CS180: Algorithms & Complexity

Stanley Wei

M. Sarrafzadeh | Fall 2023

Contents

1 Stable Matching

Definition. Given two groups S_1, S_2 each of n elements, a ***complete matching*** on S_1, S_2 is a mapping between S_1, S_2 such that every element in each group is mapped to *exactly* one element in the other.

- *Extension*: To represent an element y taking multiple matches from the other set, we can simply create multiple copies of the element (y_1, y_2, etc.)

Variation: We can associate with each element [in both sets], a *priority list/complete ranking* of which elements in the other group it "prefers" to match with.

In this case, a complete matching is called ***stable*** if there do not exist any cases in which two elements that are not matched with each other, both prefer the other over their current matches.

Problem (***Stable Matching***)**.** Given two groups of n elements, find a stable matching.

Algorithm (Gale-Shapley)

1. Pick arbitrary element $x \in S_1$. Match x with its highest-priority match $y \in S_2$.
2. Continue picking unmatched elements $x \in S_1$ until all elements in have been matched. For each $x \in S_1$: look at the highest-ranked element $y \in S_2$ that it has not already asked.
   (a) If y is either currently unmatched or has x higher in its ranking than its current match x', match x with y (and unmatch x', y).
   (b) Otherwise, move to the next-highest match for x until x is matched.

Proof of Correctness

Claim. No elements will be unmatched.

Proof. Assume, for the sake of contradiction, that there is an element in group 1 that is unmatched when the algorithm terminates.

If there is an element in group 1 that is unmatched, then there must also be an element in group 2 that is unmatched.

The only case in which the element in group 2 would not match with the element in group 1 is if the element in group 2 already found a better match; but the element in group 2 is unmatched. [Contradiction]

□

Claim. The algorithm produecs a stable matching.

Proof. Assume, for the sake of contradiction, that on some S_1, S_2, the algorithm does not produce a stable matching. We know from the previous claim that the algorithm will always produce *a* matching. Then, the matching produced by the algorithm on S_1, S_2 must not be stable, i.e. there is an element $x \in S_1$ and an element $y \in S_2$ such that y is higher in x's ranking than x's current match and x is higher in y's ranking than y's current match.

Case 1. x has not already asked y. This is not possible: per the algorithm, x will only match with an element of lower ranking than y if x has already asked every element ranked higher than its current match, including y, per the algorithm. [Contradiction]

Case 2. x already asked y. This would mean y matched with a higher-ranked element than x. x is higher in y's ranking than y's current match; therefore y moved from a higher-ranked match to a lower-ranked match (its current match). But a characteristic of the algorithm is that an element in S_2 will never move from a higher-ranked match to a lower-ranked match. [Contradiction]

□

Implementation

We can store S_1, S_2 as linked lists.

- S_1 is stored as a linked list, and the first element x in the list taken as the next element to match at each step; when x is matched, remove it from the list.
  - If an element $x \in S_1$ is unmatched, it can be inserted back into the list.

We can store priority rankings of elements $x \in S_1$ as arrays associated with each element.

- Since the priority ranking of an element $x \in S_1$ is never backtracked, we can find the next-not-asked entry in O(1).

Since the priority rankings of elements $y \in S_2$ are searched, rather than simply iterated through, there are two options for storing the priority rankings of elements $y \in S_2$:

1. Store as a vector, and search via linear search $\rightarrow$ $O(n^2) \cdot O(n)$=$O(n^3)$.
2. Pre-compute and store as a hash map (mapping indices in S_1 to rank): $O(n^2)$ for pre-computing, $O(n^2) \cdot O(1)$ search $\rightarrow$ $O(n^2)$+$O(n^2)$=$O(n^2)$ overall.

Time Complexity

Best case: Only n asks are needed for the algorithm to terminate.
Worst case: n^2 asks are needed $\rightarrow$ $O(n^2)$

2 Example Problems

2.1 Majority Problem

Problem (***Majority Problem***)**.** Given a total of n votes for m candidates, we want to find whether any single candidate has a majority (strictly more than half the votes).

