COL 351 : Analysis & Design of Algorithms

Minor Exam

Sept 17, 2024 | Rohit Vaish

Problem 1

Problem 1 [20 points]
Suppose there are n agents and m items. The agents have *identical* valuations for the items, that is, for any $j \in \{1, 2, \ldots, m\}$, item j valued at v_j by all agents, where v_j is an integer (note that v_j can be negative). For $j \neq j'$, it is possible that $v_j \neq v_{j'}$.

The value of a set of items (i.e., a *bundle*) is the sum of items in that set. That is, for any bundle of items S, $v(S) := \sum_{j \in S} v_j$.

The goal is to partition the m items among the n agents in a *fair* manner. Denote an allocation by $A := (A_1, A_2, \ldots, A_n)$, where A_i is the subset of items assigned to agent i. We require that for any $i \neq k$, $A_i \cap A_k = \emptyset$ (i.e., items are not shared between bundles) and $\cup_i A_i$ is the entire set of items (i.e., no item is left unallocated). An allocation is deemed fair if, for any pair of agents i and k, the value of bundle A_i is "within an item" of the value of bundle A_k; specifically, for every pair of agents i and k:

- for every item $j \in A_i$ such that $v_j < 0$, we have $v(A_i \setminus \{j\}) \geqslant v(A_k)$, and
- for every item $j \in A_k$ such that $v_j \geqslant 0$, we have $v(A_i) \geqslant v(A_k \setminus \{j\})$.

Your task is to design a polynomial-time algorithm for computing a fair allocation.

(a) [**2 points**] Write a concise, high-level description of your algorithm using plain English with minimal notation (1-2 sentences).

Assign the items in nonincreasing order of absolute values.

At each step:

* if the item is non-negatively valued, assign it to the least happy agent;
* otherwise, if the item is negatively valued, assign it to the happiest agent.

(b) **[6 points]** Write your algorithm's pseudocode with clearly stated input and output. Define any additional notation used.

input : a set of n agents

a set of m items

the value v_j for each item j

output : a fair allocation A of the items among the agents

① Sort the items in nonincreasing order of absolute values.

Reindex the items so that $|v_1| \geq |v_2| \geq \cdots \geq |v_m|$.

(2) Initialize the allocation $A := (\underbrace{\emptyset, \emptyset, \ldots, \emptyset}_{n \text{ entries}})$.

(3) for $j = 1$ to m // recall that items are indexed so that $|v_1| \geq |v_2| \geq \cdots$

if $v_j < 0$

let $i^* := \arg\max_i v(A_i)$ // happiest agent

else

let $i^* := \arg\min_i v(A_i)$ // least-happy agent

$A_{i^*} \leftarrow A_{i^*} \cup \{j\}$

(4) return A

(c) [**6 points**] Prove the correctness of your algorithm, specifying the proof technique upfront (e.g., by induction, by contradiction, by case analysis, etc.).

To prove correctness, we need to show that A is

* a valid allocation (by direct argument)
* a fair allocation (by invariant and case analysis)

The output A is valid because the for-loop considers all items, and, in iteration j, item j is assigned to exactly one agent.

Let $A^{(j)}$ be the partial allocation maintained by our algorithm at the start of iteration j.

To prove fairness, we will show that for every $j \in \{1, 2, \dots, m+1\}$, $A^{(j)}$ is fair.

Consider any pair of agents h, ℓ.

Suppose $A^{(j)}$ is fair. We will show that $A^{(j+1)}$ is also fair.

Assume $v(A_h^{(j)}) \geq v(A_\ell^{(j)})$ without loss of generality.

Fairness between h and ℓ under $A^{(j)}$ implies:

* for any $g \in A_h^{(j)}$ s.t. $v_g \geq 0$, $v(A_h^{(j)} \setminus \{g\}) \leq v(A_\ell^{(j)})$, and
* " " $g' \in A_\ell^{(j)}$ " $v_{g'} < 0$, $v(A_\ell^{(j)} \setminus \{g'\}) \geq v(A_h^{(j)})$.

