Algorithm Design and Analysis
CSE222 Winter '20

Assignment 4

Due on : 02.04.2020 (11.59pm)

Full Marks : 100

Please write solutions independent of each other. You are welcome to discuss with fellow students, but strictly prohibited from copying solutions from another person. No external sources other than class notes/lecture notes uploaded may be consulted for solving these problems. Programming assignment can be done in a team of **at the most two persons**. *Violation of any of the above would be considered an act of plagiarism. Please see collaboration policy on course webpage for further information.*

Note: For each of the theory questions, you should prove your claims. For Problems (2) and (3), this proof should involve convincing us that the graphs you have constructed correctly models the problem you are trying to solve. Try to be precise. Meaningless rambles fetch negative credits.

For programming assignment, you need to submit on Google Classroom by the assigned deadline.

Problem 1 (15 points) $G = (V, E)$ is a connected graph, and $u \in V$ is a specific vertex. Suppose the depth-first search tree rooted at u be T that includes all nodes of G. Suppose there is a breadth-first search tree rooted at u which is same as T. Prove that G = T. (In other words, if T is both a depth-first search tree and a breadth-first search tree rooted at u, then G cannot contain any edges that do not belong to T.)

Problem 2 (20 points) Inspired by the example of that great Cornellian, Vladimir Nabokov, some of your friends have become amateur lepidopterists (they study butterflies). Often when they return from a trip with specimens of butterflies, it is very difficult for them to tell how many distinct species they've caught—thanks to the fact that many species look very similar to one another.

One day they return with n butterflies, and they believe that each belongs to one of two different species, which we'll call A and B for purposes of this discussion. They'd like to divide the n specimens into two groups—those that belong to A and those that belong to B—but it's very hard for them to directly label any one specimen. So they decide to adopt the following approach.

For each pair of specimens i and j, they study them carefully side by side. If they're confident enough in their judgment, then they label the pair (i, j) either "same" (meaning they believe them both to come from the same species) or "different" (meaning they believe them to come from different species). They also have the option of rendering no judgment on a given pair, in which case we'll call the pair ambiguous.

So now they have the collection of n specimens, as well as a collection of m judgments (either "same" or "different") for the pairs that were not declared to be ambiguous. They'd like to know if this data is consistent with the idea that each butterfly is from one of species A or B. So more concretely, we'll declare the m judgments to be consistent if it is possible to label each specimen either A or B in such a way that for each pair (i, j) labeled "same", it

is the case that i and j have the same label; and for each pair (i, j) labeled "different", it is the case that i and j have different labels. They're in the middle of tediously working out whether their judgments are consistent, when one of them realizes that you probably have an algorithm that would answer this question right away.
Give an algorithm with running time $O(m+n)$ that determines whether the m judgments are consistent.

Problem 3 (20 points) You have a collection of n lock-boxes and m gold keys. Each key unlocks atmost one box. However, each box might be unlocked by one key, by multiple keys, or by no keys at all. There are only two ways to open each box once it is locked: Unlock it properly (which requires having one matching key in your hand), or smash it to bits with a hammer.Your baby brother, who loves playing with shiny objects, has somehow managed to lock all your keys inside the boxes! Luckily, your home security system recorded everything, so you know exactly which keys (if any) are inside each box. You need to get all the keys back out of the boxes, because they are made of gold. Clearly you have to smash at least one box.

a) Your baby brother has found the hammer and is eagerly eyeing one of the boxes. Describe and analyze an algorithm to determine if it is possible to retrieve all the keys without smashing any box except the one your brother has chosen.

b) Describe and analyze an algorithm to compute the minimum number of boxes that must be smashed to retrieve all the keys.

Problem 4 (45 points) You have to design an algorithm to solve the 2x2x2 Rubik's Cube Puzzle. We have attached a Python library "rubik" that defines a few functions and methodologies so that you only focus on implementing the logic of the algorithm.
You can solve the puzzle by finding the shortest path between the solved configuration and the given input configuration using BFS. A graph for this problem can be formed in the following way: link a configuration A to a configuration B if it takes one twist to reach B from A or vice versa.

a) Given an input configuration, write a BFS to solve the 2x2x2 Rubik's Cube Puzzle. This should return a list of the moves required to reach from the input configuration to the solved configuration.

b) You might notice that your code may take up to a few minutes to solve (and probably crashes a lot of times due to memory shortage). We can use the following fact to design an optimization. The depth of the BFS tree from any node is 14. The tree up to level 7 has much less than half the total number of nodes in the entire tree. So, we can implement a 2-way BFS from the start and the end. The 2 BFSs grow one level at a time. You will have a path from the start to the end when a BFS from either side visits a node that has already been visited by the other BFS. Implement this optimization separately.

c) Report the times of execution for both the methods for a few test cases that you may design on your own and try to explain (informally) why the second design works better.

Write your codes for part a and b in the file "solver.py". Write your answer for part c in a separate txt file.
The input to your program should be the start and end configurations, and the output should be the list of moves required. The description for the configurations is provided in 'rubik.py'.
Test cases:

Input 1
start = (6, 7, 8, 0, 1, 2, 9, 10, 11, 3, 4, 5, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23)
end = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23)

Output 1
['Fi']

Input 2
start = (6, 7, 8, 20, 18, 19, 3, 4, 5, 16, 17, 15, 0, 1, 2, 14, 12, 13, 10, 11, 9, 21, 22, 23)
end = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23)

Output 2
['Li', 'U', 'Li', 'Ui', 'Fi', 'U', 'F', 'U', 'Li', 'U', 'Li', 'F', 'Li', 'Li']

Input 3
start = (7, 8, 6, 20, 18, 19, 3, 4, 5, 16, 17, 15, 0, 1, 2, 14, 12, 13, 10, 11, 9, 21, 22, 23)
end = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23)

Output 3
None (No solution)