**CS 344: Design and Analysis of Computer Algorithms** **Rutgers: Fall 2019**

# Practice Midterm Exam #2

*Name:* ______________________________ *NetID:* ______________

## Instructions

1. Do not forget to write your name and NetID above.
2. The exam contains 4 problems worth 100 points in total *plus* one extra credit problem worth 10 points. You have 75 minutes to finish the exam. The exam is closed-book and closed notes.
3. **Note that problems appear on both odd and even numbered pages.** There should be more than enough space to write down your solution for each problem below the problem itself. But if you ran out of space, you can also use the extra sheet at the end of the exam; if you do so, be clear about which problem you are solving.
4. Remember that you can leave a problem (or parts of it) entirely blank and receive 25% of the grade for that problem (or part). However, this should not discourage you from attempting a problem if you think you know how to approach it as you will receive partial credit more than 25% if you are on the right track. But keep in mind that if you simply do not know the answer, writing a very wrong answer may lead to 0% credit.

   The only **exception** to this rule is the extra credit problem: you do not get any credit for leaving the extra credit problem blank, and it is harder to get partial credit on that problem.
5. **You should always prove the correctness of your algorithm and analyze its runtime.** Also, as a general rule, avoid using complicated pseudo-code and instead explain your algorithm in English.
6. You may use any algorithm presented in the class as a building block for your solutions.

**Suggestion:** Leave the extra credit problem for last as it is harder than the rest and worths fewer points.

| Problem. # | Points | Score |
|---|---|---|
| 1 | 25 | |
| 2 | 25 | |
| 3 | 25 | |
| 4 | 25 | |
| 5 | +10 | |
| Total | $100 + 10$ | |

**Problem 1.** Prove or disprove the following assertions.

(a) Suppose $G(V, E)$ is a directed acyclic graph (DAG) with *two* sources $s_1, s_2$ and *two* sinks $t_1, t_2$. If we add a directed edge from any arbitrary sink to any arbitrary source, then the new graph will always contain a cycle. **(5 points)**

**Solution.** The assertion is **false**. Consider the graph below with only (black) edges $(s_1, t_1)$ and $(s_2, t_2)$ which clearly is a DAG with two sources and two sinks. If we add the directed (red) edge from $t_1$ to $s_2$, we do not create any cycle.

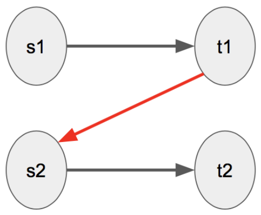

Figure 1: A counter-example to the assertion.

(b) Suppose $G(V, E)$ is an undirected graph and $(S, V - S)$ is a cut with no cut edges in $G$. Suppose we add an edge $e = \{u, v\}$ for $u \in S$ and $v \in V - S$ to $G$; then edge $e$ *cannot* belong to any cycle in $G$.

**(10 points)**

**Solution.** The assertion is **true**. Suppose by contradiction that the cut $(S, V - S)$ has no cut edge and yet when we add $e = \{u, v\}$, $e$ become part of a cycle. For $e$ to be part of a cycle, it means that there exists a path already in $G$ between $u$ and $v$. Let $P = u, w_1, w_2, \ldots, w_k, v$ be this path. Since $u \in S$ and $v \in V - S$, we should have at least one $i \in \{1, \ldots, k\}$ such that $w_i \in S$ and $w_{i+1} \in V - S$ (otherwise starting from $u \in S$ we never reach a vertex in $V - S$, in particular, vertex $v$). But this means that there is an edge $\{w_i, w_{i+1}\}$ in $G$ which is a contradiction with the assumption that the cut $(S, V - S)$ has no cut edges.

(c) Suppose $G(V, E)$ is an undirected connected graph with *distinct* weight $w_e$ for each edge $e \in E$, i.e., $w_e \neq w_{e'}$ for any two edges $e \neq e' \in E$. Then $G$ has a *unique* minimum spanning tree (MST).

