Problem 1

$\frac{10}{10}$ (a)

Given a minimum cost circulation in a graph $G = (V, E)$ show how to compute node potentials $\pi(v)$, $v \in V$, such that all reduced costs with respect to π ($c^\pi(e)$) of residual edges are non-negative. What is the running time of your algorithm?

Let G_f be the residual network of the given min cost circulation. There is no negative cycle (With respect to the cost function) in G_f (According to a thm we learned in the residual graph of a min cost circulation there is no negative cycle). Let s be an arbitrary vertex in G_f. $\forall v \in V$ let $\delta(v)$ denote the shortest path from s to v in G_f (With respect to the cost function). We shall notice that $\delta(v)$ is well defined because there are no negative cycles in G_f. Let $e = (u, v)$ be an edge in G_f. Let p_v be the shortest path from s to v, and p_u the shortest path from s to u. We shall examine the path from s to v that starts with p_u and then continue with $e = (u, v)$. The cost of the path is $\delta(u) + c(e)$ and we know that the cost of this path is $\geq \delta(v)$ because $\delta(v)$ is the shortest path from s to v. Therefore: $c(e) + \delta(u) - \delta(v) \geq 0$. We shall define $\pi(v)$ to be $-\delta(v)$. Therefore for each $e \in E(G_f)$ we have that $c^\pi(e) = c(e) + \pi(v) - \pi(u) \geq 0$.

Because we can have negative weights in G_f we can't use dijkstra to compute $\delta(v)$. Instead we shall use Bellman-Ford which has a complexity of $O(nm)$. After running Bellman-Ford we have $\delta(v)$ and therefore we can compute $c^\pi(e)$ of every edge in G_f in $O(1)$ time per edge. Therefore the total running time is $O(nm)$.

(a)

(b)

Let $\pi(v)$, $v \in V$ be potentials such that there exists a feasible circulation f whose residual edges have non-negative reduced costs with respect to π. Given π show how to find a minimum cost circulation whose residual edges have non-negative reduced cost with respect to π. What is the running time of your algorithm ?

Problem 1 [(b)] continued on next page. . .

We shall notice some important observations:
1. Let $e = (u, v)$ be an edge such that $e \in E$ or $\overleftarrow{e} \in E$ ($\overleftarrow{e} = (v, u)$). Then $c^\pi(e) < 0 \rightarrow f(e) = u(e)$ (When u is the capacity function).
Reasoning: π is a potential function such that there exists a feasible circulation f whose residual edges have non-negative reduced cost. Therefore if the reduced cost of an edge is negative then it can't appear in G_f, therefore it must be saturated - $f(e) = u(e)$.
2. Let $e = (u, v)$ be an edge such that $e \in E$ or $\overleftarrow{e} \in E$ ($\overleftarrow{e} = (v, u)$). Then $c^\pi(e) > 0 \rightarrow f(e) = -f(\overleftarrow{e}) = -u(\overleftarrow{e})$.
Reasoning: If $c^\pi(e) > 0$ then $c^\pi(\overleftarrow{e}) < 0$ and if $f(e) \neq -u(\overleftarrow{e})$ then $f(\overleftarrow{e}) \neq u(\overleftarrow{e})$ and therefore observation (1) does not hold for $\overleftarrow{e}$.

Therefore, in the feasible circulation f, the following holds for each edgee $= (u, v)$ such that $e \in E$ or $\overleftarrow{e} \in E$ ($\overleftarrow{e} = (v, u)$):
1. $c^\pi(e) < 0 \rightarrow f(e) = u(e)$
2. $c^\pi(e) > 0 \rightarrow f(e) = -u(\overleftarrow{e})$

Therefore, the only thing left for us is to find the flow for each edge $e = (u, v) \in E$ such that $c^\pi(e) = 0$ ($c^\pi(e) = 0 \iff c^\pi(\overleftarrow{e}) = 0$). We know that in order for f to be a feasible circulation the following must hold: $\forall v \in V \sum_w f(v, w) = 0$. Therefore, we need to find a feasible circulation such that the flow on every edge with a non zero reduced cost will be according to our observations. $\forall v \in V$ let $b(v) = \sum_{w: c^\pi(v,w) \neq 0} f(v, w)$ - ($b(v)$ is the previous sum limited to edges with reduced cost that is not zero). We can reduce this problem to the following one: We shall delete the edges with a non zero reduced cost from our graph. For every node $v \in V$, $b(v)$ will be the amount of flow that is contained in v due the edges that we have deleted. We need to find a flow g over the reamaining edges (those with the zero reduced cost) such that $\forall v \in V : \sum_w f(v, w) = -b(v)$. Therefore the flow f that is defined on the non zero reduced cost edges according to our observations and is defined over the zero reduced cost edges according to g will be feasible and will be according to our observations. (b)

We can find the flow g by defining a new graph $H = (V \cup \{s, t\}, E_z \cup E_s \cup E_t)$ such that s, t are new vertices that are not in V. $E_z = \{e \in E \mid c^\pi(e) = 0\}$, the capacity of each edge $e \in E_z$ is the same as in the original graph. $E_s = \{(s, u) \mid b(u) > 0\}$ and the capacity of each edge $e = (s, u)$ in E_s is $b(u)$ and $E_t = \{(v, t) \mid b(v) < 0\}$ and the capacity of each edge $e = (v, t)$ in E_t is $-b(v)$. Now we can simply run the best algorithm that we know for the max flow problem over the new graph H (With s being the source and t being the sink). If there is a feasible solution to our problem (and it's given that such a solution exists) then there is a flow that is saturating all the edges in E_s and in E_t (And this is of course a max flow because it's equal to a size of a cut - the cut between s and the other vertices). Therefore the solution to the max flow problem over H is defining the flow g over the zero reduced cost edges in G. And therefore the flow f that we defined in end of the previous paragraph is a feasible circulation in G and our observations holds for the flow f and therefore in the G_f all the edges are with a non negative reduced cost. f is a feasible circulation and there is a potential function such that all the edges in G_f are with a non negative reduced cost and therefore according to a thm proved in class, f in a min cost circulation.

