Assignment : Search and Planning
AI QSTP 2015

by Srinath, 2012B5A7560G

Part 1 - Theory

Q1) The objective of the following questions is to strengthen your ability to formulate a search problem. Search problems need not always involve position coordinates and the conventional "path-finding". Problems which doesn't seem like "search and planning" at first glance might actually be a typical search problem which you should be able to *model* it correctly so that we can use BFS/DFS/UFS/A* to solve it. (CREDITS - U.C.Berkeley AI Course)

.

Outline 1

Pacman is visiting San Francisco and decides to visit N different landmarks $\{L_1, L_2, \ldots, L_N\}$. Pacman starts at L_1, which can be considered visited, and it takes t_{ij} minutes to travel from L_i to L_j.

Find the **correct minimal state space** representation for the following scenarios.

a. Pacman would like to find a route that visits all landmarks while minimizing the total travel time.

b. Ghosts have invaded San Francisco! If Pacman travels from **Li to Lj**, he will encounter g_{ij} ghosts. Pacman wants to find a route which minimizes total travel time *without encountering more than* $\mathbf{G_{max}}$ *ghosts* (while still visiting all landmarks).

c. The ghosts are gone, but now Pacman has brought all of his friends to take pictures of all the landmarks. Pacman would like to find routes for him and each of his k−1 friends such that *all landmarks are visited by at least one individual*, while minimizing the **sum of the tour times** of all individuals. You may assume that Pacman and all his friends start at landmark L1 and each travel independently at the same speed. Formulate this as a search problem and find the correct minimal state space representation.

d. Pacman would now like to find routes for him and each of his k−1 friends such that all landmarks are still visited by at least one individual, but now minimizing the maximum tour time of any individual. Formulate this as a search problem and find the correct minimal state space representation.

<u>Outline 2</u>
<u>Hive Minds </u>- There is a maze. An insect is in the maze. It is night time. You can't see the insect but you have to get it to its goal position. You don't know where the insect is BUT you know the maze completely and YOU KNOW where the goal is. The insect doesn't know shit. All it can do is execute a set of actions given by you at the starting. The objective is to give the insect an *optimal* set of actions which if the insect executes *mindlessly, will definitely take it to the goal position irrespective of what its starting position is (which you wouldn't know).* You can treat the maze as some M x N grid with blocks in between. The insect can move up, down, left, right. If it executes any one of these actions and there is a block in that direction, it will just stay in the same place.

a. Formulate this as a search problem. Find the minimal correct space representation. Note that the objective is to build an *universal plan (sequence of moves)* such that after executing all these moves, the insect is at the goal state. Of course, you won't be able to verify that the insect has reached the goal state because you don't know where the insect is. You have to come up a state space model such that this problem can be solved with the given constraints.
b. Suggest a good admissible heuristic for the search problem. You have to give a valid argument why your heuristic is admissible.

Q2) We moved from TREE-SEARCH to GRAPH-SEARCH because of the redundancy in the former. But, we observed that A* search failed to be produce optimal results when we moved to GRAPH-SEARCH. This forced us to make our heuristic consistent in addition to being admissible.

a. Give a formal argument on why BFS/UFS produces optimal results in GRAPH-SEARCH. Why doesn't the same argument hold for A*?
b. Give a formal argument as to why A* (with admissible and consistent heuristics), for EVERY state S (not just goal state), expands nodes that reach S optimally BEFORE nodes that reach S sub-optimally.
c. I am a lazy bum. I am not going to work to produce consistent heuristics. How can I get away with JUST admissible heuristics and STILL use A* Graph Search to get *optimal* results.

__Part 2 - Programming__

Q3) In this part, you are going to implement a **general Python library** (kind of) for **DFS, BFS, UFS, and A*. Strictly adhere** to the format prescribed below.

