Theory of Algorithms Class Notes

Calvin Walker

Lecture 1: Gale-Shapley Algorithm for Stable Matching

Stable Matching Problem: There are two groups of people: $A = \{a_1 \dots a_n\}$ and $B = \{b_1 \dots b_n\}$, and we want to find a stable matching between A and B such that there is no pair of people a_i, b_j who would rather be with each other than their current partners. i.e We want to find a permutation π such that there are no $i, j \in [n]$ for whom

1. $\pi(i) \neq j$
2. a_i prefers b_j to $b_{\pi(i)}$
3. b_j prefers a_i to $b_{\pi^{-1}(j)}$

Gale-Shapley algorithm: The people in group A make offers in the order of their preference lists. When b_j receives an offer from a_i, if b_j is unmatched or preferes a_i to their current partner, they accept and become partners with a_i. Otherwise, b_j rejects and a_i moves onto the next person.

Properties:

- The Gale-Shapley algorithm terminates in $O(n^2)$ steps and results in a stable matching.

  Proof: There are n^2 possible offers and each offer is made at most once.

  Terminates: Oberve that if some a_i makes an offer to their last choice b_j, then all people except b_j must be unavailable for a_i and thus already matched. There are only $n-1$ other people in A, so b_j must be unmatched, and b_j accepts a_i's offer, and the algorithm terminates.

  Stable: Assume there are $i, j \in [n]$ such that a_i and b_j prefer each other to their current partners. Let b_j be a_i''s current partner and b_j' be matched with a_i. Then b_j is available to a_i. However, since a_i is matched with b_j', b_j must have been unavailable to a_i at some point, a contradiction.

- As Gale-Shapley progresses, (1) people in B only become happier, and (2) if b_j becomes unavailable to a_i they never become available again.

  Proof: (1) b_j only breaks off from its match if it prefers the new a_i. (2) Assume b_j becomes unavailable to a_i and then becomes available again. Let a_i' be b_j's partner when b_j became unavailable to a_i. Let a_i'' be b_j's partner when b_j became available to a_i again. Then, b_j prefers a_i' to a_i and a_i to a_i''. However, b_j must prefer a_i'' to a_i', a contradiction.

Lecture 2: Big O Notation

Big O Notation: Given functions $f, g : \mathbb{N} \mapsto \mathbb{R}^+$

1. We say that $f(n)$ is $O(g(n))$ if there exist $n_0, C > 0$ such that for all $n \geq n_0$, $f(n) \leq Cg(n)$
2. We say that $f(n)$ is $\Omega(g(n))$ if there exist $n_0, c > 0$ such that for all $n \geq n_0$, $f(n) \geq cg(n)$
3. We say that $f(n)$ is $\Theta(g(n))$ if $f(n)$ is $O(g(n))$ and $\Omega(g(n))$ such that there exist n_0, c, C' such that for all $n \geq n_0$, $cg(n) \leq f(n) \leq C'g(n)$

Examples:

- If $f(n) \leq 8n\log_2(n) + 20n + 100$ then $f(n)$ is $O(n \log n)$
- If $f(n) \geq n^2 - 3n - 2$ then $f(n)$ is $\Omega(n^2)$
- If $\frac{1}{2}\log_2 n - 2 \leq f(n) \leq 4\log_2 n + 1$ then $f(n)$ is $\Theta(\log n)$

Properties:

- Given $f, g, h : \mathbb{N} \mapsto \mathbb{R}^+$, if $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$, then $f(n)$ is $O(h(n))$
- Given $f_1, f_2, g_1, g_2 : \mathbb{N} \mapsto \mathbb{R}^+$, if $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$ then:
  - $f_1(n) + f_2(n)$ is $O(g_1(n) + g_2(n))$ and $O(\max\{g_1(n), g_2(n)\})$
  - $f_1(n)f_2(n)$ is $O(g_1(n)g_2(n))$

    Proof: There exist n_1, C_1 such that $\forall n \geq n_1(f_1(n) \leq C_1 g_1(n))$ and there exist n_2, C_2 such that $\forall n \geq n_2(f_2(n) \leq C_2 g_2(n))$. Let $n' = \max\{n_1, n_2\}$ and $C' = C_1C_2$, So for all $n \geq n'$, $f_1(n)f_2(n) \leq C'g_1(n)g_2(n)$

  Comparing logarithms, polynomials, and exponential functions:

1. $\log n^{C'}$ is $O(n^C)$
2. $n^{O(1)}$ means at most n^C for some $C > 0$. This is polynomial time
3. $2^{O(n)}$ means at most 2^{Cn} for some $C > 0$. This is exponential time.

