CS161 Practice Final Exam
Date: May 31, 2016

Instructions: Please answer the following questions to the best of your ability. If you are asked to show your work, please include relevant calculations for deriving your answer, explaining steps as you go. If you are asked to prove a result, please write a complete proof at the level expected on your homework. When writing proofs, please strive for clarity and brevity (in that order).

You have 180 minutes to complete the exam. The exam is closed-book, closed-lecture-notes, closed-internet, etc. You may use 1 sheet, front and back, of notes as reference. By signing your name below, you acknowledge that you have abided by the Stanford Honor Code while taking this exam.

Name: ______________________

SUNetID: ______________________

Signature: ______________________

Throughout this exam, unless explicitly stated otherwise, you may assume the following:

- Graphs are represented by adjacency lists.
- Hash functions provide simple uniform hashing. You can assume that hash tables use chaining to resolve collisions, so the expected time for look-up in a hash table is $O(\alpha)$, where α is the ratio between the number of elements stored and the number of slots in the hash table.

On all questions, *you may use any of the algorithmic machinery that we've discussed in this course.* Any time you are asked to give an algorithm to solve a problem, you are expected to prove the algorithm's correctness and runtime. Unless the problem explicitly asks for a proof, you may cite the runtime and proof of correctness of any algorithm presented in class.

Note: The above instructions are the instructions from the actual final exam. This exam is meant as practice and will not necessarily take the full 180 minutes.

1 True or False

For each question, indicate whether the statement is True or False (by circling `T` or `F`, respectively). Correct answers will receive 4 points. Blank responses will receive 2 points. Incorrect answers will receive 0 points.

[✓ **T**] F — Iterating over all pairs of vertices in a graph requires $\Omega(n^{1.5})$ time.

T [✓ **F**] — Poe gives Barr a comparison-based sorting algorithm that runs in $T(n) = 4T(n/5) + 5n \log \log n$. There is not enough information to know whether Poe's algorithm is correct.

[✓ **T**] F — Barr gives Poe a comparison-based sorting algorithm that runs in $T(n) = 7T(n/6) + n/42$. There is not enough information to know whether Barr's algorithm is correct.

T [✓ **F**] — Consider using a variant of Karger's algorithm to compute an $s{-}t$ cut. To make sure that s and t end up in different partitions, we will then delete any edges between the supernodes that contain s and t, and repeat until there are two supernodes. This variant of Karger's algorithm computes the min $s - t$ cut with probability $1/n^c$ for some constant c.

[✓ **T**] F — Let G be an undirected, weighted graph. Multiplying every edge weight by 2 does not change the shortest paths in G.

[✓ **T**] F — Suppose in some graph G, the closest negative cycle to a node v has all its nodes at least 5 hops away from v. Consider any node x, whose shortest path from v has fewer than 5 hops. Bellman-Ford computes $d(v, x)$ correctly.

T [✓ **F**] — Suppose G has $\Theta(n)$ edges with integer weights on the range $O(n^3)$. The runtime of KRUSKAL'S and PRIM'S algorithm will be asymptotically equivalent.

2 What's Missing?

Suppose you're given a sorted array of length $n-1$ containing all but one element of $\{1,\ldots,n\}$. Give an algorithm for determining which element is missing. Prove that your algorithm is correct and runs in $O(\log n)$ time.

We'll assume that the array is indexed from 1 to n. We will use a variant of binary search. Given A with values in the range $[i,j]$, check if $A[\lfloor (j+i)/2\rfloor] = \lfloor (j+i)/2\rfloor$ (we start with $i=1, j=n-1$). If it does, recurse on the right half of the array with bounds of $[\lfloor (j+i)/2\rfloor + 1, j]$. Else, recurse on the left half of the array with bounds of $[i, \lfloor (j+i)/2\rfloor]$. If at any point we get $j=i$, if $A[i]\neq i$, we return i. Actually, provided the original array is sorted and has length $n-1$ and all but one element of $\{1,\ldots,n\}$ appear, we will never get that $j=i$ and $A[i]=i$, so that the if condition, $A[i]\neq i$, is unnecessary.

