CS 180 Midterm Notes

Joonwon Lee

April 30, 2023

1 Gale Shapely

Question: Given a set of preferences among employers and applicants, can we assign applicants to employers so that for every employer E, and every applicant A who is not scheduled to work for E, at least one of the following two things is the case?

i) E prefers every one of its accepted applicants to A; or
ii) A prefers her current situation over working for employer E.
$\rightarrow$ Outcome is stable
Goal is a set of marriages with no instabilities.

Now show that there exists a stable matching for every set of preference lists among the men and women.

The Algorithm:
Initially all $m \in Mandw \in W$ are free

While there is a man m who is free and hasn't proposed to every woman
Choose such a man m
Let W be the highest-ranked woman in m's preference list to whom m has not yet proposed
If W is free then (m, w) become engaged
Else W is currently engage to m'
If W prefers m' to m then m remains free
Else W prefers m to m' (m, w) become engaged m' becomes free

The G-S algorithm terminates after at most n^2 iterations of the While loop

Proof:
Each iteration consists of some man proposing to a woman he has never proposed to before. So if we let $P(t)$ denote the sets of pairs (m, w) such that m has proposed to w by the end of iteration t, we see that for all t, the sizes of $P(t+1)$ is strictly greater than the size of $P(t)$. But there are only n^2 possible pairs of men and women in total, so the value of $P(.)$ can increase at most n^2 times over the course of the algorithm. At most n^2 iterations.

If m is free at some point in execution of the algorithm, then there is a woman to whom he has not yet proposed.

Proof:
Suppose there comes a point when m is free but has already proposed to every woman. Then by the fact that all women remains engaged from the point at which she receives her first proposal, each of the n women is engaged at this point in time. Since the set of engaged pairs forms a matching, there must alost be n engaged men at this point in time. But there are only n men total, and m is not engaged so a contradiction.

The set S returned at termination is a perfect matching.

Proof:
The set of engaged pairs always forms a matching. Let us suppose that the algorithm terminates with a free man m. At termination, it must be the case that m had already proposed to every woman, for otherwise the While loop would not have exited. But this contradicts the above statement, which says that there cannot be a free man who has proposed to every woman.

Consider an execution of the G-S algorithm that returns a set of pairs S. The set S is a stable matching.
Proof:
Proof by contradiction where we assume that there is an instability with respect to S (pairing). As defined earlier, such an instability would involve two pairs (m, w) and (m', w'), in S with the properties that

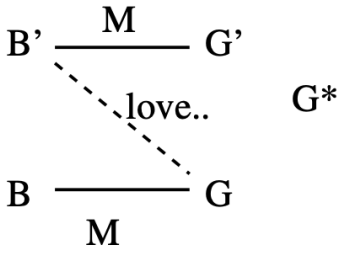
- m prefers w' to w, and
- w' prefers m to m'.

In the execution of the algorithm that produced S, m's last proposal was by definition, to w. Now we ask: Did m propose to w' at some earlier point in this execution? If he didn't then w must occur higher on m's preference list than w' which contradicts our assumption that m prefers w' to w. If he did then w' chose m" over m. However, m' is the final match with w' which means m" = m' or w' prefers her final partner m' to m". Either way this contradicts all our assumptions. Therefore S is a stable matching.

Unfairness

When men propose men get the most optimal and women get the most pessimal result.

Consider the following preference list:
m prefers w to w'
m' prefers w' to w
w prefers m' to m
w' prefers m to m'



Do all executions of G-S algorithm produce the same result?

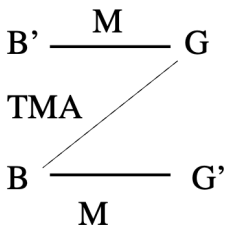
Approach: Uniquely characterize the matching that is obtained and then show that all executions result in the matching with this characterization.
What is the characterization? → Each man ends up with the "best possible/valid partner".

Proof by contradiction Suppose that some execution of the G-S algorithm results in a matching S in which some man is paired with a woman who is not his best valid partner. This means that a man is rejected by a valid partner w during execution. Since men propose in decreasing order of preference, and since this is the first time such a rejection has occurred, it must be that w is m's best valid partner best(m). This reject may have occurred as w turned down m in favor of an existing engagement or because w broke her engagement to m in favor of another man. At this moment w is paired with m'. However, we now have to ask who is m' paired with as (m, w) are a valid match. Suppose it is w'.

Since the rejection of m by w was the first rejection of a man by a valid partner in the execution, it must be that m' had not been rejected by any valid partner at the point when he became engaged to w. Since w' is a valid partner to m', it must be that m' prefers w to w', but we have seen that w prefers m' to m for she rejected m in favor of m'. Since (m', w) is not in the stable matching, it follows that (m', w) is an instability in S'. This contradicts our claim that S' is stable therefore proved by contradiction.

