CS 170 Homework 5 (Optional)

1 Study Group

List the names and SIDs of the members in your study group. If you have no collaborators, explicitly write "none".

2 Adding Many Edges At Once

Given an undirected, weighted graph $G(V, E)$, consider the following algorithm to find the minimum spanning tree. This algorithm is similar to Prim's, except rather than growing out a spanning tree from one vertex, it tries to grow out the spanning tree from every vertex at the same time.

procedure FindMST($G(V, E)$)
  $T \leftarrow \emptyset$
  while T is not a spanning tree **do**
    Let $S_1, S_2 \dots S_k$ be the connected components of the graph with vertices V and edges T
    For each $i \in \{1, \dots, k\}$, let e_i be the minimum-weight edge with exactly one endpoint in S_i
    $T \leftarrow T \cup \{e_1, e_2, \dots e_k\}$
  return T

For example, at the start of the first iteration, every vertex is its own S_i.

For simplicity, in the following parts you may assume that no two edges in G have the same weight.

(a) Show that this algorithm finds a minimum spanning tree.

(b) Give a tight upper bound on the worst-case number of iterations of the while loop in one run of the algorithm. Justify your answer.

(c) Using your answer to the previous part, give an upper bound on the runtime of this algorithm.

Solution:

This algorithm is known as Boruvka's algorithm. Despite perhaps appearing complicated for an MST algorithm, it was discovered in 1926, predating both Prim's and Kruskal's. This algorithm is still of interest due to being easily parallelizable, as well as being the basis for a randomized MST algorithm that runs in expected time $O(|E|)$.

(a) If we add an edge e_i to T, it is because it is the cheapest edge with exactly one endpoint in S_i. So applying the cut property to the cut $(S_i, V - S_i)$ we see that e_i must be in the (unique) minimum spanning tree. So every edge we add must be in the minimum spanning tree, i.e. this algorithm finds exactly the minimum spanning tree.

This content is protected and may not be shared, uploaded, or distributed.

(b) The number of components in the graph with vertices V and edges T starts out at $|V|$, with one component for each vertex. If there are k components at the start of an iteration, we must add at least $k/2$ edges in that iteration: every component S_i contributes some e_i to the set of edges we're adding, and each edge we add can only have been contributed by up to two components. This means the number of components decreases by at least a factor of 2 in every iteration. Thus, the while loop runs for at most $\log |V|$ iterations.

(c) The previous part shows that at most $\log |V|$ iterations are needed. Each iteration can be performed in $O(|E|)$ time (e.g. you can compute the components in $O(|E|)$ time, and then initialize a table which stores the cheapest edge with exactly one endpoint in each component, and then using a linear scan over all edges fill out this table) giving a runtime bound of $O(|E| \log |V|)$.

This content is protected and may not be shared, uploaded, or distributed.

3 Minimum ∞-Norm Cut

In the Minimum Infinity-Norm Cut problem, you are given a connected undirected graph $G = (V, E)$ with positive edge weights w_e, and you are asked to find a cut in the graph where the largest edge in the cut is as small as possible (note that there is no notion of source or target; any cut with at least one node on each side is valid).

Solve this problem in $O(|E|\log|V| + |V| + |E|)$ time. **Give a 3-part solution.**

Hint: Minimum Spanning Tree does not require edge weights to be positive.

Solution: Algorithm: First, negate all the edge weights in G, and pass this new graph to Kruskal's minimum spanning tree algorithm; this will give us a maximum spanning tree of the original graph. Remove the smallest-weight edge in this maximum spanning tree, and return the cut induced by its removal.

Proof of Correctness: First note that we correctly find a maximum spanning tree of the graph, since $\max_{\text{tree } T} \sum_{e \in T} w_e = \min_{\text{tree } T} \sum_{e \in T} -w_e$. It is similarly easy to see that we find a minimum infinity-norm cut in the maximum spanning tree of the graph (using any other edges from the spanning tree could not decrease the cut, since we used only the smallest possible edge).

