CS330: Analyzing Algorithms – Notes

Graphs

1. General Graph Details (Undirected and Direct)
   (a) Two forms of organizing node connections and pathways for a Graph:
      i. Adjacency Matrix: an n by n matrix where n represents the amount of nodes in the graph. The way this graph works is by the following guidelines:
         A. If there is a connection between nodes (that isn't itself), then insert a 1
         B. Otherwise, insert a 0.
      ii. Adjacency List: An m by n array where each n node has a Linked List attached to it of m edges.
   (b) A path in an Undirected Graph is a sequence of nodes connected by edges
      i. A **simple path** is if all nodes within a path are distinct (no node gets traversed over more than once)
      ii. A **connected path** is if all nodes within a graph have a path
      iii. Inversely, a **disjoint** graph is if there are existing nodes not connected to other nodes
   (c) A **cycle** is a looping of nodes in a pathway, that is the first node may also be the last node in a pathway.
   (d) A **tree** is a graph that is connected, has no cycles, and always has $n - 1$ edges (because the root node has no edges corresponding to itself). Trees may be directed or undirected.
   (e) Generally speaking, **connectivity** refers to a pathway or direct connection between nodes. A graph's overall connectivity is determined on whether each node is connected in some manner.
      i. This concept is especially useful when considering strategies for searching a graph (Breadth First Search, Depth First Search, and so on)
      ii. Proof that if a graph is connected, then BFS hits all nodes:
         A. Assume the contradiction: A node was not hit during BFS, but the Graph is still a connected graph
         B. However BFS constitutes that it cannot search another depth down in a graph until all nodes at a certain level have been identified.
         C. Thus, if BFS completed, then all possible connected nodes must have been hit

January 31st

1. 3Sum pseudocode:

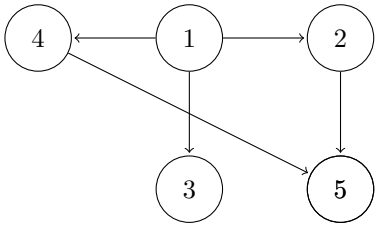
```
A = array of numbers
for i in range(0, n−1):
   for j in range(i+1, n−1):
      difference = ~(A[i] + A[j])
  if(lookup(A, difference) == 1 && A[i], A[j], difference are all unique):
      return A[i], A[j], difference
```

1. Breadth First Search pseudocode:

```
For all nodes v: L[v] <− null
level = 0, L[s] = 0
Q.enqueue(s)
while(!isEmpty(Q)):
  v = Q.dequeue()
  level = L[v]
  for each u (adjacent to v):
    if (L[u] == null):
    L[u] = level+1
          Q.enqueue(u)
```

1. 3Sum Notes:
   (a) In order to find a series of three numbers in a list that sums to 0, first begin by obtaining two unique numbers from the list (A[i] and A[j]) and equating their sum.
   (b) Then take the complement (that is, the negative/inverse of this sum of two numbers), that would result in a total of 0.
   (c) For example: if 2sum = a + b, and 2sum + c = 0, then c is the complement of a+b. Or another way of thinking: c = (a + b)
   (d) Also, sorting the list helps with this, as i and j can both be examined uniquely, and a sort function would only be an **additive property to the overall time complexity**.
      i. As an aside: The **overall time complexity** to be observed in an algorithm is the total of all time complexities.
      ii. An **additive property** is just a time complexity that is added to the overall time complexity
      iii. A **multiplicative property** is a time complexity that is multiplied to the overall time complexity
      iv. The **observed worst-case time** of an algorithm is principled on the highest/worst complexity available
      v. For example: If an algorithm is of overall time ($n^3 + n * log(n) + n$), then:
         A. The **upper bound time complexity**, or worst-case time, is $O(n^3)$.
         B. n^3, $n * log(n)$, and n are all additive properties to one another
         C. $n * log(n)$ is a multiplicative property of n and $log(n)$
   (e) It would be best in this algorithm's implementation to use a Hash Table for $O(1)$ time in both insertion and lookup.
   (f) The **lower bound time complexity**, or the best-case time (denoted as $\Omega()$) is yet to be found

2. Breadth First Search Notes:
   (a) First off, checking non-integers arrays would require usage of an associative array in which there is some attachment or priority for the values being passed, much like the process of a Hash in a Hash table.
   (b) A **queue** should be initialized in order to process each **level** in a graph before going further down through the FIFO ordering (First-in, First-out)
      i. A **level** is a denotation of how many edges into the graph a search has gone through. That is to say, if a search begins at a certain node, then every node the search traverses **downwards from this starting node** can be considered to be at another level.
      ii. The **depth** is considered the count of how many "levels" deep the search has traversed.
      iii. Both these terms can be used as generic, high-level interpretations of how to analyze searching and ortienting a graph or tree
   (c) The queue should have the standard functions at the very least. These include isEmpty?, Enqueue(node), Dequeue().
   (d) Example:
      i. Consider when we start at Node 1 as the root. The depth at that point would be 0.
      ii. Then consider the two Nodes of 2 and 4. They would be at depth 1. But so also would Node 3.
      iii. And then finally, Node 5 would be at depth 2 because it is an extension of Nodes 4 and 2.



3. Queue Example with this graph:

(a) First Level: [Lvl 0. | 1 - - - -]

(b) Next level: [Lvl 1. | - 2 3 4 -]

(c) Final level: [Lvl 2. | - - - - 5]

4. And this is the Array Representation of this Queue/Graph leveling: [0, 1, 1, 1, 2]

Important Graph Info

1. Further Graph Info:

   (a) Graph G = (V, E), $|V| = n$, $|E| = m$

Problem	Adj. Matrix	vs Adj. List
Space	$O(n^2)$	$O(n+m)$
$(U, V) \in E$	$O(1)$	$O(degree(u))$
Iterate Over (u, *)	$O(n)$	$O(degree(u))$
BFS	$O(n^2)$	$O(\sum_u degree(u))$

   (b) Terminology:

      i. **Sparse** represents a graph with a lack of nodes and connections between them. Sparse graphs are usually best examined in $O(n+m)$ time.

      ii. **Dense** represents a graph with a large amount of nodes and connections between them. Dense graphs are usually best examined in $O(n^2)$.

   (c) The smallest possible m (amount of edges) for a graph is 0. (A single node graph).

   (d) The largest possible m is $n * (n-1)/2$, where each node can have 2 edges at the sense of $\binom{n}{2}$

   (e) A (U, V) is a graph node connection.

   (f) (u, *) represents a "connection between all nodes for u"

   (g) Simplified explanation of Breadth First Search: Iterate over (u, *) for all Nodes v.

      i. Best to think of it as a summation of all the dgrees of u in the size of m twice, meaning you have to go through all n nodes (the vertices), then through m edges afterwards.

   (h) $O(\sum_u degree(u)) = O(n+m)$ essentially.

1. Simplified Lecture Notes:

   (a) **Connected components**: refers to all nodes that are reachable from a given node / vertex.

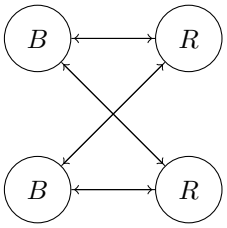
   (b) A **Bipartite Graph** is an undirected graph such that all edges can connect to all nodes that are of disjoint sets.

      i. For example:

      ii. Consider a four node graph, where two nodes are blue and two nodes are red.

      iii. If there are existing undirected edges that connect to all nodes and...

      iv. if each "end" of an edge is a different color, then the graph is Bipartite

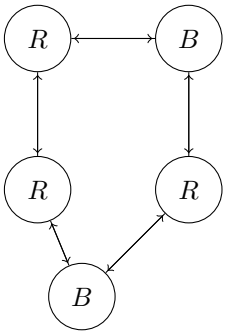


      i. Problems are generally easier to solve when the graph is bipartite.

ii. Must re-iterate, a **bipartite graph** is a graph where nodes can be separated into disjoint sets, and all edges distinctly point to disjoint nodes.

iii. Proof of that no bipartite graph can contain an odd length cycle:

A. The edges must alternate between nodes of different colors, but with an odd length of edges that is impossible, there will always result a link between nodes of like color.



(c) Additionally, if a Graph is Bipartite, the following principles hold:

i. No edge of Graph G can join 2 nodes of different layers or **levels** –> Bipartite

ii. If an edge of Graph G joins two nodes of the same layer, then that would result in an Odd Length Cycle

(d) A BFS will determine the bipartiteness of a graph, because there factually cannot be an edge in the same layer between nodes.

(e) There are K edges out from a vertex node that cycle back (which is always even).

(f) K+1 edges, (such as if they are between nodes in the same layer) result in an odd length.

(g) Point is: [Edge in the same layer] -> [Odd length cycle] -> [Not a Bipartite graph]

(h) A BFS must have single connections between nodes (i.e a graph can't be squared or have edges spanning more than 1 node)

2. Directed Graphs:

(a) Connectivity differs in the sence of **Strong Connectivity**

(b) **Strong connectivity**: Mutual reachability between nodes –> Every node can reach each other.

(c) A graph can be determined strongly in $O(m+n)$ time.

i. Pick a node

ii. Do BFS in Graph G on Node S

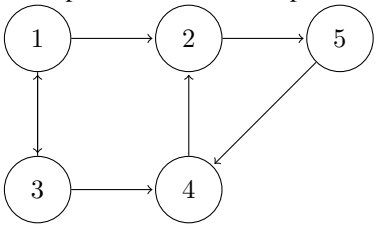
iii. Do Bfs on the Reverse of Graph G on Node S

iv. **Iff all nodes reached in both BFS, then the graph is strongly connected**

February 2nd - Continuation of Graphs

1. More Directed Graphs:

(a) Example of a Directed Graph that is **not** strongly connected:



(b) All nodes in the graph **cannot** be examined from any arbitrary starting node

(c) An **adjacency list** interpretation of this graph:

Node	Edges	
1	-> 2	-> 3
2	-> 5	
3	-> 4	-> 1
4	-> 2	
5	-> 4	

(d) And a **reverse adjacency list** is the following:

Node	Edges	
1	-> 3	
2	-> 4	-> 1
3	-> 1	
4	-> 5	-> 3
5	-> 2	

(e) Reverse Adjacency List Process:

i. Firstly, when a node's linked list is empty, simply insert the inverse of each node

ii. When a node's list is **not** empty, insert the "next" node **first**, in front of the other existing nodes in the list

iii. Carefully study how nodes 2 and 4 are in two node's lists in the original Adjacency list. The inverse means those two nodes are now in their lists.

2. **Directed Acyclic Graph**

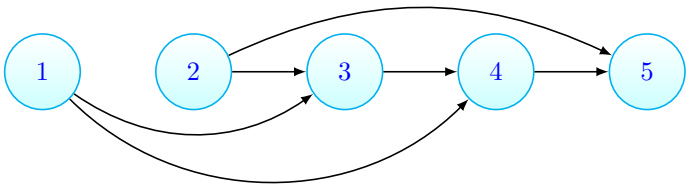
(a) A directed graph with no directed cycles is called an **directed acyclic graph**. This follows the constraint that $Edge(V_i, V_j)$ must have Node V_i precede Node V_j.

3. **Topological Order**

(a) **Topological ordering** of a directed Graph G = (V, E) is an ordering of nodes $(1...n)$ where $i < j$ for every Node V_i followed by Node V_j.

(b) Basically, every node must be followed by a node "greater than it" in the ordering. Then just connect the edges according to how they were connected in the original graph.

(c) Example with Nodes going Left -> Right:



(d) Usually, one topological ordering is sufficient, even if there may be more than one way to do so.

(e) **Important:** If there exists a cycle in the graph, a Topological Ordering is **not possible**.

(f) Therefore, there are three principles that are intertwined:

i. If there is a Topological Ordering, then the Graph is a DAG.

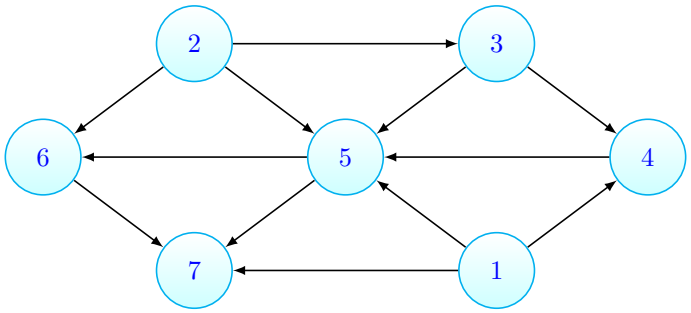
A. Proof by Contradiction: There must be a cycle if it isn't a DAG. But you can't topologlically order a graph if there is a cycle. Boom roasted.

ii. If the Graph G is a DAG, then there is a node with no incoming edges (A "first/root" node)

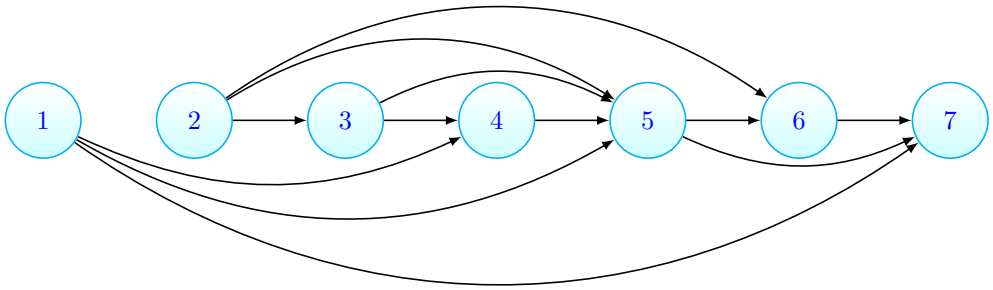
A. Proof by Contradiction: All edges must have an incoming edge. But if that were true, you could trace back indefinitely in a loop. Aka, a cycle. And DAG's can't have those.

iii. Lastly, if G is a DAG, on i vertices, it has a topological ordering. The proof of this I will cover in a following section after an example of a Topological DAG.

iv. Firstly, another example of a DAG:



v. And now a Topological Ordering of this DAG:



(g) And now I will prove that third Lemma by introduction of the 5 Step Proof process.

(h) Also, as quick aside: $\forall(x)$ means "For all existing x" and $\exists(x)$ means "There exists a particular case of x"

4. **5 Step Proof Process**:

   (a) $\forall(i)$ Prove each i (Referenced as $P(i)$) by Induction

   (b) Provide a Base Case (For example, a 1 node graph)

   (c) Assume $P(i)$ is true $\forall(i \leq j)$. Prove $P(j+1)$ induction.

   (d) Provide a proof implementation (i.e put your proof into words and mathematical principles).

   (e) Apply Inductive Hypothesis in Proof (Noted as G' in the following example).

5. And now to exemplify this, here is the proof of the third principle of DAGs and Topological Ordering mentioned above:

   (a) We have a Graph G that has $j+1$ vertices and is a DAG.

   (b) We know via the lemmas proven that: there must $\exists(v)$ where v is a node that has no incoming edges for a topological ordering.

   (c) Inductive Hypothesis: Create a Graph G' such that it has j nodes (one less than the current G) and is a DAG with a topological ordering.

   (d) Now **prepend** another node, which would be $j+1$, to this graph. Well...

   (e) Since we already have a node with no incoming edges, if we **prepend** the new node to the current graph, we will just put another node at the beginning of the topological order with no incoming edges.

   (f) Therefore, this graph G with the newly prepended $j+1$ node will be a topologically ordered Graph.

   (g) And if you keep doing this via the Inductive Hypothesis, then we have successfully shown that any additional node will just further satisfy the Graph G as a topological ordering.

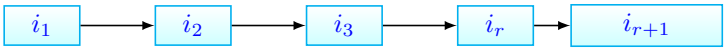
   (h) Hurray.

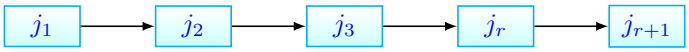
6. As a practical note, it is **necessary** to formalize all of these items in both your homeworks and your tests. You will be marked down otherwise.

7. Also, in the proof above, G' was a DAG because we assumed G was a DAG and were only able to remove nodes with outbound edges. It is vital that assumptions like this are clarified in proofs.
8. Algorithmic time complexity of Topological Order: $O(m+n)$
9. Algorithm for Topological Order:
    (a) Need a count for the remaining edges
    (b) Also intialize a set with node that do not have incoming edges
    (c) Begin by scanning the graph once ($O(n+m)$)
    (d) Then begin updating: Delete a Node v, decrement the count for all of Node v's edges, and add the endpoint nodes when the count hits 0 to the set
    (e) All of this takes $O(1)$ time per edge.
10. And that about finishes this portion of Graph information

Greedy™ Algorithms

1. **Greedy™ Algorithms** are understood as Algorithms that take the best first/immediate solution to a problem as they arise. There ain't much goin' ons up in the ol' noodle for these.
2. For example, the **Job Problem**: Need as many jobs to be filled as possible while taking as much time out a schedule as possible.
    (a) Jobs must be mutually compatible, there can be no overlap between jobs in a schedule.
3. The Greedy™ Approach:
    (a) Take the jobs as they come naturally, so long as they are sequentially compatible to one another (i.e. pick one job, and then when it's finished, pick the next available job)
    (b) There are four ways of thinking about this:
        i. Earliest Start Time
        ii. Earliest Finish Time
        iii. Shortest Intervals
        iv. Fewest conflicts
    (c) When searching for a counterrexample, always take the shortest/smallest possible.
    (d) Out of the 4 ways of analyzing this algorithm, only one works: Earliest Finish Time.
    (e) So the Algorithm is as follows:
        i. Sort the jobs in the schedule by finish time ($O(n*log(n))$)
        ii. Then parse through each job in the schedule, finding the ones that are most compatible according to finish time. Example: A job has a start time of s_i and a finish time of s_i. The next job has a start time of s_j and a finish time of f_j. To find compatibility, just check if s_j of Job 2 comes before f_i of Job 1.
        iii. Overall time complexity: $O(n*log(n))$
4. The proof of this algorithm is extremely important (and very relevant to a lot we'll cover with Greedy™ Algorithms). It is – as Professor Byers put it – **Subtle, Short, & Nuanced**. How poetic.
    (a) Proof By Contradiction: Suppose that the current Greedy™ Algorithm we've devised is **not the optimal solution**
        i. Let i, ranging from $(1...k)$, be a set of jobs devised by our Greedy™ approach.
        ii. Les j, ranging from $(1...m)$, be a set of jobs from the supposed *true optimal solution*
        iii. Let the association between i and j be as such: $i_1 = j_1$, $i_2 = j_2$, ... $i_r = j_r$, where r represents the furthest possible point i and j are identical sets.

A. Greedy™ : 

B. Optimal™: 

iv. To put it bluntly, because j_{r+1} has a differing start/finish time, that is suppose to be where the optimality comes in.

v. Yet, substituting in i_{r+1}, which generally has an earlier start time and a later finish time, would be just as optimal.

vi. So therefore, there is essentially no better/more optimal way of beating out the Greedy™ solution.

(b) This proof can be generalized to the following saying that applies to most Greedy™ Algorithms:

"Greedy Stays Ahead"

(c) Also, this Job Problem can be found under the umbrella of a series of Greedy™ Problems called **Interval Scheduling Problems**.

5. On that note, going further into Greedy™ problems, we approach the **Interval Partitioning** problems:

(a) Given a set of classes and a set of classrooms, find a schedule where no two classes occur in the same room simultaneously, and use the fewest amount of rooms possible.

(b) **Important Terminology Alert**: **Depth** in this case means the maximum number of sets of open intervals in the schedule at any given time. (i.e. a point in the schedule where there's nothing occurring)

(c) If the **number of classrooms used** matches the **depth**, then you have found the optimal solution.

(d) The Greedy™ Approach:

i. Consider lectures in order of increasing starting times.

ii. Assign each class to a compatible classroom. If a class ends before the next starting class time, re-use this classroom.

iii. If there is a conflict of existing classrooms (i.e the next class starts while the other class is still in session in that classroom), then assign a new classroom to this new class.

iv. Time complexity for this: $O(n * log(n))$

v. Implementation:

```
Sort by starting time in a priority queue (earliest to latest)
for lecture in range(1, n):
  if(room & lecture are compatible):
    assign lecture to classroom
  else
    get a new room
    put lecture in this new room
    old_room —> new_room
```

vi. Sort time: $O(n * log(n))$

vii. For loop: $O(n)$

viii. Class assignment where $f_i \leq s_j$: $O(c)$ <– Some constant factor

ix. Observe that this algorithm never overlaps classes.

x. Proof:

A. Let d equal total number of classrooms.

B. Let the first class, represented as $j - 1$ be starting in room 1 which is $d - 1$.

C. Then let the final class be starting at j in the final room d.

D. Then suppose that $j + \varepsilon$ time passes, where ε represents a small, near negligible, value but still shows that $j + \varepsilon > j$.

E. Then that means all possible classrooms are being used simultaneously by all possible classes.

F. So the **key note** here is that all schedules will use $\geq d$ classrooms.

6. The next Greedy™ problem is the **Minimizing Latness** problem:

(a) Goal: Schedule all jobs to minimize maximum lateness, where $L = max l_j$

(b) Job j requires t_j time and is due at d_j hour. It also starts at s_j and finishes at: $f_j = s_j + t_j$

(c) **Lateness** is denoted as: $l_j = max\{0, f_j - d_j\}$

(d) Possible ways of solving:

i. Earliest deadline first

ii. Smallest time first, unless a deadline is approaching, then that immediate job takes priority.

iii. Shortest processing time

iv. Smallest Slack (Slack = Least amount of worry about deadline approaching + time it takes to finish)

(e) Conclusion: Earliest Deadline First is the best approach (go figure)

(f) Implementation:

```
Sort by deadline time intervals
for job in range(1, n)
  Assign job to a time
  Compute start time and finish time
  Then go from the next time interval
Output the listing
```

(g) Overall Time Complexity: $O(n)$

(h) No idle constant time or anything, just a straightforward way of doing things.

