CSPB 3202: Introduction to Artificial Intelligence
Homework 1A

Learning objective: Interpret deterministic and stochastic decision-making problems and implement algorithms for their solution.

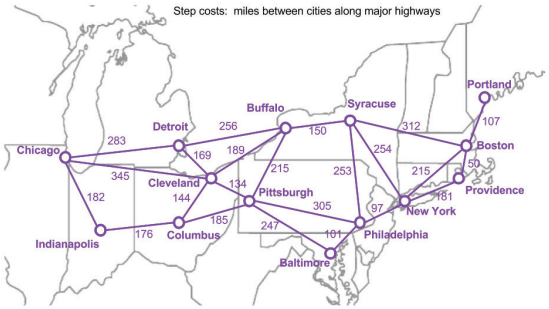
1. Neal Page's Cross-Country Drive

Neal Page (played by Steve Martin) is traveling from a business meeting in New York City to his home in the suburbs of Chicago during Thanksgiving. Based on his experience, he decided to forego the flight this time and to drive it—hoping he won't run into Del Griffith (John Candy) who would seriously derail his plans.

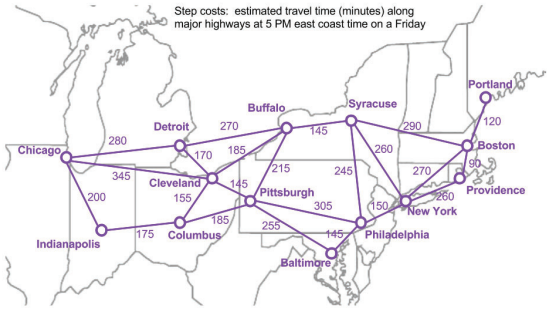
Below are crude 1980s-era graphs representing the northeastern United States. The graph on the left, **map_distances**, represents the step costs between two states on the graph (cities) using the distance between the two cities along major highways. On the right, **map_times** represents the step costs using estimated travel time (at 5 PM on a Friday, east coast time). These graphs are defined in the helper routines in the skeleton code.

If you take a look at those graphs, you will notice that for brevity's sake, we will use **lowercase** abbreviations for each city, consisting of the **first 3 letters** of the city's name. So Providence is represented by the state 'pro', for example.

map_distances



map_times



1.1 Breadth-first search

Implement a function **breadth_first(start, goal, state_graph, return_cost)** to search the state space (and step costs) defined by **state_graph** using breadth-first search:

- **start**: initial state (e.g., 'ind')
- **goal**: goal state (e.g., 'bos')
- **state_graph**: the dictionary defining the step costs (e.g., `map_distances`)
- **return_cost**: logical input representing whether or not to return the solution path cost
  - If **True**, then the output should be a tuple where the first value is the list representing the solution path and the second value is the path cost
  - If **False**, then the only output is the solution path list object

Note that in the skeleton code, two useful routines for obtaining your solution path are provided (and can be used for all the search algorithms):

- **path(previous, s)**: returns a list representing a path to state **s**, where **previous** is a dictionary that maps predecessors (values) to successors (keys)
- **pathcost(path, step_costs)**: adds up the step costs defined by the **step_costs** graph (e.g., `map_distances`) along the list of states **path**

Moodle Quiz Problem 1. Pass the BFS unit tests (see skeleton code).

1.2 Depth-first search

Implement a function **depth_first(start, goal, state_graph, return_cost)** to search the state space (and step costs) defined by **state_graph** using depth-first search:

- **start**: initial state (e.g., 'ind')
- **end**: goal state (e.g., 'bos')
- **state_graph**: the dictionary defining the step costs (e.g., `map_distances`)
- **return_cost**: logical input representing whether or not to return the solution path cost
  - If **True**, then the output should be a tuple where the first value is the list representing the solution path and the second value is the path cost
  - If **False**, then the only output is the solution path list object

Moodle Quiz Problem 2. Pass the DFS unit tests (see skeleton code).

1.3 Uniform-cost search

First, let's create our own `Frontier_PQ` class to represent the frontier (priority queue) for uniform-cost search. Note that the `heapq` package is imported in the helpers at the bottom of this assignment; you may find that package useful. You could also use the `Queue` package. Your implementation of the uniform-cost search frontier should adhere to these specifications

- Instantiation arguments of **Frontier_PQ(start, cost)**
  - **start** is the initial state (e.g., **start**='chi')
  - **cost** is the initial path cost (what should it be for the initial state?)
- Instantiation attributes/methods:
  - **states**: maintains a dictionary of states on the frontier, along with the *minimum* path cost to arrive at them
  - **q**: a list of (cost, state) tuples, representing the elements on the frontier; should be treated as a priority queue (in contrast to the **states** dictionary, which is meant to keep track of the lowest-cost to each state)
  - appropriately initialize the starting state and cost
- Methods to implement:
  - **add(state, cost)**: add the (cost, state) tuple to the frontier
  - **pop()**: return the lowest-cost (cost, state) tuple, and pop it off the frontier
  - **replace(state, cost)**: if you find a lower-cost path to a state that's already on the frontier, it should be replaced using this method.

