15-750 — Graduate Algorithms — Spring 2010

1 Cheap Cut (25 pts.)

Suppose we are given a directed network $G = (V, E)$ with a root node r and a set of terminal nodes $T \subset V$. We would like to remove the fewest number of edges from G that disconnect the most terminal nodes from the root r. More formally, let $F \subset E$ subset of edges and let $q(F)$ denote the number of terminal nodes that F disconnects from r. The goal is to find the subset of edges F that maximizes the sum $q(F) - |F|$.

Note: setting $F = \emptyset$ is a possible solution.

Solution: Add to the graph G a new node t' and connect every node $t \in T$ with a directed edge to t'. Call this new graph G'. Run a max flow algorithm on G' using r as s and t' as t. Let every edge in G' have capacity 1. Define G'_f to be the residual graph of G' with max flow f. Starting at r, run a breadth first search to construct S, the set of all vertices reachable from r in G'_f. Let $U = V - S$. Take E_c to be the edges in G' that are from a vertex in S to a vertex in U. Finally, remove from E_c all edges from any $t \in T$ to t' and call the remaining edges F. F is now a subset of edges that will maximize $q(F) - |F|$.

Timing Analysis: The runtime of the algorithm will be dominated by the cost of the max-flow algorithm. If we use the preflow push max height algorithm, it will take $O(n^3)$ time to find the max flow. It takes $O(m)$ time to calculate residual graph G_f and $O(m)$ time to construct S using a breadth first search, where $|E| = O(m) = O(n^2)$. We can find the edges from S to U in $O(m)$ time and prune the set of edges F in $O(n)$ time because there are at most $n - 1$ terminal nodes. Thus, the total runtime is $O(n^3) + O(n^2) + O(n) = O(n^3)$.

Correctness: We will show that finding a "cheap cut" in G is equivalent to finding a min cut in G'. After finding the min cut in G', we can divide the set of cut edges, E_c, into E_1, those edges that $\in E$, and E_2, the added we added from every $t \in T$ to t'. Note that by definition $E_1 = F$.

We know that $q(f)$ is the set of nodes $\in U$ after finding the min cut in G'_f, because these are the nodes that are not reachable from r in G'_f. If $|T| = k$, then there are $k - q(f)$ terminal nodes $\in S$. The max flow min cut theorem tells us that there is no augmenting path in G'_f, which means that the edges from the $k - q(f)$ terminal nodes $\in S$ to t' must be saturated. If they were not, then there would exist an augmenting path in G'_f due to that fact that r and these terminal nodes are connected and the terminal nodes directly connect to t'. Therefore, $|E_2| = k - q(f)$.

Because every edge in G' has a capacity of 1, the min cut in G' is the cut that minimizes the number of edges used in the cut, E_c. Thus, the min cut tries to minimize $|E_c| = |E_1| + |E_2| = |F| + k - q(f)$. Because k is fixed, this minimization problem is equivalent to maximizing $q(f) - |F|$.

The max-flow min-cut theorem guarantees that we can obtain the edges used to make the min cut using the process described in the algorithm (using BFS to construct S and then finding edges from S to U). Because the edges in E_2 do not actually exist in E, they must be pruned from E_c. However, removing the edges E_2 from E_c does not change the value of $q(f) - |F|$. This is because for every edge in E_2 there is exactly one corresponding terminal node that is reachable from r. Therefore, removing an edge in E_2 from E_c can be thought of as decreasing $|F|$ by 1 and simultaneously decrementing $q(f)$ by 1

because that terminal node is not disconnected from r when only the edges in E_1 are removed from G.

2 Hamming Distance Using FFT (25 pts.)

Define the *hamming distance* between two strings a and b of equal length to be the number of positions at which a and b differ. The goal of this problem is to develop an algorithm that takes as input a string s of length n and a pattern p of length m, both of which use characters from the set $\{1, \ldots, k\}$, and outputs the hamming distance between p and every contiguous substring of length m in s.

(a) Suppose we restrict the alphabet to $\{1, 2\}$. How can we compute the hamming distances of p to the substrings of s in time $O(n \log m)$?

(b) Now suppose we expand the alphabet to $\{1, 2, 3\}$. How can we compute the hamming distances of p to the substrings of s in time $O(n \log m)$? (Hint: Recall the details of the FFT string processing lectures.)

(c) Now, give a fast algorithm for computing the hamming distances of p to the substrings of s if we expand the alphabet to $\{1, \ldots, k\}$ where k is relatively small compared to n and m but not necessarily constant. What is the running time of your algorithm in terms of n, m, and k?

Solution:

a) Simply convert the alphabet to $\{0, 1\}$ and use the standard, non-wildcard FFT string matching set up, which computes $\sum_{j=1}^{m}(s_{i+j} - p_j)^2$ for all relevant i. This gives precisely the hamming distances we are after, and requires only $O(n \log m)$ time to compute.

b) Transform all instances of 1s into 0s and all instances of 2s and 3s into 1s. Using the same observation as in part (a), we can compute the number of 1s that should have been either a 2 or a 3 plus the number of 2s and 3s that should have been a 1 for every contiguous substring of length m. Repeating this procedure for 2 and 3 and summing the results gives us twice the hamming distance of each contiguous substring of length m to p. This whole algorithm consists of 3 iterations of a step that costs $O(n \log m)$, so the entire algorithm costs $O(n \log m)$.

c) Using a similar idea to part (b), we can, for each character $c \in \{1, \ldots, k\}$ set all the c's to 0s and all the non-c's to 1s to compute the number of c's that should be non-c's plus the number of non-c's that should be c's. Summing over all values of c gives us exactly twice the desired result. Clearly, this algorithm's running time is $O(kn \log m)$.