Therefore, our algorithm is to build H and then running a max flow algorithm on it. The result of the max flow algorithm will be used in order to build the min cost circulation f. Building H is $O(m + n)$ (Finding the non zero edges is $O(m)$ work, computing $b(v)$ is $O(m)$ work. The number of edges in H is linear in the size of edges in G (Because E_s and E_t are linear in the size of $V(G)$). We can use the max flow algorithm that was learned in class with complexity of $O(mn \cdot log(n^2/m))$. Building the min cost circulation f from the max flow solution is $O(m)$. Therefore the total complexity is $O(mn \cdot log(n^2/m))$.

10

10

First we shall prove that we shall end up with a max flow. At iteration, we are choosing an augmenting path, and augmenting as much flow along the path as possible. Therefore this is a private case of Ford-Fulkerson algorithm and we know that the Ford-Fulkerson always stops and finds a max flow when the capacities are integers. Therefore, the algorithm does stop and finds max flow.

We shall now prove that the when the algorithm stops the solution is the max flow with the minimum cost.

Lemma 1: *Let* $N = (G, u, c, s, t)$ be a flow network and let f and $f*$ be two feasible flows such that: 1. They have the same flow value. 2. $Cost(f*) < Cost(f)$. Then the flow $f^{\$} = (f * - f)(e)$ is a circulation in the residual network graph of N_f.

Proof: We shall prove that $f^{\$}(e) = (f * - f)(e)$ is a circulation~~flow~~ in the residual network of N_f:

1. We shall prove that for any $e = (w, v)$, $f^{\$}(e) = -f^{\$}(e^{-1})$ $(e^{-1} = (v, u)$ iff $e = (u, v))$. $f^{\$}(e) = f * (e) - f(e) = -[-f * (e) + f(e)] = -[f * (e^{-1}) - f(e^{-1})] = -f^{\$}(e^{-1})$

2. We shall prove that for any $e = (w, v)$, $f^{\$}(e) \leq r(e) = u(e) - f(e)$. $f^{\$}(e) = f*(e) - f(e) \leq u(e) - f(e)$ (Because $f * (e) \leq u(e)$)

3. We shall prove that for any $v \in V(G)$: $\sum_{w \in V(G)} f^{\$}(v, w) = 0$. For $v \in V - \{s, t\}$ we know that $\sum_{w \in V(G)} f * (v, w) = \sum_{w \in V(G)} f(v, w) = 0$ and therefore $\sum_{w \in V(G)} f^{\$}(v, w) = 0$. For the source - s, because f and $f*$ are feasible flows with the same flow value then $\sum_{w \in V(G)} f * (s, w) = \sum_{w \in V(G)} f(s, w) = |f| = |f * |$. And therefore $\sum_{w \in V(G)} f^{\$}(s, w) = 0$. In the same way we can show that for the sink - t: $\sum_{w \in V(G)} f^{\$}(t, w) = 0$.

Lemma 2: *Let* $N = (G, u, c, s, t)$ be a flow network and let f a feasible flow. Then f is a min cost flow (Among all the flows with flow value of $|f|$) iff the residual graph N_f does not contain a negative cycle in respect to the cost function. (a)

Proof: Let $Val(f)$ denote the value of the flow f. If the residual network N_f contains a cycle C of negative cost, let g be a flow in N_f such that $g(e) = min_{e \in C} u_f(e)$, for every $e \in C$, and $g(e) = 0$ otherwise. Clearly $Val(g) = 0$ and $Cost(g) < 0$. Therefore $f + g$ is a cheaper flow in N of value $Val(f)$.

Conversely, suppose that $f^*$is a flow in N with $Val(f^*) = Val(f)$ and $Cost(f^*) < Cost(f)$. The flow $f^* - f$ in the residual network has cost $Cost(f^*) - Cost(f) < 0$. According to Lemma 1, the flow $f * - f$ is a circulation in the residual graph N_f. Therefore, according to the slides from the lecture, the fact that $Cost(f*) - Cost(f) < 0$ is enough in order to prove that there exists a negative cycle in N_f with respect to the cost function.

Lemma 3: *Let* $N = (G, u, c, s, t)$ be a flow network and let f be a min cost flow (Of value $Val(f)$) in N. Let g be a flow of value δ along a cheapest path P from s to t in the residual network N_f, with respect to the cost function c. Then $f + g$ is a min cost of value of $(Val(f) + \delta)$.