(i) New proof technique for this: ***Exchange Argument***

i. Invert a Greedy™ devised schule S such that a pair of jobs swap placement and time. (Such that $i < j$ yet j is scheduled before i)

ii. There may be other inversions also, but this case should show for all.

iii. Claim: There is no lateness change

A. First, remember that *Lateness = Finish time - Deadline time*

B. l represents the lateness before the swap, and l' represents afterwards

C. We know that $l_i \leq l'_i$

D. Then $l'_j = f'_j - d_j$. However, f'_j is just the swapped finish time of i, so it's really –> $l'_j = f_i - d_j$

E. HOWEVER, we know that j must be scheduled at a later deadline, so it's difference between finish and deadline time is less than that of i

F. So, $l'_j \leq f_i - d_j$

G. And $f_i - d_j$ is literally just l_i, so $l'_j \leq l_i$, proving that there was no loss in lateness from the inversion of schedules

(j) In short, the latness of this new swap of Job j will note be greater than the latness of the original Job i in its place.

(k) This lateness proof lends to the overall proof of the Algorithm:

i. Assume the Greedy™ algorithm is S and the Optimal solution is $S*$ with the fewest inversions possible.

ii. But we just proved inversions make no difference for the original algorith. So the effect of inversions results to 0.

iii. However, if this is the case, then that contradicts that this $S*$ has the fewest inversions. Which leads to the fact the optimal solution is a schedule without inversions.

iv. Which is just S

7. Quick Recap on Greedy™ Algorithm Proof techniques:

(a) "Greedy Stays Ahead"

(b) Structural flaws -> Your algorithm can be shown to be optimal anyways, like the example above

(c) Exchange Argument

8. Finally, we arrive at the **Shortest Path Problem**:

   (a) Goal: Find the shortest route from Source S to Destination T

   (b) Quick Response: Assign values to all pathways, and find the smallest (read: best) summable path.

   (c) **Dijkstra's Algorithm**

      i. Maintain a set of nodes with the shortest path (Hint: Use a priorty queue)

      ii. Begin a pre-processing analyzation of the graph with each outgoing edge from a node being assigned a value. These "values" are much like something as how long it takes to drive down a road.

      iii. Assume the distance to each node is infinitely long at first, except at the source node.

      iv. Then, as each node is explored, analyze outgoing edge values and determine which edge value is the smallest. The "travel time" from each node is thus the sum of edges from the starting node to the current node traveled.

      v. Each explored node should be inserted into a set S of explored nodes.

      vi. Smallest Distance: $\pi(v) = SD(u) + l_e$

         A. $SD(u)$ = Smallest Distance from some node u

         B. l_e = Total sum of edge weights thus far accumulated via the shortest path

         C. $\pi(v)$ = The next possible shortest distance path in the graph where $\pi(v) <$ all other edge weights of v

      vii. After this calculation has been made, make sure the current node is updated to the new $\pi(v)$ node.

Continuation of Greedy^{TM} Algorithms

1. **Proof of Dijkstra's Algorithm:**

   (a) Claim: When a Node u is added to the set of explored nodes, S, then it is true that the distance of Node $u = \pi(u)$, which is the shortest path.

      i. Base Case: $|S|$ = 1 (meaning there's only 1 node in the graph). If $S = \{s\}$, then distance of the total edges in S is: $d(s) = 0$.

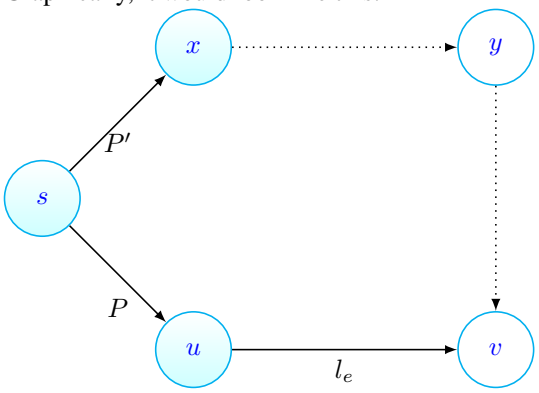
      ii. Inductive Hypothesis: Therefore, assume that is true that $|S| = K \geq 1$, where K is the arbitrary value denoting new nodes added to the set from its current state. Then if we insert a new Node v into set S, it should follow that: $\pi(v) = l_e + d(u)$; $\pi(v) == d(v)$.

      iii. So assume the contradiction:

         A. Suppose that there is a P' such that $P' \neq P + U_e$, where P would be the existing best short path and U_e is the weight of Node u's edges.

         B. Then you would have to suppose there exists some link from a node x to a node y such that it follows from pathway P' to the node v.

         C. Graphically, it would look like this:



         D. The nodes shaded in blue are explored nodes that belong to set S. The white nodes are the unexplored nodes that can be reached in the graph.

E. What this graph basically indicates is an "alternate path" to this node v that is proposed to be better than Node u through pathway P.

F. Mathematically, the contradiction can be viewed this way:

$$l(P) \geq l'(P') + l(x,y) \geq d(x) + l(x,y) \geq \pi(y) \geq \pi(v)$$

G. The first $\geq$ refers to the non-negative weights of the Pathways of P and P'. The second one refers to the Inductive Hypothesis we proposed, where all pathways with +1 greater edges are greater than the current node's pathways. The third refers to the definitive/calculable shortest distance to Node y. And the final $\geq$ refers to the pathway that Dijkstra's algorithm calculated was the shortest pathway to Node v.

iv. Basically, what all this mean is that if Dijkstra's algorithm found a pathway to v that was appreciatively smaller than all other pathways, then this contradicatory pathway P' cannot possibly serve as a better alternative route. This is predicated on the fact that the next shortest path to v from some pathway with node x must go through some node y, when we could already reach v directly from u.

2. Implementation:

   (a) For each unexplored node...

      i. Continually update the $\pi(v)$ minimum function as you explore.

      ii. Pretty easy, no?

   (b) Essentially:
   $\pi(new)$ = **IF** edge (v,u) is in Edge Set E **THEN** $d(u) + l(u,v)$ **ELSE** $\pi_{old}(v)$

3. Efficient Implementation:

   (a) Maintain a Priority Queue of unexplored nodes that are prioritized based on their shortest distance: $\pi(v)$.

   (b) Designate operations per search of a node with PQ.minimum() and PQ.updateKey($\pi_{old}(v), \pi_{new}(v)$).

   (c) Whenever we update nodes, we have m edge examinations.

   (d) So with a binary heap: $m * log(n)$ operations per.

      i. Initialize Set: $O(n)$ – Preprocess

      ii. Initialize PQ: $O(n * log(n))$ – Preprocess

      iii. for i = 1...n:

         A. PQ.min: $O(n * log(n))$ – Additive

         B. PQ.updK: $O(m * log(n))$ – Additive

   (e) Thus, the overall time complexity is $O(m * log(n))$

4. Quick Review on Trees:

   (a) A graph G= (V,E) is a tree iff;

      i. It is connected

      ii. Does not contain any cycles.

   (b) Things we know about trees:

      i. The number of edges is always 1 less than the number of nodes ($m = n - 1$)

      ii. A tree has at least 2 leaves.

   (c) A graph is a DAG if it shares the same two properties: It is connected and does not contain any cycles. Therefore, there is an existing relationship between a DAG and a Tree.

5. **Minimum Spanning Trees**

   (a) Definition: Given a Graph G, an MST is just a subset of nodes such that it is the smallest tree inside of G.

(b) Incredibly important Data Structure with a multitude of applications.

(c) Formal understanding: Given a graph G = (V,E) with valued weights c_e, an MST is a subset where $T \subseteq E$ such that T is a tree whose sum is minimized.

(d) Algorithm Strategizing:

i. Reverse-Delete Theorem: Start with all edges and delete from the heaviest weight trending down.

ii. Prim's Algorithm: Start with a root node, then greedily choose the best path based on total weight, much like Dijkstra's.

iii. Kruskal's Theorem: Best to examine edges by increasing weight per node, $l_{e_1} \leq l_{e_2} \leq ... \leq l_{e_n}$

(e) All three of these Algorithms work. However, Reverse-Delete is somewhat ineficient, so we'll only truly worry about Kruskal and Prim's.

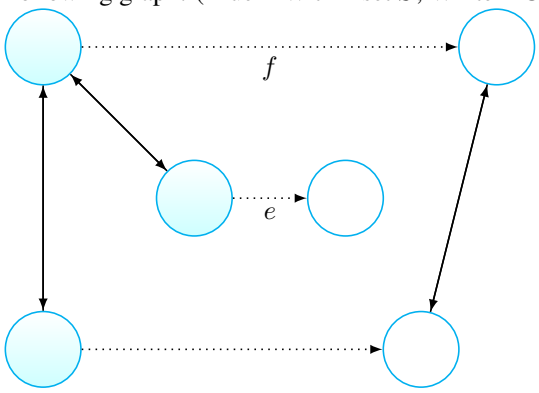
(f) Very Important Notes:

i. Simplified Assumption: Edge weights are all distinct (c_e are all different).

ii. **Cut Property**: A set S is a subset of nodes, edge e is the minimum cost edge with exactly 1 endpoint S. Then the MST must contain this edge e.

iii. **Cycle Property**: A cycle C is any cycle in the Graph, let edge f be the maximum edge cost in the Cycle C. Then the MST must **not** contain this edge f.

iv. Following graph: (Blue = Within set S, White = Unexplored nodes).



v. Cycle-Cut Intersection -> There is a cross between both the Cycle Property and the Cut Property within a graph. The edges of the graph will always be even in this case.

vi. A cycle cannot cross a cutset only once. It is impossible, as in order for a cycle to be complete, the edges have to eventually circle back to a certain node. So, there's a guarantee that parsing through a cycle will cross the cutset **at least twice**.

vii. **Proof of Cut and Cycle Properties**: (Use an Exchange Argument)

A. Suppose e is not in the MST Tree T.

B. Then let this correpsonding e create a cycle in T.

C. Therefore, e is in the cycle C and in the Cutset D corresponding to the explored Set S. Let c_e, the edge weight for edge e, be less than c_f, the edge weight for another edge f that apparently holds both the properties that e holds.

D. So both e and f are in the Cutset D and the Cycle C. And this is critical: these two existing in the Tree T simultaneously are what generate the complete cycle.

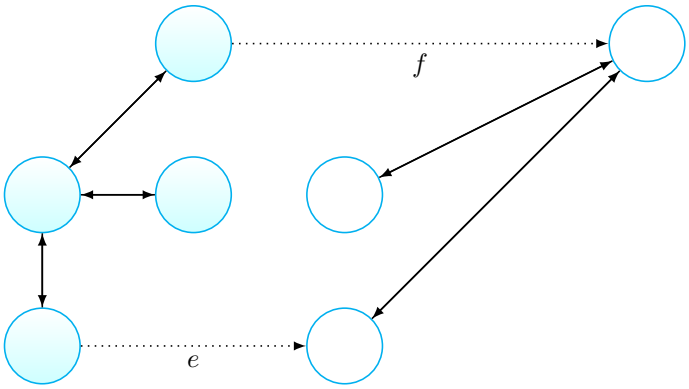
E. T' = $T \cup \{e\} - \{f\}$ is also an Minimum Spanning Tree. This Tree T' holds all over the properties of that original Tree T, without edge f.

F. But since $c_e > c_f$, then $T' < T$ in cost.

G. Yet this does not make sense, since T is supposed to be the Minimum Spanning Tree that was calculated to have the least weight.

H. So, T' is actually the real T that includes e but not f and removes the cycle and weighs the least possible.

I. And just to recall, both T and T' are trees that are **completely connected and with** $m = n-1$ **edges**.



viii. This proof works both for the Cycle property and the Cut property. The difference is between "supposing e is not in T" versus "supposing f belongs to T'" and then just coming to the conclusion that (because both e and f must be in the non-optimal tree and that $c_e < c_f$) removing f and/or injecting e into the Tree T/T' is what would be the best possible MST.

ix. In Summary:

A. f and e form a cycle within MST Tree T. Both of these edges are inclusive to the Cycle C and the Cutset D.

B. $c_e < c_f$

C. Remove c_f to remove the cycle from T

D. Therefore, T must have e.

February 16th - Continuation of Greedy™ Algorithms

1. **More on Minimum Spanning Trees**:

(a) **Dijkstra's**: Finding the shortest path in a tree from a source node s to a destination node t.

(b) **Minimum Spanning Tree**: Finding a Tree within a graph that minimize the amount of edges yet still maintains connection across all nodes.

(c) In a sense, Dijkstra's is more about starting at a certain node, and then slowly branching out in the shortest path possible until a node is found. Not all edges need to be searched, nor does the path have to connect all nodes. And these points differ for MSTs.

(d) Note the difference, it could save your life! (maybe idk)

(e) Also, here lies a distinction between the two halves of the Greedy™ Algorithms Chapter:

i. The first half is mostly about introducing elementary problems to provide concrete examples of Greedy™ Algorithm proofs.

ii. The second half is mostly about introducing critically important and extremely common Greedy Algorithms (Dijkstra's, MSTs, Huffman Codes (To be seen)), their implementations, and specific properties of their proofs that are generally applicable eslewhere.

(f) More In-Depth look at the MST Algorithms:

(g) Reverse-Delete:

i. Step 1: Repeatedly find a cycle on Graph G ($O(m+n)$)

ii. Step 2: Delete the heaviest edge ($O(n)$)

iii. Step 3: Repeat this loop ($O(m-n-1)$ –> $O(m)$)

iv. Overall Time Complexity: $O(m^2)$

v. Small Aside: $O(n) \leq O(m) \leq O(n^2)$ –> So take $O(m)$ as the upper bound.

vi. This algorithm is **not efficient**, and therefore not an algorithm you should focus on using in this class.

(h) Prim's Algorithm:

i. Step 1: Initialize a set S. Begin at any arbitrary node.

ii. Step 2: Apply the cut property when searching for the shortest edge.
iii. Step 3: Add the minimum cost edge that was found from this to a Cutset Y corresponding to set S into a Tree T and add one new explored node u to the set S.
iv. This is an efficient Algorithm that makes use of the Cut Property.
v. $n-1$ iterations, for m edges.
vi. Implementation:
A. Startup a Priority Queue
B. Maintain a set of explored nodes.
C. Maintain cheapest edge costs to a node in a set.
D. This is $O(n^2)$ with an array, $O(m*log(n))$ with a binary heap.
E. Pseudocode:

```
Prim(G, c):
  foreach (node v in the Graph of all nodes V) a[v]  <--  Set to Infinity
  Initialize an empty priority queue Q
  foreach (node v in the Graph of all nodes V) Insert v onto Q
  Initialize set of explored nodes S <-- Empty Set

  while (Q is not empty) {
    u <-- delete min element from Q
    S <-- The Union of S and the node u
    foreach (edge e = (u, v) incident to u)
      if ((v is not within the set S) and (weight of edge e < a[v]))
      decrease priority a[v]
```

vii. Side Note: Priority Queue libraries for most programming languages do not support decreasing specific priorities in the PQ. So instead, just "insert both" nodes into the set and make a conditional series of statements to check whether a node forms a cycle before deleting it from the PQ and poppping the min.

(i) Kruskal's Algorithm:
i. Step 1: Sort edge weight in ascending order
ii. Step 2: Build a set of Edge weights T.
iii. Step 3:
A. Proposition 1: If the node completes a cycle, discard it
B. Proposition 2: Otherwise, put the edge in following the Cut Property
iv. So, either the node completes a cycle or it is within a cutset.
v. This algorithm is roughly $O(m*log(n))$
vi. Sorting weights technically takes $m*log(m)$ but –> $log(m) \leq log(n^2) == 2*log(n)$, therefore, just read it as $O(m*log(n))$.
vii. For this algorithm, you will have to use a very special data structure called a **Union-Find**. It has its own branch of study and many programming languages have functions or some smaller libraries desginated for it.
viii. Some complexities for the Union-Find are so well crafted, they hover on $O(1)$ time in most cases.
ix. Pseudocode:

```
Kruskal(G, c):
  Sort edges weights so that c1 < c2 < ... < cm.
  T <-- Empty Set of Edges
  foreach (node u in the Graph of all nodes V) make a set containing singleton u
  for i = 1 to m
    (u,v) = edge ei
    if (u and v are in different sets)
      T <-- The Union of T and edge
       merge the sets containing u and v
  return T
```

(j) Class logistics note: Due to the snow day messing things up, we are **not** responsible for Clustering.

2. Huffman Codes

(a) **Question 1:** *Given a text of 32 symbols (26 unique letters, a space character, and some punctuation characters), how can we encode this text in bits?*

i. We can encode 25 different symbols using a fixed length of 5 bits per symbol. This is called **fixed length encoding**.

(b) **Question 2:** *Some symbols (e, t, a, o, i, n) are used far more often than others. How can we use this to reduce our encoding?*

i. Encode these characters with fewer bits, and the others with more bits.

(c) **Question 3:** *How do we know when the next symbol begins?*

i. Use a separation symbol (like the pause in Morse), or make sure that there is **no** ambiguity by ensuring that **no code is a prefix of another one**.

(d) In truth, you will usually use up to 8 bits for encoding of text, mostly due to odd ASCII control sequences and character fixtures. But for the 32 symbols only, you can keep it to 5 bits ($2^5 = 32$).

(e) **Definition**: Prefix code for a set that matches 0s and 1s such that no two encodings hold a connected bit.

(f) You can also suppose that certain letters (e, t, a, i, etc) occur more often and can make the bit sequences for them the shorter of the many. Supposition of frequencies means that the amount of bits used for a lesser frequent item is a waste. Therefore...

(g) **Optimal Prefix Codes Definition:** The average bits per letter of a prefix code c is the sum over all symbols of its frequency times the number of bits of its encoding: $ABL(c) = \sum_{s \in S} f_x * |c(x)|$

February 23rd Notes

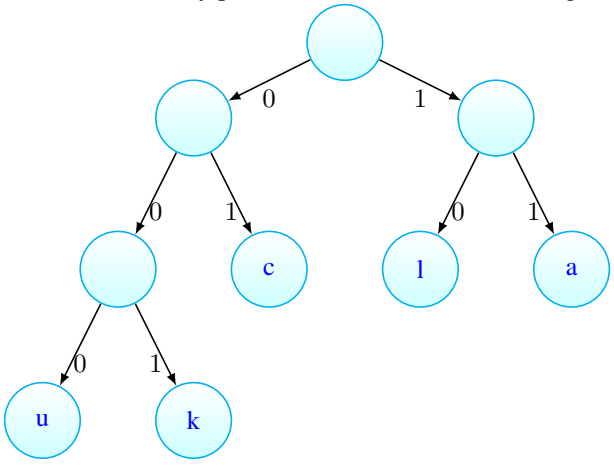
1. Continuation of Huffman Codes

(a) Main Idea: A person wishes to send messages across the electric wave, and thus need to designate the smallest yet most reasonable fixedamount of bits for letters and text.

(b) As discussed before, you can use about 5 bits to encapsulate all 32 bits that are the baseline for the alphabet. But this method is a bit of a waste when there are other, far moreoptimized strategies that won't waste an entire 5 bits for every letter.

(c) This is the point of a Prefix code: Rather than give every letter a 5 bit representation, just span a certain amount of bits over a certain amount of letters where they cannot overrepresent each other as pre-fixes. More simply, no coding of bits is a prefix of another word.

(d) Say for example, you are given the text *"akl"* and its bit representation is **1100110**. A tree orientation of these elementary pre-fix codes would be something like this.



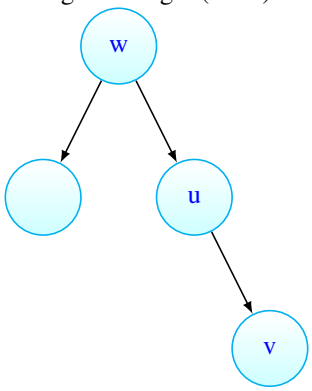
(e) So this graph shows the following pre-fix orientation for akl: a (11) | k (001) | l (10)

(f) But an even more measured approach would be to take the average bits per letter

(g) Quick Tree aside: A tree is **full** if every node that is not a leaf has two children.

(h) **Optimal Prefix Codes**

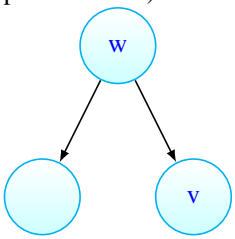
i. The average bits per letter of a prefix code c is the sum over all symbols and their corresponding frequencies in their message times the number of bits in the encoding. This is optimal if the average bit length (ABL) is minimized.



ii. The procedure of minimized the code comes to two scenarios:

A. 1. u is the root node, so remove the root u and move v to the root.

B. 2. u is an inner node, so remove u and put v into this position instead (More relatable to the picture above)



iii. But where in a tree should the higher frequency letters go?

iv. Near the top?