Claim 1. *At each iteration indexed by i,j, we learn that A contains $j-i$ elements on the range $[i,j]$.*

Proof. Initially, A contains $n-1$ elements from $[1,n]$ and $j=n$ and $i=1$, so A contains $j-i$ elements from $[i,j]$. Then, inductively, we'll assume that A contains $j-i$ elements on the range $[i,j]$ at the start of a call to our search routine. We can see that if the midpoint $m=\lfloor (j+i)/2\rfloor$ entry is too high, this means the missing element is prior to the midpoint entry. The right half of A must contain $j-m$ elements, so when we recurse on the left half, there will be $j-i-(j-m)=m-i$ elements on the range $[i,m]$, so the induction holds. If the mth entry is equal to m, it means all $m-i+1$ elements from i to m are present in the left half and the missing element is in the right half. When we recurse to the right, there will be $j-i-(m-i+1)=j-(m+1)$ elements on the range $[m+1,j]$, so again, the induction holds. □

To see correctness, when $j=i$ we have learned that A contains 0 elements from $\{i\}$, so that i is not in the array.

To see that the algorithm runs in $O(\log n)$ time, note that at each recursive call, if we start with n elements, we throw out at least $\lfloor n/2\rfloor$ elements after performing a constant number of comparisons. Thus, for any $n\geq 4$

$$\begin{aligned} T(n) &\leq T(n/2+1)+O(1) \\ &\leq T(3n/4)+O(1) \end{aligned}$$

Once $n=4$, the work remaining can be bounded by a constant, so by the Master Theorem, $T(n)\leq O(\log n)$.

3 Scheduling Jobs

Suppose you're given a list of n jobs $[j_1, j_2, \ldots, j_n]$. You have two machines, M_1 and M_2 that can run these jobs. Unfortunately, you may not schedule the jobs arbitrarily. Due to resource constraints, some pairs of jobs must be scheduled on the same machine, while other pairs of jobs must be scheduled on different machines. You are given a list of n jobs and for each job j, you are given a list $\mathsf{same}(j)$ containing the jobs which must be scheduled on the same machine as j, and $\mathsf{diff}(j)$ containing the jobs which must be scheduled on the opposite machine than j. In total, there are m such scheduling constraints. Design an algorithm that produces an allocation of the jobs onto M_1 and M_2 where all the constraints are met, or reports that no such allocation is possible. Prove that your algorithm is correct and runs in $O(m+n)$ time.

First, note that any legal allocation of the jobs must allocate every job. We can think of the allocation of jobs onto M_1 and M_2 as a graph G, where each job is represented as a node and each constraint is represented as an edge labeled as same or diff. Construct such a graph in $O(m+n)$ time. Consider initially assigning some job j to machine M_1. Now, in a greedy, depth-first manner, pick a $j' \in \mathsf{same}(j) \cup \mathsf{diff}(j)$ and assign j' to M_1 if $j' \in \mathsf{same}(j)$ and to M_2 if $j' \in \mathsf{diff}(j)$, until we encounter a job $j^\star$ that has already been assigned. At this point, verify that the assignment of $j^\star$ according to its predecessor is consistent with the previous allocation of $j^\star$. If not, we report inconsistent immediately. If we go through all jobs and constraints without reporting inconsistency, we claim we have a legal allocation.

Claim 2. *G represents a legal allocation if and only if there are no cycles with an odd number of diff edges; moreover, our algorithm will detect these cycles.*

Proof. ($\Longrightarrow$) Suppose there is a cycle with an odd number of diff edges. Then, fix a job j on this cycle. WLOG, if we assign j to a machine M_1, then after traversing the cycle, we will have been required to switch machines an odd number of times, so in particular, this means that j must be assigned to M_2. This is an inconsistency, so if there are cycles of an odd number of diff edges, no allocation can be legal. Our algorithm processes jobs in a depth first manner, so if such a cycle exists, it will discover the inconsistency.

($\Longleftarrow$) Suppose there is some inconsistency in the list of jobs. We know that this means that there is some pair of jobs j, j' where they are supposed to be both same and diff. We claim in our allocation graph, there is a cycle with an odd number of diff edges. WLOG, suppose there is a direct constraint between j and j'. Suppose $j \in \mathsf{same}(j')$. Then this is inconsistent only if j and j' are allocated to different machines. But by our greedy allocation, this means there is a path from j' to j with an odd number of diff edges (otherwise, they would have been allocated to the same machine). Thus, this means there would be a cycle with an odd number of diff edges. Suppose $j \in \mathsf{diff}(j')$. Then this is inconsistent only if j and j' are allocated to the same machine. But again, by our greedy allocation, this means that there is a path from j' to j with an even number of diff edges (otherwise, they would have been allocated to different machines). Thus, this means there would be a cycle with an odd number of diff edges. □

Because our algorithm detects cycles with inconsistencies, it is correct.

