FIT2004 S1/2020: Assignment 4 - Graphs

DEADLINE: Friday 12th June 2020 23:55:00 AEST

LATE SUBMISSION PENALTY: 10% penalty per day. Submissions more than 7 days late are generally not accepted. The number of days late is rounded up, e.g. 5 hours late means 1 day late, 27 hours late is 2 days late. For special consideration, please complete the online special consideration form.

PROGRAMMING CRITERIA: It is required that you implement this exercise strictly using **Python programming language** (version should not be earlier than 3.5). This practical work will be marked on the time complexity, space complexity and functionality of your program.
Your program will be tested using automated test scripts. It is therefore critically important that you name your files and functions as specified in this document. If you do not, it will make your submission difficult to mark, and you will be penalised.

SUBMISSION REQUIREMENT: You will submit a single python file, `assignment4.py`.

PLAGIARISM: The assignments will be checked for plagiarism using an advanced plagiarism detector. Last year, many students were detected by the plagiarism detector and almost all got zero mark for the assignment and, as a result, many failed the unit. "Helping" others is NOT ACCEPTED. Please do not share your solutions partially or/and completely to others. If someone asks you for help, ask them to visit us during consultation hours for help.

Learning Outcomes

This assignment achieves the Learning Outcomes of:

- 1) Analyse general problem solving strategies and algorithmic paradigms, and apply them to solving new problems;
- 2) Prove correctness of programs, analyse their space and time complexities;
- 4) Develop and implement algorithms to solve computational problems.

In addition, you will develop the following employability skills:

- Text comprehension
- Designing test cases
- Ability to follow specifications precisely

Assignment timeline

In order to be successful in this assessment, the following steps are provided as a **suggestion**. This is an approach which will be useful to you both in future units, and in industry.

Planning

1. Read the assignment specification as soon as possible and write out a list of questions you have about it.
2. Clarify these questions. You can go to a consultation, talk to your tutor, discuss the tasks with friends or ask in the forums.
3. As soon as possible, start thinking about the problems in the assignment.
    - It is strongly recommended that you **do not** write code until you have a solid feeling for how the problem works and how you will solve it.
4. Writing down small examples and solving them by hand is an excellent tool for coming to a better understanding of the problem.
    - As you are doing this, you may see patterns which allow you to deduce the correct algorithm to solve the problem.
    - You will also get a feel for the kinds of edge cases you will need to handle.
5. Write down a high level description of the algorithm you will use.
6. Determine the complexity of your algorithm idea, ensuring it meets the requirements.

Implementing

1. Think of test cases that you can use to check if your algorithm works.
    - Use the edge cases you found during the previous phase to inspire your test cases.
    - It is also a good idea to generate large random test cases.
    - Sharing test cases **is** allowed, as it is not helping solve the assignment. Check the forums for test cases!
2. Code up your algorithm, (remember decomposition and comments) and test it on the tests you have thought of.
3. Try to break your code. Think of what kinds of inputs you could be presented with which your code might not be able to handle.
    - Large inputs
    - Small inputs
    - Inputs with strange properties
    - What if everything is the same?
    - What if everything is different?
    - etc...

Before submission

- Make sure that the input/output format of your code matches the specification.
- Make sure your filenames match the specification.
- Make sure your functions are named correctly and take the correct inputs.

Documentation (3 marks)

For this assignment (and all assignments in this unit) you are required to document and comment your code appropriately. This documentation/commenting must consist of (but is not limited to)

- For each function, high level description of that function. This should be a one or two sentence explanation of what this function does. One good way of presenting this information is by specifying what the input to the function is, and what output the function produces (if appropriate)
- For each function, the Big-O complexity of that function, in terms of the input. Make sure you specify what the variables involved in your complexity refer to. Remember that the complexity of a function includes the complexity of any function calls it makes.
- Within functions, comments where appropriate. Generally speaking, you would comment complicated lines of code (which you should try to minimise) or a large block of code which performs a clear and distinct task (often blocks like this are good candidates to be their own functions!).

