50.021 -AI

Alex

Week 08: Search

[The following notes are compiled from various sources such as textbooks, lecture materials, Web resources and are shared for academic purposes only, intended for use by students registered for a specific course. In the interest of brevity, every source is not cited. The compiler of these notes gratefully acknowledges all such sources.]

USA Maps (aka When You Wish Upon A)*

This exercise and the two that follow it require the use of a code distribution. The code distribution may be downloaded from edimension. You should copy these files to the directory you created last week that includes search.py

Preparation

We will use search algorithms to solve a larger problem than we have considered thus far: planning routes for driving across the USA. This lab will make use of publicly-available highway data from the U.S. Federal Highway Administration `http://www.fhwa.dot.gov/planning/processes/tools/nhpn/`. Our Python representation for these data consists of two main classes (defined in highways.py and imported into maps.py):

- `Location`: An instance of this class represents a single point on a map. Each instance includes a unique identifier id_number, and latitude and longitude coordinates (`latitude` and `longitude`, respectively). Instances also contain an attribute `name`, which contains a (more-or-less) human-readable string identifier. Some locations are not named, the U.S. is a big place with some quite lonely places; for these locations, `name` will contain None.
- `Link`: An instance of this class represents a bidirectional highway between two locations. Each instance contains attributes `begin` and `end`, which hold instances of Location representing the two ends of the link. Some instances also contain an attribute `name`, which contains a (more-or-less) human-readable string identifier.

The following structures and procedures are also defined in highways.py and imported into maps.py:

- `location_from_ID`: a dictionary that maps each location's unique id number to the corresponding instance of Location

- `locations_from_name`: a dictionary that maps the name of a location to the corresponding instance of Location
- links: a list containing each Link in the database.
- `distance(loc1, loc2)`: a function that takes two ID numbers and returns an estimate of the distance between the locations they represent, in miles, incorporating an approximation for the curvature of the earth. This will serve as our cost function.
- `locations_matching_name(state, name)`: a method that takes two strings as input, and returns the names and ids of matching locations as a list of tuples. For example, `locations_matching_name('IL','PRINCETON')` returns [('ILPRINCETON C-N', 17001075), ('ILPRINCETON E', 17001089), ('ILPRINCETON', 17001088)].
- `neighbors`: a dictionary mapping each location's ID to a list containing the ID's of that location's neighbors (the locations to which it is connected by a single link).

Part 1: Representation

Use the methods and variables from maps.py and highways.py to answer the following. Decimal answers must be accurate to four decimal places.

1.1 Scale

- How many locations exist in the database?
- How many unique location names exist in the database?
- How many links exist in the database?

1.2 Data

- What is the ID number of the location whose name is 'ILPEORIA CBD'?
- What is the name of the location whose ID number is 8000302
- What elements are in the Python list containing the IDs of the neighbors of location 23000331 ?

Part 2: Search

Now we want to use the search procedures to find paths in the map.

Look at the code for the `uniform_cost_search` function, and make sure you understand it before moving forward.

As a test, we will use `uniform_cost_search` to find a path from a place near the geographic center of the U.S. (Smith Center, KS; ID number 20000071), to Cambridge (ID number 25000502).

Run the search. Hints:

- check bidirectional.py . It contains an instance of the `UndirectedGraph` class named `usa`
- you can create an instance of `GraphProblem` class from the `UndirectedGraph` named `usa`, and additionally startpoint and endpoint. `GraphProblem` is a derived class of the class `Problem`, and you can use any search algorithm to run it.

Viewing Your Paths

Looking at lists of ID numbers isn't very informative. As an alternative, you can view your paths in Google Earth or Google Maps to get an idea of how good your results are.

highways.py contains a function `to_kml(path)` which takes a list of node IDs as input, and creates a file path.kml. This kml file can be read into Google Maps (google my maps → Import) for many browsers or Google Earth for chrome browser to visualize the path you found.

