Solutions: Homework 5

CSOR W4231: Analysis of Algorithms I
Fall 2015
Prof. Mihalis Yannakakis

Problem 1

(graded by Nivvedan)

By definition, a bipartite graph G is undirected and must have the property that its set of N nodes can be partitioned into two subsets N_1 and N_2 such that $N_1 \cup N_2 = N$ and $N_1 \cap N_2 = \emptyset$, and every edge in G connects an element of N_1 to an element of N_2

(a)

If an odd-length cycle exists in a graph, we should be able to go around the circle picking an arbitrary node, and an arbitrary order. Without loss of generality let us say that the arbitrary node we picked belongs to N_1. Then, it's adjacent node in the arbitrary order we picked must belong to N_2 by definition. Continuing in the same order along the cycle, we find that the nodes in the cycle alternate between being in N_1 and N_2, until we come to last node that is adjacent to our first node. If our cycle had an odd length, then this last node must be in N_1, but this is adjacent to our first-picked node, which is also in N_1. This doesn't satisfy the definition of a bipartite graph.

Therefore, a graph which contains an odd-length cycle cannot be bipartite.

(b)

We can discover if a graph is bipartite by a simple Breadth-First Search or a Depth-First Search, and colouring adjacent nodes with two different colours. If we reach a contradiction where we detect an adjacent node of the same colour, then the graph is not bipartite. Otherwise, the nodes of the different colours form a valid bipartition. However, detecting an odd-length cycle in a little more involved, and we will do it in the following way.

Lemma 1

A tree is always bipartite. This is because when the tree is rooted at any node, the set of all nodes at odd depths and the set of all nodes at even depths form a valid bipartition.

Lemma 2

When DFS first discovers a contradicting edge, the edge is not a tree edge of the DFS forest. This is because a contradiction can only occur when revisiting an already coloured node, and a tree edge is created only when a node is first visited by DFS. Since DFS on undirected graphs only produces tree edges and back edges, the contradicting edge must be a back edge.

Claim

The contradicting edge (u, v) forms an odd-length cycle along with the $u - v$ path in the DFS forest.

To see why this claim is true, note that the existence of the edge (u, v) implies that u and v must be part of the same connected component (tree) in the DFS forest. We shall call this the DFS tree. Since a tree is always bi-partite, our DFS algorithm must have successfully coloured the $u - v$ path in the DFS tree with alternating colours, but since (u, v) is a contradicting edge, u and v must be of the same colour, which implies that the path has an odd number of nodes. Along with the edge (u, v) this path forms an odd-length cycle.

```
function IS-BIPARTITE(G)
    for node in G.nodes do
        node.colour = 0
    end for
    for node in G.nodes do
        if node.colour == 0 then
            result, cycle, cycleComplete = FIND-NODES(G, node, 1)
            if result == False then
                return result, cycle
            end if
        end if
    end for
/* The graph is bipartite if control reaches here. We now find the bipartition. */
    nodes1 = []
    nodes2 = []
    for node in G.nodes do
        if node.colour == 1 then
            nodes1.append(node)
        else
            nodes2.append(node)
        end if
    end for
    return True, [nodes1, nodes2]
end function
```

```
function FIND-NODES(G, node, colour)
    node.colour = colour
    if colour == 1 then
        colour = 2
    else
        colour = 1
    end if
    for neighbour in node.neighbours do
        if neighbour.colour == node.colour then
            cycle = [neighbour, node]
            return False, cycle, False
        end if
    end for
    for neighbour in node.neighbours do
        if neighbour.colour == 0 then
            result, cycle, cycleComplete = FIND-NODES(G, neighbour, colour)
            if result == False then
                if cycleComplete then
                    return result, cycle, cycleComplete
                else if node == cycle[0] then
                    return result, cycle, True
                else
                    return result, cycle.append(node), cycleComplete
                end if
            end if
        end if
    end for
    return True, [], True
end function
```

IS-BIPARTITE sets up the graph, assigning the colour of all nodes to a default colour, and starts the DFS search from an arbitrary node by calling FIND-NODES. If a an odd cycle is found, the False flag is returned along with the cycle. Else, if all nodes have been visited without an odd cycle, the colours of the nodes are used to discover the bipartition.