3 Something about Laplacian (25 pts.)

The Laplacian of a graph encodes a number of interesting properties of the graph. In this problem, we will examine some (random) facts about graph Laplacians.

a) **Positive Semi-definiteness.** An $n \times n$ real symmetric matrix A is *positive semi-definite* if for all $x \in \mathbb{R}^n$, $x^\top Ax \geq 0$. We know a lot of nice properties of such matrices from the theory of linear algebra. For instance, it is known that the following statements are equivalent: 1) all eigenvalues of A are non-negative, 2) for all $x \in \mathbb{R}^n$, $x^\top Ax \geq 0$, and 3) there exists a matrix B such that $B^T B = A$. Indeed, an example of positive semi-definite matrices is the Laplacian of a undirected graph.

Using any of the equivalent definitions above, prove that if $G = (V, E)$ is a simple undirected graph with weights given by $w : V \times V \to \mathbb{R}_+ \cup \{0\}$, then $L(G)$ is positive semi-definite. We adopt the convention that $w(i, j) = 0$ if $(i, j) \notin E$.

b) **Max-Cut.** In the MAX-CUT problem, we are given a weighted undirected graph $G = (V, E, w)$, and we are interested in finding a partition of V into S and $\overline{S} = V \setminus S$ such that $\sum_{(i,j)\in S\times\overline{S}} w(i, j)$ is maximized. This corresponds to maximizing the number of edges crossing the cut in the unweighted version. Let Max-Cut(G) denote the value of the best cut on G. That is,

$$\text{Max-Cut}(G) := \max_{S\subseteq V} \sum_{(i,j)\in S\times\overline{S}} w(i, j).$$

Assume that all edge weights are non-negative. Show that

$$\text{Max-Cut}(G) = \max_{x\in\{-1,1\}^n} \frac{1}{4} x^\top L(G)x. \max\left\{ \frac{n}{4} x^\top L(G)x : x \in \left\{-\tfrac{1}{\sqrt{n}}, \tfrac{1}{\sqrt{n}}\right\}^n \right\}.$$

Solution: Let $A = [a_{i,j}]_{n\times n} = L(G)$, and $x \in \mathbb{R}^n$ be given. Consider that the i-th row of Ax, denoted by $(Ax)_i$, is $\sum_{j=1}^n a_{i,j} \cdot x_j$, so

$$x^\top Ax = \sum_{i=1}^n x_i \cdot (Ax)_i = \sum_{i=1}^n \sum_{j=1}^n a_{i,j} x_i x_j.$$

Now since $A = L(G)$ is symmetric and $a_{i,i} = \sum_{j\neq i} w(i, j) = \sum_{j\neq i} w(j, i)$, we have

$$\begin{aligned} x^\top Ax &= \sum_{i=1}^n a_{i,i} x_i^2 + 2\sum_{i<j} a_{i,j} x_i x_j = \sum_{i=1}^n \sum_{j\neq i} w(i, j) x_i^2 - 2\sum_{i<j} w(i, j) x_i x_j \\ &= \sum_{i<j} w(i, j)(x_i^2 - 2x_i x_j + x_j^2) = \sum_{i<j} w(i, j)(x_i - x_j)^2 \geq 0, \end{aligned}$$

which proves that $L(G)$ is positive semi-definite.

For a cut $(S, V \setminus S)$, define

$$\mathit{val}(S, V \setminus S) = \sum_{(i,j)\in S\times(V\setminus S)} w(i, j).$$

In part (a), we established that $x^\top L(G)x = \sum_{i<j} w(i, j)(x_i - x_j)^2$ for any $x \in \mathbb{R}^n$. Specifically, if $x \in \{-1, 1\}^n$, we have

$$x^\top L(G)x = 4 \sum_{\substack{i<j \\ x_i\neq x_j}} w(i, j),$$

which suggests the following natural bijection between cuts and ± 1 vectors: the i-th component of the vector $x \in \{-1, 1\}^n$ is 1 if and only if i belongs to the set S in the cut $(S, V \setminus S)$. As such, if x is the vector corresponding to $(S, V \setminus S)$, then

$$\frac{1}{4} x^\top L(G) x = \sum_{\substack{i<j \\ x_i \neq x_j}} w(i,j) = \mathit{val}(S, V \setminus S)$$

Since every cut has a corresponding ± 1 vector and vice versa, we conclude that Max-Cut$(G) = \max_{x \in \{-1,1\}^n} \frac{1}{4} x^\top L(G) x$.

