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}.

   Solution: We treat the set of English words as an undirected graph where two words are adjacent if they differ by only one letter. We can find a word ladder in this graph

using DFS or BFS, which runs in time $O(|V| + |E|)$ where $|V| = W$ and $|E| = eW$. However, there were many possible ways to do the runtime analysis for this problem and they hinged on how you found neighbors of a word. If you assumed that you had a neighbor-finding function that used IsWORD and ran in time $O(e)$, then you should have found a running time of $O(eW)$. If you assumed the neighbor-finding function ran in constant time because it was a function of the length of the word or noted that e was upperbounded by this constant then you found a runtime of $O(W)$.

If you used BFS on an implicit graph, you could also say the worst-case running time was $O(e^{L_{w_1,w_2}})$ since BFS is guaranteed to find the shortest path. However, this is not well defined when there is no word ladder so a better solution is $\min(eW, e^{L_{w_1,w_2}})$.

Grading:

- 10/10 for anything like the analyses above
- 8/10 if you gave a correct algorithm but made some minor errors with the runtime analysis.

(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 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?

Solution: Consider the algorithm that, given a word w, scans through all possible children c of w and expands the one with $D(c, w_2) = D(w, w_2) - 1$. Since there is some word ladder of length $D(w, w_2)$ from w to w_2 we know c must exist. This requires visiting $D(w_1, w_2) = L_{w_1,w_2}$ words so you should have found a runtime of $O(eL_{w_1,w_2})$ or $O(L_{w_1,w_2})$ depending on whether you declared in part (a) that e was upper bounded by a constant.

Note that in order for this solution to work, we must consider the graph "implicitly" and not build it ahead of time. Building the graph takes $O(V + E)$, which destroys the advantage given to us by D.

Grading:

- 5/5 for the algorithm above
- 4/5 if you had the right idea but said the number of words between w_1 and w_2 was the length of w_1 rather than L_{w_1,w_2}.
- 3/5 if you gave a search algorithm that only searched to depth $O(L_{w_1,w_2})$ but failed to use the full power of D to find an algorithm linear in L_{w_1,w_2} rather than exponential. Or if you constructed the graph explicitly rather than implicitly.

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.

Solution: Do a BFS from v. If you ever reach an edge (u, v) pointing back to v again during this BFS, then there is a cycle containing v that starts with the path used to get from v to u, and ends with the edge (u, v).

Checking for if a node is v doesn't add any complexity to BFS, so the algorithm still runs in $O(V + E)$ time.

Grading:

- 7/7: For an algorithm like above
- 5/7: If you gave an algorithm that ran in $O(|V| + |E|)$ time and found a cycle but not necessarily one involving v.

(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.

Solution: Run a DFS on G. If you discover a backedge (an edge connecting the current vertex to an ancestor), declare that there is a cycle. Otherwise, there is no cycle. We can check for back edges without affecting the asymptotic running time of DFS, simply by keeping a set of vertices currently on the stack. (Or checking if any children are colored gray in the notation of CLRS.) Note that the set of vertices currently on the stack is *not* the same as the set of vertices that has been visited. Once vertices are popped off the stack (when we backtrack), they are still in the list of vertices that have been visited but no longer in the list of vertices that are on the stack. This was a very common mistake.

Grading:

- 8/8: For an algorithm that successfully found a cycle.
- 6/8: If you said something about "back edges" but also implied that a back edge could connect a current vertex to an already-finished (not on the stack or black) vertex.
- 4/8: If you said that you ran DFS until you found a vertex you had already seen. Or if you assumed you were given v again and rewrote the algorithm from part (a) as DFS.

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.

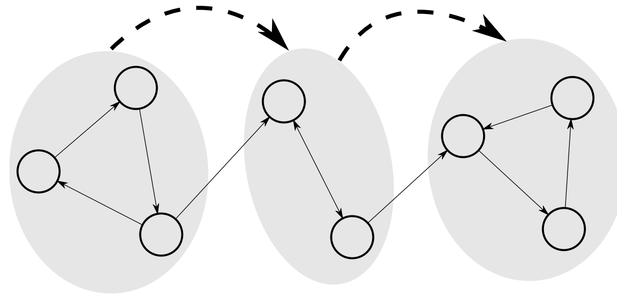

Figure 1: The connected clusters of a graph. The connected clusters are shown in gray. The cluster graph consists of the connected clusters and the edges shown by the thick dashed lines.

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.