FIND-NODES runs the DFS. It first checks if any of the neighbours have the same colour, and if so return the False flag with the start of the cycle (the node and the neighbour). If not, then DFS is run recursively on all the neighbours one by one. If any of the neighbours have found a cycle, and if the current node completes the cycle, then the cycleComplete flag is set. Else, if the cycle is already complete, the return values are simply passed on. Otherwise, the current node is appended to the growing cycle list.

Correctness

Correctness follows from the proof of the lemmas and the claim above. In addition we only have to show that appending the current node to the recursively returned partial cycle is sound. To see why, note that the contradicting edge is a back edge, which means that, in the DFS tree, the contradicting neighbour v is an ancestor of the node u being explored when the contradiction was found. The returns to the calling function until the v is found precisely traces the $u - v$ path in the DFS tree as outlined in the claim.

Running Time

IS-BIPARTITE just runs the setup, and discovers the bipartition once DFS is complete. Apart fro the DFS loop, this only takes O(n) time as constant work is performed per node. FIND-NODES is a modification of DFS which does a constant number of additional checks to the neighbours in each call. So, the running time of the DFS loop in IS-BIPARTITE and all FIND-NODES will be the same as that of DFS which is O(n + e). Therefore, the total running time is O(n + e).

Problem 2

(graded by Mengting)

a.

Proof. Suppose the root of G_π only has one child. If the root is removed from G_π, since there is only one child connected to the root, then all nodes within the child's subtree are still connected to each other. So the removal of the root cannot disconnect G_π.

If the root has two children, as there are no cross edges in depth-first tree of undirected graphs, the two children of the root are only connected via the root node, without any other edges. Thus if the root is removed, then the two subtrees will become disconnected from each other.

Therefore, if and only if the root has at least two children, the root is an articulation point of G_π.

b.

Proof. For a non-root vertex v of G_π, if v has a child s s.t. there doesn't exist any back edge from s or any descendants of s to a proper ancestor of v, then if v is removed, the subtree rooted at s will become disconnected from other part of the tree, say disconnects G_π.

If v's child s has a back edge form s or any descendants of s to a proper ancestor of v, then even after removing v, the subtree rooted at s can still be connected to other part of the tree via the back edge from s or any descendants of s to v's ancestor.

If v has no child at all, then removing v will leave all other nodes still connecting to each other. Thus, v should have a child to be an articulation point at least.

Therefore, the non-root vertex v of G_π is an articulation point of G_π if an only if v has a child s s.t. there exist no back edge from s or any descendants of s to a proper ancestor of v.

c.

Sol. We can modify the DFS algorithm $DFS - VISIT(G, u)$ in the textbook to compute $v.low$ for each vertex.

```
function MODIFIED-DFS($G$)
    for each vertex $u \in G.V$ do
        $u.color = WHITE$
        $u.\pi = NIL$
    end for
    $time = 0$
    for each vertex $u \in G.V$ do
        if $u.color == WHITE$ then
            MODIFIED-DFS-VISIT(G, u)
        end if
    end for
end function
```

```
function MODIFIED-DFS-VISIT($G, u$)
    $time = time + 1$
    $u.d = time$
    $u.low = u.d$
    $u.color = GRAY$
    for each $v \in G.Adj[u]$ do
        if $v.color == WHITE$ then
            $v.\pi = u$
            MODIFIED-DFS-VISIT(G, v)
            $u.low = min(u.low, v.low)$
        else if $v.color == GRAY$ then
            $u.low = min(u.low, v.d)$
        end if
    end for
    $u.color = BLACK$
```

```
16:        $time = time + 1$
17:        $u.f = time$
18:  end function
```

Each time we scan the adjacency list for a vertex u, initialize $u.low$ as $u.d$. And then for each vertex in the adjacency list, if we first scan it, we record $u.low$ as the minimum of $u.low$ and its children's $v.low$. If there exists a back edge, then $u.low$ should be the minimum of $u.low$ and the descendant v of u's $v.d$. After examining all the nodes through DFS, then all the $v.low$ for all $v \in V$ should have been calculated.

