INDIAN STATISTICAL INSTITUTE

End Semestral Examination

M. Tech (CS) - I Year, 2018-2019 (Semester - II)

Design and Analysis of Algorithms

Date: 18.04.2019 Maximum Marks: 100 Duration: 4.0 Hours

Note: Read the instructions very carefully.

This is a 4-page question paper. The question paper has **6** questions in Group-A and **9** questions in Group-B, each of 12 marks. You can answer at most **4** questions from Group-A and the maximum you can score in Group-A is 40. You can answer at most **6** questions from Group-B and the maximum you can score in Group-B is 60. Read the questions carefully to choose the questions you want to answer.

Mark explicitly in the first page of your answer script the questions that you have attempted. Only those answers will be checked.

If you design an algorithm, compute its space and time complexity, and give its proof of correctness.

Whenever we say that, $\mathcal{P}$ is a linear program (LP), we mean $\mathcal{P}$ is of the form

$$\begin{array}{lllll} \text{Maximize} & \mathbf{c}^T\mathbf{x} & & \\ \text{subject to} & A\mathbf{x} & \leq & \mathbf{b} \\ & \mathbf{x} & \geq & \mathbf{0} \end{array}$$

Group A

(QA1) Let A be a set of n elements. The *Union-Find* data structure allows us to maintain disjoint sets by building a forest-like data structure τ of trees for supporting two set operations – FIND(a) that returns the name of the set containing a, $\forall a \in A$; and UNION(A, B) is an operation that takes two sets A and B and merges them into a single set.

(i) Describe the union-by-rank and path-compression heuristic algorithm for building τ for Union-Find.

(ii) Show that for any node x in τ, rank$(x) <$ rank$(\Gamma(x))$, where $\Gamma(x)$ denotes the parent of x in τ.

(iii) Show that any root node of rank k has at least 2^k nodes in its tree.

(iv) Show that there can be at most $\frac{n}{2^k}$ nodes of rank k.

(v) Show that if there are m FIND operations using the path-compression heuristic, the total number of operations is at most $O(m \log^* n)$, where $\log^* n$ is defined to be the number of successive $\log$ operations that need to be applied to n to bring it down to less than or equal to 1.

[4+1+2+1+4=12]

(QA2) Given two vectors $\mathbf{a} = (a_0, a_1, \ldots, a_{n-1})$ and $\mathbf{b} = (b_0, b_1, \ldots, b_{n-1})$, define the convolution of $\mathbf{a}$ and $\mathbf{b}$ as the vector $\mathbf{c} = \sum_{(i,j):i+j=k; i,j<n} a_i b_j$. Design an $O(n \log n)$ time algorithm to compute $\mathbf{c}$, by showing the following steps explicitly: (i) the algorithm; (ii) time complexity and (iii) proof of correctness. [5+2+5=12]

(QA3) Let $\sigma = \langle a_1, \ldots, a_i, \ldots, a_m \rangle$ be a stream of m elements in the streaming model, where each $a_i \in \{1, 2, \ldots, n\}$. Let f_{a_i} denote the number of occurrences of item a_i in the stream σ. Consider the Frequency Estimation problem in the *streaming model* stated as follows. Given an error parameter $0 \leq \epsilon \leq 1$, the task in Frequency Estimation problem is to maintain for each item a_i in σ, an estimate $\widehat{f}_{a_i}$, such that $f_{a_i} - \epsilon \cdot m \leq \widehat{f}_{a_i} \leq f_{a_i}$.

(i) Design an algorithm to solve the Frequency Estimation problem.

(ii) Estimate the (extra) space taken by your streaming algorithm.

(iii) Prove that your algorithm returns an estimate within the stated bounds, for each $a_i \in \sigma$.

[5+2+5=12]

(Ans:) In the class, we proved $f_{a_i} - \frac{m}{K} \leq \widehat{f}_{a_i} \leq f_{a_i}$, where K was the extra space or number of counters used. Here, the answer is asked in terms of an error parameter ϵ. So, use $\lceil \frac{1}{\epsilon} \rceil$ counters and everything else remains same as the Misra-Gries algorithm discussed in class.

(QA4) Let $G = (V, E)$ be a *flow network* with a *source vertex* $s \in V$ and a *sink vertex* $t \in V$. Each edge $e \in E$ has a positive integral capacity value c_e.

(i) Describe the *maximum flow* problem in G by defining the *capacity* and *conservation* conditions.

(ii) State and prove the *max-flow min-cut* theorem.

(iii) Show how the Ford-Fulkerson algorithm can be used to solve the problem of *bipartite matching* in polynomial time.

[2+5+5=12]

(QA5) (i) Define the Hamiltonian Cycle problem.

