CS 330, Fall 2025, Homework 3

Due: Wednesday, 9/24, at 11:59 pm on Gradescope

Homework Guidelines

Collaboration policy If you choose to collaborate on some problems, you are allowed to discuss each problem with at most 3 other students currently enrolled in the class. Before working with others on a problem, you should think about it yourself for at least 45 minutes.

You must write up each problem solution by yourself without assistance, even if you collaborate with others to solve the problem. You must also identify your collaborators. If you did not work with anyone, you should write "Collaborators: none." It is a **violation of this policy to submit a problem solution that you cannot orally explain to an instructor or TA.**

Typesetting Solutions should be typed and submitted as a PDF file on Gradescope. You may use any program you like to type your solutions. LaTeX, or "Latex", is commonly used for technical writing (`overleaf.com` is a free web-based platform for writing in Latex) since it handles math very well. Word, Google Docs, Markdown or other software are also fine.

Solution guidelines For problems that require you to provide an algorithm, you must give the following:

1. a precise (and concise) description of the algorithm in English and pseudocode (*),
2. a proof of correctness,
3. an analysis of the asymptotic running time of your algorithm.

You may use anything we learned in class without further explanation. This includes using algorithms from class as subroutines, stating facts that we proved in class, e.g., correctness of subroutines, running time of subroutines and use the notation. Your description should be at the level that it is clear to a classmate who is taking the course with you.

You should be as clear and concise as possible in your write-up of solutions.

A simple, direct analysis is worth more points than a convoluted one, both because it is simpler and less prone to error and because it is easier to read and understand.

(*) It is fine if the English description concentrates on the high level ideas and doesn't include all the details. But the reader should not have to figure out your solution solely based on the pseudocode. You can also add comments to your pseudocode, in fact that is best practice.

David Bunger, Collaborators: None

Problem 1. *The Quickest Round Trip (10 points)*

Your relatives are in town and insist that you take them on a walk through your neighborhood. You really don't want to go as your class is starting soon and you don't want to be late. Because of this you decide to walk the shortest route possible. Since this is sightseeing, you can't walk along the same stretch of road twice. Your goal is to find a route that will get you out and back home as soon as possible without doubling back.

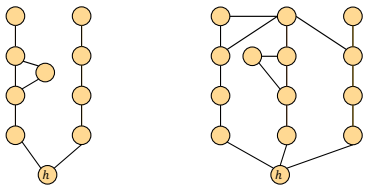
You are given the street map as an **undirected** graph G, where nodes are intersections and edges are roads. As per usual, you may assume that G is given in the form of an adjacency list with the usual implementation. For simplicity, we assume that each street is of equal length and takes the same amount of time to walk. (Thus, G is unweighted.) There is a home node h, from which you depart (along any edge you choose) and to which you must return (along any other edge). A route is valid if it never uses the same edge twice.

Write an algorithm that finds a valid route that uses as few edges as possible.

Inputs The adjacency list of an undirected graph G and home vertex h.

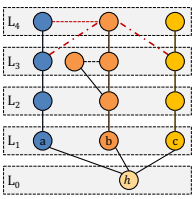
Outputs A Boolean (true/false) indicating whether a valid route from h exists. If one exists, then return the length of the shortest path and a list of nodes on this path. (If there are multiple solutions, you only have to return one.)

For example, in the left-hand graph below, there is no valid route from h. In the right-hand graph, there are three valid routes from h of length 8 (your algorithm just has to find one of them).



For full credit, your algorithm should run in time $O(m+n)$ where $n = |V|$ and $m = |E|$.

Hint: Consider the BFS tree of a graph whose root is h. Moreover, consider subtrees that have each of the immediate children of h as their roots. Observe that an edge connecting two different subtrees creates a valid route. The picture below highlights the different subtrees. The two red edges are part of the shortest cycle(s).