And since the running time of DFS is $\Theta(E+V)$, and the modified steps only take constant time for comparison and assignment. Thus the modified version of DFS also takes $\Theta(V+E)$ time. Besides, since G is connected, then $|V-1| \leq |E| \leq V^2$, we have $T(n) = \Theta(V+E) = O(E)$.

d.

Scanning each node in G by DFS, we can check whether one node is articulation point or not by:

- For the root node v, if it has at least two children, then it is an articulation point; otherwise, it isn't. Therefore, we count the number of children: Initialise the count at the beginning of MODIFIED-VISIT(G, u), by adding the line $childcount = 0$; for each new child v of u, that is for each $v \in G.Adj[u]$ s.t. $v.color == WHITE$, increment the childcount. At the end of the MODIFIED-DFS-VISIT procedure, if $u.d == 1$ and $childcount > 1$, then we print u. In this case, u is the root and it has 2 or more children.

- For nonroot vertext v, according to the definition of $v.low$, if follows that a vertex u has a child v s.t. there exists no back edge from v or any descendant of v to a proper ancestor of u if and only if $v.low \geq u.d$. Therefore, a nonroot vertex u is an articulation point if and only if it has a child v s.t. $v.low \geq u.d$. Thus, to print the set of articulation points, we perform the MODIFIED-DFS algorithm of the graph starting from an arbitrary vertex r using the MODIFIED-DFS-VISIT presented in part c with a few changes. In the procedure, add after the line MODIFIED-DFS-VISIT(G, v) the line $if(v.low \geq u.d$ and $u.d \neq 1)$, then print u. That is, if u is not the root node and $v.low \geq u.d$, then we print u as an articulation point.

Since the scanning time via DFS if $\Theta(V+E)$, and the check steps only take constant time. As is proved in part c, $\Theta(V+E) = O(E)$ since G is connected. Therefore, the above algorithm to compute all articulation points takes $O(E)$ time in total.

Problem 3

Graded by Xinyue

3.a

Algorithm: The algorithm is as follows:

1. Construct a graph $G = (V, E)$, where V represents the set of variables and E is initially empty.
2. For every non-strict inequality $x_i \leq x_j$, add edge (i, j) to E.
3. Use the SCC algorithm to find any strongly connected components which runs in $O(n + m)$ time. Group all of the variables in a SCC together into one node, which can be done by storing in an map data structure. The reason that I can do this is all nodes in a SCC with all $\leq$ edges must have same values.
4. Construct a new graph composed of the components, where an edge connects component A and component B if there are variables $x_i \in A, x_j \in B$ such that $x_i \leq x_j$. Then,in a similar fashion, add edges to this graph to represent the strict inequalities like $x_i < x_j$. If we found that x_i and x_j are in the same SCC for an inequality $x_i < x_j$, then it's an inconsistent assignment.
5. Use a DFS search to check if there are any cycles in this new graph which takes $O(n+m)$ time. There is a consistent assignment to the variables if and only if there are no cycles in the new graph.

Justification: The assignment is consistent if and only if there is no cycles of inequalities consisting of a strict inequality. If there is a cycle in step 5, then it must contain a strict inequality, or else the nodes in the cycle would have been merged as one component in step 3. Clearly, if there is such a cycle then combining the inequalities along the cycle yields that a variable must be strictly smaller than itself, which is false, thus there is no solution. On the other hand, non-strict cycles are ok, because all the variables in the cycle can be equal to each other.

Running time: This algorithm runs in $O(n + m)$ as mentioned in the Algorithm section.