Solution: This is Kosaraju's Algorithm. We do a depth first search through G. Then we do a depth first search through G^T except every time we restart it, we use the unexplored vertex with the latest finishing time in G (we could sort this using counting sort or simply keep a stack onto which we push each vertex when we finish it in G).

CLUSTERCOMPONENTS(G, V, E)

1 DFS(G) // *keeping track of finishing times*
2 $G^T \leftarrow$ transpose(G)
3 $c \leftarrow 0$
4 **for** each vertex $v \in V$ taken in decreasing order of finish time
5   **if** v has been visted
6   **continue**
7   DFS-VISIT(v, c) in G^T // DFS-VISIT*(u, c) assigns u's cluster to be c*
8     $c \leftarrow c + 1$

Running Time: We do one DFS through G and one DFS through G^T for a running time of $O(|V| + |E|)$.

Proof of Correctness: For this proof, we use the book's coloring notation so that we denote an unexpanded vertex as white, a vertex that is currently being expanded as gray, and a fully expanded (finished) vertex as black. This is done by initializing every vertex to white, coloring a vertex v gray as the first line of DFS-VISIT(v), and coloring v black immediately before returning from DFS-VISIT(v).

Lemma 1: Let $u, v \in V$ be two vertices of G. Assume u can reach v but v cannot reach u. Then during any DFS run on G, v cannot be gray (currently being expanded) when DFS-VISIT(u) is called.

Proof: We proceed by contradiction. Assume v is gray when we call DFS-VISIT(u). Then we expand u while expanding v, which implies a path from v to u. This is a contradiction.

Lemma 2: Let $u, v \in V$ be two vertices of G. Assume u can reach v but v cannot reach u in G. Then after any DFS on G, the finishing time of u, $f[u]$, will be larger than the finishing time of v, $f[v]$.

Proof: Consider the color of v when we first call DFS-VISIT(u). By Lemma 1, v cannot be gray. It is possible that v is white (unexplored), in which case we will expand v as a descendent of u during DFS-VISIT(u). Since we expanded v after u, we will not return from the expansion of u until after we return from the expansion of v and thus u will have a higher finishing time than v. It is also possible that v is black (already finished); it may have been explored while expanding some other vertex that could reach v. In this case, we have already assigned a finish time to v and, since we have yet to assign a finish time to u, $f[v] < f[u]$. Therefore, the finishing time of v must be less than the finishing time of u.

Correctness: Let $u, v \in V$ be vertices of G. CLUSTERCOMPONENTS places u and v in the same cluster if and only if u can reach v in G and v can reach u in G.

Proof: We consider three cases that fully cover the possible reachability relationships between u and v and show that, in each case, the algorithm places u and v correctly.

i. u can reach v in G and v can reach u in G:
Note that this means that u can reach v in G^T and v can reach u in G^T. Consider the point at which DFS-VISIT(u) is called in G^T. This must be either before the

expansion of v has been started (v is white) or during the expansion of v (v is gray) since if the expansion of v is started before the expansion of u, we must visit u during the expansion of v. If v is white when DFS-VISIT(u) is called then we will expand v while visiting u and u and v will be in the same tree of the DFS on G^T. If v is gray when DFS-VISIT(u) is called then we are expanding u during the expansion of v and clearly u and v are in the same tree of the DFS on G^T.

ii. v can reach u in G, but u cannot reach v in G:
Consider when we call DFS-VISIT(u) during the depth first search of G^T. Assume that u is being expanded as the descendant of some vertex x, on which we originally called DFS-VISIT(x) on line 7 of CLUSTERCOMPONENTS. Note that this means that at this point x had the largest finish time of all unexplored vertices. Now x can reach u in G^T. Therefore, u can reach x in G and, since u cannot reach v in G, x must also not be able to reach v in G. However, since v can reach u in G and u can reach x, v can reach x in G. Thus by Lemma 2, v had a higher finishing time than x in G. Therefore when DFS-VISIT(x) was called, v must already have been expanded and assigned a cluster and v and u cannot possibly be assigned the same cluster.

iii. u cannot reach v and v cannot reach u in G:
Again consider when we first call DFS-VISIT(u) during the depth first search of G^T. Assume we originally called DFS-VISIT(x) for some vertex x with the largest finish time of the unexplored vertices on line 7 of CLUSTERCOMPONENTS. If x cannot reach v in G^T then clearly v will not be explored during a search that expands only descendents of x. If x can reach v in G^T, note that this means v can reach x in G and u can reach x in G. Since u can reach x in G, x must not be able to reach v in G. Therefore, v can reach x in G, but x cannot reach v in G and, by Case ii, x and v will be in separate clusters. Since x and u are in the same cluster, u and v must also be in different clusters.

