Dept. of Computer Science **University of Toronto**

Assignment # 2 (Due July 13, 11:59 pm)

CSC 373H1 **Summer 2019**

Worth: 5%

1. **[20 marks]**

   (a) **[10 marks]** Give an efficient algorithm that takes the following inputs:
   - $G = (V, E)$, a connected undirected graph,
   - $w : E \to \mathbb{Z}^+$, a weight function for the edges of G,
   - $T \subseteq E$, a minimum spanning tree of G,
   - $e_1 = \{u, v\} \notin E$ (for $u, v \in V$), an edge not in G,
   - $w_1 \in \mathbb{Z}^+$, a weight for e_1,

   and that outputs a minimum spanning tree T_1 for the graph $G_1 = (V, E \cup \{e_1\})$ with $w(e_1) = w_1$.

   For full marks, your algorithm must be more efficient than computing a MST for G_1 from scratch. Justify that this is the case by analysing your algorithm's worst-case running time.

   Finally, write a detailed proof that your algorithm is correct. (Note that this argument of correctness will be worth at least as much as the algorithm itself.)

   (b) **[10 marks]** Give an efficient algorithm that takes the following inputs:
   - $G = (V, E)$, a connected undirected graph,
   - $w : E \to \mathbb{Z}^+$, a weight function for the edges of G,
   - $T \subseteq E$, a minimum spanning tree of G,
   - $e_0 \in E$, an edge in G,

   and that outputs a minimum spanning tree T_0 for the graph $G_0 = (V, E - \{e_0\})$, if G_0 is still connected — your algorithm should output the special value Nil if G_0 is disconnected.

   For full marks, your algorithm must be more efficient than computing a MST for G_0 from scratch. Justify that this is the case by analysing your algorithm's worst-case running time.

   Finally, write a detailed proof that your algorithm is correct. (Note that this argument of correctness will be worth at least as much as the algorithm itself.)

Solution. (a) **Algorithm:**

```
AddMST($G, w, T, e_1, w_1$):
    Use BFS on $T$ to find the unique path $P$ from $u$ to $v$ in $T$ (where $\{u, v\} = e_1$).
    Let $e_0$ be an edge on $P$ with maximum weight.
    if $w(e_0) > w_1$:
        return $T - \{e_0\} \cup \{e_1\}$
    else:
        return $T$
```

Runtime: BFS takes time $\Theta(|V| + |E|)$; finding e_0 takes time $\mathcal{O}(|E|)$; total time is $\Theta(|V| + |E|)$.

Correctness: Because edge e_1 is the only difference between G and G_1, it is the only edge whose addition may result in a different MST from T. This happens only when e_1 can be swapped with some edge in T with higher weight: exactly what the algorithm does.

(b) **Algorithm:**

```
DelMST($G, w, T, e_0$):
    if $e_0 \notin T$:
        return $T$
    else:
        Let $e_0 = \{u, v\}$.
```

```
Run BFS on the edges of $T - \{e_0\}$, starting from $u$;
    assign colour *white* to every vertex encountered.
Run BFS on the edges of $T - \{e_0\}$, starting from $v$;
    assign colour *black* to every vertex encountered.
Loop over every edge in $E - \{e_0\}$ to find a minimum-weight edge $e_1$
    with one *white* endpoint and one *black* endpoint.
**if** there is no such edge $e_1$:
    **return** NIL
**else**:
    **return** $T - \{e_0\} \cup \{e_1\}$
```

Runtime: BFS takes time $\Theta(|V| + |E|)$; finding e_1 takes time $\mathcal{O}(|E|)$; total time is $\Theta(|V| + |E|)$.

Correctness: Because edge e_0 is the only difference between G and G_1, it is the only edge whose removal may result in a different MST from T. From the proof of correctness of Kruskal's algorithm, we know that it is always "safe" to add an edge of minimum weight between two connected components (while constructing a MST): exactly what the algorithm does.

□

2. **[20 marks]** Given a flow network $G = (V, E)$, a **maximum bottleneck path** between two vertices u and v is a path between u and v that allows the most flow to go through from u to v. As usual, the maximum amount of flow that can go through such a path is equal to the capacity of the bottleneck edge on that path.

   Give an algorithm that finds a maximum bottleneck path between two given vertices. Prove the correctness and analyze the running time of your algorithm. For full marks, your algorithm must be more efficient than brute-force.

