L06: Problem Solving with Search

Faculty of Engineering, Khon Kaen University

Submission: https://autolab.en.kku.ac.th

==========

* Submit an answer to a question with a file with `txt` extension. E.g., an answer for Q1 should be submitted in a text file "`Q1.txt`"

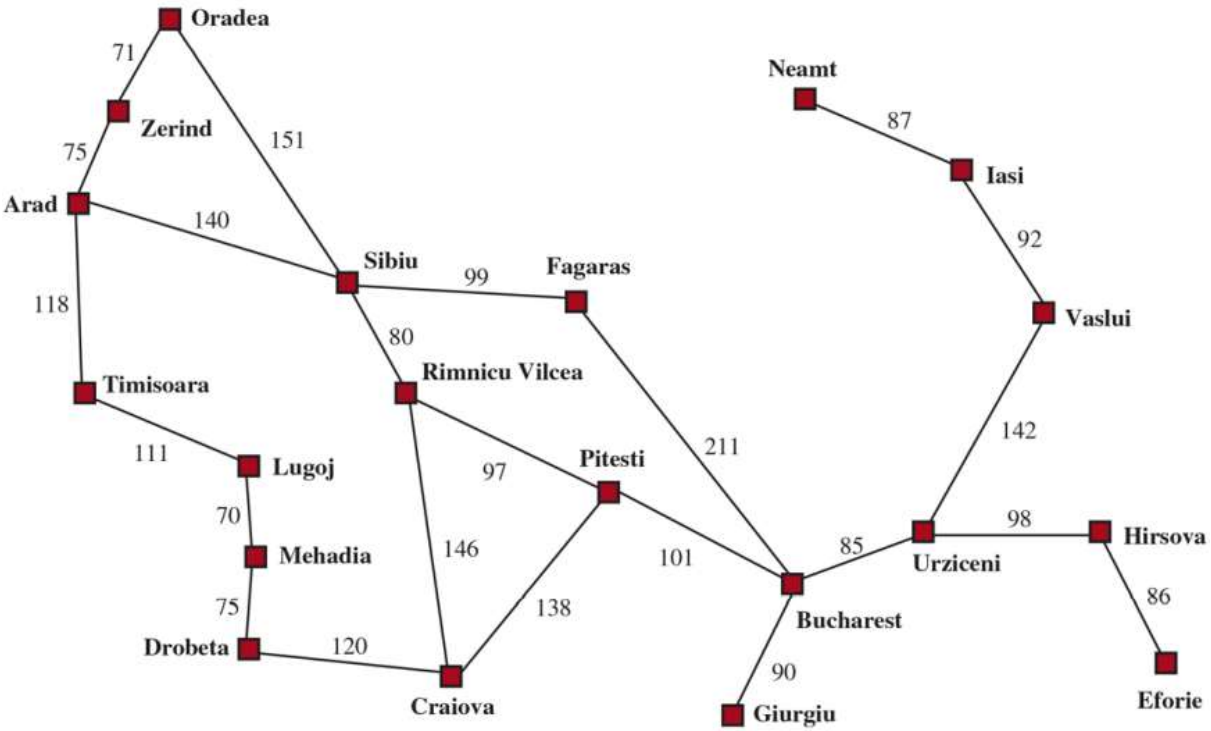
* Submit a program to a (programming) problem with a file with a proper extension. E.g., a python program for P2 should be named "`P2.py`"

* Each question or problem is worth 840 points.

* Autograder is run on Python 3.9 without any non-standard library. For example, `math` is a standard library and can be imported, while `numpy` is not and any attempt to import will cause an error.

==========

P1. Roaming Romania. Given a driving map of Romania, a current town and a destination to go, find a route to get to the destination.



Write a function `P1_route(start, goal)` to take `start` for a current town and `goal` for a destination, then find a route from the starting town to the destination. Note: any route is fine. No preference on shortest path or passing fewest towns or anything. Geting there is good enough.