It only remains to show that any cut of the nodes in the graph has exactly the same infinity norm in the graph overall as in the maximum spanning tree; this will prove the correctness of our algorithm (since we have already proven its correctness in the tree). To see this, we will use the cut property for maximum spanning trees (which follows immediately from the cut property for minimum spanning trees, applied to the negated graph). This property is: for any cut in the graph, its largest edge (or one of its largest edges, if the largest edge is not unique) must be contained in the maximum spanning tree. Since the infinity norm of the cut is equal to the weight of its largest edge, this means that the infinity norm of the cut in the tree is at least its infinity norm in the graph; it is also at most the infinity norm in the graph, since no edges exist in the tree which do not exist in the graph.

Runtime Analysis: Creating a new graph with every edge negated takes $O(|V|+|E|)$ time. Once we have the maximum spanning tree, the smallest-weight edge can be found with a simple $O(|E|)$ time search; once it is removed, the nodes in the two resulting components can be enumerated with a $O(|V| + |E|)$ time traversal. So overall, we have taken linear time to convert this problem to an MST problem.

We can then run Kruskal's algorithm in $O(|E|\log|V|)$ time to find the MST. Finding the smallest-weight cut requires traversing over all edges. Hence, the final runtime is $O(|E|\log|V|+|V| + |E|)$.

Alternative solution

Sort the array of edge weights and binary search for the largest edge in the cut. For an edge with weight w_m, consider G', the graph obtained by keeping only the edges from E with weight $\leq w_m$. If G' is connected, then the answer is $\geq w_e$. Otherwise, the answer is $< w_m$.

Once you find the ∞ norm w_e of the cut, you can recover the cut by taking any connected

This content is protected and may not be shared, uploaded, or distributed.

component and its complement after removing all edges of weight $>= w_e$. Overall runtime is $O(|E|log|V|)$.

This content is protected and may not be shared, uploaded, or distributed.

4 Penguin Fishing

PNPenguin has just had his 170th birthday, and to celebrate this great milestone, the penguins are planning a feast to celebrate PNPenguin's birthday for this year and future years, and they need to catch fish to serve at this feast.

The PNPenguins live in Antarctica, which can be represented as N ice floes and M two-way bridges each connecting two ice floes. Some of the ice floes contain fish. The penguins can walk much faster than they can swim, and their goal is to station a certain number of penguins in various places, so that they can catch all of the fish. Hiring penguin fishermen is expensive, so the penguins want to catch all the fish using as few penguin fishermen as possible. Each penguin fisherman can only travel between ice floes connected by a bridge; however, they will be able to catch all the fish in the ice floes they can travel between.

Unfortunately, due to global warming, the penguins' habitat is slowly melting. For each of the next N years, an ice floe m_i will melt. When an ice floe melts, if the ice floe has fish, the fish at that ice floe will remain, but all of the bridges connecting that ice floe to other floes will disappear. Knowing this, write an efficient algorithm asymptotically faster than $O(N^2)$ to find the number of penguins p_i needed to catch all the fish in year i for each of the next N years.

Please provide a 3-part solution.

Solution:

Algorithm: Process the years in reverse order, and use a union-find data structure. Represent each ice floe as a vertex, initially with no edges in the graph. For every year, in reverse order, go through all the edges connecting the vertex m_i, as from our standpoint, these edges will now be added back to the graph, instead of disappearing. Each time we add an edge back, we will unite the two components using union-find, if they weren't already in the same component.

Keep a running count c of the number of components in the graph with fish. Initially, this will be the total amount of ice floes containing fish. Additionally, we store a boolean array f where $f[i]$ represents whether or not component i contains fish, only applicable for i that are the parent of their respective component. Every time we add an edge to the graph with our union-find data structure, we run into one of several cases, based on the vertices that the edge connects:

1. The two vertices are already in the same component. In this case, we do nothing
2. The two vertices are not in the same component, and neither components contain fish. In this case, we also do nothing
3. The two vertices are not in the same component, and one of the two components contain fish. In this case, we make sure to update the $f[parent(newcomponent)]$ to be true, if it wasn't already
4. The two vertices are not in the same component, and both components contain fish. In this case, we decrement c by one.