**(10 points)**

**Solution.** The assertion is **true**. Suppose by contradiction that there are two different MSTs $T$ and $T'$ for $G$. We sort the edges in both $T$ and $T'$ in increasing order of their weight. Let $e_i$ and $e'_i$ be the first edges that are different between $T$ and $T'$, i.e., $e_1 = e'_1, \ldots, e_{i-1} = e'_{i-1}$ but $e_i \neq e'_i$ – such an edge should exist otherwise $T = T'$.

By symmetry, let us assume without loss of generality that $w_{e_i} < w_{e'_i}$. We insert edge $e_i$ to $T'$ to get the graph $T' \cup \{e_i\}$. There is a cycle in this graph and moreover this cycle cannot be only consisting of edges in $e'_1, \ldots, e'_{i-1}$ and $e_i$ (otherwise, since these edges are equal to $e_1, \ldots, e_{i-1}$ and $e_i$, we will have the same cycle in $T$ also, but $T$ is a tree).

Let $e'$ be any edge in this cycle other than $\left\{e'_1, \ldots, e'_{i-1}, e_i\right\}$ and consider $T' \cup \{e_i\} - \{e'\}$. Since we removed an edge from a cycle, $T' \cup \{e_i\} - \{e'\}$ remains connected and since the new graph has $n - 1$ edges (since $T'$ has $n - 1$ edges), we obtain that it is a tree. Finally, since $w_{e_i} < w_{e'}$ as $w_{e_i} < w_{e'_i}$ and $w_{e'_i} \leq w_{e'}$ (as $e'$ can only be $e'_i$ or some $e'_j$ for $j > i$), we obtain that weight of this new tree is smaller than the weight of $T'$, a contradiction with $T'$ being a MST.

**Problem 2.** You are given a set of $n$ (closed) intervals on a line:

$$[a_1, b_1], [a_2, b_2], ..., [a_n, b_n].$$

Design an $O(n \log n)$ time *greedy* algorithm to select the *minimum* number of intervals whose union is the same as the union of all intervals. You may assume that the union of intervals is equal to $[\min_i a_i, \max_j b_j]$.

*Example:* If the following 4 intervals are given to you:

$$[2, 5], [3, 9], [2.5, 9.5], [4, 8],$$

then a correct answer is: $[2, 5], [2.5, 9.5]$. **(25 points)**

**Solution.** We give the following algorithm and analyze it using an exchange argument.

*Algorithm:*

1. Sort the intervals based on their *starting* point and let $A[1 : n]$ denote the intervals in the sorted order.
2. Let $d = A[1].start$.
3. While any interval is left:
   (a) Iterate over the intervals in the sorted order and find the interval $i$ with maximum $A[i].end$ among all intervals where $A[i].first \leq d$ and $A[i].end > d$.
   (b) Add $i$ to the solution and let $d = A[i].end$. Continue to the next interval that was not checked in the line above.

*Proof of Correctness:* Let $G = \{g_1, \ldots, g_k\}$ be the output of the greedy algorithm and $O = \{o_1, \ldots, o_\ell\}$ be an optimal solution (both sorted in increasing order of their starting points). By definition of optimal solution $\ell \leq k$ and we are going to prove that in fact $\ell = k$.

Find the first index $j \in \{1, \ldots, k\}$ where $g_1 = o_1, \ldots, g_{j-1} = o_{j-1}$, but $g_j \neq o_j$. If $\ell \neq k$, such an index $j \leq k$ should exists as otherwise the union of first $j - 1$ intervals already equals the entire intervals and in particular means $d = \max_i b_i$ when we have added $g_{j-1}$ to the greedy solution – this means that we would have never added any new interval after $g_{j-1}$ in $G$ by definition of the algorithm.

