CS 170 Algorithms Fall 2014 David Wagner Sol 4

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.

Solution #1:

Main idea. We're looking for the longest path in this DAG, so topologically sort the graph. Then, for each vertex v (in topologically sorted order), we compute the length of the longest path ending at v, in terms of other lengths previously calculated.

Pseudocode.

```
FindCompletionTime(Graph G):
1.  Linearize the graph G, and let v_1, ..., v_n denote the vertices of G in topologically sorted order.
2.  Create a temporary array ℓ[1..n]. Set m := 1.
3.  For j := 1, 2, ..., n:
4.      Set ℓ[j] := 1.
5.      For each edge (v_i, v_j) ∈ E:
6.          If ℓ[i] + 1 > ℓ[j]:
7.              Set ℓ[j] := ℓ[i] + 1.
8.      If ℓ[j] > m:
9.          Set m := ℓ[j].
10. Return m.
```

Or, better yet, you could write this more concisely as follows:

```
FindCompletionTime(Graph G):
1'. Linearize the graph G, and let v_1, ..., v_n denote the vertices of G in topologically sorted order.
2'. For j := 1, 2, ..., n:
3'.     Set ℓ[j] = 1 + max(0, max{ℓ[i] + 1 : (v_i, v_j) ∈ E})
4'. Return max{ℓ[j] : 1 ≤ j ≤ n}.
```

Additional explanation. The idea is that $\ell[j]$ will be the length of the longest path ending at vertex v_j. Due to the linearization, any path ending at v_j can only pass through (some subset of) $v_1, v_2, \ldots, v_{j-1}$. Therefore assuming there is at least one edge into v_j,

$$\ell[j] = 1 + \max\{\ell[i] : (v_i, v_j) \in E\}$$

which depends only on $\ell[i]$ for $i < j$. If there are no edges into v_j, we set $\ell_j = 1$. Once we have computed $\ell[1], \ldots, \ell[n]$, the solution to the problem will be $\max(\ell[1], \ldots, \ell[n])$.

Correctness. We start by showing a useful loop invariant. After loop iteration j, $\ell[j]$ is the earliest time that we can finish compiling module v_j.

Base case: Once the graph is linearized, v_1 has no prerequisites. Therefore, it can be compiled in the first minute. Line 4 sets $\ell[1]$ accordingly.

Inductive case: A module can be compiled as soon as all its prerequisites have finished compiling. Therefore, $\ell[j]$ should be the largest of the values $\ell[i]+1$, where i ranges over all prerequisites for v_j. This is computed by the inner loop in lines 5–7.

By the invariant above, $\ell[i]$ is the earliest we can start compiling v_i. So the maximum value of the array is the module with the longest chain of prerequisites, i.e., the earliest time that we can finish compiling the entire program. This maximum value is tracked in m, and updated on each iteration of the outer loop.

Running time. $\Theta(|V|+|E|)$.

Justification of running time. The topological sort in line 1 takes $\Theta(|V|+|E|)$ time. The loop starting on line 3 processes each vertex once, and the inner loop examines each edge exactly once. Thus, this algorithm does $\Theta(1)$ work per vertex and $\Theta(1)$ work per edge, for a total of $\Theta(|V|+|E|)$.

We are assuming that G is represented in a way that allows us to easily discover the list of in-edges to any specified node. The standard adjacency list representation provides the reverse: it associates with each vertex v, the list of all edges out of v, while we want to associate with v the list of all edges into v. This requires augmenting the adjacency list representation with an additional list of in-edges. Fortunately, this can be computed with a simple linear-time computation.

Comment: This illustrates a common design paradigm for algorithmic problems on dags. We iterate through the vertices in topologically sorted order, computing some information about each vertex and saving that information. Think of it as labelling each vertex with some value, that will be helpful for solving the problem. We try to select what information we compute, so that given the labels on all of the predecessors of vertex v, we can easily compute the proper label for vertex v. If we can do this, then the topologically sorted order ensures that by the time we reach v, we'll already have labelled all of its predecessors and we'll have all all the information needed to easily compute v's label.

Foreshadowing: This design technique is closely related to dynamic programming, a technique we'll see later in the course.

Comment: This technique can also be used to compute the "critical path" in a boolean circuit. The critical path is a longest chain of gates such that each gate is dependent on a previous gate in the chain, in other words, the path between input values and output values that has the longest delay. This dictates the latency of a circuit.