Lecture 3: Greedy Algorithms

Interval Scheduling Problem 1 (Maximizing number of jobs): Given n jobs with a set time interval $[a_i, b_i]$ and one processor, find a schedule S which accepts the maximum number of jobs without having two jobs running at the same time. Formally, we say that a sequence $S = (i_1, \ldots, i_m)$ is a valid schedule if for all $j \in [m-1](b_{i_j} \leq a_{i_{j+1}})$

Greedy Algorithm for Interval Scheduling 1: When choosing the next job, always choose the job which can be finished first.

- *Stored Data*: $S_k = (i_1, \ldots, i_k)$ of the jobs which have been accepted thus far, and the time $t_k = b_{i_k}$
- *Initialization*: $S_0 = \emptyset$ and $t_0 = 0$
- *Iterative Step*: Choose the next job i_{k+1} from the set $\{i \in [n] \mid a_i \geq t_k = b_{i_k}\}$ such that b_{i_k} is minimized. Update S_{k+1} and $t_{k+1} = b_{i_{k+1}}$, if there are no available jobs left then the algorithim terminates and we take $S = S_k$

Properties:

- Given a valid schedule $S = (i_1, \ldots, i_m)$, for all $j \in [m]$, we denote the time at which the jth hob in S finishes as $t_j(S) = b_{i_j}$. For all $j > m$, we say $t_j(S) = \infty$
- If the greedy algorithim gives a valid schedule $S = (i_1, \ldots, i_m)$, then for any other valid schedule $S' = (i_1, \ldots, i'_m)$, $m' \leq m$
  Proof (Using the lemma proved below): Consider $j = m+1, t_{m+1}(S) = \infty$ so we must have that $t_{m+1}(S') = \infty$. Thus, $m' \leq m$
- If $S = (i_1, \ldots, i_m)$ is the schedule given by the greedy algorithm then for any other valid schedule $S' = (i_1, \ldots, i'_m)$, for all $j \in \mathbb{N}$, either $t_j(S) \leq t_j(S')$ or $t_j(S') = \infty$
  Proof: By induction on j. For the inductive step, assume that $t_k(S) \leq t_k(S')$. Note that $a_{i'_{k+1}} \geq t_k(S') \geq t_k(S)$ so job i'_{k+1} is available to S. Since S always chooses the job with the earliest completion time out of the available jobs, $t_{k+1}(S) = b_{i_{k+1}} \leq b_{i'_{k+1}} = t_{k+1}(S')$.
- This algorithm can be implemented in $O(n \log n)$ time by first sorting the jobs in increacing order of b_i

Interval Scheduling Problem 2 (Mimimizing maximum lateness): Given n jobs, each of which has length $l_i > 0$ and a deadline d_i, complete the jobs one by one such that the maximum lateness of any single job is minimized. Formally, define $t_S(i_k) = \sum_{j=1}^{k} l_{i_j}$ to be the time at which job i_k is finished, and find a schedule $S = (i_1, \ldots, i_n)$ such that $\max_{j \in [n]}\{t_S(i_j) - d_{i_j}\}$ is minimized.

Greedy Algorithm for Interval Scheduling 2 (Earliest Deadline First): Take the schedule S_{greedy} such that $d_{i_1}, \ldots, d_{n_1}$ are in increacing order.

Properties:

- The schedule S such that $d_{i_1}, \ldots, d_{n_1}$ are in increacing order minimizes $\max_{j \in [n]}\{t_S(i_j) - d_{i_j}\}$
  Proof: Given an optimal schedule S that is not S_{greedy}, we can obtain S_{greedy} by exchanging pairs of jobs without increacing maximum lateness. Therefore, S_{greedy} is optimal.
- If S is an optimal schedule, and $d_{i_j} > d_{i_{j+1}}$ then the schedule S' obtained by swapping i_j and i_{j+1} is also optimal.
  Proof:
  1. Swapping i_j and i_{j+1} does not affect the lateness of any other job as for all $j' \in [n] \setminus \{j, j+1\}$, $t_{S'}(i_{j'}) = t_S(i_{j'})$
  2. $t_S(i_{j+1}) = t_{S'}(i_j) = t_{S'}(i_{j+1}) + l_{i_j}$ as schedule S' puts job i_{j+1} before job i_j