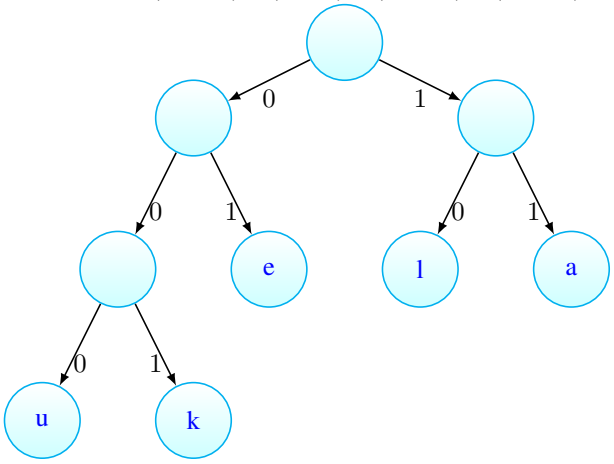
A. The Greedy^{TM} Approach: Make a top down tree, where the set S of values corresponding to letter frequencies is split into subsets of S_1 and S_2 where both subsets have almost equal frequencies.

B. S = [(a: .32), (e: .25), (k: .20), (l: .18), (u: .05)]

C. S_1 = [(a: .32), (l: .18)]

D. S_2 = [(e: .25), (k: .20), (u: .05)]

E. The ABL is: $(3 * .05) + (3 * .2) + (2 * .25) + (2 * .18) + (2 * .32)$



(i) It turns out, we can do better by altering the philosophy a little bit: Put the least frequent letters at the greatest available depths, and put the greater values on the right hand side.

(j) So according the graph above: Swap L and E, then Swap L and L.

(k) This is called the **Bottom-up approach**

(l) Here's the process:

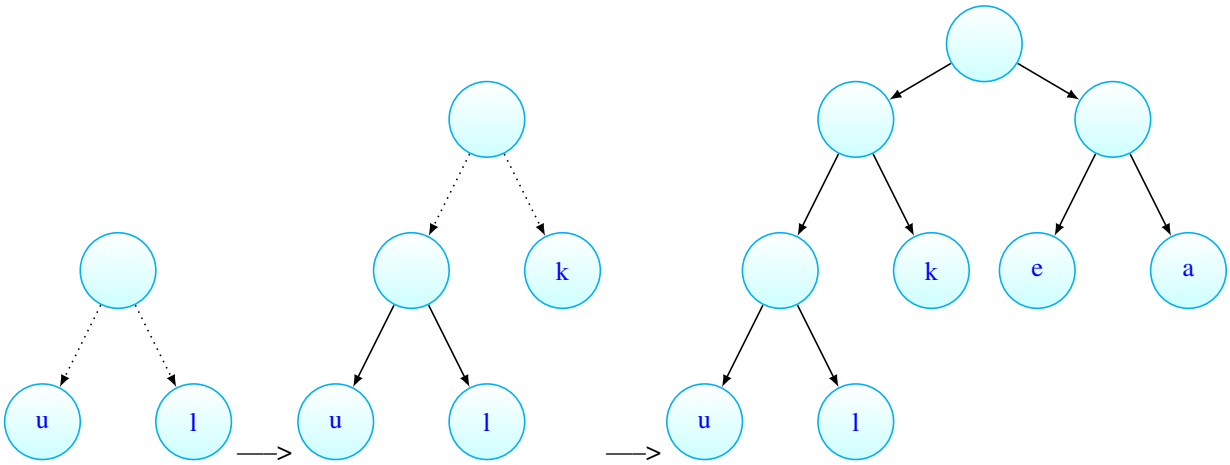
i. Start with the 2 smallest nodes.

ii. Put a root node in between them

iii. Then build the tree backwards.

iv. Point is that the smallest nodes available always formulate a tree first, and then the tree grows vertically as you "append" greater frequency letter nodes.

v. the following sequence of trees highlights this (the dotted lines are "Trees edges that are to be added"



(m) This is the **Huffman Encoding**

(n) Pseudo Code:

```
Huffman(S)
   if (|S| = 2)
      return tree with root and 2 leaves
   else
      let y and z be lowest−frequency letters in S
      S* = S
      remove y and z from S*
      insert new letter m in S* with fm = fy + fz
      T* = Huffman(S*)
      T = add two children y and z to leaf m from T*
      return T
```

(o) The time complexity without a Priority Queue: $O(n^2)$

(p) Now with a PQ: $O(n * log(n))$

(q) The Proof is just an exchange argument: Suppose T* is not the optimal tree, but this optimal tree must also include two nodes of the relative frequency. Then you know the optimal weight - this f_m is also the optimal tree, but that doesn't hold for T*, so that's just a contradiction.

(r) The more comprehensive proof on the slides is just simply an exchange proof with the weights of the python code taken into account. Rather straightforward.

2. And that just about closes out the Greedy^{TM} Algorithms chapter.

Divide and Conquer Algorithms

1. At its simplest understanding, **Divide and Conquer** algorithms simply break a large problem into smaller problems of the same issue to ensure th eproblem can be more easily solved and then recombined.
2. There are billions of applications where this technique is used.
3. Such as Mergesort:
   (a) Divide the array in half.
   (b) Sort the two halves
   (c) Recursively, do this until all elements sorted and are split into length = 1 arrays
   (d) then merge!
   (e) Running time: $O(n * log(n))$
4. There's a proof technique for this called **Telescoping**: Always insert the smallest numbers to the point you can't, and prove the combine.
5. **Counting Inversions**
   (a) A "preference list of songs"
   (b) Gauging their similarity metric: how alike are they
   (c) Lemma: Songs i and j are inverted if $i < j$ but $a_i > a_j$
      i. $1- > 2- > 3- > 4- > 5$
      ii. $1- > 3- > 4- > 2- > 5$
      iii. Values $3 - 2$ and $4 - 2$ are inverted
   (d) A Brute Force comparison takes $O(n^2)$ at worst. That would be a case in which all elements except the middle element are inverted.
   (e) However, this is inefficient.
   (f) Can you do counting inversions for sorting the values?
   (g) Example:
      i. A listing split in two: $1- > 5- > 4- > 8- > 10- > 2$ and $6- > 9- > 12- > 11- > 3- > 7$
      ii. In the first list, the following are inversions: $4 - 2$, $5 - 2$, $5 - 4$, $5 - 8$, and $10 - 2$
      iii. In the second list, the following are inversion: $6 - 3$, $9 - 3$, $9 - 7$, $11 - 3$, $11 - 7$, $12 - 3$, $12 - 7$, $12 - 11$
      iv. So solving for both list (sorting) would take however many attempts in worst case (all elements inverted) counting time.
      v. This sadly is also $O(n^2)$
   (h) So what can we do?
   (i) Best Strategy:
      i. Do inversions on split lists
      ii. Merge the split inversions
      iii. Results in $O(n * log(n))$
      iv. Example:
         A. Split lists of: $7- > 10- > 3$ and $11- > 2- > 16$
         B. After the inversions are solved, we get: $3- > 7- > 10$ and $2- > 11- > 16$
         C. Then just merge the list in order. 2, and then 3, and then 7 and then 10 and so forth.
      v. What's logically appreciable here is the number of extraneous operations is limited and the amount of splits is also limited.
      vi. Splitting the list halfwise results in $2 * (T(n/2))$ done only once.
      vii. So the overall master approach is: $O(2 * T(n/2) * n)$

viii. And this segways into an extremely important Divide and Conquer principle

6. Before introducing the principle, it is important to note something about Divide and Conquer Algorithms: More often then not, in fact basically always, the best and/or only way or proving a D&Q Algorithm is via a technique known as **Recurrences**, which you should already be somewhat familiar with.

7. Master Theorem

(a) Takes the following form: $T(n) = a * T(n/b) + f(n)$

(b) Divide: $a * T(n/b)$

(c) Conquer: $f(n)$

(d) $T(n)$ is at least as large as the divide portion and the conquer portion. If both portions are equal, tack a $log(n)$ onto the overall time complexity.

(e) Solving for the Divide Portion: $n^{log_b a}$

(f) Solving for the Conquer Portion: Simply take the O of $f(n)$.

(g) Example (Mergesort):

i. Split the item in half ($b = 2$) and then continually grow in halves ($a = 2$)

ii. Divide: $n^{log_2 2} = n$. Therefore the Divide portion takes $O(n)$ times.

iii. Conquer: Simply a linear merge of $O(n)$.

iv. Divide == Conquer. Therefore, there's a "tax" of $log(n)$ that must be tacked onto the $O(n)$ time.

v. Thus, the time complexity of Mergesort via the Master Theorem is: $O(n * log(n))$

(h) Proof of Correctness of Divide and Conquer Algorithms via the Master Theorem are usually not very deep and are pretty intuitive.

(i) Also, important side note: Virtually all algorithms in this chapter are important and used in real life. Be sure to take them to heart.

8. **Closest Pair**

(a) Objective: Given n points on a plane, find the smallest pair between them all.

(b) This principle is also shared with many other algorithms, so this point can be generalized. But for the example will think simply about the closest pair.

(c) Assumption: No two points have the same coordinate

(d) Brute Force Approach: Simply calculate all the distances between points and take the minimum. Inefficient as it takes $O(n^2)$ time.

(e) A 1 Dimensional Approach: Sort all the numbers in a list, then take the twi neighbors in the middle. There was a long while where this was the barrier that mathematicians and CS people couldn't get over – transition from 1D to 2D.

(f) **Best Algorithm**:

i. Draw a line L such that it divides all points roughly in half ($1/2 * n$)

ii. Continually split in half the sub-problems

iii. Recrusively find the smallest distance pairs within each split segment

iv. Dilemma: What about points that are on opposite sides of line L? Could they be closer together than any point pairs inside their respective region?

v. Solution:

A. Calculate the distance within each split segment of all points

B. There's then a minimum distance, δ, that is equivalently spread across the dividing line L.

C. Then, calculate the distance within this "delta zone" across line L such that the distance between the two points both horizontally and vertically is minimized and not smaller than δ

D. The vertical element is extremely important to bear in mind, because two points may have a smaller distance than δ horizontally, but a significantly larger distance vertically.

E. At worst, point distances calculated in this delta zone are $O(n)$ time.

vi. Implementation:

A. Split a region in 2 by the line L
B. Calculate the left half region's closest distance pair
C. Do the same for the right
D. Then, taking the minimum of these two split regions, calculate any distances within points on both side of line L that are within the smallest distance found thus far.
E. Be sure to delete any points greater than this distance, and search points vertically, either from the bottom or from the top
F. Then report whatever smallest distance was found after this search (it may still be the minimum of what was found between the two split regions).

(g) Via the master theorem, this results in: $O(2 * T(n/2 + O(n * log(n)))$ –> Do the Master Theorem calculations, you find that the polynomial powers both equal n, therefore, the overall time complexity is: $O(n * log^2 n)$

(h) In summation, simply split the points into region, find the minimum of each region, then choose whichever is smaller, make a zone across the dividing line L that is this minimum distance, check if there are any points in this zone with a smaller distance than it, and then finally report the minimum distance.

(i) I highly implore you to look at the PPT on Piazza for a far more in depth visual perspective, or try drawing the scenario yourself with randomly scattered points.

9. **Integer Multiplication**

(a) The conventional approach is to simply do the carry over of bits with an $O(n^2)$ time complexity. The traditional style, where you multply the ones place by the other value, the tens place by the other value, and so on.

(b) However, there must be an approach better than $O(n^2)$.

(c) How about the following approach:

i. Split an n digit number in halves and do the following:
ii. $x = 2^{n/2} * x_1 + x_0$
iii. $y = 2^{n/2} * y_1 + y_0$
iv. $xy = 2^n * x_1 y_1 + 2^{n/2} * (x_1 y_0 + x_0 y_1) + x_0 y_0$

(d) This generates the following Master's Theorem layout: $4 * T(n/2) + O(n)$ ==> $O(n^2)$

(e) So this is still not enough. Which leads to the **Karatsuba Multiplication**

i. Take the xy equation from above and split the inner paraentheses to include a *third* multiplication:
ii. $xy = 2^n * x_1 y_1 + 2^{n/2} * ((x_1 + x_0)(y_0 + y_1) - x_1 y1 - x_0 y_0) + x_0 y_0$
iii. The biggest change is the manipulation between the internal parenthese by turning the two additions into another set of multiplication and reducing that by the product of the two similar variables.

(f) The Master's Theorem then produces: $3 * T(n/2) + O(n)$ ==> $O(n^{log_2 3})$ => $O(n^{1.585})$

10. **Matrix Multiplication**

(a) Conventional approach using the mathematic notation is $O(n^3)$

(b) $8 * T(n/2) + O(n^2)$ => $O(n^3)$

(c) Is there a way to minimize this? Of course.

(d) Lessen the amount of multiplications by increasing the amount additions. This will lessen the amount of recursions in the Master's Theorem.

(e) Do the following:

i. Divide the blocks into $1/2n$
ii. Compute 14 of these matrices with 10 additions.
iii. Mutliply back 7 of these matrices recursively.

(f) Master's Theorem is: $7 * T(n/2) + O(n^2)$ ==> $O(n^{2.81})$

(g) This can be altered and there is contention as to whether how fast the lowerbound can actually be made. But the idea is that reduction of subproblems speeds up the overall algorithm.

Dynammic Programming

1. **Algorithmic Paradigms**
   (a) Greed: Solve for the quickest, provable solution and optimize the solution as you go.
   (b) Divide And Conquer: Break a problem into sub-problems, solve independently, and then combine the solutions.
   (c) Dynamic Programming: Solve for overlapping smaller sub-problems by optimally building from the previous solution to the consecutively larger problems.

2. Revisiting the Weighted Interval Scheduling
   (a) The greedy approach is to take the earliest finish time and build from that going forward.
   (b) But in this weight schedule problem, salaries are taken into account in order build a "weight" for each job.
   (c) Dynamic Approach: Label jobs by finishing time. $p(j)$ = largest $i < j$ where i is compatible with j
   (d) There are two cases:
      i. First: J is in the optimal solution.
      ii. Second: J is not in the optimal solution.
      iii. if j is in the optimal solution, then only jobs from $1...p(j)$ are in the optimal solution.
      iv. if j is not in the opt solution, then only jobs from $1...j-1$ are in the optimal solution.
         A. IF $j = 0$, then 0
         B. ELSE max$\{value_j$ + Opt$(p(j))$, Opt$(j-1)\}$
   (e) From a purely algorithmic standpoint this does not parse very efficiently: $O(n^2)$.
   (f) **Memoization** – a key dynammic Programming technique that stores the value of all smaller sub-problem solutions.
   (g) As the power point pseudocode shows, the caching of memory in an array allowed for the largest possible job from the smallest job originally to be built up.
   (h) This building of problem solutions slowly increases the size of the overall problem size.
   (i) The algorithm then becomes $O(n * log(n))$
   (j) There is also a bottom-up approach of memoization that takes the caching into account in the actual algoirthm, rather than doing a preliminary search through.

3. Segmented Least Squares
   (a) Given a plot of points, find the best possible approaximative line that most accurately plots the Sum Squared Error.
   (b) Given n data points, order the points by x axis. Find the sequence of lines that minimizes $f(n)$ for the line structure where n is a list of points.
   (c) Choice of $f(n)$ should balance accurracy with parsimony (number of lines).
      i. The function: E + cL, Where E is the SSE of all lines, c is some constant, and L is the number of lines.
      ii. Input: Set of data points (ordered) and a constant factor c
      iii. Notation:
      $OPT(j)$ = optimal solution for data points $1, 2, ...j$
      $e(i, j)$ = minimum SSE for points i through j.
      iv. Compute OPT(j) as:
      { **IF** j = 0 then 0,
      { **ELSE** min$\{e(i, j) + c + OPT(i-1)\}$
   (d) Here's the Pseudocode: (Input denoted as n and c)

```
Segmented Least Squares(n, c) {
  M[0] = 0
  for j in range(1,n):
    for i in range(1,j):
      Compute LSE for e_{i*j} in range p_{i} to p_{j}

  for j in range(1,n):
    M[j] = min(e_{i*j} + c + M[i−1])

  return M{n}
}
```

(e) The overall time complexity of this algorithm is $O(n^2)$ once dynamically programmed.

4. **Knapsack Problem**

(a) These are the involved values:

i. n objects
ii. w_i weight
iii. v_i value
iv. Knapsack has a total capacity of weight W

(b) Take the corresponding table as an example:

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

(c) Assume also the following: the capacity of the Knapsack is $W = 11$

(d) So what items should be put into the knapsack to maximize value and weight?

(e) A greedy approach would tell us that the 5th, 2nd, and 1st would be the best, basing off something like the Weighted Interval Schedule idea.

(f) But we can do better: The best is to choose items 3 and 4 (a yield of 40)

(g) OPT(i) id dependent on two cases:

i. Case 1: An item i is in the optimal Knapsack solution. This means the items ranging from $i-1$ within a weight capacity of $W - w_i$ are left to search through.
ii. Case 2: An item i is not in the optimal solution. This means the items ranging from $i-1$ within the weight capacity that was passed in, W, are left to search through.
iii. It comes down to whether the weight capacity is "decreased" with the addition of an item or not, essentially.
iv. Key Idea: Maintain a running value of the capacity W and the items being recursed over, keeping in mind what each case implies.

5. More on Knapsack

(a) Much like $W = 11$ referenced before with the same table, consider the possibility of including item 4.

i. If added, the weight capacity to work with shrinks to 5
ii. Otherwise, we still maintain 11 to work with but have the rest to choose from.
iii. Naively filling out the table accordingly like this can be upwards of $O(n^3)$, so with a more nuanced way of copying over values of total domain size c we can get it to $O(c)$
iv. We can achieve something like this by **backtracking** – parsing backwards through a "table" of all existing combinations of the items in a bag. Think of it as a big matrix.

A. First case of backtracking: the current item, n_i, is included. Therefore, rise to the "row" above n, $(n-1)$, and subtract the weight capacity $W - w_i$.

B. Second case of backtracking: the current item is not included, so rise to the "row" above n, $(n-1)$, and maintain the same weight capacity W.

C. A crucial point to understand here is where n gets compared too, comparing the total cost with **either** of the two cases above.

D. So, compute OPT(i) as follows:
{ **IF** $i = 0$ then 0
{ **ELSE IF** $w_i > W$ then $OPT(i-1, W)$
{ **ELSE** max$\{OPT(i-1, W), v_i + OPT(i-1, W - w_i)\}$

v. Hilariously, you can think of backtracking as a form of Dijkstra's – Finding the "optimal" path of items from a source index to a target index.

vi. That feels a bit overkill though, personally.

(b) Cautionary Code Note: Be very careful to do examinations in the algorithm in correct order, the evaluation of a row i must come before weight comparison W. For most, that's an intuitive step one would naturally take though.

(c) Pseudocode: (Inputs are amount of objects n, range of weights to w_n, and values in kind)

```
for w in range(1, W):
  M[0, w] = 0

for i in range(1, n):
  for w in range(1, W):
    if(w_{i} > w):
      M[i, w] = M[i−1, w]
    else:
      M[i, w] = max{ M[i−1, w], v_{i} + M[i−1, w − w_{i}] }

return M[n, W]
```

(d) Another Code Note: Carefully study which weights are being compared and understand how the maximum is vital to understanding which recursive, memoized result is best.

(e) To argue correctness of this code, consider what adding i is implying to the end result, what happens if you both **do** and **don't** include it, and then rationalize what would *recursively* occur per procedure. In truth, pseudocode and the like aren't hugely important with these proofs if some elegant and brief statements can be had.

(f) Time complexity here is not technically polynomial, because it is dependent on the weight capacity. A large weight capacity with a large amount of weight differences to consider can be arbitrarily huge. But that's not a critical point until Heuristics I think.

6. **Sequence Alignment**

(a) Comparing strings to check if there's a spell error. More briefly, spellcheck.

(b) The first crucial idea is to think about string similarity – how similar a given string is to other possible strings.

(c) This generates the concept of **Edit Distance** – How many edits are needed to turn one string into another. There are different mismatch and gap penalties depending on what's in the string.

(d) Goal: Find a legal alignment at minimum cost.

i. Difficult in this is finding sup-problems, as most of them are immediately unclear and can change drastically from a high level perspective once altered at a low level point.

ii. Trick is to record and store the prefix for any possible matching of the first string sub-problem with the second.

A. With a string X = $x_1, x_2, ...x_n$, and another string Y = $y_1, y_2, ...y_n$, find an aligment at minimum cost.

B. **Definition**: An Alignment is a set of ordered pairs $x_i - y_j$ such that each item occurs in **at most** one pair and no crossings.

C. A pair **crosses** if for $(x_i - y_j)$ and $(x'_i - y'_j)$, it is that $(i < i')$ but $(j > j')$

D. The specifications of the actual mismatch and gap penalties is negligible unless you are actually implementing the algorithm, so for now don't worry about the specifics.

iii. There are three cases for the algorithm's operation:

A. Case 1: If $(x_i - y_j)$ is included in the Optimal solution. In this case, simply pay the mismatch and alignment costs and carry on.

B. Case 2: The optimal solution leaves (x_i) unmatched. What that means is the position in the string X has either a blank space or an extra character in alignment with Y. So pay the gap mismatch and cost of alignment.

C. Case 3: The optimal solution leaves (y_j) unmatched. Same as above but for string Y, with basically the same penalties.

D. Following these three cases with matching between strings is the best way of doing this via strict Dynamic programming. But there is a catch you'll see soon.