This content is protected and may not be shared, uploaded, or distributed.

Before each year, add the current value of c to an array. After all N years, reverse this array, and return it as the answer.

Proof of Correctness: The minimum number of penguins we need to send is equivalent to the number of connected components in the graph with at least one fish. This is sufficient, because if we send one penguin to each such component, we will definitely be able to catch all the fish, and it's necessary, because if we leave at least one connected component with nonzero fish without any penguins, it will be impossible to catch any of the fish in that component. We look "backwards in time" to simulate the melting, as removing edges is difficult to simulate with union-find (it'd require a persistent union-find data structure, which is out of scope for the course), but adding edges is easier. Before adding any edges, every vertex will be isolated, so the answer will simply be the amount of total vertices with fish, as we'll have to send a penguin to each one. For future weeks, we only care about the new edges that we're adding in by adding a new vertex, and for each such edge, it'll only change the value c if both components that we're uniting have fish.

Runtime: Union-find takes $O(log(n))$ per find operation, and $O(log(n))$ per union operation, and we'll do this $max(T, M)$ times. We'll also process each of the N vertices once at the start, making the total runtime $O(N + Mlog(N))$.

This content is protected and may not be shared, uploaded, or distributed.

5 Preventing Conflict

A group of n guests shows up to a house for a party, but the host knows that m pairs of these guests are enemies (a guest can be enemies with multiple other guests). There are two rooms in the house, and the host wants to distribute guests among the rooms, breaking up as many pairs of enemies as possible. The guests are all waiting outside the house and are impatient to get in, so the host needs to assign them to the two rooms quickly, even if this means that it's not the best possible solution. Come up with a $O(n+m)$-time algorithm that breaks up at least half of the pairs of enemies.

Give a 3-part solution.

Solution:

Main Idea: We add the guests one-by-one, each time adding a guest to the room that currently has less of their enemies (in the case of a tie, we choose one of the rooms arbitrarily).

Proof of Correctness: Let m_i be the number of the ith guest's enemies that have already been added to a room when we're adding the ith guest. We break up at least $m_i/2$ of these enemy pairs by adding this guest to the room with less of their enemies. So we break up at least $\sum_i m_i/2$ of the $\sum_i m_i$ total enemy pairs as desired.

Running Time: Each guest is iterated through once, and we can figure out which room to assign them in time proportional to the number of enemies they have (using e.g. an array tracking which room we've assigned each individual to so far), i.e. each pair of enemies is considered at most twice, so the total time is $O(n+m)$.

(Comment: This problem is equivalent to the max-cut problem: Given an undirected graph, in the max cut problem we want to split the vertices into two sets so that as many edges as possible have one endpoint in each set)

This content is protected and may not be shared, uploaded, or distributed.

6 Twenty Questions

Your friend challenges you to a variant of the guessing game 20 questions. First, they pick some word $(w_1, w_2, ..., w_n)$ according to a known probability distribution $(p_1, p_2, ..., p_n)$, i.e. word w_i is chosen with probability p_i. Then, you ask yes/no questions until you are certain which word has been chosen. You can ask any yes/no question, meaning you can eliminate any subset S of the possible words with the question "Is the word in S?".

Define the cost of a guessing strategy as the expected number of queries it requires to determine the chosen word, and let an optimal strategy be one which minimizes cost. Design an $O(n \log n)$ algorithm to determine the cost of the optimal strategy.

Give a 3-part solution.

Note: *We are only considering deterministic guessing strategies in this question. Including randomized strategies doesn't change the answer, but it makes the proof of correctness more difficult.*

Solution:

This solution is inspired by the observation that in binary coding, each bit of a codeword we read further narrows the possible symbols being encoded, just like a question in the game above. This correspondence is made rigorous in the proof of correctness.