Solution #2:

Main idea. We immediately start compiling all of the modules with no prerequisites ("source nodes" in the graph). Then, we can delete all of those modules and any edges out of them, and repeat the same process, until we've finished compiling the program. With suitable data structures, we can make this run in linear time.

Pseudocode.

```
FindCompletionTime(Graph (V,E)):
1.  Set S := ∅ and clock := 0.
2.  For each v ∈ V:
3.          Set deps[v] := the number of edges into v.
```

4. If deps$[v] = 0$, add v to S.
5. While $S \neq \emptyset$:
6. Start compiling all of the modules in S.
7. Set $T := \emptyset$.
8. For each edge $(s,t) \in E$ where $s \in S$:
9. Set deps$[t]$:= deps$[t] - 1$.
10. If deps$[t] = 0$, add t to T.
11. Set $S := T$ and clock := clock $+ 1$.
12. Return clock.

Proof of correctness. The following invariant will hold at the start of any iteration of the while loop (line 6): for each $v \in V$, deps$[v]$ counts the number of edges $(u,v) \in V$ such that u has not been previously compiled in a previous iteration of the loop. (In other words, deps$[v]$ stores the in-degree of v if we imagine deleting all of the vertices that have been previously compiled.)

This can be proven by induction on the number of iterations of the loop. Lines 2–4 ensure that it holds before the first iteration. Also, if it holds before any particular iteration, then lines 8–9 adjust deps$[\cdot]$ to take into account the reduction in number of pending edges, so it will hold after that iteration.

Also, it is easy to see that at each iteration of the loop, S holds the set of vertices $S = \{v : \text{deps}[v] = 0\}$. In the first iteration, this follows from lines 1 and 4. In any subsequent iteration, it follows from the way that lines 6 and 10 work; line 10 checks each vertex for which deps$[\cdot]$ has changed, and if deps$[t]$ became 0, it adds t to T, which ensures it will be in S for the next iteration of the loop.

Thus, this algorithm produces a valid schedule for compilation.

Why does it output the shortest schedule? We will prove it by induction. If we denote with S_i the set of modules that our algorithm compiles when clock $= i$, then our algorithm compiles the modules according to the schedule:

$$S_0 \to S_1 \to \cdots \to S_t$$

where $t+1$ is the time reported from our algorithm. Similarly if we denote with S'_i the set of modules that the algorithm with the shortest schedule compiles when clock $= i$, then the shortest schedule looks like:

$$S'_0 \to S'_1 \to \cdots \to S'_{t'}$$

where $t'+1$ is the time of the shortest schedule. We will prove by induction that, for all $j = 0, 1, 2, \ldots, t'$, by the time $j+1$, our algorithm will have compiled at least the same modules as the schedule with the shortest time. In other words, we will prove that $S'_j \subseteq S_j$ for all j.

Base case: At time $t = 0$ our algorithm includes in S_0 all the sources of G, so by the time $t = 1$ our algorithm has compiled all the sources of G. Actually at $t = 0$ the only modules that can be compiled are the sources of G since the other modules have prerequisites (or equivalently incoming edges in G). Therefore by time $t = 1$ our algorithm will have compiled at least the same modules as the schedule with the shortest time.

Inductive step: Assume that by the time j our algorithm will have compiled at least the same modules as the schedule with the shortest time, i.e., our algorithm has compiled the modules $A = S_0 \cup S_1 \cup \cdots \cup S_j$, the algorithm with the shortest schedule has compiled the modules $B = S'_0 \cup S'_1 \cup \cdots \cup S'_j$ and $A \supseteq B$. Now consider the sets of modules S_{j+1} and S'_{j+1} and let's denote with M_{j+1} the set of modules that exist in S'_{j+1} but not in S_{j+1}. Consider a module $x \in M_{j+1}$. Since x is compiled at time $j+1$ in the shortest schedule, all the prerequisite modules of x are in the set B. Since $A \supseteq B$ all the prerequisite modules of x are also in the set A or equivalently we should have deps$[x] = 0$ by the time $j+1$. But since deps$[x] = 0$ by time $j+1$ and our algorithm has not added x to S_{j+1} (by definition of M_{j+1}), then our algorithm has added x to a S_k with

$k \leq j$ and is already compiled by the time $j+1$. Therefore all elements in M_{j+1} will have been compiled by our schedule by the time $j+1$ and consequently by the time $j+1$ our algorithm will have compiled at least the same modules as the schedule with the shortest time.

