CSCI-B 551 Assignment 1 - Searching

by Jagabattula D., Shadaksharaswamy C., Kommalapati A.

Part 1 : Birds, heuristics, and A*

Let us go over the problem statement once again:

> Given five birds sitting in a random order wearing a unique number from 1 to N, write a program to rearrange the birds to be in order (e.g. 12345). In any one step, exactly one bird can exchange places with exactly one of its neighboring birds.

We can use the A* search algorithm in which we search a set of states from all the possible permutations of the birds' numbers. The abstraction of the problem is as follows:

- States: The state description specifies the current arrangement of the birds.
- Initial state: Any state can be designated as the initial state.
- Actions: Exactly one bird can exchange places with exactly one of its neighboring birds.
- Transition Model/Successor Function:
    - Generates all possible arrangements by performing one action at a time.
- Goal state: If all the birds are in order (12345).
- Action cost: Each exchange action costs 1.

A* search is Best First Search with the evaluation function $f(s) = g(s) + h(s)$, where $g(s)$ is the cost to travel from the initial state to the current state s, and the $h(s)$ is the cost that is required to travel from the current state s to the goal state.

We need to choose an admissible heuristic to ensure that A* search finds the optimal solution. In our program, we have used the following heuristic function, which is the absolute difference between the current position of the bird and its expected position:

$$h(s) = \sum_{i=1}^{5} |s_i - i|$$

The corresponding code in python:

```
def h(state):
    cost = 0
    for i in range(len(state)):
        cost += abs((state[i]) - (i+1))
    return cost
```

To ensure that the state with the lowest cost is popped, a priority queue has been implemented by sorting the python list in reverse order based on the costand popping the last element.

Part 2: The 2022 Puzzle

Going over the problem,

> A 2022 Puzzle is like the 9-puzzle but: (1) it has 25 tiles, so there are no empty spots on the board; (2) instead of moving a single tile into an open space, a move in this puzzle consists of either (a) sliding an entire row of tiles left or right one space, with the left- or right-most tile 'wrapping around' to the other side of the board, (b) sliding an entire column of tiles up or down one space, with the top- or bottom-most tile 'wrapping around' to the other side of the board, (c) rotating the outer 'ring' of tiles either clockwise or counterclockwise, or (d) rotating the inner ring either clockwise or counterclockwise. The goal of the puzzle is to find a short sequence of moves that restores the canonical configuration given an initial board configuration.

We can use the A* search algorithm in which we search a set of states from all the possible permutations of the puzzle. The abstraction of the problem is as follows:

- States: The state description specifies the current arrangement of the numbers
- Initial state: Any state can be designated as the initial state.
- Actions: The agent can perform the following 24 actions:
  - U_i where $i = \{1, 2, 3, 4, 5\}$ indicating sliding up the i^{th} column
  - D_i where $i = \{1, 2, 3, 4, 5\}$ indicating sliding down the i^{th} column
  - L_i where $i = \{1, 2, 3, 4, 5\}$ indicating sliding left the i^{th} row
  - R_i where $i = \{1, 2, 3, 4, 5\}$ indicating sliding right the i^{th} row
  - I_c, I_{cc} - Rotating the inner ring clockwise and counter-clockwise respectively
  - O_c, O_{cc} - Rotating the outer ring clockwise and counter-clockwise respectively
- Transition Model/Successor Function:
  - Generates all possible arrangements by performing one action at a time.
- Goal state: If the board is in its canonical configuration.
- Action cost: Each action costs 1.

We need to choose an admissible heuristic to ensure that A* search finds the optimal solution. In our program, we have used the following heuristic function:

$$h(s) = \sum_{i=1}^{5}\sum_{j=1}^{5} min(4 - |s_{ij}[i] - i|, |s_{ij}[i] - i|) + min(4 - |s_{ij}[j] - j|, |s_{ij}[j] - j|)$$

with the corresponding python code:

```
def h(state):
    cost = 0
    for i in range(ROWS):
        for j in range(COLS):
            x, y = math.ceil(state[i][j] / ROWS) - 1, state[i][j] % COLS - 1
            cost += min(abs(i - x), (ROWS - 1) - abs(i - x)) + min(abs(j - y), (COLS -
1) - abs(j - y))
    return cost
```

with the intuition that, in order for the heuristic to be admissible, it should not overestimate the true cost. The $min(4 - |s_i - i|, |s_i - i|)$ accounts for the wrap around case where the true cost would be lesser that the manhattan distance. Unfortunately, the heuristic does not make A* converge in less than 15 minutes. But, the heuristic works well for intermediate level boards.