In the stable matching S*, each woman is paired with her worst valid partner

Proof by contradiction
Suppose there was a pair (m, w) in S where m is not the woman's worst valid partner of w. Then there is a stable matching S' in which w is paired with a man m' whom she likes less than m. In S', m is paired with a woman $w' \neq w$ since w is the best valid partner of m, and w' is a valid partner of m, we see that m prefers w to w'. But from this it follows that (m,w) is an instability in S' therefore contradicts the statement that S' is stable.



2 Basics of Algorithm Analysis

2.1 Worst-Case Running Times and Brute-Force Search

Bound on the largest possible running time the algorithm could have over all inputs of a given size N, and see how this scales with N.

2.2 Polynomial Time as a Definition of Efficiency

Arithmetically, we can formulate this scaling behavior as follows. Suppose an algorithm has the following property. There are absolute constants $c > 0$ and $d > 0$ so that on every input instance of size N, its running time is bounded by cN^d primitive computational steps (at most proportional to N^d). In any case, if this running time bound holds for some c and d, then we say that the algorithm has a polynomial running time, or that it is a polynomial time algorithm. Ex) $N \to 2N : cN^d$ to $c(2N)^d = c * 2^d * N^d$ which is a slow down by a factor of 2^d which is a constant.

2.3 Asymptotic Order of Growth

Asymptotic Upper Bounds
Let T(n) be a function - say the worst case running time of a certain algorithm on an input of size n. We say that T(n) is O(f(n)) read as T(n) is order f(n). For sufficiently large n, the function T(n) is bounded above by a constant multiple of f(n) if there exist constants $c > 0$ and $n_0 \geq 0$ so that for all $n \geq n_0$ we have

$\mathbf{T(n)} \leq \mathbf{c} * \mathbf{f(n)}$. We say that T is asymptotically upper bounded by f.

This definition requires a constant c to exist that works for all n.
ex) $T(n) = pn^2 + qn + r \leq pn^2 + qn^2 + rn^2 = (p + q + r)n^2$ for all $n \geq 1$. Therefore, $T(n) \leq cn^2$ where $c = p + q + r$

Asymptotic Lower Bounds
There is a complementary notation for lower bounds. We want to show that an upper bound is the best one possible. To do this, we want to express the notion that for arbitrarily large input sizes n, the function T(n) is at least a constant multiple of some specific function f(n) $T(n)$ is $\Omega(f(n))$. If there exists constants $\epsilon > 0$ and $n_0 \geq 0$ so that for all $n \geq n_0$, we have $T(n) \geq \epsilon * f(n)$. We will refer to T in this case as being asymptotically lower bounded by f. We need to reduce the size of T(n) until it looks like a constant times n^2 rather than inflating it to look like n^2.

Asymptotically Tight Bounds
We can show that a running time T(n) is both O(f(n)) and also $\Sigma(f(n))$. Then in a natural sense we've found the right bound. T(n) grows exactly like f(n) to draw from within a constant factor. We denote this as $\Theta(f(n))$
Let f and g be two functions that

$$\lim_{x \to \infty} f(n)/g(n)$$

exists and is equal to some number $c > 0$. Then $f(n) = \Theta(g(n))$

Properties of Asymptotic Growth Rates

Transitivity If a function f is asymptotically upper bounded by a function g, and if g in turn is asymptotically upper bounded by function h, then f is asymptotically upper bounded by h. A similar property holds for lower bounds.

a) If f = O(g) and g = O(h) then f = O(h)
b) If f = $\Omega(g)$ and g = $\Omega(h)$ then f = $\Omega(h)$

Sums of Functions
Suppose that f and g are two functions such that for some other function h, we have f = O(h) and g = O(h) then f+g = O(h)

Let k be a fixed constant, and let $f_1, f_2, ...f_k$ and h be functions such that $f_i = O(h)$ for all i. Then $f_1 + f_2 + ...f_k = O(h)$.

Suppose that f and g are two functions (taking non-neg values) such that $g = O(f)$. Then $f + g = \Theta(f)$. In other words, f is an asymptotically tight bound for the combined function f+g.

Summary:
If f= O(g) and g = O(h) then f = O(h) *Same goes for Ω
If f = $\theta(g)$ and $g = \theta(h)$, then $f = \theta(h)$
If f = O(h) and g = O(h) then f+g = O(h)
$f_1 + f_2 + ...f_k = O(h)$ then the sum of all the functions is $O(h)$
g = O(f) then f+g = θ(f)

Asymptotic Bounds for Some Common Functions

Polynomials

Recall that a polynomial is a function that can be written in the form $f(n) = a_0+a_1n+a_2n^2+...a_dn^d$ for some constant $d > 0$, where the final coefficient a_d is nonzero. d is called the degree of the polynomial. The asymptotic rate of growth is determined by their higher order term (the one that determines the degree).