(e) Pseudocode: (Input are length of X as m, length of Y as n, string X, string Y, a penalty of δ, a penalty of α)

```
Sequence Alignment (m, n, X, Y, D, A):
  for i in range(0, m):
    M[0, i] = i*D
  for j in range(0, n):
    M[0, j] = j*D
  for i in range(1,m):
    for j in range(1,n):
      M[i, j] = min( A[x_{i}, y_{j}] + M[i−1, j−1], D + M[i−1, j], D + M[i, j−1] )
  return M[m, n]
```

(f) Time complexity of the algorithm is $\Theta(m * n)$, both in time and space.

(g) Wait, in space too? That's not good, this could be possible huge (10GB array for some biology stuff it seems)

(h) So, what do we do to fix it?

(i) With Divide and Conquer of course.

(j) Using **Linear Space**, we can turn the space portion into a $O(n + m)$ space, while maintaining something near $O(m * n)$ time. In actuality, it does have to increase however to $O(n * m * log(n))$

(k) This just uses something somewhat similar to memoization (well it is that but it's kind of weird), wherein half the "matrix" gets disbanded upon comparison.

(l) Basically, when a matching is found, the **upper left** frame from where that matching occurs is "kept" and likewise the **bottom right** frame is also kept.

(m) The powerpoint images (which I cannot embed here because they're in the slides themself) are better at explaining it then words, so I would implore you to take a look.

(n) Just think of a node on a big matrix being "gotcha" point. Everything in the upper right and lower left frames (that is, everything that is to the left of the node and below it, and vice verse with the upper right) gets thrown away after a finding.

7. **Bellman-Ford – Very Important!**

(a) So, this Algorithm is extremely important for a number of reasons, but mainly because it will, without a doubt, be on the final and it's actually used everywhere for real life purposes.

(b) But before getting into specifics, let's revisit the Shortest Path talk from the graph chapter.

i. In this instance of graphs, unlike with Dijkstra's, we'll be working with weighted, directed graphs that **can have negative weighted edges**.

ii. This is important to remember because Dijkstra's (as we've learned it) cannot handle negative edges, in my find itself in endless loops or wrong pathways otherwise.

iii. Also, on that note, we should specify that it is required we remove negative cycles from our new graph searches, as they are not something we will be accounting for.

(c) The analogy used in class to understand shortest path in graphs with negatives was about baseball. And I really liked it. So I wanna use a discreetly specific example to show this.

i. The analogy was that baseball trades require give and take to ultimately get what you want.

ii. So, here's a specific trade that has good mechanics for that.

iii. Back in the Summr of 2014, the Boston Red Sox were last in the division nearing the trade deadline at the end of July. Their offense was absolutely abysmal, and their pitching wasn't that much greater outside of their ace, Jon Lester.

iv. So, when the deadline arrived, then Boston GM (since fired) made the difficult choice of "selling", in that he gave away top pieces at the deadline knowing full well the team wasn't going anywhere. The main trade he made involved Jon Lester.

v. The trade was the following: Jon Lester and Johnny Gomes were traded to the Oakland Athletics in exchange for Left Fielder Yoenis Cespedes and some cash considerations.

vi. This is where the algorithm is considered. Jon Lester was a top of the line pitcher, so trading him away was essentially a negative. With the return being Yoenis Cespedes, whom also was a top of the line player, the return was seemingly equalized.

vii. Yet, the overall cost of giving 2 players for 1 (and some cash) can be considered a net negative. So the first branching edge had a negative value, a giving away of just too much (but considering the circumstances...)

viii. Fast forward a half year later. The season is over, and the Red Sox ended in last place. They decided it was necessary to make some changes in the offseason. This included another trade, stikingly involving Yoenis Cespedes.

ix. The trade was the following: Yoenis Cespedes to the Detroit Tigers in exchange for pitcher Rick Porcello. In this exchange, it seemed as though the Red Sox were giving up too much (another negative edge for ol' Ben Cherington).

x. But it turned out the trade was very productive for the Red Sox. Rick Porcello, a slight unknown at the time of the trade, became one of the leading pitchers for the Red Sox (and still is).

xi. Although rather long winded (and kind of unnecessary but I figure specific examples sit in people's minds more strongly), this transaction was a series of give and takes where the end result the Red Sox wanted was achieved.

xii. Porcello replaced Lester, and at a younger age, perceived to be a longer lasting solution for the team. So, despite the negative edge at first when trading away Lester, they eventually got back the positive amount in Porcello.

xiii. And that is what the new Shortest Path search is supposed to be reminiscent of: a series of edges traversed where the eventual target node is reach with the best achievable results based on expenses dolled out and profits returned.

(d) If the prior exposition feels like it was a bit too much, I apologize. But at the very least, it should serve as a specific memory to sit in the back of your mind. Reason being that if you were to forget the formation of the algorithm or any principles, the bizarro story you read in your notes about baseball trades might help you recover bits and pieces of information before it all comes back to you. I use it pretty frequently, a way of associative learning.

(e) Anyhow, back to the algorithm.

(f) The algorithm should take into account all possible weight paths that are collectively short from the source node to the target node. Then backtrace to the best possible (minimized pathway).

(g) There are two cases to follow:

i. OPT(i, v) = length of the shortest $v->t$ path P using at most i edges.

ii. Case 1: P uses at most $i-1$ edges then OPT(i, v) = OPT$(i-1, v)$

iii. Case 2: P uses exactly i edges.

iv. Case 1 ties directly with Case 2: If Case 2 holds, then the first edge, say $v->w$ is used in the optimal solution, and then there remains some path from $w->t$ where there is **at most** $i-1$ edges.

v. Basically, Case 1 affirms that if a specific edge is taking, then all smaller sub-group of edges will have a pathway that works. So with Case 2, the first edge works, and Case 1 verifies that there exists a sub-path after that first edge that works.

vi. As follows:
{ **IF** $i = 0$ then 0
{ **ELSE** – min{OPT$(i-1, v)$ min{OPT$(i-1, w)$+ c_{vw}}}

vii. min(OPT$(i-1, v)$) => The shortest edge found thus far.

viii. min(OPT$(i-1, w)$) => All possible (1 node) hopes from v to w that finds the shortest edge from v.

(h) It happens that the traditional matrix way of backtracing does not work well here. Time complexity is reasonable ($O(m * n)$) but the space is dreadful ($O(n^2)$).

(i) So we need a more sound approach, and that's where good ol' pointers come in.

(j) Few things are better than the nitty and gritty of pointer manipulation. The trick is rather simple, simply use pointers as "trackers".

(k) As the value changes, have a pointer track specifically where the source node came from and another pointer (or perhaps a series of pointers) to the node that it was changed to. Otherwise, simply maintain pointers to the value you were on.

(l) This becomes nothing more than an abstract Linked List.

(m) And the new space needed becomes $O(m + n)$, which is "better" for most practical standards with reasonable m.

(n) Pseudocode: (Input is a Graph G, a source node s, and a target node t, and V is the set of all vertices and E for edges)

```
Push—Based—Shortest—Path(G, s, t):
  for v in V:
    M[v] = infinity
    successor[v] = some value phi

  M[t] = 0
  for i in range(1, n—1):
    for w in V:
      if(M[w] was previously updated in an iteration):
        for v where (v,w) belongs in E:
          if(M[v] > M[w] + c_{vw}):
            M[v] = M[w] + c_{vw}
            successor[v] = w
      else if no M[v] changed in iteration i:
        :stop:
```

(o) By the way, this is called Bellman-Ford. It's really nice. And that's that.

8. **Distance Vector Protocol**

(a) A practical application of graph searching.

(b) Node = router, Edge = communication link, Cost of edge = delay of communication

(c) Dijkstra's require global network information (NSA probably likes this one (just joking NSA haha don't arrest me)), Bellman-Ford uses only local knowledge from neighbors (Your local basement hacker probably sniffs this one).

(d) The algorithms run even without complete discrete synchronization from the networks.

(e) Each router has a vector for shortest paths. So naturally, the algorithm uses the nearest probable routers by running n computations and picking.

(f) "Routing by rumor" is a really awful but good way of describing it. Listen to what the routers are saying and base the shortest path off that hearsay.

(g) There are some wacky shenanigans that can occur though: Counting by infinity and looping.

(h) Algorithm mistakenly hears a rumor about a knew short path in 2 hops
-> Oh cool gotta try this
-> Passes through the first node, then back to the other one without realizing it
-> Hmm this place looks familiar
-> Does the same thing again
-> Let's Go Looping Through Networks^{TM} by Dr. Dijkstra and Mr. Bellman

9. And that just about covers Dynamic Programming. Hurray.

Network Flow

1. How to push a flow through a Network to maximize output.
2. In contrast to previous chapters, this chapter will only introduce 2 (yes, two) algorithms: Max-Flow and Min-Cut.
3. **Minimum Cut Problem**
   (a) A **flow network** is an abstraction for material flowing through edges. (Think of water in pipes)
   (b) the **capacity** is the total cap value of flow coming out of a source node to a target node on an edge.
   (c) An $s-t$ **cut** is a partition of s side of nodes and a t side of nodes. Basically think of it as grouping nodes into cliques of specific nodes. Nodes all on one path, nodes in the bottom, nodes of a similar property, etc.
   (d) A cut for these problems are important as to describing what edges cross a source node.
   (e) Minimum Cut is extremely important -> It will serve as a basis for a lot of the Max-Flow properties and help assure proofs.
   (f) An $s-t$ flow is a function that satisfies:
      i. $\forall e \in E$: $0 \leq f(e) \leq c(e)$ –> **Capacity**
      ii. $\forall v \in V$: $\sum_{e-in-V}(f(e)) = \sum_{e-out-V}(f(e))$ –> **(Flow In = Flow Out)**
   (g) The last one is particularly important – If the source node flows out 10 gallons of water, then 10 gallons of water best be coming into the target node.
   (h) All of this leads into Max-Flow, which has a similarity to Min-Cut but with some subtle differences.
4. **Maximum Flow**
   (a) The **value** of a flow is how much is flowing out of a source node on the summation of its edges.
   (b) A flow is the optimal setting of a Min-Cut where value of a flow is maximized and best possible capacity constraints are met.
   (c) Basically, the best possible outbound of flow that can be achieved in where all Min-Cut requirements listed above are met. That means no exceeding capacity and flow in must equal flow out.
   (d) Flow Value Lemma:
      i. If there is a flow F and an $s-t$ cut, the net flow across the cut is equal to the total base leaving source node s.
      ii. Simply, the amount leaving the source node / source cut should be equal where any domain of a $s-t$ Min-Cut is achievable on the graph.
      iii. If you have an $s-t$ Min-Cut, this flow value should always be the same.
      iv. This flow value should also **always equal the final flow in towards the target node** t
      v. The proof is as simple as this: If there is a flow between some node set V that is a Min-Cut, than any other Min-Cut can be interchanged to represent these sums.
   (e) The concept of **Weak Duality** is that the value of the flow is at most the total capacity of the cut.
   (f) For example: IF cut capacity = 30, then the flow value ≤ 30
   (g) Corollary to this: If F is a flow and (A, B) is a cut, then IF $v(f)$ = cap(A, B), then F is a Max-Flow and (A, B) is a Min-Cut

(h) Wait what?

(i) Basically, because you reached the absolute total capacity, it is not possible to exceed it nor can any other cut be less than it. Therefore it must be a Min-Cut. And remember, a Min-Cut is a tricky way of getting a Max-Flow.

(j) **IMPORTANT**: You **MUST** assert a Certificate of Optimality with these problems.

(k) Certificate of Optimality? Is it expensive? Is it legal?

(l) Yes to both. Simply assert the following: If a cut capacity and flow value are both found, then the optimal solution is therefore found.

(m) This makes your proofs way more approachable.

(n) Another important note: Checking a Max-Flow/Min-Cut is really easy, way easier than finding one in fact. Can virtually always do this relatively quickly.

(o) So let's get into the actual, ya know, algorithm:

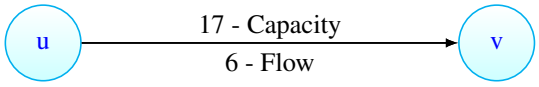
i. As always we start with everyone favorite: Greedy™

ii. Start with empty edges, and fill $s - t$ cut edges accordingly.

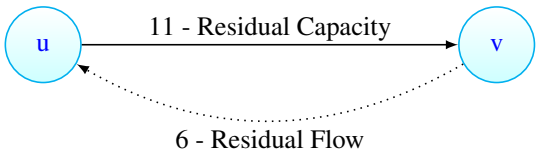
iii. Sadly, you'll get stuck with a non-optimal solution though.

iv. So to fix this, we can use a **Residual Graph**

v. A residual graph essential "undoes" the amount of flow sent out from an edge by sending the difference back towards it. It looks like this:



vi. Now send 6 units through and...



vii. It is simply the difference of the original total capacity minus the flow sent through.

viii. Retain only positive values everytime you make this Residual Graph.

ix. In fact, there's a specific routin:

x. **Ford Fulkerson**

A. Initialize 0 flow across all edges.

B. Find first greedy path, then put in the residual capacity for the edges accordingly.

C. These edges in the residual graph operate in the opposite direction.

D. Continually find forward direction paths, and then create residual edges in the opposite direction each time.

E. Basically, this all about **augmenting**: altering a pathway for a specific flow that was passed in, and returning the corresponding residul flow (Original Capacity - Flow in) back at it.

F. This augmenting can take place multiple times, and it is **very much possible** that an augmented path gets re-augmented after another traversal through.

G. Basically, if an edge with capacity 10 has 5 flow sent in, its residual flow is also 5. But that means a second time through, another 5 flow might be passed in, and then residual flow would be 0. So carefully consider the residual flow.

H. IF there is no path that exists from the source node s to the target node t (that follows the requirements of the Min-Cut), then there cannot be anymore augmented paths.

I. If the cur of vertices reachable to the source node s equals the flow value, then the algorithm is finished.

xi. The trickiest part of this algorithm is subtracting edges for the residual flow.

xii. Also, the amount of paths to augment can be a prime factor in determining algorithm efficiency.

xiii. The following 3 Theorems hold for the Ford Fulkerson, and more importantly, **The proof of one of them will assert the proof of them all**:

A. There's a cut (A, B) such that $v(f) = cap(A, B)$

B. Flow F is a Maximum Flow.

C. No augmenting paths relative to F

D. One-to-Two and Two-to-Three are straightforward, but One-to-Three is trickier.

E. One-to-Two: The Corollary to the Weak Duality mentioned above is exactly the proof needed.

F. Two-to-Three: Contrapositive – If there's an augmenting path, we can improve the overall flow by sending some along this path.

G. One-to-Three: If A is a set of vertices that contains source node s, then B is a set of vertices containing target node t. Then this is precisely the cap(A, B)

xiv. The Running time for the algorithm varies:

A. Every value for capacity is integral between 1 and C where C is the maximum.

B. Every flow value $f(e)$ and every residual capacity $c_f(e)$ remains an integer.

C. The main theorem is such that: The algorithm terminates in $v(f*) \leq n * C$.

D. So if $C = 1$, then it is $O(m * n)$ time. and when $C > 1$ is has potential of $O(m * n * C)$

E. This means it is not necessarily polynomial time (you would need a $log(C)$ to maintain that, not just $O(C)$).

xv. However, we can optimize and potentiall remove this C by making proper augmented path choices.

xvi. Specifically, if we maintain a set of vertices where we throttle a sufficiently large bottleneck capacity.

A. Take a graph.

B. Filter all edges less than a predefined δ value.

C. Run Ford-Fulkerson on the filtered graph.

D. Lessen the value of δ and then re-run Ford Fulkerson.

E. Repeatedly do this step until δ includes all values within the graph.

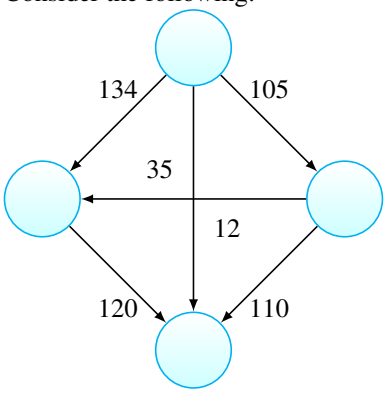
F. You can start δ at C and decrease by a power of 2. Which is $log(C)$

G. Consider δ to simply be a scaling factor, where δ is a "threshold".

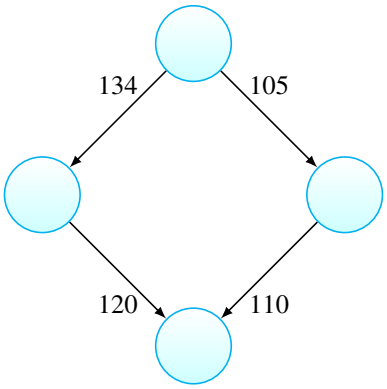
H. By that I mean that only edges with capacity that is at or exceeds the value of δ should be augmentable at a time.

I. So when δ is decreased, more edges are allowed onto the graph for augmenting.

J. Consider the following:



K. Then if C were to be a value of 100, the two edges on the inner part of the graph would "disappear" during a run of the Ford-Fulkerson Algorithm:



L. And this is essentially the point: In order to properly augment the best paths, its best to work with the biggest paths first and slowly wittle them down.

M. The proof is really just this: If the algorithm terminates then it is a Max Flow because integrality variant lemma tells us $G_f(\delta)$ => G_f and once δ is finished you can't augment anymore paths.

xvii. In the following pseudocode, E is a set of edges and D is delta.

xviii. Pseudocode: (Inputs are a Graph G, source node s, target node t, a capacity c)

```
Scaling—Max—Flow(G, s, t, c):
  for e in E:
    f(e) = 0
  D = Smallest power of 2 greater than or equal to C
  G_{f} = Residual Graph

  while(D >= 1):
    G_{f}(D) = Delta based residual graph
    while (There is a Path P that can be Augmented in G_{f}(D)):
      f = Augment(f, c, P)
      update G_{f}(D)
    D = D/2

  return f
```

(p) This scaling implementation forces each path augmentation to run in $O(m * log(C))$ time, with an overall complexity of $O(m^2 * log(C))$.

5. So now that the actual algorithms are established, let's look at varying applications:

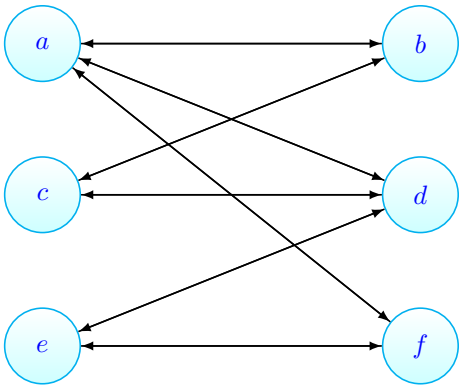
6. **Bipartite Matching**

   (a) So, as you may recall, a Bipartite graph is a special type of graph where edges connect nodes into, essentially, independent/disjoint sets. (If you don't remember, scroll up and read in te Graph section)

   (b) So Bipartite Matching is a special case of Network Flow: A matching is found if each node in the Graph is found in at most M edges where no two edges share an endpoint.

   (c) Maximum Bipartite Matching are unique (there is only one solution) because adding any other existing edge in the graph would mean there are two edges sharing a node endpoint.

   (d) This is tricky at first to grasp, but simply think of a graph as a picture of the following:



(e) According to this image, we can partition this graph into the following matchings: (A-B, C-D, E-F) OR (A-D, C-B, E-F) OR (A-F, C-B, E-D) etc.

(f) The main idea is that with each matching, two nodes are only connected by one edge. No node has more than two incoming/outgoing edges from it that are in the matching.

(g) But how do we find these algorithmically in a Max Flow situation, where there's a unique solution?

(h) Create a super graph of the original, with a super source s and a super sink t:

i. Create another graph, with the original graph remaining in the center but with the following attachments

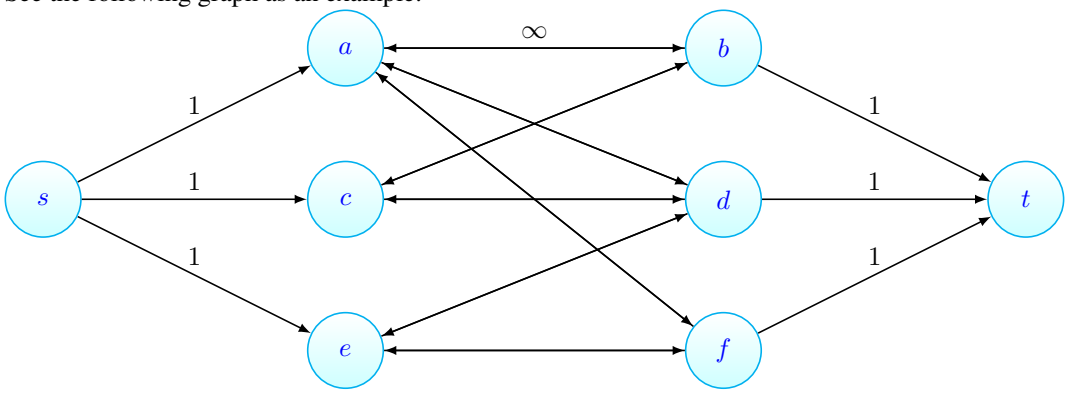
ii. On the left side, have a node s that precedes every left node in the original Bipartite graph. This is the source node by which flow will be exerted out, it is a "super source node".

iii. On the right side, have a node t that succeeds every right node in the original Bipatite graph. This is the target node by which all flowmust eventually end in, it is a "super sink node".

iv. The edges that go out and come in to the source and sink must have capacity in relative units to what the algorithm is looking for. So, each edge can have 1 for example in the case where 5 nodes are being parsed over at unit 1.

v. The edges within the bipartite graph can *usually* be given an original notation of ∞ as the algorithm will eventually determine the capacity as it works.

vi. See the following graph as an example:



vii. This is probably the quickest and easiest way of thinking of these graphs, by orientation relative to s and t.

viii. Proof:

A. So, the flow generates from s is sent out on k differnt edges. So it has cardinality of k.

B. We know k is an integer that matches the lefthand side of the bipartite graph, with unit 1 flowing out of s

C. As per a cut property, the amount leaving s plus a grouping of the lefthand side of the nodes should be equal to the opposite side.