Algorithm 1: QuickestRoundTrip (G, h)

```
   /* G is an adjacency list of an undirected graph.  h is the home vertex                    */
 1 subtree ← {} /* create an empty hash table for the subtrees                                 */
 2 path ← None/* creates a variable for the final path                                        */
 3 for u ∈ G[h] do
 4     subtree[u] = u/* adds each layer 1 node to subtree with its value as itself           */
 5 (Gdist, Gtree, subtree) ← BFSSubtree(G, h, subtree)/* runs a modified version of BFS
       where it adds each node to subtree with the same value as its parent node            */
   /* this results in subtree containing each node within the BFS tree, and each node has the
       value of whichever layer 1 node it came from                                         */
 6 u_final ← None
 7 v_final ← None/* creates nodes to contain the connecting points for the final path       */
 8 for u ∈ G do
 9     for v ∈ G[u] do
           /* iterates through each node and each of its neighbors                         */
10         if subtree[u] ≠ subtree[v] and u ≠ h and v ≠ h then
               /* if a neighbor is in a different subtree and neither node is the starting node,
                  a path has been found                                                     */
11             if u_final == None or (Gdist[u] + Gdist[v] < Gdist[u_final] + Gdist[v_final])
                then
12                 u_final = u
13                 v_final = v
                   /* if the distance of the current path is shorter than the distance of the
                      final path (or if there is no final path), sets the connecting points for
                      the final path to be the current connecting points                    */
14 u_node ← u_final/* creates a node to trace the parents back to h                          */
15 v_node ← v_final/* creates another node to trace the parents back to h from the other side
       */
16 u_path ← []/* creates a list to contain the path back to h from u                       */
17 v_path ← []/* creates the same list for v                                               */
18 while u_node ≠ None do
19     u_path.append(u_node)/* adds the current node to u_path                             */
20     u_node = tree[u_node]/* iterates to the parent of u_node                            */
       /* this creates a path starting with u and following its parents to h               */
21 while v_node ≠ None do
22     v_path.append(u_node)/* adds the current node to v_path                             */
23     v_node = tree[v_node]/* iterates to the parent of v_node                            */
24 v_node.reverse()
   /* this creates a path starting with h and following down the path to v                 */
25 path = v_path + u_path/* creates a full path by combining v_path and u_path             */
26 if path == None then
27     return False/* if no path was found, returns False                                  */
28 return True, path, path.length() − 1/* returns True, the final path, and the length of the
       final path                                                                           */
```

The algorithm creates a subtree hash table to track which of the layer 1 nodes each node is connected to. It fills this table during a modified version of BFS that simply sets a node's subtree to that of its parent node. It then loops through the original graph to find neighboring nodes that are parts of different subtrees. When it finds one, it finds the distance of the path by finding the distance from both connecting nodes to the start. If it is shorter than the distances of the final nodes, it replaces the final nodes with the current ones. Once finding the pair of nodes with the shortest combined distance to the start, it constructs the path by following each node back to the start through its parents. Finally, it returns the results.

We have proved that BFS is able to find the shortest path to any one node. This algorithm simply finds two neighboring nodes part of two different subtrees and checks their distance to the start, then continues until it finds a pair with the shortest distance. Since these distances are calculated correctly by BFS, we know the pair we find will contain the shortest combined path, and therefore find the shortest possible loop.

We know the runtime of BFS is $O(n+m)$. This modified version only adds $subtree[u] \leftarrow subtree[v]$ to each iteration, which is $O(1)$, meaning the modified version of BFS still has a runtime of $O(n+m)$. Next, each edge is checked to see if it connects two subtrees, giving a runtime of $O(m)$. If they do connect, it checks the distance table for each of these nodes and the final nodes. Checking the distance table has a runtime of $O(1)$, so this loop still has a runtime of $O(m)$. Finally, constructing the path has a worst-case runtime of $O(n)$ if every node in the graph is part of the final path. Combined, this gives the algorithm a runtime of $O(n + m)$.

Problem 2. *Elvish languages of Middle-Earth (10 points)*

J.R.R. Tolkien, the author of the famous trilogy Lord of the Rings, was also a professor of philology (the study of language in oral and historical sources). Along with his novels he developed the mythology and origin of Middle-Earth and constructed multiple of its languages [1]. He created the entire Elvish language family with 15 languages, grammar and vocabulary. The most famous of this is the inscription in the One Ring

Recently Tolkien scholars have discovered a so far unknown manuscript which they suspect contains the description of another new language. The language consist of characters never seen by the scholars. There is a list of strings, a.k.a. "words" that the scholars believe are written in alphabetical order. Help the scholars decipher what the alphabet might look like and what the alphabetical order among the characters might be.

We call the discovered language *developed* if we can indeed infer an alphabetical order from the manuscript (note that there may be multiple valid orders). If we cannot, then the language is *incomplete*.

Example 1: a *developed* language.

- \$%%∗
- &&#&
- &&#@

One possible ordering of characters is \$, &, @, %, ∗, #. (Check for yourself that the words are indeed in alphabetical order.)

Example 2: an *incomplete* language.