Solution. **Algorithm:**

```
MAXBOTTLENECKPATH($G, u, v$):
    for each vertex $w \in G$ do
        $maxFlowTo[w] = 0$
        $prev[w]$ = Undefined
        add $w$ to $Q$
    end for
    $maxFlowTo[u] = \infty$
    $R = \{\}$
    while $R \neq V$ do
        $z$ = vertex $\notin R$ with maximum $maxFlowTo[z]$
        $R = R \cup \{z\}$
        for each neighbour $w$ of $z$ do
            $alt = \min\{maxFlowTo[z], f_{(z,w)}\}$
            if $alt > maxFlowTo[w]$ then
                $maxFlowTo[w] = alt$
                $prev[w] = z$
            end if
        end for
    end while
```

```
$S = []$
$w = v$
while $prev[w]$ is defined:
    insert $w$ at the beginning of $S$
    $w = prev[w]$
end while
insert $u$ at the beginning of $S$
return $S$
```

Worst-case running time is $\mathcal{O}(|E|\log|V|)$ using a max priority queue implementation, analogous to the original Dijkstra algorithm.

Proof of correctness: Let $d(w)$ be the label found by the algorithm and let $\delta(w)$ be the widest path distance from u-to-w. We want to show that $d(w) = \delta(w)$ for every vertex w at the end of the algorithm, showing that the algorithm correctly computes the maximum bottleneck path flow between u and v. We prove this by induction on $|R|$ via the following lemma:

Lemma: For each $x \in R$, $d(x) = \delta(x)$.

Proof (by Induction):

Base case ($|R| = 1$): Since R only grows in size, the only time $|R| = 1$ is when $R = \{u\}$ and $d(u) = \infty = \delta(u)$, which is correct.

Inductive hypothesis: Let w be the last vertex added to R. Let $R' = R \setminus \{w\}$. Our I.H. is: for each $x \in R'$, $d(x) = \delta(x)$.

Using the I.H.: By the inductive hypothesis, for every vertex in R', we have the correct max flow label. We need only show that $d(w) = \delta(w)$ to complete the proof.

Suppose for a contradiction that the widest path from u-to-w is Q and has max flow value $l(Q) > d(w)$. Now, Q starts in R' and at some point leaves R' (to get to w which is not in R'). Let xy be the first edge along Q that leaves R'. Let Q_x be the u-to-x subpath of Q. Clearly: $\min\{l(Q_x), f_{xy}\} \geq l(Q)$.

Since $d(x)$ is the length of the widest u-to-x path by the I.H., $d(x) \geq l(Q_x)$, giving us $\min\{d(x), f_{xy}\} \geq l(Q)$. Since y is adjacent to x, $d(y)$ must have been updated by the algorithm, so $d(y) \geq \min\{d(x), f_{xy}\}$. Finally, since w was picked by the algorithm, w must have the largest max flow label: $d(w) \geq d(y)$.

Combining these inequalities in reverse order gives us the contradiction that $d(w) > d(w)$. Therefore, no such shorter path Q must exist and so $d(w) = \delta(w)$. This lemma shows the algorithm is correct by "applying" the lemma for $R = V$.

□

3. **[20 marks]** Given a flow network $G = (V, E)$ and a max flow F, we want to find out if we can increase the max flow of the network by increasing the capacity of a *single* edge in the network. Further, if such an edge exists, we want to find one that has the maximum effect on the increase of the max flow.

   Give an algorithm to solve this problem. Prove the correctness and analyze the running time of your algorithm. For full marks, your algorithm must be more efficient than brute-force.

   Solution. [Wrong Solution] Given a maximum flow F in the graph, increasing the capacity of an edge (u, v) will only increase the maximum flow if there is a positive-capacity path from s to u and from v to t in the residual graph G_F. If this isn't the case, then there won't be an augmenting path in the residual graph after making the increase, and therefore the maximum flow will remain maximum.