D. Then that would mean 1 unit edges are only possible once per edge in the Bipartite graph

E. So we know that the subset size of edges in a cut, say M, has a cardinality equal to k

F. And that is the original flow being generated.

(i) A Bipartite Matching is **perfect** if each node in the Bipartite matching appears in exactly one edge only.

i. The number of nodes on one side have to equal the number of nodes adjacent to it on the other side (Adjacent meaning they're are equal amount of nodes with 1 edge attachments only)

ii. This should work with any subset of nodes from the Bipartite Matching.

iii. This is called the Marriage Theorem: IF there is a bipartite graph with |L| = |R|, then it is a perfect matching if a subset of nodes for one side S is $\leq$ a correpsonding subset on the other side $N(S)$

iv. The proof is a bit pedantic, so let me simplify it:

A. Take a Max-Flow/Min-Cut Graph

B. There must be a subset of nodes on the graph that has a outbound value that is greater than the capacity.

C. If that's the case, there must be an assortment of nodes (two pairs, to be specific) within this subset that has a matching flow to the capacity

D. This subset must have a pairing with outbound nodes that "exceed" the capacity, such that if C is the overflowing capacity, then C_a is the half that works and C_b is the amount added to C_a that will make it too large.

E. Then the subset that corresponds to C_a must be within C, so this essentially equals the Min-Cut of $cap(A, B)$ for a Min-Cut Graph when C_b is considered.

F. Then, recalling the idea that $N(S)$ is a subset that has the correct corresponding flow out of the original subset to reach the max flow, there is a "right hand subset" of nodes that complements C_b, call it R_b

G. Thus, if you add together this R_b with C_b, you get the $cap(A, B)$.

H. But wait, there's more: That must mean the nodes that are left from C are the "complement" of C_a. And the cardinality of these nodes **must** be less than R_b. So...

I. If $R_a < R_b$, we can reason that $cap(A, B) - C_b = R_a$. Then that must mean $C - C_b = C_a$

J. So C_a is proper subset choice for the flow to maintain a perfect matching.

K. I'm not sure how necessary this proof will be, so if it appears to be a bit much, don't worry.

(j) Time Complexity of this stuff:

i. Genering augmenting: $O(m * n)$

ii. Capacity Scaling: $O(m^2)$

iii. Shortest Augmented Path: $O(m * n^{1/2})$

(k) There's also non-Bipartite Matching, but that's harder and not needed. Although, you can get the average complexity to be about $O(n^4)$ for it, with best case of $O(m * n^{1/2})$

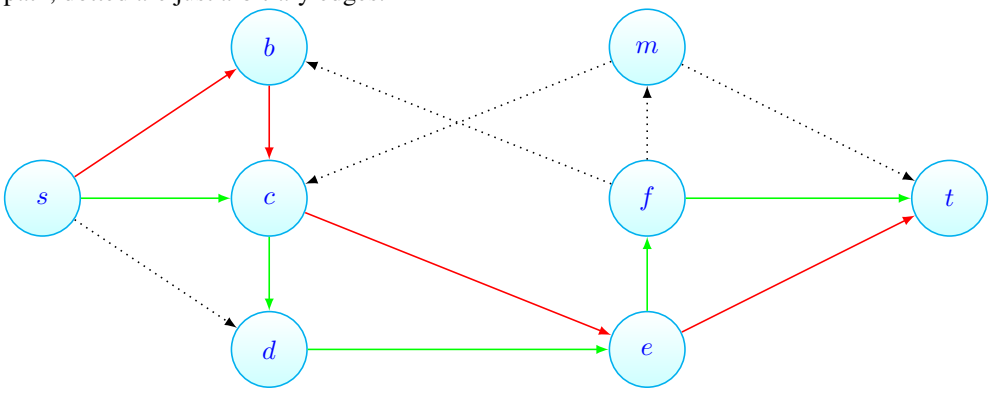
7. **Disjoint Paths**

(a) Working with Directed graphs some more. First, with **Edge-disjoint paths**

(b) **Disjoint edge paths** are paths from a source node s to a target node t where both paths take completely different roads.

(c) That's to say each path has completely unique edges from one another, there is no edge on either path that is shared among them.

(d) Take a look at the following graph and pay attention to the following: Green is one path, Red is another path, dotted are just arbitrary edges.



(e) Finding Max Flow:

i. To find the max flow, assign unit capacity (i.e. just put a 1) for every edge.

ii. Because each edge has a unit capacity, that means each edge's flow will produce 1 unit.

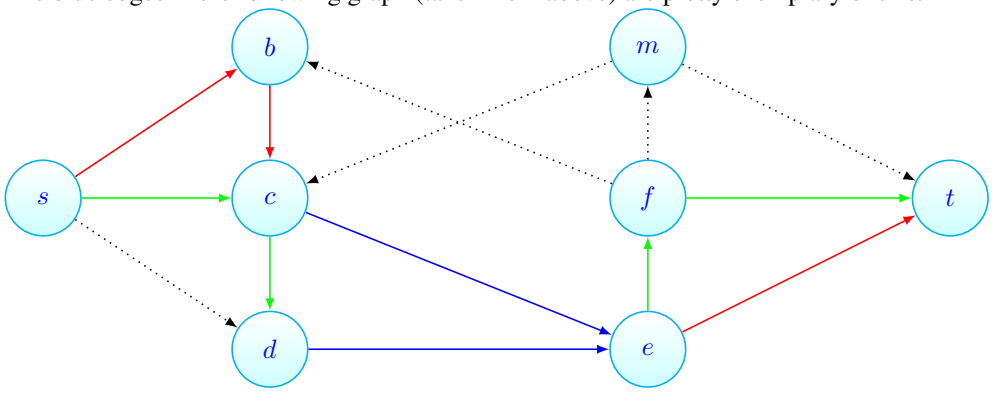
iii. Which means other edges along a path have to be ready to take in that 1 unit of flow.

iv. Which means if you force this all the way to t, however many times until the graph is maximized, you have a max flow with unique edge-disjoint paths.

(f) Now consider **Network Connectivity**

i. This one is somewhat simpler: Find a minimum number of edges such that, if you remove these edges, t can no longer be reached from s

ii. The blue edges in the following graph (taken from above) are pretty exemplary of this.



iii. The proof of this is also easy:

A. Suppose the max paths to t are included in some set F.

B. So if you remove just one path in this set of F, then s cannot reach t

C. Why? Because the size of F is k, and the number of edge-disjoint paths from above is k. So removing just one makes it smaller than k.

8. **Extensions to Max Flow**

(a) **Circulation with Demands**

i. Supply and Demand. Each edge has a weight, graph has overall capacity, and there exists a demand.

ii. The demand is this: Flow(in) - Flow(out)

iii. Flow must be less than or equal to the capacity but also positive.

iv. If demand per vertex, say $d(v)$ is: $d(v) > 0$ then DEMAND; $d(v) < 0$ then SUPPLY; $d(v) = 0$ then EQUAL SHIPMENT

v. So how do we check if this exists?

vi. Well, it is a **necessity** that the sum of supplies = sum of demands.

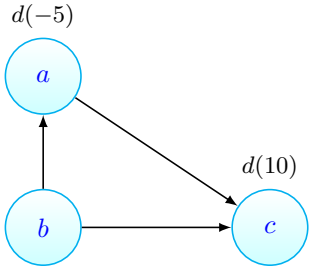
vii. Max Flow Formulation:

A. Set up a source s and a target t on the ends of the graph.

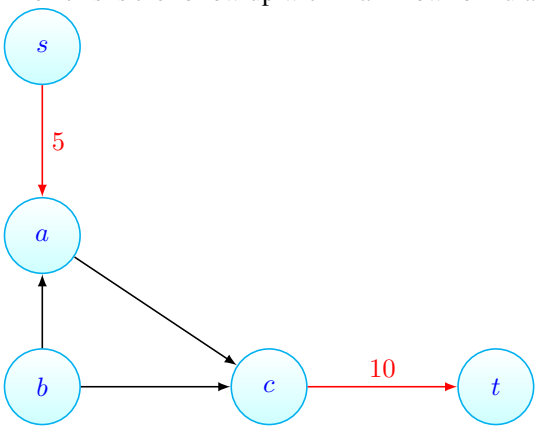
B. If there is a supply on an vertex ($d(v) < 0$), add an edge from s to the supply node that is of equal absolute value. (IF $d(v) < 0$ then the edge should be $-d(v)$ from source s)

C. Inversely, if there is a demand on a vertex ($d(v) > 0$), add an edge from this node to t that should be of equal outgoing value (IF $d(v) > 0$ then edge-to-node t is $d(v)$)

D. Take a look at the first following graph. If this before max flow formulation...



E. Then this is the follow up with Max Flow formulaiton:



viii. If there is a sum of demands and supplies that is bigger than the capacity $cap(A, B)$, then there is **not** a circulation in the graph.

ix. You can also do Circulation with **lower bounds** meaning that rather than the weight of an edge being greater than 0, it must be greater than the lower bound.

x. For this, all you have to do is put a secondary integer on each edge, and at the vertex where the $d(v)$ is, take into account the lower bound like this: $d(v) - L(e)$ OR $d(v) + L(e)$.

9. **Survey Designs**

(a) A survey of n consumers about m products.

(b) Can ony survey a person about a product if they own it.

(c) Ask a consumer if they own it between certain questions, and then ask consumers that own the product about the actual product.

(d) It just so happens the proper way to solve this issue is with a combination of Max Flow from above:

i. Formulate a Circulation graph based on a Bipartite graph between C consumers and P products

ii. There should be a super source s connected to all consumers, and a super ink t with incoming edges from all products (A bipartite graph for matching, as above).

iii. The Circulation graph from this should have normal supply and demand, as well as lower bounds.

iv. And voila, a normal Max Flow solution wil properly determine the best survey.

10. **Image Segmentation**

(a) Dividing an image into smaller regions, in effort to identify individual objects from a photo.

(b) Pixels in an image will represent a graph in this case. Each pixel will be a vertex, and edges are denoted by a pair of a pixel and its correpsonding neighbors.

(c) There must be designation for what pixel is in the foreground, denoted i, and what pixel is in the background, denoted j. There is also a penalty p for the labeling and distinguishing of the two.

(d) Goal: Maximize the amount of (foreground + background) pixels - (pixel penalty)

(e) This graph is a bit odd, there's no original source or sink node and the graph is undirected. So you have to turn it into a Minimization problem.

(f) Specifically, you have to look to achieve the greatest sum **with** penalty **added**:
$\sum_{j \in B}(a_j) + \sum_{i \in A}(b_i) + \sum_{i,j \in E}(p_{ij})$

(g) So, do the following:

i. Establish a "foreground" node and a "background" node s and t respectively.

ii. Link s, the foreground node, to the first background node j as an anti-parallel

iii. Link t, the background node, to the first foreground i as an anti-parallel

iv. The difference between i and j is specifically the penalty p. Which in this case will only be 1.

(h) I cannot imagine this will be a critical topic on the final, but it's nice to have in mind.

11. **Project Selection**

(a) Set of possible projects P with associated revenues p_v

(b) Also a set of prerequisites in which project v and project w must be done together, $(v, w) \in E$

(c) A subset of projects P is feasible if it includes both the projects and the project's prerequisites.

(d) Create a graph with a super source and a super sink, with each having a capacity relative to p.

(e) If $p_v > 0$, then set an edge from the source to the vertex. Else if $p_v < 0$, then set an edge from the vertex to the target with $-p_v$ value.

(f) As most often is the case, set infinity between each other edge of nodes.

(g) A Min-Cut is found if the subset of projects from P minus the set of nodes from source is the optimal capacity.

(h) This is verified by the fact a min-cut must have the same flow capacity as a Max Flow, and that edges are given ∞ capacity to work with initially.

(i) Again, I can't imagine this will be a critical piece on the final, but it's good to know anyhow.

12. **Baseball Elimination**

(a) I'll enjoy talking about this one :)

(b) w are wins, l are loss, and r are remaining games

(c) If $w_i + r_i < w_j$ then team i is eliminated.

(d) Elimination depends on the combination of how many games have been won, how many are left, and whom they're against.

i. A set of teams exists. The is one unique team, call it u. We want to know if u can finish with the most wins.

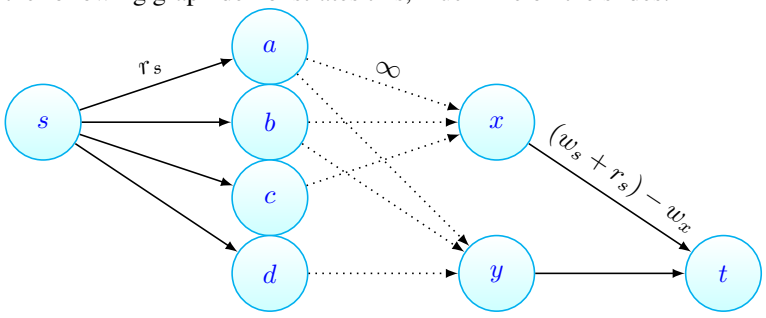
ii. But u has rival teams x and y. Team x has w_x wins, and additional r_{xy} games against y.

iii. So formulate a graph with the following: A super source and super sink. A bipartite graph centered between where the left side is games, and the right side is teams.

iv. Set flow out of super source s under the assumption that u wins all remaining games.

v. Then, assum flow from each team to sink t is the flow from source s minus the wins generared by the team node.

vi. the following graph demonstrates this, much like on the slides:



vii. We can then make a claim: The team u cannot be eliminated IFF the Max Flow algorithm **saturates all edges flowing out from the super source** s

viii. The capacity constraint on edges flowing into t ensures no team will win "too many games"

ix. And we can assert that a team is eliminated more generally with the following:

A. The total number of wins across all teams is denoted by $W(T)$.

B. The total number of remaining games across all team is denoted by $R(T)$.

C. The total number of team is the cardinality of T, $|T|$.

D. IF for a team u the following is the case, then they are eliminated from playoff contention:

$$\frac{W(T)+R(T)}{|T|} > w_u + r_u \text{ where}$$

x. The proof:

A. A unique team u is eliminated iff there is a Subset $T*$ within T that eliminates u.

B. This means $T*$ is simply a team or collection of teams who have a $(w_u + r_u) - w_x$ such that it prevents the super source from fully saturating their outgoing edges.

C. Or more basically, that team or set of teams has enough more games to play to eliminate u based on their record as of that moment.

D. Consider a max flow formulation with a Min-Cut of (A, B)

E. It is possible that (in the following graph), there is a game $x - y$ that is within the left side on the cut (the source side), A because they both exist in $T*$. So $x - y \in A$.

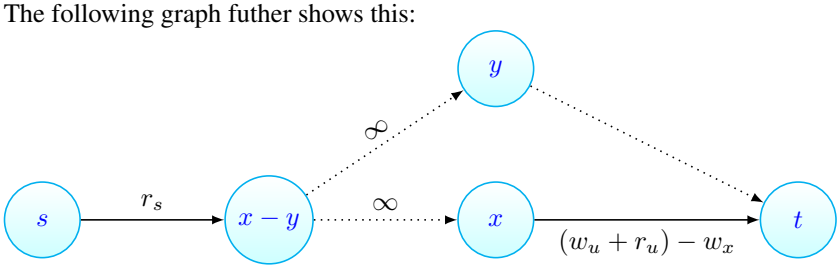
F. Infinite capacity edges guarantee that $x \in A$ and $y \in A$.

G. However, including these teams in this game increases the flow into sink t and the capacity decreases for the cut.

H. Which simply shows that:

$$\frac{W(T*)+R(T*)}{|T*|} > w_u + r_u$$

I. The following graph futher shows this:



xi. Again I'm not sure how critical this stuff is, but having it is better than not.

13. And that essentially ends this chapter. But there's one thing I would like to note: the ending example applications as applications themselves are a bit much for a final, but the biggest takeaway from them are the formulations of the graph and the proofs. Specifically the Bipartite Matching, Disjoint Edge paths, and Circulation graphs. These all seem to have influence on other applications of the Max Flow/Min-Cut application hemisphere. So carefully review those three (among review of the others if you wish) for their proofs and how the graph is actually oriented and setup.

NP and Computation Intractibility

1. This chapter is all about reductions and gauging how difficult a theoretical algorithm is for a problem. The **Intractibility** of NP problems and the like in particular.

2. Using the techniques we can consider whether its even possible before trying to implement an algorithm.

3. Three specific realms to consider:

   (a) **NP-Completeness** –> $O(n^k)$ Algorithm is unlikely

   (b) **PSPACE-Completeness** –> $O(n^k)$ Certification algorithm unlikely. Hard to even theorize an algorithm that is polynomial.

   (c) **Undecidability** –> :[Just_give_up.jpg]: There is no algorithm theoretically possible.

4. The algorithms defined in this Chapter are **Reductions** - determining an algorithm's complexity by reducing a problem of like kind to it.

5. **Polynomial-Time Reductions**

   (a) A **polynomial time reduction** is a sequence in which a proble X can be "reduced" to another problem Y if X can be solved as a certain "special case"of Y.

(b) Here's a table of algorithms established as polynomial versus algorithms that are not yet found to be:

Found	Not Found
Shortest Path	Longest Path
Matching	3D-Matching
Min-Cut	Max-Cut
2-SAT	3-SAT
Planar 4-Color	Planar 3-Color
Bipartite Vertex Cover	Vertex Cover
Primality Testing	Factoring

(c) For example, Multiplication is simply a series of Additions. $3x3$ is the same thing as $3 + 3 + 3$. On top of that, we can examine a problem's time complexity relative to anothers (that isn't as smooth as this Multiplication to Addition example, but the idea is generally the same)

(d) A more apt example is the Max Flow and Maximum Bipartite Matching. We know that Max Flow can work as a Maximum Bipartite Matching.

(e) The sequence of reduction goes like this:

i. Take an algorithm P_F (For Max Flow)
ii. We know it operates in polynomial time, it's been proven and implemented.
iii. Then take another algorithm P_M (For Max Matching)
iv. We conjecture, based off knowledge of both algorithms, that P_M can operate as a special case of P_F
v. So, we attempt to solve P_M as a case of P_F
vi. If we then can find a solution to P_M, we can trace it back as a solution for P_F
vii. And if that is the case then we have found a polynomial time algorithm for this problem.
viii. This notation will **always take the form of the following notation**: $P_M \leq_p P_F$
ix. NOTE: Be very careful, as this means P_M reduces **to** P_F. Not **from**.

(f) So then, is ($P_{ShortestPath} \leq_p P_F$) a reasonable reduction?

(g) Yes! Why? Because finding the Shortest Path in a graph using Dijkstra's algorithm is just a subtle form of the Max Flow problem!

(h) So to be absolutely clear, we can make observations about polynomial reductions:

i. **IF** $(X \leq_p Y)$ AND we know that Y runs in polynomial time, then X will also run in polynomial time.
ii. **IF** $(X \leq_p Y)$ AND we know that X cannot be solved in polynomial time, than neither can Y.
iii. Lastly, **IF** $(X \leq_p Y)$ AND $(Y \leq_p X)$ then we can assert: $(X \equiv_p Y)$

(i) These observations are essentially universal when it comes to reductions. Please remember them!

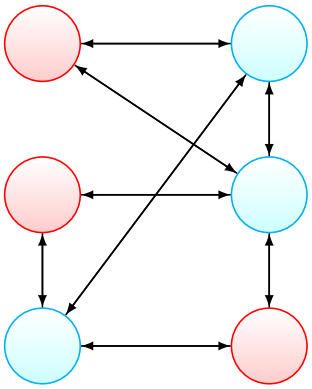
6. Now, let's examine differing techniques and strategies for Reductions:

7. **Reduction by Simple Equivalence**

(a) We'll use a specific example involving the **Independent Set** problem and the **Vertex Cover** problem.

(b) An **Independent Set** is a set of nodes in a graph such that no two nodes share a direct edge. Every node in the independent set is indirectly connected to one another by non-independent nodes.

(c) See the following graph (Red are in the Independent set and Blue are not in it):



(d) Now, a **Vertex Cover** are a set of nodes such that, for every edge on the graph, at least one of its endpoints is a node in the set.

(e) Specifically, the set of nodes touches every single edge at least once.

(f) But wait. If they touch every edge, then there is a guarantee two nodes may be incident to each other.

(g) If that's the case, then that would mean the nodes in this subset are guaranteed not to be in the independent set.

(h) So in truth, the **Vertex Cover** is simply the inverse of the **Independent Set**. So all the Blue nodes in the graph above are in the Vertex Cover and the Red ones are not.

(i) And voila, we've reduced the Vertex Cover problem to an implementation of the Independent Set problem. And that's essentially the proof. We can show that two nodes can't be directly connected in the Independent Set, and that the Vertex Cover is guaranteed to have this as each edge has to be toughed by an end node, so one of these two nodes will absolutely be in the Cover.

