CIS 210, Fall 2016
Introduction to Computer Science

Main Menu

- Class home page
- Piazza
- How to Succeed
- Schedule
- Assignments
- References
- Exams

Routing

This assignment is due at 5pm on Friday, November 4. Use Canvas to turn in routes.py

Purpose

Like “set membership and string reversal,”, this project involves recursion; you mainly have to carefully design and implement just one recursive function.

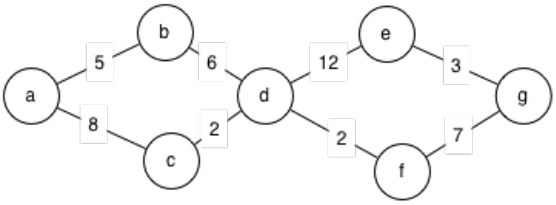
Pair Assignment

You are encouraged to use *pair programming* to complete this assignment. You can share code with one other classmate, who should be listed as an author in your program docstring. You may discuss general approach and design with other students, without sharing code; cite them in your docstring clearly distinguishing design discussion from code authorship.

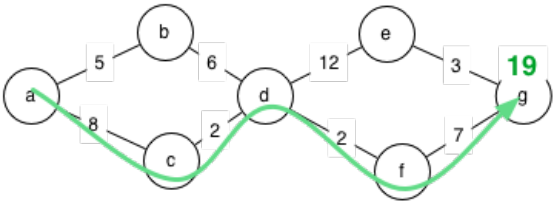
The amount of actual programming required for this project is small, but be sure you understand how the main program logic works.

How far to Barad Dur?

Suppose you are given a map of a particularly boring country in which cities are named a,b,c,d,e,f, and g, with road distances in furlongs as follows:



Your program must determine the shortest possible distance between two cities. For example, if it is asked the difference between a and g on the given map, it should determine that the shortest route is 19 furlongs:



Requirements

I have provided some code to get started. You don't have to use it, but if you do, the FIXME comments indicate places where you will need to add or modify code.

Road map descriptions look like this:

```
#
# This sample map corresponds to the diagram above
#
a,b,5
a,c,8
b,d,6
c,d,2
d,e,12
d,f,2
e,g,3
f,g,7
```

The input and output in the command or terminal window will look like this:

```
> python3 routes.py a g sample-map.txt
Distance from a to g is 19.0
```

In case of a misspelled city, your program will report that it does not appear on the map:

```
> python3 routes.py a x sample-map.txt
Destination  x  is not on the map
```

It can also cope with cities that are present on the map, but with no route between:

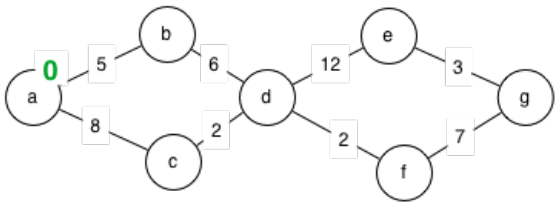
```
> python3 routes.py a x sample-map-noreach.txt
You can't get from a to x
```

How to solve it

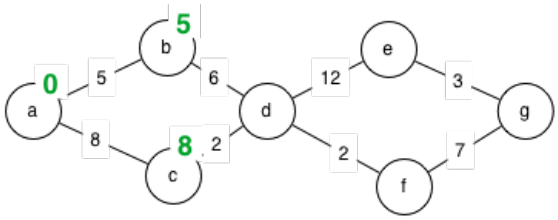
A strategy

There are several possible algorithms, but the one I suggest is a simple depth-first search using a “single source shortest path” approach. This approach finds the shortest distance from a single source (the starting city) to *all* other reachable cities using depth-first search. Here is a step-by-step example for finding the distance from city a to city g:

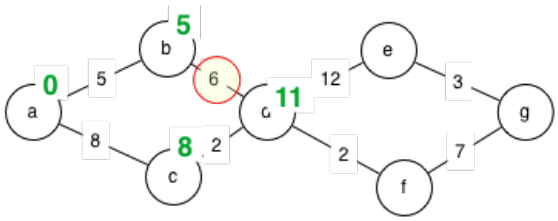
We begin by noting that the distance from city a to city a is zero furlongs.