Therefore, we conclude that by time $t'+1$ our schedule will have compiled at least the same modules as the schedule with the shortest time. However, by time $t'+1$ the schedule with the shortest time has compiled all modules in G and subsequently our algorithm by time $t'+1$ has also compiled all modules in G which means that our algorithm outputs the shortest time.

Running time. $O(|V|+|E|)$.

Justification of running time. Lines 1–3 can be implemented in $O(|V|+|E|)$ time. Each module is compiled only once so each edge is examined exactly once in line 8. Therefore lines 8–10 will do $O(1)$ work per edge (if we represent S, T as linked lists, adding to them can be done in $O(1)$ time and enumerating over them can be done in $O(1)$ time per item, so each iteration of the loop in lines 8–10 takes $O(1)$ time). This means that lines 5–11 take $O(|V|+|E|)$ time in total. The total running time is $O(|V|+|E|)$.

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).

Solution:

Main idea: We represent the situation in Grudgeville with an undirected graph: the set of vertices is just the set of people, and there is an edge $\{u, v\}$ if person u and person v can't stand each other. The sheriff's problem is essentially to divide the vertexes into two sets, the "island group" and the "mainland group" such that no two nodes with an edge (grudge) are in the same group.

We use a depth-first search traversal of the graph and divide the people into two groups by labelling each person as 0 or 1 (representing "island" and "mainland"). Once one person is assigned a label of a group, everyone they hate (or equivalently everyone to whom they are connected with an edge) must be assigned the opposite label. If we identify during the DFS traversal that we have assigned the same label to two people connected with an edge, we report failure.

Pseudocode:

```
SplitPeople(G)
1.  For all vertices v ∈ V:
2.      Set visited(v) := false
3.  For all vertices v ∈ V:
4.      If not visited(v):
5.          AssignLabels(v,1)
```

```
AssignLabels(v, curlabel)
1.  Set visited(v) := true
2.  Set label(v) := curlabel
3.  For each edge (v,u) ∈ E:
4.      If not visited(u):
5.          AssignLabels(u, 1 − curlabel)
6.      else
7.          If label(v) = label(u):
8.              Return FAILURE
```

Proof of correctness: Here is a useful invariant: if G' is the subgraph of G containing the individuals given a label so far, and the algorithm has not yet reported failure, then the label() array contains a satisfactory assignment of citizens to destinations. In other words, no two citizens in G' who hate each other will be assigned to the same location. Further, the solution for each connected component is unique (except that we can swap island and mainland for every citizen in the connected component, to obtain one other solution; so if there is any valid solution at all, there will be exactly two complementary solutions in each connected component).

We will prove this by induction on the number of individuals who have been assigned some label so far.

Base case: If only a single individual is assigned a destination, any assignment is satisfactory; one individual can't have any grudges, and we can assign him to the mainland without any trouble. We can only send him to one or two destinations, and these are equivalent under exchanging mainland and island. Therefore, the solution is unique up to complementation.

Inductive case: Assume that the label() array contains a mapping from individual to labels such that no two individuals connected with an edge have the same label, and that v is the next vertex visited by the algorithm. We must have reached v in one of two ways:

(1) by re-starting exploration at v after exhausting the previously-explored connected components of the graph,

(2) via an edge from some adjacent and already-visited vertex.

In case (1), we can assign an arbitrary destination to v (line 5 of SplitPeople()), and we are guaranteed that the assignment is unproblematic: if there are no edges between different graph components, then the decisions made for the vertices in one connected component cannot affect the other.

In case (2), because at least one neighbor has been given a label (0 or 1) – since that neighbor is already visited – there are two scenarios:

(a) If all the neighbors of v have the same label and that label is different than the one given to v, the solution is still unique (since there are edges among people with different labels), up to swapping everybody's label.

