FIT2004 S1/2017: Assignment 4 (Weight: 6 Marks)

THIS PRAC IS ASSESSED! Interview during Week 10 Lab

Submission deadline: Midnight 11:59pm Sun 7 May 2017

CLASS: This programming exercise has to be prepared in advance, and submitted via unit's Moodle page by 11:59pm Sunday 7 May 2017. Implement this exercise strictly using **Python programming language**. During your **assigned** Week 10 lab, your demonstrator will interview you based on your **submitted** version by downloading it from Moodle. This practical work is marked on the performance of your program **and also on your understanding of the program**. A perfect program with zero understanding implies you will get **zero** marks! "Forgetting" is not an acceptable explanation for lack of understanding. You must write all the code yourself, and may not use any external library routines that will invalidate the assessment of your learning. The usual input/output and other basic routines are exempted
DEMONSTRATORS are not obliged to mark programs that do not compile or that crash. Time allowing, they will try to help in tracking down errors, but they are not required to mark programs in such a state, particularly those that do not compile. Therefore, keep backup copies of working partial solutions.
OBJECTIVE: This exercise will give you practice on modeling algorithmic scenarios into graph description, and implementing graph traversal and path-finding algorithms on those modelled graphs. Path and traversal problems involving graphs are ubiquitous across many areas of science, and hence repeated practice on them is necessary to become proficient.

Supporting Material

To complete this assignment, make use of the supporting material uploaded on FIT2004s Moodle page under the `MyAssessments` section.

Background

Consider the following definitions, given an undirected and unweighted graph G:

Distance The *distance* between two vertices x and y of G, denoted by $d(x, y)$, is the length of the shortest path between these vertices. When there is no path between the two vertices, the distance $d(x, y) = \infty$.

Diameter The *diameter* of a graph is the **maximum** $d(x, y)$ over all vertices x and y of G.

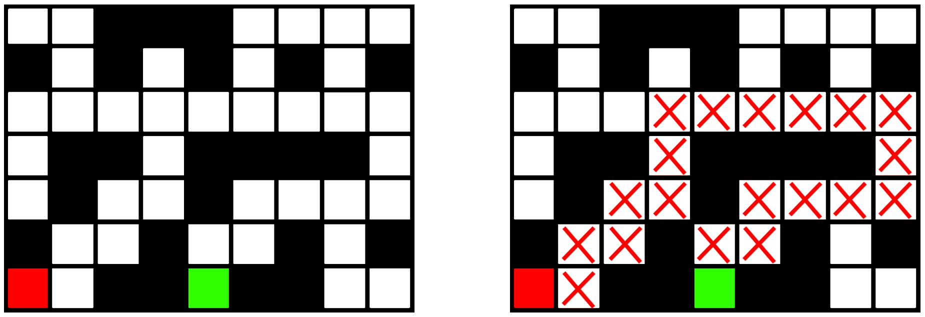
Connected graph A graph is said to be *connected* if and only if its diameter is **not** infinite.

Connected component A connected component of a graph G is subgraph S (involving a subset of vertices in G) such that:

- for any pair of vertices x and y in S we have $d(x, y) < \infty$, and
- there exists no vertex z in G that is **not** in S with $d(x, z) < \infty$, where x is any vertex in S. In other words, the subgraph contains a **maximal subset** of vertices in G that are connected.

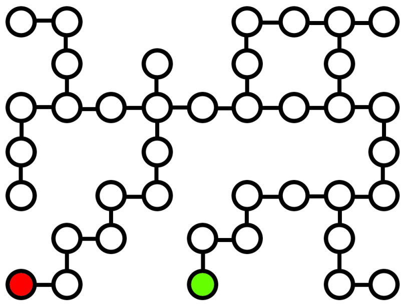
Task 1 of 2[*]

Everybody loves mazes, yet solving them can be tricky. Task 1 is to write a program that helps us solve a maze. We will consider a maze as an $n \times m$ grid, where exactly one of the cells on the grid is the entrance, and one is the exit (these cells might not necessarily be on the boundary, perhaps we are being dropped into the maze by a helicopter!) Every other cell of the grid is either a wall, which we cannot pass through, or it is open space that we can walk into.