(ii) Show that the Hamiltonian Cycle problem is NP-complete.
[**Hints:** You can try a reduction from 3SAT.] [2+10=12]

(QA6) Answer the following: (i) Define *polynomial reducibilty*; (ii) Define the class NP; (iii) Define the class NP-complete; (iv) Define the class co-NP;

(v) Show that the problem of Linear Program belongs to the class NP $\cap$ co-NP;

(Ans:) This question can have a cheeky answer! We know that LP $\in$ P (this was mentioned in class). Why? Not because of SIMPLEX (as it can be exponential!), but because of Khachiyan or Karmarkar's polynomial time algorithms that were proved around 1970s and '80s. Now, as we know that P $\subseteq$ NP $\cap$ co-NP, so LP $\in$ NP $\cap$ co-NP.

But what about a student answering this question before 1970s? Can (s)he not prove this result? Yes, one can do it using duality. The *decision problem* of LP is "Does there exist a $\mathbf{x} \in \mathbb{R}^n$ such that $A\mathbf{x} \leq \mathbf{b}$, $\mathbf{x} \geq \mathbf{0}$, such that $\mathbf{c}^T\mathbf{x} \geq \alpha$?" It is easy to show that LP $\in$ NP. To show that LP $\in$ co-NP, we need to show that $\overline{\text{LP}} \in$ NP. This we can do using the dual LP.

(vi) Suppose Π is an NP-complete problem. Show that Π is polynomially solvable if and only if P=NP.

[2 $\times$ 6 = 12]

Group B

(QB1) (i) Use SIMPLEX method to solve the following LP:

$$\begin{aligned} \text{Maximize} \quad & x_1 + 2x_2 - x_3 \\ \text{subject to} \quad & 2x_1 + x_2 + x_3 \leq 14 \\ & 4x_1 + 2x_2 + 3x_3 \leq 28 \\ & 2x_1 + 5x_2 + 5x_3 \leq 30 \\ & x_1, x_2, x_3 \geq 0 \end{aligned}$$

Ans: The optimal vector is $(5, 4, 0)$ and the objective function value is 13.

(ii) Deduce the conditions on A, $\mathbf{b}$ and $\mathbf{c}$ so that the primal LP $\mathcal{P}$ and its dual $\mathcal{D}$ are the same LP.

(Ans:) A should be skew symmetric, i.e., $A + A^T = 0$, and $\mathbf{b} = -\mathbf{c}$.

[8+4=12]

(QB2) In the *set cover* problem, we have a universe $\mathcal{U} = \{u_1, \ldots, u_n\}$ of n elements. Let $\mathcal{S} = \{S_1, \ldots, S_m\}$ be a set of m sets, where each set $S_i \subseteq \mathcal{U}$. Each set S_i has a weight $w_i \geq 0$. The problem in *set cover* is to find a minimum weight collection of subsets of $\mathcal{S}$ that covers all elements of $\mathcal{U}$.

(a) Write an integer linear program (ILP) for the *set cover problem* using decision variables x_i to indicate whether the set S_i is included in the solution or not.

(b) Relax the above ILP and round the optimal solution of the linear program as follows: given the optimal solution $\mathbf{x}^*$ of the linear program, we include the subset S_i in our solution if and only if $x_i^* \geq \frac{1}{f}$, where f is the maximum number of sets in which any element appears and x_i^* is the i-th component of $\mathbf{x}$.

For this rounding scheme, show that the set generated is a set cover and is an f-factor approximation algorithm. [4+8=12]

(Ans:) (a) In order to ensure that every element u_i is covered, it must be the case that at least one of the subsets S_j containing u_i is selected, i.e., $\sum_{S_j : u_i \in S_j} x_j \geq 1$ for each u_i. Let $x_j = 1$ if S_j is included in the solution; otherwise, $x_j = 0$. The resulting ILP is:

$$\begin{aligned} \text{Minimize} \quad & \sum_{j=1}^{m} w_j x_j & & & \\ \text{subject to} \quad & \sum_{S_j : u_i \in S_j} x_j & \geq & 1 & \forall u_i \in \mathcal{U} \\ & x_j & \in & \{0, 1\} & \end{aligned}$$

(b) The ILP is relaxed by replacing $x_j \in \{0, 1\}$ by $x_j \geq \{0, 1\}$. Let the optimal values of the ILP and the relaxed LP are Z^{ILP} and Z^{LP}. Clearly, $Z^{LP} \leq Z^{ILP}$. Let x^* denote the optimal solution of the LP. The rounding scheme is as follows: We include the set S_j in optimal solution if $x_j^* \geq \frac{1}{f}$ where f is the maximum number of sets in which any element appears. Let I denote the indices j of the subsets in this solution. We round the fractional solution x_j^* to an integer solution $x^{'}$ by setting $x_j^{'} = 1$ if $x_j^* \geq \frac{1}{f}$, and $x_j^{'} = 0$, otherwise.