Logarithms

For every $b > 1$ and every $x > 0$, we have $\log_b n = O(n^x)$

Exponential

For every $r > 1$ and every $d > 0$, we have $n^d = O(r^n)$

From Discussion:

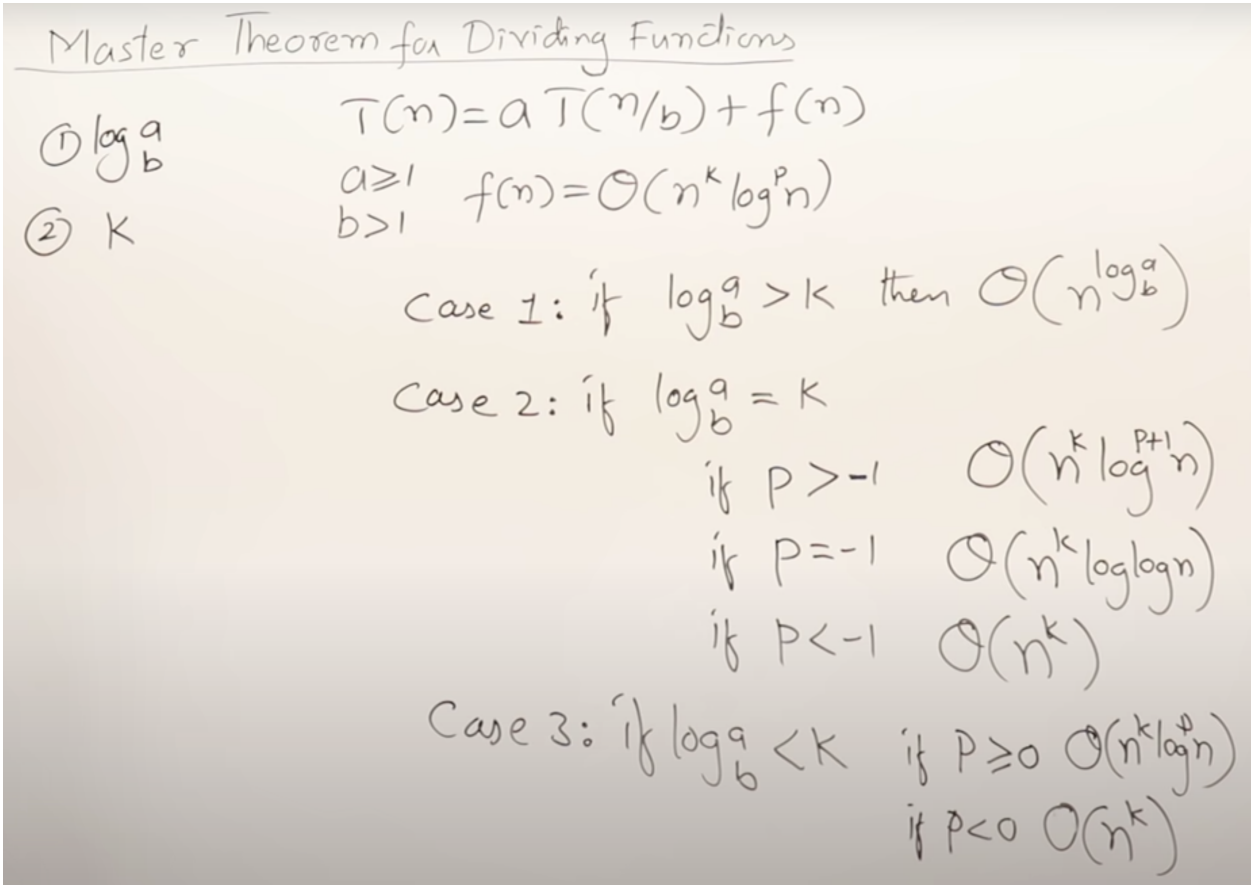
$$\lim_{n\to\infty} f(n)/g(n) = 0 \rightarrow f(n) = O(g(n)) \text{ or } g(n) = \Omega(f(n))$$

$$\lim_{n\to\infty} f(n)/g(n) = \infty \;\rightarrow g(n) = O(f(n)) \text{ or } f(n) = \Omega(g(n))$$

L'hopital's rule

$$\text{If } \lim_{x\to a} \frac{f(x)}{g(x)} = \frac{0}{0} \text{ or } \lim_{x\to a} \frac{f(x)}{g(x)} = \frac{\infty}{\infty}, \text{ then } \lim_{x\to a} \frac{f'(x)}{g'(x)}$$

