Homework 6 Solutions

1. 22.2-6 (pg. 539)

   There are two types of professional wrestlers: good guys, and bad guys. Between any pair of professional wrestlers, there may or may not be a rivalry. Suppose that we have n professional wrestlers and we have a list of r pairs of wrestlers for which there are rivalries. Give an O(n + r)-time algorithm that determines whether it is possible to designate some of the wrestlers as good guys and the remainder as bad guys such that each rivalry is between a good guy and a bad guy. If it is possible to perform such a designation your algorithm should produce it.

   Assumptions:
   1. A rivalry (A, B) represents the fact that A has a rivalry with B. It does not specify whether or not A or B is the good or bad guy.
   2. Some wrestlers may not be involved in rivalries. These wrestlers are neither good guys nor bad guys. The algorithm is not concerned with them.
   3. Any possible solution of good guys and bad guys is acceptable. For example, consider the following setup:

   Wrestlers = {A, B, C, D, E}, Rivalries = { (A, B), (B, C), (D, E) }

   There are four possible solutions for this set up:

   1. Good guys = {A, C, D}, Bad guys = {B, E}
   2. Good guys = {A, C, E}, Bad guys = (B, D}
   3. Good guys = {B, E}, Bad guys = {A, C, D}
   4. Good guys = {B, D}, Bad guys = {A, C, E}

   An correct algorithm could return any one of these solutions.

   Setup: We represent the rivalries as a graph, G = (V, E). The vertices are the wrestlers, and the edges are the rivalries. Therefore, |V| = n, and |E| = r.

   Algorithm:
   1. Discard all vertices of degree 0. These are wrestlers who have no rivalries. We are not concerned with them.
   2. Separate all connected components of the remaining graph. Process each component individually in the following steps.
   3. Let C = (V_c, E_c) be the connected component. Choose one vertex r at random (or deterministically-it doesn't matter) from V_c. Without loss of generality (remember, any possible solution, is correct), let r be classified as a Good guy.
   4. Run a modified breadth-first search from the root. Instead of maintaining d[] and f[] arrays, save the path-length from each vertex to r. All vertices with odd path lengths to r are Bad guys; all vertices with even path lengths to r are Good guys.

5. Examine every edge in E_c. If the edge is between two vertices whose path lengths to r are both even or both odd, then we have established a rivalry between two Good or Bad guys. If this is the case, we return false – it is not possible to partition the wrestlers.
6. If we are able to examine every edge in every connected component without returning false, then we have successfully partitioned the wrestlers in step 4.

Complexity: The major time-related component of this algorithm is the breadth-first search, which has O(V + E) time.

2. 22.2-8 (pg. 539)

Let G = (V, E) be a connected, undirected graph. Give an O(V + E)-time algorithm to compute a path in G that traverses each edge in E exactly once in each direction. Describe how you can find your way out of a maze if you are given a large supply of pennies.

To solve this problem, we use a variant of depth-first search. Every edge is marked the first and second time it is traversed with unique marks for each traversal. Edges that have been traversed twice may not be taken again.

Our depth-first search algorithm must ensure that all edges all explored, and that each edge is taken in both directions. To ensure that all edges are explored, the algorithm must ensure that unexplored edges are always taken before edges that are explored once. To ensure that edges are taken in each direction, we simply backtrack every time the depth-first search reaches a dead-end. The search keeps backtracking until a new unexplored edge is found. This way, edges are only explored in the reverse direction during the backtracking.

Complexity: This algorithm is based on depth-first search, which has time complexity O(V+ E).

3. 2.3-4 (pg. 548)

a.) Show that edge (u, v) is a tree edge or forward edge if and only if $d[u] < d[v] < f[v] < f[u]$

Proof:

→: Assume that edge (u, v) is a tree or forward edge. If (u, v) is a tree edge, then by the definition of tree edge v is first discovered by exploring edge (u, v). If (u, v) is a forward edge, then by the definition of forward edge v is an ancestor of u. In either case, $d[u] < d[v]$. Since v was discovered after u, then must be finished prior to u being finished, hence $f[v] < f[u]$. Every vertex must be discovered before it can be finished, so $d[v] < f[v]$. Putting all three inequalities together yields $d[u] < d[v] < f[v] < f[u]$.

←: Assume that for vertices u and v, d[u] < d[v] < f[v] < f[u]. d[v] < f[v] can be established trivially. d[u] < d[v] and f[v] < f[u] imply that v was discovered after u, and finished before u. Therefore, edge (u, v) is a tree edge if v was discovered by traversing edge (u, v). Otherwise, (u, v) is a forward edge.

b.) Show that edge (u, v) is a back edge if and only if d[v] < d[u] < f[u] < f[v]

Proof:

→: Assume that edge (u, v) is a back edge. Then by definition, v is an ancestor of u, so d[v] < d[u]. Since v was discovered before u, u will finish before v finishes. Hence, f[u] < f[v]. d[u] < f[u] can be established trivially. Therefore d[v] < d[u] < f[u] < f[v].

←: Assume d[v] < d[u] < f[u] < f[v]. f[u] < f[v] and d[v] < d[u] imply that v was discovered before u and finished after u. Therefore v is an ancestor of u. Therefore (u, v) is a back edge.

c.) Show that edge (u, v) is a cross edge if and only if d[v] < f[v] < d[u] < f[u]

Proof:

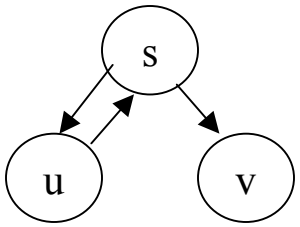
→: Assume that edge (u, v) is a cross edge. Therefore, there is no parental or ancestral relationship between u and v. d[u] < f[u] and d[v] < f[v] can be established trivially. If d[u] < d[v], then edge (u, v) would indicate a parental relationship between u and v in the depth-first tree, which cannot happen by the definition of cross edge. Hence, d[u] > d[v]. Similarly, we cannot have d[u] < f[v], because this would indicate that v was finished (but not discovered) after u was discovered, but before u was finished, which cannot happen. Therefore, d[v] < f[v] < d[u] < f[u]

←: Assume d[v] < f[v] < d[u] < f[u]. Since f[v] < d[u]], there is no parental or ancestral relationship between u and v. Therefore edge (u, v) is a cross edge.

4. 22.3-8 (pg. 548)

   Give a counterexample to the conjecture that if there is a path from u to v in a directed graph G, then any depth-first search must result in d[v] ≤ f[u].

	d	f
s	1	6
u	4	5
v	2	3



   We perform a DFS starting at vertex s. We then discover vertex u. Since the only edge out of u is (u, s), and s has been found, we finish u. Next, we discover and finish v. Finally, we finish s.

5. 22.4-2 (pg. 552)

   Give a linear-time algorithm that takes as input a directed acyclic graph G = (V, E) and two vertices s and t, and returns the number of paths from s to t in G.

   The basic idea here is to start at vertex t, and use depth-first search in reverse direction until we reach vertex s. Each and maintains a counter which indicates the number of unique reverse paths found from vertex t.

   1. Initialize counters to 0 for all vertices.
   2. Start depth-first-search in reverse direction using vertex t as a root.
   3. For each edge (u, v) examined in the breadth-first search.
      Counter(v) = max{ Counter(v) + 1, Counter(v) + Counter(u) }
   4. Return Counter(s)