Problem Set 1

Important:

- Write your name as well as your NU ID on your assignment. Please number your problems.
- Submit both results and your code.
- Give complete answers. Do not just give the final answer; instead show steps you went through to get there and explain what you are doing. Do not leave out critical intermediate steps.
- This assignment must be submitted electronically through Gradescope by January 27th 2025 (Monday) by 11:59 PM.

1 Route Planning

In route planning, the objective is to find the best way to get from point A to point B, just like in Google Maps. In this homework, we will build on top of the classic shortest path problem to allow for more powerful queries. For example, not only will you be able to explicitly ask for the shortest path from the Northeastern building to Starbucks, but you can ask for the shortest path from Northeastern back to your apartment, stopping by the gym, Subway, and the library (in any order) along the way.

We will assume that we have a map of a city (e.g., San Jose) consisting of a set of locations. Each location has:

- a unique label (e.g., 5674926221).
- a (latitude, longitude) pair specifying where the location is (e.g., 42.44219431, -25.253427).
- a set of tags which describes the type of location (e.g., amenity=food).

There are a set of connections between pairs of locations. Each connection has a distance, in meters, and can be traversed in both directions. If the distance from A to B is 100 meters, then the distance from B to A is also 100 meters.

There are two city maps that you'll be working with: a grid map (createGridMap) and a map of San Jose (createSanJoseMap), which is derived from Open Street Maps. We have also included instructions on how to create your own maps in README.md.

1.1 Grid City

Consider an infinite city consisting of locations (x, y) where x, y are integers. From each location (x, y), one can go east, west, north, or south. You start at $(0, 0)$ and want to go to (m, n), where $m, n \geq 0$. We can define the following search problem to capture this:

- $s_{start} = (0, 0)$
- $Actions(s) = \{(1, 0), (-1, 0), (0, 1), (0, -1)\}$.
- $Succ(s, a) = s + a$.
- $Cost((x, y), a) = 1 + \max(x, 0)$ (it is more expensive as x increases).

- $IsEnd(s) = \mathbb{1}[s = (m, n)]$

1. What is the minimum cost of reaching location (m, n), with $m, n \geq 0$, starting from location $(0, 0)$ in the above city? Describe one possible path achieving the minimum cost. Is it unique, i.e., are there multiple paths that achieve the minimum cost?

2. **True or False** Specify if the following statements are true or false:

   - Uniform Cost Search (UCS) will never terminate because the number of states is infinite.
   - UCS will return the minimum cost path and explore only locations between $(0, 0)$ and (m, n); that is, (x, y) such that $0 \leq x \leq m$ and $0 \leq y \leq n$.
   - UCS will return the minimum cost path and explore only locations whose past costs are less than or equal to the minimum cost from $(0, 0)$ to (m, n).

1.2 Finding Shortest Paths

We first start out with the problem of finding the shortest path from a start location (e.g., the Northeastern Building) to some end location. In Google Maps, you can only specify a specific end location (e.g., Subway).

In this problem, we want to give the user the flexibility of specifying multiple possible end locations by specifying a set of "tags" (e.g., so you can say that you want to go to any place with food versus a specific location like Subway).

1. Implement the class *ShortestPathProblem* so that given a *startLocation* and *endTag*, we can find the minimum cost path corresponding to the shortest path from startLocation to any location that has the endTag. Specifically, you need to implement,

   - *startState()* which returns an object of type State containing the location of the starting state. The memory argument can be kept as None for this problem.
   - *isEnd(state)* which returns a boolean indicating whether *state* has the endTag.
   - *successorsAndCosts(state)* which returns a list of tuples of the form:

     (successorLocation: str, successorState: State, cost: float).

   I would recommend to start by getting familiar with the whole code. Make sure to go over the class State in *util.py* and the class CityMap in *mapUtil.py* as they will be needed in this part.

   Recall the separation between search problem (modeling) and search algorithm (inference). You should focus on modeling, i.e., defining the ShortestPathProblem. The default search algorithm, UniformCostSearch (UCS), is implemented for you in *util.py*.

2. Run *python mapUtil.py > readableSanJoseMap.txt* to write a file of the possible locations on the San Jose map along with their tags. Each tag is a [key]=[value]. Here are some examples of keys:

   - landmark: Hand-defined landmarks (from data/sanjose-landmarks.json)
   - amenity: Various amenity types (e.g., "park", "food")
   - parking: Assorted parking options (e.g., "underground")

   Choose a starting location and end tag, perhaps that's relevant to your daily life, and implement getSanJoseShortestPathProblem() to create a search problem. Then, run *python grader.py 1b-custom* to generate *path.json*. Once generated, run *python visualization.py* to visualize it. It will open in your browser. Try different start locations and end tags. Pick two settings corresponding to the following:

   - A start location and end tag that produced new insight into traveling around San Jose.
   - A start location and end tag where the minimum cost path found isn't desirable. Is this due to incorrect modeling assumptions? **Hint:** Do you access to all location data point on the map?