  This implies that $t_S(i_{j+1}) - d_{i_{j+1}} > t_{S'}(i_{j+1}) - d_{i_{j+1}}$ and $t_S(i_{j+1}) - d_{i_{j+1}} > t_{S'}(i_j) - d_{i_j}$ as $d_{i_j} > d_{i_{j+1}}$. Thus, the lateness of job i_{j+1} for schedule S is greater than the lateness of both job i_j and i_{j+1} for schedule S'. So the maximum lateness for S' is less than or equal to the maximum lateness of S.

Lecture 4: Dijkstra's Algorithm

Shortest Path Problem: Given a directed graph $G = (V, E)$ with edge lengths $\{\ell_{uv} : (u, v) \in E\}$, a starting vertex s, and a destination vertex t, find the shortest path from s, to t.
Properties:

- If the edge lengths are non-negative, at each step, Dijkstra's algorithm correctly computes the distance from s to each vertex $u \in S$
  Proof: By induction on .. Inductive step: assume the distances $\{d_u : u \in S\}$ are correct. Let (u, v) be the next edge that Dijkstra's algorithm considers where $v \notin S$. This gives a path P of length $d_u + \ell_{uv}$. If P' is another path from s to v, let (u', v') be the first edge on P' leaving s. $d_{u'} + \ell_{u'v'} \geq d_u + \ell_{uv}$, so the length of P' is greater than or equal to P.

Algorithm 1 Dijkstra's Algorithm

Let $S \subseteq V$ be the set of explored nodes, and $E_{unexplored}$ be the set of unexplored edges.
while $S \neq V$ **do**
  Choose the edge $e = (u, v) \in E_{unexplored}$ which minimizes $d_u + \ell_{uv}$
  If $v = t$, we've found the shortest path.
  If $v \notin S$, we add v to S and set $d_v = d_u + \ell_{uv}$, and add $v's$ edges to $E_{unexplored}$
end while

Lecture 6: Minimum Spanning Trees

Minimum Spanning Tree: Given a graph $G = (V, E)$, the mimimum spanning tree is a set of edges $T \subseteq E$ such that (V, T) is a tree and $\sum_{e \in T} \ell_e$ is minimized.

Kruskal's Algorithm: Sort the edges E in order of their length. We then go through the edges one by one and take each edge which does not form a cycle.

- Stored Data: A set T of the edges taken thus far
- Iterative Step: We take the next edge $e \in E$. If $T \cup \{e\}$ contains a cycle, discard e, otherwise add e to T. The algorithm terminates when $|T| = n - 1$ or there are no edges left to consider

Determining if $T \cup \{e\}$ creates a cycle can be done in $O(\alpha(n))$ time via union-find where $\alpha(n)$ is the inverse Ackermann function.

Prim's Algorithm: Iterativley accepts the shortest edge which leads to a new vertex.

- Stored Data: A set T of edges and a set S of verticies taken thus far, and a set of edges $E_{candidate}$ with their lengths ℓ_e
- Iterative Step: Remove the shortest edge $e = \{v, w\} \in E_{candidate}$. If w is not in S, then we add w to S and e to T, and update $E_{candidate}$ with all of w's edges. The algorithm terminates when $S = V$ or $E_{candidate}$ is empty (in which case G is not connected).

A cut is a partition (L, R) of the verticies of G where $L \neq \emptyset$ and $R \neq \emptyset$. We sat ab edge $e = \{u, v\}$ crosses the cut (L, R) if u is in L and v is in R or vice versa.

Theorem: For each edge $e \in E$, e is in the MST if and only if there exist a cut (L, R) such that e is the shortest edge crossing (L, R)

Proof: If e is the shortest edge crossing (L, R), assume $e \notin T$. If so, adding e creates a cycle, which contains some other edge e' crossing (L, R). So $T' = (T \cup \{e\}) \setminus \{e'\}$ has shorter total length than T, a contradiction.
If $e = \{u, v\} \in T$, take L to be the set of verticies reachable from u using the edges $T \setminus \{e\}$. If there were a shorter edge e' crossing (L, R), $T' = (T \cup \{e'\}) \setminus \{e\}$ would have shorter total length.

