using smaller steps to narrow in on the best regions.

    Simulated annealing is also useful for this problem as it would allow the occasional bad moves early on, helping us escape potential local optima. Then as the temperature decreases, focus more on fine-tuning the best configurations found.

    Genetic algorithms could also work well. By creating a population filled with different configurations and mutating generations to improve espresso quality, we might be able to explore the space more thoroughly.

12. 1

    / \

    2 3

    / \/ \

    4 5 6 7

    / \/ \/ \/ \

    8 9 10 11 12 13 14 15

    / \/ \/ \/ \/ \/ \/ \/ \

    16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

13. It is desirable to combine iterative deepening with alpha-beta pruning from a practical standpoint for several reasons. Firstly, it helps with move ordering; iterative deepening gives info about good moves at shallower depths, which we can use to look at better moves first. Secondly, it allows for search to be stopped at any time and still return a valid result, this helps when there are time limits. Thirdly, it uses less memory than

searching full depth. Also we can change how deep we search based on how much time we have or how complicated the position is, making pruning more effective and efficient.

14. I have read and understood the academic integrity policy as outlined in the course syllabus for CS4100. By pasting this acknowledgement in my submission, I declare that all work presented here is my own, and any conceptual discussions I may have had with classmates have been fully disclosed. I declare that generative AI was not used to answer any questions in this assignment. Any use of generative AI to improve writing clarity alone is accompanied by an appendix with my original, unedited answers.