   You can add new landmarks by following the instructions in the README.md to use your own map and landmarks.

1.3 Finding Shortest Paths with Unordered Waypoints

Let's introduce an even more powerful feature: unordered waypoints! In Google Maps, you can specify an ordered sequence of waypoints that a path must go through – for example, going from point A to point X to point Y to point B, where $[X, Y]$ are "waypoints". The points X could be a gas station whilst the point Y is maybe a friend's house.

However, we want to consider the case where the waypoints are unordered: X, Y, so that both $A \to X \to Y \to B$ and $A \to Y \to X \to B$ are allowed. Moreover, X, Y and B are each specified by a tag like in subsection 2.2 (e.g., amenity=food).

This is a nice feature if you think about your day-to-day life; you might be on your way home after a long day, but need to stop by the grocery store to grab a bite of food, the gym for a workout and the bookstore to buy some notebooks. Having the ability to get a short, quick path that hits all these stops might be really convenient, rather than searching over the various waypoint orderings yourself.

1. Implement the class *WaypointsShortestPathProblem* so that given a *startLocation*, set of *waypointTags*, and an *endTag*, the minimum cost path corresponds to the shortest path from *startLocation* to a location with the *endTag*, such that all of *waypointTags* are covered by some location in the path. Note that a single location can be used to satisfy multiple tags. Like in subsection 2.1, you need to implement,

   - *startState()* which returns an object of type State containing the location of the starting state. Think carefully about the choice of the *memory* argument. The variable *memory* must be a "Hashable" data type due to the implementation of the search algorithm which uses a dictionaries and has instances of the State class as keys. For example, it can be any non-mutable primitive (str, int, float, etc.) or a tuple.
     Hint: You can use memory to keep track of the tags you have already visited.
   - *isEnd(state)* which returns a boolean indicating whether *state* has the endTag **AND** whether all the waypointTags were visited.
   - *successorsAndCosts(state)* which returns a list of tuples of the form:

     (successorLocation: str, successorState: State, cost: float).

   There are many ways to implement this search problem, so you should think carefully about how to design your State. We want to optimize for a compact state space so that search is as efficient as possible.

2. If there are n locations and k waypoint tags, what is the maximum possible number of states given a suitable state defintion for part 1 of subsection 2.3 that UCS could visit?

3. Choose a starting location, set of waypoint tags, and an end tag (perhaps that captures an interesting route planning problem relevant to you), and implement *getSanJoseWaypointsShortestPathProblem()* to create a search problem. Then, similar to part 2 of subsection 2.2, run *python grader.py 2c-custom* to generate *path.json*. Once generated, run *python visualization.py* to visualize it which will open in browser once again.

1.4 Speeding up Search with A*

We will now explore how to speed up search by reducing the number of states that need to be expanded using some A^* heuristics. In class, we mentioned that the A^* algorithm is identical to UCS expect that A^* uses the sum $g(n) + h(n)$ whilst UCS uses $g(n)$ solely, where $g(n)$ gives the path cost from the start node to node n and $h(n)$ is a heuristic function which estimates the cost of the cheapest path from node n to the goal. We will implement A^* via a reduction to UCS.

1. Implement aStarReduction() which takes a search problem *problem* and a heuristic function *heuristic* as input and returns a new search problem. Specifically, you need to implement:

- *startState()* which returns an object of type State containing the location of the starting state. Note that this will not change from the original problem.
- *isEnd(state)* which returns a boolean indicating whether *state* has the endTag. Note that this will not change from the original problem.
- *successorsAndCosts(state)* which returns a list of tuples of the form:

  (successorLocation: str, successorState: State, cost: float).

  You can also use the output of this function for the original problem and modify the cost element to also account for the heuristic function.

2. We will now develop a straight line distance from a state to any location with the end tag. Implement StraightLineHeuristic by filling:

- the constructor *__init__(self, endTag, cityMap)* in which you can find all the Geolocations associated with endTag.
- *evaluate(state)* in which you will return the minimum distance from *state* to a location with the end tag. Note that you can use the function *computeDistance* in *mapUtil.py* which already find the straight line distance between two geolocations.