Of the edges (u,v) that have this property, the maximum amount of extra flow that can be pushed from s to t after increasing the capacity of (u,v) is bounded. To push any flow across this edge, we'll have to find a path from s to u and a path from v to t. When doing so, there will always be a bottleneck edge in each of the two paths, and the flow can't increase by more than the minimum of these two bottlenecks. Accordingly, one option for solving the problem is to do the following:

Algorithm:

```
FindSingleEdgeOfMaxEffect($G, F, s, t$):
    Let $G_F$ be the residual graph obtained from $G$ due to the max flow $F$
    Find a maximum-bottleneck path from $s$ to every other node reachable from $s$ in $G_F$
    Find a maximum-bottleneck path to $t$ from each node that can reach $t$ in $G_F$
    For each edge $(u,v)$ crossing the min cut, the maximum amount of extra flow that can be
        pushed across the edge is given by the minimum of a max-bottleneck path from $s$ to $u$
        and a max-bottleneck path from $v$ to $t$
    Return an edge with the maximum effect on the max flow.
```

Runtime: Since we run MaxBottleneckPath from two sources, the runtime of the above algorithm is $\mathcal{O}(|E|\log|V| + |E|) = \mathcal{O}(|E|\log|V|)$ steps.

□

4. **[20 marks]**

(a) **[8 marks]**
Suppose we want to compute a shortest path from node s to node t in a *directed* graph $G = (V,E)$ with integer edge weights $\ell_e > 0$ for each $e \in E$.
Show that this is equivalent to finding a *pseudo-flow* f from s to t in G such that $|f| = 1$ and $\sum_{e\in E} \ell_e f(e)$ is minimized. There are no capacity constraints.
Part of this problem requires you to **define** precisely what we mean by "pseudo-flow" in a general, directed graph. This is a natural extension of the notion of flow in a network.

(b) **[12 marks]**
Write the shortest path problem as a linear or integer program **where your objective function is *minimized*, based on your answer to the previous part**. Give a detailed justification that your solution is correct.

Solution. (a) For any directed graph $G = (V,E)$ with integer edge weights $\ell_e > 0$ (for $e \in E$) and any two vertices $s \neq t \in V$, a *pseudo-flow from s to t* is a function $f : E \to \mathbb{N}$ such that for every vertex $v \in V - \{s,t\}$, $f^{\text{in}}(v) = f^{\text{out}}(v)$ (where $f^{\text{in}}(v)$ and $f^{\text{out}}(v)$ are defined as for "normal" flows in networks). For any pseudo-flow f from s to t in a directed graph G, the *value* of f is defined as: $|f| = f^{\text{out}}(s)$.

Every pseudo-flow f from s to t with $|f| = 1$ corresponds to a path in G from s to t: $|f| = 1$ means $f^{\text{out}}(s) = 1$, which means there is exactly one edge (s, v_1) with $f(s,v_1) = 1$ and $f(s,u) = 0$ for all other edges (s,u) (where $u \neq v_1$). The same reasoning applied inductively shows that there is some sequence of vertices $v_1, v_2, \ldots, v_k$ such that $f(s,v_1) = f(v_1,v_2) = \cdots = f(v_k,t) = 1$ but $f(e) = 0$ for every other edge e.

Similarly, every path $s, v_1, \ldots, v_k, t$ in G corresponds to a pseudo-flow f from s to t with $|f| = 1$ by setting $f(e) = 1$ for every edge e on the path and $f(e) = 0$ for every edge e not on the path.

Hence, finding a shortest path in G from s to t is equivalent to finding a pseudo-flow f with $|f| = 1$ such that $\sum_{e\in E} \ell_e f(e)$ is minimized.

(b) **Variables:** $x_{u,v} \in \mathbb{Z}$ for every edge $(u,v) \in E$ ($x_{u,v}$ is meant to be equal to $f(u,v)$ for some pseudo-flow f from s to t).

Objective Function: minimize $\sum_{(u,v)\in E} \ell_{u,v} x_{u,v}$ (by the answer to the previous part, this is equivalent to finding a shortest path).

Constraints:

- $x_{u,v} \geq 0$ for all $(u,v) \in E$ (pseudo-flow values cannot be negative);
- $\sum_{(u,v)\in E} x_{u,v} = \sum_{(v,u)\in E} x_{v,u}$ for all $v \in V - \{s,t\}$ (pseudo-flow is conserved);
- $\sum_{(s,v)\in E} x_{s,v} = 1$ (looking for $|f| = 1$).

By the reasoning from the previous part, we already know that paths from s to t in G are equivalent to pseudo-flows f from s to t with $|f| = 1$.