Proof: *Let* $f*$be an arbitrary flow of value $Val(f*) = Val(f) + \delta$ in N. We need to show that $Cost(f + g) \leq Cost(f*)$. We shall consider the flow $(f * - f)$ in the residual network N_f. Its value is $Val(f * - f) = \delta$. Any flow in N_f can be decomposed as the sum of flows along augmenting paths and cycles in N_f. Therefore, the flow δ can be decomposed to the paths $p_1, ..., p_k$ such that $\sum_{i=1}^{k} flow(p_i) = \delta$ and the cost $Cost(f * - f)$ can be represented as the sum $Cost(f * - f) = \sum_{i=1}^{k} flow(p_i) cost(p_i)$. f is a min cost flow of value $Val(f)$ and therefore N_f does not contain negative cost cycles (By Lemma 2). Therefore a cheapest path P in the residual network exists. As each path P' in N_f has cost of at least $c(P)$, we get that $Cost(f * - f) = \sum_{i=1}^{k} flow(p_i) cost(p_i) \geq c(P) \sum_{i=1}^{k} flow(p_i) = c(P)\delta$. Thus, $Cost(f + g) = Cost(f) + \delta c(P) \leq Cost(f) + Cost(f * - f) = Cost(f*)$.

We can now prove that when the algorithm stops the solution is the max flow with min cost. We already showed that the algorithm stops after a final number of iterations t and that when the algorithm stops we have a max flow.

We'll show by induction on the number of iterations, that after the i-th iteration, the current flow f_i is a min cost flow (among all the flow with $Val(f_i)$). We shall define the flow in iteration 0 as the initial flow which is zero in all the edges.

Base: $i = 0$. $f_0 = 0$. We know that all the costs are non-negative and therefore the best cost that we can wish for is 0. We shall notice that if $\forall e: f(e) = 0$ then we have a 0 flow with a.0 cost, and therefore it's a min cost flow (Among the flows with flow value 0).
Step: We shall assume that after the $k - th$ iteration, f_k is a min cost flow (among the flows with flow value of $|f_k|$). We shall prove that after the $(k+1) - th$ iteration, $f_{k+1}$is a min cost flow (among all the flows with a flow value of $|f_{k+1}|$). We know that f_k is a min cost flow (among the flows with flow value of $|f_k|$) and we know that the algorithm will augment flow along the cheapest path (with respect to the cost function), and therefore, according to $Lemma\,3$, the new flow f_{k+1} will be a min cost flow (among the flows with flow value of $|f_{k+1}|$).

In the beginning of the answer, we showed that after the last iteration, t, the flow is a maximal flow. And now we showed that it's a min cost maximal flow.

□

(b)

Assume the maximum cost of an edge is C and the maximum capacity of an edge is U. Suggest a concrete implementation of this algorithm and analyse its running time.

Problem 2 [(b)] continued on next page...

We shall describe the algorithm:

1. Let $\pi(v)$ be a potential function. Init $\forall v \in V : \pi(v) = 0$. Init f to be zero flow on all the edges.
2. While there is a path from s to t in the residual graph G_f
   (a) Find the shortest paths from s in G_f using Dijkstra according to the reduced cost $c^{\pi}(e)$
   (b) Let $\delta(v)$ be the cost of the shortest path from s to v in G_f according to original cost. Define $\pi(v) = -\delta(v)$
   (c) Let p be the shortest path from s to t in G_f. Augment as much flow along the path as possible.

Correctness: We are using *Dijkstra* and therefore we need to make sure that the reduced cost is non negaitve for edges in G_f. We'll prove that in the beginning of each iteration the reduced cost are non negative for the edges in G_f. We'll prove this by induction on the number of iterations.

Base: $i = 0$. The flow is zero, and therefore in G_f we have only edges $e \in E$. We defined $\pi(v) = 0$ and therefore the reduced cost are like the original cost which we know that are non negative.

(b)

Step: We shall assume that in the beginning of iteration $i = k$ the reduced cost were non negative for the edges in G_f. Therefore we could run *Dijkstra* on the reduced cost $c^{\pi}(e)$ and get the tree of the shortest paths from s in G_f. Using this tree we can compute $\delta(v)$ - the cost of the shortest path from s to v according to the original cost (Look at the paths in the tree and compute the cost according to the original costs). Our new potential function is $\pi(v) = -\delta(v)$. We know that $\delta(v) \leq \delta(u) + c(u, v)$ - because $\delta(v)$ is the cost of the shortest path from s to v. And therefore: $c^{\pi}(u, v) = c(u, v) + \pi(v) - \pi(u) \geq 0$. Now we are augmenting flow according to the shortest path from s to t. The only edges that will be added to G_f as the result of the augmenting action, are edges $e = (v, u)$ such that $\overleftarrow{e} = (u, v)$ is an edge on the shortest path from s to t that we used in order to augment the flow. But $c^{\pi}(\overleftarrow{e}) = c(\overleftarrow{e}) - \delta(v) + \delta(u)$. And $\delta(v)$ is the shortest path from s to v and $\delta(u)$ is the shortest path from s to u. But $\overleftarrow{e} = (u, v)$ is on the shortest path from s to t and therefore according to the attributes of the shortest path, we have that $\delta(v) = \delta(u) + c(u, v)$. Therefore $c^{\pi}(\overleftarrow{e}) = c(\overleftarrow{e}) - c(\overleftarrow{e}) = 0$. If $c^{\pi}(\overleftarrow{e}) = 0$ then $c^{\pi}(e) = 0$ and therefore the edges that we added to G_f have a zero reduced cost, and we know that the other edges that were in G_f had non-negative zero cost, and therefore in the beginning of the next iteration the reduced cost of the edges in G_f will be non negative.