Formally, let $w = \max\{\frac{1}{4} x^\top L(G) x : x \in \{-1, 1\}^n\}$. Then, if $(S^*, V \setminus S^*)$ is a cut such that $\mathit{val}(S^*, V \setminus S^*) =$ Max-Cut(G), the corresponding vector x satisfies $\frac{1}{4} x^\top L(G) x =$ Max-Cut(G), so $w \geq$ Max-Cut(G). If x^* is a vector such that $\frac{1}{4} x^{*\top} L(G) x^* = w$, then the corresponding cut $(S, V \setminus S)$ has the property that $\mathit{val}(S, V \setminus S) = w$, so $w \leq$ Max-Cut(G). We conclude that $\max\{\frac{1}{4} x^\top L(G) x : x \in \{-1, 1\}^n\} = w =$ Max-Cut(G).

4 Randomized s-t Connectivity (25 pts.)

Suppose we're given an undirected graph $G = (V, E)$ and two nodes s and t in G, and we want to determine if there is a path connecting s and t. As we have seen in class, this is easily accomplished in $O(|V| + |E|)$ using DFS or BFS. The space usage of such algorithms, however, is $O(|V|)$.

In this problem, we will consider a randomized algorithm that solves the s-t connectivity problem using only $O(\log |V|)$ space. Here is a very simple algorithm which we will analyze.

Step 1: Start a random walk from s.
Step 2: If we reach t within $4|V|^3$ steps, return "`CONNECTED`." Otherwise, return "`NO`."

Assume that G is not bipartite. Your proof doesn't have to match the constants below exactly; they just have to be in the same ballpark.

a) For any node $u \in G$, $h_{u,u} = \frac{2|E|}{\deg(u)}$.

b) Prove that for any edge $(u, v) \in E$, $C_{u,v} \leq |E|$, where $C_{u,v}$ is the commute time between u and v. *(Hint: Rayleigh's Monotonicity Principle.)*

c) Let $\mathcal{C}(G; v)$ be the expected length of a walk starting at u and ending when it has visited every node of G at least once. The *cover time* of G, denoted $\mathcal{C}(G)$, is defined to be

$$\mathcal{C}(G) := \max_{v \in V(G)} \mathcal{C}(G; v).$$

Show that the cover time of $G = (V, E)$ is upper bounded by $|V||E|$. *(Hint: Construct an Euler's tour on a spanning tree.)*

d) Conclude that the algorithm above is correct with probability at least 7/8. *(Hint: Markov's inequality.)*

Solution: We show the following:

Claim: For an edge $uv \in E$, the commute time between u and v, denoted by $C_{u,v}$, is at most $2|E|$.
Proof. Let G' be obtained from G by removing all the edges except for uv, so the effective resistance between u and v in G' is 1. By Rayleigh's monotonicity principle, we know that the effective resistance between u and v is higher in G' than in G. This means that $R_{uv} \leq 1$. Now we know from class that $C_{u,v} = C \cdot R_{uv}$, so $C_{u,v} \leq C = \sum_i C_i = \sum_i \deg(i) = 2|E|$.

Lemma: The cover time of a graph $G = (V, E)$ is at most $2|V||E|$.

Proof. Let T be an undirected spanning tree of G. Since G is connected, T has $|V| - 1$ edges. Create T' from T by replacing every undirected edge uv with directed edges (u, v) and (v, u). Let $u_0, u_1, \ldots, u_{2|V|-1} = u_0$ be an Euler's tour of T'. It is easy to see that $\mathcal{C}(G) \leq \sum_{i=0}^{2|V|-1} h_{v_i,v_{i+1}}$, where $h_{u,v}$ denotes the hitting time from u to v. Now note that in this tour each original tree edge was traversed twice, one in each direction. We have $\sum_{i=0}^{2|V|-1} h_{v_i,v_{i+1}} = \sum_{e \in T} C_e$. Combining this with the claim we proved above, we have $\mathcal{C}(G) \leq 2|V||E|$.

Corollary: The given algorithm is correct with probability at least $3/4$.

Proof. If s and t are not connected, the algorithm reports so correctly because the random walk will never reach t. If s and t are indeed connected, let X be a random variable for the number of steps we take to reach t from s, so $\mathbf{E}[X] = h_{s,t} \leq \mathcal{C}(G) \leq 2|V||E| \leq |V|^3$ (remember that $|E| \leq \binom{|V|}{2} \leq |V|^2/2$). Now our algorithm gives a wrong answer if and only if $X \geq 4|V|^3$. Therefore, the algorithm is correct with probability $1 - \mathbf{Pr}\big[X \geq 4|V|^3\big]$, which can be bounded using Markov's inequality as follows:

$$1 - \mathbf{Pr}\big[X \geq 4|V|^3\big] \geq 1 - \frac{\mathbf{E}[X]}{4|V|^3} \geq 1 - \frac{1}{4} = \frac{3}{4}.$$

This completes the proof.

