Assignment-1 Problem solving agents

Your objective is to implement SIMPLE-PROBLEM-SOLVING-AGENT(s) based on the pseudocode in **Figure 3.1 of AIMA 3rd edition book chapter 3 page 67**. Please read and understand the pseudocode before proceeding further. **Your solution should match the pseudocode.**

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: seq, an action sequence, initially empty
              state, some description of the current world state
              goal, a goal, initially null
              problem, a problem formulation

  state ← UPDATE-STATE(state, percept)
  if seq is empty then
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
      if seq = failure then return a null action
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

There are two problems in this assignment. For implementing the solutions, please do the following:

1. Hardcode the *goal* (goal-test), instead of automatically formulating it based on a performance measure. The best way to hardcode the goal is as follows:
    a. Write a function (python) to take as input a node and test if the state represented by the node passes the goal conditions.
    b. Pass this python function as an argument *(yes! In python functions can be passed as arguments)* to the SEARCH method along with other arguments such as the initial-sate, precondition-action-effect and path-cost.
2. The *SEARCH* method should take in an additional argument "search_type" which could take values from the set { BFS, DFS, BDBFS, ASTAR, etc}. This argument determines which algorithm will be used for the search.
3. Just printing the action-sequence as output is sufficient. Make sure to print the action sequence in way that is easier to understand and evaluate.

Careful planning will allow you to implement a general solution and reduce the total amount of code you have to write. We will measure how much of the code has been reused/shared between the two problems below and incentivize accordingly.

Problem-1: Path finding

For this problem, you need to build a basic taxi-agent. The search algorithm(s) should find path between any two cities (provided as input). Implement the following search algorithms:

1. BFS
2. DFS

3. Bi-directional BFS
4. A*

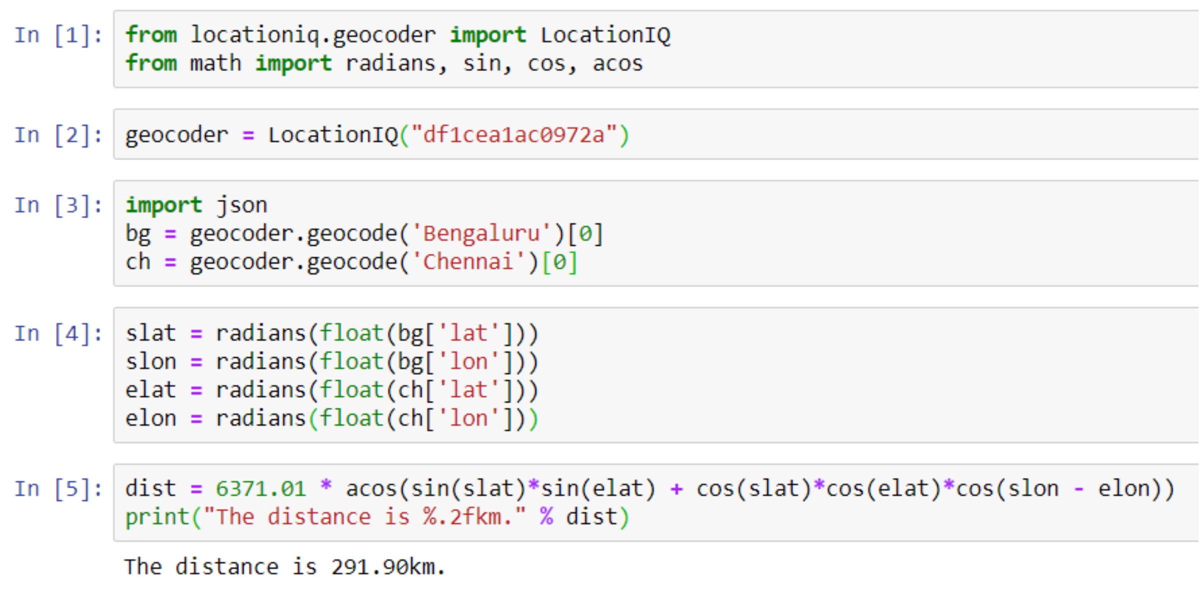
The attached dataset has driving distance between all adjacent Indian capitals (Example: Chennai is the capital of Tamilnadu. Tamilnadu has three adjacent states: Kerala, Karnataka and Andhra Pradesh. So, the dataset will have three rows indicating edges between Chennai<->Bengaluru, Chennai<->Trivandrum, Chennai<->Amaravati) **Think of each row as representing a weighted edge in a graph**, **where the capitals are nodes/vertices and distances are the weights**.

