CS180: FINAL STUDY GUIDE
CONNOR JENNISON

Algorithms

- Stable Matching
- Interval Scheduling Problem
- Kahn's Algorithm for Topological Ordering
- BFS/DFS
- Greedy Interval Scheduling (Multiple Processors)
- Dijkstra's Algorithm
- Kruskal's Algorithm
- Prim's Algorithm
- K-Clustering Algorithm
- Mergesort Algorithm
- Counting Inversions
- Closest Pair of Points
- Weighted Interval Scheduling: Recursive Procedure
- Segmented Least Squares
- Knapsack Problem
- Sequence Alignment
- Shortest Path in Graph (negative weights too)
- Ford-Fulkerson
- Bipartite Matching
- Disjoint Paths

Chapter 1: Introduction

Stable Matching

While $\exists$ an unmatched man...

- Pick a man that hasn't been matched yet (arbitrary), called m
- Take the highest woman left in his priority list, called w and attempt to temporarily match them
    - If the woman is unmatched, she automatically accepts
    - If the woman is matched with another man m', compare in her priority list
        - If m is higher than m' on her list, she leaves m' who is now unmatched again, and

        matches with m.
    - Else, m' is higher than m on her list, she keeps m' and m remains unmatched
  - In either case, remove w from the priority list of m, since she has been considered either way

Interval Scheduling

- Sort the intervals by ending time from earliest to latest
- While we have intervals to look at still
  - Add the earliest ending interval to the schedule
  - Eliminate all overlapping intervals

Chapter 2: Algorithm Analysis

Computational Tractability

- An algorithm is efficient if it has a polynomial runtime

Asymptotic Order

- **Order**: a function $T(n) = O(f(n))$ if $\exists$ constants n_0, c such that $\forall\ n \geq n_0$, we have $T(n) \leq cf(n)$
  - Can think of order as an **upper bound**.
- **Omega Notation:** a funtion $T(n) = \Omega(f(n))$ if $\exists$ constants n_0, c such that $\forall n \geq n_0$, we have $T(n) \geq cf(n)$
  - Can think of Omega notation as a **lower bound**
- If a problem/algorithm is $O(n_1)$ and $\Omega(n_2)$ such that n_1 = n_2, then we say the problem is $\underline{\Theta(n)}$

Common Running Times

- $O(log(n))$,
- Linear: $O(n)$
- $O(nlogn)$
- Quadratic: $O(n^2)$
- Non-Polynomial

Chapter 3: Graphs

Basic Definitions/Applications

- **Graph** - a graph is defined by a set of verticies (nodes) and edges (links). we write this formally by saying $G = (V, E)$ with, for example, $V = \{a, b, c, d\}$, $E = \{(b, c), (a, b), (a, c)(a, d)\}$
  - graphs can be **directed/undirected**
  - graphs can be **weighted/unweighted**
  - graphs can be **connected/disconnected**
- If a graph has 2 or less edges with an odd number of edges attatched to it, it is called an **Eulerian Graph**

Connectivity/Traversal

- Breadth First Search
  - Investigate all points nearest our starting point before moving to the next point
  - Queue-based
  - BFS Tree: A node at level i in the tree has a shortest path to the starting point of i
  - $O(e + v)$
- Depth First Search
  - Investigate all points down a path when searching
  - Stack-based
  - $O(e + v)$

Graph Representation/Implementing Graphs

- Adjacency matrix
  - $[i, j]$ is 0 if there is no edge between i and j, 1 otherwise
  - if graph is undirected, then this matrix is symmetric
- Adjacency List
  - array of linked lists, each element of array represents node, linked list of all nodes it is connected to
  - better for directed graphs
- BFS
  - Use the queue ("first in first out")
- DFS
  - Use the stack ("last in last out")

Testing Bipartiteness

- Algorithm
  - Run BFS and denote the ith layer of the tree as L_i
    - At an even layer, color all nodes red
    - At an odd layer, color all nodes blue
  - After BFS, check all edges to ensure each node has a different color.