Grading: This was a tough problem, although it is covered in CLRS under Strongly Connected Components. I was overall very impressed with how well people answered it. In general, I used the rubrik below as a guideline, but extra credit was given for especially creative or well thought-out algorithms even if they were not analyzed completely correctly.

- 10/10 for a working solution that runs in $O(|V| + |E|)$ and a good correctness proof.
- 8-9/10 for a solution that is correct and has an attempt at justification, but not a complete proof depending on how well done the justification was. An incorrect solution that was very close (such as ordering by discovery times rather than finishing times) also received an 8 or 9.
- 7/10 for a slower solution that finds the correct clusters with the correct running time analysis

- 6-7/10 for a correct solution with very little justification or for a solution with significant flaws that still runs in $O(V+E)$
- 5/10 for a working solution that runs in $O(|V|+|E|)$ but no correctness proof.
- 4/10 for a slower solution that finds the correct clusters with an incorrect running time analysis.
- 3/10 for a solution that runs in $O(|V|+|E|)$ time but is not correct and has no correctness proof or a slower solution with an incorrect running time and no correctness proof.

(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.

Solution: After you have finished computing the connected clusters, go through each edge e of G. If this edge connects two vertices in different clusters and there is not already an edge between these clusters, add that edge. We look at each edge once so this just adds $O(|E|)$ and the running time is still $O(|V|+|E|)$.

Grading:

- 5/5 if you gave any algorithm that worked and ran in O(V+E). A number of algorithms would have generated repeat edges and/or self-loops, but since this is easy to deal with, full credit was still given.
- 4/5 if your algorithm could fail to find all edges

Grading:

- 5/5 for anything that basically worked even if it needed another pass to get rid of duplicate edges.
- 4/5 if you wrote an algorithm that was right in principle, but failed to find some edges.

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

Solution: Assume there exists some cycle $\langle c_1, c_2, ..., c_n \rangle$ for $c_1, c_2, ..., c_n \in \mathcal{V}$. Then there is some vertex $v_1 \in c_1$ that has a directed edge to some vertex $v_2 \in c_2$. Since all clusters are connected, v_2 must be able to reach some vertex in c_1, which can reach v_1. Therefore, v_1 and v_2 are connected, contradicting the maximality of c_1 and c_2.

Grading:

- 5/5 for anything like above. You didn't actually have to mention that the clusters were maximally sized if you referenced your algorithm (which was supposed to create maximally sized clusters), but it was better if you did.
- 4/5 if you were close but wrote a proof that was confusing to read or really failed to make use of the fact that the clusters were maximally sized. Or if you just said that a cycle of connected clusters was a connected cluster without drawing on the definition of connected cluster.

Part B: Due Thursday, November 4th

1. (**50 points**) Mazes

Solution: You should have receieved individual feedback about your code. If you are confused by it or would like example solution code for this problem please let us know. The rest of this "solution" lists some common errors and how they were dealt with grade-wise.

Suggestions and Lessons Learned:

- Python lists are natural stacks, but are inefficient queues. Use the Queue or deque class for a better queue implementation.
- In general, check or at least mentally upper bound the runtime of any built-in function that you use. `http://wiki.python.org/moin/TimeComplexity` has some discussion of Python time complexities. Please let me know if you know of something better.
- For DFS, you want to store visited nodes (or colors or finish times or parents etc) in a list of size $|V|$. Give each vertex a number $v_0, ..., v_n$ and then store the visit bool/color/time/parent of v_i in the ith entry in the list. This allows constant time access and makes more sense than a hash table since we know the keys exactly.
- If you are trying to store bool/color/time/parent, create an object with those fields rather than a huge list of them.

Unit Tests: I believe very strongly in penalizing code heavily for failing unit tests that we had provided to you. Writing functional code is good, but it is essentially useless if you cannot write it to a spec so that can then be used to interact with another piece of code. Therefore, if your code failed one of the four unit tests you were given, **half of the credit** for the code that failed was taken away. On ocassion if the error was minor enough I could believe that at some point you passed the tests before introducing it, I only took off one quarter credit.

I added some unit tests and ran all code through them (if you failed a test you were not given, I was, of course, much more lenient). The final testing suite I ran the code through was:

- The four unit tests you were given (BFS, DFS, MazeCreator, BFSvsDFS).
- A more complete version of the BFS, DFS, and MazeCreator tests that checked adjacency of the path.
- A maze with no path through it.
- A simple set of arguments to MAZE_CREATOR to ensure that you actually used the input as the start or finish square.

