CS 338 – Activity 2 – Map App

Prof. Andrew Shallue

September 5, 2024

1 Setup

Take a look at the starter code for Activity 2. The environment in this case is the class `Map`, which implements a partial map of cities in Illinois via a sparse graph representation. The graph is a list of `City` objects. Each `City` has two pieces, a name and a list of adjacent cities. The adjacency list is actually a list of pairs, where the first part of the pair is the name of the adjacent city and the second part of the pair is the distance to the adjacent city.

The `Agent` keeps track of the name of its current location, along with the distance it has moved. It is a problem-solving agent, where its goal is to reach some other city using the shortest possible path. To do this it first uses search to craft a plan, then it executes the plan by moving along adjacent cities and updating the distance it has traveled.

I have implemented the agent's actions for you. Your goal in this activity is write search algorithms that create the plan.

Here is a PEAS description of the environment:

Performance measure	Environment	Actuators	Sensors
shortest distance	graph model of IL	move to adjacent	see adjacent cities

2 Search by hand

Pull up the map of IL I created. In a breadth-first search, we first touch the cities adjacent to the start city, then the cities at level 2, then the cities at level 3, and so on.

By hand, perform a breadth-first search that gets us from Rock Island to Cairo. What is the resulting list of cities?

In a depth-first search, we pick an ordering of the neighbors, and then follow the same direction as far as it can go, without repeating any of the nodes we have already seen.

Pick an ordering, then perform a depth-first search by hand that gets us from Rock Island to Cairo. Note that this is heavily dependent on ordering, so there is no one right answer.

3 Fixing a bug

I have given you a first search function that uses `queue.Queue` to implement breadth-first-search. The queue stores pairs, where the first part is the name of a city and the second part is a list of cities which gives the path from the initial city to the current city. When the target city is found, we return the path generated.

However, this function is named `bad_bfs` because it is terrible. [Andrew breaking the fourth wall here - this was my actual first attempt, and it took a lot of work to turn it into something that was actually decent]

There are lots of problems, but first we will attack a bug causing serious issues. Run the code and see that the path being generated is orders of magnitude larger than it should be.

The problem is the line

```
new_path = current_path
```

and it heavily depends on what Python is doing "under the hood." Create a toy example similar to what I am doing with that segment of code. What happens? Given the setup of the search, what should be the behavior? What should that line of code be replaced with?

Make sure to test your new version.

4 Improving the running time

Even after fixing the bug, there are significant improvements that are possible.

1. The improvements I have in mind require a different queue data structure. Look up `collections.deque` at

   `https://docs.python.org/2/library/collections.html#collections.deque`

   and create a new bfs implementation that uses `collections.deque` instead of `queue.Queue`.

2. It is a bad idea to check for the solution when a node is pulled off the frontier - much better to check when the node is expanded and neighbor cities are being added. Make this change, and document how many steps it saves.

3. We should only add a neighbor city to the frontier data structure if it is not already in the frontier, and not already explored. Create a list of explored cities, update it appropriately, and check it. To check whether a city is in the frontier, use a list comprehension to pull out the cities from the list of pairs. Source that may help:

   `https://stackoverflow.com/questions/2917372/`
   `how-to-search-a-list-of-tuples-in-python`

After these improvements, no city should be pulled off of the queue more than once. Check that this is the case with your implementation.