Master's Theorem



3 Interval Scheduling - Greedy

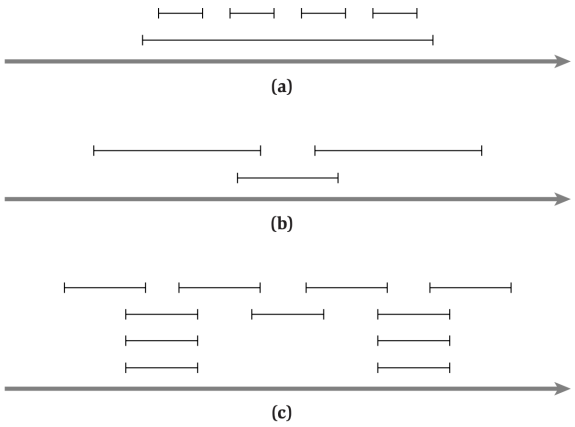
The basic idea in a greedy algorithm for interval scheduling is to use a simple rule to select a first request i_1. Once a request i_1 is accepted, we reject all requests that are not compatible with i_1. We then select the next request i_2 to be accepted, and again reject all requests that are not compatible with i_2.We'll say that a subset of the requests is compatible if no two of them overlap in time, and our goal is to accept as large a compatible subset as possible. Compatible sets of maximum size will be called optimal.

3.1 Select the available request that starts earliest

This doesn't yield an optimal solution. If the earliest request i is for a very long interval, then by accepting request i we may have to reject a lot of requests for shorter time intervals. Since our goal is to satisfy as many requests as possible, we will end up with a sub-optimal solution. The worst case scenario is when the first task goes on for the entire allocated time. Refer to Example (a).

3.2 Start out by accepting the request that requires the smallest interval of time

Better than 2.1 but still sub optimal.



Here (b) would prevent us from accepting the other two, which forms the optimal solution.

3.3 Least conflict

Here (c) the optimal solution is four intervals. But the greedy method suggests the middle request in the second row and thereby ensures a solution of size no greater than three.

3.4 Accept request that finishes first - MOST OPTIMAL

We ensure that our resource becomes free as soon as possible while still satisfying one request. In this way we can maximize the time left to satisfy other requests.

THE ALGORITHM R to denote the set of requests that we have neither accepted nor rejected yet, and use A to denote the set of accepted requests.

```
Initially let R be the set of all requests, and let A be empty
While R is not yet empty
  Choose a request i∈R that has the smallest finishing time
  Add request i to A
  Delete all requests from R that are not compatible with request i
EndWhile
Return the set A as the set of accepted requests
```

Proof
1. A is a compatible set of requests
2. Optimal
We want to show that $|A| = |Optimal|$. The idea is to find a sense in which our greedy algorithm stays ahead of this solution O. We will compare the partial solutions that the greedy algorithm constructs to initial segments of the solution O, and show that the greedy algorithm is doing better in a step-by-step fashion.

We introduce some notation to help with this proof. Let $i_1...i_k$ be the set of requests in A in the order they were added to A. Note that $|A|$ = k. Similarly, let the set of requests in O be denoted by $j_1...j_m$. Our goal is to prove that k = m. Assume that the requests in O are also ordered in the natural left-to- right order of the corresponding intervals, that is, in the order of the start and finish points. Assume that the requests in O are also ordered in the natural left-to-right order of the corresponding intervals, that is, in the order of the start and finish points. Note that the requests in O are compatible, which implies that the start points have the same order as the finish points. Our intuition for the greedy method came from wanting our resource to become free again as soon as possible after satisfying the first request. And indeed, our greedy rule guarantees that $f(i_1) \leq f(j_1)$. This is the sense in which we want to show that our greedy rule stays ahead (each of its intervals finishes at least as soon as the corresponding interval in the set O). Thus we now prove that for each $r \geq 1$, the r^{th} accepted request in the algorithm's schedule finishes no later than the r^{th} request in the optimal schedule.

For all indices $r \leq k$ we have $f(i_r) \leq f(j_r)$.
Base: For r=1 the statement is clearly true: the algorithm starts by selecting the request i_1 with minimum finish time.

Inductive step: Assume as our induction hypothesis that the statement is true for r - 1, and we will try to prove it for r. In order for the algorithm's r^{th} interval not to finish earlier as well, it would need to fall behind as shown.

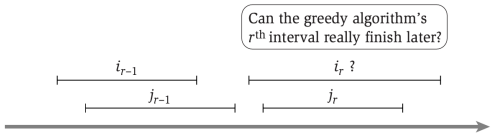


Figure 4.3 The inductive step in the proof that the greedy algorithm stays ahead.

There is a simple reason to why this could not happen. Rather than choosing a later finishing interval, the greedy algorithm has the option (at worst) of choosing j_r and thus fulfilling the induction step. More precisely, $f(j_{r-1}) \leq s(j_r)$. Combine this with $f(i_{r-1}) \leq f(j_{r-1})$, we get $f(i_{r-1}) \leq s(j_r)$ which means $f(i_r) \leq f(j_r)$.