Now consider the solution $O' = \{g_1, \ldots, g_{j-1}, g_j, o_{j+1}, \ldots, o_\ell\}$. We claim that $O'$ also has the same union as union of all intervals. Consider the choice of $d$ before we pick $g_{j+1}$ in the greedy. By construction, all intervals with end point before $d$ are already covered in both $G$ and $O$ and no interval with endpoint after $d$ is covered yet. Greedy picks the interval $g_{j+1}$ with the largest endpoint among such intervals. Moreover $o_j.start$ needs to also be at most equal to $d$ as otherwise $O$ will not cover the gap between $d$ and $o_j.start$ (remember that union of all intervals is $[\min_i a_i, \max_j b_j]$). As such, $g_j.end \geq o_j.end$ and hence if $O$ could cover $[d, o_j.end]$ and cover the entire line, $O'$ covers even more than $[d, g_j.end] \supseteq [d, o_j.end]$ and thus has the same union as union of all intervals. This means that $O'$ is also another optimal solution for the problem.

We are now done since we can keep exchanging all elements of $O$ with $G$ one by one and obtain that $G$ is also another optimal solution.

*Runtime Analysis:* The algorithm involves sorting the intervals in $O(n \log n)$ (say, by merge sort), and running over the intervals one by one in $O(n)$ time. Hence, total runtime is $O(n \log n)$.

**Problem 3.** Suppose you are given a directed acyclic graph (DAG) $G(V, E)$, a source vertex $s$, and a sink vertex $t$ (these two are *not* the only sources or sinks in $G$). We say that a vertex $v \in V - \{s, t\}$ is an $(s, t)$*-independent vertex* in $G$ if there is *no* path in $G$ from $s$ to $t$ that passes through $v$. Design an $O(n+m)$ time algorithm for finding all $(s, t)$-independent vertices in $G$.

**(25 points)**

**Solution.**

*Algorithm:*

1. Run a graph search (DFS/BFS) starting from vertex $s$ to find all vertices reachable from $s$ in $G$ denoted by $S$. Define $\overline{S} := V - S$.
2. Create a new graph $G'$ by reversing the direction of every edge in $G$. Run a graph search (DFS/BFS) starting from vertex $t$ to find all vertices reachable from $t$ in $G'$ denoted by $T$. Define $\overline{T} := V - T$.
3. Return $\overline{S} \cup \overline{T}$.

*Proof of Correctness:* First, we claim that a vertex $v$ is $(s, t)$-independent if and only if $v$ is either not reachable from $s$ or $v$ cannot reach $t$ (or both):

- If $v$ is $(s, t)$-independent then $v$ is not reachable from $s$ or $v$ cannot reach $t$. Suppose by contradiction that this is not the case and hence both $s$ can reach $v$, say via path $P_{sv}$, and $v$ can reach $t$, say via path $P_{vt}$. But then since $G$ is a DAG, we also now that vertices in $P_{sv}$ and $P_{vt}$ are disjoint except for $v$ since all vertices in $P_{sv} - \{v\}$ have a topological order less than $v$ and all vertices in $P_{vt} - \{v\}$ have a topological order more than $v$. This means that if we go from $s$ to $v$ and then from $v$ to $t$ using paths $P_{sv}$ and $P_{vt}$, we will obtain a $s$-$t$ path that goes through $v$, a contradiction.
- If $v$ is either not reachable from $s$ or $v$ cannot reach $t$ (or both) then $v$ is $(s, t)$-independent: either $s$ cannot reach $v$ or $v$ cannot reach $t$ – either way, $v$ cannot be on any path from $s$ to $t$, since otherwise we could take the $s$-$v$ part and $v$-$t$ part of those paths and so $v$ become both reachable from $s$ and also able to reach $t$, a contradiction.

Now note that the set $\overline{S}$ is the set of vertices not reachable by $s$ by definition of graph search. Also, note that vertex $t$ can reach any vertex $v$ in $G'$ if and only if $v$ can reach $t$ in $G$ (as we reversed the direction of edges). So $\overline{T}$ is the set of vertices that cannot reach $t$ in $G$ by definition of graph search. Hence, $\overline{S} \cup \overline{T}$ is precisely the set of vertices that are either not reachable from $s$ or cannot reach $t$ (or both). By the first part of the argument, this is exactly the set of $(s, t)$-independent vertices.

