Introduction to Algorithms: 6.006
Massachusetts Institute of Technology October 19, 2010
Professors Konstantinos Daskalakis and Patrick Jaillet Handout 4

Problem Set 4

This problem set is divided into two parts: Part A problems are theory questions, and Part B problems are programming tasks.

Part A questions are due **Tuesday, November 2nd** at **11:59PM**.

Part B questions are due **Thursday, November 4th** at **11:59PM**.

Solutions should be turned in through the course website. Your solution to Part A should be in PDF form using LaTeX or scanned handwritten solutions. Your solution to Part B should be a valid Python file, together with one PDF file containing your solutions to the two theoretical questions in Part B.

Templates for writing up solutions in LaTeX are available on the course website.

Remember, your goal is to communicate. Full credit will be given only to the correct solution which is described clearly. Convoluted and obtuse descriptions might receive low marks, even when they are correct. Also, aim for concise solutions, as it will save you time spent on write-ups, and also help you conceptualize the key idea of the problem.

Part A: Due Tuesday, November 2nd

1. (**15 points**) Word Ladders

   A common word game is to take two English words, w_1 and w_2, of the same length and try to traverse a "word ladder" from w_1 to w_2 by changing one letter at a time. The challenge of the game is to ensure that each time you change a letter, you still have a valid English word. For example, a word ladder from HEAP to SORT is

   HEAP, HEAT, PEAT, PERT, PORT, SORT

   There may be many or no word ladders for some words.

   For any word w, we assume that there are at most e English words within one letter of w and that w can reach at most W other words. For two words w_1 and w_2, let L_{w_1,w_2} be the number of words in the shortest word ladder between w_1 and w_2.

   (a) (**10 points**) Given two words of the same length, w_1 and w_2, give an algorithm for finding a word ladder (it need not be the shortest word ladder) between w_1 and w_2 or declaring that one does not exist. You may assume that you have access to a function IsWord that runs in constant time and takes any string and returns TRUE if that string is a valid English word and FALSE otherwise. Do your runtime analysis in terms of e, W, and L_{w_1,w_2}.

   (b) (**5 points**) Now assume you are given access to the function D that takes two words w_1 and w_2 and returns the length of the shortest word ladder from w_1 to w_2. Explain how

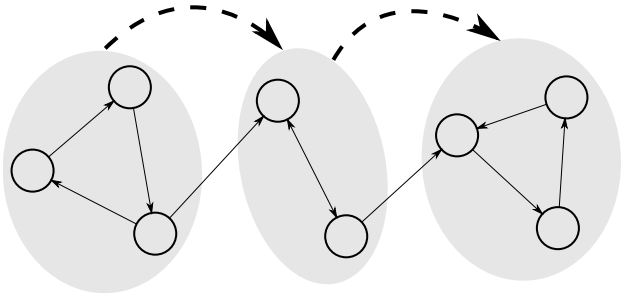


Figure 1: The connected clusters of a graph. The connected clusters are shown in gray. The cluster graph consists of the connected clusters and the edges shown by the thick dashed lines.

to use D to speed up your algorithm. Assuming $D(w_1, w_2)$ runs in constant time, what is now the running time of the algorithm in terms of L_{w_1,w_2}, e, and W?

2. **(15 points)** Cycle Detection

   A cycle is a path of edges from a node to itself.

   (a) **(7 points)** You are given a directed graph $G = (V, E)$, and a special vertex v. Give an algorithm based on BFS that determines in $O(V + E)$ time whether v is part of a cycle.

   (b) **(8 points)** You are given a directed graph $G = (V, E)$. Give an algorithm based on DFS that determines in $O(V + E)$ time whether there is a cycle in G.

3. (**20 points**) Clustering

   Given directed graph $G = (V, E)$, a *connected cluster* of the graph is a set of vertices $U \subseteq V$ such that each vertex $u \in U$ can reach all other vertices in U. For example, a cycle is a connected cluster.

   The *connected clusters* of a graph G are a set of connected clusters $\{c_1, c_2, ..., c_n\}$ where each vertex of the graph only belongs to one cluster and the clusters are maximally sized. A cluster is maximally sized if there is no other set of vertices we could add to it and still have a connected cluster. In other words, a cluster c is maximally sized if there exists no two vertices $v \in c$ and $u \notin c$ such that v can reach u and u can reach v. An example of the connected clusters of a directed graph is shown in Figure 1.

   We consider that a vertex can reach itself trivially so that for a graph with no edges, the connected clusters are just each vertex individually.

