Only for the personal use of students registered in CS 344, Spring 2022 at Rutgers University. Redistribution out of this class is strictly prohibited.

CS 344: Design and Analysis of Computer Algorithms **Rutgers: Spring 2022**

Practice Midterm Exam #2 Solutions

Instructions

1. Do not forget to write your name and NetID above, and to sign Rutgers honor pledge below.
2. The exam contains 4 problems worth 100 points in total *plus* one extra credit problem worth 10 points.
3. The exam should be done **individually** and you are not allowed to discuss these questions with anyone else. This includes asking any questions or clarifications regarding the exam from other students or posting them publicly on Piazza (**any inquiry should be posted privately on Piazza**). You may however consult all the materials used in this course (video lectures, notes, textbook, etc.) while writing your solution, but **no other resources are allowed**.
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 there is almost no 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 or homeworks as a building block for your solutions.

Rutgers honor pledge:

On my honor, I have neither received nor given any unauthorized assistance on this examination.

Signature:_____

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

Only for the personal use of students registered in CS 344, Spring 2022 at Rutgers University. Redistribution out of this class is strictly prohibited.

Problem 1.

(a) Suppose $G = (V, E)$ is a directed graph with positive weight w_e on each edge e. Let

$$P = s, v_1, v_2, \ldots, v_k, t$$

be a shortest path from a vertex s to a vertex t in G. Prove that for every $i \in \{1, \ldots, k\}$, the path

$$P_i = s, v_1, \ldots, v_i$$

is also a shortest path from s to v_i. **(12.5 points)**

Solution. We prove this by contradiction. Suppose there exists an index $i \in \{1, \ldots, k\}$ so that P_i is not the shortest path from s to v_i. Let us call this path $Q_i = s, u_1, u_2, \ldots, v_i$. Thus, weight of Q is strictly less than that of P_i.

Now consider the *walk* P' from s to t obtained by replacing the first part of s to v_i in P by the path Q instead, i.e.,

$$P' = \underbrace{s, u_1, u_2, \ldots, v_i}_{Q}, v_{i+1}, \ldots, t, \quad \text{vs.} \quad P = \underbrace{s, v_1, v_2, \ldots, v_i}_{P_i}, v_{i+1}, \ldots, t.$$

Note that this is a walk and not necessarily a path because some vertices in Q may have been repeated in the remainder of the path from v_i to t in P.

The total weight of the edges in the walk P' is strictly less than that of P because weight of Q is strictly less than that of P_i and the remainder of P' and P are the same. Thus, we found a walk with smaller weight from s to t with weight strictly less than the weight of the shortest path from s to t (as P was supposed to be a shortest path). This is a contradiction because we can turn P' to an actual path from s to t by "shortcutting" the repeated parts of the walk i.e., in P' if a vertex w appears both as $u_i \in Q$ and $v_j \in P - Q$, we just connect u_{i-1} directly to v_j in the new path. But this means that we found a path from s to t with smaller weight than the shortest path P, a contradiction.

This means that our assumption was wrong and for every index $i \in \{1, \ldots, k\}$ the path P_i is the shortest path from s to v_i.

Only for the personal use of students registered in CS 344, Spring 2022 at Rutgers University. Redistribution out of this class is strictly prohibited.

(b) Suppose $G = (V, E)$ is an undirected connected graph with positive weight w_e on each edge e. Prove that if the weight of some edge f is *strictly smaller* than weight of all other edges in G, then *every* minimum spanning tree (MST) of G contains the edge f.

(Note that to prove this statement, it is *not* enough to say that some specific algorithm for MST always picks this edge f; you have to prove *all* MSTs of G contain this edge). **(12.5 points)**

Solution. Fix any arbitrary MST T of G and suppose by contradiction that f is not part of T. Consider the subgraph $T + f$ obtained by adding the edge f to T. Since T is a tree, this subgraph will have one cycle containing the edge f. Let e be the heaviest weight of this cycle and note that by the promise of the problem, $w_f < w_e$.

Now consider the subgraph $T + f - e$. This subgraph has $n - 1$ edges and is connected, as $T + f$ was connected and we removed an edge from the cycle of this subgraph (which does not change connectivity). As such, $T + f - e$ is a spanning tree. But since $w_f < w_e$, weight of this spanning tree is strictly smaller than the weight of T, a contradiction with T being a MST of G that does not contain the edge f.

Only for the personal use of students registered in CS 344, Spring 2022 at Rutgers University. Redistribution out of this class is strictly prohibited.

Problem 2. Recall the job scheduling problem from the class: we have a collection of n processing jobs and the length of job i, i.e., the time to process job i, is given by $L[i]$. This time, you are given a number M and you are told that you should finish all your processing jobs between time 0 and M; any job not fully processed in this window then should be paid a penalty that is the same across all the jobs. The goal is to find a schedule of the jobs that minimizes the penalty you have to pay, i.e., it minimizes the number of jobs not fully processed in the given window.

