CS170–Fall 2015 — Homework 4 Solutions

Nguyet Minh Duong, SID 24444722

Collaborators: Henry Kwan & James Carlson

1. Depth-First Search Warm-up

(a) - A B D E G F C H I

- Labeling pre and post values:

Node	Pre	Post
A	1	12
B	2	11
C	13	18
D	3	6
E	4	5
F	8	9
G	7	10
H	14	17
I	15	16

- Labeling the graph:
  A – B: Tree
  B – D: Tree
  B – G: Forward
  G – D: Cross
  G – F: Tree
  D – E: Tree
  E – D: Backward
  A – F: Forward

  C – H: Tree
  C – I: Forward
  H – I: Tree

(b) There are 8 strongly connected components in this graph.

2. Directed Paths

1. Main Idea
   Pick a node with no incoming edges. Run depth-first search on the tree, starting at the node we picked. Record the order in which the node is post-visited. Run through the nodes from the biggest post visit to the smallest post visit, if possible. Post visit value of A is greater than post visit value of B. And there must be a path from A to B. If there is no such path, the graph does not contain a directed path that goes through all the nodes.

2. Pseudocode

```
Procedure ContainsPath(graph G):
  V := list of all the vertices
  E := list of all the edges
  s := iterate through V, find v such that indegrees(v) = 0,
       return the first v that satisfy this condition
  post-visits[] := run depth-first search on graph starting at vertex s
       record the vertex in order as they are post-visited
  // the first vertex that is post visited, is the first element in post-visits[]
  while post-vists[] contains elements:
    F := last element (a vertex) from post-visits[]
    S := second to last element (a vertex) from post-visits[]
    if edge (F, S) exists:
      remove F from post-visits[]
    else:
      return "There isn't a path"
  return "There is a path"
```

3. Proof of Correctness
   Proof by Contradiction: Our algorithm does not work.

   Case 1: Our algorithm gives us a line when there is no line.
   This is not possible by how we defined our path to be. We decided that the algorithm only outputs "There is a path" when there is a forward edge from the vertex with the bigger post-visit value to the vertex with the lower post-visit value. This means there there must be a path or else the algorithm would break. Therefore contradiction.

   Case 2: Our algoirthm does not give us a line when there exists a line.
   Our algorithm follows the outgoing edge. This means that the vertex can have an outgoing edge to either a vertex with a smaller post-visit value or a bigger post-visit value. However if the algorithm does not return a line, it means one of the two conditions: it does not have an outgoing edge or if the outgoing edge has a post-visit value bigger than it. If the vertex we are on has a lower post-visit value than the one its directed edge is pointing to, it creates a cycle. However this contradicts our original given, which is that the graph given to us is a acyclic.

4. Running Time Analysis
   $O(|V| + |E|)$
   Our first search to find the vertex such that there is no in-degrees. In the worst case scenerio,

we search through all of the vertices. This gives us the run time of $O(|V|)$ for this case. In order to record all the post-visits, we use depth-first search, which has the run time of $O(|V| + |E|)$. We know that the length post-visit list is the same amount as $|V|$ because all vertices in the graph must be post-visited at some point. So in worst case scenerio, we look through all the items in the list, which has the size of $|V|$, that run time is $O(|V|)$.

To find the run time, we add all of the run time together, which results in $O(3|V| + |E|) \equiv O(|V| + |E|)$. This is linear time.

3. A Game of Choosing Edges in a DAG

1. Main Idea
   We take in consideration of the two turns, player A and player B. In our algorithm, we need to make sure that it returns the values accordingly depending on which turn the vertex is on. This means that if it is a winning value (true) on a vertex of the Player B, we need to return false. We need to do the same thing for the winning value as well in A's case.

2. Pseudocode

```
Procedure AbsWinner(node s, graph G):
  values := a dictionary that has node as its key, win condition as its value
  return FindCondition(s, G)

  Procedure FindCondition(node A, graph G):
    if node A is in dictionary values:
      return values[A]
    else if outDegree(A) is 0:
      values[A] = false
      return false
    else:
      for all node N, where A has an outgoing edge to N:
        condition := FindCondition(N, G)
        values[N] = condition
        if condition is false:
          values[A] = true
          return true
      values[A] = false
      return false
```

3. Proof of Correctness
   Direct Proof: We are literally recording the winning condition of each stage. We know that in the very beginning state, if the node Player A (P_A) is on, P_A loses. In the losing case, we will call it to be false. However if it has outgoing edge, it is now Player B's (P_B) turn. In this case, if the node P_B is on has no outgoing edge, he will lose – return "false". However, this is not false to P_A, in fact this is a winning case for P_A. Now if any of P_A's children in the graph returns "false", it is in reality a "true" for P_A because now there is a child P_A can go to that is pessimal for P_B.