(b) If at least one of the neighbors of v (let's call this neighbor w) has been given the same label as v then no possible assignment can succeed and the algorithm reports failure (line 8 of AssignLabels()). We will prove that our algorithm correctly reports failure. Let's assume that there is a label of v such that the graph $G' \cup \{v\}$ can have a valid assignment. However, our algorithm assigns to the vertices of a tree edge alternating labels (note that if curlabel $= 0$ then $1 -$ curlabel $= 1$ and similarly if curlabel $= 1$ then $1 -$ curlabel $= 0$). Therefore the only way that v is assigned the same color with w is due to the fact that there is a back edge (v,w), i.e., there is a cycle that contains v and w. Let's examine that cycle:

$$w_1 - w_2 - \cdots - w_k - v - w - w_1$$

Can k be even number? Well, it means that there should be an even number of edges in the previous cycle. Then our algorithm should have followed an odd number of tree edges from w to v and since on tree edges we alternate labels, it means that v and w should have been given different labels (which is not the case here). Therefore k should be an odd number. But in this case, if we try to alternate labels starting from vertex w and following the previous path we find that v should be assigned the same label with w, which means that there is no valid assignment of $G' \cup \{v\}$ and this is a contradiction. Therefore our algorithm will report correctly failure if we try to visit the vertex v.

At the end of the algorithm, all individuals will have been assigned a label and $G' = G$. By the invariant above, the algorithm will have either returned failure, or else found a satisfactory assignment.

Running time: At each vertex, the algorithm examines every neighbor; this means that the work at each vertex is proportional to the number of neighbors. The work on the graph as a whole is therefore proportional to the number of nodes plus the number of edges, which is just $|V| + |E|$. So the algorithm's running time is $\Theta(|V| + |E|)$.

Notice that our input is an edge-list, not an adjacency list. Fortunately, we can do the conversion in $\Theta(|E|)$ time. Build a hash table of nodes, and have each node store its adjacency list. We can populate the table in one pass over the edges, creating nodes as needed.

Common mistake: Many people proved that if their algorithm outputs "Yes", then this is the correct answer (i.e., if their algorithm finds a partitioning, then this is a valid partitioning). This was the easy part to prove. However, a very common mistake was to forget to prove that if your algorithm outputs "No", then "No" is the correct answer: in other words, if the algorithm does not find a valid partitioning, then there is no way to partition the people. The latter is much trickier to prove: we have to prove that *every* single solution is invalid, not that your *current* solution is invalid.

Revised 10/8 to add this explanation of the common mistake.

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.

Solution:
Main idea: We can traverse the directed graph G (using a slightly modified DFS) where the vertices of G are the states and the directed edges of G are defined by the function `update`. In particular, given a state s we can examine all possible values of the sensor input i (there are 16 possible inputs) and we can determine the neighbors (states) to be further explored by calling `update(s,i)`. When we find a neighbor, we can check the output o and verify if this is a valid output by calling `doublegreen(o)`.

Pseudocode

SafeSystem(s_0)
1. Set globally_safe := true
2. ExploreSystem(s_0)
3. Return globally_safe

ExploreSystem(s)
1. Set visited(s) := true
2. Set list_of_neighbors := []
3. For $i := 0, 1, \ldots, 15$:
4. (t, o) := update(s, i)
5. If doublegreen(o) = true:
6. Set globally_safe := false
7. Add t to list_of_neighbors
8. For each $t \in$ list_of_neighbors:
9. If not visited(t):
10. ExploreSystem(t)

Additional explanation: The idea is that the global variable globally_safe will be set to false if there is a call to update(s, i) that returns a pair (t,o) such that doublegreen(o) = true.

Proof of correctness. The following invariant will hold after constructing the list of neighbors of a state s (just before line 8): If there is an input i' such that (t, o) = update(s, i') and doublegreen(o) returns true then the variable globally_safe has been set to false after constructing the list of neighbors of s.

This can be proved by contradiction. Let's assume that there is an input i' such that (t, o) = update(s, i'), doublegreen(o) returns true and the variable globally_safe has **not** been set to false after constructing the list of neighbors. However, the for-loop at line 3 explores all possible inputs i, so at some iteration we will have $i = i'$. Then, in that iteration we will call (t, o) = update(s, i') (line 4), we will find that doublegreen(o) returns *true* (line 5) and therefore we will set the variable globally_safe to false (line 6). Since there is no way to set the variable globally_safe back to true, after all the iterations of the for-loop in line 3 (i.e., after constructing the list of neighbors of s) the variable globally_safe is false, which is a contradiction.