We call an element u_i is covered if this solution contains some subset containing u_i. Because the optimal solution x^* is a feasible solution to the linear program, we know that $\sum_{S_j : u_i \in S_j} x_j^* \geq 1$ for element u_i. So at least one term must be at least $\frac{1}{f}$. Therefore, $j \in I$ and the element u_i is covered. Hence, the collection of subsets $S_j, j \in I$ is a set cover.

It is clear that the algorithm runs in polynomial time. By our construction, $1 \leq f.x_j^*$ for each $j \in I$. From this and the fact that each term $f.w_j.x_j^*$ is nonnegative for $j = 1, 2, \ldots, m$, we see that

$$\sum_{j \in I} w_j \leq \sum_{j=1}^{m} w_j.(f.x_j^*) = f \sum_{j=1}^{m} = f.Z^{LP} \leq f.Z^{ILP}.$$

(QB3) Consider the *vertex cover* problem, where the input is a graph $G = (V, E)$ and a positive integer k. The problem is to find if there exists a set $V'(\subseteq V)$ of at most k vertices such that all edges in G are incident to vertices in V'. Now, suppose it is known that $k = O(\log_2 n)$. Describe an efficient algorithm to find if G has a vertex cover of size at most k. Provide a proof of correctness and analyze the time complexity. [5+4+3=12]

(Ans:) Use the $O(2^k n)$ or $O(n^2 + 2^k k^2)$ algorithm studied in the class. Now, as $k = O(\log_2 n)$, $2^k = O(n)$. So, we have a polynomial time algorithm for this version of the vertex cover problem.

(QB4) In the k-center problem, the input is a set of points $P = \{p_1, \ldots, p_n\}$ in $\mathbb{R}^2$ and a positive integer $k \leq n$; the distance between two points $p_i, p_j \in P$ is measured using the usual ℓ_2 distance. The output of the problem is a *partition* of P into k clusters $C_1, C_2, \ldots, C_k$ such that the diameter of the clusters is minimized. The diameter of the clusters is defined as

$$\max_j \max_{p_a, p_b \in C_j} d(p_a, p_b)$$

where $d(p_a, p_b)$ denotes the ℓ_2 distance between p_a and p_b.

Show that there is no ρ-approximation algorithm for the k-centre problem for $\rho < 2$ unless NP=P.

[Hints: You can prove this by using a reduction to the *Dominating Set* problem. In the *Dominating Set* problem for an undirected graph $G = (V, E)$, we want to find if there exists a set $D \subseteq V$ of size at most k so that every vertex $v \in V \setminus D$ has a neighbor in D.] [12]

(Ans:) Given an instance $< G, k >$ of *dominating set*, we define an instance $< G' = (V', E'), k >$ of the k-center problem as follows. $V' = V$ and E' is a complete graph on V'. The weights $w(e')$ on each edge $e' \in E'$ is given as follows —

$$w(e') = \begin{cases} 1 & \text{if } e' = (u, v) \text{ such that } (u, v) \in E \text{ i.e., } (u, v) \text{ was an edge in } G; \\ 2 & \text{otherwise} \end{cases}$$

That is the distance between adjacent vertices is set to 1 and between the non-adjacent vertices, it is set to 2. Notice that the distances are set such that it is a valid k-center instance.

Observation 1 *There is a dominating set of size k in G if and only if the optimal radius of the k-center problem in G' is* 1.

This is easy to see as the vertices in the dominating set D serve as the k-centers and vice-versa.

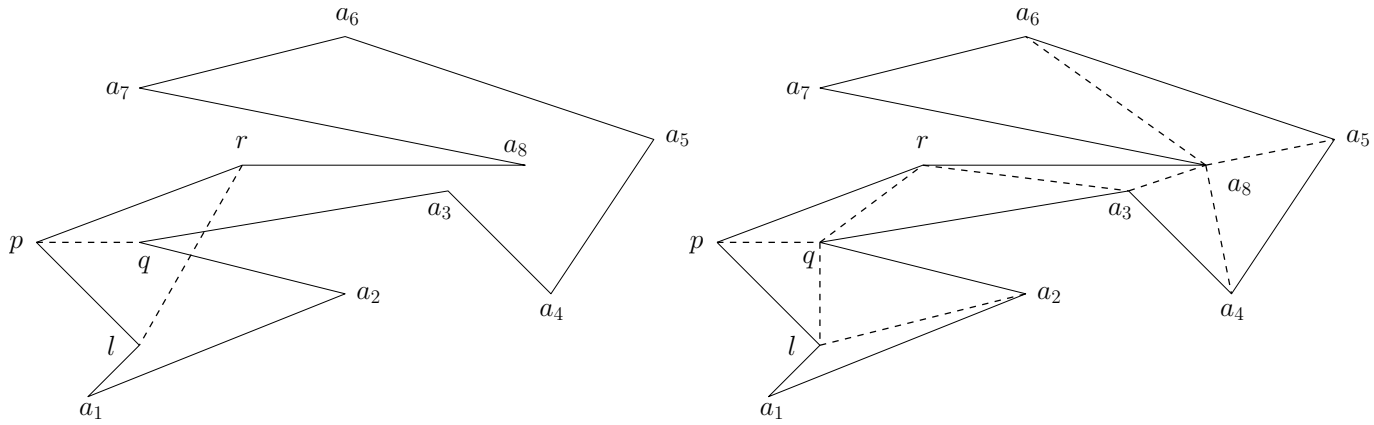
Any solution to the k-center problem of radius less than 2 on G' is actually a solution with radius 1. Now if there exists a ρ-approximation algorithm $\mathcal{A}$ for the k-centre problem for $\rho < 2$, run $\mathcal{A}$ on G'. We will get a k-center solution of G' with radius value strictly less than 2 – this implies a solution with radius 1. Now use Observation 1 to get an optimal solution to the dominating set problem. $\mathcal{A}$ takes polynomial time, so dominating set is solved in polynomial time and hence, P = NP.

