Table of Contents

Topic Review for Finals

Gale-Shapley and Stable Matching

Full list of preferences, no ties. Gale-Shapley returns a perfect and stable matching. Men are matched to their best valid partners. Women are matched to their worst valid partners. The runtime is $O(n^2)$, n being the number of men or the number of women, which are equal. Important things to know for proving stuff is that men go down their list of preferences, and women's offers get better.

$O(n^2)$

Greedy Algorithms

Unweighted Interval Scheduling

Input is a set of requests (time intervals). Goal is to return a set of non-overlapping intervals of maximum cardinality possible. The size of each interval doesn't matter, we just want as many non-overlapping intervals as possible. Selecting earliest start time then deleting conflicts doesn't work, selecting by increasing number of overlaps then deleting conflicts doesn't work, selecting by increasing interval length and deleting conflicts doesn't work. The algorithm is to select by **earliest finish time**! You can prove this by the exchange argument and induction.

$O(n \log n)$.

Minimum Spanning Tree

Kruskal

Sort all edges in increasing order of cost. Pick the smallest edge. If it forms a cycle with the tree we have so far, throw it out. Else, include it. Repeat that until all nodes are included.

$O(E \log E)$ if using the disjoint set data structure.

Prim

Start from random node. Check outgoing edges of the tree we have so far, pick the one with the lowest cost. If it creates a cycle, throw it out. Else, keep it. Repeat that until all nodes are included. If all weights in a graph are unique, then there is a unique minimum spanning tree, so all MST algorithms returns the same thing. Just a useful thing to know, the time complexity of DFS is $O(V + E)$.

$O(E \log E)$ if using priority heaps to store edges.

Borůvka

Initialize all vertices to be a component. While there is more than one component (i.e. no spanning tree yet), find the cheapest edge for each component and add it to our current trees (this is done in parallel for all components). Requires distinct edge weights. Because imagine a triangle with the same weight on each edge. Each one can choose the edge to the left and we would end up with a cycle.

$O(E \log E)$

Cut Property
It is a way to prove MST algorithms. The claim could be that given node v and edge $e = (v, w)$ that is the cheapest between the set S it's in and $V - S$, e is included in every MST. The cut property says that assume the optimal solution doesn't have e, but has e' that connects S to $V - S$. There must be a path from v in S to w in $V - S$. Then adding e to this optimal MST creates a cycle. Just remove e' and add in e to create a new solution. This solution is cheaper or not more expensive than O.

Huffman Coding
Not done in homework or practice sets. Do not need to know proof. Just need to know reduction to other problems.

We want binary representations for each letter in a lexicon, want shorter strings for more frequently used letters and longer strings for the less frequently used ones. Also! (This is important.) We don't want a letter's coding to be the prefix to another letter's coding, i.e. if a's code is 10, then other letter's code can't start with 01, like c can't be 101. Another objective is to have the average length of all codes as short as possible.

The answer is to use trees, with the nodes as the letters traversed. This way, one letter's code can't be another letter's prefix. A more frequent letter appears shallower (upper) in the tree. What they want us to know if we're *given* a tree, just know to assign the letters to increasingly deeper leaves in decreasing frequency. They also want us to know how to construct a tree given a table of letters and their frequencies. This is done by picking the two least frequent letters or components and making them siblings in the tree. Then take the sum of their frequencies and make the frequency on their parent that. Repeat.

$O(n \log n)$.

Dynamic Programming

Weighted Interval Scheduling
Interval scheduling, but with value associated to each interval. Want compatible set of intervals of greatest value. Sort by finishing time first! This is important! All recurrence schemes are useless without some metric of order. Interval n could be or not be included. The same problem has to be solved on the previous $n - 1$ intervals. Define $p(i)$ to be the latest interval that doesn't conflict with interval i. If it is included, then the latest interval that's also there is the latest that doesn't overlap. If it isn't included, then the latest one would be the previous in the ordered sequence. The recurrence scheme is $\max[v_{i-1}, v_i + v_{p(i)}]$, base case is $v_0 = 0$.

$O(n \log n)$.

Segmented Least Squares
Not done in homework or practice sets. Do not need to know proof. Just need to know reduction to other problems.

Given a bunch of points and a real number $c > 0$, you want to fit them into a number of lines of best fits. For each line of best fit, there's an error of fitting the points to the line. The goal is minimizing the penalty $c|S| + \sum_{i \in S} error$, S being the segments. The way to think about this is that if the last segment of the optimal solution is $p_i \dots p_n$, then the value of the optimal solution is $error_{i,n} + c + opt(i-1)$. The recurrence is thus $opt(j) = min_{1 \le i \le j}[error_{i,j} + c + opt(i-1)]$. The base case is $opt(0) = 0$.

$O(n^3)$.

Sequence Alignment

Not done in homework or practice sets. Do not need to know proof. Just need to know reduction to other problems.