Complexity: We shall notice that in every iteration the reduced cost of an edge is $c^{\pi}(e) = c(e) + \delta(u) - \delta(v)$. Where $\delta(v)$ is the shortest path from s to v in G_f according to the original costs. Therefore $\forall v \in V : -C \cdot (n-1) \leq \delta(v) \leq C \cdot (n-1)$ (Because the max length of a path is $|V| - 1 = n - 1$ and the cost of every edge in the residual graph is between $-C$ and C. Therefore, $0 \leq c^{\pi}(e) \leq C + 2 \cdot C \cdot (n-1) = C \cdot (2n-1)$. And the cost of a simple path P from s to v in G_f according to the reduce cost is $0 \leq c^{\pi}(P) \leq c(P) + \delta(s) - \delta(v) = c(P) - \delta(v) \leq C \cdot (n-1) + C \cdot (n-1) = C \cdot (2n-2)$. And all the costs are integer, therefore we can use *Dial's* implementation for *Dijkstra* with integral weights. therefore we shall keep $2C \cdot n$ buckets and therefore distance updating and finding the min value will be $O(1)$. Therefore running *Dijkstra* according to *Dial's* implementation will cost us $O(C \cdot n + n + m) = O(C \cdot n + m)$.

Building the new potential in every iteration is $O(n)$, augmenting the flow and updating G_f is $O(n)$ and therefore the total cost in an iteration is $O(C \cdot n + m)$. We need to find an upper bound to the number of iterations. We know that the capacities are integral and the max capacity value is U. Therefore in every iteration we are augmenting at least 1 unit of flow. We know that each cut in the graph is an upper bound to the max flow. If we'll look on the cut $(V - \{t\}, \{t\})$ we shall find that the cut can't be larger than $O(U \cdot n)$. Because at most $n-1$ vertices are connected to t and the capacity of each edge is bound by U. Therefore we can have at most $O(U \cdot n)$ iterations. Therefore the total cost of the algorithm is $O(U \cdot n(C \cdot n + m))$. □

A final observation: If C is too large than maybe we should use the regular *Dijkstra* algorithm with a *Fibonacci Heap* and get $O(m + nlogn)$ work per iteration and then the total costr of the algorithm will be: $O(U \cdot n(nlog(n) + m))$.

$\frac{10}{10}$

We shall define a new graph H which is the same as G except adding two vertices s and t, and we shall add the edges $\{(s,x)|x \in V_1\}$ and the edges $\{(y,t)|y \in V_2\}$. Let $u(e)$ denote the capcity of an edge. We shall define that $\forall e \in E(H) : u(e) = 1$. The cost of every edge (x,y) such that $x \in V_1$ and $y \in V_2$ will be according to the given $c(e)$ function. The cost of every edge (s,x) such that $x \in V_1$ will be 0, and the cost of every edge (y,t) such that $y \in V_2$ will be 0 as well.

The min cost flow problem presented in class is to find a maximum flow of minimum cost.
We shall notice that the maximum flow is at most n. This is because we have the cut $(\{s\}, V(H) - \{s\})$ and its value is n and we know that each cut is an upper bound to the max flow. Furthermore, there exists an integral flow with value n - for example the flow in which $f(e) = 1$ for every $e = (s,x)$ such that $x \in V_1$, $f(e) = 1$ for every $e = (y,t)$ such that $y \in V_2$ and $f(e) = 1$ for every $e = (v_{1,i}, v_{2,i})$ such that$1 \le i \le n$ (When $v_{1,i}$ corresponds to the i-th vertex in V_1). Therefore the value of the max flow is exactly n.

We shall notice that all the capacities in H are integer ($\forall e : u(e) = 1$) and therefore there exists an integral min cost maximal flow.

Lemma 1 : Every integral max flow solution in H with cost C corresponds to a perfect matcing in G with cost C.
Proof : Let f be an integral max flow solution. The capacity of every edge in $E(H)$ is 1 and therefore in an integral solution the flow of each edge e is 0 or 1. We saw before that the value of f must be n, therefore $f(e) = 1$ for every $e = (s,x)$ such that $x \in V_1$ and $f(e) = 1$ for every $e = (y,t)$ such that $y \in V_2$ (Otherwise the flow will be less than n and therefore not optimal). Therefore for evey node $x \in V_1$ there is one unit of flow that enters to x, and therefore there must be one unit of flow that exits from x, and because it is an integral solution there exists exactly one edge $e = (x,y)$ such that $f(e) = 1$ and $y \in V_2$ (all the rest have 0 flow). In the same way we can argue that for every $y \in V_2$ there exactly one edge $e = (x,y)$ such that $f(e) = 1$ and $x \in V_1$. Therefore if we'll take all the edges $e = (x,y)$ such that $f(e) = 1$ and $x \in V_1$ and $y \in V_2$ we'll get a perfect matching. Furthermore, the cost of this flow is exactly the cost of the matching that we got.

(a)

Lemma 2 : For every perfect matching in G with cost C, we have an integral max flow solution in H with cost C. Let M be a perfect matching solution in G with cost C.
Proof : We shall define the following integral max flow: $f(e) = 1$ for each $e = (s,x)$ and $x \in V_1$, $f(e) = 1$ for each $e = (y,t)$ and $y \in V_2$ and $f(e) = 1$ for every $e = (x,y)$ such that $e \in M$. For all the rest $f(e) = 0$. This is a valid flow because for every $x \in V_1$ there exactly one unit of flow enters to it (from s) and exactly one unit of flows exits from it (M is a matching and therefore we have exactly one edge (x,y) such that $f(e) = 1$), in the same way we can argue that there is exactly one unit of flow that enters every $y \in V_2$ and there exactly one unit of flow that exists every $y \in V_2$. One can notice that the value of the flow is n and therefore it's max flow, and the cost of this max flow is exactly the cost of the perfect matching.