(QB5) Let P be a simple polygon (edges of the polygon intersect only at vertices, no holes inside the polygon) of n vertices; the vertices are given in a clockwise order. Your problem is to design and analyze the time complexity of an efficient algorithm to compute the area of P. You can either design an algorithm of your own, or read the text below to build towards the solution.

The solution consists of *triangulating* P, i.e., breaking P into triangles by drawing non-crossing diagonals. A diagonal of P is a line segment $\overline{pq}$ that stays completely inside P, where p and q are vertices of P. See Figure 1 and read the caption for hints of the solution. The diagonal $\overline{pq}$ divides P into two simple polygons – $pra_8a_7a_6a_5a_4a_3qp$ and pqa_2a_1lp. All the dotted lines in Figure 1(b) are non-crossing diagonals of P.

Now, answer the following sub-questions to build towards the solution.

- Design an algorithm to compute a diagonal of P that divides P into two simple polygons.
- If P is divided into two smaller simple polygons, can you recurse to find the triangulation of P, and hence, its area?
- To estimate the time complexity, one needs to find the number of non-crossing diagonals and the number of triangles in P that the diagonals generate. Show, using induction or otherwise, that the number of non-crossing diagonals and the number of triangles in P is $n - 3$ and $n - 2$, respectively. [12]



(a) $\overline{pq}$ is a diagonal, but $\overline{rl}$ is not. (b) A triangulation of P using diagonals.

Figure 1: A simple polygon P, its diagonals and a triangulation. If there was no segment of the polygon cutting the line segment $\overline{rl}$, then $\overline{rl}$ is a diagonal. How do you identify points like p, r and l? If $\overline{rl}$ is not a diagonal, then you have to identify a point q inside $\triangle plr$ such that $\overline{pq}$ is a diagonal. There can be many points of the polygon P inside $\triangle plr$. From them, how do you identify the point q?

(Ans:) As mentioned in the question, the idea is to triangulate P and compute the area of each triangle and adding those areas up to get the total area of P.

How to get a diagonal? We do that by characterizing the points p, r and l. The algorithmic steps are as follows.

Get the leftmost vertex of P, i.e., the vertex with the least x-coordinate in P – call it p. Now from the clockwise order in which P is presented, pick up the preceding (l) and succeeding (r) vertices of p in P in $O(1)$ time. We have to now test if $\overline{rl}$ qualifies to be a diagonal. Check if $\overline{rl}$ is intersected by all other edges ($O(n)$ of them) of P – if none intersects, then $\overline{rl}$ is a diagonal; else, look at the leftmost point of P inside $\triangle plr$, call the vertex q, $\overline{pq}$ is surely a diagonal as no edge of P can cut it. So, in $O(n)$ time we have found a diagonal.

Once we find a diagonal, we break P into two simple polygons and can recurse. The question is how long will this continue. Let P be broken into two simple polygons P_1 of n_1 vertices and P_2 of n_2 vertices, both inclusive of the vertices forming the diagonal. What is the relation between n_1, n_2 and n: $n_1 + n_2 - 2 = n$. By induction hypothesis, P_1 will have $n_1 - 3$ non-crossing diagonals triangulating P_1 into $n_1 - 2$ triangles and P_2 will have $n_2 - 3$ non-crossing diagonals triangulating P_2 into $n_2 - 2$ triangles. So, the number of non-crossing diagonals in P is $(n_1 - 3) + (n_2 - 3) + 1$; the last added 1 is because of the diagonal that divided P. Completing the calculation for the non-crossing diagonals, we have $(n_1 - 3) + (n_2 - 3) + 1 = (n_1 + n_2) - 5$ $= (n + 2) - 5 = n - 3$. Similarly, the number of triangles in P will be $(n_1 - 2) + (n_2 - 2) = (n_1 + n_2) - 4$ $= n - 2$. Think over this result! Whatever the shape of the polygon may be, the number of diagonals and triangles is an invariant, i.e., it depends only on n.