1) Create a file **search.py**. In the file, create an **abstract** (google it if you don't know) class **SearchProblem**. This class provides **a general template** for an arbitrary search problem. Note that you SHOULD NOT implement any methods of this class. This class has the following methods
   a) **getStartState(self)** - returns state for the search problem
   b) **isGoalState(self, state) -** boolean checking whether input state is goal state or not.
   c) **getSuccessors(self, state) -** returns a list of triplets (nextState, action, stepCost)
   d) **getCostOfActions(self, actions)** - *actions* is a list of valid actions. This method returns the total cost of actions.
2) In the same file, implement global methods **depthFirstSearch, breadthFirstSearch, uniformCostSearch, aStarSearch**. The first three takes a **problem** as an input. This **problem** is an instance of a class which *inherits and implements the methods of* **SearchProblem** class. The **problem** object contains all the machinery required to model a search problem. Once you give it to the corresponding search algorithm - it is going to return a **list of legal actions** required to solve the problem (that is, to reach the goal state from start state). Note that A*, in addition to taking a **problem** object as input, also takes a function **heuristic** as an input. **heuristic** takes a **state** object as input and returns a real number.

You should realize the point of doing this. You now have general purpose search algorithms which does not depend on any specific problem. So when you encounter a search problem you will first create a **specificproblem.py** file where you will import the **search.py** file you had just implemented. In **specificproblem.py** you will have to create **specificProblemState** class and **specificSearchProblem** class (this inherits the **SearchProblem** class and implements its methods).

specificProblemState class *captures and models* the state space representation of your search problem. This depends on your specific problem, so what inputs and methods you implement as a part of this can vary. Generally, have a **isGoal** function, **legalMoves** function (returns a list of legal moves from current state). Note that **legalMoves** should not take any inputs (like a **state** object or anything). This is because

legalMoves is a method of some **state** object. So the input is kind of “built-in”. An object of **speicifcProblemState** represents a **state** of the problem. Also have a **result** function which takes an **action** as an argument and returns a new **state** obtained by acting **action** on the current state.

specificSearchProblem inherits **SearchProblem** class you defined in **search.py** and implements its methods. This class should definitely have the problem’s **state** (an object of **specificProblemState** class) as its attribute. In addition to this, you can take a cost function as input if your search problem has weighted actions.

Q4) <u>Solving the Eight Puzzle</u>

Once you are done with the above question, use the guidelines suggested above to solve the Eight Puzzle problem. So you have to create a **eightpuzzle.py** file, implement a **EightPuzzleState** class to model the state space representation and **EightPuzzleSearchProblem** class to formulate it as a search problem. Then have a main function in the file, in which you create the appropriate objects and use an appropriate search algorithm to solve it **optimally**. The output while running “**python eightpuzzle.py**”, should show the starting puzzle, show the set of actions one by one with the corresponding eight puzzle state. Basically you should show how the starting eight puzzle **evolves** into the final state. If you are this yoyo-python-ninja, you can do all this in a fancy GUI way (using pygame or tkinter), but it is not required. You can present the output in a **neat** command line format.

Q5) <u>A* Search and Designing Heuristics (CREDITS - Tushar Nagarjan)</u>

In this problem, you are given a MxN rectangular maze. The legend is as follows:

- **s**: start position
- **t**: treasure
- **w**: wall
- **.**: land

There are multiple treasures in the land. Your goal is to collect all the treasures in an optimal way. Main objective of this question is for you to design good admissible and consistent heuristics which will minimize the total number of nodes expanded. You can keep track of number of nodes expanded by incrementing whenever you call **getSuccessors** function in your search algorithm.
Your lower baseline is the trivial heuristic (which returns 0 always). You should be able to do MUCH better even with very naïve heuristics.

The maze input is given in a file called “m.txt”. The code to parse the input is already given in **treasurehunt.py.** You have to define the state space class, search problem class (as you did in Q4), and import **search** and invoke **aStarSearch**. (Note - I asked you to find the expanded nodes by incrementing the number of nodes expanded whenever **getSuccessors** function is being called. How will you do this if **aStarSearch** function is in another file? Figure out a work around. If you are not able to do this, you can just copy **aStarSearch** function into your current file and plug in your expanded nodes logic there.)

Define multiple **heuristic** functions in **treasurehunts.py** and try to minimize the number of nodes expanded as much as possible. Make sure that your heuristic functions are admissible and consistent. Only then you will get optimal result. Optimal result = Minimal cost of travelling the maze. You can assume uniform step cost of 1 throughout.