We know that there exists an integral min cost max flow, and we shall denote it by $f*$ and its cost by $c*$. Therefore we can find it by running the min cost algorithm described in class. By *Lemma* 1 this flow corresponds to a matching $M*$ with cost $c*$. And by *Lemma* 2, every other perfect matching has a cost $c \le c*$. Therefore $M*$ is the min cost perfect matching and by *Lemma* 2 we know how to build $M*$ from $f*$.

Problem 3 continued on next page...

(b)

/. A bus which can carry at most 50 people travels along a path through locations $1, 2, ..., n$ in this order. For every $1 \leq i \leq n$, and $1 \leq j \leq n$, such that $i \in j$, you are given q_{ij}, which is the maximum number of people that want to travel from point i to point j; and the price f_{ij} for a single ticket that takes you from point i to point j. Show how to use the minimum cost flow algorithm to find a "plan" for the bus driver to make maximum profit. A plan means how many people to load/drop at each location such that it never carries more than 50 people.

$i<j$

Let $C = 50$. We shall define the following graph: for each station $1 \leq i \leq n$ we shall define a vertex r_i and a vertex o_i. We shall add two vertices s and t. We shall add an edge (o_i, r_j) for all $1 \leq i \leq j \leq n$ with capacity q_{ij} and cost $-f_{ij}$, an edge (r_i, o_i) for each $1 \leq i \leq n$ with capacity C and cost 0, an edge between (r_i, r_{i+1}) for $1 \leq i < n$ with capacity C and cost 0 and the edges (s, r_1) and (t, r_n) with capacity C and cost 0.

We shall notice that the flow can be larger than C (The cut $(\{s\}, V(G) - \{s\})$ has size C and every cut is an upper bound for a flow). Furthermore, there is a flow with the value C - $f(s, s_1) = C$, $f(s_i, s_{i+1}) = C$ for $1 \leq i < n$ and $f(s_n, t) = C$ (The flow on all the other edges will be zero). Therefore the value of the max flow is C. The capacities are integral and therefore there is a solution for the min cost max flow which is integral.

Lemma 1 : Every plan P with cost $Cost(P)$ corresponds to an integral flow f with cost $-Cost(P)$. *Proof* : Let P be a plan with cost C. We shall define an integral flow f: Let $f(s, r_1) = C$. Let p_{ij} be the number of people that will travel from point i to point j according to the plan, define $f(o_i, r_j) = p_{ij}$. Let $O_i = \sum_{j=i+1}^{n} p_{ij}$, define $f(r_i, o_i) = O_i$. We shall define now define $f(r_i, r_{i+1})$, for our convenience we shall define $r_0 = s$ and $r_{n+1} = t$. We already defined $f(r_0, r_1)$, we shall now define $f(r_i, r_{i+1})$ for $1 \leq i \leq n$ recursively: Let$E_i = \sum_{j=1}^{i-1} p_{ji}$, define $f(r_i, r_{i+1}) = f(r_{i-1}, r_i) + E_i - O_i$. We shall prove that this is a flow: $0 \leq f(s, r_1) = C$. Because P is a plan we know that $0 \leq f(o_i, r_j) = p_{ij} \leq q_{ij}$. One can see that O_i is the number of people that takes the bus in point i according to the P. P is a valid plan and therefore $0 \leq f(r_i, o_i) = O_i \leq C$. One can see that $f(r_{i-1}, r_i)$ is $C - l_i$ when l_i is the number of people that took the bus in point $k < i$ and left the bus in point $j \geq i$. Therefore l_i is the number of people that are on the bus when the bus travels from point $i - 1$ to point i. We know that P is a valid plan, and therefore $l_i + O_i - E_i \leq C$. Therefore $C - l_i - O_i + E_i \geq 0 \rightarrow f(r_i, r_{i+1}) = r(i_{i-1}, r_i) + E_i - O_i \geq 0$. We now need to show that for every r_i, o_i such that $1 \leq i \leq n$, the flow that goes in equal to the flow that goes out. This is true for r_i because we defined that $f(r_i, r_{i+1}) = f(r_{i-1}, r_i) + E_i - O_i$, therefore $flow_{in}(r_i) = f(r_{i-1}, r_i) + E_i = f(r_i, r_{i+1}) + O_i = flow_{out}(r_i)$. And it is also true for o_i because $flow_{in}(o_i) = O_i = \sum_{j=i+1}^{n} p_{ij} = flow_{out}(o_i)$. One can notice that according to the definition of $f(r_i, r_{i+1})$, we get that $f(r_n, t) = C$, because in r_n we get that $O_n = 0$ and l_n must be equal to E_n (All the people that are still on the bus will leave at the final station r_n) and therefore $f(r_n, t) = C - l_n + E_n + O_n = C - l_n + l_n + 0 = C$. Therefore we have a valid flow, and one can see that the cost is only according to the edges (r_i, o_j) and therefore it's equal to $-\sum_{i=1}^{n} \sum_{j>i}^{n} p_{ij} f_{ij} = -Cost(P)$ (The minus is because we defined the cost to be with an opposite sign to the one given).

Lemma 2 : Every integral flow f with cost $Cost(f)$ corresponds to a valid plan P with cost $-Cost(f)$. We shall define a plan P: p_{ij} (The number of people that will go from station i to station j according to the plan) will be equal to $f(o_i, r_j)$. We need to see that this is a valid plan. The flow is integral and therefore p_{ij} has a logic meaning. $p_{ij} = f(o_i, r_j) \leq u(o_i, r_j) = q_{ij}$, and therefore the number of people that travel from i to j is at most q_{ij}.