Design a greedy algorithm that given the array $L[1:n]$ of job lengths and integer M, finds the scheduling that minimizes the penalty in $O(n \log n)$ time. **(25 points)**

A complete solution consists of three part: the algorithm (**10 points**), the proof of correctness (**10 points**), and the runtime analysis (**5 points**).

Solution. *Algorithm:*

1. Sort the job in increasing order of their length using merge sort.
2. Output this ordering as the schedule.

Proof of Correctness: An obvious observation is that minimizing the penalty is equivalent to maximizing the number jobs that are fully processed. So we focus on the second task instead. We now prove the correctness of the algorithm using an exchange argument (there are multiple ways to do an exchange argument for this problem; we simply pick one of them here).

Let $G = \{g_1, g_2, \ldots, g_k\}$ denote the indices of the jobs fully processed by the greedy algorithm, sorted in increasing (non-decreasing) order of their length. Let $O = \{o_1, o_2, \ldots, o_\ell\}$ denote the indices of the jobs fully processed in *some optimal* solution, again sorted in increasing (non-decreasing) order of their length (note that we cannot assume $k = \ell$; this is in fact precisely what we want to prove in the first place).

We now show how to exchange O to G. Let j be the first index where O and G differ, i.e., $g_1 = o_1, \ldots, g_{j-1} = o_{j-1}$ but $g_j \neq o_j$. Since both G and O are sorted in increasing order of their job-lengths, and by definition of the greedy, we know that $L[g_j] \leq L[o_j]$. We replace the order of processing jobs g_j and o_j in O to get a new solution. Now if we denote $O' = \{g_1, \ldots, g_j, o_{j+1}, \ldots, o_\ell\}$ in this new solution, all these jobs will be fully processed because $L[g_j] \leq L[o_j]$ and thus we have

$$\sum_{i=1}^{\ell} L[o'_i] \leq \sum_{i=1}^{\ell} L[o_i] \leq M.$$

Since size of O' is equal to size of O, it is also optimal.

We can thus continue doing the exchange with index $j+1$ and so on until we entirely exchanged O to G, and thus proving that G is also optimal.

Runtime analysis: The algorithm needs $O(n \log n)$ time to sort the input using merge sort.

Only for the personal use of students registered in CS 344, Spring 2022 at Rutgers University. Redistribution out of this class is strictly prohibited.

Problem 3. Crazy City consists of n houses and m bidirectional streets connecting these houses together, and there is always at least one way to go from any house to another one following these streets. For every street e in this city, the cost of maintaining this street is some positive integer $c_e > 0$. The mayor of Crazy City has come up with a brilliant cost saving plan: destroy(!) as many as the streets possible to maximize the cost of destroyed streets (so we no longer have to pay for their maintenance) while only ensuring that there is still a way for every house to reach mayor's house following the remaining streets.

Design an $O(m \log m)$ time algorithm that outputs the set of streets with *maximum total cost* that should be destroyed by the mayor. **(25 points)**

A complete solution consists of three part: the algorithm or reduction (**10 points**), the proof of correctness (**10 points**), and the runtime analysis (**5 points**).

Solution. *Algorithm:*

- Create a graph $G = (V, E)$ where V is the set of n vertices and E is the set of bidirectional streets. Let weight w_e of each edge e be equal to the cost c_e.
- Compute an MST T of the graph G using Kruskal's or Prim's algorithm.
- Output all edges of G that are not in T, i.e., $G - T$ as the solution.

Proof of correctness: Consider the graph G. Destroying a subset of streets while only ensuring that there is still a way for every house to reach mayor's house following the remaining streets, is equivalent to destroying as many edges in G as possible while keeping G connected (note that the only way for all houses to reach the Mayor's house is that the remaining streets are connected; in other words, all houses reaching mayor's house is equivalent to all houses reaching each other as well). This means that the remaining streets should form a spanning tree.

Next, note that *maximizing* the cost of destroyed streets is equivalent to *minimizing* the cost of *not* destroyed streets. Thus the goal is to simply pick a spanning tree with minimum cost. This is exactly the MST problem, hence the correctness of the algorithm.

Runtime Analysis: Creating the graph G takes $O(n + m)$ time. Running Kruskal's or Prim's algorithm on this graph also takes $O(m \log m)$ time. Also, since the graph is connected to begin with, we have $m \geq n - 1$ and thus $O(n + m) = O(m) = O(m \log m)$. So the total runtime is $O(m \log m)$.

Only for the personal use of students registered in CS 344, Spring 2022 at Rutgers University. Redistribution out of this class is strictly prohibited.