3.b

Algorithm:

1. Perform the above algorithm in section 3.a thus we can get a DAG of SCC, here we assume .

2. Perform a topologic sort on this DAG, thus we can get a topological order of the SCCs. Here we assume the graph G is represented by an adjacency list and the adjacency list is ordered by the topological order.

3. Construct an array called *value* to keep the values of each component. Initialize every entry to 1.

4. Then do the following:

```
function SET-VALUE($value, G$)
    for i=1 to G.adjList.length do
        for child in G.adjList[i] do
            if edge (i, child) is i<child then
                $value[child] = max(value[child], value[i] + 1)$
            else
                $value[child] = max(value[child], value[i])$
            end if
        end for
    end for
end function
```

5. To create the final solution, for each SCC i in G, assign $value[i]$ to all the variables in SCC i.

Justification: The algorithm is correct because each SCC with no incoming edge should have value of 1, since that is the lowest possible value. Each strict inequality $i < j$ means that j must be at least one more than i, while $i \leq j$ means that j should be at least as large as i. The algorithm takes the max of each relationship to ensure that all inequalities are satisfied.

Time Complexity: Step 1 takes $O(n + m)$ according to part 3.a, step 2 takes also $O(n+m)$, step 3 takes $O(n)$, step 4 takes $O(n+m)$ and step 5 takes $O(n)$. Thus the overall time complexity is $O(n + m)$.

Problem 4

Graded by Xinyue

4.a

The proof is based on induction.

Inductive Hypothesis: Kruskal's algorithm chooses edge e_i if and only if e_i is in T.

Initialization: Suppose that $i = 1$. At the beginning of Kruskal's algorithm, every node is in its own tree, so e_1 will always be selected. And, e_1 is certainly in T. To show this, we can let $e_1 = (u, v)$ and assume e_1 is not in T. Then there is a path from u to v in T that doesn?t include e_1. But, e_1 is less than or equal to every other edge, so according to property P every edge on that graph would have to have equal weight to e_1. And, ties in weight are broken in L by which edge is in T. So, there is a contradiction, so e_1 is in T.

Inductive step: Suppose that the hypothesis holds for e_1, ..., e_{i-1}. For edge e_i, we need to prove in both directions. Let $e_i = (u, v)$:
$\Rightarrow$: Suppose that Kruskal's algorithm includes e_i in the tree. Then u and v are in different trees so far. Then there is no path from u to v using the edges added so far. If e_i were not added, then some other edge e_j with $j > i$ would have to be added to complete the path. But, this violates property P of the tree because j would have weight greater than e_i, so e_i is in the tree. (Note that if e_i and e_j had equal weight, e_i would be in the tree due to the tiebreaker).

$\Leftarrow$: Suppose that Kruskal's algorithm does not include e_i in the tree. Then Kruskal's algorithm has already selected edges that form a path from u to v. By the inductive hypothesis, this means that there is already a path from u to v in T using e_1, ..., e_{i-1}. Then e_i cannot be in T , because that would form a cycle.

So, e_i is in T if and only if Kruskal's algorithm selects e_i. And, Kruskal's algorithm forms a minimum spanning tree. So, by running Kruskal's on L, we construct T, which must be a minimum spanning tree.

4.b

Algorithm: Suppose the weight of edge $e' = (u, v)$ was decreased. The algorithm is as follows:

1. Create a set of the edges on the path from u to v in T, and call it Q. This can be done using DFS, because in a tree $E = V - 1$, which takes O(n) time.
2. Check if the new weight of e' is smaller than any weight in Q. If not, do nothing. If so, then add e' to T, then find the edge in Q with largest weight and remove it from T. Checking if e' is smaller than any edge in Q takes $O(n)$ time because a tree has at most $n - 1$ edges. Adding e' takes $O(1)$ time because we are adding to two adjacency lists. Finding the largest weight in Q takes $O(n)$ time because there are at most $n - 1$ elements in Q (E = V-1). Removing the edge takes O(n) time because it requires traversing two adjacency lists.