(b)

We need to show that no more that $C = 50$ people are on the bus at given time. It is enough to show that when the bus travels from point i to point $i+1$ there are at most C people on the bus. According to the plan that we derived from the flow, the number of people that are on the bus from point i to point $i+1$ is equal to $\sum p_{kj}$when $k \leq i$ and $j \geq i+1$ (The number of people that took the bus before station $i+1$ and that are leaving ths bus after station i). I will explain why this sum is at most C, it will not be a formal proof but will explain the idea. We will assume by contradiction that this number is larger than C. The flow from s to r_1 is at most C because of the capacity. A flow that went from $o_i$to r_j $(j > i)$ can't return to a node $k < j$. (This is due to the structure of the graph). Therefore, the initial flow was at most C, and once that amount of flow goes from $k \leq i$ to $j \geq i+1$ we can't use it again in nodes with index$\leq i$. Therefore, once we passed C units of flow from $k \leq i$ to $j \geq i+1$ we don't have any more flow to pass from nodes with index $\leq i$ to nodes with index $\geq i+1$ (Without breaking the rule that the flow that goes in equal to the flow that goes out) and therefore we can't pass more than C units of flow from node $k \leq i$ to a node $j \geq i+1$ and therefore this number if bounded by C. Therefore our plan is a valid plan., and the cost of the plan equal to $Cost(P) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} p_{ij} f_{ij} = \sum_{i=1}^{n} \sum_{j=i+1}^{n} f(i,j)c(i,j) = -Cost(f)$. □

We know that there is a solution of the min cost max flow which is integral. Therefore there exists an integral min cost max flow $f*$ with cost $c*$. According to *Lemma* 2 $f*$ corresponds to a plan $P*$ with cost $-c*$. And according to *Lemma* 1, any other plan P with cost c corresponds to an integral flow f with a cost $-c$, such that $c* \leq -c$, and therefore $-c* \geq c$. But $-c*$ is the cost of the plan $P*$ and c is the cost of another plan P and therefore $P*$that we found is the plan with the max profit.

Problem 4

$\frac{10}{10}$

Consider the dynamic connectivity algorithm presented in class. Assume that you get as an input a graph $G = (V, E)$ and a spanning forest F, together with a level $l(e)$ for each edge $e \in E$. Suggest an algorithm that decides if F and the level function satisfy the invariants of the dynamic connectivity algorithm. Analyze the running time of your algorithm.

The invariants are:
1. The forest is a maximum spanning forest with respect to the levels of the edges
2. Let F_i be the subforest of edges of level $\geq i$, then each tree in F_i is of size no larger than $\frac{n}{2^i}$.

We shall use the kruskal algorithm for finding a MST: (The weights are the levels of the edges)

1. Let S be the set of all the edges in G.
2. While S is not empty
   (a) Remove an edge with min weight from S
   (b) If that edge connectes two different trees, then add it to the forest, combining the two trees into a single tree
   (c) Otherwise, discard that edge

We shall use the following property of Kruskal Alg (I will not prove this but it can be shown): For each MST there is an order such that the Kruskal Alg will produce the MST when running by that order

Therefore we shall define the following order:
Let S_i be the set of all the edges with weight i. Let ST_i be the edges in S_i that are in F and let SR_i be the edges in S_i that are not in F. Therefore we shall use the following order:
$O =< ST_1, SR_1, ST_2, SR_2, ..., ST_{logn}, SR_{logn}$ >this is a valid order because the edges are ordered according to their weights. If F is indeed a maximum spanning forest, then the kruskal algorithm should create F when running on G with the order O. Therefore, we can check invariant 1, by running the Kruskal algorithm on G with order O, and for every edge e we shall check that $e\, connects\, two\, different\, trees \iff \exists i : e \in ST_i$. We can notice that in the when we are connecting two different trees by an edge e and creating a new tree T, we can check that the size of T is $\leq \frac{n}{2^{l(e)}}$ (We are iterating over the edges according to O and therefore when creating T by connecting with e, the tree T will have edges with level $\leq l(e)$ and therefore we can check invariant two in the same time).

We shall now give a practical implementation, we shall use two data structures:
1. $Union - Find$: For each edge we need to check if the two endpoints are in the same tree or not, and if not then we need to unite them. We shall use the implementation in which k operations on the data structure will cost us $klog^*(n)$. And we shall save in the root of every tree in the union find algorithm, the size of that tree (The number of elements in the tree), when we are calling union we can update this value in $O(1)$ without hurting the performance.
2. We shall notice that our weights are integers between 1 and $log(n)$, and when using kruskal we are iterating over them in increasing order. Therefore we shall use an array with $1...log(n)$ cells, and each cell with contain a pointer to a linked list with all the edges with that weight. Creating the data structure will cost us $O(logn + m)$.

Problem 4 continued on next page...

