FIT2004 S1/2021: Assignment 4 - Graph Algorithms

DEADLINE: Friday 28^{th} May 2021 23:55:00 AEST

LATE SUBMISSION PENALTY: 10% penalty per day. Submissions more than 7 calendar days late will receive 0. 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 visit this page:
`https://www.monash.edu/connect/forms/modules/course/special-consideration`
and fill out the appropriate form. **Do not** contact the unit directly, as we cannot grant special consideration unless you have used the online form.

PROGRAMMING CRITERIA: It is required that you implement this exercise strictly using the **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, and your documentation.
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. In previous semesters, 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 completely to others. If someone asks you for help, ask them to visit a consultation 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 will also get a feel for the kinds of edge cases your code will have to deal with.
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.
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.
- Make sure you zip your files correctly (if required)

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!).

Graph class (0 Marks)

This assignment is about graphs and graph algorithms. As such, both problems relate to graphs in some way. It is strongly recommended that you construct graphs to solve these tasks, and given that you are constructing graphs, you will probably find it easiest if you implement a **graph class** to store your graphs.

A very basic example implementation of an adjacency-list based graph class will be provided on Ed, and you may use this as your starting point for a class.

1 Liquid Trading (10 marks)

In this task, you are a trader who specialises in valuable liquids. You have come to a new town, and your goal is to trade with the local people in order to maximise the value of the liquid you have.

Each person in the town is willing to make various trades at various ratios. For example, one person might trade water for mercury at a rate of 1L water for 0.01L mercury. Since you are from the big city, you know the prices of each of these liquids, and so you can determine which trades are worthwhile and which are not.

You are only in the town for a limited time, and trading takes time, so you have a maximum number of trades you can conduct before you need to move on, though you may choose to trade fewer times than this maximum. You also only have one container (with unlimited capacity!), and you cannot mix liquids or they will become worthless, so you must trade all of your current liquid whenever you make a trade.

To solve this problem you will write a function, `best_trades(prices, starting_liquid, max_trades, townspeople)`

1.1 Input

Liquids each have an ID. There are `n` liquids, and each one has a unique ID from the range `[0..n-1]`.

`prices` is an array of length `n`, where `prices[i]` is the value of 1L of the liquid with ID `i`.

`starting_liquid` is the ID of the liquid you arrive with. You always start with 1L of this liquid.

`townspeople` is a list of lists. Each interior list corresponds to the trades offered by a particular person. The interior lists contain 3 element tuples, `(give, receive, ratio)`. Each tuple indicates that this person is willing to be given liquid with ID `give` in exchange for liquid with ID `receive` at the given ratio. For example, `(2,5,1.5)` indicates that you can give liquid with ID 2, and receive 1.5 times as much liquid with ID 5. Note that people are not necessarily willing to perform these trades in both directions, i.e. this example does **not** mean that you can give liquid 5 and receive liquid 2.

For each liquid, there will be at least one townsperson who is willing to trade for that liquid.

1.2 Output

`best_trades` should return the maximum value that you can obtain after performing at most `max_trades` trades.

1.3 Example

```
prices = [10,5,1,0.1]
starting_liquid = 0
max_trades = 6
townspeople = [[(0,1,4),(2,3,30)],[(1,2,2.5),(2,0,0.2)]]
best_trades(prices, starting_liquid, max_trades, townspeople)
>>>60
max_trades = 2
best_trades(prices, starting_liquid, max_trades, townspeople)
>>>20
```

The solution to the first case is to perform the following trades:

```
starting value: $10
trade 1L of 0 for 4L of 1: value $20
trade 4L of 1 for 10L of 2: value $10
trade 10L of 2 for 2L of 0: value $20
trade 2L of 0 for 8L of 1: value $40
trade 8L of 1 for 20L of 2: value $20
trade 20L of 2 for 600L of 3: value $60
```

The solution to the second case is to perform the following trade:

```
starting value: $10
trade 1L of 0 for 4L of 1: value $20
```

Note that in this example, although we were permitted to make 2 trades, making a second trade was not profitable, so we chose to stop after making 1.