Justification: let's call property P the property that the weight of every non-tree edge (u,v) is at least as large as the weight of all the edges on the u-v path of the tree T. If e' has larger or equal weight than any edge on the path of T between u and v, then property P is still preserved: the new weight of e' is still at least as large as the weight of every edge on the path from u to v, and no other edges are changed.

Otherwise, suppose that we add $e' = (u, v)$ and remove edge $e'' = (u', v')$ to form the new tree T'. Property P is still maintained. First note that because e'' had the largest weight of every edge on Q, and because the path from u' to v' in T' consists of edges in $Q + e' - e''$,e'' is at least as large as every edge on the new path from u' to v'. Further, consider any two other vertices s and t in the graph such that (s, t) is an edge but is not in the new tree. If the old path from s to t did not go through e'', then the new path is the same as in the old tree, and P is preserved. If the old path from s to t did go through e'', call that old path Q'. In the new tree, the path from s to t will consist of a subset of edges in Q', Q, and e'. Because the old tree satisfied property P , the weight of (s, t) is at least as large as the weight of every edge in Q'. Further, the weight of (s, t) is at least as large as e'' because the old tree satisfied property P. And, e'' was the maximum of the edges in Q and e'. So, (s, t)?s weight is at least as large as any weight on the path from s to t in the new tree, so P is maintained.

This algorithm is correct - by removing the largest weight we are maintaining the property P for T, and as we proved in part a, as long as P is maintained, the resulting tree will be a minimum spanning tree.

Running Time: This algorithm runs in $O(n)$ time as analyzed in the algorithm section.

Problem 5

(graded by Qiming Chen)

Part a

Algorithm

Firstly, we put the $n - 1$ edges of $(p(u), u)$ contained in p array to a new graph G' represented in adjacency list L. Secondly, we check if G' is a tree rooted at s.

```
1: function BUILDGRAPH(p)
2:     for each node $u \in N$ do
3:         $L[u] \leftarrow$ new empty list
4:     end for
5:     for each node $u \in N$ do
6:         if $u \neq s$ then
7:             append $u$ to $L[p(u)]$
```

```
 8:         end if
 9:     end for
10:     return $L$
11: end function
```

The building phase takes $O(|N| + |E'|)$ time, where $|E'|$ is the number of edges in graph G'. Considering that there are only $|N| - 1$ edges inserted, the running time is $O(|N|)$.

Next, we check if there is a cycle in G'. We can use DFS, and check the number of nodes that will be visited.

```
1: function DFS(L, node)
2:     $count \leftarrow$ count + 1
3:     $visited[u] \leftarrow 1$
4:     for $child \in L[node]$ do
5:         if $visited[child] = 0$ then
6:             $DFS(L, child)$
7:         end if
8:     end for
9: end function
```

Finally, we check if the number of nodes visited during DFS is $|N|$.

```
 1: function checkTree(p)
 2:     $L \leftarrow$ buildGraph(p)
 3:     for each node $u \in N$ do
 4:         $visited[u] \leftarrow 0$
 5:     end for
 6:     $count \leftarrow 0$
 7:     $DFS(L, s)$
 8:     if $count = |N|$ then
 9:         return true
10:     end if
11:     return false
12: end function
```

If not, we return false.

Correctness

We know that in graph G', every node has only one parent, and there are only $|N| - 1$ edges. Then if we can reach all the nodes using DFS, every node is connected to s, and the graph must be a tree. Then the graph will be a graph rooted at s.

Time Complexity

Building the graph is $O(|N|)$, and doing DFS on a graph with $O(|N|)$ nodes and $O(|N|)$ edges is also $O(|N|)$. So the combined time complexity is $O(|N|)$.