- **Condition**: Algorithm should use a constant amount of extra shortage
- Assume votes are presented in the form of an n-length array of numbers 1, 2, ..., m

Observation. If a given candidate m has a majority [$> \frac{n}{2}$ votes] across a set of n votes, then if we remove 2 votes (one vote for m, and one vote not for m), m will still have a majority [$> \frac{n-2}{2}$ votes] across the remaining $n-2$ votes.

Alternatively: if a given candidate m must have $> \frac{n}{2}$ votes to win, then if we remove 2 votes as above, m will only need $> \frac{n}{2} - 1$ votes in the remaining set to win; any other candidate (aside from the other candidate whose vote was removed) still needs $\frac{n}{2}$ to win.

We can use this observation to find an algorithm that progressively reduces/simplifies the problem to arrive at a solution.

Algorithm

1. Take the 1st element (vote) as our *temporary majority candidate*, and initialize its *vote counter* to be 1.
2. Advance a pointer through the array. At each element:
   (a) Case 1: The element is the same as the temporary majority. Then increment the majority vote counter by 1.
   (b) Case 2: The element is different from the temporary majority. Then decrement the majority candidate vote counter by 1.
      - This is analogous to implicitly removing one element of the temporary majority + the just-found element
   (c) If the majority vote counter reaches 0, there is no current majority; set the temporary majority to be the next element and continue
3. When the loop finishes:
   (a) Case 1: The temporary majority is nonzero, and has at least 1 vote. Then that specific element is the only one that might have a majority; rescan all the votes to determine if that element has a majority.
   (b) Case 2: The temporary majority is null. Then there is no majority.

Implementation

Only 4 pieces of extra storage needed:

1. A pointer to start of array
2. Our current index in the array
3. One variable denoting the [temporary] majority candidate
4. One variable containing the no. of votes for that majority candidate

2.2 Famous Person Problem

Problem (***Celebrity Problem***)**.** Given a set of n people, we define a [the] *famous person* to be a person who does not know anyone, but is known by all other people (where "know" is a one-way relation).

We want an algorithm that, given a set of people, can efficiently find the famous person in the set. Similar to the last problem, we want to find a way to progressively reduce the size of the problem in order to arrive at a solution.

Observation. Given two people A, B:

1. Case 1: A knows B. Then A is not a famous person, and B may be a famous person.
2. Case 2: A does not know B. Then B is not a famous person, and A may be a famous person.

Algorithm

1. While the set of people is of size > 1:
   (a) Pick two people A, B from the set.
      i. Case 1: A knows B; then we remove A from the set.
      ii. Case 2: A does not know B; then we remove B from the set.
2. Return the remaining element.

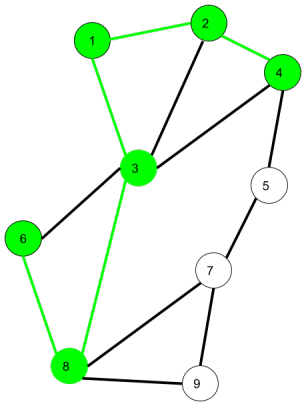
3 Graphs

3.1 Introduction to Graphs

Graphs

Definition. A **graph** is composed of **nodes** and **edges** between nodes.

- A ***digraph*** is a graph where each edge has an associated direction.
- A ***weighted graph*** is a graph where each edge has an associated weight.
- *Notation*: n nodes, m edges
  - Given a graph with n nodes, the maximum number of possible edges is $\frac{(n)(n-1)}{2} \sim n^2$.



Definition. A **path** is a sequence of nodes/vertices v_1, ..., v_k of a graph, where for any v_n, v_{n+1} there exists an edge (v_n, v_{n+1}).

- A *simple path* is a path where all vertices are distinct.
- A *cycle* is a path where the endpoint is the same as the start point.
- The **distance** between two nodes u, v is the length of the shortest path $u \to v$.
- A graph is called **connected** if, for any two vertices in the graph, there exists a path between them.

Trees

Definition. A graph is called a **tree** if it is connected and does not contain a cycle.

- Nodes in a tree are referred to as *descendants* of a root node, with according *parent/child* relationships between adjacent nodes in the tree.