Finally the algorithm:

```
(1) Create $U$ an empty union find DS. Create a singleton set fo every $v \in V$
(2) Create $A$ the DS described in (2).
(3) Insert all the edges in $E(F)$ to $A$. Now iterate over $E(G)$ and add every edge e that is not in $E(F)$
    to $A$ (Therefore $A$ is according to the order $O$)
(4) j=0
(5) valid=true
(6) While $A$ is not empty
    (I) If $A[j] = null$
        (A) $j = j + 1$
        (B) continue.
    (II) Else
        (A) Let $(u, v) = A[j]$.
        (B) Remove the edge from $A[j]$
        (C) Let $inForest = 1 \iff (u, v) \in F$.
        (D) Let $inTheSameTree = 1 \iff (u, v)\, are\, in\, the\, same\, tree\, according\, to\, U$
        (E) If $inForest \oplus inTheSameTree \neq 1$
            (i) $valid = false$
            (ii) break
        (F) If $inTheSameTree = 0$
            (i) Use union operation in $U$ in order to unite them.
            (ii) Let s be the size of the new tree (after the union)
            (iii) If $s > \frac{n}{2^{l(e)}}$
                (a) $valid = false$
                (b) break
(7) return valid
```

Correctess: It's easy to see that the algorithm will return $valid = true \iff F\, and\, the\, level\, function\, satisfy\, thei\, nvariants\, of\, the\, dynamic\, connectivity\, algorithm$

Complexity: Creating U is $O(n)$. Creating A is $O(log(n) + m)$. For every e, taking e from A is $O(1)$. Removing is $O(1)$. Checking if A is empty is $O(1)$. Finding if they are in the same tree and union is $O(log * n)$.
Therefore: $O(n + m + log(n) + mlog * (n)) = O(n + mlog * (n))$.

$\frac{10}{5}$

Let S be a set of horizontal segments on the line. The set S satisfies the following "nesting" property: Every two segments $s_1, s_2 \in S$ are either disjoint or one contains the other. In addition each segment s has a cost $c(s)$. Assume that the costs of different segments are different. Note that such a family of segments defines a natural forest where s_1 is the parent of s_2 if s_1 contains s_2 and there is no segment s that is contained in s_1 and contains s_2. A query is a point q on the line and the answer should be the largest cost of a segment containing q, or an indication that there is no segment containing q.

(a)

Show how to preprocess S into a simple data structure that can answer such queries in $O(logn)$ time per query. What is the size and the construction time of your data structure?

We shall use a $1D - \mathit{SegmentTree}$ that was taught is the beginning of the course. But instead of saving in every internal node, the labels of the relevant segments, we'll save only the min cost value of them. And when we'll call query on a point q, we'll go over the query path and calculate the min value over the min cost values that will be in the internal nodes along the query path. If in every internal node, there is no min cost value, then there is no segment in S that contains q. Otherwise, the min value that we computed is the min cost among the segments that contain q.

Correctness: A query on a regular segment tree return the list of all the intervals that contains the point. Therefore our variation will return the min cost value over all the segments that contain the query point. (a)

Complexity: Building the segment tree is $O(nlogn)$. Computing the min cost in every internal node does not change the complexity of building the tree. In every node we are only saving the min-cost value and therefore the size of the structure is $O(n)$. A query is $O(logn)$ because the path is $O(logn)$ and the operation on every internal node is $O(1)$.

(b)

We want to support also an insert and a delete operations. When we insert a segment we are guaranteed that the nesting property is preserved. Describe a data structure in which each operation (insert/delete/and query) takes $O(logn)$ time. (Hint, use dynamic trees and handle high degree vertices in away similar to the algorithm for maintaining a spanning tree in a dynamic planar graph.)

Problem 5 [(b)] continued on next page...

My answer will be based on the following paper: "AN OPTIMAL DYNAMIC DATA STRUCTURE FOR STABBING-SEMIGROUP QUERIES" By PANKAJ K. AGARWAL, LARS ARGE, HAIM KAPLAN, EYAL MOLAD, ROBERT E. TARJAN, AND KE YI.

As stated in the question, we can build a natrual forest F such where where s_1 is the parent of s_2 if s_1 contains s_2 and there is no segment s that is contained in s_1 and contains s_2. If a node s has more than one children then we order its children in increasing order of their left endpoints. We shall notice that if we'll add the interval $\xi = [-\infty, \infty]$ to S then F becomes a tree T; We shall add the interval ξ with cost $c(\xi) = \infty$.We define the weight of an edge e from an interval to its parent to be $c(s)$. It's easy to see that the min cost over the edges on the path from a node s to its root is the min cost between s and all the intervals that contain s (Because $c(\xi) = \infty$). For a point q let s_q be the smallest interval that contains q. Therefore the answer to our query which we shall denote by $M(q)$ is equal to $min_{e \in \Pi(s_q, T)} c(e)$.

A weakness of T is that an insertion or deletion of an interval may require insertions and deletions of many edges. We thereofre represent T by a binary tree $\mathbb{B}$; The nodes of $\mathbb{B}$ are the same as the nodes of T.
The root of the tree $\mathbb{B}$ is ξ.
A left child of a node v in $\mathbb{B}$ is the first child of v in T, or *null* if v is a leaf in T. The weight of an edge between v and its left child is the weight of the interval v.
A right child of a node v in $\mathbb{B}$ is the right sibling of v in T, or *null* if v is the rightmost child of its parent in T. The weight of the edge from v to its right child is 0.

Assuming that costs are positive

Let $\Pi(s, \mathbb{B})$ be the path from s to the root $\mathbb{B}$ in $\mathbb{B}$. Therefore, according to the definitions of T and $\mathbb{B}$, for any node $s \in B$: $min_{e \in \Pi(s, \mathbb{B})} c(e) = min_{e \in \Pi(p(s), T)} c(e)$. Therefore $M(q) = min\{min_{e \in \Pi(s, \mathbb{B})} c(e), c(s_q)\}$.

$\boxed{\text{(b)}}$

It's easy to verify that an insertion or a deletion of an interval requires only $O(1)$ insertion and deletions of edges *to/from* $\mathbb{B}$.