We will refer to any such g and g' as a certificate item.

Case I: If neither h nor ℓ receives an item in j^{th} iteration.

Then fairness is maintained between h and ℓ in j^{th} iteration as the certificate items are intact.

Case II : If h receives a negative-value item in j^{th} iteration.

Then, h must be the happiest agent before item j is assigned, i.e., $h \in \arg\max_i v(A_i^{(j)})$.

If h continues to have larger value than ℓ after j is assigned, (i.e, $v(A_h^{(j)} \cup \{j\}) \geq v(A_\ell^{(j)})$), then fairness is maintained as the certificate items in $A_\ell^{(j)}$ and $A_h^{(j)}$ are intact.

Otherwise, we have that $v(A_h^{(j)} \cup \{j\}) < v(A_\ell^{(j)})$.

In this case, item j in agent h's bundle is a certificate item because

$$v(A_h^{(j+1)} \setminus \{j\}) = v(A_h^{(j)}) \geq v(A_\ell^{(j)})$$

since h was the happiest agent at the start of iteration j.

Key idea: All negative value items in $A_h^{(j)}$ and all nonnegative value items in $A_\ell^{(j)}$ have at least as much absolute value as item j, and are therefore also certificate items.

Thus, fairness is maintained.

Case III: If ℓ receives a nonnegative-value item in j^{th} iteration.

A similar analysis as in Case II holds. If h has larger value than ℓ, then old certificates work. Otherwise, invoke absolute value selection rule.

Note that cases I, II, III are mutually exclusive and exhaustive.

Thus, fairness is maintained at the end of j^{th} iteration. ∎

(d) [**3 points**] Show that your algorithm runs in polynomial time.

Sorting the items takes $O(m \log m)$ time.

The for-loop runs for m iterations. In each iteration, searching for happiest / least happy agent takes $O(n)$ time.

Thus, $O(m \log m + mn)$ time overall, which is polynomial in the input size.

(e) **[3 points]** Suppose now there is an additional *quota* constraint, defined as follows: Let $q_1, q_2, \ldots, q_n$ be nonnegative integers that add up to m, i.e., $\sum_{i=1}^{n} q_i = m$. In addition to being fair, an allocation must also assign exactly q_i items to agent i.

Prove or disprove: Given any quotas, a fair allocation satisfying the quotas always exists. To prove this, you should present a polynomial-time algorithm. (If you are suggesting a modification to the algorithm in part (a), instead of writing the entire pseudocode, just emphasize the main difference between the two algorithms.) To disprove, you should present a counterexample.

There is a counterexample with two agents and two items.

$$v_1 = 1 \qquad v_2 = -1 \qquad q_1 = q_2 = 1$$

Dropping the first item brings its owner's value to 0, which is still greater than -1.

Problem 2

non negative

Problem 2 [20 points]

In the *longest path* problem, we are given a weighted directed graph $G = (V, E, w)$ and a vertex $s \in V$, and we are asked to find the longest simple path (i.e., no vertex is repeated) from s to every other vertex in G. For a general graph, it is not known if there is a polynomial-time algorithm to solve this problem. If we restrict G to be acyclic, however, this problem can be solved in polynomial time. Our task in this problem is to discover such an algorithm.

(a) [**8 points**] Write your algorithm's pseudocode with clearly stated input and output. Define any additional notation used.

Hint: You have seen an algorithm for the single-source shortest-paths problem. Can you use this algorithm (or a modification of it) on a (possibly modified) input to find the longest paths?

input: a directed acyclic graph $G = (V, E, w)$, a fixed vertex $s \in V$.

output: the longest path (if any) for s to every other vertex of G.

① Compute a topological ordering, say σ, of graph G.
Reindex the vertices so that $\sigma(v_i) = i$.

② // run a modified Dijkstra algorithm starting from s

$X := \{s\}$. Let r be such that $\sigma(s) = r$.

$A[s] := 0$ and $A[v_i] := \text{NULL}$ for all $i < r$.