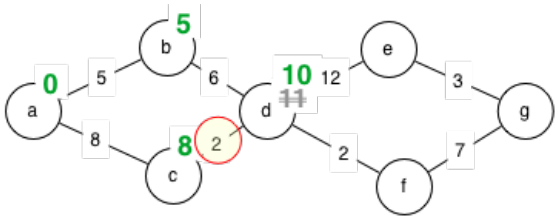
Then we calculate the distances of the neighbors of a.



From each city, we make recursive calls to calculate the distance of its neighboring cities. Suppose the recursive call on city b occurs next, so we calculate a distance of 5+6 = 11 for city d.

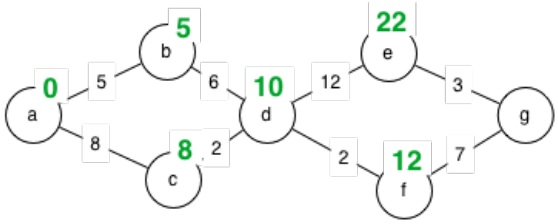


Later, a recursive call from city c finds a shorter distance to d:

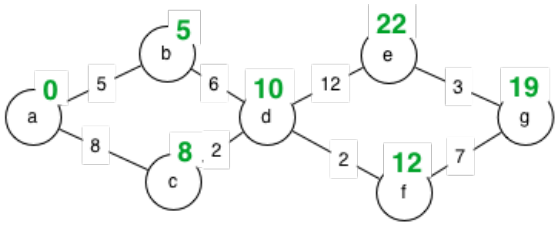


When we find a shorter route to a city that has already been visited, the new shorter distance becomes the new estimate. Since this could improve estimates of distance for other cities reachable from the current city (e.g., any estimates of distancs to e, f, and g that we have made using the estimate of 11 furlongs to city d), we must again make the recursive calls from the city with the improved estimate.

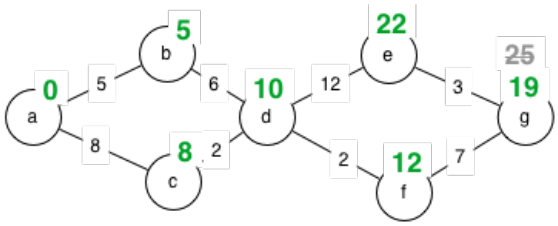
Using the improved estimate of 10 furlongs from a to d, we obtain distances to e and f:



Again we have two ways to reach city g. Suppose this time, just by luck, we first explore the path going through f, and obtain an estimate of 19 for g.



Later we explore the road from e to g. However, it produces a worse estimate, 25 furlongs, so we keep the old estimate of 19 and do not repeat exploration from city g.



This algorithm is not guaranteed to be fast (in the worst case we could make four different estimates of the distance to city g, and with a more complicated road map we might explore every city many times). Better algorithms, which you will learn later if you continue to take CIS courses, propagate distance estimates in a similar way but may visit cities in an order that differs from depth-first search.

From strategy to code

Expressing the above strategy in Python code depends a lot on how the road map and the current estimates of distance are represented. The starter code represents the road map in a variable roads as a dictionary. The sample map above would be represented this way:

```
{
  'a': [('b', 5.0), ('c', 8.0)],
  'c': [('a', 8.0), ('d', 2.0)],
  'b': [('a', 5.0), ('d', 6.0)],
  'e': [('d', 12.0), ('g', 3.0)],
  'd':  [('b', 6.0), ('c', 2.0), ('e', 12.0), ('f', 2.0)],
  'g': [('e', 3.0), ('f', 7.0)],
  'f': [('d', 2.0), ('g', 7.0)]
}
```

Consider for example if we inquire about roads from city a, that is roads["a"]. We would obtain a list of connections and distances in this form:

```
[('b', 5.0), ('c', 8.0)]
```

The first element of this list is a tuple ('b', 5.0). A tuple can be “deconstructed” (that is, separated into its components) with a Python assignment statement to multiple variables. For example, if we write
to_city, dist = ('b', 5.0)
the value 'b' will be stored in to_city and the value 5.0 will be stored in dist. Your textbook also explains the Python tuple type.