Example of usage

In this assignment you will be creating a Graph class, and implementing methods for that class. This example is to clarify how your class will be called in the testing code. In other words, you should be able to call the methods of your class in the way shown below.

```
#In your file...
class Graph:
    #contents of your class

#In the marker's tester code...
#assignment4 is your .py file, Graph is the name of the class you wrote
from assignment4 import Graph

#this creates a graph from the specified file name
g = Graph("example_graph_file")

root, depth = g.shallowest_spanning_tree() #this runs task 2

#run task 3, with 3 as the start, 7 as the desination,
#2, 5 & 11 as ice locations, and 4 & 1 as drink locations
length, path = g.shortest_errand(3,7,[2,5,11],[4,1])
```

1 Setup (5 marks)

In this assignment you will consider some optimisation problems on a graph. To solve these problem, you first need to read the data representing the graph into an appropriate data structure. You need to create a class, `Graph`, and write the `__init__(self, gfile)` function, which generates a `Graph` object from a file, as defined below.

1.1 Input

`gfile` is a string, which is the name of a file containing information about the graph. The first line contains a single intenger n, which is the number of vertices in the graph. Each following line of the file consists of three integers separated by spaces. The first two integers are vertex IDs (numbers in the range $[0, n-1]$), and the third (which is non-negative) is the weight of that edge. Every edge of the graph will be represented by exactly one line of `gfile`.

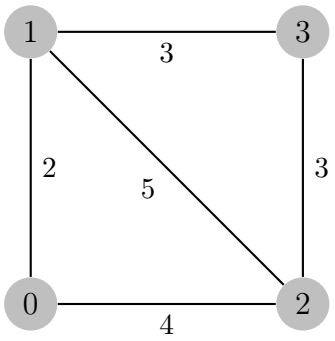
The graph represented by this file will be **connected**, **undirected** and **simple**.

1.2 Output

For this task, no output is required. Note that the other two tasks in this assignment involve implementing methods of the `Graph` class, so be sure to read those tasks carefully, and think about what your `__init__` function will need to do in order to allow you to solve the later tasks in the appropriate time complexities.

Example:
If gfile contained the following:,

```
4
0 1 2
0 2 4
1 2 5
1 3 3
2 3 3
```



1.3 Complexity

Your `__init__()` function must run in $O(V^2)$ time. It must use no more than $O(V^2)$ auxiliary space.

- V is the number of vertices in the graph

Note that big O notation is an upper bound.

2 Shallowest spanning tree (10 marks)

You want to find the most convenient suburb to live. You have constructed a graph which represents all the suburbs you might choose. Each edge of the graph represent a public transit link between two suburbs.

Each suburb in the graph also contains places that you will need to visit regularly. You want to choose a suburb to live such that the farthest suburb from your new house is as close as possible, by public transport. Since you enjoy reading, you don't care how long each leg of the journey is, only how many times you need to change (i.e. how many edges you traverse).

In this task, you will find the shallowest spanning tree of the graph stored in your `Graph` class. In other words, find a spanning tree which minimises the number of edges from the root of the spanning tree (where you will choose to live) to the deepest leaf (the farthest suburb). For this task, you will **ignore the edge weights**. To do this you will write a method of your graph class, `shallowest_spanning_tree(self)`.

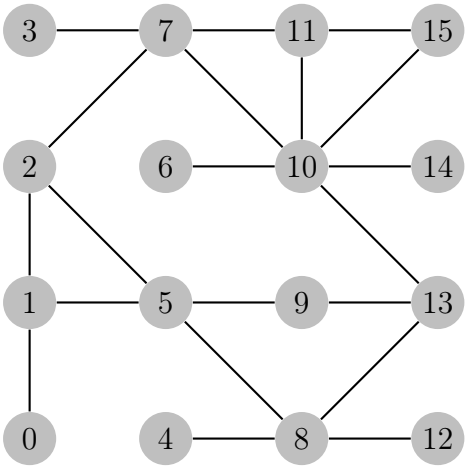
2.1 Input

No inputs required.

2.2 Output

`shallowest_spanning_tree(self)` returns a tuple containing two items. The first item is the vertex ID of the root which gives the shallowest spanning tree. The second item is an integer, representing the height of the shallowest spanning tree. Recall that the height of a tree is the length of the longest path from root to a leaf (where length of a path is the number of edges on that path).