Feasible solutions to the integer program are also equivalent to pseudo-flows f from s to t with $|f| = 1$: the linear constraints correspond exactly to the properties satisfied by integer pseudo-flow values through the correspondence $x_{u,v} = f(u,v)$.

Hence, solutions to the integer program are equivalent to paths from s to t where the value of the objective function is equal to the total weight of the path. Any solution that minimizes the objective function is therefore equivalent to finding a shortest path.

□

5. **[20 marks]**

(a) **[10 marks] Traveling Salesman Problem (TSP)**: Given a directed graph $G = (V,E)$ with an integer weight $w(e)$ for each edge $e \in E$, find a simple cycle over all the vertices (a "circuit") with minimum total weight. (Note that the weights $w(e)$ can be positive or negative.)

Show how to represent an arbitrary instance of the TSP as an integer program. Justify that your representation is correct, and describe how to obtain a solution to the instance of the TSP from solutions to your integer program.

(b) **[10 marks] Maximum Bipartite Matching Problem**: Given an undirected bipartite graph $G = (V_1, V_2, E)$, where $E \subseteq V_1 \times V_2$, find a set of disjoint edges of the maximum size (where two edges are *disjoint* if they have no common endpoint).

Give a linear (or integer) program that corresponds to this problem. Describe clearly every component of your answer. Then justify the correctness of your linear program: explain clearly what each variable and constraint represents and how solutions to each problem correspond to each other (and what that tells you about the relative values of those solutions).

Solution. (a) The following integer program represents an arbitrary instance of TSP.

Variables: one variable x_e for each edge $e \in E$

Objective function: minimize $\sum_{e\in E} w(e) \cdot x_e$

Constraints:

- $0 \leq x_e \leq 1$ for each x_e
- $\sum_{(u,v)\in E} x_{(u,v)} = 1$ for each vertex $v \in V$
- $\sum_{(v,u)\in E} x_{(v,u)} = 1$ for each vertex $v \in V$
- $\sum_{u\in V_1, v\in V_2, (u,v)\in E} x_{(u,v)} \geq 2$ for each nontrivial partition of vertices into V_1, V_2 (nontrivial means V_1 and V_2 are both non-empty)

The first constraint ensures that each edge e is either selected ($x_e = 1$) or not ($x_e = 0$)—remember that this is an *integer* program so variables cannot take on fractional values.

The second and third constraints ensure that each vertex has exactly one edge going in and one going out, which is necessary for having a simple cycle through every vertex.

The last constraint ensures that the edges selected cannot form a collection of disjoint cycles (a possibility not ruled out by the second and third constraints), by requiring that there are at least 2 edges crossing every possible nontrivial partition of the vertices.

Any solution $x^*_{e_1}, \dots, x^*_{e_m}$ to the integer program above yields a circuit (because of the constraints) with minimum total weight (by our choice of objective function).

(b) Let $V_1 = \{u_1, \dots, u_k\}$ and $V_2 = \{w_1, \dots, w_\ell\}$.

Variables: $x_{i,j}$ for all i, j with $(u_i, w_j) \in E$ — *intention: matching* $M = \left\{(u_i, w_j) : x_{i,j} = 1\right\}$

Objective Function: maximize $\sum_{(u_i,w_j)\in E} x_{i,j}$ — *equal to* $|M|$

Constraints: $x_{i,j} \in \{0, 1\}$, for all $(u_i, w_j) \in E$

$$\left.\begin{array}{l} \sum_{w_j \in V_2 : (u_i, w_j) \in E} x_{i,j} \le 1, \text{ for each } u_i \in V_1 \\ \sum_{u_i \in V_1 : (u_i, w_j) \in E} x_{i,j} \le 1, \text{ for each } w_j \in V_2 \end{array}\right\} \textit{no two edges in } M \textit{ have a common endpoint}$$

Every matching M over G gives rise to a feasible solution by setting $x_{i,j} = 1$ iff $(u_i, w_j) \in M$: since no two edges share an endpoint, every constraint is satisfied. Moreover, the size of M is equal to the value of the objective function, so the maximum value of the objective function is at least as large as the size of the maximum matching.

Every feasible solution to the integer program yields a matching M by selecting every edge (u_i, w_j) such that $x_{i,j} = 1$: the constraints guarantee that no two edges share an endpoint and the value of the objective is equal to the size of M. So, the size of the maximum matching is at least as large as the maximum value of the objective function.

□