$B[s] := \emptyset$ " $B[v_i]$ " " " " .

for $i = r+1$ to n // consider vertices in topological order

if there exists some edge (u, v_i) such that $u \in X$:

* pick (u^*, v_i) that maximizes $A[u^*] + w_{u^* v_i}$ where $u^* \in X$

* add v_i to X

* $A[v_i] := A[u^*] + w_{u^* v_i}$ and $B[v_i] := B[u^*] \cup (u^*, v_i)$.

else

$A[v_i] = \text{NULL}$, $B[v_i] = \text{NULL}$ // no path from s to v_i

return A, B

(b) **[3 points]** Briefly justify why your algorithm runs in polynomial time.

Let $m := |E|$ and $n := |V|$.

The topological ordering algorithm runs in $O(m+n)$ time.

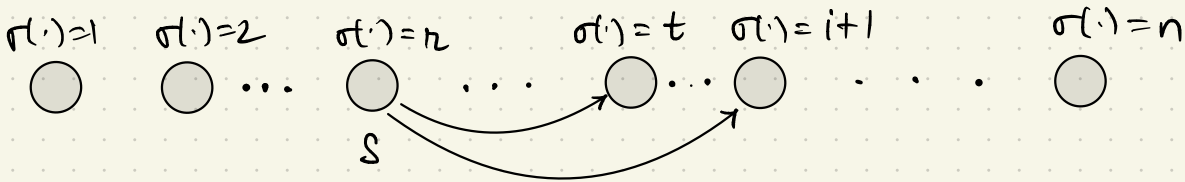
The modified Dijkstra algorithm runs $O(n)$ iterations of for-loop. In each iteration, the algorithm takes $O(m)$ time to find the edge (u^*, v_i).

Overall, our algorithm takes $O(n \cdot m)$ time, which is polynomial in input size.

(c) [**6 points**] Prove the correctness of your algorithm, specifying the proof technique upfront (e.g., by induction, by contradiction, by case analysis, etc.).

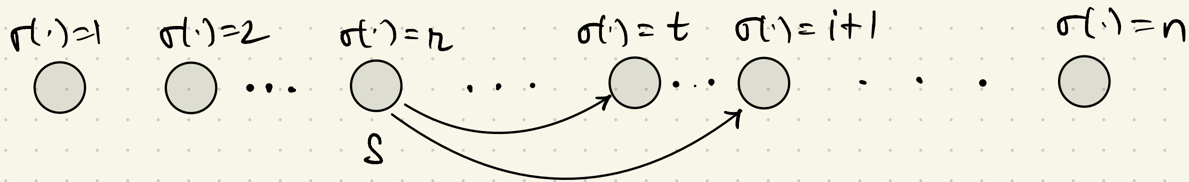
We will prove correctness by induction on the number of iterations.

Let r be such that $\sigma(s) = r$.



Key lemma: If there exists a simple path from s to a vertex v_{i+1}, then the last edge of this path must be of the form (v_k, v_{i+1}) where $r \le k < i+1$.

The lemma follows from the fact that σ is a topological ordering.



For the first $(r-1)$ vertices, our algorithm returns NULL. This is correct because, due to key lemma, there is no path from s to any of these vertices.

Suppose the algorithm encounters a nonempty frontier for the first time in the t^{th} iteration. Clearly, $t \geq r$.

Then, the edge (s, v_t) must be the unique path from s to v_t (by key lemma). We will treat this as the base case.

$\forall\ i \geq t$ $\quad P[i]$: $A[v_i]$ is the length of longest path from s to v_i
$B[v_i]$ is the longest path from s to v_i.

Consider the $(i+1)^{th}$ iteration. By key lemma, the last edge of any $s \rightsquigarrow v_{i+1}$ path (if one exists) must be a frontier edge of the form (u, v_{i+1}) where $u \in X$.