Properties of Trees:

1. An n-node tree has exactly $n - 1$ edges
2. A tree is a maximum network without a cycle.

Representations of Graphs

1. **Adjacency Matrix**: Represents an n-node graph as an $n \times n$ matrix, where the i, j cell is 1 if $\exists$ an edge (v_i, v_j) in the graph, 0 otherwise.
   - Suitable for *dense graphs*, where nodes generally have close to n neighbors.
     - **Advantage**: Checking whether two given nodes are adjacent is a fixed O(1).
     - **Disadvantage**: Given a node, the cost of finding all adjacent nodes is a fixed O(n) due to needing to scan an entire row of the array.
   - In weighted graphs, the value of a cell i, j is set to the *weight* of the edge (v_i, v_j), if it exists.
2. **Adjacency List**: Represents an n-node graph via n node objects, where each node contains a linked list of all edges containing that node.
   - Suitable for *sparse graphs*, where nodes generally have much less than n neighbors.
     - **Advantage**: Given a node x, the cost of finding all nodes adjacent to x is only directly proportional to the number of neighbors of x, not to n.
     - **Disadvantage**: Given a node x, the cost of checking if it is adjacent to a node y is proportional to the number of neighbors of x.
   - In digraphs, a node may have two linked lists for holding incoming and outgoing edges, respectively.

In both cases: the complexity of representing a graph with n vertices, e edges is $\underline{\mathrm{O}(n + e)}$.

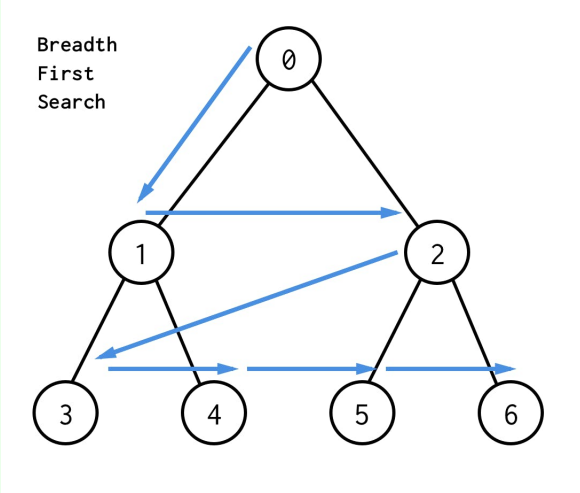
3.2 Graph Traversal: BFS & DFS

Two main algorithms for traversing unweighted graphs: **BFS** and **DFS**.

Breadth-First Search (BFS): *"Explore things that are close together first"*

Given a graph $G(V, E)$:

1. Begin at a root node u. Call the set $\{u\}$ "*layer 0*".
2. From the root node: find all adjacent nodes [***neighbors***] in G. Call the set of neighbors of u, "*layer 1*".
3. Once all neighbors of u have been found, find all neighbors of neighbors of u not already in layers 0/1. Call this set "*layer 2*". Continue finding layers 3, 4, etc. until all nodes have been added to a layer.



BFS Trees

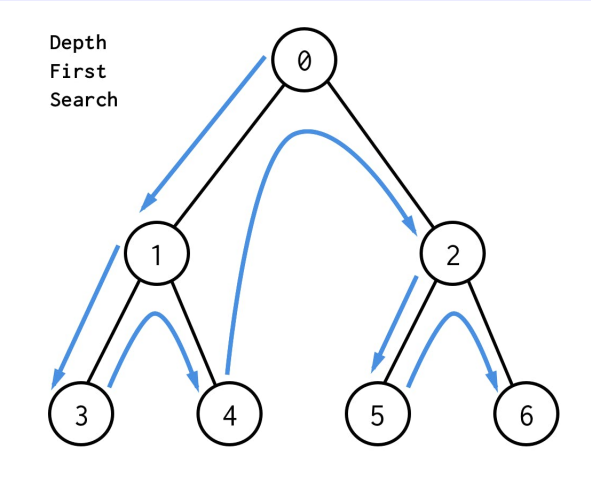
Definition (***BFS Tree***)**.** During BFS, we call every edge used when first visiting a node, a **tree edge**. A subgraph of G composed of only tree edges from an instance of BFS [rooted at u] is called a **BFS tree** [rooted at u].