Main idea: Create a Huffman tree on $(w_{1\cdots n})$ with weights $(p_{1\cdots n})$ and return the expected length of a codeword under the corresponding encoding.

Proof of correctness: Note that any guessing strategy gives a prefix-free binary encoding of the words $(w_{1\cdots n})$, where each word w_i is encoded by sequences of yes/no answers which would lead you to conclude that w_i was chosen. This encoding is prefix-free because the game only ends when all words except one have been eliminated.

Additionally, any prefix-free encoding of the words can be made into a guessing strategy as follows. Let $x_n \in \{0,1\}^n$ represent the sequence of yes/no answers received on the first n questions, with 1 corresponding to yes and 0 corresponding to no. Then asking at step $n+1$ whether the word can be encoded by a string with the prefix $x_n \circ 1$ (i.e. the answers so far followed by a yes) will result in the final sequence of answers being a valid encoding of the chosen word.

In this correspondence, the expected code length equals the cost of a guessing strategy. Therefore finding an optimal strategy is equivalent to finding a prefix-free encoding of the words with minimum expected codelength, which is exactly what Huffman coding does. To get the final answer, we calculate the expected codelength of the optimal strategy, which can be done by DFS from the root of the Huffman tree.

Runtime: A Huffman tree can be constructed in $n \log(n)$ time (this is dominated by the time to sort the probabilities). The average codelength (and thus cost of the associated strategy) can be calculated in $O(n)$ time by DFS. Therefore the total runtime is $O(n \log(n))$.

This content is protected and may not be shared, uploaded, or distributed.

7 Rigged Tournament

Peter is in charge of organizing a football tournament with n teams. The tournament is a single-elimination tournament: if teams i and j play, the team that loses is out of the tournament and cannot play any more games. There are no ties.

Peter's shady friend Jeff has given him the following inside information: if teams i and j play, then they will score a combined total of $f(i, j) \geq 0$ points in that game and furthermore Jeff can rig the match so that the team of his choice wins. Peter wishes to find a tournament schedule which (1) maximizes the number of points scored in the tournament, and (2) makes his favorite team, team i^*, win the tournament. Give an efficient algorithm to solve this problem and provide its runtime; proof of correctness is not required.

Note: teams need not play an equal number of games in the tournament. For example, if the teams are $\{1, 2, 3, 4\}$, then a valid tournament schedule (where (i, j) means i plays j and i wins) is $[(1, 2), (1, 3), (4, 1)]$. Here, team 4 wins the tournament.

Solution: Let $G = (V, E)$ be the complete graph on n vertices with edge weights $\ell(i, j) = f(i, j)$. The key observation is that a tournament schedule corresponds to a tree in this graph, and the weight of the tree is the number of points scored across all games in that tournament schedule. Therefore, let T be the Maximum Spanning Tree in G. The weight of T is the maximum number of points scored in the tournament. To extract the sequence of games, simply run DFS or BFS from i^*, and play the games by largest depth first, making the team with a lower depth always win. For the runtime, DFS/BFS takes O(n) time, which means the MST algorithm is the bottleneck. This takes $O(|E| \log |V|) = O(n^2 \log n)$ since there are $O(n^2)$ edges (the complete graph).

This content is protected and may not be shared, uploaded, or distributed.

8 Sum of Products

This question guides you through writing a proof of correctness for a greedy algorithm. You have n computing jobs to perform, with job i requiring t_i units of CPU time to complete. You also have access to n machines that you can assign these jobs to. Since the machines are in high demand, you can only assign one job to any machine. The jth machine costs c_j dollars for each unit of CPU time it spends running a job, so assigning job i to machine j will cost you $t_i \cdot c_j$ dollars (each job takes the same amount of CPU time to complete, regardless of which machine is used). Your goal is to find an assignment of jobs to machines that minimizes the total cost.

Assume the jobs and machines are sorted and have distinct runtimes/costs, i.e. $t_1 > t_2 > \ldots > t_n$ and $c_1 > c_2 > \ldots > c_n$.