Thus, any path from s to v_{i+1} must be of the form

$$s \rightsquigarrow \underset{(u \in X)}{u} \longrightarrow v_{i+1}$$

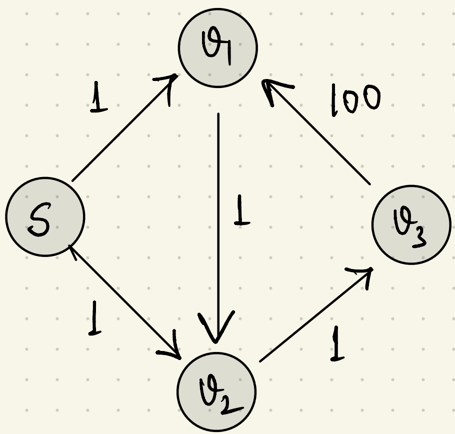
Length of such path = length of $s \rightsquigarrow u$ + length of (u, v_{i+1}).

By induction hypothesis, $B[u^*]$ is the longest path to u^*.

By greedy selection, $B[v_{i+1}]$ must be the longest path to v_{i+1}.

Similarly, $A[v_{i+1}]$ is the length of the longest path. $\boxtimes$

(d) [**3 points**] Explain, using a counterexample, why your algorithm does not work when G is not acyclic.



The topological ordering algorithm can return $\sigma = (s, v_1, v_2, v_3)$.

Thus, our algorithm will return the longest path to v_1 as $s \to v_1$.

However, the correct answer is

$$s \to v_2 \to v_3 \to v_1.$$

Problem 3

Problem 3 [20 points]
(a) **[5 points]** Let $T = (V, E)$ and $T' = (V, E')$ be two different spanning trees of a graph G. Prove that there exists an edge $e \in E \setminus E'$ and an edge $e' \in E' \setminus E$ such that both $T \cup \{e'\} - \{e\}$ and $T' \cup \{e\} - \{e'\}$ are spanning trees of G.

Let $e = \{u, v\}$ be any edge in $E \setminus E'$.

Since T' is a spanning tree, there must be a path between u and v in T'. Thus, $T' \cup \{e\}$ creates a cycle, say C'.

Since T is also a spanning tree, there must exist some edge $e' \in C' \setminus T$.

We will show that $T^* := T' \cup \{e\} \setminus \{e'\}$ is also a spanning tree.

Note that T^* has $n-1$ edges where $n = |V|$.

① T^* is connected because any walk between pair of vertices x, y that goes through e' in $T' \cup \{e\}$ can be re-routed through the vertices in $C' \setminus \{e'\}$ in T^*.

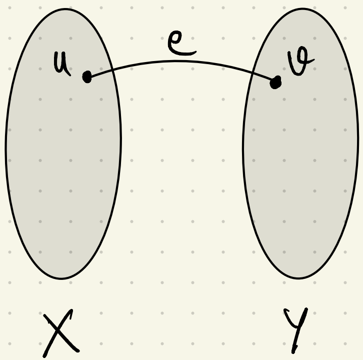
② T^* is spanning because $T' \cup \{e\}$ is a spanning subgraph of G. Removing e' maintains connectedness, thus T^* must be spanning.

③ T^* is a connected graph on n vertices and has $n-1$ edges. So, it must be a tree and thus is acyclic. $\boxtimes$

NOTE: The above proof does not show that $T \cup \{e'\} \setminus \{e\}$ is a spanning tree because adding e' to T could lead to removal of a different edge. See next page for a different proof.

Consider any $e \in T \setminus T'$. Say $e = \{u, v\}$.

Then, $T \setminus \{e\}$ is a forest with two trees. Call their vertex sets X and Y.

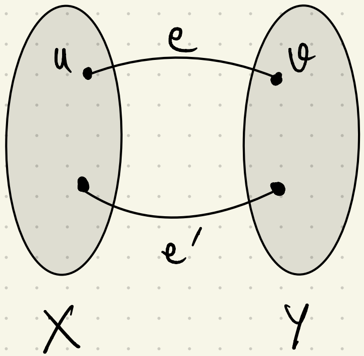


Edge e must be a crossing edge for the cut (X, Y) in graph G. Thus, $u \in X$ and $v \in Y$.

Since T' is spanning tree and $e \notin T'$, $T' \cup \{e\}$ must contain a cycle, say C, such that $e \in C$.