(j) Vertex cover is verfiably NP-Complete, and thus so is Independent Set.

8. **Reduction from Special Case to General Case**

(a) Using another example, this time involving **Set Cover** and the Vertex Cover.

(b) A **Set Cover** is when, given a set U, a collection of subsets $S_1 ... S_n$ of U, and an integer k, is there a collection of subsets $\leq k$ such that appending them together produces U

(c) For example: Say there is a company hiring applicants to fulfill X, Y, and Z.

i. Applicant Jim has X skills

ii. Applicant Mary has Z skills

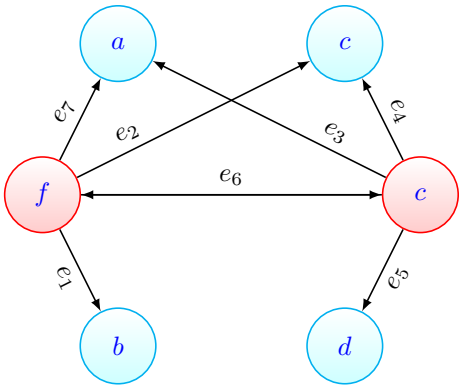
iii. Applicant Frank has X and Y skills

iv. Clearly, the subset of (Mary $\cup$ Frank) would fulfill the company's hiring requirements.

(d) In order to prove $P_{VC} \leq_p P_{SC}$, we can just set up a Vertex Cover graph using an instance of the Set Cover mapping. Each subset S_i from the Set Cover is simply a series of edges to nodes. So, finding the subset of nodes that touches every edge is the same as finding the subsets that, when concatenated together, equal the total set U.

(e) So if $U = 1, 2, 3, 4, 5, 6, 7$, $k = 2$, and the Set Cover is $S_c = 3, 4, 5, 6 \cup S_f = 1, 2, 6, 7$ then...

(f) The Vertex Cover is simply the nodes C and F such that C touches edges e_3, e_4, e_5, e_6 and F touches edges e_1, e_2, e_6, e_7 on a graph that has at most 7 edges.



9. **Reductions via "Gadgets"**
   (a) An example with the **Satisfiability** Problem.
      i. A **Literal** is a variable or its negation: x or $\bar{x}$
      ii. A **Clause** is a disjunction (OR) of literals: $x_1 \vee x_2 \vee x_3$
      iii. **Conjunctive Normal Form** is a conjunction of clauses labels by ϕ: $\phi = C_1 \wedge C_2 \wedge C_3$
      iv. **SAT** is the truth of a conjunctive normal formula.
      v. Finally, **3-SAT** is a SAT where each clause has exactly 3 literals.
   (b) Fair Warning: It turns out working with reductions with 3-SAT is absolutely terrible and a complete pain.
   (c) In this case, we can show that (3-SAT $\leq_p$ Independent Cover).
   (d) To construct the Graph for an Independent Set, just do the following:
      i. The independent set contains 3 Vertices for each clause, one for each literal, so set the graph up as such.
      ii. Connect 3 literals in a clause in a triangle
      iii. Then connect each literal to its negation in a differing triangle
   (e) Proof:
      i. If S is independent set of size k, then S must contain exactly 1 vertex from each of the "triangles" in the graph to equal k.
      ii. Set each of the literals in the triangle for the set S to true, with other correpsonding edge nodes set accordingly. There then is a satisfied 3-SAT of clauses.
      iii. Then truth assignment is maintained, and the Independent Set contains k nodes that are completely independent of one another simply by selecting a true literal from each "triangle".
10. So, after all that, we've shown three strategies for reductions, and we find a new property available: **Transitivity**
11. Reductions are **transitive**: $X \leq_p Y$ -> $Y \leq_p Z$ -> then $X \leq_p Z$
12. So, from all we've seen above: (3-SAT $\leq_p$ INDEPENDENT-SET $\leq_p$ VERTEX-COVER $\leq_p$ SET-COVER)
13. Additionally, consider the following of **Self Reducibility**:
   (a) **Decision Problem**: Does there exist a Vertex Cover of size $\leq k$
   (b) **Search Problem**: Find vertex cover of minimum cardinality
   (c) **Self Reducibility**: Search Problem $\leq_p$ Decision Problem
14. This concept applies to all NP-Complete problems we are learning, and is the reason for focusing on the Decision version of a problem. And now it enables us to explore the meaning of NP.
15. **Definition of NP**
   (a) An algorithm is meant to solve a problem in a certain amount of steps given some initial input and parameters by which to complete the problem.

(b) An algorithm is **polynomial** if it runs to completion in $f(SizeofProblemSpace)$ steps, such that the amount of steps for function f is a polynomial.

(c) By problem space, I mean the observable and experimental combinations and possible solutions of a given problem. This varies drastically depending on a problem, for example the problem on the powerpoint used a string as input, and was examining whether this string exists in a set of strings. The problem space was then the length of the string it needed to examine in order to tell whether it belonged in the set or not.

(d) Now, a **Certifier** in this case is simply a way of verifying the existence and correctness of a solution that is polynomial. The powerpoint refences the same problem, but makes the remark that the string should be "certified" by a correctly assumed other string t that acts as a "witness" for the original string in the Set X.

(e) What a certifier essentially represents is the way to verify that a solution is correct and works for a given problem.

(f) And this brings us to **NP** problems: A problem is NP IF it is a decision problem that has a poly-time certifier. Or in other words, you can observe with a certifier that there exists a solution computable in poylnomial time.

(g) NP stands for **Nondeterministic Polynomial** time.

(h) Here's the example from the powerpoint on certifiers and basic NP:

i. Given an integer, determine whether it is composite.

ii. Well, very easily, create a function that takes two integers, the integer to determine compositeness and another integer that is less than or equal to it and also positive.

iii. Then see if it is a multiple of the integer to determine compositeness, and the answer is contingent upon that result.

iv. The catch here is that, without the secondary integer to determine compositeness, finding the compositeness of a given integer takes extremely long and is difficult to do.

v. But the secondary integer verifies it is possible to determine the compositeness, given a special factor that helps determine the solution.

vi. This secondary integer, thus, acts as a certifier. It determines whether the first integer belongs in the Composite "set". And this will hold true for any case of an intesger to determine compositeness and a predetermined secondary integer.

(i) This idea of certificates allows NP algorithms to be explored and determined.

(j) **Hamiltonian Cycle**

i. Given an undirected graph, determine whether a simple cycle exists in the graph that visits every node. The cycle should touch each node once during its path back to the source node (whever it starts from).

ii. The certificate is just a permutation of the nodes, an ordered arrangement of the nodes (pick one)

iii. Then check if the Set of nodes in the graph contains each node in the permutation exactly once, and there is an edge between any two adjacent pair of nodes.

iv. This is possible to do, but also shows that it requires determining permutations, then determining edge placement. This is clearly an NP problem as it will take an undertermined amount of time that may be polynomial, but extremely high.

(k) With this in mind, it's important to remember that the certifier is just telling us that there exists a polynomial time capabilityof solving these problems, not that we actually have an algorithm for it.

(l) There is a big difference between P, EXP, and NP:

i. P -> Decision problems with polynomial time algorithms

ii. EXP -> Decision problems with exponential time algorithms (those are really bad by the way)

iii. NP -> Decision problems for which we have polynomial time certifiers.

iv. We can be sure that $P \subset NP$ and that $NP \subset EXP$

v. Why? Think of the time of each type. P is polynomial. NP is what we think can be polynomial. EXP is exponential. Polynomial can operate within the bounds of NP, and NP can operate within the bounds of EXP.

vi. The proofs on the same slide are a tiny bit more pedantic but essentially say the same thing.

(m) But wait a minute....wait just a minute....

(n) If NP are really just polynomial time algorithms, then doesn't that mean....

(o) **P = NP??**

(p) Oh boy, now you've done it. Inb4 some math geeks come running at you with calculators and some notes of proofs.

(q) There happens to be a ***MILLION DOLLAR PRIZE*** for proving that P = NP (Decision Problems of polynomial time = Certificate problems)

(r) If P = NP were ever to be proven, then it would prove there are efficient algorithms for things like Factor, 3-SAT, Traveling Saleman and the like of NP Hard and NP Complete.

(s) Inversely, if P = NP was proven **not** true, then none of these problems could ever possibly have efficient algorithms.

(t) There isn't any consensus on the matter and its still causing frustration for plenty of mathematicians and CS out there, endlessly tinkering away with their calculators and scribbling their proofs....how sad....

16. **NP-Completeness**

(a) **Polynomial Transformations**

i. A problem X **polynomial reduces** to problem Y if X can be solved as Y, and done so in polynomial amount of steps with overall polynomial operations.

ii. A problem X **polynomial transforms** to problem Y if for a solvable instance of X, we can likewise construct a solvable instance for Y. Specifically, an instance of X can **only** be solvable IFF the instance for Y **is** solvable. IF Y's instance, call it y, does not solve for Y, then X also will not be solved by its own instance.

iii. The difference between the first and second is this: The first reduction may have polynomial references to the type of Y problem in its quest to solve X. The second reduction may have only one reference, at the very end, to Y in order to solve X.

(b) A problem is **NP Complete** iff for all other NP problems, it can be reduced to them and they to it.

(c) That is, if a problem Y is in NP, then for every other problem X in NP, it holds that $X \leq_p Y$

(d) This also strengthen the proposition of P = NP. If P = NP, then Y is solvable in polynomial time (because its in NP). And if Y can be solved in polynomial time, then all other NP-Complete problems X can also be solved as such, so NP $\subset$ P. And we already know P $\subset$ NP, so this would imply P = NP.

(e) P = NP is so weird.

(f) Now consider an example NP Complete problem:

(g) **Circuit SAT**

i. Given a circuit built of AND, OR, and NOT gates that can have differing combinations of them, is there a way to set the inputs to generate a 1 output. (Circuit operates in 0/1 bits)

ii. Yup, extremely easy to check. Given a circuit, you can simply treat it as a tree with different combinations for gates as paths in the tree. Then just pass in the inputs.

iii. But this is given an already formulated circuit, so it's less about constructing the circuit and more checking whether an input works.

iv. Hence why it is NP Complete.

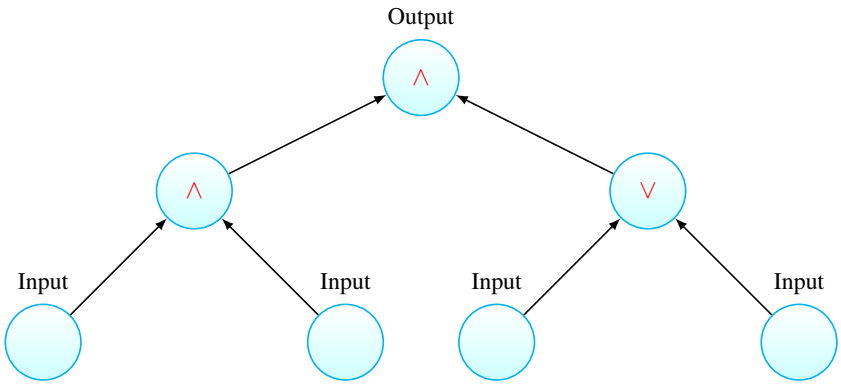
v. But we can assert that any algorithm that takes in amount of bits and produces a yes/no can be constructed as a circuit of some kind.

vi. Meaning you can make a tree of different gate combinations in each path and provide an ultimate answer.

vii. And the circuit tree will be the same size as the algorithm's time complexity.

viii. Little note: the leafs of the tree should have edges directed at the parent nodes, and this should go all the way to root per level. So instead of a root node pointing down to to leaf nodes, the two leaf nodes point up to the root. That way the input are the leaves, and the circuit traces back to the root.

ix. Here is what a tree of a circuit would look like:



x. We can easily check whether an instance like 1101 would work.

xi. The proof on the slides is a little awkward, but it essentially says the following:

A. An algorithm that needs a yes/no answer determine by bits can always be a circuit. Why? Well you can just use NOTs, ANDs and ORs to get the same exact result you want given any input. If the input is 1 bit (and the bit is set to 1), but you want an answer where the bit is set 0, then just make a circuit that has the negation of this bit. And vice versa and just about anything in between will work the same for any input desiring any result.

B. Because we know this, we can just turn a likewise polynomial algorithm into a circuit. And if we do that, the circuit is guaranteed to be as big as the polynomial problem space.

C. And then there is a subtle point about using a certifier: The certifier has to be polynomial steps long. This is to ensure the ciruit is operating on a polynomial problem space in polynomial time (If its NP).

D. That part is awkward and somewhat vexing to explain, but basically, if the certifier is polynomial steps long, that will ensure the paths with a verified solution from an input will operate in polynomial operations. I think.

E. Don't get too hung up on the proof, it's more or less just saying "you can turn a bit algorithm into a circuit sat and just make sure it is as big as it runs and boom you've got it" (This also proves its NP Hard, but we'll get to that later)

(h) **Establishing NP Completeness of a problem** Y

i. Show that Y is in NP.

ii. Choose another NP Complete problem X

iii. Prove that $X \leq_p Y$

iv. IF you prove this, then Y is NP-Complete because all NP Complete problems can reduce to one another and transform to one another.

(i) For example, we can show that **3-SAT** $\leq_p$ **Circuit SAT**

i. Because Circuit SAT is NP-Complete, we can assert that Circuit SAT $\leq_p$ 3-SAT.

ii. And we know 3-SAT is in NP.

iii. So, to make it into a circuit problem, establish each circuit gate node in the tree as a SAT clause, and instantiate the clause to represent the gate at the node.

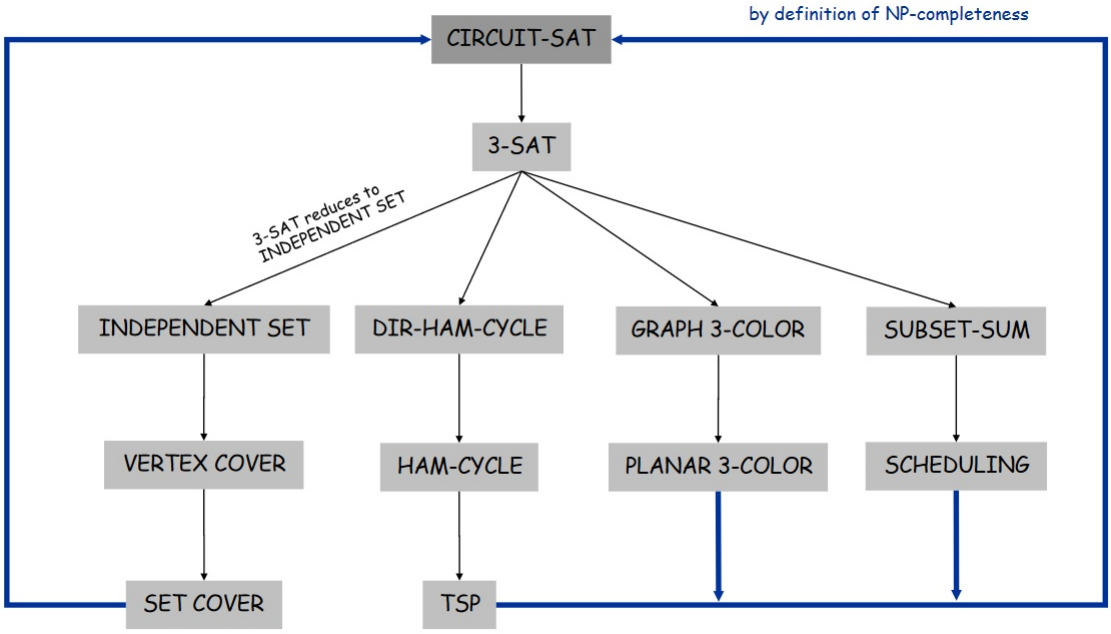
iv. Be aware of the clauses that are "below" this node in the tree, so that each instation of the clauses works.

v. Hard code an output requirement of 1 and an input requirement at the last clause as 0.

vi. And then make sure any clause that did not have 3 variables of the form x_i are elongated or compressed to exactly 3 in length.

vii. And voila, you have a circuit SAT that works exactly for the 3-SAT, meaning 3-SAT $\leq_p$ Circuit SAT, and Circuit SAT is NP Complete, so 3-SAT must be NP Complete also.

(j) Here's a great picture from the slides that show a series of NP Complete problems that reduce to one another:



(k) There are six genres of NP Complete Problems:

i. Packing problems (Set-Packing, Independent Set)
ii. Covering problems (Set-Cover, Vertex-Cover)
iii. Constraint Satisfaction problems (SAT, 3-SAT)
iv. Sequencing problems (Hamiltonian Cycle, Traveling Salesperson)
v. Partitioning problems (3D-Matching, 3-Color)
vi. Numerical problems (Subset Sum, Knapsack)

(l) There are some exceptions, like Factoring, Nash Equilibrium in that they don't belong to those genres.

(m) NP Completeness is pretty huge, the biggest thing that enables CS to branch into other disciplines like Mathematics and Economics and so forth.

(n) And there are a **ton** of different fields by which it is critically important. I encourage you to do some sleuthing of your own to see what can be explored.

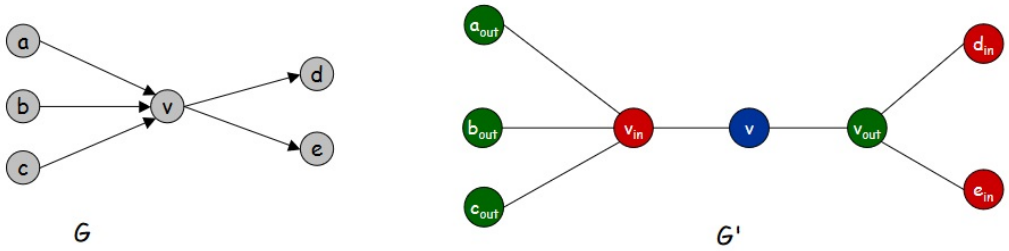
17. So far we've seen Packing, Covering, and Satisfaction NP Problems. Thus, let's look at some more.

18. **Sequencing Problems**

(a) **Directed Hamiltonian Cycle**

i. We touched upon it slightly before, but let's revisit it: A **Hamiltonian Cycle** is a simple cycle (touches a node on its path only once) that contains every node in the graph.
ii. We know a Hamiltonian Cycle is in NP.
iii. But what about a **Directed** Hamiltonian Cycle? Well, it's actually very easy to show that Ham-Cycle $\leq_p$ Dir-Ham-Cycle
iv. Simply expand the directed graph into an undirected graph by using "mediator" nodes in between cuts of nodes. By doing so, you constrain the pathway and check if each node is touched only once and every node is in the cycle.
v. Think of it like this: If there are 3 nodes with edges all pointing at one node, then put in a middle man node between the 3 nodes and the 1 node being pointed to. Then make it undirected between them all. The edges from the 3 nodes should funnel into the middle man node, which should then connect to the other single node, and vice versa.

vi. It looks like this:



vii. Finding that a undirected cycle is easy from this, just parse over the nodes as you would with the original Ham-Cycle algorithm.

viii. Finding/proving that the directed portion works is also simple, because reverting back from undirected edges implies that there must have been an original Ham-Cycle in the first place.

(b) **3-SAT reduces to Directed Hamiltonian Cycle**

i. This is not easy, so bear with me.

ii. IF a SAT is satisfiable, then construct a Dir-Ham

iii. Construct a graph with 2^n Hamiltonian cycles (seriously) such that they map 1-to-1 with truth assignments.

iv. Forma graph with a specific source node s, have all combinations of clauses within the middle of the graph between s and a target node at the very end t that will loop back to s for the cycle.

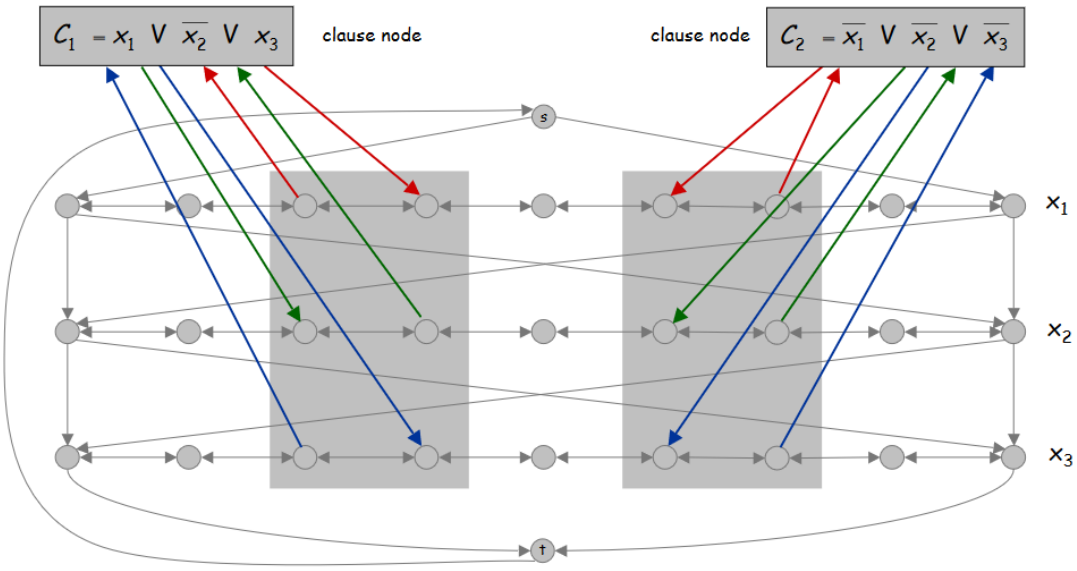
v. Make the possible combinations of truth clauses in the middle of the graph **undirected among edges of the same level**.

vi. Each level represents the "state" of each of the literals in a clause. So the first level can be all relative to the literal x_1.

vii. Then have nodes that exist outside of this portion that specifically relate to the Clauses themselves. They can act as "placeholders" while the horizontal edges on a level of the graph are parsed. There should be 6 edges in total (3-SAT). This is done specifically not to violate other clauses.

viii. IF by the end of the cycles' circulation there is a satisfied truth assignment that encroaches upon the original graph as directed (with each node touched only once) then huzzah.

ix. It looks like this (from the slides):



x. I told you working with 3-SAT is terrible. I recommend reading this part over and studying what's actually going on. The first time through doesn't make much sense.

xi. Now time for the fun part – the Proof:

A. The 3-SAT portion (right direction) is relatively simple:

B. Traversing a row on the Hamiltonian Graph, if a cycle exists, will have either a value of 1 going from left to right, or a value of 0 going from right to left.