Implementation and Running Time
We can make our algorithm run in time **O(nlogn)**. We begin by sorting the n requests in order of finishing time and labeling them in this order, that is we will assume that $f(i) \leq f(j)$ when $i < j$. This takes $O(nlogn)$. In an additional $O(n)$ time, we construct an array S[1...n] with the property that S[i] contains the value s(i). We now select requests by increasing f(i). We continue iterating through subsequent intervals until we reach the first j for which $s(j) > f$. This part of the algorithm is $O(n)$.

4 Minimum Lateness - Greedy

One approach would be to schedule the jobs in order of increasing length t_i, so as to get the short jobs out of the way quickly.

Counter example $\rightarrow$ $t_1 = 1$ and $d_1 = 100$ while the second job is $t_2 = 10$ and $d_2 = 10$.

Another approach is sort in increasing order of slack time $d_i - t_i$.

Counter example $\rightarrow$ $t_1 = 1$ and $d_1 = 2$ while the second job has $t_2 = 10$ and $d_2 = 10$. Sort by increasing slack would place the second job first in the schedule and the first job would incur a lateness of 9.
MOST OPTIMAL - Earliest Deadline First
Trivial: There is an optimal schedule with no idle time.
Now prove that it is actually optimal:
Start by considering an optimal schedule O. Our plan here is to gradually modify O, preserving its optimality at each step, but eventually transforming it into a schedule that is identical to the schedule A found by the greedy algorithm. This is called an exchange argument.

We say there is an inversion if a job i with deadline d_i is scheduled before another job j with deadline $d_j < d_i$. If there are jobs with identical deadlines then there can be many different schedules with no inversions. However, we can show that all these schedules have the same maximum lateness L.

4.1 All schedules with no inversions and no idle time have the same maximum lateness

If two different schedules have neither inversions nor idle time, then they might not produce exactly the same order of jobs, but they can only differ in the order in which jobs with identical deadlines are scheduled. In both schedules, the job with deadline d are all scheduled consecutively. Among the jobs with deadline d, the last one has the greatest lateness, and this lateness does not depend on the order of the jobs.

4.2 There is an optimal schedule that has no inversions and no idle time

Proof. By (4.7), there is an optimal schedule $\mathcal{O}$ with no idle time. The proof will consist of a sequence of statements. The first of these is simple to establish.

(a) *If $\mathcal{O}$ has an inversion, then there is a pair of jobs i and j such that j is scheduled immediately after i and has $d_j < d_i$.*

Indeed, consider an inversion in which a job a is scheduled sometime before a job b, and $d_a > d_b$. If we advance in the scheduled order of jobs from a to b one at a time, there has to come a point at which the deadline we see decreases for the first time. This corresponds to a pair of consecutive jobs that form an inversion.

Now suppose $\mathcal{O}$ has at least one inversion, and by (a), let i and j be a pair of inverted requests that are consecutive in the scheduled order. We will decrease the number of inversions in $\mathcal{O}$ by swapping the requests i and j in the schedule $\mathcal{O}$. The pair (i, j) formed an inversion in $\mathcal{O}$, this inversion is eliminated by the swap, and no new inversions are created. Thus we have

(b) *After swapping i and j we get a schedule with one less inversion.*

The hardest part of this proof is to argue that the inverted schedule is also optimal.

(c) *The new swapped schedule has a maximum lateness no larger than that of $\mathcal{O}$.*

It is clear that if we can prove (c), then we are done. The initial schedule $\mathcal{O}$ can have at most $\binom{n}{2}$ inversions (if all pairs are inverted), and hence after at most $\binom{n}{2}$ swaps we get an optimal schedule with no inversions.

So we now conclude by proving (c), showing that by swapping a pair of consecutive, inverted jobs, we do not increase the maximum lateness L of the schedule. ■

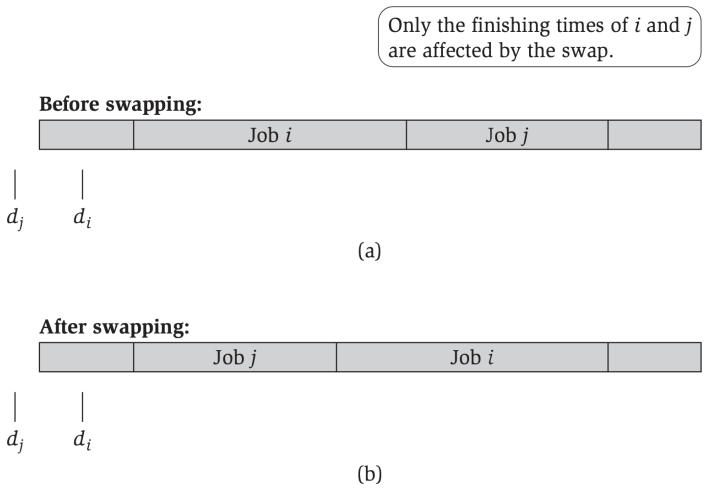
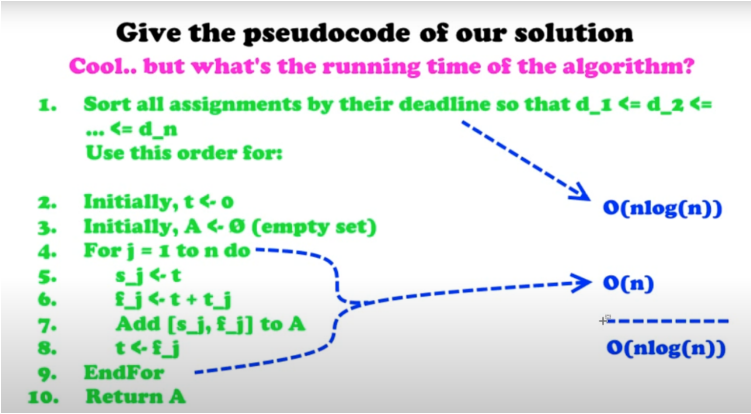