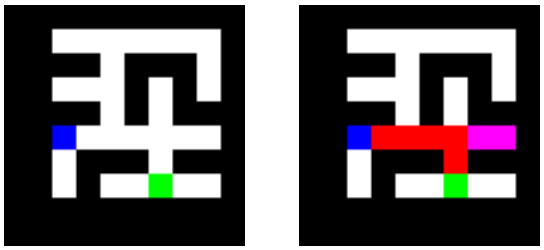
   (a) (**10 points**) The transpose of a directed graph $G = (V, E)$ is the graph $G^T = (V, E^T)$ where E^T is the edge set E with the directions of each edge flipped. G^T has the same connected clusters as G. Specifically, if and only if a vertex v can reach a vertex u in G *and* in G^T, v and u are in the same connected cluster. Use this fact (you do not need to prove it) to write an algorithm for finding the connected clusters of a directed graph

represented using adjacency lists in $O(V + E)$ time. You should give a running time analysis for your algorithm and a proof of correctness.

(b) (**5 points**) Once we have identified the connected clusters of the graph, we can create another graph, the cluster graph, $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ from these clusters. The vertices of $\mathcal{G}$ are the connected clusters of G so that each vertex of $\mathcal{G}$ is a set of the vertices of G. We draw an edge from a cluster $c_1 \in \mathcal{V}$ to a cluster $c_2 \in \mathcal{V}$ if, in G, there is a directed edge from a vertex $v_1 \in c_1$ to a vertex $v_2 \in c_2$. In other words, we draw a directed edge from c_1 to c_2 if some vertex in c_1 can reach some vertex in c_2. An example of a cluster graph is shown in Figure 1

Show how to modify your algorithm from Part 3a to output the cluster graph, $\mathcal{G}$, rather than just the connected clusters of G. You are already computing $\mathcal{V}$ so you just need to show how to find $\mathcal{E}$. This should not change the running time of your algorithm.

(c) (**5 points**) Argue that $\mathcal{G}$ is a DAG. (*Hint: Argue by contradiction.*)



'W'	'W'	'W'	'W'	'W'	'W'	'W'	'W'	'W'	'W'
'W'	'W'	'O'	'O'	'O'	'O'	'O'	'O'	'O'	'W'
'W'	'W'	'W'	'W'	'O'	'W'	'W'	'W'	'O'	'W'
'W'	'W'	'O'	'O'	'O'	'W'	'O'	'W'	'O'	'W'
'W'	'W'	'W'	'W'	'O'	'W'	'O'	'W'	'W'	'W'
'W'	'W'	'B'	'O'	'O'	'O'	'O'	'O'	'O'	'W'
'W'	'W'	'O'	'W'	'W'	'W'	'O'	'W'	'W'	'W'
'W'	'W'	'O'	'W'	'O'	'O'	'E'	'O'	'O'	'W'
'W'	'W'	'W'	'W'	'W'	'W'	'W'	'W'	'W'	'W'
'W'	'W'	'W'	'W'	'W'	'W'	'W'	'W'	'W'	'W'

Figure 2: A 10x10 maze created using DFS. The starting square is blue and the ending square green. Its solution is shown to the right in red. The purple shows the nodes that DFS expanded in finding this solution. The matrix encoding of the maze is shown on the far right.

Part B: Due Thursday, November 4th

1. (**50 points**) Mazes

   (a) (**20 Points**) You will first write an algorithm for creating a maze. We have written a Maze class for you (in `maze.py`) that encodes a maze as a matrix in the following manner:

   - A 'W' indicates a square with a wall in it
   - An 'O' indicates an open square
   - A 'B' indicates the starting square
   - An 'E' indicates the ending square

   An example of a matrix encoding of a maze is shown in Figure 2. Any given square is adjacent to the squares above, below, and to its left and right. The class also stores the number of rows and columns in the matrix and the coordinates of the starting and ending squares. The coordinates are stored as [row, column].

   Your job is to fill in the CREATE_MAZE method in the `maze_creator_skeleton.py` file to create a maze using depth first search. In maze creation, we begin with a maze of all walls (so a matrix of all 'W's). We do a depth-first search through the maze from a provided ending square. At each square, we push all unseen adjacent squares onto the stack. (Think about the best method for pushing adjacent squares onto the stack - what happens, for example, if you always push the square below the current square onto the stack last?) When a square is popped off the stack, we remove the wall in the square *and* the wall between it and its parent square. Therefore, in the creation of the maze, we think of each square as adjacent to squares 2 away from it so that there is a wall to remove in between. You must keep track of which wall should be removed when a square is popped off the stack.

