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 , 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.

# 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.

.

## 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.

Kruskal can be in runtime complexity 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 .

Prim can be in runtime complexity 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.

Borůvka can be in runtime complexity.

## 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.

Huffman coding’s runtime complexity is .

# 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 could be or not be included. The same problem has to be solved on the previous intervals. Define to be the latest interval that doesn’t conflict with interval . 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 , base case is .

.

## 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 , 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 , being the segments. The way to think about this is that if the last segment of the optimal solution is , then the value of the optimal solution is . The recurrence is thus . The base case is .

Segmented least squares’ runtime complexity is .

## 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 and 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 matching with such that if there’s two matchings, and , that if then . The main objective is that given meaning the cost of changing an to a and meaning the cost of inserting or deleting an , we want to minimize costs of editing for and to match. Base cases are and . Recurrence scheme: because there’s either a matching between and , or is not matched with anything in , or is not matching with anything in .

Sequence alignment’s runtime complexity is .

## Bellman-Ford

Given a graph and two nodes from it, and , find the (value of) the shortest path from to . 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 and for all nodes other than . The recurrence scheme is , where the two arguments here mean the it’s the cost of the shortest path from to using at most edges. The recurrence scheme is as such because you either use at most edges, or you use edges and the node immediately before is .

Bellman-Ford’s runtime complexity is .

# Divide and Conquer

## Integer Multiplication

meow

## Convolution

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

# Randomized Algorithms

## Linear-Time Median Finding

meow

## Hashing

meow

## Prime Testing

# Network Flow

## Ford-Fulkerson

meow

## Min Cut

meow

## Applications

meow

# NP-Completeness

## SAT problem

meow

## Independent Set

meow

## Vertex Cover

meow

## Set Cover

meow

## Hamiltonian Path

meow

## Traveling Salesman Problem

meow

# Computability

## Halting Problem

meow

## Co-Halting Problem

meow

## Accept Problem

meow

# Approximation Algorithms

## Greedy Methods

### Knapsack

meow

## Linear Programming

### Vertex Cover

meow