5 Separators in outer planar graphs (25 pts.)

We say that a graph is *outer planar* if it is

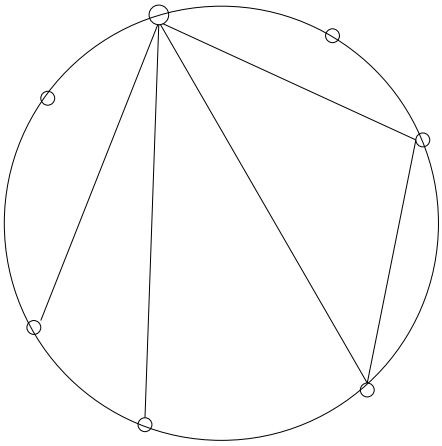
1) planar, and
2) there exists a planar embedding such that all of the vertices are on the boundary of the outer face.

Prove that all outer-planar graphs have 2-vertex-separators. Do not give a proof by algorithm. Use the tools developed in class.

HINT: Recall that augmenting the graph does not change the size of the separators.

Solution:

Start with a planar drawing of G. There is some outer face containing all of the vertices. Triangulate the other faces. Observe that the dual of the triangulation is a binary tree. It suffices to find a 1-edge separator in the dual. We did this in class. The two ends of the edge are vertices of the desired separator.



6 Maximum Independent Set in Trees (25 pts.)

Give a parallel algorithm that find a **maximum** independent set in a tree. Your algorithm should require at most $O(n \log n)$ work at run in $O(\log n)$ time.

Solution: This solution directly calculates the size of the maximum independent set.

First, consider storing a vector (a, b) at each node v where a is the number of max independent set not containing v of the tree rooted at v, and b is the number of the max independent set of the same tree, possibly containing v.

The RAKE operation is as follows. Let $(c_1, d_1), \ldots, (c_k, d_k)$ be the vectors stored at the children of v.

$$\begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{k} d_i \\ \max\{1 + \sum_{i=1}^{k} c_i, \sum_{i=1}^{k} d_i\} \end{bmatrix}$$

The value of a is just the sum of the weights of the max independent sets at each child. The value of b is the max of a and w_v plus the sum of the weights of the max independent sets at each child that don't contain that child.

If we have evaluated all but one child of v, we use the COMPRESS operation. Let (x, y) denote the vector to be computed at the child that has not yet been evaluated. Let (c, d) be the sum of the vectors stored at the other children. We can now write the compress operation as a matrix operation over the semiring with operations $(max, +)$ replacing the usual operations of $(+, *)$ respectively.

$$\begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} 0 & d \\ 1+c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} d+y \\ \max\{1+c+x, d+y\} \end{bmatrix}$$

It is not hard to show that this is indeed a semiring. Now, we know we can multiply these matrices to get new matrices. This corresponds to collapsing a chain of nodes with exactly one unevaluated child. This suffices to specify the usual parallel tree contraction algorithm. The result to return is b for the root of the tree. The runtime is $O(\log n)$ as with the usual parallel tree contraction.

7 An NP-Completeness Problem (25 pts.)

Suppose the $G = (V, E)$ is a weighted directed graph where edges can have negative weights. The **Zero-Weight-Cycle Problem** is to find a simple cycle in G such that the sum of its edge weights is zero. Show the Zero-Weight-Cycle Problem is NP-Complete.

Solution: First, given a simple cycle in G, we can determine whether the sum of its edge weights is zero in polynomial time. Thus **Zero-Weight-Cycle** $\in$ **NP**.

Then we reduce the **Subset Sum Problem** to this problem. The subset sum problem is: given a set of integers, determine whether the sum of some non-empty subset equal exactly zero. So consider a set of integers $S = \{a_1, \ldots, a_n\}$, we construct a weighted directed graph G with $2n$ vertices, such that every element a_i corresponds to two vertices v_i and u_i. For each v_i, add an edge from v_i to u_i with weight a_i and add edges from every vertex u_j to it with weight 0. For each u_i, add edges from this vertex u_i to every other v_j with weight 0. If we find a zero-weight-cycle in G, then all the weights from v_i to u_i along the cycle must be zero. On the other hand, if we get a subset $S' \subseteq S$ which sums to zero, we construct a cycle by picking all edges (v_i, u_i) corresponds to the element in S' and connect those edges by those zero weight edges and finally obtain a zero weight cycle. Thus this problem is at least as hard as subset sum problem. Since the subset problem is NP-complete, we have **Zero-Weight-Cycle** $\in$ **NPC**.

8 Approximation Algorithms for MaxCut (25 pts.)

Now that we've shown that finding the Maximum Cut in a graph is NP-complete, the next natural question is whether we can approximate it. For each of the following parts you should give an algorithm, prove the approximation ratio, and prove that it runs in polynomial time.

(a) Design a randomized algorithm that does not even look at the edges of the graph, but outputs a cut that is an expected 2-approximation of the maximum cut.

(b) A *local search* algorithm for a problem typically works as follows. First, start with an arbitrary solution. Then continue to make small, local steps to improve the solution, until there are no more such improving steps. Design a local search algorithm for the Maximum Cut problem that is a 2-approximation. Hint: Let your local steps be switching a single node from one side of the cut to the other.

