FIT2004 S1/2021: Assignment 4 - Graph Algorithms

DEADLINE: Friday 22nd October 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 **or** use the google form linked on the Moodle page for short (< 5) day extensions.

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 to solve the assignment 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)

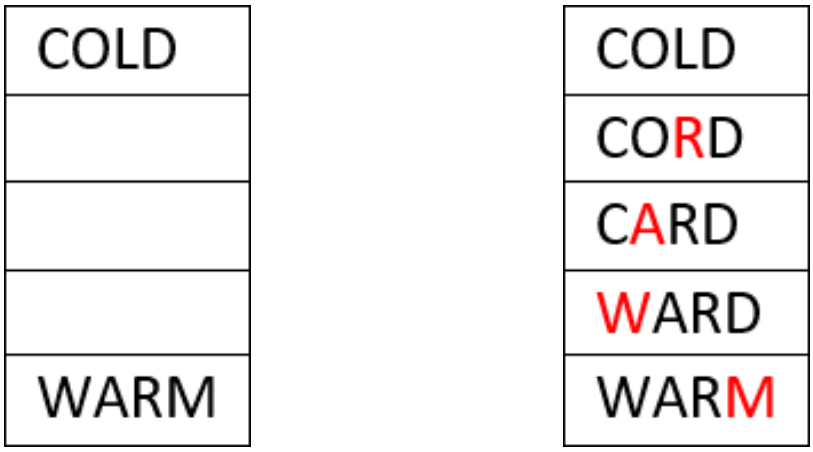
Documentation (3 marks)

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!).

Background

A word ladder is a series of words, all the same length, where each word differs from the previous word by exactly one character. In this assignment we will refer to the first word as the **start_word** and the last word as the **target_word**. A word ladder puzzle is normally presented by giving the **start_word**, then some number of empty spaces, then the **target_word**. To solve the puzzle, the player must find a sequence of words that completes the chain. These words are called the intermediate words. In general, a solution to a word ladder will not involve repeated words, since this loop could be removed without invalidating the solution. An example of such a puzzle and one possible solution are:



Graph class (0 Marks)

This assignment is about graphs and graph algorithms. As such, both problems relate to graphs in some way. You must write a class `WordGraph` , and the solutions to the tasks in this assignment will be implemented as methods of that class.

Both tasks relate to word ladders. An instance of `WordGraph` represents all the words we are allowed to use in order to construct word ladder. In other words, you will not be constructing word ladders from all possible English words, but rather from whichever strings are stored in the graph you are working with.

The `__init__` method of `WordGraph` must take as input a nonempty list of words and construct an appropriate graph from them. The words will contain only lowercase a-z characters, and will all be **the same length**, which will not be 0.

In later tasks, we will sometimes refer to a word by its index in this input list (i.e. we associate each word/vertex with an index $0..n-1$ in the usual way).

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 Word ladders (10 marks)

Consider the following problem: You are given a collection of words, all of which are **target words** for word ladders. You need to find the **start word** for which finding word ladders to all the targets is as easy as possible.

To solve this problem, you will write a method of your `Graph` class, `best_start_word(self, target_words)`.

1.1 Input

`target_words` is a nonempty list of indices of words in the graph.

1.2 Output

`best_start_word` returns an integer, which is the index of the word in the graph which produces the overall shortest word ladders to each of the words in `target_words`.

More precisely, it returns the index of the word for which the **longest** word ladder to **any** of the words in `target_words` is **as short as possible**.

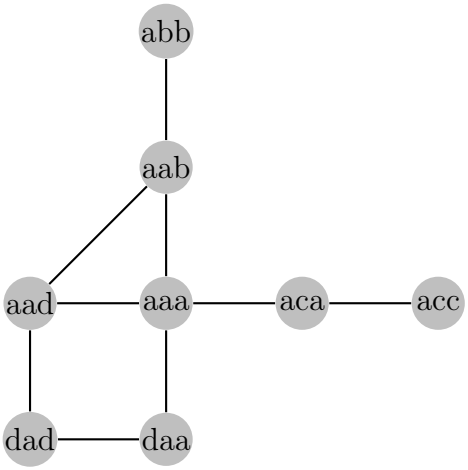
If there are multiple such words, just return one arbitrarily (i.e. it does not matter which one you return, but only return one).

If no such word exists, return -1.

1.3 Example

```
words = ["aaa","aad","dad","daa","aca","acc","aab","abb"]
g = WordGraph(words)

#target words are dad, abb, acc. Best start word is aaa
print(g.best_start_word([2,7,5]))
>>> 0

#target words are dad, aab. Best start word is aad
print(g.best_start_word([6,2]))
>>> 1

#target words are aaa, aca, acc. Best start word is aca
print(g.best_start_word([0,4,5]))
>>> 4
```

To visualise these examples, the graph below is provided.