Note that each road is represented twice. For example, the road between a and b is represented as a road from a to b and as a road from b to a, both with the same distance 5.0.

In addition to the road map, your program will need to keep its current estimates of distance to each city. This can easily be done using a Python dictionary structure, with city names as keys and estimated distances as values. In the starter code, I have used the variable distances to store these estimates. Note that if a city has not yet been visited, it will not appear in the distances dict. If it is still not in the distances dict after exploring all cities reachable from the start city, there must not be any path from the start city to the destination.

I've provided quite a bit of starter code with this project, so you can concentrate on function dfs.

In addition to the starter code, I've provided a few road maps. ME-distances.txt is taken from a set of distance estimates found online. Distances in ME-distances.txt are in hours on horseback, rather than furlongs, but that doesn't matter. sample-map.txt is the map used as an example above; sample-map-noreach.txt is a variation that adds a pair of cities x and y that are not reachable from anywhere else. skip-map.txt is a map in which the shortest paths are not the most direct paths. This may be useful in testing your program to make sure it is finding the shortest distance rather than finding the route with the smallest number of steps.

Efficiency

In computer science and software development, we usually characterize the efficiency of an algorithm or program as a function of problem size, reasoning about the *worst possible* case. If you consider the depth first search in this regard, it is easy to construct a worst case that, for *n* cities and *2n* roads, explores 2^n paths before it finally finds the shortest path. (Convince yourself by constructing a map with 5 cities that could take up to 32 steps.) We say it the algorithm is *exponential* in the number of cities. That's bad.

There are much, much better algorithms, but they are a little more complicated. If you have finished implementing the simple depth-first search, and would like to try implementing one of the more efficient algorithms, drop a note on Piazza and I'll provide some guidance on implementing a better search algorithm.

Aside: Furlongs are an ancient measure of distance, used originally by the Roman Empire and based on how long a person could walk in an hour. In most of the world they have long ago been replaced by SI measures. In

the U.S. they are still a legal measure, although the length of a furlong varies slightly from state to state. This is a crazy way to measure anything, but not much crazier than feet, inches, ounces, and miles.

Grading rubric

Functional correctness					**40**
	Exactly meets input/output spec	10	5 = minor discrepancy, 0 = ignored spec		
	Correct calculations	30	30 = works for all cases, 20 = works for almost all cases (including all of the provided examples), 10 = works for most cases (e.g., all but one of the example maps), 5 = works for some cases		
Other requirements					35
	Header docstring	5	5 = as specified, 3 = minor issue, e.g., as #comment, 0 = missing or incomplete		
	Function header docstrings	8	8 = good docstrings in all functions, 6 = minor problems, 4 = incorrect or multiple missing, 0 = docstrings not provided		
	Recursion	12	12 = Excellent use of recursion; recursive function is simple and clean; 8 = Correct use of recursion, but function is unnecessarily complex (e.g., because a test is repeated before each recursive call, when it could be done once in the called function); 4 = Serious problems in organization of the recursive function, or how it is called.		
	Program style and readability	10	10 Good variable names, indentation, etc --- very readable code, 8 = minor issues, such as inconsistent indentation, 5 = major issues that interfere with readability of code, 0 = unreadable mess		
Total					75

I can't anticipate all issues that may be encountered in grading, so points may be deducted for other issues not listed in the rubric. A program that does not compile and run (e.g., because of a syntax error) starts with 0 points for functional correctness, but the grader at his or her discretion may award some partial credit.