Definition (***Layer***)**.** The **layer** of a node v in a BFS tree rooted at u is defined as the distance from the root node u to v in the tree, and is equivalent to the length of the shortest path $u \to v$ in G.

Depth-First Search (DFS): "*Explore everything in one direction first*"

Given a graph $G(V, E)$:

1. Begin at a root node u.
2. From the root node: pick a random neighbor of u, and visit it.
3. Upon visiting a node: visit any neighboring node that has not already been visited. Continue doing so until we reach a node with no unvisited neighbors, then backtrack until we reach a node with unvisited neighbors.



DFS Trees

Definition (***DFS Tree***)**.** A subgraph of G composed of only the edges taken from an instance of DFS is called a **DFS tree**.

- Any edges in G not in the DFS tree, form a cycle with the edges of the DFS tree.

BFS vs DFS

BFS, DFS are "*orthogonal algorithms*":

- **BFS**: Used for finding **distances** from a node u to other nodes in the graph
- **DFS**: Used for finding **cycles** in a graph

Implementations

Breadth-First Search (Implementation)

Given a graph $G(V, E)$:

1. Begin with a root node u. Initialize a **queue** [LIFO] containing unvisited nodes. Add all neighbors of the root node to the queue.
2. While the queue is nonempty: pop a node v from the queue. For every edge (v, w) with v as an endpoint: if the node w has not yet been visited, add w to the queue and record the edge (v, w).

Depth-First Search (Implementation)

Given a graph $G(V, E)$:

1. Begin at a root node u. Initialize a **stack** [FIFO] containing unvisited nodes. Add all neighbors of the root node to the stack.
2. While the stack is nonempty: pop a node v from the stack. For every edge (v, w) with v as an endpoint: if the node w has not yet been visited, add w to the stack and record the edge (v, w).

Time Complexity

For both BFS, DFS: every vertex is added to the stack/queue exactly once [$O(V)$], during the first visit to that vertex. Similarly, every edge is seen exactly twice (once on each endpoint) $\rightarrow$ $O(E)$. Taken together: $O(V + E)$.

There are two methods of interpreting this complexity:

1. ***Linear***: $O(V + E)$ is linear with respect to V, E.
2. ***Exponential***: The maximum number of edges in a graph is dependent on V; namely, proportional to V^2. If we treat E as dependent on V: $O(V + E) \rightarrow O(V + V^2) \rightarrow O(V^2)$.

Given a graph with n nodes, m edges: complexity is typically said to be $O(m + n)$, linear with respect to the input size.

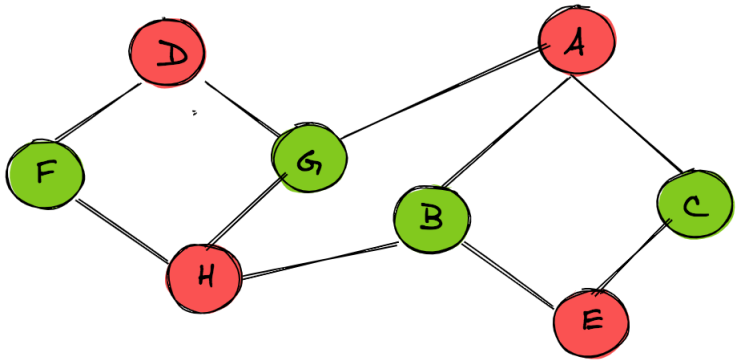
3.3 Extensions of BFS/DFS

Graph Coloring

Definition (***Bipartite graph***)**.** A **bipartite graph** [*2-colorable graph*] is a graph where the nodes can be partitioned into two sets X and Y such that every edge in the graph is between a node in X and a node in Y.

- Equivalently: A bipartite graph is a graph with **no odd cycles**.
- Components of a graph may be individually referred to bipartite/not bipartite.

Problem (***Testing Bipartiteness***)**.** Given a graph $G = (V, E)$ [undirected] with one component, determine whether G is bipartite [2-colorable].



Approach: Assign each vertex one of two colors (ex: red or blue). If the graph is bipartite, then we must be able to find an assignment such that given a red vertex, all of its neighbors are blue, and vice versa.