Connectivity in Directed Graphs

- **strongly connected component** - two points a and b are strongly connected if $\exists$ a path from a to b as well as a path from b to a
  - The set of strongly connected components is a disjoint set
  - Any strongly connected component must be a cycle

Directed Acyclic Graphs/Topological Sort

- Terms
  - **directed acyclic graph** - a directed graph with no cycles
  - **in-degrees** - the number of directed edges coming in to that vertex, $deg^{-}(v)$
  - **out-degrees** - the number of directed edges leaving a vertex. Denoted $deg^{+}(v)$
  - **source** - a vertex v such that $deg^{-}(v) = 0$
- **Kahn's Algorithm** (only on a DAG): $O(e + n)$
  - Choose a source
  - Output the source
  - Update in-degrees of all of the vertecies connected to the source
    - If any of these become a source, add them to the source list
  - Move back to step 1 and perform this recursively.
- Topological Sort Properties
  - Not necessarily unique, can be more than one solution

Chapter 4: Greedy Algorithms

- **Greedy Paradigm** - Take a problem. Look at one or two items in the problem and make a decision very quickly without really haven't seen the entire problem. Once make decision you stick to it.

Greedy Interval Scheduling

- Problem from before greedily chooses interval to select. Greedy algorithm stays ahead of all other algorithms (for proof)
- Multiple processors: put the next ending task on the lowest available processor.

Shortest Paths in a Graph

- **Dijstra's Shortest Path Algorithm** (http://www.geeksforgeeks.org/greedy-algorithms-set-6-dijkstras-shortest-path-algorithm/)
  - Runtime using adjacency matrix: $O(n^2)$, good for dense graphs
  - Runtime using heap: $O(elogn)$, good for sparse graphs

Minimum Spanning Tree Problem

- **spanning tree (ST):** a graph with the following properties
  - it is a tree
  - it has the minimum number of edges ($n-1$ assuming graph has n verticies)
  - it spans/touches every vertex
- **minimum spanning tree (MST):** a spanning tree such that it has minimum weight
- **Minimum Spanning Tree Therom**
  - Take a graph, and split it into two partitions that has the following properties
    - The two partitions are disjoint
    - Each of the two partitions is non-empty
    - All points in the graph are in either of two partitions
  - Consider all edges between partition 1 and partition 2, and denote the minimum weighted edge e_{min}.
  - Then ∃ a Minimum Spanning Tree that includes e_{min}.
- **Prim's MST Algorithm**

```
• Given a graph G, n verticies, e edges. Assume MST exists
• Create two partitions
    - L: contains an arbitary vertex
    - R: contains all other verticies
• While there are still verticies in R
    - Find the minimum edge between L and R, denoted as e[min]
    - Move e[min] into list for MST
    - Take vertex connected to e in partition R, and move it to partition L

Same runtime as Dijkstra's
```

- **Kruskal's MST Algorithm**

```
Given a graph G, n verticies, e edges, Assume MST exists
• Sort all the edges by weight in non-decreasing order
• While there are less than (n-1) edges in the MST
    - Consider the minimum edge that has not been considered
    - If including this edge in the MST creates a cycle
        * Discard and move on
    - Else including this edge in MST doesn't create cycle
        * Add to MST and move to next edge

O(elogv) assuming given sorted edges)
```

- Union-Find/Kruskal
    - Every vertex is in it's own set at the start. We have two operations
        - **union**: take two verticies, and take the union of them
        - **find**: take two verticies, and see if they are in the same group
    - Makes checking for cycles, adding them to groups more efficient
    - Implemented as a Balanced BST
- **K-Clustering**
    - Start with each vertex in its own roup
    - Perform Kruskal until we have the desired k clusters.

Chapter 5: Divide and Conquer

Mergesort Algorithm

- Merging: keep a pointer and compare values at pointer until merged
    - Arays must be sorted

    - Time complexity: $O(n+m)$
- Recurrance relation for mergesort

$$T(n) = 2T(\frac{n}{2}) + cn$$
$$\vdots$$
$$T(n) = 2^i T(\frac{n}{2^i}) + icn$$

    - Solving $\frac{n}{2^i} = 1$ gives that $i = log(n)$, so we get $O(nlogn)$

Further Recurrance Relations

Assume we have the recurrance $T(n) \leq qT(\frac{n}{2}) + c$

- If q > 2, this problem is bounded by $O(n^{log_2(q)})$
- If q = 1, this problem is bounded by $O(n)$

Counting Inversions

- Do the mergesort algorithm, splitting into left half A and right half B with caveat.
    - If when we merge, the element in B is first, increment $count$ by the number of elements yet to be considered in A.
- By doing this, we count inversions by counting every time numbers switch.