To see the runtime, note that we construct a graph on n nodes and m edges using DFS. This runs in $O(m+n)$ time.

4 Unique Requests

Suppose you're given a list of n web requests $r_1, \ldots, r_n$. Each request r_i contains the ID of the server to which the request was sent. You are tasked with finding the first request in the list which is sent to a unique server, or report that no such request exists. Design an algorithm that runs in expected $O(n)$ time. Prove it's correctness. You should assume that there are too many server IDs to store information about each server.

Construct a hash table with $\Theta(n)$ entries. Then, for each request $r_1, \ldots, r_n$, hash the server ID, and search for it in the hash table. If the ID is not present, add it to the hash table with an associated field `count`. Set `count` to 1. If the ID is present, add 1 to `count`. Then, go through the requests again in order from $r_1, \ldots, r_n$. Return the first r_i such that the `count` stored in the hash table is equal to 1, or return NIL if no such entry is found.

This algorithm requires two passes across an array of n entries, as well as two look-ups in a hash table for n entries. We assume the hash table guarantees simple uniform hashing (and that we resolve conflicts with chaining), so the expected run time of each operation is $O(\alpha) = O(1)$. Thus, the expected running time will be $O(n)$.

To see correctness, note that if a request goes to a unique server, its count will be set to 1 and never incremented. We process the requests from earliest to latest, so we will find the first of such requests first and return it. If there are no unique requests, then every request will have count of at least 2, and we will return NIL.

5 KT 6.27

The owners of an independently operated gas station are faced with the following situation. They have a large underground tank in which they store gas; the tank can hold up to L gallons at one time. Ordering gas is quite expensive, so they want to order relatively rarely. For each order, they need to pay a fixed price P, for delivery in addition to the cost of the gas ordered. However, it costs c to store a gallon of gas for an extra day, so ordering too much ahead increases the storage cost. They are planning to close for a week in the winter, and they want their tank to be empty by the time they close. Luckily, based on years of experience, they have accurate projections for how much gas they will need each day until this point in time. Assume that there are n days left until they close, and they need g_i gallons of gas for each of the days $i = 1, \ldots, n$. Assume the tank is empty at the end of day 0. Give an algorithm to decide on which days they should place orders, and how much to order so as to minimize their total cost.

For every day i from n to 1, we compute the cost $C[i]$ of the optimal purchasing strategy through the nth day if the tank is empty at the start of day i as follows. We start with $C[n+1] = 0$. We'll say that $j \in \mathsf{Legal}(i)$ if $i \leq j \leq n$ and $\sum_{k=i}^{j} g_k \leq L$. Then, we can compute the following recurrence for all $i = n, \ldots, 1$.

$$C[i] = P + \min_{j \in \mathsf{Legal}(i)} \sum_{k=i}^{j} (k-i)c \cdot g_k + C[j+1]$$

We claim that the cost of the optimal strategy is stored in $C[1]$. To see this, first note that at any point of purchase, an optimal strategy will buy enough to exhaust the entire purchased supply of gasoline at the end of some future day. That is, suppose at day i, gas is purchased to last through day j. Then in the optimal strategy, after day j, the tank will be empty. Clearly, if more gas was purchased than this (but not enough for the entire $j+1$st day) then we would still need to make a purchase on day $j+1$, again paying P as in the case when no gas for the $j+1$st day was purchased, but would have paid an additional extra storage cost, making the strategy suboptimal.

Thus, we've identified an optimal substructure – on the ith day, it suffices to pick the future legal day j that minimizes the storage cost of the fuel through day j plus the cost of starting with an empty tank on day j. This is precisely captured by our recurrence, so if we work backwards and eventually return $C[1]$, we will discover the optimal cost of purchasing. (If we want to actually discover the strategy, we can keep a "minimizer" pointer from i to the j that minimizes $C[i]$ for each i, which will allow us to reconstruct the strategy in $O(n)$ time.)