Figure 4.6 The effect of swapping two consecutive, inverted jobs.

4.3 The schedule A produced by the greedy algorithm has optimal maximum lateness L

Optimal schedule with no inversions exists. All schedules with no inversions have the same maximum lateness, and so the schedule obtained by the greedy algorithm is optimal.

4.4 Implementation and Running Times



5 Graphs

Every n-node tree has n-1 edges
Let G be an undirected graph on n nodes. Any two of the following statements implies the third.
1) G is connected
2) G does not contain a cycle
3) G has n-1 edges

5.1 Graph connectivity and graph traversal

5.1.1 Breadth First Search

Explore outward from s in all possible directions, adding nodes one layer at a time. Thus we start with s and include all nodes that are joined by an edge to s. This is the first layer of the search. We then include all additional nodes that are joined by an edge to any node in the first layer. This is the second layer. We continue until there are no longer new nodes.

We can define layers L1, L2, L3... constructed by the BFS algorithm:

- L0 is the set consisting of just s.
- Layer L1 consists of all nodes that are neighbors of s.
- Layer j+1 consists of all nodes that do not belong to an earlier layer and that have an edge to a node in layer Lj.

The layer numbers tells us the shortest distance from the start node. BFS is not only determining the nodes that s can reach, it is also computing the shortest path to them. FIFO therefore we use a queue.
Also produces a breadth-first-search tree.

```
BFS(s):
  Set Discovered[s] = true and Discovered[v] = false for all other v
  Initialize L[0] to consist of the single element s
  Set the layer counter i = 0
  Set the current BFS tree T = ∅
  While L[i] is not empty
    Initialize an empty list L[i+1]
    For each node u ∈ L[i]
      Consider each edge (u,v) incident to u
      If Discovered[v] = false then
        Set Discovered[v] = true
        Add edge (u,v) to the tree T


       Add v to the list L[i+1]
     Endif
   Endfor
   Increment the layer counter i by one
 Endwhile
```

The above implementation of the BFS algorithm runs in time $O(m+n)$ (linear in the input size), if the graph is given by the adjacency list representation.

Proof: Note that there are at most n lists L[i] that we need to set up, so this takes $O(n)$ time. Now consider the nodes u on these lists. Each node occurs on at most one list, so the For Loop runs at most n times over all iterations. So the total time is at most $O(n^2)$. To get improved $O(m+n)$ time bound, we need to observe that the For loop processing a node u can take less than $O(n)$ time if u has only a few neighbors. Now the time spent in the For loop considering edges incident to u is 2*(number of edges). We just need an additional $O(n)$ to set up the list so $O(m+n)$.

5.1.2 Depth First Search

```
DFS(s):
  Initialize S to be a stack with one element s
  While S is not empty
    Take a node u from S
    If Explored[u] = false then
       Set Explored[u] = true
       For each edge (u,v) incident to u
         Add v to the stack S
       Endfor
    Endif
  Endwhile
```

DFS probes its way down long paths, potentially getting very far from s, before backing up to try nearer unexplored nodes. LIFO use stack.

The DFS algorithm yields a natural rooted tree T on the component containing s, but the tree will generally have a very different structure.
Rather than having root-to-leaf paths that are as short as possible, they tend to be quite narrow and deep. However, as in the case of BFS we can say something quite strong about the way in which non-tree edges of G must be arranged relative to the edges of a DFS tree T. We can also state that non-tree edges can only connect ancestors of T to descendants.

The running time is $O(m+n)$. We refer to this as linear time, since it takes $O(m+n)$ time simply to read the input.The main step in the algorithm is to add and delete nodes to and from the stack S, which takes O(1) time. Thus, to bound the running time, we need to bound the number of these operations. The total number of nodes added to S is at most 2m. Thus $O(m+n)$.

6 Testing Bipartiteness

If a graph G is bipartite, then it cannot contain an odd cycle. Alternate coloring of nodes between black and red. This coloring procedure is similar to the BFS. Therefore the complexity is the same as BFS $O(m+n)$.

