CS 180(Algorithms) Notes

Chapter 1 - Introduction: Some Representative Problems

- Stable matching problem is where you are given a set of n men and n women, and you need to find a suitable matching given that each man ranks his preferences of women and each woman ranks her preferences of men.
- A matching is a set of ordered pairs of one man and one woman for each pair.
    - A matching S is perfect if the number of ordered pairs equals the n (the number of men and the number of women).
- An unstable pair is when given a man m and a woman w, m prefers w to his current partner and w prefers m to her current partner.
- A stable matching is a perfect matching with no unstable pairs.
- Stable matchings don't need to exist for the roommate problem where you have 2n people, and each person ranks roommates from 1 to 2n - 1.
- The Gale-Shapley algorithm guarantees to find a stable matching.

```
GALE–SHAPLEY (preference lists for men and women)

INITIALIZE S to empty matching.
WHILE (some man m is unmatched and hasn't proposed to every woman)
  w ← first woman on m's list to whom m has not yet proposed.
  IF (w is unmatched)
    Add pair m–w to matching S.
  ELSE IF (w prefers m to her current partner m')
    Remove pair m'–w from matching S.
    Add pair m–w to matching S.
  ELSE
    w rejects m.

RETURN stable matching S.
```

- The Gale-Shapley algorithm terminates after at most n squared iterations of the while loop.
- The set S returned at termination of the algo is a perfect and stable matching.
- The G-S algo runs asynchronously and all executions of the algo yield the same matching and a man-optimal assignment and subsequently a woman pessimal stable matching.
- 5 Representative Problems
    - Interval Scheduling - Having a set of jobs with start and end times, and finding the maximum subset of jobs that don't overlap (Aka finding the most jobs that you can fit in your schedule.)

    - Weighted Interval Scheduling
    - Bipartite Matching - Given a bipartite graph, find a max cardinality matching.
    - Independent Set - Given an independent set, find a max cardinality independent set.
    - Competitive Facility Location

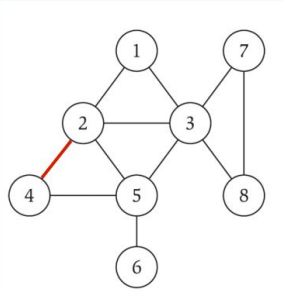
Chapter 2 - Basics of Algorithm Analysis

- For many nontrivial problems, there is a natural brute-force search algorithm that checks for every possible solution.
    - Normally takes 2^n time or worse, so it's unacceptable in practice.
- An algorithm is polynomial time if when the input size doubles, the algorithm only slows down by a constant factor C.
- If an algorithm is poly-time, it is called efficient.
- Worst case running time is the bound on the largest possible running time the algorithm could have over all inputs of size N.
- Types of algorithm analysis:
    - Worst case: Running time guarantee for any input of size n
    - Probabilistic: Expected running time of a randomized algorithm.
    - Amortized: Worst case running time for any sequence of n operations.
    - Average-case: Expected running time for a random input of size n.
- An algorithm is efficient if it achieves qualitatively better worst-case performance than brute-force search and if it has a polynomial running time.
- **Big O**: Let T(n) be the worst-case running time of a certain algorithm on input of size n.
    - We can say T(n) = O(f(n)) if T is asymptotically upper bounded by f and there exists some constant c such that T(n) <= c * f(n) for all n
    - Ex) pn^2 + qn + r = O(n^2) or O(n^3) although the first is more precise.
- **Big Omega**: We can say T(n) = omega(f(n)) if T is asymptotically lower bounded by f.
    - We can say T(n) = O(f(n)) if T is asymptotically lower bounded by f and there exists some constant c such that T(n) >= c * f(n) for all n
    - Ex) pn^2 + qn + r = omega(n^2) or omega(n) although the first is more precise.
- **Big Theta**: We can say T(n) = theta(f(n)) if there exists some constants c1 and c2 such that c1 * f(n) <= T(n) <= c2 * f(n). Also f(x) = Θ(g(x)) iff f(x) = O(g(x)) and f(x) = Ω(g(x))
    - Ex) pn^2 + qn + r = theta(n^2).
- If T(n) = O(f(n)) and omega(f(n)), then T(n) = theta(f(n))
    - Basically, if lim as n goes to infinity of (T(n) / f(n)) = some constant
- Common runtimes
    - Constant time O(1): Constant number of steps every time
    - Linear time O(n): Running time is proportional to size.
        - Ex) Compute maximum of n numbers
    - Linearithmic time O(n log n)
        - Ex) Divide and conquer algorithms like mergesort.
    - Quadratic time O(n^2)