Theorem: For all edges $e \in E$, $e \notin T$ if and only if there exists a cycle C such that e is the longest edge of C

Correctness of Kruskal's Algorithm: If e is the shortest edge crossing a (L, R), adding e cannot create a cycle so e will be taken. If Kruskal's algorithm takes $e = \{u, v\}$, let T be the set of edges which were taken so far. Take L to be the set of verticies reachable from u with the edges in T. There cannot be a shorter edge e' crossing (L, R), as otherwise e' would've been considered before e.

Correctness of Prim's Algorithm: Observe that at each step, Prim's algorithim takes the shortest edge crossing the cut between S and the remaining verticies. So Prim's algorithim only takes edges in T. If G is connected, the output of Prim's algorithim bust be connected. So all edges of T must be taken.

Lecture 7: Divide and Conquer Algorithms

Mergesort: A recursive sorting algorithm which takes $O(n \log n)$ time. Given an array:

1. Mergesort elements 0 through $\lfloor \frac{n}{2} \rfloor - 1$ of the array
2. Mergesort elements $\lfloor \frac{n}{2} \rfloor$ through $n - 1$ the array
3. Merge step: If A and B are two sorted arrays of lengths n_1 and n_2 then we can merge these arrays into a sorted array C of length $n_1 + n_2$ as follows:

- Start with $i = 0$ and $j = 0$
- If $j \geq n_2$ or $i < n_1$ and $a_i < b_j$ then set $c_{i+1} = a_i$ and increment i. If $i \geq n_1$ or $j \leq n_2$ and $b_j \leq a_i$ then set $c_{i+1} = b_j$ and increment j.

Lecture 8: Recurrence Relations and the Master Theorem

Two methods for solving recurrence relations:

1. Expand out the recurrance relation by substituting it into itself repeatedly. Intuitive but only works for simple relations. Example: $T(1) = 1,\ T(n) = T(\frac{n}{2}) + 5$

$$T(n) = T(\frac{n}{2}) + 5 \tag{1}$$
$$= T(\frac{n}{4} + 5) + 5 \tag{2}$$
$$= T(\frac{n}{2^k}) + 5k \tag{3}$$
$$T(n) = T(1) + 5\log_2(n) \tag{4}$$

2. Make an educated guess and solve for the coefficients.

Master Theorem: If $T(n) \leq aT(\frac{n}{b}) + O(n^c)$, letting $c_{crit} = \log_b(a)$,

1. If $c < c_{crit}$, $T(n)$ is $O(n^{c_{crit}})$
2. If $c > c_{crit}$, $T(n)$ is $O(n^c)$
3. If $c = c_{crit}$, $T(n)$ is $O(n^{c_{crit}} \log(n))$

Example: $T(n) \leq 2T(\frac{n}{2}) + O(n)$ So $a = 2,\ b = 2,\ c = c_{crit} = 1$ for a runtime of $O(n \log n)$

Example: Stoogesort:

- If $n \leq 2$, swap the elements if needed. Otherwise, Stoogesort the first 2/3 of the array, Stoogesort the second 2/3 of the array, Stoogesort the first 2/3 of the array.
- Calls itself 3 times so $T(n) \leq 3T(\frac{n}{\frac{3}{2}}) + O(1) \rightarrow a = 3,\ b = \frac{3}{2},\ c = 0,\ O(n^{\log_{1.5} 3})$

Lecture 9: More Divide and Conquer

Counting Inversions: Given an array A of n distinct numbers, how many pairs $i < j \in [n]$ are such that $a_i > a_j$?

- Base case: If $n = 1$, the array is already sorted and there are no inversions. Otherwise,
  1. Apply this algorithm to each half of A, and let $count_L$ and $count_R$ be the number of inversions in each half.
  2. If A and B are two sorted arrays of lengths n_1 and n_2 then we can count the inversions ($a_i > b_j$) and merge these sequences into C as follows:
     - Initialization: Start with $i = 0$, $j = 0$, and $count_{between} = 0$
     - Iterative step: $j \geq n_2$ or $i < n_1$ and $a_i < b_j$ then set $c_{i+1} = a_i$ and increment i. If $i \geq n_1$ or $j \leq n_2$ and $b_j \leq a_i$ then set $c_{i+1} = b_j$, increace $count_{between}$ by $n_1 - i$, and increment j.