- \$%%∗
- &&#&
- &&#\$

The second example depicts an invalid list. This is because, if the first two words are in sorted order, then we must have \$ before &; but if the final two words are in sorted order, then we must have & before \$, which is a contradiction.

Note that you are not guaranteed that every letter will appear in the first position (refer to the examples above).

[1] `https://en.wikipedia.org/wiki/Languages_constructed_by_Tolkien`

1. This problem can be represented as a graph problem. Write an algorithm that takes as input the manuscript M. M is a nested *array*, such that $M[i]$ consists of the character array of word i, e.g. the first entry in example 1 is $M[0] = [\$, \%, \%, *]$. You may assume that all words are unique. The output should be the graph adjacency list G as a nested hash table. M consists of W words and the language consists of L distinct characters. For simplicity we assume that the length of each word is the same constant c. (Note, that it is possible that this graph is disconnected, or even may contain singleton nodes.)

2. Write an algorithm that takes as input G and either returns the characters in alphabetical order if such an order exists, or return "no alphabetical order". If there are multiple possible orders, than any one of those is acceptable.

Algorithm 1: LanguageGraph (M)

```
1 G ← {}
2 c = M[1].length()
3 for i ∈ Range(M.length() − 1) do
      /* loops through each word except the last                          */
4     for j ∈ Range(c) do
          /* loops through each character in the current word             */
5         if M[i][j] ≠ M[i + 1][j] then
              /* checks if the character of the current word doesn't match the same character
                 of the next work                                          */
6             G[M[i][j]] = {M[i + 1][j], 1}
7             break
              /* makes a connection from the first character to the second character and then
                 ends the loop                                            */
8 return G
```

With a list of words in alphabetical order, the first character of each word must be in alphabetical order. If two words share the same first character, then the second character in those words must be in alphabetical order. If the second matches, then the third, and so on until the end of the word. This algorithm compares a word with the following word in the list, checks each character until it finds characters that do not match between the words, and then makes a connection from the first to the second. The result is a graph where each character has a connection to all characters that are known to be after it in alphabetical order.

In the worst-case scenario, each word matches aside from their last characters, since all words are assumed to be unique. With this, the runtime for lines 4-7 would have a runtime of $O(c)$. Since the for loop on line 3 runs for each word in M, the total runtime would be $O(Wc)$, where W is the number of words in M.

Algorithm 2: AlphabeticalOrder (G)

```
1 parents, times ← DFSCycle(G)/* runs the version of DFS (shown in lecture) where
    DFS-Visit checks if a node is discovered but unfinished and returns "cycle detected with
    edge (u,v)".  In this case, if DFS-Visit returns this, DFSCycle will return None,None.
    */
2 if parents == None then
3 |   return "no alphabetical order"/* if a cycle is detected within DFSCycle, there is no
  |      alphabetical order possible.                                                    */
4 order ← [] order = sort(times, key = lambda k : times[k][1])/* sorts the list of keys in
    times using the second value in each tuple, or the time finished.                    */
5 order.reverse()/* reverse the order of this list so the characters are in the order of
    latest finish time first.                                                            */
6 return order
```

This algorithm uses DFS to determine whether or not the given graph G contains a cycle or if it is a DAG. This method was demonstrated in lecture by checking if a node had already been discovered but not completed when the algorithm arrived at that node. If there is a cycle in G, that would mean the graph is trying to say a letter earlier in the alphabetical order comes after the current letter, which isn't possible for a correct alphabetical order. If the sub algorithm, $DFS - Visit$, detects a cycle it will return "cycle detected with edge (u,v)." If $DFSCycle$ receives this from $DFS - Visit$, we will have it return $None, None$. Finally, if $DFSCycle$ returns $None, None$, we know either G is empty or there is a cycle, and will return "no alphabetical order possible". Knowing that G is a DAG means there is a valid alphabetical order, and we can find it by finding a topological order of the nodes. To do this, line 4 sorts the nodes in $times$ based on the finish time of each node, or $times[k][1]$. We then reverse this list to get the final list of nodes sorted by latest finish time first. We know from lecture that this is able to produce a valid topological order for a DAG, and we already confirmed that G is a DAG. Since there are no backwards facing edges in this list, we know all the characters come after the previous character in the list, meaning they are now in alphabetical order.

We know the runtime of $DFSCycle$ is $O(n+m)$. We also know the runtime of the sorting algorithm is $O(nlog(n))$. Line 5 needs to interact with each node once to reverse the order, giving a runtime of $O(n)$. Combined, this gives us a total runtime of $O(nlog(n) + m)$.