You can view your path by uploading the KML file to:

- Google Maps (google my maps → Import)
- Google Earth for chrome browser
- `http://veloroutes.org/` (have to click on satellite and back to map to refresh the map)

Part 3: Path Cost

We also want an easy way to compute the total cost of a path. Write a function `path_cost` that takes a list of id numbers as input, and returns the total cost of that path in miles (the sum of the distances between locations). You can use the function `def distance(id1, id2)` from highways.py !

Answer these questions about `path_cost`.

- What is the cost of the path: [17001088, 17001089, 17001106, 17001147, 17001168], in miles?
- What is the cost of the path returned by your search (Smith Center, KS; ID number 20000071), to Cambridge (ID number 25000502), in miles?

Part 4: Evaluation

We will now try running uniform cost search in a few different trials to try to get a feel for how it performs in different circumstances.

Run your search through the following scenarios (from the indicated city to Cambridge (25000502), and make note of the total cost of the path found, as well as the number of states expanded, in each case.

Ask yourself for the following start points which are given below:

- How many states were expanded by BFS?
- What was the total cost of the path found with BFS?
- How many states were expanded by uniform cost search with no heuristic?
- What was the total cost of the path found with no heuristic?
- BFS and Uniform Cost Search are each supposed to return optimal paths, yet the paths they return are different. How do the paths returned from BFS and Uniform Cost Search differ? Why?

The start points are given as:

- Pittsfield, MA (25000379)
- Altoona, PA (ID 42001780)
- Smith Center, KS (ID 20000071)

Part 5: Heuristics

In uniform-cost search, we select the next node to expand that minimizes $F(n) = path\_cost(n)$. In this section, we will consider A* (uniform cost search with a heuristic function), in which we select the node that minimizes $F(n) = path\_cost(n) + H(n)$, where H is a heuristic function that estimates the distance from n to the goal. Repeat your specific searches from the previous section, this time using the geodesic distance between the current state and the goal (which can be computed with the distance function) as a heuristic. Answer these questions about running the search (to Cambridge) with a heuristic.

Ask yourself for the following start points which are given below:

- How many states were expanded by A* search with distance to goal heuristic?
- What was the total cost of the path found by A* search with distance to goal as heuristic?

- Pittsfield, MA (25000379)
- Altoona, PA (ID 42001780)
- Smith Center, KS (ID 20000071)
- how does the heuristic affect the cost of the returned paths?
- how does the heuristic affect the effectiveness of the search?

Coding! - BiDirectional Graph BFS Search

BiDirectional Graph BFS Search is a pair of Graph BFS searches - one tree starts from the start state and one tree starts from the goal state – until they meet. Thus you maintain two frontiers and two explored sets. In every iteration you run:

- one expansion-insert step for the tree originating at the goal,
- and one expansion-insert step for the tree originating at the start.

You have found a solution when you expand a node and one of the children that are created in the expansion are in the frontier of the other search tree, that is when the fringes meet. Note this change to goal tests :) .

Why is that cool? If originally you would find a solution at depth d, then you would find a solution with two trees of depth $d/2$. Saves times and space :) as you know $2^{10/2} + 2^{10/2} = 64 \neq 2^{10} = 1024$.

Bidirectional Search is a good example for the tradeoff between different heuristics: a simple heuristic is computing the distance between the node and the root of the other tree. This is a no very tight but fast to compute heuristic.

One can compute a more precise heuristic, namely the distance between the node and the fringe of the other tree – but that heuristic can get very quickly very expensive, as the fringe grows exponentially.

Some things you need to keep in mind:

- Look at the definition of the class Node in search.py, in particular, the definition of the path method.
- Note the difference between frontier and explored.
- We have provided you a version of FIFOQueue, called MyFIFOQueue, that allows you to get the node corresponding to a given state (the getNode method).

Tasks:

- implement it. run a first test: Smith Center (location ID 20000071 in the USA graph) to Cambridge (location ID 25000502 in the USA map).
- Run test_map and compare the number of nodes expanded and the running time using BFS, Uniform Cost, A*, and bi-directional search for a search from Smith Center (location ID 20000071 in the USA graph) to Cambridge (location ID 25000502 in the USA map).
- Are the paths found by bi-directional (breadth-first) search guaranteed to be identical to those found by normal (single direction) breadth-first search?

...