The previous invariant combined with the fact that the variable globally_safe can't be set to true after calling ExploreSystem(s_0) implies that if we see an unsafe state the variable globally_safe will eventually be false. The only thing that remains to be shown is that our algorithm visits all possible states that are reachable from the original state s_0. This follows from the fact that we are using depth-first search, which is proven to visit all reachable states in the graph. Or, we could prove it directly by contradiction, using the same argument found in the book used to prove depth-first search correct. Let's assume that there is a state t that is reachable by the original state s_0 and our algorithm does not visit that state. Choose a path from s_0 to t and look at the last state on that path that our algorithm actually visited. Call this state z and let w be the state immediately after it on the same path:

$$s_0 \to \cdots \to z \to w \to \cdots \to t$$

So state z was visited but w was not. This is a contradiction: Since there is an edge $z \to w$, there is an input i' such that (w, o') = update(z, i') and thus our algorithm adds w to the list_of_neighbors of z. Therefore w will be visited due to the loop at line 8 where we will call ExploreSystem() on all the states in the list_of_neighbors of z.

Running time. We visit every state (vertex in the directed graph G) exactly one time because we call ExploreSystem() only if a state is unvisited and once we call ExploreSystem(s) we set visited(s) to true. Also, in the procedure ExploreSystem() we generate the list_of_neighbors in $O(1)$ time since there are 16 iterations of the loop at line 3 and the calls update(s,i) and doublegreen(o) take constant time. Also, our algorithm examines each edge $s \to t$ of G exactly once (when we call ExploreSystem(s) and we iterate in loop at line 8). Since each state has at most 16 neighbors there are $O(16n) = O(n)$ edges in G. Therefore the total runtime is $O(n)$.

Alternatively, we could note that this is just depth-first search, so it runs in $O(|V| + |E|)$ time. Here the number of vertices is the number of states, n, and the number of edges is $16n$, so the total running time is $O(n + 17n) = O(n)$.

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?

Solution: $f(r)$ = the number of children of r.

Proof: The depth-first search will only return to the root node if it has completely explored the subtree rooted at each node. Therefore, the subtrees rooted at each child are unconnected, and removing r will disconnect each of them from the others.

Alternative proof: By Problem 3, there are no cross-edges in the DFS tree. Therefore, the subtrees rooted at each child of r are not connected to each other in any way, so removing r will disconnect each of them from the others.

(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?

Solution: $f(v) = 1+$ the number of children of v.

Proof: This case differs from the previous one only in that there's another component, corresponding to the ancestors of v. The lack of non-tree edges from descendants to ancestors means that removing v disconnects every child of v from every ancestor of v. And as before, the children are all partitioned from eachother by removing v.

(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.)

Solution: We can do this by traversing the tree [not the original graph], either breadth- or depth-first. At each node v, set $d(v)$ to the depth of the node's parent, plus one. The key is to use a traversal order such that each node is visited (to calculate its depth) before any of its children are visited, so that we can calculate each child's depth as a function of its parent's depth.

(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.

Solution: Let N denote the number of children c of v with the property that $up(c) \geq d(v)$, i.e., $N = |\{c : c \text{ is a child of } v \text{ and } up(c) \geq d(v)\}|$. Then $f(v) = N + 1$.

This case differs from that in part (b) because it might be that a child of v is connected to an ancestor of v by some route, not through v. This is equivalent to saying that v, some child, and some ancestor are connected by a cycle. We can use $up(\cdot)$ to detect this case.

Claim 1 *The set of nodes visited before node v, which includes all the tree ancestors of v, form a single connected component, which will not be split by removing v.*

Proof: The instant before we added v to the tree, the DFS tree connected all these nodes, and didn't include v. Removing v from the tree won't change that. □

Claim 2 *If c is a child of v in the DFS tree, and $up(c) < d(v)$, then removing v from the graph will not disconnect c from the tree component above v.*

Proof: By the definition of *up*, there is a chain of edges from c, possibly through descendants of c, to a node y at some depth less than v. By the definition of depth, all nodes at depth less than $d(v)$ are connected by a path through the root that does not include v Thus, removing v will not disconnect c or its children from that component. □

Claim 3 *If removing parent v will not disconnect child c from the component above v, then $up(c) < d(v)$.*

Proof: Since removing v does not disconnect c, there must be some path in the original graph, not containing v, that starts at c and ends at an already-visited node in the component "above" v. Call the end node in the path y. All nodes in the path except c and y will be descendants of c, since any node reachable from c that hasn't already been visited will be a descendant of c in the DFS tree.

So at the time that c was first visited, y had been visited, but had unexplored edges. Since DFS processes nodes in last-in-first-out (stack) order, this means y is an ancestor of c, and thus of v. Therefore, $d(y) < d(v)$. By definition of $up(\cdot)$, this means that $up(c) < d(v)$, since there is a chain of descendants of c leading to y with depth less than v. This proves the claim.

