

CS303 (Spring 2008)— Solutions to Assignment 4

Exercise 22.2-3

The new running time will be $\Theta(n^2)$. The inner loop (when a node is explored) will take at most $O(n)$ time, as one can never do worse than having to check the entire adjacency matrix row corresponding to node v for edges, which takes time $O(n)$. On the other hand, one always has to go through the entire row (even if very few edges are out of v), as otherwise, the algorithm cannot be sure that it is not missing any edges. Thus, the inner loop also takes $\Omega(n)$. Since this applies for each node v , the running time overall is $\Theta(n^2)$.

Exercise 22.2-6

This “good guys”/“bad guys” terminology really hides a standard graph problem. Is it possible to partition the nodes of a graph into two sets, S and its complement \bar{S} , such that all edges are between S and \bar{S} , and no edges are inside either S or \bar{S} ? A graph with this property is called *bipartite*, so the problem asks you to come up with an algorithm to test if a given graph G (the “wrestler rivalry graph”) is bipartite, and if so, produce the corresponding partition. Bipartite graphs arise naturally in many settings, for instance as possible assignments of jobs to machines, projects to consultants, men to women, students to classes, etc. In most of these cases, it makes no sense to think of assigning a job to a job, or a machine to a machine, hence edges only exist across the two partitions.

Now for the algorithm: the first thing we observe is that if there are multiple connected components in G , then we can do the computation for each separately; any way of combining them will be fine. So we may as well assume that there is only one component. The algorithm is quite simple: start from an arbitrary node s , and compute the distances $d[v]$ from s for each node v , using BFS. Everyone with an even value $d[v]$ (including s) is declared a “good guy”, and everyone with an odd value of $d[v]$ is declared a “bad guy”. If there is ever an edge/rivalry between two guys declared good, or two guys declared bad, then the algorithm answers that the task is impossible. Otherwise, it outputs the classification.

It is clear that when the algorithm does output a partition, it is a legal one, because the algorithm explicitly checks that it is legal before outputting it. However, we still need to make sure that it does not conclude incorrectly that a division is impossible. In other words, maybe the labeling the algorithm tried was just a bad choice, but a partition was still possible. We prove that this cannot happen. For assume that in the assignment the algorithm found, there is an edge from a good guy u to another good guy v (or, both u and v are bad guys). Then, because both $d[u]$ and $d[v]$ are even (or both are odd), there is a path of even length from u to v (the one through s). Combined with the edge from u to v that caused the algorithm to claim the partition was impossible, this forms a cycle of odd length ($k \in \{3, 5, \dots\}$). Let v_1, v_2, \dots, v_k be the nodes on this cycle. Then, if v_1 is good, v_2 would have to be bad, v_3 would have to be good, etc. All v_i with odd i would have to be good, and all v_i with even i would have to be bad. In particular, v_k would be good, but then the edge between v_1 and v_k forms a problem. On the other hand, if v_1 is bad, we get the same problem, with “good” and “bad” reversed. Either way, it is impossible to divide these nodes into two sets, so the algorithm was correct in claiming the task was impossible. So we just proved the following theorem:

Theorem 1 *A graph G is bipartite if and only if it contains no cycles of odd length.*

The running time of the algorithm is first the $\Theta(n+r)$ for the BFS subroutine. After that, the algorithm has to go through each edge, and test (in constant time) if the edge causes a problem, i.e., is between two good nodes or two bad nodes. This test takes time $O(1)$, so the whole loop takes time $O(r)$, and the running time is $\Theta(n+r)$.

Exercise 22.3-6

We can actually just take the BFS procedure, and replace the queue by a stack. Here, following the presentation of the book, we will also compute the discovery and finish times $d[]$ and $f[]$ (though that is not necessary for your solution).

Algorithm 1 DFS without recursion

```
1: Mark all nodes unenqueued, unexplored.
2: Set the time  $t = 0$ 
3: Put  $s$  on the stack.
4: while the stack is not empty do
5:   Pop the top node  $v$  off the stack.
6:   Increase  $t$ , and set  $d[v] = t$ .
7:   for each neighbor  $u$  of  $v$  do
8:     if  $u$  is unenqueued and unexplored then
9:       Set  $\text{pred}[u] = v$ , and put  $u$  on top of the stack.
10:      Mark  $u$  enqueued.
11:     end if
12:   end for
13:   Increase  $t$ , and set  $f[v] = t$ .
14:   Mark  $v$  explored.
15: end while
```

Exercise 22.4-2

The key observation to this problem is the following: Suppose v is any node, and it has incoming edges from k nodes u_1, u_2, \dots, u_k . Suppose also that we already know that u_1, u_2, \dots, u_k have p_1, p_2, \dots, p_k paths going to them from s . Then, any of those paths can be combined with the last hop to v to form a distinct path from s to v . Thus, v has exactly $p_1 + p_2 + \dots + p_k$ paths from s into it.

The only problem is that we need to be sure of the number of incoming paths into each of the u_i before we finalize the number for v . In other words, we need to process the nodes in an order such that when we get to a node v , all of its predecessors have already been processed. That is where topological sort comes in, since it gives us just such an ordering.

Thus, our algorithm is as follows: First, compute a topological sort of all nodes. This numbers the nodes v_1, v_2, \dots, v_n such that all edges into v_i (if any) are from nodes v_j with $j < i$. Now, process the nodes in this ordering. When processing a node v_i , distinguish two cases: (1) If $v_i = s$, then set $p[v_i] = 1$. (2) If $v_i \neq s$, then set $p[v_i] = \sum_{v_j: (v_j, v_i) \in E} p[v_j]$, where the sum is of course 0 if there are no such v_j . (This notation means that we sum up the $p[v_j]$ values for all nodes v_j such that there is an edge (v_j, v_i) in the graph.) Finally, output $p[t]$.

To see that this is correct, we can use induction on i to prove that all $p[v_j]$ for $j = 1, \dots, i$ are correct. The base case is $i = 0$, and there is nothing to prove. For the induction step, we notice that because of the topological sort, when we process v_i , all of its predecessors have already been processed. By the induction hypothesis, their values $p[v_j]$ are correct, and by the first paragraph, we thus compute the right $p[v_i]$. If $v_i = s$, then the value is correct because there is exactly one path from s to itself.

The running time of topological sort is $\Theta(n + m)$. The loop runs over all nodes, and for each node does constant work for each incoming edge. Thus, the total work for the loop is also $O(n + m)$. So the running time is $\Theta(n + m) + O(n + m) = \Theta(n + m)$.