To complete the time complexity calculation, we see that $n - 3$ diagonals are to be drawn in all and to draw each diagonal, we need $O(n)$ time. Thus, the total time taken is $O(n^2)$.

Though you will get full credit for an $O(n^2)$ answer, one can show by using sophisticated computational geometric techniques that this problem can be solved in $O(n)$ time.

(QB6) Consider a 2SAT expression involving n variables and m clauses where each clause is a disjunction of exactly two literals. Show that 2SAT is polynomially solvable.

[**Hints:** Can you reduce it to a path finding problem in a suitably constructed graph?] [12]

(Ans:) Consider a graph $G = (V, E)$ where V denotes the set of $2n$ literals (variables and their negated forms).

How do we now create the edge set E?

Consider a clause $(x \vee y)$ that we want to make TRUE in a 2SAT expression F — suppose x is not TRUE, then y must be true and if y is not TRUE, then x must be true. Thus $(x \vee y) \equiv (\overline{x} \Rightarrow y) \equiv (\overline{y} \Rightarrow x)$

These implications and hence relations can be encoded as edges of G. So, add the directed edges $(\overline{x}, y)$ and $(\overline{y}, x)$ to E if and only if there exists a clause $(x \vee y)$ in F. To convince yourself, draw the graph for $F = (\overline{x} \vee \overline{z}) \wedge (y \vee \overline{z}) \wedge (\overline{x} \vee z) \wedge (y \vee z)$.

The size of G is thus polynomial in n, the number of variables and m, the number of clauses. We now need to relate the satisfiability or unsatisfiability of a 2SAT expression with existence of paths in the graph. The precise relation is as follows.

Observation 2 *F is unsatisfiable if and only if there exists a variable x such that there exist both the paths x to $\overline{x}$ and $\overline{x}$ to x in G.*

One can prove the above by using the fact that

Observation 3 *If G contains a path from u to v, it should also contain a path from $\overline{v}$ and $\overline{u}$.*

To write a proof for Observation 2 using contradiction, assume that there exist paths $\overline{x} \rightsquigarrow x$ and $x \rightsquigarrow \overline{x}$ and F is satisfiable. First observe that if there is a directed edge $u \rightarrow v$, then there exists a clause $(\overline{u} \vee v)$ in F. The edge from u to v implies that if u is TRUE, then v has to be TRUE to make their corresponding clause $(\overline{u} \vee v)$ to be TRUE.

Break the path $x \rightsquigarrow \overline{x}$ as $x \rightsquigarrow p \rightarrow q \rightsquigarrow \overline{x}$ where from x to p all are TRUE and from q to $\overline{x}$ all are FALSE. Now consider the edge (p, q). We should have a clause $(\overline{p} \vee q)$ in the clause F, but as $(\overline{p} \vee q)$ is FALSE, F is also now FALSE and we have a contradiction.

In the above proof we started with x to be TRUE. Do the same with x as FALSE and we are done.

As to the algorithm, run a strongly connected component algorithm in the directed grapg G and look at each component. If it has both x and $\overline{x}$ in it, then we say that F is unsatisfiable; else we report that F is satisfiable. There are $2n$ vertices and $2m$ edges, thus the running time is bounded by $O(n + m)$.

(QB7) Your input is a set of n points $P = \{p_1, \ldots, p_n\},\ p_i \in \mathbb{R}^2$, and a positive integral parameter $k \leq n$. We define a k-clustering of P to be a *partition* of P into k non-empty sets $C_1, \ldots, C_k$, such that the *spacing*, defined to be the minimum distance between any pair of points lying in different clusters/partitions, is maximized. The *spacing* between two clusters C_i and C_j, $i \neq j$, is

$$\min_{p_a \in C_i, p_b \in C_j, i \neq j} d(p_a, p_b)$$

where $d(p_a, p_b)$ denotes the usual ℓ_2-distance between p_a and p_b. Thus the *spacing* of a k-clustering is

$$\min_{1 \leq i < j \leq n} \quad \min_{p_a \in C_i, p_b \in C_j} d(p_a, p_b)$$

Design an algorithm for computing a k-clustering of P. Analyze its time complexity. Provide a proof of correctness.

[**Hints:** Do not allow yourself to be fooled into thinking that it is only a geometric problem. You can also think graphs and consider the fact that minimum distances are to be maximized! You can think in terms of a polynomial solution by constructing a particular graph structure.] [5+2+5=12]

Ans: Let each point in the plane be a vertex; the weight among pairs of vertices is the Euclidean distance between the points. On this graph G, run any minimum spanning tree algorithm to form a tree T on n vertices. Now, to get a k-clustering, we need to form k components. This is obtained by just deleting the most expensive $k - 1$ edges from T. Alternately, one can start building the MST by using the Kruskal's algorithm and stop the moment $n - k$ edges have been added to generate k connected components.

The time complexity is thus dominated by the MST algorithm. The number of edges is $O(n^2)$ in G; so the time complexity is $O(n^2 \log n)$.