By double crossing lemma, there exists an edge $e' \in C$ such that e' crosses the cut (X, Y) and $e' \neq e$.

We will now show that $T \cup \{e'\} \setminus \{e\}$ and $T' \cup \{e\} \setminus \{e'\}$ are spanning trees.



Claim 1: $T \cup \{e'\} \setminus \{e\}$ is a spanning tree.

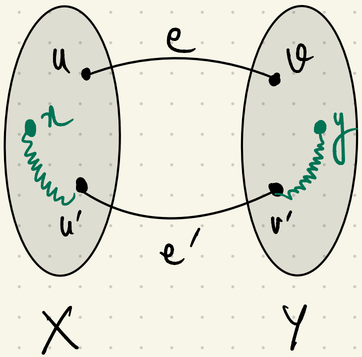
Proof: Let $\overline{T} = T \cup \{e'\} \setminus \{e\}$.

In $T \setminus \{e\}$, all vertices within X (respectively, within Y) are connected. Since e' is a crossing edge, it connects some vertex in X with some vertex in Y, thereby connecting the sets X and Y. Thus, $\overline{T}$ is connected.

$\overline{T}$ is spanning because it connects every vertex in G to every other vertex in G.

Since $\overline{T}$ has $n-1$ edges and is connected, it must be a tree. ▨

Claim 2: $T' \cup \{e\} \setminus \{e'\}$ is a spanning tree.



Proof: Let $T^* := T' \cup \{e\} \setminus \{e'\}$.

Consider any pair of vertices $x \in X$ and $y \in Y$ such that the unique path between x and y in T' includes e'. We will argue that x and y are connected under T^*. This would prove that T^* is connected and has the spanning property. Furthermore, since T^* has $(n-1)$ edges, we would obtain that T^* is a tree, and therefore a spanning tree.

Recall that $T' \cup \{e\}$ contains a cycle C such that $e, e' \in C$.
Thus, any walk between x and y in $T' \cup \{e\}$ that goes through e' can be re-routed to exclude e'.

This implies that there must be a path between x and y in $T' \cup \{e\}$ that excludes e'.

Therefore, x and y are connected in T^*. $\blacksquare$

(b) **[15 points]** Let $G = (V, E)$ be an undirected, unweighted, and connected graph with each edge colored either red or blue. Design an $\mathcal{O}(|E| \cancel{\log} |V|)$ algorithm that, given as input an integer k and the graph G, determines whether there exists a spanning tree of G that contains exactly k red edges. Write the pseudocode of your algorithm, and provide a brief justification of its correctness and running time.

input: graph $G = (V, E)$ with red / blue edges, integer k

output: Does there exist a spanning tree of G with exactly k red edges?

① Assign weight 0 to all red edges and weight 1 to all blue edges. Compute MST T^{max} of this graph. Let $m^{max} :=$ # red edges in T^{max}.

② Assign weight 0 to all blue edges and weight 1 to all red edges. Compute MST T^{min} of this graph. Let $m^{min} :=$ # red edges in T^{min}.

③ return NO if $k < m^{min}$ or $k > m^{max}$.

④ Initialize $T := T^{min}$

for every red edge $e \in T^{max}$

- if $e \notin T$
  - * add e to T
  - * remove an edge e' from the induced cycle such that $e' \notin T^{max}$. // e' must exist
  - * update T // $T \cup e \setminus \{e'\}$ is spanning tree due to Part (a).
- if # red edges in $T = K$
  - return YES

return No

Correctness: T^{max} and T^{min} are spanning trees of G with the maximum and minimum possible number of red edges.

If $m^{min} \le K \le m^{max}$, our algorithm starts with T^{min} and incrementally turns it into T^{max} until K red edges are achieved.

Note: # red edges may not strictly increase after each iteration.

Running time: Finding T^{max} and T^{min} via Prim's or Kruskal's algorithm takes $O(|E| \log |V|)$ time. The for-loop performs $O(n)$ iterations. In each iteration, identifying the edge e' takes $O(m)$ time.

Overall, the algorithm takes $O(mn)$ time.