1.4 Complexity

`best_trades` must run in $O(T * M)$ time where

- T is the total number of trades available
- M is `max_trades`

2 Optional Delivery (7 marks)

In this task, we wish to travel from one city to another. Traveling is costly, so we want to get to our destination as cheaply as possible. However, there is a way we can make some money on our way. We can pick up an item from one particular city, and deliver it to another particular city. We want to determine whether it will be cheaper to perform this delivery during our journey, or just go directly to our destination.

To solve this problem, you will write a function `opt_delivery(n, roads, start, end, delivery)`

2.1 Input

`n` is the number of cities. The cities are numbered `[0..n-1]`.

`roads` is a list of tuples. Each tuple is of the form `(u,v,w)`. Each tuple represents an road between cities `u` and `v`. `w` is the cost of traveling along that road, which is always non-negative. Note that roads can be traveled in either direction, and the cost is the same. There is at most 1 road between any pair of cities. `roads` will represent a simple, connected graph

`start` and `end` are each an integer in the range `[0..n-1]`. They represent the city you start (from now on called the "start city") and the city you need to reach (from now on called the "end city"), respectively.

`delivery` is a tuple containing 3 values. The first value is the city where we can pick up the item (from now on called the "pickup city"). The second value is the city where we can deliver the item (from now on called the "delivery city"). The third value is the amount of money we can make **if** we deliver the item from the pickup city to the delivery city.

Note that we can only perform the delivery at most once (i.e. we cannot accrue value by repeatedly delivering the item for a profit).

2.2 Output

`opt_delivery` returns a tuple containing 2 elements. The first element is the cost of travelling from the start city to the end city. This cost includes the profit we make from the delivery, if we choose to make the delivery (so it could be negative, in the event that the delivery is worth more than the total travelling cost).

The second element of the tuple is a list of integers. This list represents the cities we need to travel to in order to achieve the cheapest cost (in order). It should start with the start city, and end with the end city. As seen in the example below, it is possible to need to visit a city twice.

Note that it is possible for a city to be any combination of the start, end, pickup or delivery city. So we might need to start and end in the same place (start = end), or we might need to deliver to the place we are going (end = delivery), or any other combination.

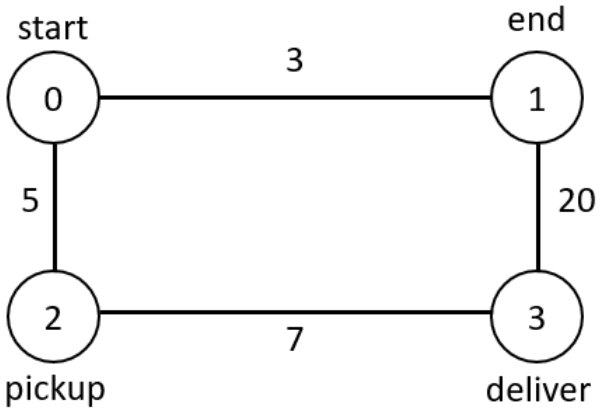
2.3 Example

```
n = 4
roads = [(0,1,3),(0,2,5),(2,3,7),(1,3,20)]
start = 0
end = 1
delivery = (2,3,25)
profit = 25
opt_delivery(n, roads, start, end, delivery)
>>> (2, [0,2,3,2,0,1])

delivery = (2,3,20)
opt_delivery(n, roads, start, end, delivery)
>>> (3, [0,1])

delivery = (2,3,100)
opt_delivery(n, roads, start, end, delivery)
>>>(-73, [0,2,3,2,0,1])
```

2.4 Image of the roads in example



2.5 Complexity

`opt_delivery` must run in $O(Rlog(N))$ where

- R is the total number of roads
- N is the total number of cities

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!