Problem 4. We have an undirected graph $G = (V, E)$ with positive weight $w_e > 0$ over each edge $e \in E$. We are additionally given two vertices $s, t \in V$ and are promised that at least one of the shortest (minimum weight) paths from s to t uses at most 10 edges. Design and analyze an algorithm that in $O(n + m)$ time finds a shortest path from s to t in G (the shortest path found by your algorithm does not necessarily need to be the one that uses at most 10 edges). **(25 points)**

A complete solution consists of three part: the algorithm or reduction (**10 points**), the proof of correctness (**10 points**), and the runtime analysis (**5 points**).

Solution. *Algorithm:* We simply run Bellman-Ford algorithm for 10 iterations of the for-loop. Formally,

1. Let $d[s] = 0$ and $d[v] = +\infty$ for $v \in V - \{s\}$.
2. For $i = 1$ to 10 times:
   - For every edge $e = (u, v) \in E$:
     (a) If $d[v] > d[u] + w_e$, update $d[v] \leftarrow d[u] + w_e$.
3. Run the algorithm for finding s-t path using the array of distances $d[]$ from Lecture 19.

Proof of correctness: In the class, we inductively proved that after iteration i of the for-loop, $d[v] \leq dist_i(s, v)$ where $dist_i(s, v)$ is the distance of s to v using paths with at most i edges.

Let $P = s, v_1, v_2, \ldots, t$ be the shortest path from s to t that uses at most 10 edges. By the above, after iteration 10, $d[w] \leq dist_{10}(s, w)$ for all vertices $w \in P$. But by the promise of the problem, $dist_{10}(s, w) = dist(s, w)$ for all these vertices as P is a shortest path itself (in the entire graph). Since in the Bellman-Ford algorithm $d[v] \geq dist(s, v)$ (see Lecture 19), we have that for all $w \in P$, $d[w] = dist(s, w)$.

Finally, the algorithm for finding the shortest path can find the path P, or some other shortest s-t path using the array d as $d[w]$ for $w \in P$ is computed correctly and so edges of path P can no longer be updated (and thus $d[v_{i+1}] = d[v_i] + w_{v_i v_{i+1}}$).

Runtime analysis: The first step of the algorithm takes $O(n)$ time to update array d. Then we have 10 iterations of a for-loop, each taking $O(m)$ time, so $O(n + m)$ time in total for the algorithm.

Only for the personal use of students registered in CS 344, Spring 2022 at Rutgers University. Redistribution out of this class is strictly prohibited.

Problem 5. [**Extra credit**] Design and analyze an algorithm that given an undirected graph $G = (V, E)$, in $O(n+m)$ time determines whether or not G is *bipartite*. Recall that a graph is bipartite if its vertices can be partitioned into two sets L and R such that all edges of the graph are between L and R. **(+10 points)**

Solution. We assume G is connected; otherwise, we can run this algorithm on each connected component of G individually.

Algorithm: Pick an arbitrary vertex $s \in V$. Run BFS (for shortest path) from s and define the layers $L_1, L_2, \ldots,$ of vertices where

$$L_i := \{v \in V \mid d[v] = dist(s, v) = i\}.$$

For each vertex v, stored which of these sets it belongs to.

Iterate over the edges E of G and for each edge $e = (u, v) \in E$, if both u and v belong to the same level L_i, output G is not bipartite and terminate. Otherwise, if the for-loop never terminated, output G is bipartite.

Proof of Correctness:

- Part one: If G is bipartite, then the algorithm also outputs bipartite. Assume toward the contradiction that the algorithm has output G is not bipartite. For this to happen, there should be an edge (u, v) such that $dist(s, u) = dist(s, v) = i$. Pick the shortest paths P^u and P^v from s to u and v, respectively. Let j be the last index where $P^u_j = P^v_j$ and suppose w is the j-th vertex of these paths.

  This means that there is a cycle C in G starting from w going to u, taking the edge (u, v), and then going back from v to w. Moreover, the length of this cycle is an odd number, in particular $2 \cdot (i-j)+1$. Recall that we assumed G is bipartite and without loss of generality we can assume $w \in L$. Then, the next vertex of the cycle C should be in R, the next one in L, and so on and so forth. But because C has an odd length, this forces w to also be in R, a contradiction. This means that G cannot be a bipartite graph if the algorithm outputs not bipartite.

- If the algorithm outputs bipartite, then G is also bipartite. Consider adding s to L, L_1 to R, L_2 to L, L_3 to R, and so on and so forth. We prove that all edges of G are between the sets L and R.

  Since $L_1, L_2, \ldots,$ are computed by BFS, we know that any other edge (u, v) in the graph should be either inside one layer, or from one layer to one after or before it (otherwise, there is a shorter path starting from s to one endpoint of the edge, a contradiction). Moreover, since we checked in the algorithm that there is no edge inside one layer, all edges of G are between one layer and the next. Thus, they are all between L and R by the construction above.

Runtime analysis: Running BFS takes $O(n + m)$ time. Marking the layer of each set also takes $O(n)$ time and checking the edges require another $O(m)$ time. So, in total, the runtime is $O(n + m)$.