     This works because for each element of B added to C, $b_j < a'_i$ for all remaining a'_i in A, so each of these is an inversion with b_j.

Karatsuba algorithm for multiplication: Given two n-digit numbers x and y, taking $k = \lceil \frac{n}{2} \rceil$, we can write $x = 10^k x_1 + x_2$ and $y = 10^k y_1 + y_2$ where x_1 and y_1 are $(n-k)$-digit numbers and x_2 and y_2 are k digit numbers. We now have that:

$$xy = 10^2 k x_1 y_1 + 10^k x_1 y_2 + 10^k x_2 y_1 + x_2 y_2$$

So we can compute xy by taking the sum of smaller products. With some clever rearranging we can only compute three products:

$$xy = 10^2 k x_1 y_1 + 10^k((x_1 + x_2)(y_1 + y_2) - x_1 y_1 - x_2 y_2) + x_2 y_2$$

The reccurrence relation for which is $T(n) = 3T(\frac{n}{2}) + O(n)$ which gives that $T(n)$ is $O(n^{\log_2 3})$

Lecture 10: Even More Divide and Conquer

Closest Pair of Points:

- Sort the points by their x-coordinante. Split the points into the left half L and the right half R
- Find the closest pair of points in L and the closest pair of points in R. Let d_L and d_R be the distances between them. Let $d = \min\{d_R, d_L\}$
- We still need to check the pairs of points between L and R. For this, its sufficient to check the middle strip M of points within distance d of the dividing line $x = x_{mid}$.
  - Split the middle strip into $\frac{d}{2} \times \frac{d}{2}$ boxes. Oberve that if the closest pair of points is between L and R, they must be within two rows of each other. And each box has at most one point as if a box contained two points, these points would be within $\frac{d}{2}$ of each other and thus on the same side, contradicting our choice of d.
  - So its sufficient to compare each point in M with the next eleven points (sorted by y-coordinante)

Implementation:

- Preprocessing: Sort the points by both their x and y-coordinates.
- Divide: Split the points into L and R st. x_{mid} is the median of the x-coordinates
- Conquer: We find the closest pairs of points in L and R, and $d = \min\{d_L, d_R\}$. We exctact the points of L and R in order of their y-coordinate and give this as input to the recursive algorithm.
- Let M be the set of poinst with distance d of the line $x = x_{mid}$. For each point in M, compare it to the next eleven points.

Lectore 11: Dynamic Programming

Weighted Inverval Scheduling: Given n jobs, each with a set time interval $[a_i, b_i]$ and a weight w_i, find a schedule S which maximuzes the total weight of accepted jobs. ie. fid a valid schedule $S = (i_1 \dots i_m)$ which maximizes $\sum_{i=1}^{|S|} w_{i_j}$

- Preprocessing: Sort the times $\{a_i : i \in [n]\} \cup \{b_i : i \in [n]\}$ and let $(t_1 \dots t_n)$ be the sorted times.
- Consider the following subproblem: for all $t \in [n]$, what is the maximum total weight that can be completed by time t?
- Define S_t to be the set of all valid schedules completed by time t, and let $w(t) = \max_{S \in S_t} \sum_{j=1}^{|S|} w_{i_j}$, which gives us the following recurrence relation:
  - If $t_{k+1} = a_i$ for some $i \in [n]$, then $w(t_{k+1}) = w(t_k)$
  - If $t_{k+1} = b_j$ for some $j \in [n]$, then $w(t_{k+1}) = \max(w(t_k), w(a_j) + w_j)$