Note that there is some redundancy between the information stored in **states** and **q**. I only suggest to code it in this way because I think it's the most straightforward way to get something working. You could reduce the storage requirements by eliminating the redundancy, but it increases the time complexity because of the function calls needed to manipulate your priority queue to check for states (since that isn't how the frontier queue is ordered).

Now, actually implement a function to search using `uniform_cost` search, called as **uniform_cost(start, goal, state_graph, return_cost)**:

- **start**: initial state
- **goal**: goal state
- **state_graph**: graph representing the connectivity and step costs of the state space (e.g., **map_distances** or **map_times** below)
- **return_cost**: logical input representing whether or not to return the solution path cost
  - If **True**, then the output should be a tuple where the first value is the list representing the solution path and the second value is the path cost
  - If **False**, then the only output is the solution path list object

Moodle Quiz Problem 3. Pass the UCS unit tests (see skeleton code).

Moodle Quiz Problem 4. Use each of your search functions to find routes for Neal to travel from New York to Chicago, with path costs defined by the distance between cities.

Moodle Quiz Problem 5. Which algorithm yields the shortest path?

Any Chance of Making it for Turkey Time?

Neal did a great job of leaving enough time for his drive until a final meeting was scheduled for the end of the day. Now he's waiting on a very laconic client to sign off on his work. The final meeting finally finishes, leaving him about 15 hours (in fact, 940 minutes) to make it to Chicago.

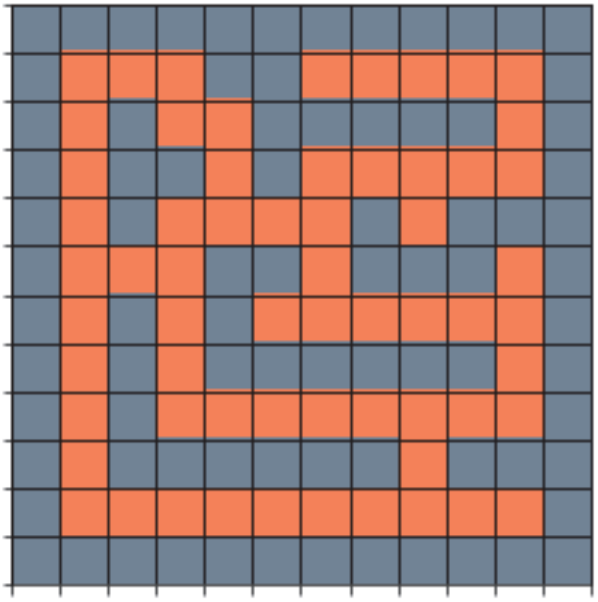
Moodle Quiz Problem 6. Use your choice of search function to show the list of cities that Neal will traverse to get to Chicago on time, should such a path exist.

1.4 Optimization

Since time is a factor (he may not make it in time for the relish plate!), Neal ought to optimize his route from New York to Chicago to minimize the total time required. Because Neal is a wiley dude, he knows some pretty slick search algorithms. But because he's just an ad dude, he has no idea how to code.

Moodle Quiz Problem 7. Use your choice of search function to show the list of cities that Neal would traverse to get to Chicago as quickly as possible.

2. Maze World



Consider this maze, where gray tiles represent walls and orange tiles represent open space where you can walk. We can represent this maze using a binary `numpy` array as follows, where 1s represent walls and 0s represent open space; this is provided in the skeleton code.

Very importantly, note that the *first* row of the **maze** array corresponds to the *bottom* row of tiles in the figure. This is a choice made carefully to reflect the fact that we are going to search for a solution path through this maze in *physical* space, so it is useful for our coordinate system to match Cartesian coordinates. This is in contrast to using the first row of the **maze** array to represent the top of the maze, which looks intuitive.

2.1 Maze-to-Graph Function

Write a function **maze_to_graph(maze)** that:

- takes as input a binary maze **maze**, stored as a `numpy` array, where 0 represents an open space and 1 represents a wall
- returns a graph dictionary in a similar style to **map_distances** and **map_times** (from the problem **1.**)

- the keys are tuples giving the states (coordinate pairs) within the maze (e.g., (1,1) represents the lower-left open space, (2,1) represents the space **to the right** of (1,1), and (0,0) represents the lower-left corner, a wall location); thus, the coordinates within the maze are like Cartesian coordinates, and the x- and y-axes are the bottom and left walls of the maze, respectively
- the values are themselves dictionaries, where the keys are other states within the maze and the values are the actions taken to move to that state
- the actions are moves from the list `{`N',`S',`E',`W'}`.

Moodle Quiz Problem 8. Pass the maze-to-graph unit test.

2.2 Conducting Search with Maze-to-Graph

- The initial state is (1,1)
- The goal state is (10,10)

Moodle Quiz Problem 9. Use your depth-first search function to solve the maze and provide the solution path.

Moodle Quiz Problem 10. Use your breadth-first search function to solve the maze and provide the solution path and its length.

If your codes are sufficiently general, the output from **maze_to_graph** should be suitable to be fed straight into your search routines.