        - Ex) Anything with 2 nested for loops.
    - Cubic time O(n^3)
        - Ex) Anything with 3 nested for loops.
    - Polynomial time O(n^k)
    - Exponential time O(2^n)

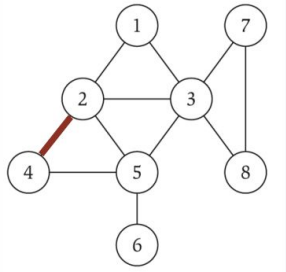
Chapter 3 - Graphs

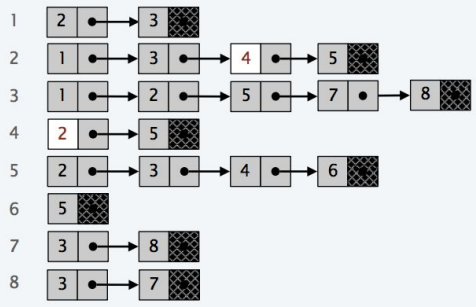
- Graphs are made up of nodes and edges
    - Undirected graphs have edges with no direction.
    - Directed graphs have edges with direction.
- An undirected graph is a tree if it is connected and does not contain a cycle and has n-1 edges.
- Graphs can be represented with adjacency matrices which are n by n binary matrices that specify a 1 if there is an edge between two vertices and a 0 for no edge.
    - Identifying all the edges takes O(n^2)



	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	0	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

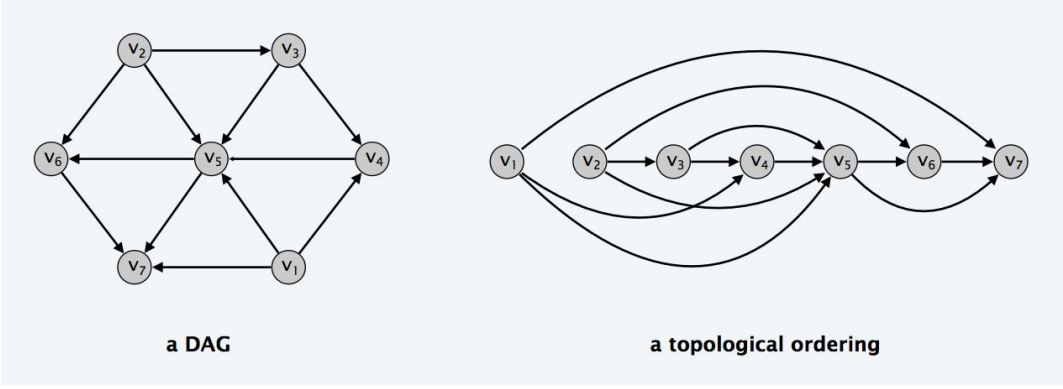
- Adjacency lists are better with sparse graphs. Adjacency lists are basically node indexed array of lists.
    - Identifying all the edges takes O(m+n) time





- A path is a sequence of nodes and a path is simple if all nodes are distinct.
- An undirected graph is connected if for every pair of nodes u and v, there is a path between u and v. The graph is complete when there is always an edge between u and v.
- A cycle is a path that ends up at the beginning vertex.
- A cut is a partition of the nodes into two subsets S and V - S.

