FIT2004 S2/2020: Assignment 4 - Graph Algorithms

DEADLINE: Friday 6^{th} November 2020 23:55:00 AEST

LATE SUBMISSION PENALTY: 10% penalty per day. Submissions more than 5 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 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 named `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 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;
- 3) 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.
4. It is often helpful to solve the relevant tutorial problems, or review them. You can watch the tutorial solution videos for a detailed explanation.
    - 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.
5. 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.
6. Write down a high level description of the algorithm you will use.
7. 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 filename matches the specification.
- Make sure your functions are named correctly and take the correct inputs.

Documentation and submission (4 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!).

You are also required to submit your assignment in the format specified.

0 Graph class

For both tasks in this assignment, the input is a graph. Although there are some differences between the graphs, both of them have a set of vertices, a set of undirected edges, and a **property** value for each vertex.

Generally, it is good practice to use a class for representing complex objects (like a graph). Therefore, it is highly recommended that you construct a graph class to store information about your graph, and use it in both task 1 and task 2.

Whether or not you choose to use a graph class, you will need to store the graphs given as input. The input file formats are described below. At the start of each of your functions for Tasks 1 and 2, you will need to process these files into an appropriate form. Note that these files encode the problem, but you are free to do whatever you like with the data, as long as you solve the problem.

0.1 Input

Graphs will be represented by two files, `vfile` and `efile`.

`vfile` has a single integer V on the first line, which is the number of vertices in the graph. V lines follow, each consisting of two integers. The first integer on each line is a number between 0 and $V-1$ (which are in no particular order), and each number will be unique. These are the vertex IDs. A space will follow, and then the second value, which indicates a property of the vertex. The meaning of the second value is explained in more detail in task 1 and 2. The possible values for this second number are $0, 1, 2$.

`efile` has a single integer E on the first line, which is the number of edges in the graph. E lines follow, each consisting of two integers. Both integers are vertex IDs, and each line represents an undirected edge. In other words, if there is an edge (u, v), then the file will **either** contain a line `"u v"` or a line `"v u"`. It will not contain both.

The graph represented by these files will be **simple** and **connected**.

1 Go analysis (13 marks)

In this task, we will solve a simple problem related to the ancient board game Go. The gameplay itself, while fascinating, is not relevant to this task.

In the game of Go, the board is a graph. Typically this graph is a square grid, where the intersections correspond to vertices, but it can be played on any arbitrary graph, and for this question we will not restrict ourselves to the square grid.

Each vertex of the graph may be empty, or may have a black or white stone placed on it. We will refer to these vertices as being **coloured** black or white.

A **chain** is a set of vertices where

- All the vertices in the set are coloured black, or all the vertices in the set are coloured white
- There is a path from each vertex in the set to each other vertex in the set which only passes through vertices in the set
- No vertex in the set is adjacent to any vertex of the same colour which is not in the set

Intuititvely, a chain is a bunch of vertices which are all the same colour, form one big clump and is maximal (i.e. if you have 4 stones in a row, we don't say that 3 of them form a chain, we need to include all four).

Chains can be **captured** or **free**. A chain is **free** if there is at least one empty vertex adjacent to some vertex in the chain. A chain is **captured** if it is not **free**.

In this task, you will write an function `captured_chains(vfile, efile)`, which finds all captured chains present in a particular state of a Go game.

Note If you know the rules of Go/how to play, please note that the input positions may not be possible states of a Go game, see the example for one such possibility.

1.1 Input

the input to this function are the two files, `vfile` and `efile`, in the format described in Section 0. The **property** value of each vertex corresponds to the colour of that vertex, as follows:

- 0: empty vertex
- 1: vertex coloured black
- 2: vertex coloured white

1.2 Output

The output of `captured_chains()` is a list of lists. The number of internal lists is equal to the number of captured chains in the input graph. Each list corresponds to a different chain. Each

interior list contains the vertex IDs of all the vertices in the captured chain to which it corresponds (in no particular order). The interior lists may also be in any order.

If there are no captured chains, then `captured_chains()` returns an empty list.

1.3 Example

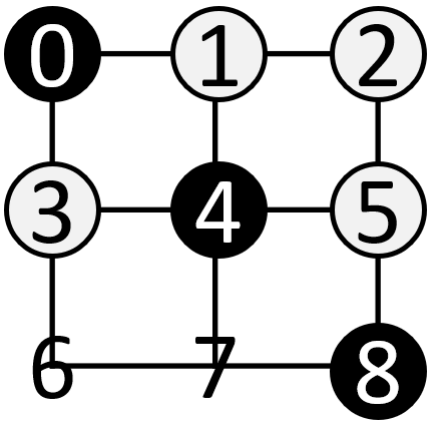
`vfile`

```
9
0 1
1 2
2 2
3 2
4 1
5 2
6 0
7 0
8 1
```

`efile`

```
12
0 1
1 2
0 3
1 4
2 5
3 4
4 5
3 6
4 7
5 8
6 7
7 8
```

Visual representation of the input (intersections correspond to vertices)



Output

`[[0], [1, 2, 5]]` (the two lists, and the 1,2,5 in the second list, can be in any order)

1.4 Complexity

`captured_chains` must run in $O(V+E)$ where

- V is the number of vertices in the input graph
- E is the number of edges in the input graph

2 Terrain pathfinding (13 marks)

In this task, we will find the quickest path for a vehicle through difficult terrain. There are three types of terrain,

- Plain
- Hill
- Swamp

The vehicle in question is highly advanced, and is capable of transforming into three forms,

- Wheel form
- Tank form
- Hovercraft form

The vehicle travels at different speeds based on the combination of terrain type and the current form. However, it also takes time for the vehicle to transform, so it may not always be optimal to transform into the appropriate form before traversing a certain type of terrain.