Solution:

(a) For each vertex we will randomly and independently flip a fair coin to decide which part of the cut the vertex will be on. In other words, each vertex has a probability of $1/2$ of being on one side of the cut and $1/2$ of being on the other, and the outcomes for different vertices are independent. Let X be a random variable equal to the number of edges that are cut (i.e. the expected value returned by our algorithm), and for each edge e let X_e be an indicator variable that is 1 if e is cut and 0 if it is not. It is easy to see that the probability that an edge is cut is exactly $1/2$, so $\mathbf{E}[X_e] = 1/2$. So by linearity of expectations we get that $\mathbf{E}[X] = \mathbf{E}[\sum_e X_e] = \sum_e \mathbf{E}[X_e] = \sum_e \frac{1}{2} = |E|/2$. Clearly any cut can cut at most $|E|$ edges, so $|OPT| \leq |E|$. Thus $\mathbf{E}[X] \geq |OPT|/2$, as required.

(b) Our algorithm is as follows. We first start with an arbitrary cut $(S, \overline{S})$ in the graph. Let v be some arbitrary vertex, which is in either S or $\overline{S}$. v has some edges to vertices on its side of the

cut, and some edges to vertices on the other side of the cut. If it has more edges to vertices on the other side of the cut, then we can improve the cut by switching the side of v. We keep doing this (in arbitrary vertex order) as long as there are vertices for which switching would help. This is clearly a local search algorithm. Note that the algorithm does eventually terminate in polynomial time since every switch increases the size of the cut by at most 1, and the size of the maximum cut is at most $|E|$.

We now need to prove that it is a 2-approximation. For each vertex v, let d_v be its degree, let d_v^u be the number of edges incident to v that are not cut by our solution, and let d_v^c be the number of edges incident to v that are cut by our solution. Note that $d_v = d_v^c + d_v^u$ and that $d_v^c \geq d_v^u$ (since otherwise we would switch the side of v so the algorithm would not have terminated yet). Then $2|E| = \sum_v d_v = \sum_v (d_v^c + d_v^u) \leq \sum_v 2d_v^c$, so $\sum_v d_v^c \geq |E| \geq |OPT|$. Now note that in the sum $\sum_v d_v^c$ we count every cut edge twice, once for each endpoint. Thus the number of edges in our cut is at least $|OPT|/2$, as claimed.

Practice Questions from Midterm Review

9 Interval Placement (25 pts.)

After recently talking with Catherine, you have decided to try to piece together some of the early history of the CMU CS department. In particular, you would like to figure out the approximate dates that former graduate students began their time here, as well as when they left. Let $P_1, \ldots, P_n$ be the former students that you are interested in. Unfortunately there are very few records going back that far, and the memories of the faculty and staff who were around then are a bit hazy. In fact, you only have two types of information:

1. For some i and j, person P_i left CMU before person P_j arrived.
2. For some i and j, there was a time when both P_i and P_j were at CMU.

You've talked with all of the people who were around back then, and now have a collection of this kind of information. While this information obviously doesn't suffice to pinpoint dates, it is not even clear if your collection is internally consistent, i.e. if there is some assignment of entry and exit dates to people such that all of the information you've collected is true. Give an efficient algorithm that either verifies that the information is consistent or reports (correctly) that it is not.

Solution: We will formulate it as a graph problem. For each student i, we construct two nodes $v_{i,s}$ and $v_{i,f}$, which correspond to i's start and end dates. Put a directed edge from $v_{i,s}$ to $v_{i,f}$. An edge from u and v in this problem should be thought of as u comes no later than v. Following this interpretation, we do the following for each piece of evidence: (a) if P_i left before P_j arrived, we create an edge from $v_{i,f}$ to $v_{j,s}$, and (b) if P_i and P_j were contemporaries, we add edges $(v_{i,s}, v_{j,f})$ and $(v_{j,s}, v_{i,f})$.

The following are clear from the construction. First, the graph has $2n$ nodes and at most m edges, where m is the number of information pieces gathered. Second, we can run DFS on this graph to determine whether or not it is a DAG. Third, both the graph construction and DFS can be done in $O(n+m)$ time. Going back to the original problem, we claim that the graph is a DAG if and only if the gathered information is consistent.

We say that the gathered information is consistent if and only if there is a linear ordering of all start and end dates consistent with what we gathered. Immediately we have that if the gathered information is consistent, any linear ordering of the dates naturally gives a topological ordering of the nodes of our graph, so the graph must be a DAG. Conversely, if our graph is a DAG, any topological ordering of the graph's nodes provides a linear ordering of time points consistent with all the gathered information.

10 Fair Carpooling (25 pts.)

There are n people, $p_1, p_2, \ldots, p_n$. Every day, some subset of these people carpool together. Let S_1 be the set that carpool together on day 1. There are m such sets, $S_1, \ldots, S_m$. For each set, one of the people in the set must be chosen to be the driver that day. Driving is not desirable, so the people want the work of driving to be divided as equitably as possible. Your task in this problem is to give an algorithm to do this efficiently and fairly.

The fairness criterion is the following: A person p is in some of the sets. Say the sizes of the sets that she's in are $a_1, a_2, \ldots, a_k$. Person p should really have to drive $\frac{1}{a_1} + \frac{1}{a_2} + \cdots + \frac{1}{a_k}$ times, because this is the amount of resource that this person effectively uses. Of course this number may not be an integer, so let's round it up to an integer. The fairness criterion is simply that she should drive no more times than this many times.

