CMPSC 565: Design and Analysis of Algorithm **(Date: 09/16/2022)**

Homework #1 Solution

Instructor: Dr. Antonio Blanca *TA:* Guneykan Ozgul

Problem 1. Processing the Graph

(a.) There are two cases possible: $\texttt{pre}(u) < \texttt{pre}(v) < \texttt{post}(v) < \texttt{post}(u)$ or $\texttt{pre}(v) < \texttt{post}(v) < \texttt{pre}(u) < \texttt{post}(u)$. In the first case, u is an ancestor of v. In the second case, v was popped off the stack without looking at u. However, since there is an edge between them and we look at all neighbors of v, this cannot happen. So, the given statement is true.

(b.) Do a DFS on the tree starting from r and store the previsit and postvisit times for each node. Since the given graph is a tree, and we started at the root, the DFS tree is the same as the given tree. Thus, u is an ancestor of v if and only if $\texttt{pre}(u) < \texttt{pre}(v) < \texttt{post}(v) < \texttt{post}(u)$.

Problem 2. One-way Streets

a. We view the intersections as vertices of a graph with the streets being directed edges, since they are one-way. Then the claim is equivalent to saying that this graph is strongly connected. This is true iff the graph has only one strongly connected component, which can be checked in linear time.

(b.) The claim says that starting from Westgate Building, one cannot get to any other SCC in the graph. This is equivalent to saying that the SCC containing the vertex corresponding to Westgate Bldg is a sink component. This can easily be determined in linear time by first finding the components, and then running another DFS from the vertex corresponding to Westgate Bldg, to check if any edges go out of the component. *(In fact, it can even be done* while *decomposing the graph into SCCs by noting that the algorithm for decomposing progressively removes sinks from the graph at every stage. A component found by the algorithm is a sink iff there are no edges going out of the component found* before *it.)*

Problem 3. Make It Strongly Connected

The idea is to find all the SCCs first (if there is only one, we are good) and shrink all the vertices on a SCC into one vertex (eliminate inner edges). Then, look at each SCC as a vertex. This way to construct a new graph G' whose vertices are the SCC of G. This way, G' is a DAG. So the minimum number of edges that we have to add is equal to $max(|V_{in=0}|, |V_{out=0}|)$, where $V_{in=0}/V_{out=0}$ is all the vertices in G' with in/out-degree 0.
Finding all SCCs takes $\mathcal{O}(|V|+|E|)$. G' has $\mathcal{O}(|V|)$ vertices and converting this DAG into a cycle takes $\mathcal{O}(|V|)$. So we get the total running time of $\mathcal{O}(|V|+|E|)$.
We have to keep track of all the SCCs so we need $\mathcal{O}(|V|)$ space. Then G' also has $\mathcal{O}(|V|)$ vertices and edges, so this can also be done using $\mathcal{O}(|V|)$ space.

Problem 4. 2SAT

a. Notice that the intuition behind the construction of G_I is the following:

$$(x \vee y) \equiv ((\hat{x} \Rightarrow y) \wedge (\hat{y} \Rightarrow x))$$

When x is false, if the clause is to be true, y should be true and vice versa (corresponding to the second directed edge). So, we can look at each arrow as an assignment.
If there is a path from literal v to literal u in G_I, then there is a path from $\hat{u}$ to $\hat{v}$, too. The reason is as follows. Let the path form v to u be $P = v, v_1, v_2, \cdots, v_t, u$. This means that we have the following edges $(v, v_1), (v_1, v_2), \cdots, (v_t, u)$. Notice that by our construction, if we have the edge (v, v_1), we also add the edge $(\hat{v}_1, \hat{v})$. Hence, we have the following edges in G_I: $(\hat{u}, \hat{v}_t), (\hat{v}_t, \hat{v}_{t-1}), \cdots, (\hat{v}_1, \hat{v})$, which is a path from from $\hat{u}$ to $\hat{v}$ in G_I.
Now, assume that there is a path from literal x to literal $\hat{x}$. Show this path by $P' = x, x_1, x_2, \cdots, x_r, \hat{x}$. First of all, they belong to the same SCC due to the claim that we proved above. Let S be such a SCC. Either x or $\hat{x}$ must be true. Without the loss of generality, assume x is true (which means $\hat{x}$ is false). This implies that x_1 is true. This propagates all the way to x_r and makes x_r to be true if I has to be satisfied. x_r being true implies that $\hat{x}$ should be true as well. This is a contradiction since both x and $\hat{x}$ cannot be true at the same time.