   The algorithm continues like this recursively until it hits a node without any children and automatically return false, as that is the pessimal case we do not want to be in. Then it recurse back up, turning into "true" whenever a pessimal route for the opponent is found. At the end, the result on P_A's starting node will give either true or false. True stating that there is a pessimal route for P_B such that they cannot win no matter the condition, and false if no matter how P_A plays, P_B can always change up the game to his winning conditions.

4. Running Time Analysis
   $O(|V| + |E|)$

We go down the tree through all the possible values. While this should be a polynomial running time, it isn't because we end up recording the values. This means that we travel through all of the edges and vertices once to record all the values. This is $O(|V| + |E|)$ time. Note that we have to record all the vertices and its winning values. Because recording into the dictionary takes $O(1)$ time, and we do it for all the vertices, this running time becomes $O(|V|)$. Now when we add all of this up, and recurse back up, it is a total of $O(|V| + |E| + |V|) \rightarrow O(|V| + |E|)$ since we ignore constants.

4. Counting Paths

(a) **Proof by Induction** on K, the length of the path from vertex s to vertex k

Base Case: $K = 1$, this is by the given definition, it tells us the how many possible paths there are with the length 1 from the and row position s to any column position k. If there are any paths, it should be one because this is an adjacency matrix so it lists how which vertex can go to which. And it does so in length of one, and there is only one possible path to do that in this condition.
Induction Hypothesis: Assume for $N \leq K$, the value at $(A^N)_{s,t}$ is the number of paths from vertex s to vertex t with N steps.
Induction Step: For $N = K+1$, our matrix is $A^N = A^{K+1} = A^K * A^1$. From our hypothesis, we know that A^K will create a matrix such that the values in row s and column t tells us how many different paths we can take with k-steps. Now if we multiply it against A, it will figure out the new values such that the it is how many different paths there are that has N-length. We can prove this by solving just one slot, such as $s = 1, t = 1$:

$$\begin{bmatrix} v_{1,1} & v_{1,3} & v_{1,2} \\ v_{2,1} & v_{2,2} & v_{2,3} \\ v_{3,1} & v_{3,2} & v_{3,3} \end{bmatrix} \begin{bmatrix} u_{1,1} & u_{1,2} & u_{1,3} \\ u_{2,1} & u_{2,2} & u_{2,3} \\ u_{3,1} & u_{3,2} & u_{3,3} \end{bmatrix}$$

If we want to calculate the new value of $(1, 1)$, we would do the following to calculate it: $(A^N)_{1,1} = (v_{1,1}u_{1,1} + v_{2,1}u_{1,2} + v_{3,1}u_{1,3})$ because this is essentially multiplying the different possible paths and routes in two different case. And as a result, we add all the values together as the possible amount of paths to go to.

(b) 1. Main Idea
The edges from v_0 to v_1 can be considered as vectors going for v_0 to v_0. So now we can consider it similar to how we did it in part A, by multiplying the vector by itself k-times to get how many possible ways to get to a specific vertex.

2. Pseudocode

```
Procedure NumOfPath(num of path k, starting pos S, ending pos T, graph G):
  Create an adjacency matrix of v_0 to v_1 (first and second vertex in G)
  A := matrix formed by v_0 x v_1
  F := top right corner of A
  b := binary version of k
  F' := copy of F where we are going to change
  RA := is the same size as F, is the identify matrix
  arr := list of size of binary b where
    1 is the left most digit of b
  while b is not 0:
    n := b and 1 # bit operation
    if n = 1:
      RA = RA * F'
    F' = F' * F'
  ans := value value at (s, t) in RA
  return ans
```

3. Proof of Correctness
   Proof by Induction on K, where k is the number of steps in a possible path.
   Base Case: $K = 1$, trivial. The matrix given will be the adjacency matrix. You can only go to the ones it is adjacent to.
   Induction Hypothesis: This works for $N \leq K$. **Induction Step**: By following our algorithm in 4a. This algorithm should work to tell us how many different paths we can take to take path from s to t of length k.

4. Running Time Analysis
   $O(n^3 logk)$

   The reason for this is because we do matrix multiplication. Matrix multiplication takes $O(n^3)$ time. And because we used binary to figure out which values to multiply by (so that we save time), it becomes $O(logk)$ times that we multiply because binary means base of 2. We want to figure out exactly how many multiplication is necessary (in an optimal manner – which binary gives us the best case). So because we have to do multiplication $logk$ times, we have to multiply by n^3 everytime since that's how long matrix multiplication takes. Hence our resulting run time.

5. Fixing Up a Basketball Cup

1. Main Idea
   We will graph the points given to us, and we will turn the graph into a directed acyclic graph, where any node that starts a cycle will contain a list that has all the values of the other nodes that exists in the cycle. Then we record the number of nodes without any incoming edges. We return the node and the list it contains as the winning teams. From there, any of the teams can be a winning team, depending how we do the playoffs.