For example, say that on day 1, Alice and Bob carpool together, and on day 2, Alice and Carl carpool together. Alice's fair cost would be $1/2 + 1/2 = 1$. So Alice driving both days would not be fair. Any solution except that one is fair.

Give a polynomial-time algorithm for computing a fair solution. (And thereby proving that a fair solution always exists!)

Solution: Define a directed graph as follows.

There is a source s. Then there is a first layer with n vertices $p_1, p_2, \ldots, p_n$. Then there is a second layer with m vertices $s_1, s_2, \ldots, s_m$. Then there is a sink t.

For each $i \leq n$, there is an edge (s, p_i) with weight $\left\lceil \frac{1}{a_{i1}} + \frac{1}{a_{i2}} + \cdots + \frac{1}{a_{ik_i}} \right\rceil$, where the $a_{i1}, a_{i2}, \ldots, a_{ik_i}$ are the set sizes of sets that person p_i is in (as described in the problem description).

For each $i \leq n$ and $j \leq m$, there is an edge (p_i, s_j) of weight 1 if and only if person p_i is in set S_j.

Finally, for every $j \leq m$, there is an edge (s_j, t) of weight 1.

To get the driving assignments, we use Ford-Fulkerson to find a flow between s and t, which we know will only give integer flows across any edge in this case (Theorem 26.11 in CLRS). If there is a unit of flow across edge (p_i, s_j), then person p_i drives for the set S_j.

Clearly the time complexity is polynomial, as required, since Ford-Fulkerson is polynomial and the graph is only of polynomial size in n and m.

On the issue of correctness, I'll argue as follows. First, note that the weight of the incoming edge to each p_i means that person can be assigned to drive at most the number of times he or she is limited to by fairness. So the fairness constraint is definitely satisfied. The only remaining question is whether every set S_j has a driver assigned to it. For this I argue as follows.

I now show there exists a flow that saturates all of the edges pointing to t, which by the integrality theorem implies there exists an integer flow that does so, which gives us our solution. Let $w_i =$

$\frac{1}{a_{i1}} + \frac{1}{a_{i2}} + \cdots + \frac{1}{a_{ik_i}}$. Put w_i flow through each (s, p_i) edge. Then for each (p_i, s_j) edge that exists, send $\frac{1}{|S_j|}$ flow. This exhausts all the flow that was incoming to p_i. Note also that now each s_j has flow 1 incoming. So each (s_j, t) gets exactly flow 1 across it, as required. Although this flow sends noninteger amounts across some edges, it is a maximal flow (the (s_j, t) edges are all saturated), so the integrality theorem implies that there exists an integer flow of this size as well (which must therefore saturate all the (s_j, t) edges), as required.

11 Parenthesis checking with uncertainty (25 points pts.)

Let S be a string over the 3 letter alphabet $\{(,), ?\}$. Give an algorithm to check whether there is an assignment that replaces each '?' with either a left or right parentheses yielding a correctly nested set of parentheses.

Make sure you include in this order. 1) A well defined statement of the subproblems you will solve. 2) a high level recursive description of how the subproblems are solved. 3) a proof of correctness and 4) a run time analysis.

Full credit will be given for an $O(n^2)$ algorithm.

Solution:

Here's some notation we'll use. The input is $x = x_1 x_2 \cdots x_n$, where we let x_i be the i-th character of x. Note n is the length of x. We will let $x[i:j] = x_i x_{i+1} \cdots x_j$ denote the substring of x from the i-th character to the j-th. Lastly, we will say that a string x is *good* if its question marks can be mapped to parens to make it well-balanced.

We give a dynamic programming solution as follows:

Subproblems: For each $1 \le i \le k$ and for each $0 \le j \le i$, is the string $x[1:i] \cdot)^j$ good? (Here, $\cdot$ is string concatenation and a^j is the length j string consisting only of the character 'a'. Thus $x[1:i] \cdot)^j$ is obtained by adding j right parens to the end of $x[1:i]$). We define $A[i,j] = 1$ if the answer is yes, otherwise $A[i,j] = 0$. Our original problem is then equal to $A[n, 0]$.

Basis and Recursion: Define $B[i,j] = 0$ if $j > i$ or $j < 0$. Similarly, let $B[1,0] = 0$, $B[1,1] = 0$ if $x_1 = '('$ and $B[1,1] = 1$ otherwise. The recursion is then (for $i \ge 2$):

$$B[i,j] = \begin{cases} B[i-1, j-1] & \text{if } x_i = '(' \\ B[i-1, j+1] & \text{if } x_i = ')' \\ \max\{B[i-1, j-1], B[i-1, j+1]\} & \text{if } x_i = '?' \end{cases}$$

Note we can make various optimizations, such as $B[i,j] = 0$ if $i + j$ is odd, and the final answer is no if $x_1 = ')'$ but we will ignore these considerations for now.

Correctness: Trivially $A[n, 0]$ is the answer we seek. We will proceed by induction on i to show the recurrence is correct: That is, $B[i,j] = A[i,j]$ for all (i,j) for which $A[i,j]$ is defined. The base case, $i = 1$, is trivial.