Another way to see it: We proved in Q3 that there are no cross edges in a DFS tree. All non-tree edges connect ancestors and descendants. If removing v doesn't disconnect a child, then there's an edge to a non-descendant, and therefore that edge must go to an ancestor. Ancestors have depth less than $d(v)$ and therefore $up(c) < d(v)$. □

Claims 2 and 3 establish that removing v disconnects each child from the component above v iff $up(c) < d(v)$. There can't be paths from one child to another, not passing through v or an ancestor of v; otherwise the DFS would have taken that path, and not returned to v. Let N be the number of children c of v with the property that $up(c) \geq d(v)$. Removing v forms $N+1$ components: one for each child with the appropriate up, and one for the component above v.

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

Solution:

Main idea. By definition, $up(v)$ is the minimum of v's neighbors' depths, and the ups of v's descendants. Formally,

$$up(v) = \min\big(\min\{d(w) : \{v,w\} \in E\},\ \min\{up(w) : w \text{ is a child of } v\}\big).$$

We can thus compute $up(v)$ by traversing the DFS tree bottom-up.

Pseudocode.

ComputeUp(v):
1. For each child c of v in the DFS tree:
2. Call ComputeUp(c).
3. Set $up(v) := \min(\min\{d(w) : \{v,w\} \in E\},\ \min\{up(w) : w \text{ is a child of } v\})$.

Calling ComputeUp(r), where r is the root of the DFS tree, computes $up(v)$ for each vertex $v \in V$.

Proof of correctness. We will prove the correctness of our algorithm by using induction on the height of the vertex v (the difference between the depth of v and the depth of its deepest descendant).

Base case: The base case is when we call ComputeUp(v) with of a single vertex v that has no children. In this case, the procedure ComputeUp(v) sets $up(v)$ to $\min\{d(w) : \{v,w\} \in E\}$. This is the correct value since v has no descendants thus it should be the depth of the shallowest node w such that there is some graph edge $\{v,w\} \in E$ (see the definition of part (d)).

Inductive step: Assume that our procedure ComputeUp() sets the correct value to $up(w_i)$ if it is called with input the subtree rooted at w_i ($i = 1,2,\ldots,k$) where $w_1,\ldots,w_k$ are the children of v. (This is guaranteed by the inductive hypothesis, since $w_1,\ldots,w_k$ each have smaller height than v.) Thus, when we call ComputeUp(v), the procedure sets the correct values to $up(w_1),\ldots,up(w_k)$ by calling ComputeUp(w_i) for each child w_i of v (loop at lines 1-2). It finally sets

$$up(v) := \min(\min\{d(w) : \{v,w\} \in E\},\ \min\{up(w) : w \text{ is a child of } v\}),$$

which is the correct value of $up(v)$ since by definition, $up(v)$ is the minimum of v's neighbors' depths, and the ups of v's descendants. Therefore ComputeUp(v) sets the correct value to $up(v)$ which completes the proof.

Running time. $O(|V|+|E|)$. This follows because each vertex is processed once, and edges are only processed twice, once for each end of the edge.

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

Solution: This follows immediately from parts (c)–(e). Pick some node as the root. Then compute $up(\cdot)$ and $d(\cdot)$ at each node. Then, compute the function defined in part (d).

The running time is $\Theta(|V|+|E|)$. We needed to make three passes over the graph, one to compute $d(\cdot)$, one to compute $up(\cdot)$, and a third to compute $f(\cdot)$. In each pass, we process each vertex once, and each edge at each vertex. This is $\Theta(|V|+|E|)$ for each pass, and $\Theta(3|V|+3|E|) = \Theta(|V|+|E|)$.

Note that in a practical implementation, some of these separate passes could be combined, without affecting the asymptotic complexity of the algorithm.

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.