Algorithm (Graph Coloring)

1. Assign a random node to be the root node of a BFS, and color it red.
2. Use BFS to generate a BFS tree of the graph.
3. Assign all nodes in every odd layer to be blue, and all nodes in every even layer to be red.
4. The graph is bipartite if and only if there are no edges between two nodes of the same color (i.e. in the same layer).

Runtime: $O(m + n)$ [BFS]

Strong Connectedness

Definition. A **digraph** G is called "**strongly connected**" if, for any two nodes u and v in G, there exists a path from each to the other (u and v are *mutually reachable*).

Problem (***Strong Connectivity***)**.** Determine if a digraph G is strongly connected.

Algorithm (Strong Connectivity)

1. Pick an arbitrary vertex $u \in G$; from u, run BFS on G. If every node is included in the resulting BFS tree, then there is a path from u to every node in the tree.
2. Look at the graph G_{rev}: G, but with the direction of all edges reversed. Run BFS from u on G_{rev}. If every node is included in the resulting BFS tree, then there is a path from every node in the tree to u.
3. If there is a path from u to every node, and a path from every node to u, then G is strongly connected.

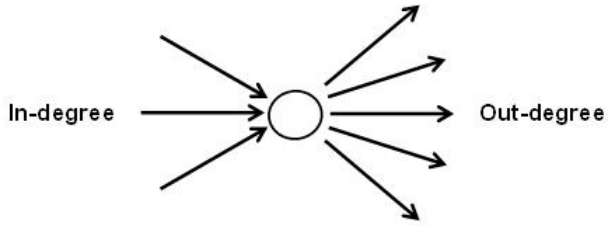
Runtime: $O(m + n)$ for BFS

3.4 Topological Sorting

Digraph Terminology

Definition (***Degree***)**.** The in-degree and out-degree of a node u in a digraph are the number of edges into and out of u, respectively.

- A node with an in-degree of 0 is called a ***source***.
- A node with an out-degree of 0 is called a ***sink***.



Definition. A **directed acrylic graph** (**DAG**) is a directed graph with no cycles.

Remark. Any DAG must have at least one source node.

Observation. Removing nodes and/or edges from a DAG, preserves direct acylicity.

Topological Ordering

Definition. A **topological ordering** of a digraph G is an ordering $v_1, v_2, ...$ of nodes of G such that for any edge $v_i \to v_j$ in G, $j > i$.

- A topological ordering need not be unique.

Problem (***Topological Sorting***)**.** Given a directed graph $G(V, E)$, find a topological ordering of G [or determine that none exists].

Observation. A graph G has a topological ordering only if G has no cycles, i.e. if G is a DAG.

Algorithm (Topological Sorting)

1. Initialize an empty topological ordering.
2. While not all nodes have been added to the ordering:
   (a) Pick any source node u, and append u to our ordering.
      - If there is no source node, then G is not a DAG; then we terminate.
   (b) Delete u from G, as well as any edges with u as an endpoint.
      - If G is a DAG, then G will still be a DAG after removing u.

Implementation

Given a graph with n vertices, e edges, we can pre-compute the in-, out-degree of all nodes before beginning the topological ordering algorithm.

This can be done through a loop through the edges of the graph in $O(e)$ time.

During the topological sorting process, source nodes can be stored using a stack/queue.

- Before sorting, we can find all sources [nodes with an in-degree of 0] in the original graph by scanning all vertices in linear time [$O(n)$].
- At each step, we pop a source from the queue to add to our ordering.
- When deleting an edge:
  - We can decrement the in-degree of the node on end of the edge by 1.
  - If this causes the in-degree of the node to become 0, then we add it to our queue.

The time complexity of the loop is thus $O(n + e)$: each node is popped from the queue exactly once [$O(n)$], and each edge is looked at/removed exactly once [$O(e)$].