> **In this problem, what is the branching factor of the search tree?**
>
> The branching factor for the search tree in this problem is 24, as there are 24 possible actions to perform on a given state, all of them valid.

> **If the solution can be reached in 7 moves, about how many states would we need to explore before we found it if we used BFS instead of A* search? A rough answer is fine.**
>
> If the solution can be reached in 7 moves, the depth of the tree is $d = 7$. Since the branching factor for the tree is $b = 24$, and the time complexity of BFS search is given by $O(b^d)$, the no. of states we need to explore before we found the solution would 24^7 states.

Part 3: Road Trip!

Going over the problem,

> On a dataset of major highway segments of the United States (and parts of southern Canada and northern Mexico), including highway names, distances, and speed limits, and a dataset of cities and towns with corresponding latitude-longitude positions, nd good driving directions between pairs of cities given by the user.

Using A* search with the following abstraction:

- State space - The set of all roads and highways present in city-gps.txt and road-segments.txt
- Initial state - The start city in city-gps.txt and road-segments.txt.
- Transition function/Successor function - Given current city/highway, returns all neighbouring cities/highways along with the distance, speed limit and the highway taken to reach from the current city/highway to the neighbouring city/highway.
- Action Cost - Assuming s is the current city, s' is a successor we have:
    - $g(s')$ for segments: the cost is 1.
    - $g(s')$ for distance: The distance from the start city to the city s'.
    - $g(s')$ for time: The ratio of distance from s to s' to the speed limit from s to s', added iteratively from the start state to s'.

- $g(s')$ for delivery: Cumulative sum of the delivery times from start city to s'.
    - An additional time is added simulating human error caused by exceeding the speed limit given by $p * 2 * (t_{road} + t_{trip})$, where $p = tanh(l/1000)$ if the speed limit ≥ 50 mph, 0 otherwise, for a road length of l.
- Goal state - Goal state is the destination city/highway.

We need to choose an admissible heuristic to ensure that A* search finds the optimal solution. In our program, we have used the the Haversine distance, given by

$$
\begin{aligned}
d &= 2r \arcsin\left(\sqrt{\operatorname{hav}(\varphi_2 - \varphi_1) + (1 - \operatorname{hav}(\varphi_1 - \varphi_2) - \operatorname{hav}(\varphi_1 + \varphi_2)) \cdot \operatorname{hav}(\lambda_2 - \lambda_1)}\right) \\
&= 2r \arcsin\left(\sqrt{\sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) + \left(1 - \sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) - \sin^2\left(\frac{\varphi_2 + \varphi_1}{2}\right)\right) \cdot \sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right) \\
&= 2r \arcsin\left(\sqrt{\sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) + \cos\varphi_1 \cdot \cos\varphi_2 \cdot \sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right)
\end{aligned}
$$

where,

φ_1, φ_2 are the latitude of point 1 and latitude of point 2, λ_1, λ_2 are the longitude of point 1 and longitude of point 2

and,

$$
\operatorname{hav}(\theta) = \sin^2\left(\frac{\theta}{2}\right) = \frac{1 - \cos(\theta)}{2}
$$

The Haversine distance is the straight line distance from between two cities, accounting for the curvature of the earth.

The corresponding python code:

```
def haversine(lat1, long1, lat2, long2):
        phi1 = lat1 * (math.pi / 180)
        phi2 = lat2 * (math.pi / 180)
        lambda1 = long1 * (math.pi / 180)
        lambda2 = long2 * (math.pi / 180)
        radius_of_earth = 3958.8

        return 2 * radius_of_earth * math.asin(
            math.sqrt(
                (math.sin((phi2 - phi1) / 2) ** 2) +
                (math.cos(phi1) * math.cos(phi2) * math.sin((lambda2 - lambda1) / 2) **
2)
            )
        )
```