    - The cutset of a cut S is the set of edges with exactly one endpoint in S.
- A cycle and a cutset intersect in an even number of edges.
- A minimum spanning tree is one that is a subgraph of G, has n-1 edges, is connected, and there is a unique simple path between every pair of nodes, and the sum of edge costs are minimized.
- An undirected graph is a tree if it is connected and does not contain a cycle. Any two of the following imply the third.
    - G is connected.
    - G does not contain a cycle.
    - G has n-1 edges.
- Two problems with graphs.
    - Connectivity problems: Given two nodes, is there a path between the two?
    - Shortest path problem: Given two nodes, what is the length of the shortest path?
- Directed graph is strongly connected if for every two nodes in the graph, there is a path from u to v and from v to u.
    - Those two nodes are said to be mutually reachable.
- Breadth-first search: Explore outward from the starting vertex in all possible directions, one “layer” at a time.
- A graph is bipartite is the graph can be separated into two vertex sets, and all the edges in the graph go from a vertex in one set to a vertex in the other set.
    - A graph is bipartite if and only if it does not contain an odd length cycle.
- In a directed graph, two nodes are mutually reachable if there is both a path from u to v and a path from v to u.
- A graph is strongly connected if every pair of nodes is mutually reachable.
- A directed acyclic graph is a directed graph that contains no directed cycles.
- Topological ordering of a directed graph is an ordering so that the edges only go from left to right.



- If G is a DAG, then G has a node with no incoming edges.
    - Because if there isn’t one like that, then there would be at least one cycle.
- If G is a DAG, then G has a topological ordering.
- DAGs have no directed cycles.
- A topological ordering of G is an order of the nodes such that for every edge (v_i and v_j), then i < j. All edges point forward in the ordering.

    - Basically for every directed edge uv, u becomes before v in the ordering.
- The strongly connected components of a DAG have vertices that are reachable by every other vertex in that component.

Chapter 4 - Greedy Algorithms

- For the interval scheduling problem, you can consider jobs in some natural order.
    - Earliest Start Time: Consider jobs in ascending order of start time
    - Earliest Finish Time: Consider jobs in ascending order of finish time
    - Shortest Interval: Consider jobs in ascending order of finish - start time
- The earliest finish time first algorithm is optimal.
- There is also the problem of minimizing lateness.
    - Shortest processing time first: Consider jobs in ascending order of processing time.
    - Earliest deadline first: Schedule jobs in ascending order of deadline.
    - Smallest slack: Schedule jobs in ascending order deadline - processing time.
- The earliest deadline first schedule is optimal.
- To analyze greedy algorithms, use one of these approaches.
    - Greedy algo stays ahead: Show that after each step of the greedy algorithm, its solution is at least as good as any other algorithm.
        - Define your solution. Your algorithm will produce some object X and you will compare it against some optimal solution X*.
        - Define your measure: Goal is to find a series of measurements you can make of your solution and the optimal solution. Basically some function that takes in the greedy solution and the optimal solution and tells you some measurement.
        - Prove Greedy Stays Ahead: Prove that f(X) >= f(X*) or vice versa for all values. The argument is normally done through induction.
        - Prove optimality: Done by contradiction by assuming the greedy solution isn't optimal and then using the fact that greedy stays ahead to derive a contradiction.
    - Structural: Discover a simple structural bound asserting that every possible solution must have a certain value.
    - Exchange argument: Gradually transform any solution to the one found by the greedy algorithm without hurting its quality.
- Shortest path problem given a directed graph, a source, a destination, and with each edge having a length or weight.
- Dijkstra's algorithm is a way to solve the shortest path problem. It is a greedy approach to the problem.
- Prim's, Kruskal's, and Boruvka's algorithms are all ways of finding a minimal spanning tree.
    - Prim's algorithm: Initialize a tree with a single vertex. Grow the tree by one edge: Of the edges that connect the tree to vertices not yet in the tree, find the

