CSOR 4231: Analysis of Algorithms I, Problem Set 5

Jing Qian - jq2282

November 23, 2018

Problem 1

a.

We could prove this by contradition. Assume a bipartite graph contains a cycle C of odd lenth. Let $v_1, v_2, \cdots, v_k$ be all the nodes listed in the cycle C and k is an odd number. Then there is an edge between v_1 and v_2, an edge between v_2 and v_3, $\cdots$, and an edge between v_k and v_1 because C is a cycle.

Since it is a bipartite graph, every edge in the bipartite graph connects a node of N_1 with a node of N_2. We could assume that all the odd nodes are in the subset N_1 and all the even nodes are in the subset N_2 (Similar provement works if odd nodes are in N_2 while even nodes are in N_1). In other words, $v_1, v_3, \cdots v_k \in N_1$ and $v_2, v_4, \cdots v_{k-1} \in N_2$.

Above we show that modes v_1 and v_k are in the same subset and there is an edge between v_1 and v_k, which disallow the property of bipartite graph that every edge connects nodes from different subsets. Our assumption that a bipartite graph contains a cycle C of odd lenth is wrong. So if a graph contains a cycle of odd length then it is not bipartite.

b.

We could solve this problem using the property of bipartite that every edge connects nodes from different subsets and find the cyle of odd length according to Part a. With a modified BFS, let nodes with the odd depth be in subset N_1 and nodes with even depth be in subset N_2 and check whether there is an edge connecting two nodes in the same subset.

If there is an edge connecting two nodes u and v from the same subset, we trace back the parents of these two nodes in iteration and since it is a cycle, these two nodes would have a common ancestor ca. The cycle $u, \cdots, ca, \cdots, v, u$ would have an odd length and is what we want. If we scan through all nodes and found no cycle with odd length, the graph would be bipartite and we separate nodes into the subset N_1 and N_2 according to the parity of their depth.

The code is as following:

Check_Bipartite

```
CHECK_BIPARTITE(G)
Starting at any node s:
```

```
d[s] = 0
p[s] = NONE
Q = {s}
for each v in N-{s}: #N is the set of all nodes in G
    d[v] = Infinity
    p[v] = NONE
while Q not empty:
    u = Dequeue(Q)
    for each v in Adj[u]:
        if d[v] = Infinity:
            d[v] = d[u]+1
            p[v] = u
            Enqueue(Q, v)
        else if d[v] mod 2 == d[u] mod 2:
            C = {} #C is the cycle of odd lenth
            while v != u:
                Enqueue(C, u, v)
                u = p[u]
                v = p[v]
            Enqueue(C, u)
            Exit
N1 = {}
N2 = {}
for each v in G:
    if d[v] mod 2 == 1:
        N1 = N1 + {v}
    else:
        N2 = N2 + {v}
```

The time complexity of this algorithm is O(n+e). The initialization (line 48-54) takes O(n) time. The while loop (line 55-69) takes O(e) time. If the graph is bipartite, the separation (line 72-76) takes O(n) time. In total, O(n+e).

C

Let's assume all the 1-variables are nodes belong to the same subset N_1 and all the 0-variables are nodes belong to the same subset N_2. Then the inequality $x_i \neq x_j$ is the edge between $x_i \in N_1$ and $x_j \in N_2$ or reverse. We could abstract this problem as an bipartite graph problem and use the algorithm in Part 1.b. If the graph is not bipartite, there are edges between nodes in the same subset. In other words, for $x_i \neq x_j$, x_i and x_j both are 0 (or 1), which is impossible to have a solution. If the graph is bipartite, there are no edges between nodes in the same subset and all the constraints are consistent. We could separate the variables into two subsets and that is the solution we want. The time complexity is the same with that in Part 1.b., which is O(nodes + edges). So O(n+m)

Problem 2

Dijkstra's algorithm outputs the minimum weight path of graph G from node s to t. To choose the minimum weight path with fewest number of edges, we only need to augment an variable e to keep record of the number of edges in the path and choose the minimal e when there is a tie between different path with the same weight.