1.4 Complexity

`best_start_words` must run in $O(W^3)$ time where

- W is the number of words in the instance of `WordGraph`

Note that this time complexity is **larger** than needed in most cases. There are several simple optimisations which can be made, in order to reduce the complexity of the solution for various classes of graphs. Doing this is not necessary for full marks.

2 Constrained word ladders (17 marks)

In this task, we want to construct a word ladder with some additional constraints.

The **first** constraint is that we want to minimise the **alphabetic distance** of the word ladder. The alphabetic distance of a word ladder is the sum of the alphabetic distances between each pair of consecutive words in that ladder.

We define the alphabetic distance between two words in a ladder as the difference in alphabetic position between the letters in the two words which are different. For example,

- "cat" and "bat" have distance 1, since "b" is the second letter of the alphabet, and "c" is the third.
- "bat" and "bet" have distance 4, since "a" is the first letter of the alphabet, and "e" is the fifth.
- ""bet" and "wet" have distance 21, since "b" is the second letter of the alphabet, and "w" is the twenty-third

So the alphabetic distance of the word ladder "cat" → "bat" → "bet" → "wet" is 1+4+21=26.

The **second** constraint is that we need to use one of a set of particular words in the ladder.

To solve this problem, you will write a method of `WordGraph`, `constrained_ladder(self, start, target, constraint_words)`

2.1 Input

`start` and `target` are indices of vertices. `start` is the index of the word where the word ladder must start, and `target` is the index of the word where the word ladder must end.

`constraint_words` is a list of indices. **At least one** of these words must appear in the word ladder.

2.2 Output

`constrained_ladder` returns the list of indices of vertices (in order) corresponding to words representing the word ladder which

- starts with `start`
- ends with `end`
- contains at least one word from `constraint_words` (it may contain more than one, and the word can appear at any point in the word ladder, including the first or last word)
- has the **minimum alphabetic distance** out of all word ladders which satisfy the first three conditions.

If there are multiple word ladders which satisfy all of the above properties, return any **one** of them.

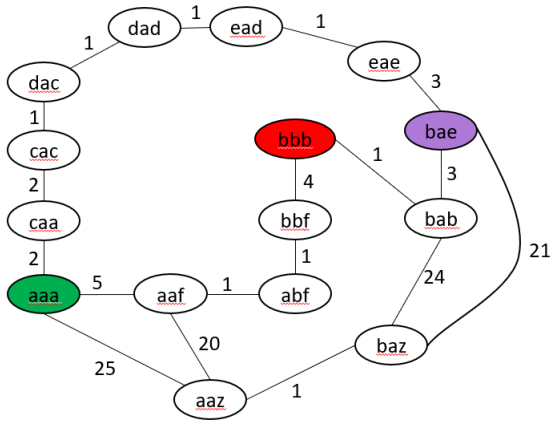
If there are no such ladders, return `None`.

2.3 Example

```
words = ['aaa','bbb','bab','aaf','aaz','baz','caa','cac',
         'dac','dad','ead','eae','bae','abf','bbf']
start = 0
end = 1
detour = [12]
g = WordGraph(words)
g.constrained_ladder(start, end, detour)
>>>[0, 6, 7, 8, 9, 10, 11, 12, 2, 1]

#this corresponds to the list of words
#['aaa', 'caa', 'cac', 'dac', 'dad', 'ead', 'eae', 'bae', 'bab', 'bbb']

detour = [2]
g.constrained_ladder(start, end, detour)
>>>[0, 3, 13, 14, 1, 2, 1]

#this corresponds to the list of words
#['aaa', 'aaf', 'abf', 'bbf', 'bbb', 'bab', 'bbb']
```

This graph corresponds to the first of the two examples above. Green indicates the start vertex, red the end vertex, and purple the constraint vertices (only one vertex is a constraint in each example above). The numbers on edges represent the alphabetic distance between words.

The second example **differs from the diagram** in that "bab" is a constraint, instead of "bae".



2.4 Complexity

`constrained_ladder` must run in $O(Dlog(W) + Wlog(W)))$ where

- D is the number of pairs of words in `WordGraph` which differ by exactly one letter
- W is the number of words in `WordGraph`

Note that the required complexity **does not** involve any term related to the size of `constraint_words`. In other words, the algorithm should run in the same time complexity regardless of how large this list is.

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!