Hint

(1) Breadth-first search is complete and also among the easier to implement. It may not be very efficient, but for this small problem, it is ok.

(2) Take it easy: do it step by step.

(3) Test every step to make sure it really work.

Output example, when supplying `Arad` and `Bucharest` for the arguments, the function returns a string `Arad - Sibiu - Fagaras - Bucharest` as shown below:

Run:

Test P1

```
from P1 import P1_route

if __name__ == '__main__':

    print(P1_route("Arad", "Bucharest"))
```

Screen:

```
Arad - Sibiu - Fagaras – Bucharest
```

Note that your program might have found different routes from what shown in the examples. As long as they are proper routes to the destinations, they are fine.

The accompanied template provides the map data.

P2. Traveling route. Given a traveling map, a current town and a place to go, find a route passing the fewest towns to get there.

Write a function `P2_route(map_fname, start, goal)` to take `map_fname` for a filename whose content is map information (as shown below), `start` for a current town and `goal` for a destination, then find a route from the starting town to the destination. To avoid the traffic, it is preferable to have **the route visiting the fewest towns** as possible. (Hint: breadth-first search gives you the fewest steps.)

Output example, when supplying `P2romania.txt`, `Arad` and `Bucharest` for the arguments, the function returns a string `Arad - Sibiu - Fagaras - Bucharest` as shown below:

Run:

Test P2

```
from P2 import P2_route

if __name__ == '__main__':
    r = P2_route('P2romania.txt', 'Arad', 'Bucharest')
    print(r)
```

Screen:

```
Arad - Sibiu - Fagaras - Bucharest
```

for `P2romania.txt`, whose content looks like

```
P2romania.txt
Arad : 91, 492; Timisoara : 118, Zerind : 75, Sibiu : 140
Oradea : 131, 571; Zerind : 71, Sibiu : 151
Hirsova : 534, 350; Eforie : 86, Urziceni : 98
 :              :              :             :
```

Notice: the format goes *<town name>* : *<x location>*, *<y location>*; *<town>* : *<distance>*, ...

P3. Shortest path. Given a traveling map, a current town and a place to go, find the shortest route to get there.

Write a function `P3_route(map_fname, start, goal)` to take `map_fname` for a filename whose content is map information (as shown below), `start` for a current town and `goal` for a destination, then find **the shortest route** from the starting town to the destination. (Hint: uniform-cost search with distance as path cost gives you the lowest cost.)

Output example, when supplying `P2romania.txt`, `Arad` and `Bucharest` for the arguments, the function returns a string `Arad - Sibiu - Fagaras - Bucharest` as shown below:

Run:

```
                                    Test P3
from P3 import P3_route

if __name__ == '__main__':
    r = P3_route('P2romania.txt', 'Arad', 'Bucharest')
    print(r)
```

Screen:

```
Arad - Sibiu - Rimnicu Vilcea - Pitesti - Bucharest
```

Confer this result to P2 example.

for `P2romania.txt`, whose content looks like

```
P2romania.txt
```

```
Arad : 91, 492; Timisoara : 118, Zerind : 75, Sibiu : 140
Oradea : 131, 571; Zerind : 71, Sibiu : 151
Hirsova : 534, 350; Eforie : 86, Urziceni : 98
 :              :             :            :
```

Notice: the format goes <*town name*> : <*x location*>, <*y location*>; <*town*> : <*distance*>, ...

P4. Solving the three-missionary puzzle. The situation goes as follows. Three missionaries and three cannibals are on one side of a river, along with a boat that can hold one or two people. Find a way to get everyone to the other side without ever leaving a group of missionaries in one place outnumbered by the cannibals in that place.

Write a function `P4_crossriver(initial)` to take `initial` as a tuple of integers representing numbers of missionaries and cannibals on the waiting bank, numbers of missionaries and cannibals on the crossed side, and whether a boat is on the waiting bank (1: waiting bank, 0: crossed side). E.g., a classic problem starts with `(3, 3, 0, 0, 1)` for 3 missionaries and 3 cannibals on the waiting bank, none are on the crossed side, and a boat is on the waiting bank. The goal state will be `(0, 0, 3, 3, 0)` for none are left on the waiting bank, 3 missionaries and 3 cannibals are on the crossed side, and boat is on the crossed side. (Boat is not important, but it will be there since no one will row it back.)