minimum weight edge and transfer it to the tree. Repeat this until all vertices are added to the graph.
    - Kruskal's algorithm: Consider edges in ascending order of weight, and add edge to the tree unless it would create a cycle.
    - Reverse-delete algorithm: Consider edges in descending order of weight, and remove edge unless it would disconnect the graph.
    - Boruvka's algorithm: Idk couldn't understand it.
- Greedy algorithms are often used to solve optimization problems, where you want to minimize or maximize a quantity, based on a certain set of constraints.
    - Normally greedy algorithms look at what the most obvious local solution is.
    - They build up the solution in small steps, each time making a choice to optimize some underlying criterion.
- When trying to write a proof that shows a greedy algorithm is correct,
    - Need to show that your algorithm produces a **feasible** solution, a solution that obeys the constraints.
    - Need to show that your algorithm produces an **optimal** solution, a solution that maximizes or minimizes the appropriate quantity.
- When you can actually prove a greedy algorithm works, you show that one local decision rule can be used to construct optimal solutions.

Chapter 5 - Divide and Conquer

- Refers to a class of algorithmic techniques in which one breaks the input into several parts, solves the problem in each part recursively, and then combines the solutions to these subproblems into an overall solution.
- Analyzing the running time of divide and conquer algorithms involves solving a recurrence relation.
- Mergesort is the classic divide and conquer algorithm.

Chapter 6 - Dynamic Programming

- Dynamic programming works by implicitly exploring the space of all possible solutions, by decomposing things into subproblems, and then building up solutions to larger and larger subproblems.
    - It comes very close to brute force search, but the catch is that although we work through a large set of possible solutions, we don't examine all of them explicitly.
- In general, dynamic programs work by the following.
    - Reduce problem to a series of decisions
    - Design a recursive procedure to make decisions in an ordered way.
    - Each decision should leave us with a subproblem to solve that looks like the original problem.
    - Show that instead of recursion, we can iteratively fill out a table with solutions.
- Basically, we are building up tables of partial answers

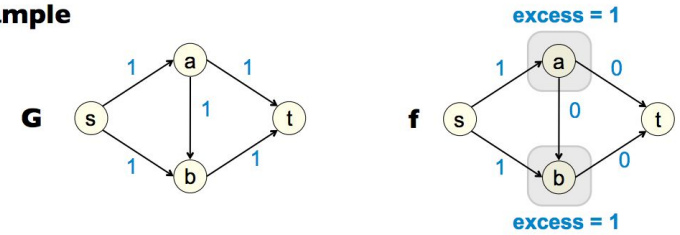
- The technique of saving up values that have already been computed is called memoization.
- The main idea with dynamic programming is coming up with recurrences.
- To develop an algorithm based on dynamic programming, one needs a collection of subproblems derived from the original problem that satisfies the following.
    - There are only a polynomial number of subproblems.
    - The solution to the original problem can be easily computed from the solutions to the subproblems.
    - There is a natural ordering on subproblems from the smallest to the largest, together with an easy to compute recurrence that allows one to determine the solution to a subproblem based on the solutions to some number of smaller subproblems.

Chapter 7 - Network Flow

- The max flow problem is defined as a problem where you are given a flow network and the goal is to arrange the traffic so as to make as efficient use as possible of the available capacity. Basically the goal is to find a flow of maximum value.
- Network models have capacities on the edges, source nodes which generate traffic and sink nodes which absorb traffic.
- Nodes other than the source and the sink are called internal nodes.
- Each edge in the graph will be represented by f(e) which represents the amount of flow carried by edge e. The flow must be below the capacity c(e) of the edge and the amount of flow carried into node v (all nodes beside source and sink) must be equal to the flow coming out.
- The minimum capacity of any such division of the network is called a minimum cut.
- When trying to find the max flow graph, we can push forward on edges with leftover capacity, and push backward on edges that are already carrying flow.
- Given a flow network G and a flow f on G, we define the residual graph of G as
    - Having the same node set.
    - For each edge of G on which f(e) < c(e), there are c(e) - f(e) leftover units of capacity on which we could try pushing flow forward. We include that edge in the new residual graph with a capacity of c(e) - f (e). These edges are called forward edges.
    - For each edge of G on which f(e) > 0, there are f(e) units of flow that we can undo if we wanted to, by pushing the flow backward. We include that edge in the residual graph with a capacity of f(e) but the direction is reversed.
    - Every edge in the residual graph has a residual capacity which is equal to the original capacity of the edge minus the current flow.
- There will be two edges between nodes in a residual graph if the amount of flow in the original graph is nonzero but also not equal to the capacity.
- The Ford Fulkerson algorithm is a way to find the max flow from s to t.
    - Start with initial flow as 0.

