CS 335: Lab Assignment 7

(TAs in charge: Huzefa Chasmai, Vishwajeet Singh)

Acknowledgement: This lab assignment takes inspiration from Solving Every Sudoku Puzzle, which is an essay by Peter Norvig on how to solve Sudoku. The code is partially taken from the website. We thank Peter Norvig for making the project available to the public. Also OSMnx module developed by Geoff Boeing which made it possible to get geographic and street network of Mumbai alongwith stunning graphs. We thank him for making such an awesome library and open-sourcing it.

Code

The base code for this assignment is available in this zip file. The following files are the most relevant ones for you; you will only have to edit `sudoku.py` and `maps.py`.

File Name	Description
sudoku.py	Where your logic for converting Sudoku into a **search** problem resides.
maps.py	Where your logic for converting a 2-D map into a **search** problem resides.
search.py	Your search algorithms will reside here.
main.py	The main file which actually runs the search algorithms on the given tasks.
util.py	Useful data structures for implementing search algorithms.

Task 0: Understanding the Code Base (Ungraded)

In `sudoku.py` and `maps.py`, you will find a skeleton code which you need to fill, these convert the abstract problems into fully defined search problems. The search algorithms need to go in `search.py` which will actually solve the search instances.

You need to get familiar with NetworkX module. It makes it easier to manipulate complex graphs like maps.

Now you will implement different search algorithms to solve interesting problems like Sudoku and How to get to Marine Drive in the shortest path possible if you decide to cycle there (That's quite a mouthful, why don't we just call it Shortest Path Problem). Remember that a search node must contain not only a state but also the information necessary to reconstruct the path which gets to that state from the start state.

Important note: Some of your search functions need to return a list of *nodes* that will lead you from the start to the goal.

Important note: Make sure to **use** the `Stack`, and `PriorityQueue` data structures provided to you in `util.py`! These data structure implementations have particular properties that are required for compatibility with the autograder.

Hint: The algorithms are quite similar. Algorithms for DFS and A* differ only in the details of how the fringe (or *frontier*) is managed. So, concentrate on getting DFS right and the rest should be relatively straightforward. Indeed, one possible implementation requires only a single generic search method which is configured with an algorithm-specific queuing strategy. (Your implementation need *not* be of this form to receive full credit.)

Section 1 : Sudoku Problem (6 Marks)

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

(Image source: https://en.wikipedia.org/wiki/Sudoku#/media/File:Sudoku_Puzzle_by_L2G-20050714_standardized_layout.svg)

Task 0 : Understanding the code base for Sudoku Problem

General Structure of the Problem

We will be solving the 9 x 9 sudoku problem where each square in the grid can be filled with a value between 0-9. The squares are named according to the following convention : row || col, where row is an alphabet 'A-I' and column is a digit '1-9' and || stands for concatenation. The resulting 81 squares are as follows : `['A1',`

`'A2', .... , 'I8', 'I9']`. We maintain a dictionary of possible values that each square can take. So the dict called values has the following form : `values = {'A1' : '1278', 'A2' : '1', ... , 'I8' : '2673', 'I9' : '12'}` . Basically, the keys are the squares and the values are the possible digits that can be in that square based on the current conditions. This is your state.
Now the sudoku solution has the following 3 constraints : 1) Each row must contain the set(1-9) exactly, 2) each column must contain the set(1-9) exactly and 3) Each of the 3 x 3 square formed by dividing the 9 x 9 grid in 3 parts both by column and row must contain the set(1-9) exactly. Each of the squares belonging to a unit are mentioned in the unit list and further a dict is created between squares and all the units that this square belongs to. We also define peers to be the squares that might be directly affected by fixing/changing values for a particular square.
Now we take a grid as an input in the form of a string of values. Many sudoku problems are provided in the files : `'data/sudoku/hardest.txt'` and `'data/sudoku/top95.txt'`. We encourage you to solve at least one of the hardest ones yourself to appreciate the algorithm. Go through the parse_grid function and the grid values function to visualise the sudoku problem as a grid. Use the display function to display the **`values`** dictionary in a format suitable for visualisation.