The code is as following:

ModifiedDijkstra

```
MODIFIED_DIJKSTRA(G, s, t)
d[s] = 0
p[s] = NONE
e[s] = 0
for each v in N-{s}: #N is the set of all nodes in G
    d[v] = Infinity
    p[v] = NONE
    e[v] = Infinity
Q = N
while Q not empty:
    u = Extract-Min(Q)
    for each v in Adj[u]:
        if d[v] > d[u] + w(u, v):
            d[v] = d[u] + w(u, v)
            p[v] = u
            e[v] = e[u] + 1
        else if d[v] == d[u] + w(u, v) and e[v] > e[u] + 1:
            p[v] = u
            e[v] = e[u] + 1
```

According to the code above, we only add one variable and an *else if* in every iteration of the while loop. So the time complexity would be the same with original Dijkstra algorithm. And we still output the minimum weight path, but choose the one with the minimum number of edges when there are multiple minimum weight paths.

Problem 3

a.

First, if there is path from x_v to x_u, we have $x_v \leq x_i < x_j, \cdots, \leq x_u$. Hence $x_v \leq x_u$, which contradicts the assumption that set C contains a strict inequality $x_u < x_v$. So If the set C contains a strict inequality $x_u < x_v$ such that the graph G has a path from x_v to x_u, it is not consistent.

Second, if the set C is not consistent, there must be a cycle in the graph G. If all edges in the cycle are non-strict inequality, i.e. $x_u \leq x_v$, we still have a consistent set C because all the nodes could be the same. But if one edge in the cycle is strict inequality, set C is not consistent. Let's say the edge is $x_u < x_v$, then the rest edges of this cycle becomes a path from x_v to x_u. So if the set C is not consistent, it must contain a strict inequality $x_u < x_v$ such that the graph G has a path from x_v to x_u.

In conclusion, the set C is not consistent if and only if it contains a strict inequality $x_u < x_v$ such that the graph G has a path from x_v to x_u.

b.

We could use the property in Part 3.a. to solve this problem. If there is a strict inequality $x_u < x_v$ such that the graph G has a path from x_v to x_u, set C is not consistent. We could use a modified toplogical sort to solve this problem. We sort using DFS and color every nodes to show whether they are visted or not. If expaning the node u and find its adjacent node v has been reached, we check whether the constrain between u and v is a strict inequality. If it is, C is not consistent. If the constrain between u and v is not strict, we go to the other node w of the back edge (u, w) and check whether there is a strict inequal edge in the path (w, u). If there is, C is not consistent. If there isn't, go to the other adjacent nodes of v and continue. This algorithm still takes $O(n + m)$ like normal DFS because we only add $O(m)$ edge check.

c

We could use the algorithm in Part 3.b to solve this problem. If the set of inequalities has a solution over the positive integers, the set is consistent. We could use the algorithm in Part 3.b. to output a path of all nodes in the graph G: $x'_1 \leq x'_2 < \cdots, \leq x'_n$. Start with the minimal variable x'_1, we assign it as 1. If $x'_1 \leq x'_2$, $x'_2 = 1$. Or if $x'_1 < x'_2$, $x'_2 = 2$. To generalize, if we got the value of x'_i, if $x'_i \leq x'_j$, $x'_j = x'_i$; if $x'_i < x'_j$, $x'_j = x'_i + 1$. The assignment takes an extra O(m) time than the algorithm in Part 3.b. because we have to check all the edges in the path. But in total, the time complexity remians O(n+m).

Problem 4

a.

First, let's assume that the root R of G_π is not an articulation point of G while it has at least two children in G_π: C_1 and C_2. Since R is not articulation point, according to the definition, C_1 and C_2 are connected in a path without R. Since it is a DFS, if C_2 is reachable to C_1 without going through R, C_1 must be an ancestor of C_2. However, we know that the only ancestor of C_2 is the root R, which contradicts with the induction that C_1 is an ancestor of C_2. So, if the root R has at least two children in G_π, it is an articulation point of G.

Second, if the root R of G_π is an articulation point of G while it has only one or no child. If R has no child, G contains only one node. When remove R, there is nothing inside G and hence no disconnection. So R is not an articulation point if it has no child. If R has one child, when remove R, the child of R becomes the root of the new DFS tree and the graph G remains connected. So R is not an articulation point if it has one child.

In conclusion, the root of G_π is an articulation point of G if and only if it has at least two children in G_π.

b.

First, if v has a child s such that there is no back edge from s or any descendent of s to a proper ancestor of v, when we remove v, the child s has no connection with the ancestors of v and the graph G become disconnected. So v is the articulation point.