Given two finite-length strings x and y from the same alphabet. Matchings, if represented as lines connecting matching letters between the two strings, cannot intersect. I.e. A matching is a pair (i, j) matching x_i with y_j such that if there's two matchings, (i, j) and (i', j'), that if $i < i'$ then $j < j'$. The main objective is that given $\alpha_{a,b} \ge 0$ meaning the cost of changing an a to a b and $\delta_a \ge 0$ meaning the cost of inserting or deleting an a, we want to minimize costs of editing for x and y to match. Base cases are $opt(\epsilon, \mathrm{y}) = \sum_{\mathrm{i}=1}^{|\mathrm{y}|} \delta_{y_i}$ and $opt(\mathrm{x}, \epsilon) = \sum_{\mathrm{i}=1}^{|\mathrm{x}|} \delta_{x_i}$. Recurrence scheme: $\min[\alpha_{x_i,y_j} + opt(i-1, j-1), \delta_{x_i} + opt(i-1, j), \delta_{y_j} + opt(i, j-1)]$ because there's either a matching between x_i and y_j, or x_i is not matched with anything in y, or y_j is not matching with anything in x.

$O(|x||y|)$.

Bellman-Ford

Given a graph and two nodes from it, s and t, find the (value of) the shortest path from s to t. Having negative cycles in graphs is impossible to get a shortest path, because you will just loop on it forever instead of moving forward to the goal. Dijkstra's algorithm will not work on graphs with negative weight edges (not even negative cycles, just weights), because it relies on the simple fact that adding a path can never make a path shorter; when there's negative edges, a myopic cheap edge might lure you down a path of positive weights while an expensive one could hide negative edges behind. The closed node will never have to be reopened in an all-positive graph. The base cases for Bellman-Ford are $opt(0, t) = 0$ and $opt(0, v) = \infty$ for all nodes other than t. The recurrence scheme is $opt(i, v) = \min[opt(i-1, v), min_{w \in V}(opt(i-1, w) + c_{vw})]$, where the two arguments here mean the it's the cost of the shortest path from v to t using at most i edges. The recurrence scheme is as such because you either use at most $i - 1$ edges, or you use i edges and the node immediately before is w.

$O(n^3)$.

Divide and Conquer

The Master Method

$$T(n) = aT\left(\frac{n}{b}\right) + f\left(n^d\right) \quad \text{where } a \ge 1, b > 1$$

n is the size of the problem.

a is the number of subproblems in the recursion.

$\frac{n}{b}$ is the size of each subproblem.

$f(n^d)$ is the cost of the work done outside the recursive calls, which includes the cost of dividing the problem and the cost of merging the solution to the subproblems.

Case $a = b^d$
$O(n^d \log n)$

Case $a < b^d$
$O(n^d)$

Case 3 $a > b^d$
$O(n^{\log_b a})$

Integer Multiplication
Multiplying two numbers x and y, each of n places. The solution is as follows. Write x as $x_1 \times 10^{\frac{n}{2}} + x_0$, write y similarly as $y_1 \times 10^{\frac{n}{2}} + y_0$. Compute $p = multiply(x_1 + x_0, y_0 + y_1)$. Compute $x_1 y_1 = multiply(x_1, y_1)$, $x_0 y_0 = multiply(x_0, y_0)$. The answer is $x_1 y_1 \times 10^n + (p - x_1 y_1 - x_0 y_0) \times 10^{\frac{n}{2}} + x_0 y_0$.

$O(n^{\log_2 3})$ because a is 3, b is 2, and d is 1.

Convolution
Only to recognize a convolution problem, not details of how to run the algorithm.

Given two vectors $a =< a_0 \dots a_{m-1} >$ and $b =< b_0 \dots b_{n-1} >$, want

$$\sum_{\substack{(i,j):\, i+j=k \\ i<m, j<n}} a_i b_j$$

Randomized Algorithms

Linear-Time Median Finding
We pick a pivot at random at each recursive step. We want the pivot to be "good," that at least $\frac{3}{4}n$ elements in the array is less than it and at least $\frac{3}{4}n$ elements is greater than it. The expected number of of repeats to get such a pivot in a recursive step is 2. Using the Master method on this produces $T(n) = 1 \times T\left(\frac{3n}{4}\right) + O(n)$, as each subproblem is at the biggest $\frac{3n}{4}$.

$O(n)$

Hashing
The goal of hashing is to have a good function $h: U \to [0 \dots n-1]$ such that it stores $x \in S$ at position $h(x)$ and have very few collisions. Formally, a universal hash function is one where the probability of a collisions is $\leq \frac{1}{n}$. Making a universal hash function involves picking a prime number $p > |S|$.

$$H = \{select\ \overline{a} = (a_0 \dots a_{m-1}) \in [0 \dots p-1]\ h_{\overline{a}}(x) = \sum_{i=0}^{m-1} a_i x_i \ mod\ p\}$$

Probability of a collision here is $\frac{1}{p}$.