C. This means, each Clause in the graph will have at least 1 row between its 3 literals such that a value can be returned in the correct direction. (If you need a 0, you can get a 0, and if you need a 1, you can get a 1, all based on direction).

D. The Cycle portion (left direction) is harder:

E. So, because each row for a clause literal has an outer bound series of "nodes" that it has to go to and from as placeholders, we can assert that if there IS a cycle, the path has to go to these Clause nodes outside of the row, then come back on a different edge (its next edge back from the Clause), and resume at the next node. That means, if we just removed these edges that parse out to the clause, or at the very least normalize their path such that it is basically the same as going from one node to the next node adjacent to it, we can keep the path to only the row.

F. Then if that is the case, we can traverse a row from left to right and right to left, while maintaining a path that cycles around.

G. And as the 3-SAT portion of the proof shows, going in a completed direction on a row guarantees a result will be yielded.

H. So the cycle must exist.

xii. Now breathe. If that was very confusing, I don't blame you. I would recommend re-reading the proofs and comparing them with the big graph picture from above, and carefully consider what's happening when the path traverses left and right in a row.

(c) **Longest Path**

i. Its name basically says it all. Longest path while touching nodes only once. Like the Shortest Path algorithm, but harder.

ii. We can assert that 3-SAT $\leq_p$ Longest Path

iii. Oh god not 3-SAT not again please.

iv. Relax, it's easier this time: First, redo the graph for Directed Hamiltonian Cycle without the edge looping around from the end node to the start node (so as to make sure the cycle doesn't occur)

v. Then, just show that Ham-Cycle $\leq_p$ Longest Path, which is as simple as "A Hamiltonian Cycle has to touch every node, so it has to go over as many edges as possible to do so, and because we remove the cycle edge that loop it back around, its just the longest node from a source to a target"

vi. That wasn't so bad, now was it?

(d) **Traveling Salesperson**

i. Given a set of citites, is there a path that travels to each city only once and gets back to the original city by the end that is as shortest as possible?

ii. Essentially just the shortest path from a source to a target that has to touch every node and has to cycle back to the source. No biggie.

iii. Well, we know that Ham-Cycle also has to touch every node and be a cycle...

iv. So we can assert Ham-Cycle $\leq_p$ TSP

v. Proof: With a Ham-Cycle Graph, create "cities" at nodes with a distance function that is given 1 if the edge between "cities" is in the Ham-Cycle and 2 if the edge between "cities" is not in the Ham-Cycle.

vi. You are then guaranteed to touch every node and get the "shortest" path distance in the Graph. This will be $\leq$ to the number of nodes in the graph. *Cough* Excuse me, I meant "cities."

19. **Partitioning Problems**

(a) **3D-Matching**

i. Given n of instructors, n of courses, n of times and the instructors are clearly defined for what classes they'll teach: Can you set it up so all classes are taught at different times?

ii. So essentially, instructors and courses and times are all disjoint sets of the same size.

iii. What the final set represents is a series **triples**, n of them (the same amount as each set). A triple takes the form of [Instructor | Course | Time]

iv. So the question is asking: Can you make a sechdule where each one of these triples is a **unique** combination when the set of instructos, courses and times are all put in one big bag (Instructors∪Courses∪Times). There can't be any two triples that share, for example, the same instructor at two different times, in the final set.

v. Hmm....sounds like you are working towards a solution that is a **set** of **independent clauses**....

vi. Almost like....you want an **Independent Set** based around the truth evaluation from a **3-SAT**

vii. Oh wait. (Yeah ok, I get the snarky sarcasm you can stop now) Sorry, anyways we can show 3-SAT $\leq_p$ Independent Set $\leq_p$ 3D-Matching

viii. 3-SAT again....this one is a doozy:

A. First, create a gadget for each variable x_i (of the 3-SAT form) with "2k core and tip elements"

B. "What's a 2k core? And tip elements? Are they expensive?"

C. "2k core" means that the alternate "clause literals" for a clause are within the gadget edge. A tip element is outstanding truth value for a literal that will be "chosen" by the algorithm

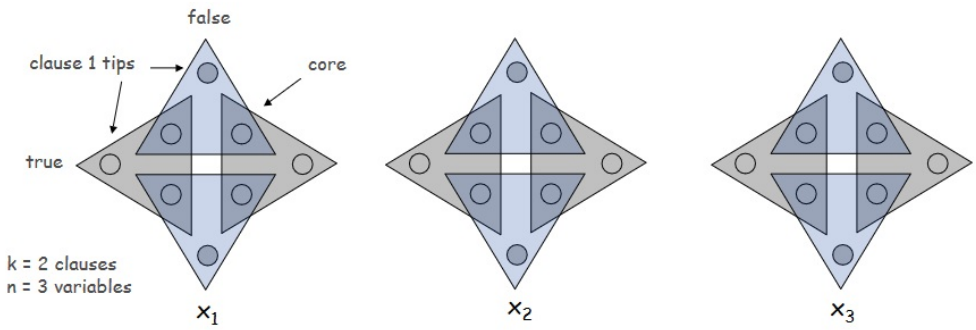
D. So think of it more this way: If k = 2, then that means each gadget should take the form of a diamond, with 4 triangles conjoined together at the base.

E. The "core" of this diamond is dependent on each triangle. Since the triangle base will comprise of 2 elements, the total core will be 4 (2k -> k = 2 -> 2k = 4)

F. This also means there will be 4 tip elements that represent the chosable elements for truth values. The triangles essentially are created as the triples from the 3D-Matching.

G. Also, with the diamond formation, we need to split the color of the triangles in halves: 2 of them red, and 2 of them blue. Why? Because you can only solve the 3D-Matching if the truth values are taken from likewise colored triangles.

H. Observe the following picture from the slides, it adheres to the description above:



I. Notice specifically the orientation of the tips. Each one of them is independent of each other triangle.

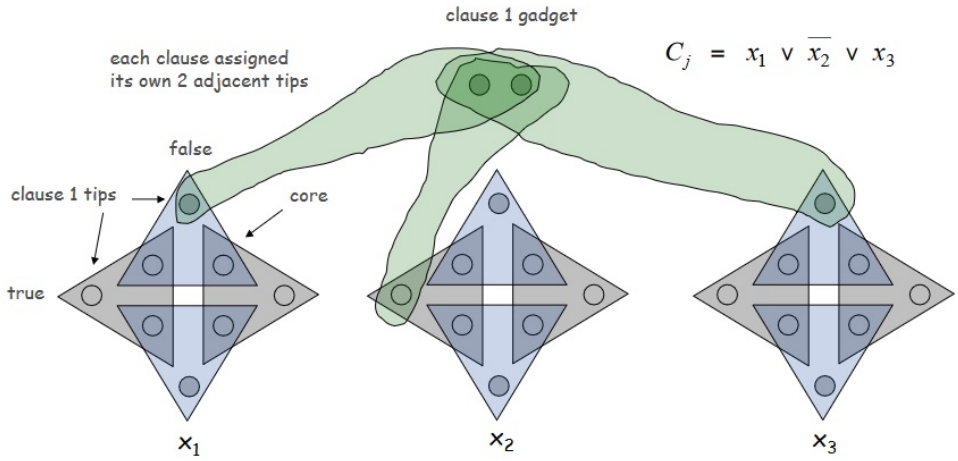
J. However, the elements in the base of the triangles **are not**. They are double counted by **different colored triangles**

K. But remember, 3D-Matching should have no overlap while everything is accounted for. So to account for all 4 elements in the "inner square" of the diamond, you can only select two triangles of the same color.

L. Now, here's the leap: We need to formulate the actual clauses for the 3-SAT. But how? By linking together tips from different diamonds with 2 other truth values that are completely independent of all the diamonds.

M. This is a gadget, done specifically to verify a Clause for the 3-SAT.

N. Here's a picture first, and then I will elaborate:



O. Now, notice the two dots in the three-times-overlapped green area? Those elements are generated as an independent clause from 3-SAT.

P. x_1 provides this clause with a TRUE element, which returns a 1 with the clause as is.

Q. x_2 provides this clause with a FALSE element, which returns a 2 with the clause. SO, the clause then negates the value from x_2

R. x_3 provides this clause with a TRUE element, which returns 1 with the clause as is.

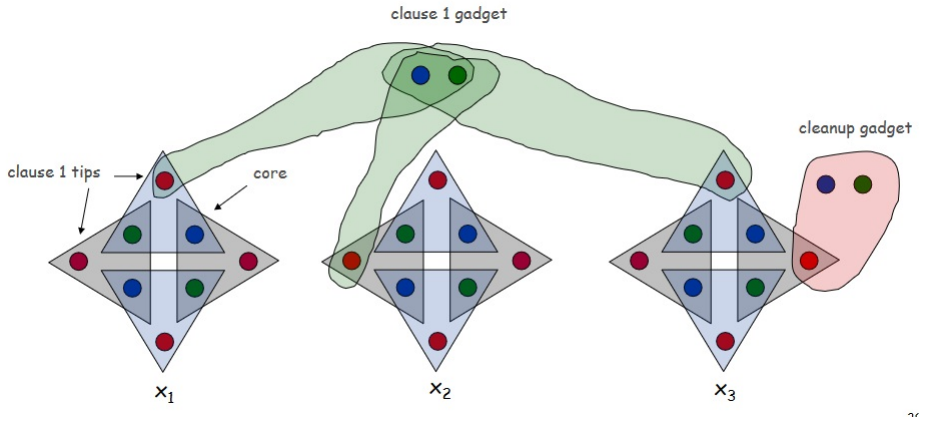
S. So we have a satisfiable clause for 3-SAT.

T. But there's one last thing: There are a bunch of tip elements that are not tracked in the clause.

U. Well now what do we do? We provide another gadget, a "Garbage Collector", that cleans up these tips by grouping them also with 2 independent elements. That way 3D-Matching in the non-Independent Set are satisfied.

V. Its another extension just like the green one above, but you can make it exclusive, however many times, to each tip element not covered by the green clause.

W. And so, finally:



ix. That was a weird one, so if you feel a bit confused don't worry. But this seems like a very important example of how to properly use gadgets for an NP problem, so implore you to read it over and study it carefully.

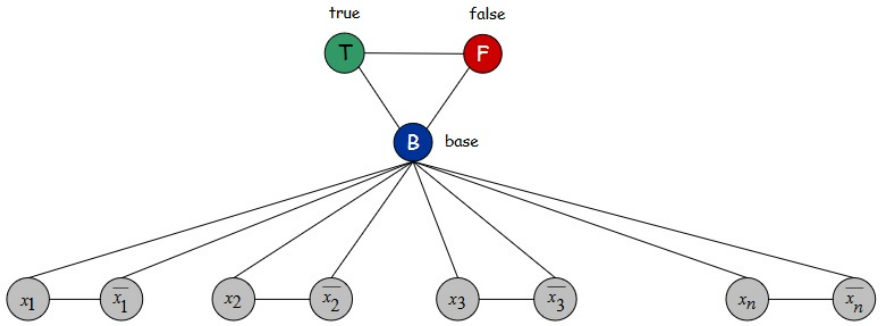
(b) **Graph Coloring - 3-Colorability** (Remember to bring your crayons)

i. Given an undirected graph, can you color the nodes RGB so that no two adjacent nodes have the same color?

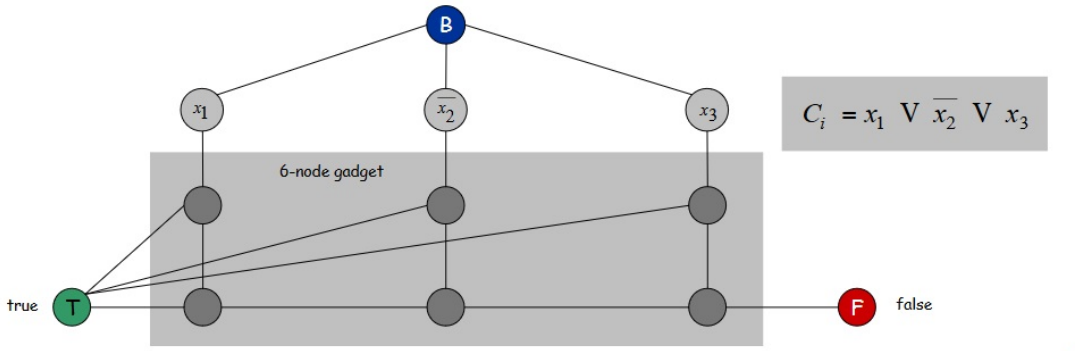
ii. This relates to **Register Allocation** –> Given a machine with registers, assign programs so that no more than k registers are used and no two programs needed at the same time are assigned to the same register.

iii. You can build and **Interference Graph** that basically connects nodes as programs and edges indicate two programs are "active" together.

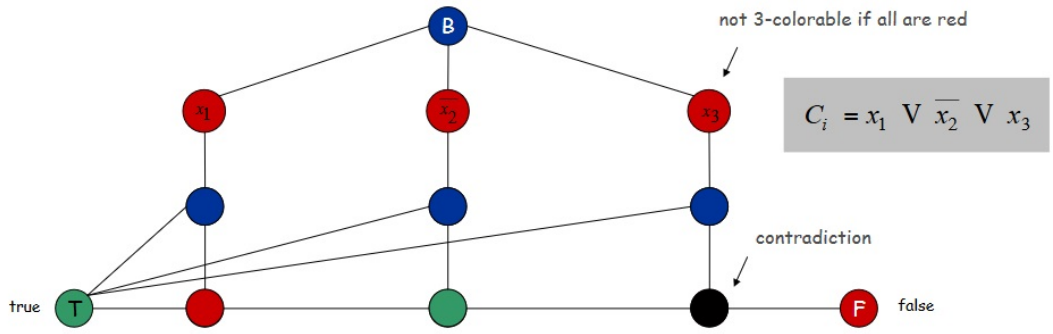
iv. This is so that you can assert 3-Color $\leq_p$ Register Allocation.
v. Anyhow, back to 3-Color
vi. To show 3-Color we can...show 3-SAT $\leq_p$ 3-Color
vii. God I hate 3-SAT. Well, here we go again:
   A. Make a node for every literal.
   B. Make 3 nodes labeled T (True), F (False), and B (Base).
   C. Make negation nodes for every literal node. Connect the literal nodes to their negation nodes.
   D. Connect the TFB in a triangle.
   E. Connect all literal nodes and their negation nodes to the Base node B.
   F. Again I'm just going to use the slide pictures for this because they're really good:



   G. Via this picture we are absolutely sure all nodes can be alternate colors and the literals can be T/F
   H. Now, to connect to 3-SAT in the most painful way imaginable, add 6 nodes and 13 edges to each Clause. Seriously.



   I. Once again, straight from this picture we can see that all nodes can be alternatively colored, literals will be guaranteed T/F, and we will be ensured a truth value in the clause.
   J. Because, if you color all the literals as false, you cannot getting alternation in color nor a truth for the 3-SAT.



   K. And that basically finishes this. Not nearly as complicated as it could have been, but from an explanation standpoint its hard to describe in words.

L. I would recommend near memorizing what the pictures are like, or at the very least understand explicitly setting them up.

(c) These NP problems are more about the pictures. Their proofs are far more comprehensive via the pictures than the words, be careful to go over the pictures and study them.

20. **Numerical Problems**

(a) **Subset Sum**

i. Give n set of numbers, and a value W, is there a subset of the numbers that adds up to W?

ii. The set of numbers can be all natural numbers by the way. And W can be an countable number likewise.

iii. We use binary arithmetic for this so the polynomial reduction has to be with binary encoding.

iv. "What reduction do we use?" you naively ask. I then respond, "Well, with the wonderful incarnation of satan of course. 3-SAT!"

v. 3-SAT $\leq_p$ Subset Sum for the millionth time.

vi. Given a 3-SAT instance, with n variables and k clauses, form $2 * n + 2 * k$ decimal integers, each of $n + k$ digits in length.

vii. So if there are 3 literals, and 3 clauses, then the integers should be 6 digits long.

viii. This process can "create" digits by operating in binary and setting bits according to the literals and the clauses.

ix. That is, if the table is [x1 | x2 | x3 | C1 | C2 | C3], then a binary value can be assigned to each variable in according with an integer (byte size per each of the 6 variables)

x. Here's a table from the powerpoint to simplify this:

$C_1 = \bar{x} \vee y \vee z$
$C_2 = x \vee \bar{y} \vee z$
$C_3 = \bar{x} \vee \bar{y} \vee \bar{z}$

	x	y	z	C_1	C_2	C_3	
x	1	0	0	0	1	0	100,010
¬ x	1	0	0	1	0	1	100,101
y	0	1	0	1	0	0	10,100
¬ y	0	1	0	0	1	1	10,011
z	0	0	1	1	1	0	1,110
¬ z	0	0	1	0	0	1	1,001
dummies to get clause columns to sum to 4	0	0	0	1	0	0	100
	0	0	0	2	0	0	200
	0	0	0	0	1	0	10
	0	0	0	0	2	0	20
	0	0	0	0	0	1	1
	0	0	0	0	0	2	2
W	1	1	1	4	4	4	111,444

xi. That's essentially its own proof, the logic of it is intuitive although it may not seem so at first. After a second or third glance at it, it becomes pretty obvious what's ocurring.

(b) **Scheduling (With Release Times)**

i. Given a set of jobs (n jobs), each with a processing time, a release time, and a deadline, it is possible to schedule **all** jobs on one machine such that each job runs from release to deadline during its process time without interuption?

ii. So, to reduce it we use...not 3-SAT? Oh thank goodness.

iii. Subset Sum $\leq_p$ Schedule w/ Release Times (Although, because 3-SAT $\leq_p$ Subset Sum, by transitive property 3-SAT $\leq_p$ Schedule. You can go ahead do that yourself if you'd like :D)

iv. Given a Subset Sum with a set of integers of $w_1 ... w_n$ and a target integer W, create n jobs with a processing time $t_i = w_i$, release times of 0, and no preemptive deadlines.

v. The deadlines just become the processing time + the last point where the prior job finished.

vi. The first job, job_0, should have processing time of 1, a release time of W, and a deadline of (processing + release = $W + 1$).

vii. Now, the places to put jobs are: Anywhere from 0 to W and $W + 1$ to the end of the schedule.

viii. And then this continues in likewise D&Q type fashion.

(c) So numerical NP problems seem to be about how to represent the integers or ordering of elements/jobs. And specifically how they relate to their reduction.

21. Just about done with the chapter, but a last few small things:

22. We **will continue to talk about NP problems** in the following chapters, but in different contexts and more about the problem rather than the reduction/how it's NP.

23. The biggest takeaway of NP is that reductions are all about showing that the Problem on the Left is just a solvable case of the Problem on the Right. The proofs and examples are specifically to show how reductions work. But understanding the overall relevance of a reduction is the most important thing, not just the proofs.

24. And I have one final spew about NP, pertaining to NP Completeness

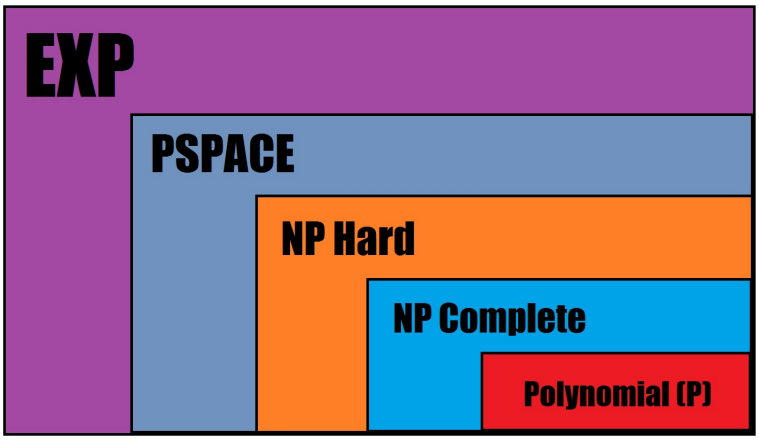
25. **NP Hard vs NP Complete**

    (a) **NP Complete is a subset of NP Hard**

    (b) Here's a table:

NP-Hard	NP-Complete
Is in NP	Is in NP
NP Based A exists where $A \leq_p B$	All NP Complete A exist where $A \leq_p B$
Do not always reduce to one another	All NP Complete reduce to other NP Complete

    (c) Here's a picture of how the space for NP works.