   Once the maze is created, you must choose a starting square. You can do this randomly or find a better method, but you must ensure that the starting square can reach the ending square. Both starting and ending square should be stored [row, column].

   Feel free to add any helper functions or classes you may require.

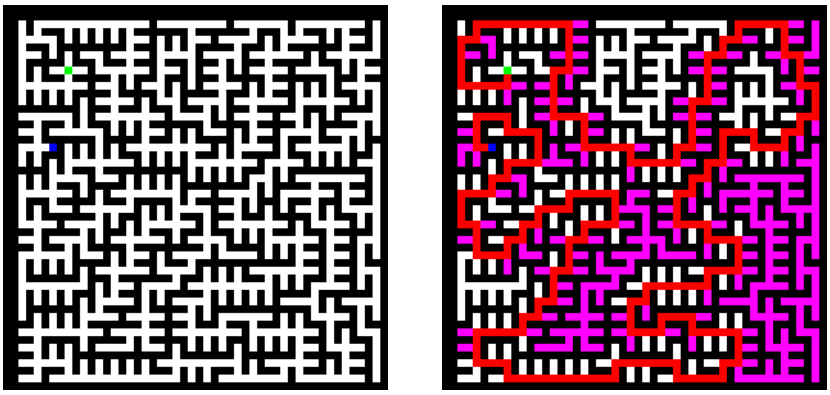


Figure 3: A maze created using DFS and its solution.

At this point, your code will not pass any of the provided unit tests. However, you should look at your output mazes and make sure that they look reasonable. You can test your maze creator by running:

```
python maze_creator_skeleton.py rows columns out-file [scale]
```

This will create a maze of size rows by columns using your creation routine and write it to the output-file, which should end in .ppm or .txt. If it ends in .ppm, your maze will be output as a PPM image. Otherwise, the ASCII representation of your maze will be written to a text file. If you are creating a small maze and writing it out as an image, you can scale the image so that each maze square takes up multiple pixels using the scale factor. For small images, a scale factor of 300/rows is about right. See the code for more details.

An example of a 50x50 maze created using depth first search is shown in Figure 3. You should not find it trivial to solve the mazes output by your algorithm.

We have provided code for you (in `mazeIO.py`) that can read and write mazes from an ASCII file or from a ppm image file using the methods in the ppmIO and asciiIO classes. We highly recommend that you write your mazes out to a ppm file so you can look at them. GIMP, Image Viewer, XV, and FSpot can all open PPM files. You can also use the freely available convert program to convert .ppm to .png or some other more common image format. You will need to do this in order to include images of your created mazes in your writeup. However, if you cannot find any way of reading and converting ppm images, the ASCII output will give you at least some visual interpretation of your maze. Calling `maze_creator_skeleton.py` from the command line with a .txt output file will automatically output an ASCII version of the maze rather than a ppm image.

(b) (**20 points**) Now you will write an algorithm to solve the mazes you have created. A solution to the maze consists of a list of adjacent open squares leading from the starting square to the ending square. A grid square is adjacent to the squares above, below, and to the left and right of it.

You should finish implementing the SOLVE_MAZE method in `maze_solver_skeleton.py`. The SOLVE_MAZE function takes one parameter setting whether it should solve the maze using depth first ("DFS") or breadth first search ("BFS"). You should implement both. The function should return the path found and the nodes expanded during the search. Both the path and the visited node should be lists of [row, column] coordinates indicating squares in the maze. The path should begin with the starting square of the maze and terminate with the ending square of the maze.

Feel free to add any helper functions or classes you may require.

Once you have finished implementing the solver, your code should pass all unit tests (the third test may take a few seconds to run; that's OK). In addition you can look at the output of your code using

```
python maze_solver_skeleton.py in-file out-file search-type [scale]
```

Here input-file should be a ppm image or ASCII file of the type output by MazeCreator with the correct .ppm or .txt extension. The search-type should be either DFS or BFS. You can write your own ASCII files to test your maze if you do not yet have your maze creator working. An example ASCII file is in many_paths.txt.

The solver will write your solution to the output file. If the output file has a .ppm extension, it will output a ppm image of the type in Figures 2 and 3 with the path shown in red and other visited nodes shown in purple. For more details, see the code.

(c) (**10 points**) Is DFS guaranteed to find the shortest path through the maze? Is BFS? On what types of mazes (if any) will DFS expand fewer nodes than BFS? Defend your explanations with examples from your implementation as well as a theoretical argument.