Finding Closest Pair of Points

- We do this by using mergesort in the following way
    - Recursively find the closest pair among the "left half" of points and the "right half" of points, then use informatino to get rest of solution.

Chapter 6: Dynamic Programming

Weighted Interval Scheduling: Recursive Procedure

We start with a couple of assumptions

- Up to a certain coordinate x_i, we must know the optimal solution from the beginning to x_i
- For any $x_j < x_i$, we must know the optimal solution from the beginning to x_j

If there is an interval I_j such that it overlaps and goes past x_j then we compute two assumptions

- $I_j \notin S$ (solution)
  - $S_j = S_{j-1}$
- $I_j \in S$
  - $S_k = S_{l_j} + w_j$
  - S_{l_j} refers to the solution at x_i, the first ex coordinate before the beginning of j

We store all of the tentative solutions that we build up, and once we get to the end, we backtrack to find the optimal solution.

Principles of Dynamic Programming

- We split the problem into multiple subsets
- The subsets are not necessarily disjoint
- For each subproblem, we get an optimal solution, then combine these to get a solution to the problem as a whole.
  - Continue to do this recursively until each subproblem is small enough
- ALSO: Good way to think is that problems have a certain number of parameters we can optimize over.

Segmented Least Squares: Multi-way Choices

- We have that $OPT(j) = \underset{1 \leq 1 \leq j}{min}(e_{i,j} + C + OPT(i-1))$
- Basically, we start at bottom and for a given solution, we try to draw a new line from a point i to j and then see which has the smallest error
- $O(n^3)$

Subset Sums/Knapsacks: Adding a Variable

We are given a knapsack with size S. We have many items of different values, and sizes: $I_k = (s_k, v_k)$. There is an unlimited amount of each of the given items. Our goal is to place items in the knapsack such that we have the maximum size.

What parameters do we have that we can use dynamic programming on

- Number of items
- Size of knapsack

We have a table below that shows the Size of the knapsack against which items we are considering. We want to use dynamic programming to fill out each cell.

	S = 1	S = 2	S = 3	...	S = n
Only Item 1					
Add Item 2					
⋮					

In our algorithm, we will fill

- For item j, for knapsack size i
  - Item $j \in S$
    - Denote v_x as the value at item j, index $i - S_j$
    - $S_{i,j} = v_x + v_j$
  - Item $j \notin S$
    - $S_{i,j} = S_{i,j-1}$
  - Take $S_{i,j}$ as the maximum of these two values.
- Eventually, we will have the value for all items and knapsack size S
- The comparisons in each value of the array take constant time, so for this algorithm we have $O(nC)$
  - This is not polynomial because we have two different variables that the order depends on, but we could have, for example, $C = 2^n$
  - We call functions of this type **pseudo-poynomial**.

Shortest Path in a Graph

- **Bellman-Ford Algorithm**: want to find minimum $s - t$ path.
- Denote the shortest $v - t$ path of at most length i as OPT(i,v)
  - $OPT(i, v) = \min(OPT(i-1, v), \min_{w \in V}(OPT(i-1, w) + c_{vw}))$
- Build a matrix of the verticies, return $OPT(n-1, s)$.
- Runs in $O(ev)$ time.

Subsequence Problem

We are given a sequence of unique integers. We define the following

- A subsequence is increasing if for all i, j in the subsequence such that $i < j$, $S_i < S_j$
- A subsequence is contiguous if all elements in the subsequence are adjacent and in the same order as in the regular sequence

Find the largest increasing subsequence (not necesarrily contiguous)

Assume that for each of the first i elements of the sequence, we know the optimal solution. We now want to find the optimal solution for the first $i+1$ points. We have two cases to consider

- x_{i+1} is not in the solutions
  - the optimal solution up to x_i is the same as the optimal solution up to x_{i+1}
- x_{i+1} may be in the solution
  - for each of the the optimal soutions at x_k in $x_1, \ldots, x_i$
    - if $x_{i+1} > x_k$
      - add 1 to the solution corresponding to x_k
  - denote the optimal solution in this case as the