Example:
If the `Graph` class contained the following graph (edge weights not shown, since they are not relevant to this task):



The value returned by `shallowest_spanning_tree` would be `(2, 3)`, meaning there exists a spanning tree rooted at vertex `2`, with depth `3`, and no shallower spanning tree exists.

Note that is it possible for there to be multiple shallowest spanning trees rooted at different nodes. If that is the case, all of them are correct answers.

2.3 Complexity

`shallowest_spanning_tree(self)` must run in $O(V^3)$ where V is the number of vertices in the graph.

3 Shortest errands (12 marks)

You have borrowed a car, and you want to visit a friend. Before you get to your friend's, you want to pick up some ice, and then some ice cream, which needs to be kept cold.

There are many possible places to pick up both of these items, but you must pick up ice before you pick up the ice cream, and you want to drive the shortest distance. To solve this problem, you will write a function `shortest_errand(self, home, destination, ice_locs, ice_cream_locs)`. This function will find the shortest walk which solves this problem.

3.1 Input

`home` and `destination` are each a vertex ID. `ice_locs` and `ice_cream_locs` are lists, each containing at least one vertex ID. It is possible for locations to be any combination of the four location types.

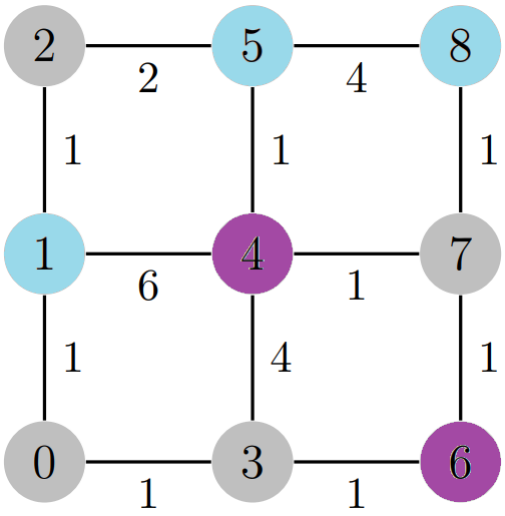
3.2 Output

`shortest_errand` must return a tuple, where the first item is the length of the shortest walk you have found. The second item should be a list of vertices with the following properties:

- The list of vertices represents a walk from `home` to `destination` (So the first vertex should be `home`, the last should be `destination`, and other elements should be the vertices visited on the walk, in order)
- There is at least one vertex from the set `ice_cream_locs` which appears **after** a vertex from the set `ice_locs`

In other words, you can visit as many vertices as you like from each list, but you must visit a location from `ice_cream_locs` after you have had a chance to buy ice at one of the locations from `ice_locs`.
Note that the shortest walk may reuse edges and vertices, i.e. it may be optimal to go somewhere and then retrace your steps.

Example:
If the `Graph` class contained the following graph:



The result of calling `shortest_errand(0, 8, [1,5,8], [4,6])` would be `(6, [0,1,0,3,6,7,8])`. Ice locations are shown in blue, and ice cream locations are shown in purple.

3.3 Complexity

`shortest_errand` must run in $O(Elog(V))$, where E is the number of edges in the graph and V is the number of vertices in the graph.

Warning

For all assignments in this unit, you may **not** use python **dictionaries** or **sets**. This is because the complexity requirements for the assignment are all deterministic worst case requirements, and dictionaries/sets are based on hash tables, for which it is difficult to determine the deterministic worst case behaviour.

Please ensure that you carefully check the complexity of each inbuilt python function and data structure that you use, as many of them make the complexities of your algorithms worse. Common examples which cause students to lose marks are **list slicing**, inserting or deleting elements **in the middle or front of a list** (linear time), using the `in` keyword to **check for membership** of an iterable (linear time), or building a string using **repeated concatenation** of characters. Note that use of these functions/techniques is **not forbidden**, however you should exercise care when using them.

These are just a few examples, so be careful. Remember, you are responsible for the complexity of every line of code you write!