(b.) Order SCCs in topological order: $C_1, C_2, \cdots, C_t$. Now, we describe a truth assignment if no SCC has a literal and its negation in it.
Assign TRUE to literal x iff it appears after its negation in the topological order (we have a topological order, so one must appear first). We prove that this type of assigning values, satisfies I.

Claim. There is no edge like (u, v) such that u is a assigned TRUE and v is assigned FALSE.

Proof: Assume for the sake contradiction that $Val(u) = TRUE$ and $Val(v) = FLASE$. Let u be in SCC C_j. On the other hand, we also have the edge $(\hat{v}, \hat{u})$. Since $Val(u) = TRUE$, so $Val(\hat{u}) = FALSE$. Based on part (a), $\hat{u}$ is in some other SCC C_i. Due to our value assigning procedure, the fact that u is assigned to be TRUE means that C_i appears before C_j in the topological order ($i < j$). Now, let's look at v. Let the SCC containing $\hat{v}$ be C_k. Since $Val(v) = FALSE$, it means that C_k appears after C_j (so, C_j sits between C_i and C_k) but any edge from C_k to C_i contradicts the topological order.
□

Now, the last part tis to show that this guarantees that I is satisfied. Assume there is a clause like $x \vee y$ which is not satisfied. i.e. $Val(x) = Val(y) = FALSE$. This means we have an edge $(\hat{x}, y)$ such that $\hat{x} = TRUE$ and $y = FALSE$, which contradicts the claim. So, no such clause exists.

Problem 5. Verify max flow

First, we construct the residual graph G^f, in linear time $O(|V| + |E|)$. Second, we can perform DFS on G^f to determine the set K of nodes that are reachable from S. This also takes linear $O(|V| + |E|)$ time. If K contains T then the flow is not maximum. If K doesn't contain T, then we calculate the value of the occurring cut in linear time and compare it to the given flow. If and only if those quantities are equal, the flow given to us is indeed maximum.

Problem 6. Ford Fulkerson

(a.) You can pass a units of flow from the path $s - V_1 - t$, and a units from $s - V_4 - t$. Also, we can pass 1 untit of flow from the path $s - V_2 - V_3 - t$, so the total amount of flow you can pass is $2a + 1$

(b.) The step of Ford-Fulkerson algorithm are in the Table 1

(c.)

Step	Augmenting path	Sent flow	Edge $V_2 - V_1$ in Residual graph	Edge $V_4 - V_3$ in Residual graph	Edge $V_2 - V_3$ in Residual graph
0			$r^0 = 1$	r	1
1	p_0	1	r^0	r^1	0
2	p_1	r^1	r^2	0	r^1
3	p_2	r^1	r^2	r^1	0
4	p_1	r^2	0	r^3	r^2
5	p_3	r^2	r^2	r^3	0

Table 1: Ford-Fulkerson iterations

After step 1 as well as after step 5, the residual capacities of edges $V_2 - V_1$, $V_4 - V_3$ and $V_2 - V_3$ are in the form r^n, r^{n+1} and 0, respectively, for some $n \in \mathbb{N}$. This means that we can use augmenting paths p_1, p_2, p_1 and p_3 infinitely many times and residual capacities of these edges will always be in the same form. Total flow in the network after step 5 is $1 + 2(r^1 + r^2)$. If we continue to use augmenting paths as above, the total flow converges to $1 + 2\sum_{i=1}^{\infty} r^i = 3 + 2r$.

Problem 7. Removing an edge

Construct the residual graph G_r. Add capacity 1 on the edge that has been increased. Now, apply BFS to find an augmenting path. If now such path is found, the max flow stays put, otherwise we can update the max flow. Notice that we just need to do this only once since the augmenting path (if exists) increases the flow by 1 and this is the maximum possible increase. Running a BFS and finding an augmenting path, both take $\mathcal{O}(|V| + |E|)$ time.