  Proof: Observe that if $t_{k+1} = a_i$ then any shcedule which finishes by time t_{k+1} also finishes by time t_k, so $w(t_{k+1}) = w(t_k)$.
  Lower Bound: Observe that there is a schedule S that finishes by time t_k with total weight $w(t_k)$, so $w(t_{k+1}) \geq w(t_k)$. There is also a schedule S that finishes by time a_j with total weight $w(a_j)$. Adding job j to this schedule gives a schedules S' which finishes by time $k_{k+1} = b_j$ and has total wieght $w(a_j) + w_j$, so $w(t_{k+1}) \geq w(a_j) + w_j$. So $w(t_{k+1}) \geq \max(w(t_k), w(a_j) + w_j)$.
  Upper Bound: There is a schedule S' that finishes by time t_{k+1} with weight $w(t_{k+1})$. If S' does not contain job j then S' finishes by time t_k and thus has weihgt at most $w(t_k)$. If S' contains job j then S' consists of a schedule finishing at time a_j plus the job j. So $w(t_{k+1}) \leq \max(w(t_k), w(a_j) + w_j)$
- Remark: If the assumption that there are no ties does not hold, then we can preprocess the problem by subtracting ε_j from each b_{i_j} and adding ε_j to each a_{i_j}, or employing some other tie breaking constraint.

Bellman-Ford Algorithm for Shortest Paths

Shortest Paths: In this variant, edge weights can be negative, but G cannot contain a negative cycle.

- We consider the following subproblems: for all $k \in [n]$, what is the minimum length of a walk from s to v which uses at most k edges?
- Let $d_k(v)$ be the minimum length of a walk from s to v which uses at most k edges
- If there are no negaitve cycles in G, then for all $k \in \mathbb{N}$, and all $v \in V$, $d_k(v)$ is also equal to the minimum length of the a path from s to v which uses at most k edges
- We have the following recurrence relation:

$$d_{k+1}(v) = \min\{d_k(v), \min_{u \in V:(u,v) \in E} \{d_k(u) + \ell_{uv}\}\}$$

  Proof:

- Upperbound: There is a walk W of length $d_k(v)$ from s to v that uses $k < k+1$ edges, so $d_{k+1}(v) \leq d_k(v)$. For all $u \in V$ there is a walk of length $d_k(u)$ from s to u which uses at most k edges. Adding the edge (u, v) to this walk we obtain a walk from s to v of length $d_k(u) + ell_{uv}$ which uses at most $k+1$ edges.
- Lowerbound: If W uses at most k edges then it has length at least $d_k(v)$, thus $d_{k+1}(v) \geq d_k(v) \geq \min\{d_k(v), \min_{u \in V:(u,v)\in E}\{d_k(u) + \ell_{uv}\}\}$. If W uses $k+1$ edges and the final edges is (u, v) then W has length at least $d_k(u) + ell_{uv}$ and thus $d_{k+1}(v) \geq d_k(u) + ell_{uv} \geq \min\{d_k(v), \min_{u \in V:(u,v)\in E}\{d_k(u) + \ell_{uv}\}\}$

- Runtime: let $n = |V|$ and $m = |E|$, Eacch $d_k(v)$ takes time $O(\text{indegree}(v))$ to compute. So the algorithm takes $O(mn)$ time.
- Finding Negative Cycles: We can check if there are any updates when we look at walks of length n. Since all paths in G have length $n-1$ if there are any updates then there is a negative cycle, and there will be updates for all k.

Lecture 12: Knapsack and Longest Common Subsequence:

Knapsack: Given n objects with weights $w_1 \dots w_n$ and values $v_1 \dots v_n$, and a weight capacity c, what is the maximum total value which can be taken wihtout exceeding the weight capacity c?

- Subproblems: For each $k \in [n]$ and $c' \in [c]$, what is the maximum total value we can obtain from the first k objects without exceeding c'. Define $v(k, c') = \max_{I \subseteq [k]:\sum_{i\in I} w_i \leq c'}\{\sum_{i\in I} v_i\}$
- $v(k+1, w) = \max\{v(k, w), v(k, w - w_{k+1}) + v_{k+1}\}$

Longest Common Subsequence: Given two strings $a_1 \dots a_i$ nad $b_1 \dots b_j$, what is the longest common subsequence $S(i, j)$?