Part b

Algorithm

We can go through every edge in G, and use an array $mark$ to record if edge $(p(u), u)$ is contained in the graph G for each node $u \in N$.

```
function checkEdge(G, p)
    for each node $u \in N$ do
        $mark[u] \leftarrow 0$
    end for
    for each node $u \in N$ do
        for each node $v \in adj[u]$ do
            if $u = p[v]$ then
                $mark[v] \leftarrow 1$
            end if
        end for
    end for
    for each node $u \in N$ do
        if $u \neq s$ and $mark[u] = 0$ then
            return false
        end if
    end for
    return true
end function
```

Correctness

We went through every edge in G, and if an edge $(p(u), u)$ is in G, $mark[u]$ will then and only then be marked as 1. Thus we will return false if and only if there exists an edge that is not in G.

Time Complexity

Initializing the array is $O(|N|)$. And go through every edge of the graph is $O(|N| + |E|)$. And finally check the array is $O(|N|)$. So the combined time complexity is $O(|N| + |E|)$.

Part c

Algorithm

We can go through every edge in G, and use an array w to record the weight of edge $(p(u), u)$ for each node $u \in N$.

```
function recordW(G, p)
    for each node $u \in N$ do
        for each node $v \in adj[u]$ do
            if $u = p[v]$ then
                $w[v] \leftarrow weight(u, v)$
            end if
        end for
    end for
end function
```

Then we update the d value for each node by traversing the tree G'.

```
function COMPUTED(L, p, node)
    if $p[node] \neq null$ then
        $d[node] \leftarrow d[p[node]] + w[node]$
    end if
    for $child \in L[node]$ do
        computeD(L, p, child)
    end for
end function
```

And the whole procedure is as below:

```
function COMPUTEARRAYD(G, L, p)
    $d[s] \leftarrow 0$
    recordW(G, p)
    computeD(L, p, s)
end function
```

Correctness

On tree T, for every node, there is only one path to the root node s, that is through it's parent. So every node's weight of the path to root will be his parent's weight of the path plus the weight from the node to it's parent. And also the d value of the root must be 0. So the correctness of the algorithm lies with induction. The base case is we correctly computed the d value for the root. And if the d value of the parent is correct, the children are also computed correct. By induction on the tree, we got every value correct.

Time Complexity

Retrieving the weight of the edges is $O(|N| + |E|)$ since we need to traverse the graph edges. And updating the d array is done by DFS on the tree, so it is $O(|N|)$. So the combined time complexity is $O(|N| + |E|)$.

Part d

Algorithm

Given the array d in part c, now we want to check if for any edge (u, v) in G, it is a short cut, such that $d[u] + w(u, v) < d[v]$.

```
function CHECKSHORTEST(G, d)
    for each node $u \in N$ do
        for each node $v \in adj[u]$ do
            if $d[u] + weight(u, v) < d[v]$ then
                return false
            end if
        end for
    end for
    return true
end function
```

Correctness

We checked if for any edge (u, v) in G, it is a short cut, such that $d[u] + w(u, v) < d[v]$.

If yes, then $d[v]$ is obviously not the distance from s, because we can go from s to v using the path $d[u]$ and (u, v), and $d[v]$ will not be optimal.

If not, we can prove that d holds the distance from s to each node and T is the shortest path tree as follows: For an arbitrary node r except s, pick an arbitrary path from s to r: $s - n_1 - n_2 - ... - n_k - r$, where s and r can also be denoted as n_0 and n_{k+1}. Since for every edge (n_{i-1}, n_i), we have $d[n_{i-1}] + weight(n_{i-1}, n_i) \geq d[n_i]$, we sum the formula and can get $\sum_{i=1}^{k+1} d[n_i] \leq \sum_{i=1}^{k+1} d[n_{i-1}] + \sum_{i=1}^{k+1} w(n_{i-1}, n_i)$, i.e. $d[n_{k+1}] \leq d[n_0] + w(path(s, r))$, i.e. $w(path(s, r)) \geq d[r]$. Thus $d[r]$ is the minimum of all the possible paths from s to r.

Thus our algorithm return true if and only if T is the shortest path tree.

Time Complexity

We went through every edge in the adjacency list, and every computation needed for the edge is $O(1)$ So the time complexity is $O(|N| + |E|)$.