Solution:

Main idea: First we will generate the directed graph G where the vertices of G are the states and the directed edges of G are defined by the function `update`. In particular, given a state s we can examine all possible values of the sensor input i (there are 16 possible inputs) and we can determine the neighbors (states) of s. Note that a car is waiting infinitely long in a direction d_i (there are 4 possible directions) if there is a cycle in G where *all* the nodes of that cycle (states) are characterized by red light in direction d_i. Otherwise there is a state with a green light in the direction d_i and a car can't wait infinitely in that direction d_i. Now consider a subgraph G_i that is induced by removing from G all the vertices (states) that have green light in direction d_i. We can find if G_i contains a cycle by using DFS. If G_i doesn't contain any cycles we conclude that a car can't wait infinitely on that direction d_i otherwise there is such a possibility. So, we can run 4 DFS traversals on the induced subgraphs G_1, G_2, G_3 and G_4 in $4\Theta(n) = \Theta(n)$ time and we can determine if there is any possible scenario where our code could leave a car waiting infinitely long for the next green light — each induced subgraph G_i has $O(n)$ vertices and $O(n)$ edges since each node in the directed graph G has 16 outgoing edges, there are at most $16n = \Theta(n)$ edges in G.

Pseudocode

GenerateGraph(s_0)
1. G is an empty adjacency list
2. ExploreSystem(s_0,G)
3. Return G

```
ExploreSystem(s,G)
1.  visited(s) := true
2.  list_of_neighbors := []
3.  For i := 0,1,...,15:
4.      (t,o) := update(s,i)
5.      Based on t and o set appropriately the values to the truth table isRed(t,dir_j) j = 1,2,3,4
6.      Add t to list_of_neighbors
7.  Add list_of_neighbors as the adjacency list of s in G
8.  For each t ∈ list_of_neighbors:
9.      If not visited(t):
10.         ExploreSystem(t,G)
```

```
GenerateInducedGraph(G',dir)
1.  G:=G'
2.  For each vertex v ∈ G:
3.      if (isRed(v,dir) == false):
4.          remove v from G by removing its adjacency list
5.  For each edge (u,v) ∈ G:
6.      if (isRed(v,dir) == false):
7.          remove edge (u,v) from G
8.  Return G
```

```
HasInfiniteRedLoops(s_0)
1.  G = GenerateGraph(s_0)
2.  For i := 1,2,3,4:
3.      G_i := GenerateInducedGraph(G,dir_i):
4.      if (hasCycle(G_i) == true):
5.          Return true
6.  Return false
```

Proof of correctness. The procedures GenerateGraph() and ExploreSystem() correctly generate an adjacency list of G since essentially they perform a DFS in G and in line 7 of procedure ExploreSystem() we add in G the adjacency list of the vertex we visit and then we recursively explore this adjacency list. Also, the procedure GenerateInducedGraph() generates correctly an induced graph G_i since it removes from G all the vertices (states) that have green light in direction d_i and all of their incoming edges.

The only thing that remains to be proved (and also it implies the correctness of the procedure HasInfiniteLoops()) is that a car can wait infinitely in a direction dir_i if and only if there is a cycle in the induced subgraph G_i.

Let's prove first the direction: a car can wait infinitely in a direction $dir_i \Rightarrow$ there is a cycle in the induced subgraph G_i. Well, if a car can wait infinitely in a direction dir_i then it means that there is a cycle in G where *all* of its vertices (or equivalently states) have red light in that direction dir_i. However, by construction of the inducted graph G_i we don't remove any vertices from the original graph G that have red light in that direction dir_i neither we remove any edges of G that go to such vertices. Thus, the same cycle of G should be also found in G_i.

The other direction: there is a cycle in the induced subgraph $G_i \Rightarrow$ a car can wait infinitely in a direction dir_i. This follows from the fact that the vertices and the edges of the induced subgraph G_i can be also

found in the graph G. Therefore, G contains the same cycle as the one found in G_i where *all* of its vertices (or equivalently states) have red light in that direction dir_i and consequently a car can wait infinitely in a direction dir_i.

Running time. Note that each induced subgraph G_i has $O(n)$ vertices and $O(n)$ edges since each node in the directed graph G has 16 outgoing edges, therefore there are at most $16n = \Theta(n)$ edges in G. The cost of generating the graph G by calling GenerateGraph() is $\Theta(n)$ since it costs as much as a DFS on a graph with $\Theta(n)$ vertices and $\Theta(n)$ edges. The cost to generate an induced graph G_i by calling GenerateInducedGraph() is again $\Theta(n)$ since we just visit each vertex and each edge of G exactly one time. Then, the cost of finding if a directed graph with $O(n)$ vertices and $O(n)$ edges has a cycle is $O(n)$ since we just run DFS and report true if we find a back edge. Finally the total running time is $O(n) + 4O(n) + 4O(n) = O(n)$.