Chapter 7: Network Flow

Max Flow/Ford-Fulkerson

- Conditions for Network Graph
  - For each $e \in E$, we have $0 \leq f(e) \leq c_e$
  - For each node v other than s and t, we have $\sum_{\text{e into v}} f(e) = \sum_{\text{e out of v}} f(e)$
- Residual Graph
  - Node set of residual graph G_f is the same as that of G
  - For each edge $e = (u, v)$ of G on which $f(e) < c_e$, we include a forward edge of capacity $c_e - f(e)$ to represent leftover units we can sitll push
  - For each edge $e = (u, v)$ of G on which $f(e) > 0$, we include a backward edge of capacity $f(e)$ to represent how many units of flow we have pushed.
- **Ford-Fulkerson Algorithm**

```
Initially, f(e) = 0 for all e in G
While there is a simple s-t path in G_f
    Denote this path as P
    Find the bottleneck of that path (minimum residual capacity)
    Send that much flow through every edge on the path
Endwhile
Return
```

Max Flows/Min Cuts in a Network

- Cut definition
  - Split network to two sets A, B s.t. $s \in A$ and $t \in B$
  - An **s-t cut** is defined as $c(A, B) = \sum_{\text{e out of A}} c_e$

- The size of the max flow is also the size of the min cut, therefore, we can use **Ford-Fulkerson** to find the size of the min cut.
- **MIN CUT FINDING ALGORITHM**
  - Perform Ford-Fulkerson on a network
  - Denote A^* as all the nodes reachable by s in the residual graph
  - Denote B^* as all other nodes
  - Return (A^*, B^*)

Bipartite Matching Problem

- Given two bipartite groups, A and B
- Direct all edges from A to B
- Create a source, s, and create an edge (s, a_i) for each $a_i \in A$
- Create a sink, t, and create an edge (b_i, t) for each $b_i \in B$
- Every edge has capacity 1
- Find max-flow using Ford-Fulkerson, max-flow corresponds to maximum matches

Disjoint Paths in Directed/Undirected Graphs

- Problem: Find maximum number of edge disjoint paths in a graph
- Two paths are **edge disjoint** if none of them share edges
- Algorithm
  - Denote two nodes as source/sink
  - Make the capacity of all edges in the flow network 1
  - Run Ford-Fulkrson

Circulations

- Conditions: Graph with lower bounds
  - (Capacity Condition) For each $e \in E$, we have $l_e \leq f(e) \leq c_e$ (lower bount)
  - (Demand Condition) For each $v \in V$, we have $f^{in}(v) - f^{out}(v) = d_v$
  - For a circulation, we have $\sum_{v \in V} d_v = 0$ or $\sum_{d_v < 0} -d_v = \sum_{d_v > 0} d_v$
- To get rid of lower bounds:
  - Subtract the capacity by the lower bound
  - Add the lower bound to the beginning node
  - Subtract the lower bound to the ending node

Survey Modeling

- Model the graph like so
  - edges of capacity c_i, c_i' from source to customer representing how many products they are asked about
  - edges of capacity 0,1 from customer to product to see if they've ever used it
  - edges of capacity p_i, p_i' from product to sink representing how many people answered about this product

Chapter 8: NP and Computational Intractability

- $y \leq_p x$ implies
  - x is at least as hard as y
  - if y is NP-HARD, so is x
  - if x can be solved in polynomial time, so can y
- How to use this definition
  - Start from an NP-Hard problem y and show x is NP-Hard.
  - Show that an arbitrary instance of y can be solved by a "black-box" that solves x.
- Thm: S is a independent set iff $V - S$ is a vertex cover
  - Proof: —> for any edge, at least one of the nodes must be in $V - S$
  - Proof: <— if two nodes in S shared edge, violates verte cover, so S is independent set
- Definition
  - A **vertex cover** is defined as a minimum set of verticies VS in a graph G such that every edge e has at least one end in VC. PROV
- NP-Completeness of Independent Set/Vertex Cover/Set Cover
  - $IS \leq_p VC \leq_p IS$
  - $VC \leq_p SC$
  - IS = Independent Set, VC = Vertex Cover, SC = Set Cover