We maintain $\mathbb{B}$ as a dynamic tree data structure. We also store the endpoints of all intervals in a balanced search tree $\mathcal{P}$. If x is an endpoint of an interval $s \in S$, we store a pointer at the node of $\mathcal{P}$ that stores x to the node of $\mathbb{B}$ corresponding to the interval s. The overall size of the structure is linear.

We shall now describe how to implement quert, insert and delete:

Query:
Let $q \in \mathbb{R}$ be a query point, we shall compute $M(q)$ as follows; We first find in $O(log(n))$ time the predecessor x of q in $\mathcal{P}$. Suppose x in an endpoint of the interval $s \in S$. We have two cases:
1. x is the right endpoint of the interval s. Therefore $s_q = p(s)$. If s_q is ξ, then we return that there is no segment that contains q. Otherwise, $M(q) = min_{e \in \Pi(s, \mathbb{B})} c(e)$, and we use the dynamic tree operation $mincost(s)$ in order to find the value $M(q)$.
2. x is the left endpoint of the interval s. Therefore $s = s_q$ and $M(q) = min\{min_{e \in \Pi(s, \mathbb{B})} c(e), c(s_q)\}$. We shall use $mincost(s)$ in order to compute $min_{e \in \Pi(s, \mathbb{B})} c(e)$ and then we shall return the min value between the result and $c(s_q)$.
The implementation of query takes $O(logn)$ time, because searching in $\mathcal{P}$ is $O(logn)$ and $mincost$ operation in dynamic trees is $O(logn)$.

Problem 5 continued on next page...

Add:
To insert an interval $s = [a, b]$, we need to update both $\mathcal{P}$ and $\mathbb{B}$. We first update $\mathbb{B}$ and add to it a node representing s and then we insert a and b to $\mathcal{P}$. When we add a and b to $\mathcal{P}$, we also store pointers in the nodes containing them to the node containing s in $\mathbb{B}$. We first find the predecessor and successor a^-, a^+ (resp. b^- and b^+) of a (resp. b) in $\mathcal{P}$. Suppose $a^-, a^+, b^- and b^+$are the endpoints of the intervals $l^-, l^+, r^- and r^+$, respectively.

We allocate a new node for s and update its children as follows: If $a^+ > b$, then s does not contain an interval of S, so s is a leaf of T. Otherwise $l^+$is an interval that s contains and it should be the leftmost child of s in T, and r^- should be the rightmost child of s in T. So we make $l^+$the leftmost child of s in $\mathbb{B}$ by preforming $CUT(l^+)$ followed by $LINK(l^+, s, c(s))$. The right child of s in $\mathbb{B}$ should be the right sibling of s in T. If $b^+$is the right endpoint of r^+, then s is the rightmost child of its parent in T so s does not have another child in $\mathbb{B}$. If $b^+$is the left endpoint of r^+, then r^+ should be next sibling of s in T, so to update $\mathbb{B}$ we perform $CUT(r^+)$ followed by $LINK(r^+, s, 0)$.

Finally we set the parent of s in T. If $a^-$is the left endpoint of l^-, then s is the leftmost child of l^- in T, and we perform $link(s, l^-, c(l^-))$. Otherwise, l^- is the left sibling of s in T, and we perform $LINK(s, l^-, 0)$.

This implementation of insert takes $O(logn)$ time: Searching a^-, a^+, b^-, b^+ in $\mathcal{P}$ is $O(logn)$. Once we locate them, we perform a constant number of CUT and $LINK$ operations, which also take $O(logn)$.

Delete:
To delete an interval $s = [a, b]$, we need to update both $\mathcal{P}$ and $\mathbb{B}$. We shall first find the interval s in $\mathbb{B}$ by searching a (or b) in $\mathcal{P}$ and using the pointer contained in the node that we found. □

If s is a left child of an interval v in $\mathbb{B}$ then s is the leftmost child of v. We have 3 options:
1. s does not have any children, the we shall $CUT(s)$, create a null node n and $LINK(n, v, c(v))$.
2. s has a left child, lc, then lc is now the leftmost child of$v$, therefore, we should perform $CUT(lc)$, $CUT(s)$, $LINK(lc, v, c(v))$. If s has also a right child, rc, then, rc is now the right sibling of the predecessor of s. Therefore we shall find p, the predecessor of s by using$\mathcal{P}$. And now we shall call $CUT(rc)$, $LINK(rc, p, 0)$.
3.s has only a right child rc. Therefore rc will now be the leftmost child of v, and we shall call $CUT(rc)$, $CUT(S)$, $LINK(rc, v, c(v))$.

If s is a right child of an interval v in $\mathbb{B}$then s is the right sibling of v. We have 3 options:
1. s does not have any children, the we shall $CUT(s)$, create a null node n and $LINK(n, v, 0)$.
2. s has a left child, lc, then lc is now the right sibling of$v$, therefore, we should perform $CUT(lc)$, $CUT(s)$, $LINK(lc, v, 0)$. If s has also a right child, rc, then, rc is now the right sibling of the predecessor of s. Therefore we shall find p, the predecessor of s by using$\mathcal{P}$. And now we shall call $CUT(rc)$, $LINK(rc, p, 0)$.
3.s has only a right child rc. Therefore rc will now be the right sibling of v, and we shall call $CUT(rc)$, $CUT(S)$, $LINK(rc, v, 0)$.

This implementation of insert takes $O(logn)$ time: Searching a or b in $\mathcal{P}$ is $O(logn)$. Finding a predecessor of a node v in $\mathcal{P}$ is $O(logn)$. Once we located them, we perform a constant number of CUT and $LINK$ operations, which also take $O(logn)$.