For A* you need the straight-line distance. There are quite a few geocoding APIs that can give you latitude and longitude of any city. I used one such API called locationIQ.

https://locationiq.com/

https://pypi.org/project/locationiq/

The code below shows example of using the API to find the straight-line distance between Chennai and Bengaluru. You can get the key directly from the locationiq website.

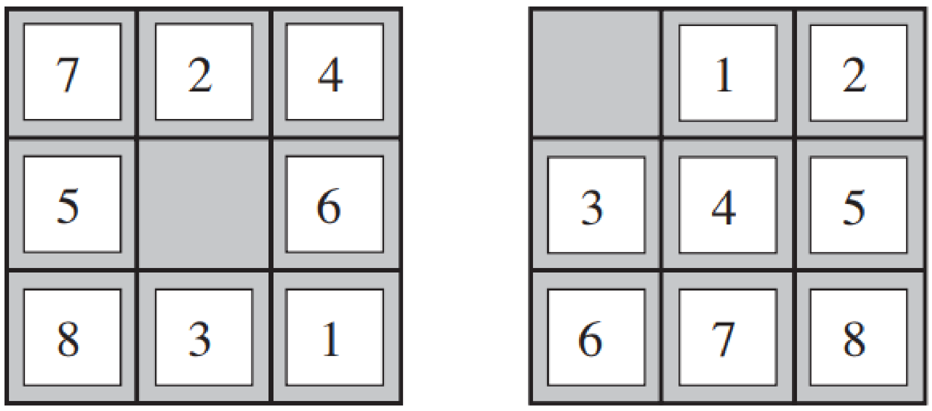


```
In [1]: from locationiq.geocoder import LocationIQ
        from math import radians, sin, cos, acos

In [2]: geocoder = LocationIQ("df1cea1ac0972a")

In [3]: import json
        bg = geocoder.geocode('Bengaluru')[0]
        ch = geocoder.geocode('Chennai')[0]

In [4]: slat = radians(float(bg['lat']))
        slon = radians(float(bg['lon']))
        elat = radians(float(ch['lat']))
        elon = radians(float(ch['lon']))

In [5]: dist = 6371.01 * acos(sin(slat)*sin(elat) + cos(slat)*cos(elat)*cos(slon - elon))
        print("The distance is %.2fkm." % dist)

        The distance is 291.90km.
```

Feel free to use any API of your choice. There might be some API which can directly give you the straight-line distance. But, one important condition. **We will not create new keys (or do any extra work). Your code should work straight out of the box**.

Problem-2: Eight Puzzle

N-Puzzle or sliding puzzle is a popular puzzle that consists of N tiles where N can be 8, 15, 24 and so on. In our example **N = 8 (Hence called Eight Puzzle)**. The Eight Puzzle problem will have 3 rows and 3 columns. See example below



Start State Goal State

The puzzle consists of 8 tiles and one empty space where the tiles can be moved. Start and Goal configurations (also called state) of the puzzle will be provided. The puzzle can be solved by moving the tiles one by one in the single empty space and thus achieving the Goal configuration. You have to implement an agent that can take in initial state and the goal. The search algorithm(s) should find the sequence of actions that can achieve the goal. Implement **BFS, DFS, Greedy Best First Search and A***. Come up with your **own heuristic function** to solve the problem.

Please remember, the above image is just an example and so don't hardcode it. Your solution should work for any 8-puzzle problem.

Warning: We will select 20 individuals (either randomly or those who are suspected of plagiarism) and conduct a surprise test. The test will be related to the Assignment. It will be in a way to tease out whether the individual did the assignment on their own or copied. If the individual fails the test, they will receive a zero for the assignment. In addition, they are automatically chosen for the test that will be conducted for assignment 2. We also reserve the right to forward your case to the academic disciplinary committee. So, proceed with caution!