Second, assuming v is the articulation point and it has a child s such that there is a back edge from s or any descendent of s to a proper ancestor of v, let's say there is a back edge $E(s, w)$ in which w is a proper ancestor of v (Similar provement if the back edge is $E(t, w)$ in which t is a descendent of s). When we remove v, the child s and its descendents are still connected with w and all the descendents or ancestors of w. G still remains connected, which contradicts with the assumption that v is the articulation point.

In conclusion, v is the articulation point if and only if it has a child s such that there is no back edge from s or any descendent of s to a proper ancestor of v.

c.

According to the definition of $v.low$, the *low* property of node v is determined by itself and its descendant. So we could use a bottom up method to compute $v.low$ all vertices.

First, let's consider the situation that v is a leaf. If there is no back edge for v or we don't take v as a descendant of itself, $v.low = v.d$. If there is a back edge to a proper ancestor $w$of v and we take v as a descendat of itself, $v.low = w.d$.

Then, let's consider the situation that v is an intermediate or top node. The $v.low$ is the minimum of $v.d$ and the $c.low$ of all its childrens c. We iterate from bottom leaves to top root and calculate the $v.low$ for all vertices $v \in V$. Because this algorithm goes through all the edges and only once, the time complexity if $O(E)$.

d.

We could combine the algorithm in Part 4.c. and the property in Part 4.b. to solve this problem.

The node v is the articulation point if and only if it has a child s such that there is no back edge from s or any descendent of s to a proper ancestor of v. If the node v has a child s such that there is a back edge from s or any descendent of s to a proper ancestor of v, the $v.low$ would be $w.d$ which is smaller than $w.d$.

So we could modify the algorithm in Part 4.c. For every v, after we compute the $v.low$, we check whether it equals to $v.d$. If $v.low = v.d$, it has no child s such that there is a back edge from s or any descendent of s to a proper ancestor of v. Then v is an articulation point. Else, v is not an articulation point. Since this modified algorithm only adds one equality check in every iteration of the original one, the time complexity remains the same, $O(E)$. So we could get all articulation points in $O(E)$ time.

Problem 5

a.

Let's assume there are t edges in the minimum spanning tree Twhile not in the spanning tree T_k computed by Kruskal's algorithm , i.e. $e_i \notin T_k$ and $e_i \in T$ (i from 1 to t).

Take the edge $e_1 \notin T_k$, $e_1 \cup T_k$ must generate a cycle C since T is a spanning tree. According to the Kruskal's algorithm, every edge in the cycle C is smaller or equal to the weight of e_1. Let's assume there's an edge f_1 in the cycle C which $f_1 \in T_k$ and $f_1 \notin T$. We could construct a new spanning tree $T_1 = T_k \cup e_1 - f_1$. Since the weight of f_1 is smaller or equal to that of e_1, the weight of T_1 is larger or equal to that of T_k, $T_k \leq T_1$.

Using the similar method, we could generate $T_2 = T_1 \cup e_2 - f_2$ and $T_1 \leq T_2$, $T_{j+1} = T_j \cup e_{j+1} - f_{j+1}$ and $T_j \leq T_{j+1}$.

Finally, all the t different edges are modified in T_k and T_k becomes T and we have: $T_k \leq T_1 \leq T_2 \cdots \leq T_j \leq T_{j+1} \leq T$. Since T is the minimum spanning tree, the only way the inequality above holds is that every spanning tree has the same weight: $T_k = T_1 = T_2 \cdots = T_j = T_{j+1} = T$. And all the deleted f_j equal to their corresponding added e_j. So T_k is minimum spanning tree.

b.

If we decrease the weight of one of the edges that is not in T, it may or may not affect the minimum spanning tree T. Let's call this modified edge $e(u, v)$ connecting two nodes u and v. Since e is not in T, there must be a path from node u to v in T. Then we could check whether there is an edge in T has a larger weight than e. If there is an edge $f(i, j)$ in the path $u - \cdots - i - j - v$ having larger weight than $e(u, v)$, we remove the edge $f(i, j)$ from T, union the edge $e(u, v)$, and have a new minimum spanning tree T^* which has smaller weight than T. If there is no edge with larger weight than e, T remains the same.

In this algorithm, we take $O(n)$ time to find the node u because T is given by adjacency list representations. Then we follow the path u to v in T and check the weight of every edge along this path, which also takes $O(n)$ time. If we modify T, the union between T and new edge e takes $O(1)$ time. So in total, the time complexity of the algorithm is $O(n)$.