Constraint Propagation

The function **`parse_grid`** calls **`assign(values, s, d)`**. We could implement this as **`values[s] = d`**, but we can do more than just that. Those with experience solving Sudoku puzzles know that there are two important strategies that we can use to make progress towards filling in all the squares:

> *(1) If a square has only one possible value, then eliminate that value from the square's peers.*
> *(2) If a unit has only one possible place for a value, then put the value there.*

It turns out that the fundamental operation is not assigning a value, but rather eliminating one of the possible values for a square, which is implemented with **`eliminate(values, s, d)`**. Once we have **`eliminate`**, then **`assign(values, s, d)`** can be defined as "eliminate all the values from **`s`** except `d`".

Note:

(1) This is done recursively so each step of assignment might end up changing multiple squares possible values.
(2) Also note that the values is python dict object and python objects are passed by reference by default, so changing the value of the dict within a function changes the values everywhere the dict is being accessed.
(3) Note that both assign and elinate include the check of whether

the constraints are satisfied and if it not being satisfied then they return False in place of values. A good practice would be to ensure that the return type is not False before performing operations on it.
(4) Go through the link Solving Every Sudoku Puzzle for details about the environment and the algorithm.

Task 1 : Formulating the Sudoku Search Problem : (3 Marks)

You need to formulate the Sudoku problem as a Search Problem and in turn implement the following functionalities in the **`sudoku.py`** file under the **`SudokuSearchProblem`** class.

1. getInitialState() : returns the initial state of your search problem
2. isGoalState(state) : returns True if input state is one of the Goal States or returns False.
3. getSuccessors(state) : returns successors as **`list(tuple(nextState, Action, EdgeCost))`**

Run your code with the following command. This serves as both running the script as well as autograder

```
python2 main.py -t 1
```

These functions are according to the definition of a Search problem presented in Section 3.1, 3.2, Russell and Norvig (2010). You have the freedom to decide the edge cost, but keep in mind that DFS doesn't really look into edge cost explicitly. Your actions would be tuples of the form (square, digit).

Key Idea Minimum Remaining Values :

The key idea in this is that the action we take at each step. I.e. changing from one state (values) to another is done by chosing a digit to place in one of the squares having minimum remaining values greater than 1. So essentially we choose the square with minimum **`values[s]`** and our action at state "**`values`**" are filling `s` with `[d for d in values[s]]`. i.e. actions from states = values is **`[(s, d) for d in values[s]]`**, here **`s = square with minimum {values(s) > 1}`**. The reason for doing this is elaborated in the section search of the above link.

Task 2 : Implementing the DFS Algorithm : (2 Marks)

You need to implement the DFS algorithm in the **`depthFirstSearch`** function of the **`search.py`** file.

Run your code with the following command.

```
python2 main.py -t 2
```

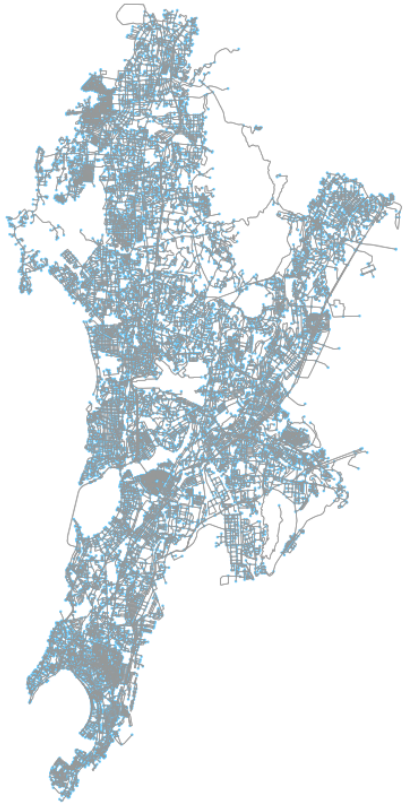
Reading :

For the definition and use of Node class refer to Section 3.3, and for DFS to Section 3.4, Russell and Norvig (2010).

Task 3 : Why DFS Algorithm? Can we perform informed search? : (1 Mark)

Can we implement informed search techniques like A star for this Problem? Answer with respect to both path cost and heuristic. Write your explanation in **`answers.txt`**.