Everyone should have received a short comment about how they did on these tests. If you had no major running-time errors in your code, it should have been able to complete the entire suite in under 5 seconds.

Style Comments: I tend to make verbose comments about coding style. No one was actually penalized for a coding style decision I did not agree with unless it affected the asymptotic running time of the code. If I thought your code was particularly nice or you had done something very clever, I very ocassionally gave some extra credit.

Double-Counting Mistakes: There were a number of mistakes (especially ones affecting the running time) that it was possible to make in both the maze solver and the maze creator. I only took off for these once. However, if you made a mistake A that slowed the creator down, the same mistake A in the solver *and* another mistake B in the solver, I took off for both A and B.

(a) (**20 Points**) Maze Creator

Grading:

- -10 if your program failed the MazeCreation test you were given for anything other than a minor typo.
- -5 if your program failed the MazeCreation test with a minor typo.
- -3 if you did not randomize the adjacency list so you got a "boring" maze.
- -1 if you kept whole paths on the stack
- The following are common mistakes that affected the aysymptotic running time of the algorithm. While I probably listed everything, your grade should only reflected the mistake you made carrying the highest penalty
  - -3 for using a linear time algorithm to check if a node had been expanded (i.e. "in list" or "count") if this caused you to take longer than 100s on the test since you should have noticed that was taking too long.
  - -2 if you marked squares as seen only after popping them off the stack. This increases the running time of your program since you will pop squares off the stack multiple times. Here is wasn't so bad since the number of adjacent

vertices is upper bounded by 4, but in general that error could lead to an extra factor of O(—V—).

- -1 if you used pop(0) or insert(0) on a list. This is inefficient for lists. If you want to pop(0) you should use a deque or a Queue.

(b) (**20 points**) Maze Solver

Grading:

- -10 if your program failed either (or both) the BFS and DFS tests you were given for any other reason than a minor typo.
- -5 if your program failed either (or both) the BFS and DFS tests you were given for a very minor typo.
- -5 if you only checked if the node had been seen before in the current path rather than in all visited nodes (this usually absorbed the -3 that went along with doing a linear time check of the path)
- -4 if BFS or DFS did not actually expand nodes in the correct order, but still managed to solve the maze.
- -2 if your code went into an infinite loop on a maze with no path.
- -1 if you threw a generic exception for a maze with no path (creating a NoPathError and throwing that would have been awesome) or you returned a path that did not go all the way to the goal without mentioning the fact.
- -1 if you kept whole paths on the queue. This takes up a factor of $|V|$ more space; you should really just keep a parents data structure.
- The following are common mistakes that affected the aysymptotic running time of the algorithm. While I probably listed all of these mistakes, your grade only reflected the one carrying the highest penalty
  - -2 Using a linear time algorithm to check if a node had been expanded (i.e. "in list" or "count"). Usually this was less noticeable in the solver because you expand fewer nodes. Making this mistake in the solver and the creator generally resulted in you losing 3 points for the creator.
  - -2 Marking squares as seen only after popping them off the stack. This increases the running time of your program since you will pop squares off the stack multiple times. Here is wasn't so bad since the number of adjacent vertices is upper bounded by 4, but in general that error could lead to an extra factor of $O(|V|)$.
  - -1 Using insert(0) or pop(0) on Python lists for BFS, because these are linear operations, which can lead to an $O(|V|^2)$ running time. You should have used a deque instead.

(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.

Solution: DFS is not guaranteed to find the shortest path, but BFS is. BFS will generally expand fewer nodes on mazes with many possible paths, but an ultimately a short distance between the starting and ending squares. There were lots of answers for this part though - the intention was really that you play with the solver and creator to get a feel for how the searches worked. So you got credit if you showed examples where DFS expanded fewer nodes than BFS and examples where BFS expanded fewer nodes than DFS and said something reasonable about them.

Grading:

- -1 if you did not show an example where DFS expanded fewer nodes than BFS *and* an example where BFS expanded fewer nodes than DFS.
- +1 (unless you were already at 10/10) for showing a maze where BFS found the shortest path and DFS did not. You would have had to create this maze by hand since the maze creator didn't make mazes with cycles.
- -3 for no empirical analysis at all. Theory is great, but it's always important to back it up with real results.
- -2 if you said that DFS will always expand no more nodes than BFS
- -3 if you did not realize that BFS always finds the shortest path.