- Subproblem: What is $S(i, j)$ for $i \in [n],\ j \in [m]$?
- Recurrence relation:
  - If $a_i = b_j$ then $S(i, j) = S(i-1, j-1) + 1$
  - If $a_i \neq b_j = \max\{S(i-1, j),\ S(i, j-1)\}$

  where we take $S(0, j) = S(i, 0) = 0$

Lecture 13: Non-crossing matchings:

Non-Crossing Matching: Given an undirected graph G a non crossing matching is a set of edges $M \subseteq E$ such that if we put the verticies $v_1 \dots v_n$ in a circle in clockwise order and draw the edges as chords, no two edges of M cross or share an endpoint, such that for all pairs of edges v_i, v_j and $v_k, v_l \in M$, where $i < j$ and $k < l$, we do not have that $i < k < j < l$ or $k < i < l < j$ Given an undirected graph G, what is the largest non-crossing matching of G?

- Let $M(i, j)$ be the size of the maximum non-crossing matching on $v_i \dots v_j$
  - Base cases:
    - If $j \leq i,\ M(i, j) = 0$
    - If $j = i+1,\ M(i, j) = \begin{cases} 1 \text{ if } \{v_i, v_j\} \in E \\ 0 \text{ if } \{v_i, v_j\} \notin E \end{cases}$
  - Recurrence Relation: We can consider which edge incident to v_j to take, if any.

$$M(i, j) = \max\{M(i, j-1), \max_{i':i\leq i'<j(v_{i'}, v_j)\in E} M(i', i'+1) + M(i'+1, j-1) + 1\}$$

Ford-Fulkerson Algorithm for Max Flow

Max Flow Probelm: Given a directed graph G with non-negative edge capacities $\{c_e : e \in E\}$, what is the maximum flow from s to t? Flow: No capacity is exceeded, for all verticies v except s and t, the flow into v equals the flow out of v.
Max-Flow/Min-Cut Theorem: The maximum flow from s to t is equal to the minimum capacity of a cut $C = (L, R)$ where $s \in L$ and $t \in R$.
Ford-Fulkerson: Start with 0 flow, and iterativley do the following

1. Compute the residual graph for the current flow, which represents the remaining capacities of the edges
2. Find a flow path from s to t in the residual graph. If no flow path exists, stop and output the current flow
3. Route as much flow as possible along the flow path and update the current flow.

For the residual graph, we have both forward edges representing the remaining capacities, and backwards edges representing the possibility of undoing flow.
Correctness: If there is no flow path, consider the cut where $L = v$ such that there is a flow path from s to v in the residual graph. Observe that:

1. All edges from L to R must be at full capactiy as otherwise the residual graph would have a forward edge from L to R with positive capactiy.
2. All edges from R to L must have 0 flow as otherwise the residual fraph would have a backwards edge from L to R with positive capactiy

This implies that the flow from s to t is the same as the flow across the cut.
Runtime: If all capacities are non-negative integers and the value of the maximum flow is F, Ford-Fulkerson runs in $O(F \cdot |E|)$ time. Flow path takes $O(|E|)$.

Maximum Bipartite Matching: Given a bipartite graph G with sets of verticies A and B, what is the largest matching of G?
Hall's Theorem: The maximum matching has size $|A|$ if and only if for all $V \subseteq A, |N(V)| \geq |V|$
Konig's Theorem: The maximum size of a matching of G equals the minimum size of a vertex cover of G. V is a vertex cover of G if every edge $e \in E$ is incident to at least one vertex in V.
Reducing the problem to max flow:

1. Add a vertex s and edges from s to all verticies in A
2. Make all edges between A and B go from A to B
3. Add a vertex t and add edges from all verticies in B to t

Given a matching M with size k, there is a corresponding flow in G' with value k. Similarly, given an integer valued flow from s to t with value k, we can obtain a matching of G with size k
Algorithm for finding maximum bipartite matching:

1. Use Ford-Fulkerson to obtain an integer valued max flow F on G'
2. Take $M = \{\{a_i, b_j\} : (a_i, b_j) \text{ has flow 1 in } F\}$

Given an undirected graph G and a matching M, an augmenting path is a path $v_1 \to \cdots \to v_k$ such that k is even and:

1. v_1 and v_k are not incident to an edge in M
2. For all odd $j \in [k]$, $e = \{v_j, v_{j+1}\} \notin M$
3. For all even $j \in [k]$, $e = \{v_j, v_{j+1}\} \in M$

Theorem: If M is not a maximum matching then there exists an augmenting path. Proof sketch: If M is not a maximum matching then there exists a matching $|M'| > |M|$. Consider the multi-graph with verticies $V(G)$ and edges $M \cup M'$. Every vertex in this multigraph has degree ≤ 2.
Proof of Konigs:

