CS 170 Algorithms
Fall 2014 David Wagner
HW 4

Due Sept. 26, 6:00pm

Instructions. This homework is due Friday, September 26, at 6:00pm electronically. Same rules as for prior homeworks. See `http://www-inst.cs.berkeley.edu/~cs170/fa14/hws/instruct.pdf` for the required format for writing up algorithms on this and following homeworks.

1. (20 pts.) Compile on a parallel cluster

We want to compile a large program containing n modules. We are given a dependency graph $G = (V, E)$: G is a directed, acyclic graph with a vertex for each module, and an edge from module v to module w means we must finish compiling v before starting to compile w. Each module takes exactly one minute to compile. We want to compile the program as quickly as possible. We are willing to use Amazon EC2 for this purpose, so we can compile as many modules in parallel as we want, as long as we satisfy the dependencies. Find a linear-time algorithm that computes the minimum time needed to compile the program.

2. (25 pts.) Peace for Grudgeville

In Grudgeville, people just can't seem to forgive. Every so often two people get into an argument, and then they hate each other forever after. Hating is a symmetric relation: If Alice hates Bob, then Bob hates Alice, too. People who hate each other can't stand to be in each other's presence. This has put a crimp on the social scene in Grudgeville, and Sheriff Brown is tired of stopping the fights that periodically break out.

Then Sheriff Brown has a flash of insight. There's an island offshore. If he could figure out a way to divide the n townsgoers into two groups, so no one hates anyone else in their group, then he could ship one group out to the island, leave the other behind on the mainland, and everyone would be happy. Can you help keep the peace?

Find an efficient algorithm to check whether such a division is possible. You're given a list of m pairs of people who hate each other. Your algorithm should check whether such a division into two groups is possible, and if so, it should output a roster stating for each person where they will live (on the mainland or on the island).

3. (20 pts.) Model checking

You've written some code to control a traffic light. At each unit of time, it receives some input from the sensors in the road (a boolean for each direction indicating whether cars are waiting), updates its internal state, and outputs which color light should be illuminated in each direction.

Having taken CS 61A, you wrote your code in a clean functional, side-effect-free style. In particular, if `s` represents the current state of the system and `i` represents the sensor input, then calling `update(s,i)` will return a pair `(t,o)` where `t` is the new state of the system and `o` represents the color of the lights in each direction.

Before you can sell your system, you need to convince regulators that it will be safe. In particular, it is vital that it never produce an output `o` that would show a green light in two perpendicular directions, since that

could cause a serious traffic accident. Merely testing your code on a bunch of possible sequences of inputs isn't enough, as there are gazillions of possible sequences, and you'll never be able to try them all. Instead, the regulators want some assurance that a double-green can never happen.

Devise an efficient algorithm to determine whether your code can ever output a double-green signal. You may assume that `s0` represents the initial state of the system when it is first turned on, that there are at most n different possible states (where $n \leq 10^6$ or so), that there are 16 possible inputs (there's one boolean for each direction, so each input is a 4-bit value), and that `update` is written in a clean functional style (no side effects, no global variables, and it behaves deterministically: if you run `update` twice with the same inputs, you'll get the same outputs). You may also assume that you have a helper function `doublegreen(o)` that returns true if `o` would display a green light in two perpendicular directions.

4. (35 pts.) Disrupt the terrorists

Let $G = (V, E)$ denote the "social network" of a group of terrorists. In other words, G is an undirected graph where each vertex $v \in V$ corresponds to a terrorist, and we introduce the edge $\{u, v\}$ if terrorists u and v have had contact with each other. The police would like to determine which terrorist they should try to capture, to disrupt the coordination of the terrorist group as much as possible. More precisely, the goal is to find a single vertex $v \in V$ whose removal from the graph splits the graph into as many different connected components as possible. This problem will walk you through the design of a linear-time algorithm to solve this problem. In other words, the running time will be $O(|V| + |E|)$.

In the following, let $f(v)$ denote the number of connected components in the graph obtained after deleting vertex v from G. Also, assume that initial graph G is connected (before any vertex is deleted) and is represented in adjacency list format. If you get stuck on one part below, continue on to the subsequent parts of the question, since many parts do not strictly rely upon previous parts. Prove your answer to each part.

(a) Perform a depth-first search starting from some vertex $r \in V$. How could you calculate $f(r)$ from the resulting depth-first search tree in an efficient way?

(b) Suppose $v \in V$ is a node in the resulting DFS tree, but v is not the root of the DFS tree (i.e., $v \neq r$). Suppose further that no descendant of v has any non-tree edge to any ancestor of v. How could you calculate $f(v)$ from the DFS tree in an efficient way?

(c) Definition: For each node v in the DFS tree, let $d(v)$ denote the depth of v in the DFS tree. In particular, the root r has depth 0; r's children have depth 1; r's grandchildren have depth 2; and so on.

Describe how to compute $d(v)$ for each vertex $v \in V$, in linear time. (You can assume the vertices are numbered $0..n-1$, so your goal is to build and initialize an array $\mathtt{d}[0..n-1]$ so that $\mathtt{d}[v]$ holds the depth of v.)

(d) Definition: If w is a node in the DFS tree, let $up(w)$ denote the depth of the shallowest node y such that there is some graph edge $\{x, y\} \in E$ where either x is a descendant of w or $x = w$. We'll define $up(w) = \infty$ if there is no edge $\{x, y\}$ that satisfies these conditions.

Now suppose v is an arbitrary non-root node in the DFS tree, with children $w_1, \ldots, w_k$. Describe how to compute $f(v)$ as a function of k, $up(w_1), \ldots, up(w_k)$, and $d(v)$.

Hint: Think about what happened in part (b); think about what changes when we can have non-tree edges that go up from one of v's descendants to one of v's ancestors; and think about how you can detect it from the information provided.

(e) Design an algorithm to compute $up(v)$ for each vertex $v \in V$, in linear time.

(f) Describe how to compute $f(v)$ for each vertex $v \in V$, in linear time.

5. (5 pts.) Optional bonus problem: More traffic

(This is an *optional* bonus challenge problem. Only solve it if you want an extra challenge.)

You solve Question 3. The regulators thank you for your excellent solution to Question 3, and then mention that they forgot to tell you they have one more requirement: they want to be sure that no car will be stuck forever at a red light, waiting for the light to turn green.

Suppose a car is present and waiting on the Northbound road, waiting for the light to turn green so it can go. If it is possible that the car could be waiting infinitely long for the green light, then your code is no good. In particular, if there exists any infinite sequence of inputs where a Northbound car is waiting infinitely long, then your code is no good. (The regulators don't care exactly what the maximum possible waiting time is, as long as it is finite: they just want assurance that your code will eventually give the car a green light.)

Design an efficient algorithm to determine whether your code is any good, i.e., whether there is any possible scenario where your code could leave a car waiting infinitely long for the next green light. Your algorithm should run in $O(n)$ time. You are not allowed to look at the implementation of `update`, and you are not allowed to assume anything else about the behavior of `update` beyond what is specified in Question 3.