Because we compute $C[i]$ for n entries, and to compute the ith entry, we have to loop over at most every $j \in \mathsf{Legal}(i)$, we can upper bound the running time to compute $C[1]$ by $O(n^2)$.

6 k-Clustering

Suppose you're given n points in $\mathbb{R}^3$. We want to cluster these points into k groups such that we maximize the minimum distance between the separated points. That is, we want to find a partition of the points into $P_1, \ldots, P_k$ such that

$$\min_{\substack{p_i \in P_i, p_j \in P_j \\ i \neq j}} \|p_i - p_j\|$$

is maximized. Design an algorithm to find an optimal k-Clustering that runs in $O(n^2 \log n)$ time; prove it's correctness and runtime.

Consider creating a graph on $O(n)$ nodes corresponding to points and $O(n^2)$ edges with weight equal to the distance between the two corresponding points, and then running Kruskal's algorithm until $n - k$ edges have been added to the set of edges. Then, return the k resulting supernodes. We claim this is a k-clustering that maximizes the separation.

Because there are n^2 distances, the running time of Kruskal's will be dominated by sorting the distances so it will run in $O(m \log n) = O(n^2 \log n)$ time.

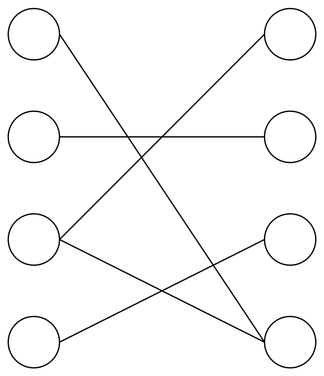
To see why this algorithm is correct, consider what we are trying to maximize. We want to obtain the partition $\mathcal{P} = P_1, \ldots, P_k$ that obtains

$$\max_{P_1,\ldots,P_k} \min_{\substack{p_i \in P_i, p_j \in P_j \\ i \neq j}} \|p_i - p_j\|$$

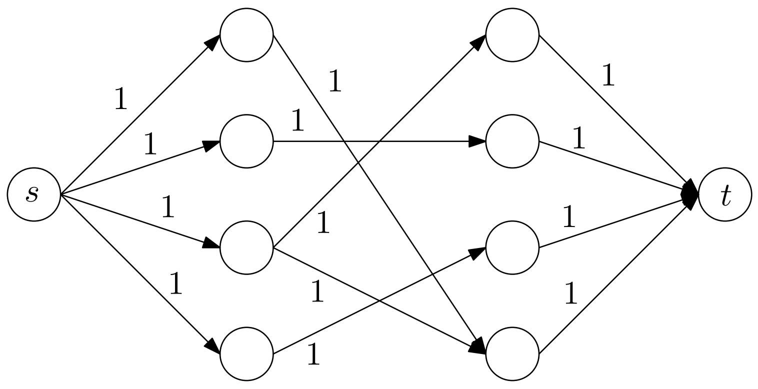
This is to say, we want to maximize the distance between the closest pair of points in different clusters. In our graph interpretation, this is equivalent to looking for the partition of the nodes that maximizes the minimum cut edge between partitions. Suppose the partition returned by Kruskal's with early stopping has separation of d, and suppose there is another k-partition $\mathcal{P}^\star = P_1^\star, \ldots, P_k^\star$ with $d' > d$ separation. Because $\mathcal{P} \neq \mathcal{P}^\star$, then we know that there is some pair of points u, v, where (u, v) was selected by Kruskal's to put them in the same cluster in $\mathcal{P}$, but where u and v are in different clusters in $\mathcal{P}^\star$. But by the inclusion rule of Kruskal's, this means that $w(u, v) \leq d$. But we also know that $w(u, v)$ is an upper bound on d' because u and v are in separate clusters in $\mathcal{P}^\star$. Thus, we get a contradiction, and our Kruskal-based clustering must be optimal.

7 Flows and Matchings

Consider the bipartite graph G below.