For the induction, consider $y := x[1:i] \cdot)^j$. Then $y = x[1:i-1] \cdot x_i \cdot)^j$. Suppose $x_i = '('$. If $j = 0$, the string is clearly not good, since we can't balance x_i, so $A[i,0] = 0$. However, we

correctly compute $B[i,0] = B[i-1,-1] = 0$. Otherwise $j > 0$ and $y = x[1:i-1] \cdot () \cdot)^{(j-1)}$ and so y is good iff $x[1:i-1] \cdot)^{j-1}$ is good. By the induction hypothesis, $B[i-1,j-1]$ is correct and thus $B[i,j]$ is correctly computed.

The case in which $x_i = ')'$ is similar, except that we get $y = x[1:i-1] \cdot)^{(j+1)}$ instead. Note that $A[i,j] = 0$ for $j > i$, since $x[1:i]$ simply doesn't have enough characters to balance the j right parens appended to the end of it. We omit the details.

Finally, the case in which $x_i = '?'$ reduces to trying both possibilities for the '?', and returning yes if either string is good. By taking the max of $B[i-1,j-1]$ and $B[i-1,j+1]$, this is exactly what we do.

Running Time: The table has size $O(n^2)$ via a simple calculation, and each entry takes constant time to compute, for $O(n^2)$ time total.

12 Separating Line (25 pts.)

Suppose we are given two sets of points in the plane, say, A and B, each of size n. The goal is to determine if the exists a line L that separates A from B. Your algorithm should run in expected $O(n)$ time. You may assume that it is OK for points of A and B to lie on the line L.

Suppose the points in A are given as $A_i = (a_x^i, a_y^i)$ and similarly for B.

1. Show how to reduce the separating line problem for A and B to a 3D LP problem.
2. It is known that 3D LP can be solved in expected linear time, but we only have seen how to do 2D LP in linear time. Show that after a linear amount of preprocessing you can reduce the problem to a single 2D LP.

   Hint: Consider lines of the form $\{(x,y) \in \mathbb{R}^2 \mid ax + by = 1\}$. Show how to move the points so that the origin will be in the set A without changing the answer.

Solution: SKETCH

A line can be defined using the equation $ax + by + c = 0$ with the two sides having ≥ 0 and ≤ 0 respectively. So the following constraints should work:

$ax + by + c \geq 0$ for all $(x,y) \in A$

$ax + by + c \leq 0$ for all $(x,y) \in B$

It suffices to check feasibility of this LP, so any objective function should work.

This can be reduced to a 2 dimensional LP by requiring $c = 1$ if the line doesn't go through the origin. If the solution must go through the origin, then we can shift the y coordinate of all the points up by 1 and shift the solution back.

13 Longest Path in a DAG (30 pts.)

Let $G = (V,E)$ be a directed acyclic graph and s and t be two distinguished vertices of G. At each vertex of this graph are some number of tokens and as you visit the vertex you can collect it tokens.

The goal is to find a path from s to t that collects the largest number of tokens. For simplicity we will only compute the maximum number of tokens over all paths from s to t

Show how to use DFS to find the maximum number of tokens on any path from s to t if a path exists. In particular, your algorithm should return $-\infty$ if there is no path from s to t. Otherwise, it should return the number of tokens. Assume that $TOKENS(v)$ returns the number of tokens at v.

1. Modify the code below by adding code to the right of the code below so that it computes the longest path.

   DFS(G)

   - ForEach vertex $u \in V[G]$
   - $\quad color[u] \leftarrow WHITE$
   - EndFor
   - $time \leftarrow 0$
   - ForEach vertex $u \in V[G]$
   - $\quad$ If $color[u] = WHITE$ Then DFS-Visit(u)
   - EndFor

   DFS-Visit(u)

   - $color[u] \leftarrow GRAY$
   - $time \leftarrow time + 1$
   - ForEach $v \in Adj[u]$
   - $\quad$ If $color[v] = WHITE$ Then DFS-Visit(v)
   - EndFor
   - $color[u] \leftarrow BLACK$

2. Outline a proof that your algorithm correctly computes the maximum number of tokens from s to t.

Solution:

The following is a rather pedantic proof from the original solutions; we do not expect this level of details on the exam.

DFS(G)

- ForEach vertex $u \in V[G] \setminus \{t\}$
- $\quad color[u] \leftarrow WHITE$
- $\quad d[u] \leftarrow (-\infty)$
- EndFor
- $d[t] \leftarrow 0$

- $color[u] \leftarrow BLACK$
- DFS-Visit(s, d)
- return $d[s]$

DFS-Visit(u, d)

- $color[u] \leftarrow GRAY$
- ForEach $v \in Adj[u]$
- If $color[v] = WHITE$ Then DFS-Visit(v, d)
- EndFor
- $D \leftarrow \max\{d[v] \mid v \in Adj[u]\}$
- If $D + 1 > d[u]$ Then $d[u] \leftarrow D + 1$
- $color[u] \leftarrow BLACK$

Proof: We will prove that for each $v \in V$, once v is colored black then $d[v]$ equals the length of the longest path from v to t. Since the initial call to DFS-Visit(s, d) will eventually color every vertex reachable from s black, this is sufficient to prove that the algorithm correctly computes the length of the longest path from s to t. (Note that finding the path given the labels $d[u]$ is easy – merely start from s and find an edge (s, u_1) with $d[s] = 1 + d[u]$, then find an edge (u_1, u_2) with $d[u_1] = 1 + d[u_2]$, and so on.) Let $f[v]$ equal the length of the longest path from v to t, and let d be the array generated by our algorithm. We prove that $d[v] = f[v]$ for all v colored black by induction on $f[v]$.