Section 2: Shortest Path Problem (9 Marks)



In this task we are going to answer what is the optimal path from Hostel 4 to Lecture Hall Complex. This is a confounding problem due to the myriads of paths possible from Hostel 4. We hope that this helps all the sleepy people in making a better decision in the morning and reach the Lectures on time.

We have also been informed that the lab takes most of your time on the weekends and you have not been able to explore Mumbai. So we also take you on a virtual tour of Mumbai. You would be travelling but obviously you are not a salesman.

General Structure of the Problem

The maps would be a **NetworkX Graph G** instance with main landmarks as nodes(**V**) and roads or paths as edges(**E**). For all the nodes in **V**, latitude and longitude are provided as attributes. Path length are the attribute for all the edges in **E**. Some functions and accessing methods of **G** which you are highly advised to use are G.nodes(), G.neighbors(node), G.node[node1](gives the attributes of node1) G[node1][node2](gives the attributes of edge between node1 and node2).

Task 4 : Formulating the Map Shortest Path Search Problem : (3 Marks)

You need to formulate the Shortest Path Search problem as a Search Problem and in turn implement the following functionalities in the **`maps.py`** file under the **`MapSearchProblem`** class.

1. getInitialState() : returns the initial state of your search problem
2. isGoalState(state) : returns True if input state is the Goal State else returns False.
3. getSuccessors(state) : returns successors as **`list(tuple(nextState, Action, EdgeCost))`**

Run your code with the following command.

```
python2 main.py -t 4
```

These functions are according to the definition of a Search problem presented in Section 3.1, 3.2, Russell and Norvig (2010).

Task 5 : Implementing the A star Search Algorithm : (2 Marks)

(Image source: https://vignette.wikia.nocookie.net/main-cast/images/6/6f/Star.jpg/revision/latest?cb=20151213212737)

You need to implement the A star Search Algorithm algorithm in the **`AStar_search`** function of the **`search.py`** file. At the end your function should return a route as list of nodes in order, consisting of both the start and end node.

Run your code with the following command.

```
python2 main.py -t 5 --show
```

Reading :

For the definition and use of Node class refer to Section 3.3, and for AStar_search to Section 3.5, Russell and Norvig (2010).

Task 6 : Implementing the Heuristic Function : (2 Marks)

You need to implement a **Consistent Heuristic** in the **`heuristic`** function of the **`search.py`** file. We have provided some useful function in the **`util.py`** file to calculate the distance between 2 points given their latitude and longitude.

Run your code with the following command.

```
python2 main.py -t 6 --show
```

Reading :

For information on Heuristics refer to Section 3.6, Russell and Norvig (2010).

Task 7 : Why A* Algorithm? Possible Heuristic for Travelling Student Problem: (2 Mark)

- What would have been the number of nodes visited in case we used a simple Shortest Path Problem Algorithm like Dijikstra's?
- Also in case the problem would have required you to reach multiple nodes what possible heuristic can you come up with for A*?

Write your solutions in **`answers.txt`** for both the problems.

Submission

You are expected to work on this assignment by yourself. You may not consult with your classmates or anybody else about their solutions. You are also not to look at solutions to this assignment or related ones on the Internet. You are allowed to use resources on the Internet for programming (say to understand a particular command or a data structure), and also to understand concepts (so a Wikipedia page or someone's lecture notes or a textbook can certainly be consulted). However, you **must** list every resource you have consulted or used in a file named `references.txt`, explaining exactly how the resource was used. Failure to list all your sources will be considered an academic violation.

Be sure to write all the observations/explanations in the **`answers.txt`** file. We have mentioned wherever there is such a requirement. Find the keyword '**`answers.txt`**' in the page.

Place all files in which you have written code in or modified in a directory named `la7-rollno`, where `rollno` is your roll number (say 12345678). Tar and Gzip the directory to produce a single compressed file (say `la7-12345678.tar.gz`). It must contain the following files.

1. `sudoku.py`
2. `maps.py`
3. `search.py`
4. `references.txt`
5. `answers.txt`

Submit this compressed file on Moodle, under Lab Assignment 7.