(3.15) *Let G be a connected graph, and let $L_1, L_2, \ldots$ be the layers produced by BFS starting at node s. Then exactly one of the following two things must hold.*

(i) There is no edge of G joining two nodes of the same layer. In this case G is a bipartite graph in which the nodes in even-numbered layers can be colored red, and the nodes in odd-numbered layers can be colored blue.

(ii) There is an edge of G joining two nodes of the same layer. In this case, G contains an odd-length cycle, and so it cannot be bipartite.

Adjacency Matrix is an nxn matrix A where A[u, v] is equal to 1 if the graph contains the edge (u, v) and 0 otherwise. If the graph is undirected, the matrix A is symmetric, with A[u, v] = A[v, u] for all nodes $u, v \in V$. The adjacency matrix representation allow us to check in $O(1)$ time if a given edge (u, v) is present in the graph. There are two disadvantages. The representation takes $\Theta(n^2)$ space when the graph has many fewer edges than n^2. Also checking for incident edges require $\Theta(n)$ time.

Adjacency list works better for sparse graphs that is those with many fewer than n^2 edges. Adjacency list requires $O(m+n)$ space. The total length of the list is $2m = O(m)$.

7 Directed Acyclic Graphs and TopSort

If G has a topological ordering then G is a DAG.

Proof: Suppose by way of contradiction, G has a topological ordering $v_1, v_2, ...n$ and also has a cycle C. Let v_i be the lowest indexed node on C and let v_j be the node on C just before v_i thus (v_j, v_i) is an edge. But by our choice of i, we have $j > i$ which contradicts the assumption that $v_1, v_2...v_n$ was a topological ordering.

In every DAG G, there is a node v with no incoming edges

Proof: Prove by showing that there is a cycle if every node at least one incoming edge.

If G is a DAG, then G has a topological ordering

```
To compute a topological ordering of G:
Find a node v with no incoming edges and order it first
Delete v from G
Recursively compute a topological ordering of G−{v}
  and append this order after v
```

7.1 Time complexity

We note that identifying a node v with no incoming edges, and deleting it from G, can be done in $O(n)$ time. Since the algorithm runs for n iterations, the total running time is $O(n^2)$. If G is very dense containing $\Theta(n^2)$ edges, then it is linear in the size of the input. We may well want something better when the number of edges m is much less than n^2. In such a case, a running time of $O(m + n)$ could be achieved by finding the nodes more efficiently. We keep track of the number of incoming edges for each node w and the set of all active nodes in G that have no incoming edges from other active nodes.

During lecture: Two ways of topo sort. DFS way is O(V+E) and Naive/Greedy way is O(Vlog(V+E)).

8 Min heaps

The root element will be at Arr[0].
Arr[(i-1)/2] - Returns the parent node
Arr[(2*i)+1] - Returns the left child node
Arr[(2*i)+2] - Returns the right child node
getMin() - O(1)
extractMin() - O(log N)
decreaseKey() - O(log N)
insert() - O(log N)

delete() - O(log N)

9 Djikstra

The Dijkstra's algorithm

This algorithm finds the length of a shortest path from vertex a to vertex z in a connected weighted graph. The weight of edge (i,j) is $w(i,j) > 0$ and the label of vertex x is $L(x)$. At termination $L(z)$ is the length of a shortest path from a to z.

L(a)=0
for all vertices $x \neq a$, $L(x) = \infty$
$T =$ all vertices
if $z \in T$
{
choose $v \in T$ with minimal $L(v)$
for each $x \in T - \{v\}$ adjacent to v set $L(x) = \min\{L(x), L(v) + w(x, v)\}$
Set $T = T - \{v\}$
}
else
return $L(z)$.

Proof by contradiction

- Let's assume for the sake of contradiction that there exists a different path traversing the graph that is shorter than the path we picked.
- That would mean that at some point, our algorithm falls behind the optimal approach.
- However, since at every interval we select the lowest cost path forward, at some junction the real lowest cost path must reveal itself to be lower (even if it's the very last node).
- And since at every step we select the lowest cost path, the only way we could have not selected it would be if it never revealed itself to be lowest.
- But there is no way something could be globally lower than our solution and yet never once reveal itself while we traverse the graph. Therefore via contradiction such a case is impossible.