The base case, $f[v] = 0$, is trivial, since $f[v] = 0$ implies $v = t$, and we set $d[t] = 0$ initially and we never change its value. (We never change $d[t]$ because we color it black before calling DFS-Visit(s, d), and thus we never call DFS-Visit on t.)

For the inductive step, we assume $d[v] = f[v]$ for all black nodes v such that $f[v] \leq n - 1$, and we will prove that $d[v] = f[v]$ for all black nodes v such that $f[v] = n$. Fix a black node v such that $f[v] = n$, and a (longest) path P from v to t of length n. Let (v, u) be the first edge in P. Since P is a longest $v - t$ path, $f[u] = n - 1$ and $\max\{f[u] \mid u \in \text{Adj}[v]\} = n - 1$. By the induction hypothesis, $d[u] = f[u]$ for all black nodes $u \in \text{Adj}[v]$. Also, by inspecting the DFS-Visit code it is clear that any node w can only be colored black after all its children have been colored black, since the last thing DFS-Visit does is color its argument node black, and before that it ensures all its white children are colored black before that. (Note it is impossible for a child w' of w to be gray the moment DFS-Visit is called on w, as this would imply a path from w' to w in G, contradicting the fact that it is a DAG.) Thus by the time we color v black, we know that

1. $d[v] = \max\{1 + d[u] \mid u \in \text{Adj}[v]\}$
2. Each child of v is colored black.
3. Each black node u with $f[u] \leq n - 1$ has $d[u] = f[u]$.

Combining these facts with observation that $\max\{f[u] \mid u \in \text{Adj}[v]\} = n - 1$ and inspecting the code, we see that $d[v]$ is correctly set to $f[v] = n$.

14 Merging Balanced Trees (30 pts.)

Let T_1, T_2 be two balanced binary trees with n nodes each. By *balanced*, we mean that the depth of each tree is $O(\log n)$. In the following questions, please give high-level descriptions of the algorithms only; that means pseudocode, *NOT* C or ML.

a) Give an $O(n)$ time algorithm that returns a sorted list of the elements of T_1 and T_2. (Yes, this is easy so please keep your answers brief.)

b) Give an $O(n)$ time algorithm that takes a sorted list of keys and returns a new balanced binary search containing the keys.

c) The previous two questions provide a linear time algorithm to merge two balanced binary search trees. Design a new algorithm that does this using Divide-and-Conquer recursion. The new algorithm should also run in time $O(n)$. Give a high level description of the algorithm. Prove that it is correct. Analyze the running time. Be sure to prove that the resulting tree is balanced.

Use the following procedure: (T_1,T_2) = SPLIT(k: key, T: BST)
SPLIT Takes as input a balanced BST returns two balanced BSTs where $T_1(T_2)$ are the keys in T less(greater) than k in T, respectively. You may assume that SPLIT runs on $O(\log n)$ time.

For your timing analysis give an $O(n \log n)$ upper bound for the case when T_1 and T_2 both have size n. For extra credit improve the upper bound to $O(n)$

Solution: SKETCH

1. Take the two inordes of T_1 and T_2, which are their elements in sorted order. Do a merge.
2. Put the middle as root, recurse on both sides. Depth is $\log n$ and time is $O(n)$ if the elements are taken from an array with indices calculated.
3. WOLOG (by swapping), we assume $|T_1| > |T_2|$. Then we find the median of T_1, x in terms of number of elements on each side, set that as the root of resulting tree. Let $(T_1^L, T_1^R) =$ SPLIT(x, T_1) and $(T_2^L, T_2^R) =$ SPLIT(x, T_2). We set the left child of x to MERGE(T_1^L, T_2^L) and the right child of x to MERGE(T_1^R, T_2^R).

   Let the runtime of merging two trees with total size n be $T(n)$. Suppose the size of the left subtree of x is m, then because T_1 is the larger tree, $m \leq \frac{3}{4}n$. So this process terminates in $O(\log n)$ steps and the tree is balanced.

   Consider the runtime, we have:

$$T(n) \leq T(m) + T(n-m) + \log n$$

   To show $T(n) \in O(n)$, we induct using the hypothesis $T(n) \leq n - 2\log n$. So it suffices to show:

$$\begin{aligned} m - 2\log(n-m) + (n-m) - 2\log m + \log n &\leq n - 2\log n \\ 3\log n &\leq 2\log(n-m) + 2\log m \\ n^3 &\leq (n-m)^2 m^2 \end{aligned}$$

Which follows from a convexity argument and $\frac{1}{4}n \leq m \leq \frac{3}{4}n$ (aka. $m \in \Theta(n)$).