Lecture 19: Edge and Vertex Disjoint Paths

Edge-disjoint Paths: Reduce to max flow my making each edge have capacity 1. Start from s and take a walk to t using edges with flow 1. Then delete the edges of the walk.
Vertex-disjoint Paths:

1. Replace each vertex v with two verticies, v_{in} and v_{out} with an edge of capactiy 1 from v_{in} to v_{out}
2. For each edge, $u \to v$, add an edge with capactiy 1 from u_{out} to v_{in} $(s = s_{out},\ t = t_{in})$
3. Find the max flow k

Menger's Theorem: Given a directed graph G and verticies $s, t \in V$ such that $s, t \notin E$, the maximum number of vertex-disjoint paths from s to t is equal to the minimum size of a vertex separator S, $S \subseteq V \setminus \{s, t\}$ such that every path from s to t contains at least one vertex in S, between s and t.
Proof: (Easy Direction) If there is a vertex separator S of size k then therre are at most k vertex disjoint paths from s to t, so each must contain a vertex in S. (Other Direction): If the max number of vertex disjoint paths is k, then the max flow in G' is k, so by the max-flow min-cut theorem there exists a cut with capacity k. We can tweak this cut to obtain a cut C' with capactiy k such that only edges $v_{in} \to v_{out}$ cross C'. We take S to be v such that $v_{in} \to v_{out}$ crosses C'.

Lecture 20: NP, NP-Hardness, NP-Completeness

Non-Deterministic Polynomial Time (NP): the class of YES/NO problems for which a solution can be verified in polynomial time. ie. for which there is a non-deterministic polynomial time algorithm.
Ex. 3 coloring problem: Given a graph G, is it possible to color the verticies of G with three different colors? 3-coloring takes $2^{\Omega(n)}$ time to solve, but can be verified in polynomial time.
Non-Deterministic: A is a non-deterministic algorithm if

1. If the answer for instance x is YES, A has a non-zero chance of saying YES.
2. If the answer to x is NO, A always returns NO.
3. $\exists B, c > 0$, such that A always terminates in Bn^c

Can verify a solution in polynomial time $\rightarrow$ there exists a non-deterministic polynomial time algorithm. $\longleftrightarrow$ there exists a non-deterministic polynomial time algorithim $\rightarrow$ there is a kind of solution which can be verified in polynomial time.
Independent Set Problem: Given a graph G, is there a set I of k verticies such that no two verticies in I are adjacent?
If I is an independent set then $V = V(G) \setminus I$ is a vertex cover.

Lecture 21: NP-completeness Reductions

NP-Hardness: A problem is NP-hard if every problem in NP can be poly-time reduced to it.
NP-completness: A problem is NP-complete if it is both in NP and NP-hard.
If probelm A is NP-hard and there is a poly-time reduction from A to B, then B is NP-hard.
Proof: For any problem p in NP, p is poly-time reducible to A. Since A is poly-time reducible to B, p is poly-time reducible to B as well.
If A and B are in NP-Complete then they are poly-time reducible to eachother.
Showing problem P is NP-complete:

1. Show that the problem P is in NP
2. Choose an NP-complete probelm A and give a polytime reduction from A to P

Thus, P is NP-hard. Ex.
Circut-SAT: Given a boolean circuit, is there an assigment of the inputs which makes the output True?
Cook-Levin Theorem: Circut-SAT is NP-complete
Idea: given a problem in NP, we can implement the verifier using a boolean circuit.
3-SAT: Given m clauses, where each clause is the OR or 3 literals, is there an assigment which satisfies the clauses?
Theorem: 3-SAT is NP-complete
Proof: To show it is NP-hard, we'll reduce circuit-SAT to 3-SAT. We'll have a variable for each wire of the circuit. For each gate of the circuit we'll add clauses to ensure that the variables behave as expected.
Theorem: Independent Set is NP-complete.
To show it is NP-Hard, we'll reduce 3-SAT to Independent Set. For each clause c_i, we'll create 3 verticies v_{i1}, v_{i2}, v_{i3}. $v_{ij} \in I$ says clause c_i is satisfied because its j'th literal is true. We'll add edges between conflicting verticies (one says a variable is true and the other says its false). We'll also add edges between the verticies within each clause.