2. Pseudocode

```
Procedure FindWinningTeam(list of pairs n):
  create an empty graph D
  for every pair in n:
    for every i, j in pair:
      if i does not exist as a node in D:
        create node i
      if j does not exist as a node in D:
        create node j
      draw a directed edge from i to j in D
  D_dag := a graph with nodes containing lists of nodes in graph D,
     D_dag is D turned into a directed acyclic graph,
     if a node with post-visit value A points to a node with post-visit value B
     such that A < B, it indicates a cycle -- and all nodes in path
     from node B to node get put into a list, and node with post-visit value B
     contains the list of nodes in the cycle.
  num := number of vertex s in V such that inDegree(s) is 0
  if num is 1:
    S := node in V such that inDegree(s) is 0
    return the list of nodes stored in S if there isn't one, return S
  else:
    return null
```

3. Proof of Correctness
   Using the pairs we received, we create a graph. We can do this because we're given two teams and the knowledge of which team beats which. As a result, we can create a graph from using the team and making a directed arrow from the winning team to the losing team. In the end, we will create a graph such that we can depth-first search on. Now we can have a list of post-visited values, and we know that as we follow the path from the biggest post-visited values to the smaller and we find that at one point, the post-visited value A in the node going to post-visited value B in the node where A ¡ B, we know that this is a cycle. The reason for this is because now this proves that there is a backward edge. A backward edge represents a cycle.

   In the cycle, any of the teams can win depending on where we we start. Therefore any of the values in the cycle can win. We can just put all the teams into a list and stick it into the biggest post-visited value node, as we can return this list at the end. Then we count the number of vertices that has no incoming edgest. This means that the specific vertex holds a winning node

because no one can beat it. However if there are more than one, we have no absolute way of deciding who wins because no one can beat either of the teams existing in the node. But if there's only one, we know that there is an absolute winner because that node is where all the possible winner exists because that node does not have any other node of teams who can beat it.

4. Running Time Analysis
   $O(n + |E|)$

   We initially go through all of n in order to build a graph. This itself costed us $O(n)$ time. Then we did a depth first search on the graph. This is $O(|E| + |V|$. Note that $|V| = n$ because it is the number of teams. When we do depth-first search, it's in $O(|E| + |V|)$ time. Then searching through every single vertex to count the number of vertices with 0 incoming edges takes $O(|V|)$. The next step is to find the single vertex that has zero incoming edges is $O(|V|)$. The total run time is now $O(n) + O(|E| + |V|) + O(|E| + |V|) + O(|V|) + O(|V|) = O(5n + 2|E|) \rightarrow O(n + |E|)$ because we ignore constants.

6. Course Prerequisites

1. Main Idea
   We go down every single possible path there is, and every level we go to is one level. We will add it together recursively upward. However we will put it into a dictionary, the node being the key and the level as the value. This will allow us to not recurse downward if we have already calculated it. We make sure if the node has children, we choose the child with the biggest value, as it is the biggest depth.

2. Pseudocode

```
Procedure MinSemesters(graph G):
  V := list of all the vertices
  E := list of all the edges
  values := an initially empty dictionary
    that has node as key, # of semesters required as the value
  create node i
  S := list of vertices without any incoming edges
  for all v in S:
    create a directed edge from i to v
  post-visited := list of nodes in order of its post-visited value,
    the one with the biggest post-visited value is the last element
  Follow path from the last element in post-visited
    by going to the next smallest post-visited value.
  If the node we are on is ever has an outgoing edge to a node with
    the post-visited value is bigger than it, this is a cycle:
      return null -- cannot form a schedule
  return Semester(s, G) - 1

  Procedure Semesters(node A, graph G):
    if A exists in values:
      return values[A]
    else if outDegree(A) is 0:
      values[A] = 1
      return 1
    else:
      max := initally 0, the path that takes the longest
      for all node N, where A has an outgoing edge to N:
        length := Semesters(N, G)
        if length > max:
          max := length
      values[A] = max + 1
      return max + 1
```

3. Proof of Correctness
   Direct Proof: By doing depth-first search, we are able to check to see if there are any cycles because if the post value of node A is smaller the the post value of node B, but A has

an outdegree to B, that means it must be a backward edge by definition. Hence we can automatically return impossible as an answer since there is no such way we can take all the classes.

We know that every single node is a class, which takes one semester to complete. Hence why if there are no children, we return 1 because that's the absolute number of semesters we need to take to complete the class. And as we recurse up a level, the node in the level above should choose the child with the highest number of required history and add one because we are looking for the highest depth.

4. Running Time Analysis
   $O(|V| + |E|)$

   Our initial look through the list of verticies to find all the nodes without any incoming edge will take $O(|V|)$ time. Then we do depth-first search to get all of the post-visited values to make sure there is no cycle, which takes $O(|V| + |E|)$ as the algorithm takes that long. And since we iterate through every single edge and vertex to calculate the length (without repeating because we store the values in a dictionary), we only go through it once. As a result, this is a $O(|E| + |V|)$. When we add all of this up, we get $O(3|V| + |E|) \rightarrow O(|V| + |E|)$ because constants do not matter.