*Runtime analysis:* First line takes $O(n+m)$ time by the runtime of DFS/BFS. Second line involves reversing direction of all edges in $O(n+m)$ time plus another DFS/BFS in $O(n+m)$ time. The total runtime is thus $O(n+m)$ as desired.

**Problem 4.** Consider the following different (and less efficient) algorithm for computing an MST of a given undirected connected graph $G(V, E)$ with edge weight $w_e$ on each $e \in E$:

1. Sort the edges in decreasing (non-increasing) order of their weights.
2. Let $H = G$ be a copy of the graph $G$.
3. For $i = 1$ to $m$ (in the sorted ordering of edges):
   (a) If removing $e_i$ from $H$ does not make $H$ disconnected, remove $e_i$ from $H$.
4. Return $H$ as a minimum spanning tree of $G$.

Our goal in this question is to prove the correctness of this algorithm, i.e., that it outputs an MST of any given graph $G$ (we ignore the runtime of this algorithm in this problem).

(a) Prove that in any graph $G(V, E)$, if an edge $e \in E$ has the largest weight among edges of some cycle in $G$, then there exists an MST of $G$ that does *not* contain the edge $e$. **(12.5 points)**

**Solution.** Fix any MST $T$ of $G$. If $e \notin T$ we are already done. Otherwise consider $T \setminus \{e\}$. A tree minus an edge consists of two separate connected components $S, V - S$. Let $e = \{u, v\}$ and since $e$ is maximum weight edge of some cycle, we know there is another path $P$ from $u$ to $v$ in $G$ so that for all $e' \in P, w_{e'} \leq w_e$.

At least one edge of path $P$ should cross the cut $(S, V - S)$ as $u \in S$ and $v \in V - S$ (or vice versa). Let that edge be $e'$ (breaking the ties arbitrarily). Consider the graph $T' = T \cup \{e'\} - \{e\}$. Firstly, $w_{e'} \leq w_e$ as stated earlier and thus the total weight of $T'$ is at most equal to the weight of $T$. Moreover, $T'$ is connected since we connected two connected components $S$, $V - S$ by an edge $e'$. Since $T'$ also has $n - 1$ edges (as $T$ had $n - 1$ edges) and is thus a tree, we have that $T'$ is another MST of $G$. But $T'$ does not have the edge $e$ as desired.

(b) Use Part (a) to argue the correctness of the above algorithm. **(12.5 points)**

**Solution.** Firstly, the algorithm always outputs a tree. This is because the only edges that are not removed from $G$ are the ones that removing them disconnects $G$. We now argue that at every step of the algorithm, $H$ is a superset of some MST of $G$. Since at the end, $H$ is itself a tree, we obtain that $H$ should be a MST.

We prove this by induction over index $i$. At the beginning of the algorithm, $H$ clearly is a superset of some MST of $G$ because $H = G$ (induction base). Suppose $H$ is superset of some MST $T$ of $G$ at the end of iteration $i$ and we prove that this is the case at the end of iteration $i + 1$ also (induction step).

If removing $e_{i+1}$ in this iteration makes $H$ disconnected, we will simply not remove it. Hence $H$ remains the same after this step and by induction hypothesis still contains an MST of $G$. Otherwise, if removing $e_{i+1}$ does not make $H$ disconnected, we know that there should be a cycle $C$ in $H$ containing this edge. Moreover, this cycle $C$ cannot contain any of the edges $e_j$ for $j < i + 1$ that are still in $H$ as keeping $e_j$ meant that removing it would make $H$ disconnected which means $e_j$ is not part of any cycle in $H$. As we sorted the edges in decreasing (non-increasing) order of their weights, we know $e_{i+1}$ has the largest weight among the edges of the cycle $C$. By Part (a), we know that there is an MST $T$ of $H$ (before deleting $e_{i+1}$) that does not use the edge $e_{i+1}$ and by induction hypothesis, $T$ is also an MST of $G$. As such, $T$ remains a subset of $H - e_{i+1}$ as well after this step, proving the induction step and so $H$ remains a superset of some MST of $G$.

This concludes the proof of correctness of the algorithm.