An example maze. Black cells denote impassable walls. White cells denote open spaces. Green and Red cells denote Entry and Exit cells respectively. The cells with Red crosses denote the shortest path through the maze connecting the Entry and Exit cells.

We will solve the maze problem by viewing them as undirected, unweighted graphs, and then seeking the shortest path from the Entry to the Exit cell. To view the maze as a graph, imagine each White cell (open space) as a vertex, then connect neighbouring White cells with an edge.



The graph corresponding to the maze shown above.

To complete this task you will be given mazes (see supporting material) encoded as an $n \times m$ grid of ASCII symbols using the following scheme:

[*]Task 1 credit: Daniel Anderson

- the '#' (hash) symbol denotes an impassable wall,
- the '.' (full-stop) symbol denotes open space,
- the letters 'S' and 'F' denote Entry (Start) and Exit (Finish) respectively.

Your task is to solve the maze. For example, the maze above would be encoded as:

```
..###....
#.#.#.#.#
.........
.##.####.
.#..#....
#..#..#.#
F.##S##..
```

The solution to the maze, using the symbol `o` to denote the shortest path would look like:

```
..###....
#.#.#.#.#
...oooooo
.##o####o
.#oo#oooo
#oo#oo#.#
oo##o##..
```

You may assume that the maze always has a valid solution (we will not give you input file where the exit is impossible to reach from the entrance.)

Specifications for implementing Task 1

- Your program should be called `solveMaze.py`
- Your program must accept as an argument, the name of an ASCII text file containing a maze in the format described above. (See the sample input file Task 1 in the supporting material.)
- Your program should output the solution to the maze in the format described above to an output file called solution.txt.
- Your program should be capable of solving mazes of size up to 500×500. A sample maze of size 500×500 is provided in the supporting materials section.

Efficiency considerations

Think carefully about how you choose to represent your graph. For large mazes, certain representations might be less efficient, or even impossible given the amount of memory that they would require. Each vertex in the resulting graph has a small maximum number of possible edges, and these edges follow a very regular pattern, so use this to your advantage.

Task 2 of 2

Task 2 is to write a Python program that builds a special graph from a given dictionary of English words and computes some interesting statistics about the resulting graph.

Your program will be given a file with a long list of **5 letter words** from the English alphabet. Construct a graph *G* where each word in this list is a vertex in the graph. There is an edge between any two vertices/words if one word can be transformed into another by **editing exactly one letter**.

Reading this graph, write a program to compute: (1) all the connected components of the graph, and (2) the diameter of each connected component of the graph.

Specifications for implementing Task 2

- Your program should be called `wordGraph.py`
- Your program must accept as an argument, the name of an ASCII text file containing a list of 5 letter words in the English language, given on individual lines. (See `sample_task2_input.txt` in the supporting material for a sample input file.)
- For the resulting graph, you should output the information of all connected components and their corresponding diameters to the output file called `components.txt`. (See `sample_task2_output.txt` in supporting material for the specified output format.)
  - Connected components should be printed out in the decreasing order of their diameters. (Two components with the same diameter can appear in any order). For any connected component, print its component number (in the order you are printing), the number of vertices it contains, and its diameter on a separate line as:
    "`--- Component = c, nVertices = n, Diameter = d`".
    The vertex set information of each connected component should be printed out as a sorted list of words written to individual lines, with a blank line separating each connected component.

    You must adhere to the format specified in `sample_task2_outout.txt` exactly! You are free to use python in-built sort for this output generation.
- Your program should be capable of processing `words5letter.txt` containing 5757 distinct five-letter words (see supporting material).

```
-=o0o=-
  END
-=o0o=-
```

NON-EXAMINABLE EXTENSION TO TASK 2:

Don Knuth in his treatise, *The Art of Computer Programming*, noted that Lewis Carroll invented a game (circa 1877) called Word-Links or Doublets. This game involved the transformation of one word to another word by changing a letter at a time. For example:

```
tears --> sears --> stars --> stare --> stale --> stile --> smile
```

In fact, for the words given in `words5letter.txt`, the shortest transformation corresponds to the shortest path between any two specified vertices (that are connected). Consider extending your Task 2 program to print out such paths (for fun)!