Then find a way to bring everyone across. The classical three-missionary puzzle will be equivalent to `num_m = 3`. The function gives an output as a list of tuples each specifying integers representing numbers of missionaries and cannibals on a boat. A boat can hold one or two people. I.e., an action can be `(1, 0)`, `(0, 1)`, `(1, 1)`, `(2, 0)`, or `(0, 2)` for one missionary takes the boat, one cannibal takes the boat, a pair of missionary and cannibal takes the boat, two missionaries takes the boat, and two cannibals takes the boat.

Do not leave missionaries outnumbered by cannibals on either bank.

Hint

(1) To standardize the function, the question has given away the idea of how to define state and action.

(2) Define (the model of) the problem, i.e., specifying transition and goal state.

**** Model is enough to represent the problem, but leave out the irrelevant details to allow efficient handling. E.g., action is enough represented by just two numbers without stating which way it is going, since it will always go to the other side back and forth. ****

(3) Solving a problem is just searching through problem state space for a solution (path or sequence of actions leading to the goal state).

Output example, when supplying 2 for the argument, the function returns a list of tuples as shown below:

Run:

Test P4

```
from P4 import P4_crossriver

if __name__ == '__main__':
    print(P4_crossriver((2,2,1,1,1)))
```

Screen:

```
[(2, 0), (0, 1), (0, 2), (1, 0), (1, 1)]
```

Note: If you are wonder, here's how it goes.

(2, 2, 1, 1, 1) is the root.

step 1 : (2, 2, 1, 1, 1) -> (2, 0) -> (0, 2, 3, 1, False)

step 2 : (0, 2, 3, 1, False) -> (0, 1) -> (0, 3, 3, 0, True)

step 3 : (0, 3, 3, 0, True) -> (0, 2) -> (0, 1, 3, 2, False)

step 4 : (0, 1, 3, 2, False) -> (1, 0) -> (1, 1, 2, 2, True)

step 5 : (1, 1, 2, 2, True) -> (1, 1) -> (0, 0, 3, 3, False)

P5. Solving word-error-rate problem. Word-error rate is a widely-used performance metric for a speech recognition or machine translation system. Word error rate can be computed from

$$wer = (S + D + I)/N$$

where *S*, *D*, and *I* are numbers of substitutions, deletions, and insertions and *N* is a number of tokens in the reference.

For example, given a reference word "grit" and a test word "greet",

	g	r	i		t
g	0				
r		0			

e			S		
e				D	
t					0

Therefore, *wer(grit, greet) = 2/4 = 0.5*.

Notice that *S* + *D* + *I* represents the minimum number of operations required.

Another example, reference is “tenacity” and a test word “density”,

	t	e	n	a	c	i	t	y
d	S							
e		0						
n			0					
				I				
s					S			
i						0		
t							0	
y								0

Therefore, *wer(tenacity, density) = 3/8 = 0.375*.

Write a function `P5_wer(ref, test)` to take `ref` as a string representing a reference word and `test` as a string representing a test word. Then find a minimal set of operations required. The function gives an output as a tuple of wer (as float) and the minimum number of operations required (as integer).

Hint

(1) What are the states and actions?

(2) Define (the model of) the problem, i.e., specifying transition and goal state.

(3) Solving a problem is just searching through problem state space for a solution, but in this case we want the minimum number of operations.

Output example, when supplying 2 for the argument, the function returns a list of tuples as shown below:

Run:

Test P5

```
from P5 import P5_wer

if __name__ == '__main__':
    wer, n = P5_wer("grit", "greet")
    print("wer = {}, n = {}".format(wer, n))
```

Screen:

```
wer = 0.5, n = 2
```

Autograder is run at 0.001 tolerance.