    - While there is an augmenting path from source to sink, add the path-flow to flow.
    - Return flow.
- The residual graph, in some ways, basically indicates that there is additional possible flow.
    - If there is a path from source to sink in a residual graph, then it is possible to add flow.
- An augmenting path is one with non-full forward edges or non-empty backward edges.
- The way to find augmenting paths for the Ford Fulkerson algo is to do a BFS or DFS on the residual graph, and find out if there is a path from the source to the sink. We find the possible flow through this path by finding the minimum residual capacity along the path, and then add the found path flow to the overall flow.
- We then need to update the residual capacities in the residual graph.
- Another way to think about the Ford Fulkerson algo is:
    - Find an augmenting path
    - Compute the bottleneck capacity (the edge in the path with the smallest capacity left)
    - Augment each edge with that bottleneck capacity and augment the total flow.
    - Repeat until we can't find an augmenting path
- https://www.youtube.com/watch?v=Tl90tNtKvxs Really good tutorial
- An s-t cut is a partition (A,B) of the vertex set V, so that s is in A and t is in B. The capacity of the cut is the sum of the capacities of all the edges out of A.
- If f is any s-t flow, and (A,B) is any s-t cut, then v(f) <= c(A,B)
- The Preflow Push algorithm basically tries to increase the flow on an edge by edge basis.
- Each node basically has a label, which is a potential and the route flow will be from high to low potential.
- A function is called a preflow if it follows the capacity constraints, but instead of the conservation property of flow into each node = flow out of each node, it computes an excess value for each node where the flow coming in - flow going out. That excess value has to be at least 0.
    - The point of preflow is to push as much flow as possible, and then start to push things back to maintain the feasibility. There will almost always be leaks in Preflow push at the beginning of the algorithm.
    - It allows for more incoming flow into a vertex than the outgoing flow.

Example



- Labeling assigns a non-negative integer label h(v) to all of the nodes. This is the notion of a height associated with a node. We think of flow as moving downhill.
    - The height of the source will be n and the height of the target is always 0. These values will never change.
    - All of the other nodes' heights will be 0.
- For any edge (v,w) in the residual graph, we have h(v) <= h(w) + 1
- Push applies if excess(v) > 0, h(w) < h(v) and the edge is in the network. You then add min(excess(v), c(v,w)) to the flow between the two nodes
- Relabel if excess(v) > 0, and for all nodes w connected to v in the residual graph, if h(w) >= h(v), then increase h(v) by 1.
    - It will always increase the height by 1 if the neighboring nodes are equal or higher than the current node v.
- One of the differences between Ford Fulkerson and Preflow Push is that FF will maintain a feasible flow at all times and gradually improve that flow. Preflow Push will also get that maximal flow, but we don't necessarily have a feasible flow until the algo terminates.
- A perfect matching in a graph is where every node in the graph is part of exactly one edge.
    - Non-perfect is where some nodes aren't part of any edge.
    - In order to have a perfect matching, let S be a subset of nodes, and let N(S) be the set of nodes adjacent to nodes in S. And for any subset S picked on the left, if the cardinality of N(S) is >= cardinality of S, then it is a perfect matching.
        - The problem is that it's pretty expensive to test this for all of the subsets.
- The maximum cardinality matching in G is always less than or equal to the max flow in G'.