Consider the clustering $\mathcal{D} = D_1, \ldots, D_k$ generated by the algorithm stated above. What is the spacing it has generated? It is the weight of the $(k-1)$-th most expensive edge of T. Let it be w^*. Now, assume any other k-clustering and argue that the *spacing* generated is at most w^*.

Consider some other k-clustering $\mathcal{C}$ that partitions P into non-empty clusters $C_1, C_2, \ldots, C_k$. We argue that the *spacing* of $\mathcal{C}$ is also at most w^*.

Since $\mathcal{D}$ and $\mathcal{C}$ are not the same, there should exist a cluster $D_r \in \mathcal{D}$ that is not a subset of any of the k sets in $\mathcal{C}$. So, there are points $p_i\, p_j \in D_r$ that belong to different clusters in $\mathcal{C}$. Let these clusters in $\mathcal{C}$ be C_k and C_l, $k \neq l$ such that $p_i \in C_k$ and $p_j \in C_l$.

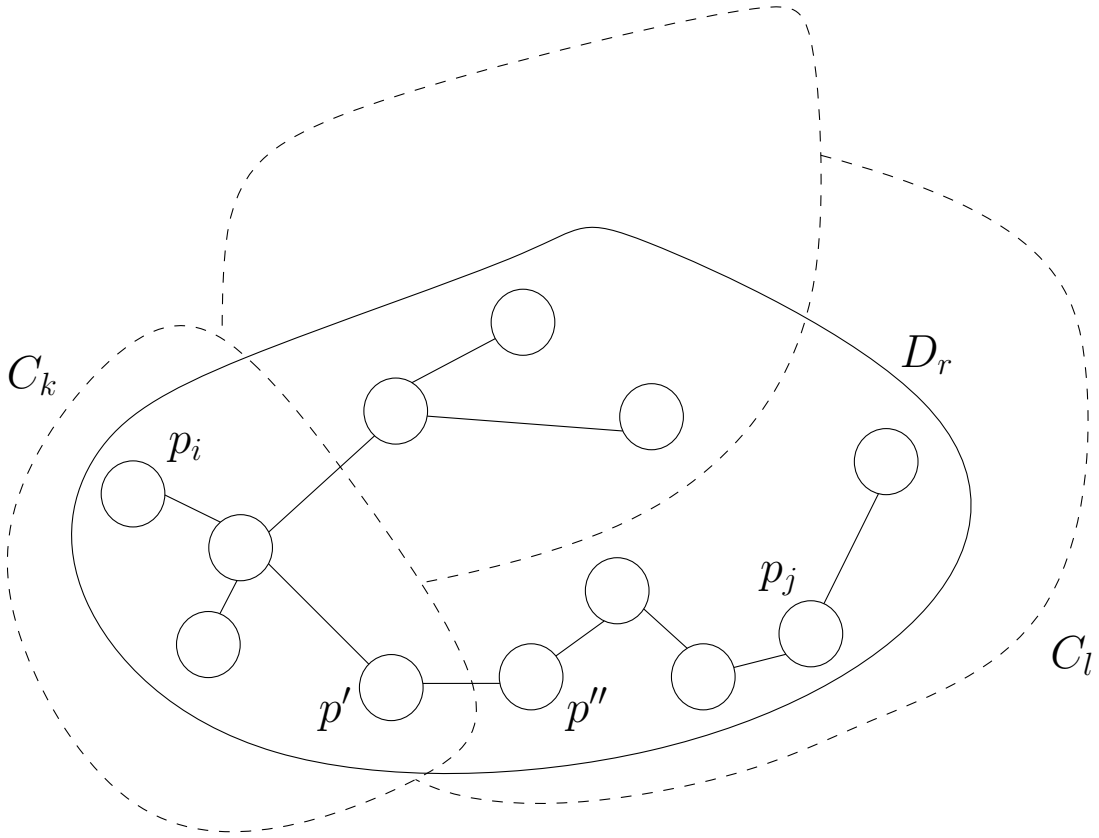


Figure 2: Illustration for the proof.

The algorithm created the cluster D_r which is a tree among the forests, so there is a path from p_i to p_j in D_r and all the edge weights in the path from p_i to p_j is at most w^*. Look at the path $p_i \rightsquigarrow p_j$ from p_i to p_j in D_r with respect to the clusters C_k and C_l. Consider the path $p_i \rightsquigarrow p_j$ in cluster D_r as it crosses over from C_k to C_l, as $p_i \rightsquigarrow p' \rightarrow p'' \rightsquigarrow p_j$. What is the distance between p' and p''? As the path $p_i \rightsquigarrow p_j$ was generated as part of the MST building algorithm, all edge weights (here, distances) are at most w^*, and hence distance between p' and p'' is also at most w^*. Now, p' and p'' belong to two different clusters in the clustering $\mathcal{C}$, and hence, the spacing of clustering $\mathcal{C}$ is at most w^*. This is what we set out to prove!