$\rightarrow$ **Overall runtime**: $O(n + e)$

4 Greedy Paradigm

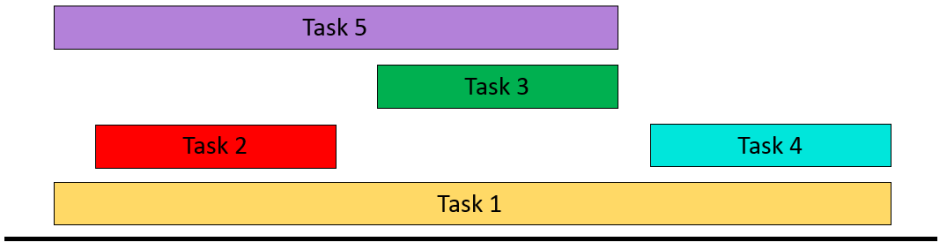
Overview (Greedy Paradigm)

Greedy paradigm for local optimization: In certain cases, we may decide to make simplifying assumptions about a problem/solution space (without exploring the entire space) for speed and efficiency in our algorithms.

Greedy assumptions can be used to find a solution that is ***locally optimal***, but ***may or may not be globally optimal***; if we can then find a proof of global optimality, a greedy solution can be generalized to the entire space.

4.1 Interval Scheduling

Problem (***Interval Scheduling***)**.** Given a number of overlapping intervals, we want to find the maximum number of non-overlapping intervals.



Greedy Heuristic: The interval that ends the earliest will always be the best choice.

$\Longrightarrow$ *Solution*: Keep taking the earliest-ending interval

Algorithm (Interval Scheduling)

1. Sort the set of intervals by end time (via heapsort, e.g.)
2. While the set is non-empty:
   (a) Choose the earliest-ending interval and add it to our solution
   (b) Remove (from the set) all intervals that overlap with the just-added interval

Runtime: $O(n \log n)$ [Sorting]

Proof of Optimality

Via a *stay-ahead argument*: Given a solution from the greedy algorithm, we want to prove that the first k intervals in the greedy solution will end no later than the first k intervals in any other solution, $\forall\, k \in \mathbb{N}$. (prove by induction)

Proof. By induction:

1. *Base case*: $k = 1$. Our greedy algorithm chooses the earliest-ending interval across all arguments, therefore any other solution for $k = 1$ will either pick that same interval (and thus end at the same time) or a different interval (and end later)

2. *Inductive step*: Assume the claim holds for $k = n$.

   Since the hypothesis is true for $k = n$, therefore the first n intervals of the greedy solution will end at the same time at, or before, any other solution.

   Then the greedy algorithm can pick, for its $(n+1)^{\text{th}}$ interval, any interval that the non-greedy algorithm can pick.

   Then $(n+1)^{\text{th}}$ interval in the greedy solution will end no later than the $(n+1)^{\text{th}}$ interval in the non-greedy solution, thus the claim holds for $k = n + 1$.

□

4.2 Shortest Path Problem

Problem (***Shortest Path/Minimum-Weight Path***)**.** Given a graph with weighted edges, find the minimum weight path from a vertex a to another vertex b.

- Assumption: All weights are non-negative. [Negative weights: use *Bellman-Ford*]

Algorithm (Dijkstra's)

1. Start at vertex a.
2. Find the neighbor x of a that is closest to a. Then the shortest path $a \to x$ is the edge from a to x.
   (a) Assign $d(a, x)$ to be the length of this edge.
3. Find next-closest vertex y to a. This vertex will be either a neighbor of a or a neighbor of x.
   (a) Fix $d(a, y)$ as the length of the edge (a, y), or the length of the edge (x, y) plus the distance from x to a. (Or the minimum of the two). Then $d(a, y)$ is the weight of the minimum-weight path $a \to y$.
4. Continue this process until the distance from all vertices to a has been determined.
   (a) Can represent all vertices v as belonging to one of three categories:
      i. *Processed*: A minimum-weight path $a \to v$ has been found.
      ii. *Intermediate*: Any path $a \to v$ has been found.
      iii. *Unprocessed*: v has not yet been seen.