Prime Testing

The probability of a number of b digits being prime is $\frac{1}{b}$. Fermat's theorem is that if n is prime, then $\forall a \in [1 \dots n-1], a^{n-1} \equiv 1 \ (mod\ n)$. So we use the contrapositive, that if $\exists a \in [1 \dots n-1], a^{n-1} \neq 1$, then n is not prime.

Network Flow

Ford-Fulkerson

Given a graph with directed edges, each with a capacity ≥ 0, we want to send as much flow as possible from s to t. Forward flow subtracts from the capacity. "Backward flow" adds to the capacity. At each step in the algorithm, choose any path from s to t and update the residual graph accordingly. It terminates when there are no more such paths. It is important that all capacities and flows are integers! We need to say that in proofs, too.

$O(mC)$, where C is the sum of the capacities leaving s and m is the number of edges.

Min Cut

The max-flow min-cut theorem says the maximum flow is equal to the total capacities of the edges in the minimum cut, i.e. the smallest total weight of the edges which if removed would disconnect the source from the sink.

The capacity of a cut A and B such that $s \in A$ and $t \in B$ is the sum of the capacities of edges leaving A. There is no subtracting capacities of edges entering A! To get the minimum cut, run Ford-Fulkerson and get the final residual graph, then A is the set of all nodes that you can reach from s.

If a flow's value equals a cut's capacity, then that is the max flow and that is the min cut. Because that's the biggest flow for that cut, and that is the smallest cut for that flow.

Applications

Disjoint Paths

Given a directed or undirected graph, we want to find the maximum number of edge-disjoint paths in it. There are k edge-disjoint graphs in a directed graph if and only if the max flow from s to t (minimum number of edges whose removal separates s from t) is $\geq k$.

Project Selection

We're given a set of projects, each with a profit, and each of which might have prerequisites, and each of those has a cost (negative profit). What is the set of projects to do (along with their required prerequisites) such that profit is maximized? The solution is to find a min cut in a graph constructed as such: connect s to all projects that generate a profit (don't consider the prerequisites, just itself) and give them their profit as edge capacity; connect projects to their prerequisites with edges of infinite capacity (this is to make sure a cut doesn't cross these edges, so as to preserve prerequisite structure); and connect all negative profit projects to t with capacity $|profit|$ (so, a positive number).

NP-Completeness

SAT problem

A bunch of conjunctions (ands) of clauses, each clause is a bunch of disjunctions (ors). Is there an assignment for all the variables to satisfy the whole thing?

Independent Set

Given a graph, is there a set of nodes of size k that aren't connected to each other?

Vertex Cover

Given a graph, is there a set of nodes whose incident edges touch all nodes in the graph?

Set Cover

Given a set of sets, is there a subset of the larger set of size k such that their union is the union of all the sets in the larger set?

Hamiltonian Path

A Hamiltonian path is a path between two vertices of a graph that touches every vertex. If the two vertices are adjacent, then it's a Hamiltonian cycle.

Traveling Salesman Problem

Goal is to get a Hamiltonian cycle, but of the cheapest edge cost.

Computability

Acceptance means the program terminates and returns true. Rejection means the program terminates and outputs false. Or a program will never terminate on an input. M recognizes L if for everything in L, M accepts it. M decides L if for everything in L, M accepts it and if for everything not in L, M rejects it.

Halting Problem

$\{L|M\ halts\ on\ L\}$

Co-Halting Problem

$\{L|M\ doesn't\ halt\ on\ L\}$

Accept Problem

$\{L|M\ accepts\ L\}$

Approximation Algorithms

Greedy Methods

Knapsack

Given a set of things with weight and value each > 0 and a weight limit, the goal is to select a subset of the things whose total weight don't exceed the weight limit and whose total value is maximized. The 2-approximation ($\frac{1}{2}Opt \leq A$), A being our answer. You do greedy twice, once sorted by density $\frac{v}{w}$, once by value and take care not to exceed the weight limit.

$O(n \log n)$

Weighted Set Cover

Always select $\min \frac{w_i}{|S_i \cap R|}$ among sets. This is an $H(d)$ approximation, where d is the size of the biggest set. $H(d)$ is the harmonic series sum $\sum_i \frac{1}{i}$. I.e. $\frac{1}{H(d)} Opt \leq A$, where A is our solution.

$O(\log n)$

Linear Programming

Weighted Vertex Cover

Each vertex now has a value. Goal is to get a vertex cover of maximum value. We can get a 2-approximation ($\frac{1}{2} Opt \leq A$, A being our solution) using linear programming. Objective function to minimize is $\vec{w} \cdot \vec{x}$, subject to the constraints $\vec{x} \in \boldsymbol{R}^n$ and $\forall$ vertices $u, v, x_u + x_v \geq 1$. Linear programming is polynomial runtime complexity. If $x_i \geq 0.5$, round it up to 1 and include it in the vertex cover; round down to 0 and don't otherwise.

Polynomial runtime complexity.