(QB8) Let $T = (V, E)$ be a tree with $|V| = n$; each vertex $v \in V$ has a weight $w_v > 0$. The *maximum weight independent set* problem is to find an *independent set* S in T such that the total weight $\sum_{v \in S} w_v$ is maximized. Design and analyze an efficient algorithm for the above problem. Prove its correctness.

[**Hints:** Can you locate overlapping subproblems and use dynamic programming? If you use dynamic programming, you have to think of two issues — (i) for a vertex $u \in V$, think whether u will/will not be in the optimum solution. If u is in the optimum solution, none of its neighbor can be in the optimum solution; (ii) in

which order would you handle the vertices; which vertex do you start from and which vertex you end at, i.e., what is your base condition?] [8+4=12]

(Ans:) For a dynamic programming solution, we need to fix two issues – the subproblems and the order in which we consider the subproblems. For a vertex $v \in V$, either v belongs to the optimum solution or not. If v belongs to the optimum solution, then none of its neighbors belong to the optimum solution and if v does not belong to the optimum solution then the optimum solution at v will be determined by its neighbors.

Now about the order in which we consider the subproblems. It makes sense to deal with leaves and then move up. So, we fix any node r apart from a leaf as the root and then consider a post-order traversal starting from r – this would compute all optimal solutions for neighbors of r before computing the optimal solution at r. In this scheme, we need to define a subtree rooted at any node v, let such a subtree be denoted as T_v. So, we will keep computing optimal solutions in different subtrees T_v and move upto computing the optimal solution in T_r. The postorder fashion gives us the order in which we build towards the optimal solution.

To consider the two situations whether v belongs to the optimal solution or not, let $\text{OPT}_{\text{include}}(v)$ denote the case where v is in the optimal solution and $\text{OPT}_{\text{exclude}}(v)$ denote the case where v is not in the optimal solution. Let us first fix the base cases, i.e. for the leaves.

Let $v \neq r$ be a leaf. Then, $\text{OPT}_{\text{include}}(v) = w_v$ and $\text{OPT}_{\text{exclude}}(v) = 0$.

For a non-leaf node v (a node that has children), if v is in the optimal solution, then we can write

$$\text{OPT}_{\text{include}}(v) = w_v + \sum_{u \in \text{child}(v)} \text{OPT}_{\text{exclude}}(u)$$

First of all the parent-children relation between v and u is determined from the post-order traversal of the tree starting from the root r. The above equality makes sense because if v is in the optimal solution, then its children cannot be.

Similarly, for a non-leaf node v, if v is not in the optimal solution, then we can write

$$\text{OPT}_{\text{exclude}}(v) = \sum_{u \in \text{child}(v)} \max\left(\text{OPT}_{\text{include}}(u), \text{OPT}_{\text{exclude}}(u)\right)$$

We continue computing according to the above recurrences and the final answer is

$$\max\left(\text{OPT}_{\text{include}}(r), \text{OPT}_{\text{exclude}}(r)\right)$$

.

(QB9) The problem is about scheduling students to work during the vacation period!

The specific problem is about finding a schedule for n students during m vacation periods where each vacation period consists of a certain number of consecutive days. Let D_j be the set of days included in the vacation period j, $1 \leq j \leq m$. So, $\bigcup_j D_j$ is the union of all these vacation days. Design an algorithm to prepare the work schedule keeping the following constraints under consideration:

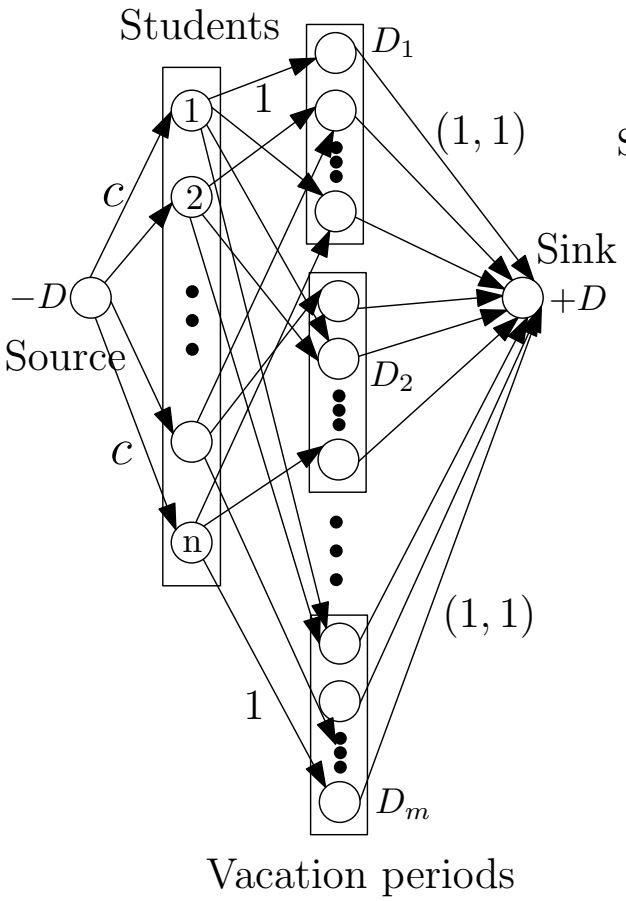
(a) A student i, $1 \leq i \leq n$, has a set of vacation days S_i when he is available for work. S_i's will obviously be spread across D_j's.