    (d) Please note: There are a gazillion more "problem spaces" that are all over the place and I do not have the time nor am insane enough to make an image to go that in-depth. We aren't really even covering PSPACE or EXP.

26. And one last tiny thing: NP Complete problems do have a "hierarchy" of sorts. Specificaly, 3-SAT reduces to **every other freaking NP Complete** problem, so when in doubt, you can always go back to 3-SAT for an NP Complete reduction.

27. And that ends the chapter for NP and Intractibilty. If you survived, then congratulations! Here's your warm milk and cookie, enjoy your comfortable sleep tonight. If you didn't, well fear not, others will dwell with you in the darkness of Niflheim.

Local Search

1. So you have an NP Hard problem, huh? And you won't to solve it? Well, theory isn't really on your side, now is it? Going to be pretty hard to solve in properly, in a timely fashion, at any given time.

2. But you've got to solve it, huh? Well, then, you can do so, but **at a cost**

3. This chapter is about solving really hard problems in the best feasible way (as of now) possible. And these are **Heuristics**: techniques for formulating algorithms for difficult problems in the best way possible. Specifically, these will be covering algorithms that forfeit optimality in favor of completion.

4. But you must forefeit one of the following:

   (a) Solving for the optimal solution.

   (b) Solving in polynomial time.

   (c) Solving any arbitrary instance of a problem (i.e. making your algorithm as general as possible)

5. **Landscape of an Optimization Problem**

   (a) **Gradient Descent: Vertex Cover**

      i. You want to solve an NP problem, Vertex Cover

      ii. An "algorithm" for doing so would **Gradient Descent**

      iii. Before this, remember what vertex cover is – a subset of nodes in a graph that touches every edge and is minimized.

      iv. And a **neighbor relation** is another set of nodes that is 1 greater or 1 less than the Vertex Cover set (delete or add a node)

      v. So with those defined, a **Gradient Descent** is very simple: Initiate the "vertex cover" set to be all nodes in the graph.

      vi. Then check if there is a neighboring set that is both 1) smaller and 2) is a vertex cover

      vii. If so, replace the vertex cover set with this neighbor set.

      viii. Think of it like this: There are 5 nodes in the full set. There's a neighbor that is 4 nodes and is a vertex cover. The new full set is the 4 nodes. And this process can continue until, at worst, every node has been deleted (n steps, which is the total number of nodes)

      ix. This algorithm somwhat forfeits optimality, because the optimal solution might be an odd orientation where the strict neighbors doesn't account for.

      x. Say if you have a line of nodes, then every other node would be the optimal solution because every other node would touch both edges incoming and outgoing.

      xi. Thus, this is a **Local Search**: an algorithm that explores possible outcomes by finding the "next best/nearby" solution to the current one in sequential fashion.

      xii. In the context of gradient descent: If there is a current working solution, check if there is neighbor solution with a lower cost, and replace the current solution with the smallest possible of lower costs. Else terminate.

6. **Metropolis Algorithm**

   (a) Physical energy states (low and high), with a biased operation for downhill (read: easier) steps, but can do uphill.

   (b) Algorithm is all about maintaining a given state, by feeding it "easy" operations most of the time and occasionally "hard" operations to offset it.

   (c) The right amount of time and states will be found by the end of execution.

   (d) This concept of balancing Energy states with lows and highs lends to the construct of **Simulated Annealing**

   (e) **Simulated Annealing**

      i. T Large => probability of a "hard" move is large.

      ii. T Small => probability of a "hard" move is small.

      iii. Idea: Find a good balance of T Large and T Small to maintain a given input state.

      iv. Cooling Schedule: Every iteration of some element or space, T gets switched.

   (f) The breakdown is like this: Take a solid and melt it really quickly, then take a molten solid and freeze it really quickly.

   (g) And then Annealing: Gradually cool from high temperatures so that the structure evens out / reaches equilibrium.

7. **Hopfield Neural Networks**

   (a) Simulating an associative memory: neighbors in a network try and correlate/assume the states of one another.

   (b) The graph configuration is simple: given a graph with integer weights, add nodes with weights that are either positive or negative 1.

   (c) If the edge weight between nodes is negative, then the nodes should be the same "state". If the edge weight between nodes is positive, then the states should be different.

   (d) An edge is **good** if the edge weight is negative and the nodes share states.

   (e) An edge is **bad** if the edge weight is negative and the nodes differ in states.

   (f) This also works with positive edge weights (good if differ in states, bad if same state).

   (g) A node is **satisifed** if the sum of good edges (from the node) is greater than the sum of bad edges (from the node).

   (h) **State Flipping Algorithm**: Repeatedly flip the state of an unsatisfied node. (While the graph is unsatisfied, go through nodes and flip their states)

   (i) This algorithm will operate in the space of the total amount of edges in the graph.

   (j) Proof:

      i. Let S be the number of satisfied nodes.

      ii. Let W be the total amount of edge.

      iii. Obviously $0 \leq S \leq W$ because worst case all nodes are unsatisfied and there will still be less than or equal to the total edges.

      iv. After each flip: All good and bad edges that **touch the node** flip (good becomes bad and vice versa). All other edges and nodes in the graph maintain their weight and states.

      v. We can show that: (S - Sum of good edges + Sum of bad edges) is $\geq S + 1$. The inequality represents the existence of unsatisfied nodes.

   (k) Notice the distinction between what the original probem is asking:

      i. Search Problem: Given a weighted graph, find a configuration for the Hopfield Neural Network.

      ii. Decision Problem: Given a weighted graph, is there a stable configuration?

   (l) The decision portion is polynomial solvable, but the search problem is not known.

8. **Maximum Cut**

   (a) Given a weighted graph with positive integer edges, find a cut of nodes where the flow/sum of weights is maximized.

   (b) There are some practical applications for this (Person-Activity scheduling, Circuit layout, statistical physics)

   (c) Algorithm:

      i. Greedily pick a random partition of nodes into segment A and segment B.

      ii. Continually choose nodes from each segment and swap the node into the other segment to maximize/improve the parition weight.

      iii. Once the best possible weight is achieved from the cut, return the segments.

   (d) The proof is awkward, so here's a verbal explanation:

      i. The locally optimal solution (that we found) is greater than or equal to half the sum of all edges weights.

      ii. We know that with local solutions we forfeit some optimality, so the sum of segment A is $\leq$ than segment B.

      iii. This means that the sum of segment A is $\leq$ the local optimal solution. ($w(A, B)$)

      iv. And we also know that the sum of segment B is $\leq$ the local optimal solution. ($w(A, B)$)

      v. So then [the sum of segment A] + [the sum of segment B] + [the local optimal solution ($A + B$)] is leq the local optimal solution times two. $2 * w(A, B)$

vi. But we know that $2 * w(A, B)$ is the sum of all edges anyhow (the parition is a cut of half the edges essentially), so we have the best local optimal solution.

(e) Local optimal solution versus the universal optimal solution (which is too hard to find) is an important distinction here.

(f) The time complexity for this local search version is still not polynomial completion, however.

(g) Using **Big Improvement Flips** will provide a better time.

(h) The algorithm is the same as the local search version, but it only swaps nodes between segments if it improves the solution $w(A, B)$ by $2\varepsilon/n$ where n is the number of nodes. (ε is an arbitrary increase by some value)

(i) The proof is also the same, just with $2\varepsilon/n$ attached to each sum of segments.

(j) This algorithm terminates in $\varepsilon * n * log($Sum of edge weights$)$

(k) There are 50% and 87% approximation algorithms for Max-Cut, but if P != NP then there is no algorithm better than 95% approximation.

9. More on **Neighbor Relations**: We discussed **1-Flip**, but **k-flip** means essentially the same thing except that the nodes are k flips different, not just 1-flip. Θn^k

10. Implementing this in **KL-Neighborhood** format means marking "flipped" nodes.

11. Marking of nodes is extremely useful in practice and in theory, helping to avoid excess node flips and increase execution operations.

12. Now, the final problem of the chapter is **Nash Equilibria**

(a) **Multicast Routing** –> Directed graph with a source node, a series of other nodes that are path ends, and edges with integers that are ≥ 0.

(b) An **agent** j will find a path from the source node to a path end node t_j, paying the edge costs as it goes.

(c) Agents will always switch to the best path available for them depending on other agents.

(d) Agents pay the integer value of a path divided by how many other agents are using it.

(e) **Nash Equilibrium** –> Solution where agents have their best paths satisfied.

(f) Local Search approaches: Agents continually jump edges depending on other edges, looking for the best path possible for them.

(g) **Socially optimal** solutions are rendition of Nash Equilibria if the cost of all agents in minimized.

(h) There can be many form of the Nash Equilibria, and they aren't always socially optimal.

(i) Stabilizing an algorithm/problem between Nash Equilibria and Social Optimality comes at a price of $\Theta log(k)$

(j) The algorithm for Nash Equilibrium has basically been said: Pick a path for an agent, and while an Equilibria has not been reached, pick an agent for the path and switch it to a path that improves the solution.

(k) The proof is actually extremely intuitive (despite dense mathematical notation on the slides): Basically, by switching an agent to a new path, there's a new total cost less than the cost before it because a cheaper pather was switched to. This occurs for all agents until a Nash Equilibria is achieved.

(l) Of course, this is all based around countably infinite sized edges in cost. (back to the stabilizing issue)

(m) So the worst Nash Equilibria you can devise that is worse than the social opmtimum is at most the size of the countable agent space for the problem.

(n) Summary for Nash:

i. Existence: Nash Equilirbia will always exist for a routing problem about agents that prioritize sharing path lengths (finding the best possible for each)

ii. The best Nash is never worse than the social optimum by greater than the agent problem space size

iii. The hard part in all of this is finding a Nash Equilirbia in polynomial time.

13. And that is it for this chapter. The algorithms all base around the concept of local search – finding the next best thing to what you already have – and deriving agorithms for hard problems in doing so.

Approximation Algorithms

1. So much like the Local Search chapter, this chapter will also work on solving NP problems in reasonable time.
2. But unlike the Local Search Chapter, these **Approximation Algorithms** will:
   (a) Run in polynomial time
   (b) Solve any instance of the problem
   (c) Find a solution within a ration δ of true optimality (the solution ratio is ≤ 1). This part is weird because we may not even know what the true optimal solution is, so proving it is tough.
3. So these are really good and pretty important.
4. **Load Balancing**
   (a) A type of job scheduling problem with a requirement that all jobs get scheduled.
   (b) All about maximizing machine scheduling of the jobs.
   (c) It has the following variables:
      i. n jobs
      ii. m machines
      iii. t processing time, which is really just the space it takes up in the available space in the schedule
      iv. There is a **makespan**: a maximum load – amount of jobs you can assign to – any machine
   (d) Firstly, before covering it specifically, we should ask: is this problem in NP?
   (e) Recall back to the original job scheduling problem, in which you only scheduled the best possible jobs. It was solved Greedily by taking the shortest job available, and then inputting the next legal available job after it finished.
   (f) The approximation goal is essentially to minimize the makespan by assigning jobs accordingly.
   (g) The Approximation Algorithm:
      i. Fix some order of the given n jobs
      ii. Take each job sequentially.
      iii. Place each job into a machine with the least processing time (most space) available.
      iv. The algorithm's implementation runs in $O(n * log(n))$ time.
   (h) Before analyzing the algorithm, let's show a reduction:
   (i) **Subset Sum $\leq_p$ Load Balancing**
      i. IF there is a partition of the Subset Sum, then....
      ii. There is a solution of the Load Balancing if job L_1 = job L_2
      iii. The corresponding machines liken to the two sets of integers [$w_j = w_i$ <==> $L_j = L_i$]
   (j) Decision version of this problem: Given K, is there a schedule such that the makespan $\leq K$
   (k) K is identified as: $\frac{\sum_{i=0}(w_i)}{2}$
   (l) Side note: You must show the certificate that the problem is NP Hard/Complete. Remember a certificate is just a way of showing it can or can't work.
   (m) Analysis of the Algorithm:
      i. Consider L' to be the optimal solution.
      ii. We should find lower bounds for it.
      iii. $L' \geq max_j * t_j$ – This works because some machine has to take on the biggest job
      iv. $L' \geq \frac{1}{m} * \sum_j t_j$ – This works via pigeonhole principle: at least one machine will be stuck with the job if there is a greater amount of jobs than machines.
      v. This means the greedy algorithm is 2-Approximation
   (n) Proof of 2-Approximation:
      i. Consider load L_i of a bottleneck machine i

ii. When job j is assigned to machine i, i had to have the smallest load. Its load before the assignment was $L_i - t_j$ which is $\leq L_k$ for all values where $1 \leq k \leq m$
iii. So we have to show $L_i \leq 2 * L'$ (this is a theorem)
iv. With the sum of the inequalities on machines, we then get $m * L_i - t_j \leq \sum_k L_k$, and we can move the m over to $\frac{1}{m} \sum kL_k$
v. This sum $\sum_k L_k$ is less than L'.
vi. So, $L_i - t_j \leq \frac{1}{m} \sum kL_k \leq L'$
vii. You can then logically connect that $\frac{1}{m} \sum kL_k + L' \leq L' + L'$
viii. And $L_i - t_j$ was based around $\sum kL_k$, so we can just use L_i
ix. Then finally: $L_i \leq 2 * L'$

(o) Point is, we used the instance of the last job to define a proper upperbound ($2 * L'$) that verifies the original issue in the proof (the theorem really)

(p) Be wary of tightening the analysis of an approximation

(q) A variant of this problem is instead to take the longest processing time by order, called the **LPT Rule**

(r) It also runs in the same time $O(n * log(n))$ and it is better because it is a $\frac{3}{2}$ Approximation

(s) Quick Proof of this new Approximation:
i. Theorem of: $L_i \leq \frac{3}{2} * L'$
ii. Order machines in decreasing process time.
iii. After m jobs scheduled, the last one has the shortest processing time (most space available).
iv. $m + 1$ is thereforegoing to be the smallest job across all existing jobs on the machines
v. The same math proof from above would work equally, with the simple diffence being the $\frac{3}{2}$ versus the 2

(t) This analysis isn't fully tight, but there is a basis there for tight analysis and algorithm completion.

5. **Center Selection**

(a) Finding the best possible center "site" such that the distance between all other sites in minimized.

(b) Best to think in terms of a Circle's area.

(c) Using the concept of dist(x, y) – which represent the distance between x and y – we can define the following:
i. dist(x, y) => 0 (identity)
ii. dist(x, y) => dist(y, x) (symmetry)
iii. dist(x, y) $\leq$ dist(x, z) + dist(z, y) (triangle inequality)

(d) dist(s_i, C) = min(c$\in$C) dist(s_i, c) = distance from s_i to closest center.

(e) (C) = $max_i dist(s_i, C)$ = smallest covering radius.

(f) So the end game is to find the smallest possible r(C), such that the cardinality of |C| => k

(g) Using random Euclidean points as centers can be infinite, so that's no good.

(h) Using a greedy approach by centering one as "best" and then adding accordingly is bad for arbitrary growth.

(i) Better Greedy Approach:
i. Set a center in the best possible space to begin
ii. Then repeatedly set centers farthest away from any existing center as possible.
iii. By the construction of the Algorithm, by the end it will be pairwise r(C)

(j) The proof is a contradiction that asserts that $r(C) \leq 2r(C*)$
i. By asserting that a radius $\frac{1}{2}r(C)$ around, then the pairing for centers in the circle
ii. Must be thus that the new pairing is less than what was there.

(k) Theorems are: $r(C) \leq 2r(C*)$ and Greedy Algorithm is 2-Approximation

(l) There is no hope of $\frac{3}{2}$ or $\frac{4}{3}$ approximation unless P = NP

6. Pricing Method: Vertex Cover
    (a) Find a Vertex cover with minimum weight.
    (b) Essentially, find each price such that it is ≥ 0 but also per edge such that the total price $\leq$ than the total weight.
    (c) Just find a Vertex Cover of the prices.
    (d) Another 2-Approximation, with verification of termination as there become fewer available nodes after a cover node is confirmed.
7. LP Rounding: Vertex Cover
    (a) With an undirected, weighted graph, find the minimum cover with each node has at least one edge incident in the cover.
    (b) Can use a binary representation for "in/out" of the Vertex cover.
    (c) So, by searching the 1-1 correspondence, you can maximize the weight with vertex that exist in the cover.
    (d) (Integer Programming) $\sum(w_i * x_i)$
        i. x_i => $0, 1$ $(i \int V)$
        ii. $x_i + x_j \geq (1)$ $(i, j \in V)$
    (e) If $x*$ is optimal in the **Integer Programming Formulation**, then S is a min weight vertex cover.
    (f) Happens that finding a representation given integer $a_{i,j}$ and b_i is a NP-Hard reducible process to Vertex Cover.
    (g) **Linear Programming** allows for both solvable and polynomial assertions with input of integers c_j, b_i, and $a_{i,j}$
    (h) The tradeoff is that the accurracy of the Linear Programming is less than that of the Integer Programming.
    (i) LP is also not equivalent to a Vertex Cover, unless we solve for rounded fractions.
    (j) All tolled, this is a 2-Approximation algorithm.
    (k) If P != NP, then p-approximation can be no less than 1.3607
8. **Load Balancing Reloaded**
    (a) Another walk through job scheduling problem
    (b) Set of m machines, and j jobs, with each job having a processing time t
    (c) The difference with the other Load Balancing is by what machines are **authorized** to take on a job.
    (d) All the proofs from a general standpoint are the same.
    (e) You can do the more specified authorization with IP Formulation and Linear Programming.

ILP formulation. x_{ij} = time machine i spends processing job j.

$$
\begin{array}{lllll}
(IP)\ \min & L & & & \\
\text{s.t.} & \sum_i x_{ij} & = & t_j & \text{for all } j \in J \\
& \sum_j x_{ij} & \leq & L & \text{for all } i \in M \\
& x_{ij} & \in & \{0, t_j\} & \text{for all } j \in J \text{ and } i \in M_j \\
& x_{ij} & = & 0 & \text{for all } j \in J \text{ and } i \notin M_j
\end{array}
$$

LP relaxation.

$$
\begin{array}{lllll}
(LP)\ \min & L & & & \\
\text{s.t.} & \sum_i x_{ij} & = & t_j & \text{for all } j \in J \\
& \sum_j x_{ij} & \leq & L & \text{for all } i \in M \\
& x_{ij} & \geq & 0 & \text{for all } j \in J \text{ and } i \in M_j \\
& x_{ij} & = & 0 & \text{for all } j \in J \text{ and } i \notin M_j
\end{array}
$$

(f) The **Lower Bound** formulation will be an optimal makespan via an acyclic graph construction.

(g) A **rounded** solution is also based off the graph formulation, verifying authorizedmachines because only positive values can be authorized.

(h) Also, another item is that if a job is a leaf in this acyclic graph and the machine is a parent, the job is a culmination of authroized processes to this job.

(i) The solution (rounded) is 2-Approximation because the sum of times of a leaf node (that is a job) is the sum of authorized job processes to that point. And that is $\leq$ the LP Relaxation that it is a minimized makespan.

(j) Conclusion:

i. Running time is the solution for Linear Programming in $m * n + 1$.

ii. Can solve this LP with flow techniques on a graph.

9. **Knapsack Problem**

(a) We talked about this before, you have a bag with weights and you have things you want to put in it, maximize the value you can put into it while staying in capacity.

(b) We tackled this with Dynamic Programming initially.

(c) We can analyze this via rounding and scaling in this chapter.

(d) First, **Subset Sum** $\leq_p$ **Knapsack**. Pretty straightforward: With a "target" value and the given weight capacity, find a set arrangement of nonnegative integers. The values are integers from a Subset Sum.

(e) So the dynamic programming implementation (way up above) has a running time $O(n * W)$ where W is a Weight capacity. But as remarked before, W repetitively can be huge, and so it won't run in polynomial time.

(f) So changing the dynamic programming to a min weight subset that yield exacts a value from the knapsack, rather than bound the weight capacity, might increase time, right?

(g) Nope. The running time becomes $O(n^2 *$ max(v))

(h) Approximation Algorithm:

i. Round all values up to lie in smaller ranges

ii. Run the dynamic programming on the rounded values

iii. Return the optimal values from the rounded instance

(i) The rounding is virtually incremental, or at least within a reasonable bounds, so that the true values are not completely off.

(j) Although optimality would have been suffered, the running time becomes $O(n^3/\varepsilon)$

(k) The proof is essentially a turning of the sum into
$\sum_{i \in Solution} \bar{v_i}$
$\leq \sum_{i \in Solution} (v_i + \Theta)$
$\leq \sum_{i \in Solution} (v_i + n * \Theta)$
$\leq (1 + \varepsilon) * \sum_{i \in Solution} v_i$

(l) Which is basically all just saying the difference in value growth within the Capacity solution is incrementally altered by at most ε

10. And that ends this chapter. Tad bit quicker than the other chapters, and essentially wraps up the class. The main portion from this chapter is just about practical creation of an aglorithm for some really difficult NP problems. Not far from the chapter prior.

Final Remarks

And that ends the notes! I hope they were helpful, they were a back and forth between the slide notes and my own personal notes from lectures. They seem to capture the general idea of everything we covered, although some parts are less obvious then others. I suggest it useful to review the slide notes in juxtaposition with the notes in this PDF. But my hope is that this is comprehensive enough for your own benefit. And I apologize if there are grammatical errors, no autocorrect makes catching them tough.