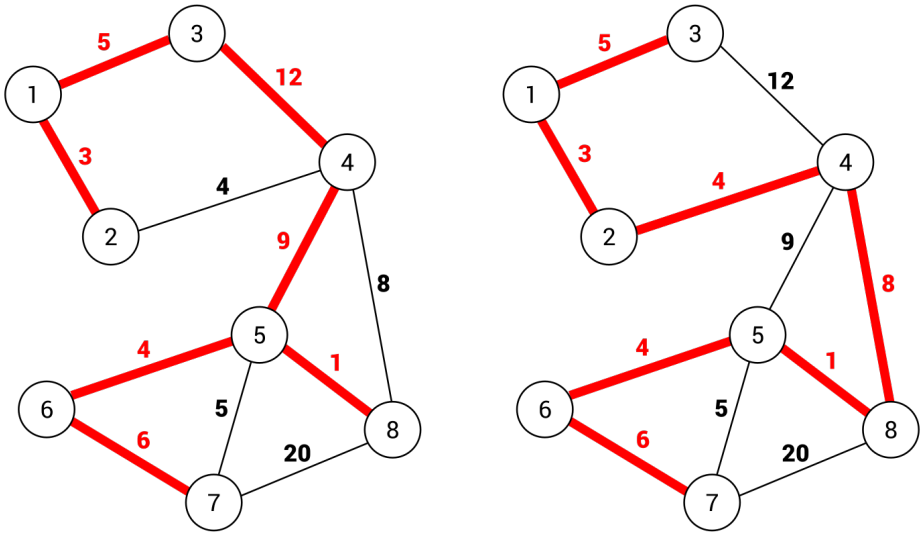
Implementation

Two implementations for storing processed/intermediate vertices:

1. Store intermediate vertices in an **array** - $O(N^2)$
   (a) N steps, at most $(N - 1)$ vertices modified per step
2. Store every intermediate vertex in a **heap** - $O(E \log N)$
   (a) O(1) to find the minimum-weight vertex at each step
   (b) $O(\log N)$ to insert a vertex; maximum E insertions

4.3 Minimum Spanning Tree

Definition. A ***subgraph*** of a graph G is a subset of the nodes/edges of G. In particular, a subgraph is called a ***tree subgraph*** if it has a tree structure.



Definition. A subgraph of a graph G is called a ***spanning tree*** of G if it is a tree subgraph, is connected, and touches every vertex in the graph.

- Spanning trees are the minimum network of edges in the graph without cycles; any network with a cycle can be reduced to a spanning tree without one
  - A spanning tree always contains $(n-1)$ edges
- BFS, DFS trees are themselves spanning trees

Definition. A ***minimum spanning tree*** (**MST**) of a graph is the spanning tree of minimum total weight.

- An MST is unique iff all edge weights are unique (no two edges have the same weight).

The MST Theorem

Problem. Given a graph G, find a minimum spanning tree of G.

Theorem 4.1. *For every MST on the graph, for any partition of G into two sets, the edge of minimum weight between partitions will be included in the MST.*

Proof. Begin with an arbitrary MST and an arbitrary partition. Assume, for the sake of contradiction, that the edge of minimum weight between partitions is not in the MST.

Let v and w be the two vertices such that the edge of minimum weight is the edge (v, w).

Then there is a path between v and w in the MST, and that path must take an edge between partitions that is of larger weight than (v, w).

Then the MST would become a spanning tree with smaller total weight if we replaced that edge with the edge (v, w).

Then there is a spanning tree with total weight smaller than the MST. [*contradiction*] □

‘

Algorithms for MST

1. Prim's Algorithm

Algorithm (Prim's)

1. Partition the vertices of G into two sets, such that one set (S_1) contains only 1 vertex and the other set (S_2) contains the other $(n-1)$ vertices.
2. While the $S_1 \neq V$:
   (a) Take the minimum edge between S_1, S_2 and add it to our subgraph. If there are multiple minimum edges, pick one.
   (b) Let $e_i = (v \in S_1, u \in S_2)$ be the just-added edge. Move u to S_1.

Runtime: $O(N^2)$ / $O(E \log N)$ [Identical to Djikstra's]

Note: Correctness of Prim's Algorithm is provided by the MST Theorem.

2. Kruskal's Algorithm

Algorithm (Kruskal's)

1. Sort all edges by weight.
2. Add the two shortest edges e_1, e_2 to the graph.
3. Continue looking at next-shortest edges e_k until $(n-1)$ edges have been added.
   For each edge $e_k = (a, b)$:
   (a) Case 1: a and b are not connected in our graph [in different connected components]. Then we add e_k to our graph.
   (b) Case 2: Both a, b are already connected in our graph (i.e. adding e_k would create a cycle). Then we do not add e_k to our graph

Time Complexity

Runtime: $O(E \cdot N)$ [using arrays]

- Our algorithm requires we keep track of the connected components in the graph, and merge them as edges are added; combining two arrays is an $O(n)$ operation.