Problem 8. Reducing maximum flow

If the minimum $s - t$ cut has size $\leq k$, then we can reduce the flow to 0. Otherwise, let $f > k$ be the value of the maximum $s - t$ flow. We identify a minimum $s - t$ cut (A, B), and delete k edges out of A. The resulting subgraph has a maximum flow value of at most $f - k$.

But we claim that for any set of edges F of size k, the subgraph $G' = (V, E - F)$ has an $s - t$ flow of value at least $f - k$. Indeed, consider any cut (A, B) of G'. There are at least f edges out of A in G, and at most k have been deleted, so there are at least $f - k$ edges out of A in G'. Thus, the minimum cut in G' has value at least $f - k$, and so there is a flow of at least this value.

Problem 9. Moving Branches

This problem can be formulated as min-cut problem. How we form the graph can be seen from the pseudo-code below. Notice that the min-cut of the graph below gives us two partitions. Lets call the partition that doesn't include the source node P1 and the partition that doesn't include the sink P2. Thus this gives us $\sum_{i \in P1} a_i + \sum_{j \in P2} b_j + \sum_{(i,j) \in E, i \in P1, j \in P2} c_{ij}$. This is equal to total cost that we wanted to minimize. Thus solving the min-cut gives us the minimized total cost. The graph is of polynomial size so creating the graph and running the min-cut algorithm takes polynomial time hence total run time is polynomial.

Note that for each branch i cutting the a_i edge means that the branch stays and cutting the b_i branch means that the branch moves; so the branches whose nodes are in in P1 are the branches that stay and the branches in P2 are the branches that move.

Algorithm 1 MovingBranches

```
procedure FINDOPTIMALBRANCHES($B[]$, $a[]$, $b[]$)
    $G$ = empty graph
    Create source node $S$ and add it to $G$
    Create sink node $T$ and add it to $G$
    for $branch_i$ in $B$ do
        Create a new node for $branch_i$ and add it to $G$
        Create edge from $S$ to $branch_i$ with capacity $a_i$ and add it to G
        Create edge from $branch_i$ to T with capacity $b_i$ and add it to G
        for $branch_j \neq branch_i$ in $B$ do
            Create a new edge with capacity $c_{ij}$ from/to $branch_i$ to/from $branch_j$ and add it to $G$
        end for
    end for
    Find minimum $s-t$ cut of $G$ and return $s - \{S\}$
end procedure
```

Problem 10. Max flow with queries

Finding maximum flow from v to u is equivalent to find the minimum cut from v to u. Let's use the later interpretation to solve the problem.
Suppose there is a single special edge ($k = 1$), on each query there are two options, either this edge belong to the minimum cut or it doesn't. If the edge doesn't belong to the minimum cut, the value of the cut won't change even if we increase the capacity of each edge arbitrarily (let's say to ∞). On the other hand, if the edge belong to the cut, then the value of the cut will be equal to the capacity of the edge + the value of the cut if the capacity of the edge was 0. With this in mind we can compute each query in $\mathcal{O}(1)$.
Let MC_0 be the value of the minimum cut if the capacity of the special edge is 0, and MC_∞ be the value of the minimum cut if the capacity of the special edge is ∞. Then for each query c_i the minimum cut will be equal to $min(MC_\infty, MC_0 + c_i)$.
We can generalize this ideas to multiple edges in the following way. For each subset of special edges, they either belong to the minimum cut, or they don't. If we fix a subset S and say those are among the special edges the only ones that belong to the minimum cut, then the value of the cut will be equal to the sum of the values of the capacities of these edges plus the minimum cut in the graph were each of these edges has capacity 0 and other special edges has capacity ∞. In a similar way as we did for the case of $k = 1$ we can pre-compute 2^k cuts, fixing S as each possible set in $\mathcal{O}(2^k max\_flow(n,m) + q2^k)$, and then answer each query in $\mathcal{O}(2^k)$. The overall complexity of this solution will be $\mathcal{O}(2^k max\_flow(n,m) + q2^k)$.