**Problem 5.** [**Extra credit**] You are given a set of $n$ cities with $R$ roads and $H$ highways between different cities, a starting city $s$, and a destination city $t$. Your goal is to go from city $s$ to city $t$ by using the minimum number of roads. You are also allowed to take any highway but that requires paying a toll and you only have enough money to pay at most one toll. Design an $O(n+R+H)$ time algorithm that finds the set of roads and the single highway (if any) you should take. You may assume that there is always a way to get from $s$ to $t$, all roads and highways are one-way, and the best solution always involve taking a highway.

**(+10 points)**.

**Solution.** *Algorithm/Reduction:* We prove this using reduction to the problem of finding an unweighted shortest path (which we can solve using BFS). The reduction consists of two steps.

Step (1): Create a graph $G(V,E)$ with $n$ vertices, one per each city, and $R+H$ edges by adding an edge $(i,j)$ if there is a road or a highway from city $i$ to city $j$. We will call the edges added per roads $R$-edges and edges added per highways as $H$-edges. At this point, the problem is equivalent to finding a shortest (unweighted) path from $s$ to $t$ that uses at most (exactly) one $H$-edge.

Step (2): We cannot readily use BFS for solving the above problem because BFS may use multiple $H$-edges. To fix this, we create a new graph $G'(V',E')$ as follows (see figure below):

- $V'$ contains two disjoint copies of vertices of $V$, denoted by $V_1 = V$ and $V_2 = V$. We use $s_1$ to denote the copy of $s$ in $V_1$ and $t_2$ to denote the copy of $t$ in $V_2$.
- We connect every vertex $u \in V_1$ (respectively, $w \in V_2$) to any other vertex $v \in V_1$ (respectively, $z \in V_2$) if and only if there is an $R$-edge $(u,v)$ (respectively, $(w,z)$) in $G$. We will then connect every vertex $u \in V_1$ to any vertex $v \in V_2$ if there is an $H$-edge $(u,v)$ in $G$.

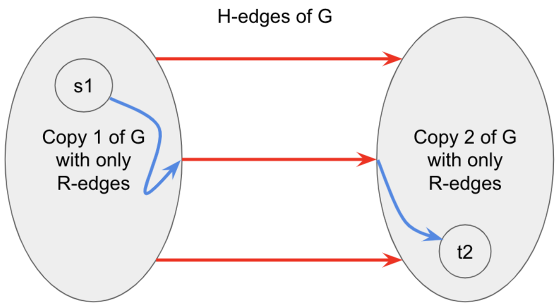

Figure 2: An illustration of the graph $G'$. Blue edges denote the shortest path from $s_1$ to $t_2$.

After creating the graph, we simply find the shortest path from $s_1$ to $t_2$ in $G'$ (using BFS) and return the name of vertices along the path as the answer to problem.

*Proof of Correctness:* Firstly, it is easy to see there is a one-to-one correspondence between paths in $G$ and travel options in the original problem, hence finding shortest $s$-$t$ path in $G$ with one $H$-edge is equivalent to solving the problem. The more interesting case is step (2) and graph $G'$. Any $s$-$t$ path $P$ in $G$ that uses only one $H$-edge corresponds to a $s_1$-$t_2$path in $G'$ by going in the first copy until we hit this $H$-edge and then continuing from the endpoint of this $H$-edge in the second copy. Similarly, any $s_1$-$t_2$ *shortest* path $P'$ in $G$ corresponds to a $s$-$t$ path in $G$ that uses only $H$-edge by simply mapping the copies of vertices in $G'$ to their original name in $G$; since $P'$ is a shortest path, we know that two copies of the same vertex are not used in $P'$ as otherwise we could have made it shorter by removing the part between the two copies (using the assumption that shortest paths contain one $H$-edge). Hence this mapping indeed creates a path in $G$ (and not a walk). This proves the correctness of the reduction.

*Runtime Analysis:* We create a graph with $\Theta(n)$ vertices and $\Theta(R+H)$ edges and run BFS over it which takes $O(n+R+H)$ time.

## Extra Workspace