The problem is represented by a graph, where each vertex corresponds to an **area** of one type of terrain. The vehicle can travel from one area to another along the edges of the graph (which represent two areas being adjacent).

We will apply some simplifying assumptions to this problem.

- The vehicle always starts in **wheel** form.
- While the vehicle is transforming, it cannot move.
- Transforming can only be done immediately after arriving at a new vertex, or at the very start of the problem. In other words, you cannot cross part of a vertex, transform, and then continue.
- Regardless of which neighboring vertex you travel to, you must pay the full cost for traversing the current vertex.

You will write a function, `terrain_pathfinding(vfile, efile, crossing_time, transform_cost, start, end)` which determines the optimal way to traverse a graph made up of areas of various terrain types.

2.1 Input

`vfile` and `efile` are the files described in Section 0, and represent the graph. The property value indicates the terrain type of that vertex, as follows:

- 0: plain
- 1: hill
- 2: swamp

`start` and `end` are each a vertex ID of some vertex the graph. The vehicle starts at `start`, and you need to find the fastest way of reaching `end`. The vehicle will need to traverse the area represented by `start`, but does not need to traverse area represented by `end`.

The `transform_cost` is a non-negative number (possibly non-integer), representing the number of hours it takes for the vehicle to transform from any form to any other form.

`crossing_time` is a dictionary, containing three dictionaries of three elements each. The cost of form `x` crossing terrain `y` (in hours) is given by `crossing_time[x][y]`. These values will be non-negative numbers (possibly non-integer) The keys for the dictionaries are the strings `"wheel"`, `"tank"`, `"hover"`, `"plain"`, `"hill"`, `"swamp"`. Note that you **will not** construct this dictionary yourself. It is one of the parameters. You may assume that the cost of accessing any element of `crossing_time` is $O(1)$.

For example, to obtain the time for crossing hills in hovercraft form, you could write the line of code
`crossing_time["hover"]["hill"]`

2.2 Output

`terrain_pathfinding()` returns a two-element tuple. The first element of the tuple is the number of hours it will take to travel from `start` to `end` by the optimal route.

The second element is a list of tuples. This list represents the sequence of vertices which are traversed by the optimal route, along with the forms in which those vertices should be traversed.

The first element of each tuple in the list is a vertex ID, and the second is one of the following strings: `"wheel"`, `"tank"`, `"hover"`. The string indicates the form that the vehicle should take when crossing that vertex. The tuples must appear in the same order as the vertices on the optimal route from `start` to `end`.

If there are multiple possible solutions, you only need to return one. If the start and end vertex are the same, return `(0, [])`

2.3 Example

```
crossing_time = {"wheel": {"plain": 1, "hill": 2, "swamp": 3},
                 "tank" : {"plain": 3, "hill": 1, "swamp": 2},
                 "hover": {"plain": 2, "hill": 3, "swamp": 1}}
```

2.3.1 vfile

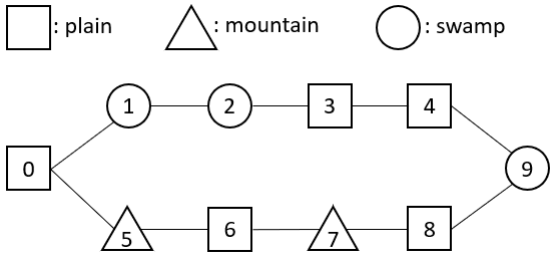
```
10
0 0
1 2
```

```
2 2
3 0
4 0
5 1
6 0
7 1
8 0
9 2
```

2.3.2 efile

```
10
0 1
1 2
2 3
3 4
4 9
0 5
5 6
6 7
7 8
8 9
```

Visual representation of the input



Output

```
>>> terrain_pathfinding(vfile, efile, crossing_time, 2, 0, 9)
(7,[(0,'wheel'),(5,'wheel'),(6,'wheel'),
(7,'wheel'),(8,'wheel'),(9,'wheel')])

>>> terrain_pathfinding(vfile, efile, crossing_time, 1, 0, 9)
#two possible answers
(7,[(0,'wheel'),(5,'wheel'),(6,'wheel'),
(7,'wheel'),(8,'wheel'),(9,'wheel')])

(7,[(0,'wheel'),(1,'hover'),(2,'hover'),
(3,'wheel'),(4,'wheel'),(9,'wheel')])

>>> terrain_pathfinding(vfile, efile, crossing_time, 0.5, 0, 9)
```

```
(6.0,[(0,'wheel'),(1,'hover'),(2,'hover'),
(3,'wheel'),(4,'wheel'),(9,'wheel')])
```

2.4 Complexity

`terrain_pathfinding` must run in $O(Elog(V))$ where

- E is the total number of edges in G
- V is the total number of vertices in G

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!