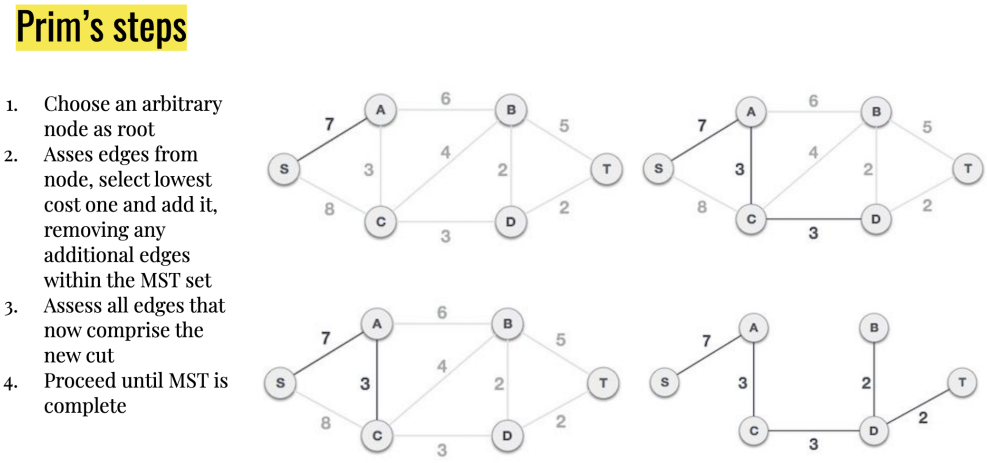
9.1 Time Complexity

O((E+V) Log V)

- We pick the minimum distance node which is an O(log V) operation. Since we are doing this V times we get O(V*Log V).
- DECREASE-KEY operation in the Relaxation procedure takes O(log(V)) time and it's being executed at most E times and we get O(E*Log V)

10 Prim's

At every step of Prim's algorithm, we find a cut (of two sets, one contains the vertices already included in MST and other contains rest of the vertices), pick the minimum weight edge from the cut and move this vertex over to the MST Set.



10.1 Time Complexity

O(ElogN)

- The simplest naive approach is to represent it as an adjacency matrix.

  Here as the cut between the growing MST and non-MST nodes is continually being reassessed, as the MST grows, every MST node must eventually be analyzed against every non-MST node resulting in an $O(N^2)$ run time.

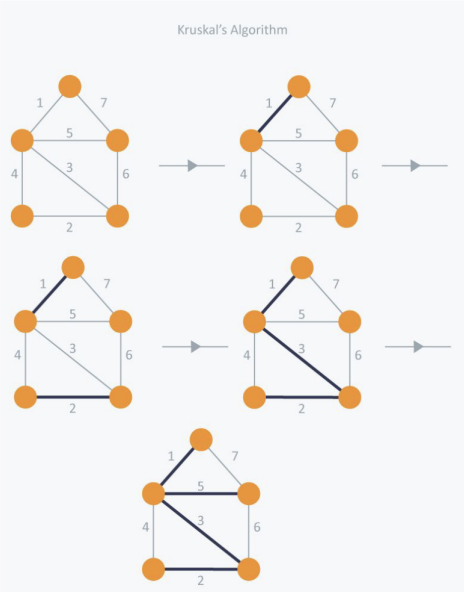
- If the input graph is instead represented using adjacency list, then the time complexity can be reduced to O(ElogN)

  With adjacency list representation, all nodes of a graph can be traversed in O(V+E) time using BFS. A Min Heap is then used to store the vertices not yet included in MST. Since Min Heap can be used as a priority queue to get the minimum weight edge from the cut, it's time complexity for operations like extracting minimum element or decreasing key value is O(LogN). Combining a min heap with an adjacency list yields a total runtime of O(ElogN).

11 Kruksal's

Kruskal's Algorithm (Cont.)

- Specific Algorithm Steps:
  - Sort the graph edges with respect to their weights, lowest to highest.
  - Start adding edges to the MST from the edge with the smallest weight until the edge of the largest weight. How do we know what edge to add or not?
    - **Only add edges which don't form a cycle**, i.e., only include edges which connect disconnected components. (add if no nodes, or one node currently added, but not if both nodes included)
  - When do we stop?
    - Continue until number of edges = N-1
  - Time complexity?
    - In Kruskal's algorithm, the most time consuming operation is sorting because the total complexity of the Disjoint-Set operations will be O(ElogN), which is the overall Time Complexity of the algorithm.



12 Prim's V.S. Kruksal's

Functionally the basic difference is in which edge you choose to add next to the spanning tree in each step.

- In Prim's, you always keep a connected component, starting with a single node.
- In Kruskal's, you do not keep one connected component but a forest. At each stage, you look at the globally smallest edge that does not create a cycle in the current forest.

Prim's algorithm is significantly faster when you've got a really dense graph with many more edges than Nodes. Kruskal performs better in typical situations (sparse graphs) because it uses simpler data structures.

13 Overall Run times

Gale Shapely: $O(n^2)$
Interval Scheduling Greedy $O(nlogn)$
Minimal Lateness $O(nlogn)$
BFS and DFS $O(V+E)$
Topological Sort $O(V+E)$
Djikstra - $O((V+E)logV)$
Prim's - $O(ElogN)$
Bellman Ford - $O(V*E)$ for dense graph $O(V^3)$