(b) Each student should be assigned to work at most c vacation days in total among the days in which (s)he is available.

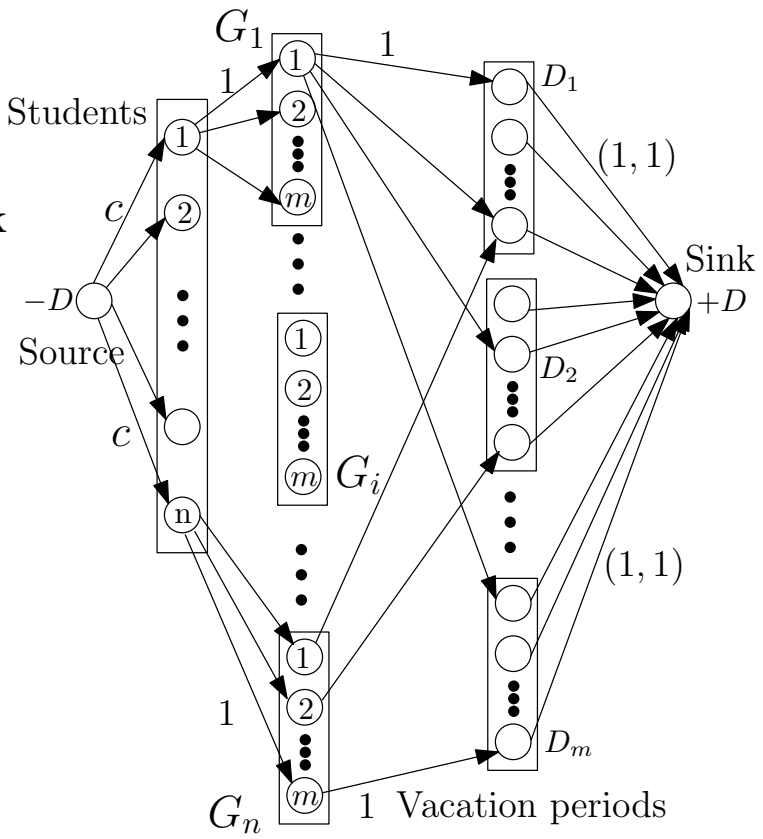
(c) For each vacation period j, each student should be assigned to work at most one of the days in the set D_j.

[**Hints:** This is a feasibility question. Can you formulate a network flow/circulation problem?] [12]

(Ans:) We have to assign/match students for vacation days, say $D = \bigcup_j D_j$ in number. At a broad level, this seems to be a flow kind of problem. Let us look at the constraints and how easily we can handle them. For a start, look at Figure 3. Handling (a) seems to be easy. For each student i, $1 \leq i \leq n$, put edges, with a capacity of 1, following the set of vacation days S_i when the student is available for work. What about (b)? We need to ensure that each student works for at most c days. To enforce this constraint, we look at the edges from source to each student and put a capacity value c over each edge. Now look at a path from the source to sink. A flow of at most c units can come out of the source; get divided with 1 unit of flow per edge from the nodes corresponding to the students and then these flows come to the nodes corresponding to vacation days. We now need to ensure that each vacation day gets served by a student and not more than one student crowd a vacation day. So, put a lower and upper bound of 1 on the capacity of each edge from the nodes of the vacation periods to the sink. What is still left? We have to ensure that D vacation days are served. We handle that by putting demands of $+D$ on the sink and $-D$ on the source. This handles everything apart from condition (c). Handling (c) seems difficult.



(a) The network for the simple case.



(b) The final case with the gadgets G_i.

Figure 3: The circulation network.

We put gadgets G_i, $1 \leq i \leq n$, for each student. Each gadget G_i has at most m nodes, labelled $(i,1), (i,2), \ldots, (i,m)$. In the earlier network shown in Figure 3(a), if there was an edge between a student i to say k nodes in D_j, then now there is an edge from student i to k nodes in gadget G_i, and from each of those k nodes in gadget G_i, there are k edge going to exactly those k nodes in D_j. Put a capacity of 1 on each newly introduced edge. Now, why does this work? For each pair of student i and vacation period j, there is now an edge with capacity 1. This forces for each vacation period j, each student should be assigned to work at most one of the days in the set D_j.

Now, we solve for circulation in the network shown in Figure 3(b).