(a) Describe the assignment of jobs to machines minimizes the total cost (no proof necessary).

Hint: What machine should we assign the longest job to? What machine should we assign the second longest job to? It might help to solve a small example by hand first.

(b) Given an assignment of jobs to machines, consider the following modification: If there is a pair of jobs i, j such that job i is assigned to machine i', job j is assigned to machine j', and $t_i > t_j$ and $c_{i'} > c_{j'}$, instead assign job i to machine j' and job j to machine i'. Show that this modification decreases the total cost of an assignment.

(c) Use part b to show that the assignment you chose in part a has the minimum total cost (Hint: Show that for any assignment other than the one you chose in part a, you can apply the modification in part b. Conclude that the assignment you chose in part a is the optimal assignment.)

Solution:

(a) Assign the longest job (job 1) to the cheapest machine (machine n), the second longest job (job 2) to the second cheapest machine (machine $n-1$), etc.

(b) The cost of computing jobs other than i and j is unaffected. The old cost of computing jobs i and j is $t_i c_{i'} + t_j c_{j'}$. The new cost of computing jobs i and j is $t_i c_{j'} + t_j c_{i'}$. The decrease in cost is then $t_i c_{i'} + t_j c_{j'} - t_i c_{j'} - t_j c_{i'} = (t_i - t_j)(c_{i'} - c_{j'})$, which is positive because $t_i > t_j$ and $c_{i'} > c_{j'}$.

(c) Suppose for an assignment of jobs to machines that no modification of the type suggested in part b is possible. Then job 1 must be assigned to machine n in this assignment - otherwise job 1 and whatever job is assigned to machine n satisfy the conditions in part b. But if we know job 1 is assigned to job n, then we can similarly conclude that job 2 must be assigned to machine $n-1$, and so on. So this assignment must be exactly the assignment we chose in part a.

Then for every assignment except the assignment we found in part a, there is another assignment that has lower total cost, i.e. the former is not optimal. This implies that the assignment from part a is the optimal assignment.

This content is protected and may not be shared, uploaded, or distributed.

9 Counting Shortest Paths

Given an undirected unweighted graph G and a vertex s, let $p(v)$ be the number of distinct shortest paths from s to v. We will use the convention that $p(s) = 1$ in this problem. Give an $O(|V| + |E|)$-time algorithm to compute $p(v) \bmod 1700$ for all vertices. Only the main idea and runtime analysis are needed.

Hint: For any vertex v, how can we express $p(v)$ as a function of other $p(u)$?

Note: As a secondary question, you should ask yourself whether the runtime would remain the same if we were computing $p(v)$ rather than $p(v) \bmod 1700$.

Solution: Main idea If v is distance $d > 0$ from s, the first $d-1$ edges in any shortest path to v will be a shortest path to a neighbor of v distance $d-1$ from s. Furthermore, this mapping from shortest paths to v and shortest paths to neighbors of v is bijective. So letting $N(v)$ be the set of neighbors of v at distance $d-1$, we get that $p(v) = \sum_{u \in N(v)} p(u)$.

Our algorithm is now: use BFS to compute all distances from s. Next, we set $p(s) = 1$, then for the remaining vertices in increasing distance order, we can compute $p(v) \bmod 1700 = \sum_{u \in N(v)} p(u) \bmod 1700$. Since we look at vertices in increasing distance order, all u in $N(v)$ have $p(u)$ computed already.

Runtime analysis BFS takes $O(|V| + |E|)$ time. Computing $p(v) \bmod 1700$ from its neighbors takes time $O(deg(v))$, so the total time to compute all $p(v) \bmod 1700$ is $O(|E|)$.

If we did not have the $\bmod 1700$, and we instead wanted $p(v)$ exactly, we would have a higher runtime, as the number of length-d paths to v can be exponential in d and so arithmetic on these numbers would not be constant time.

This content is protected and may not be shared, uploaded, or distributed.