Chapter 8 - NP and Computational Intractability

- The Independent Set problem is where you are given a graph G, a set of nodes S is independent if no two nodes in S are joined by an edge.
    - No efficient algorithm for finding an independent set from a graph G.
- Another related problem is that of finding a Vertex Cover. This is where you a given a graph G and a set of nodes S is a vertex cover if every edge in the whole graph has at least one node that is part of S.
- Y <=(p) X means that if X can be solved in polynomial time, then Y can be solved in polynomial time.
- We can prove that both are reducible to each other.
    - Independent Set <=(p) Vertex Cover
    - Vertex Cover <=(p) Independent Set
- Set Cover is a problem where given a set U of elements, a collection of subsets of U, a number k, does there exist a collection of at most k of these sets whose union is equal to all of U?
    - We can also say that Vertex Cover <=(p) Set Cover.
- A clause is a disjunction of distinct terms.

- An assignment satisfies clause C if C evaluates to 1 under Boolean logic.
- The satisfiability problem is known as SAT.
    - Given a set of clauses over a set of variables, does there exist a satisfying truth statement.
- 3-SAT is where you have a set of clauses of length 3 and you have to find a satisfying truth statement.
- We can prove that 3-SAT <=(p) Independent Set
- Reductions are transitive
- P is the set of problems for which there is a polynomial time solution.
- NP is the set of all problems for which there exists a polynomial time checker.
    - P is in NP
- The question of if P = NP basically is asking whether all problems that can be checked in polynomial time can also be solved in polynomial time.
- NP complete is when the problem is in NP and for all Y in NP, Y <=(p) X.
    - Basically the problem needs to be in NP
    - And it needs to every problem in NP is solvable if you had polynomial time solution for X.
- You can also say that for a problem X to be considered NP complete, it has to be in NP and every problem in NP has to be reduced to X.
- Carrot ^ is AND, down arrow is OR, and sideways L is NOT.

Ways to Go About Reductions

- List of NP complete problems (Useful for when you're trying to prove that a certain problem is NP complete)
    - 3 - SAT - Given a set of clauses C_1 to C_k (with each clause having 3 variables in it), over a set of variables X_1 to X_n, does there exist a satisfying truth assignment?
    - Vertex Cover - Given a graph G and a number k, does G contain a vertex cover (set of vertices where every edge in the graph has at least one endpoint as one of the vertices in the set) of size at most k?
    - Independent Set - Given a graph G and a number k, does G contain a independent set (set of vertices where there aren't any edges between any two vertices in the set) of size at most k?
    - Hamiltonian Path - Given a graph G, does there exist a simple cycle that contains every node in V?
    - Clique Given a graph G and a number k, does G contain a clique (set where all vertices have edges to one another) of size at most k?
    - Set Cover

- 3 - SAT
    - You can think of this problem as having to make 0/1 decisions about each of the k variables so that at least one of the 3 ways to satisfy each clause is met.

    - You can think of it as having to choose one term from each clause, and then find a truth assignment that causes all the terms to be 1, and therefore you've satisfied all the clauses.
- Vertex Cover

Ways to Go About Proving Greedy Algorithms

- Greedy Stays Ahead
    - Create two sets of solutions, one for the optimal and one for the greedy. Create some measure that quantifies how the greedy and the optimal are doing after x steps of the algorithm. Prove that the results of the two sets are always going to be the same using induction.
        - First show the base case. Show that it works after 1 or 0 steps. Then, assume it works for K-1, and prove that it works for K.
    - Then, prove that because the greedy solution's measures are as good as those of the optimal, then the solution must be optimal.
        - This is done though proof by contradiction. Saying that the optimal is better than greedy and then proving otherwise based on the facts you know through the measures.
- Exchange argument (By contradiction)
    - Assume the optimal solution is different (not consistent with the rule).
    - Find the first difference of the optimal solution and the greedy solution.
    - Replace the difference with the greedy choice.
    - Argue that the new solution is better and then, you will have shown a contradiction.

Midterm Solutions

- For #2, proof by exchange argument. Suppose there is a optimal solution which the restaurant are the same from r1 to rn with our greedy algorithm. Let r(i+1) for our algorithm is at x' + 10, so there exists some r(k) for optimal solution placed between [x-10, x+10] to cover x. Now we can simply move rk to x+10 without affecting the feasibility.