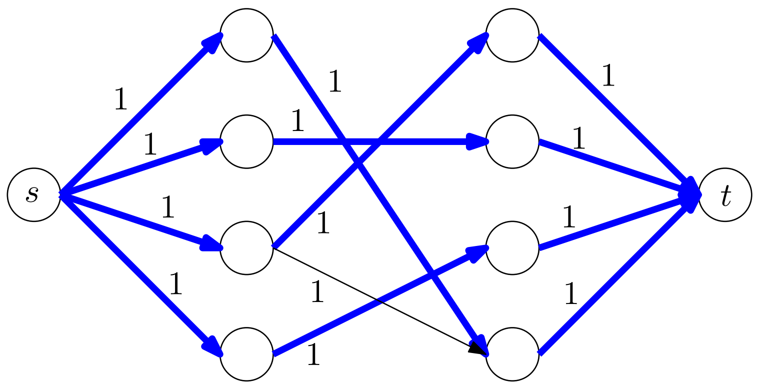


We want to find the maximum cardinality matching in G. Recall, that we can frame this as a max flow problem by reducing G to following directed graph G'.



(a) What is the maximum flow from s to t through G'?

4



(b) What is the weight of the minimum cut in G'?

4

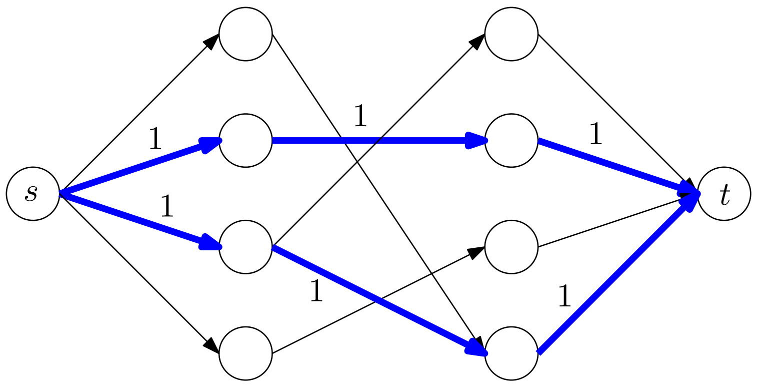
(c) How many augmenting paths will FORD-FULKERSON find while computing the maximum flow?

4 – one corresponding to each unit of flow

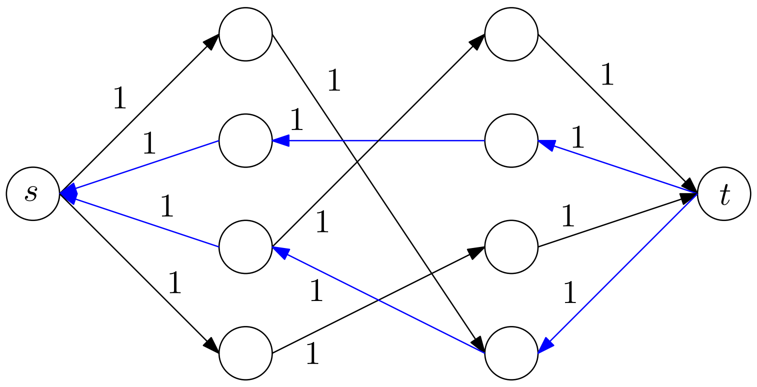
(d) Over all possible executions of FORD-FULKERSON, what is the longest augmenting path from s to t through the residual network G'_f?

5 edges

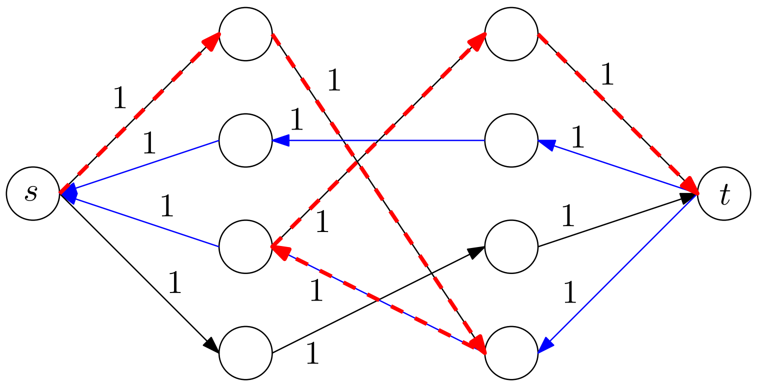
Consider what happens if the first two augmenting paths are selected to be the blue flow.



Then, consider the corresponding residual graph G_f.



Then, consider the following augmenting path through G_f. One can verify that there are no longer



augmenting paths through G_f for any flow f.