Union-Find

Q: Can we find a more efficient data structure for storing connected components?

$\implies$ **A**: Disjoint sets [Union-find]

Problem (***Union-Find***)**.** Given n elements partitioned into k sets, we want to be able to perform two operations efficiently:

1. **Find**: Given two elements, determine if they are in the same set.
2. **Union**: Given two sets, replace them with their union.

Solution: Store the elements of each set in a ***rooted tree***.

Operations:

1. **Find**: Visit both elements and determine the roots of their respective trees. The elements are in the same set iff the roots are the same.
2. **Union**: We can append one tree to the end of another, e.g. by adding the root of one tree as an element of the other.

Q: How should we arrange our rooted trees, such that both operations run efficiently?

Possible Configurations

1. Store each tree as a single path of elements (*à la* a linked list)
   - *Advantage*: Union runs in O(1)
   - *Disadvantage*: Find runs in O(n)
2. Store each tree by having all elements be children of the root (i.e. as a tree of height 2)
   - *Advantage*: Find runs in O(1)
   - *Disadvantage*: Union runs in O(n)
     - (For the merged tree to have the same depth-1 configuration, the program would need to individually point all elements of one tree toward the root of the other)
3. Store each set as a *balanced tree*, such that the height is proportional to the logarithm of the number of vertices
   - *Advantage*: Find runs in O($\log n$)

  - The maximum number of iterations needed to backtrack from an element to a tree root is $\log n$
- *Advantage*: Union runs in O(1)
  - Simply pointing the root of the shorter tree toward the root of the other is an O(1) operation, and the resulting tree still obeys the logarithm-height property

$\implies$ We can use balanced rooted trees to keep track of connected components in Kruskal's:

Kruskal's (Union-Find)

- Start by placing each vertex in its own Union-Find set of height 1
- Within Kruskal's algorithm, use Union-Find trees to keep track of connected components:
  - Checking cycles: Given an edge, can perform a Find on the two vertices to determine if they are in the same connected component
  - Adding edges: Perform a Union of the connected components on the edge

Time Complexity

Runtime: $O(E \log E)$

- $O(E \log E)$ to sort all m edges
- $O(E \log N)$ for Kruskal Union-Find loop
  - m edges; 1 Find [$O(\log n)$] and 1 Union [O(1)] per edge

MST Notes

Note: Can find many algorithms for finding MSTs just by using the MST Theorem.

Ex: "***Reverse deletion algorithm***" (relies on MST Theorem)

1. Sort the edges of the graph in descending order.
2. Loop through the edges: at each edge, check if we can delete the edge without disconnecting the graph (if so, we delete the edge).

Note: Algorithms may fail if edge weights are not unique

$\implies$ **Solution**: If two edges have the same weight, we can add a small number ϵ to one of the edges to make the weights become unique

- May result in different MSTs depending on which edge is chosen
- In the case of multiple edges sharing the same weight, we may add ϵ, 2ϵ, etc.
  - Worst-case: becomes an additional step with complexity $\mathrm{O}(n)$

5 Divide & Conquer

Overview (Divide & Conquer)

Principle (***Divide and Conquer***): *If a problem is too difficult to solve directly, we can instead solve it by partitioning it into more solvable sub-problems.*

Divide & Conquer:

- A common approach is to utilize recursive partitioning.
- Divide-and-conquer solutions actually exist for many [most?] problems, though they are not always the optimal solution to the problem.

5.1 Merge Sort

If an array is too long to sort directly, then we can instead sort it by dividing it into subarrays that can be sorted directly.

Algorithm (Merge Sort)

1. Divide the array into some number of subarrays of equal size.
    - Divide subarrays into even smaller subarrays as needed. Keep dividing until the divided subarrays are small enough to sort directly.
2. Sort all subarrays.
3. *Merging step*: Once each subarray is sorted, then we can keep combining (merge) our subarrays into larger subarrays until we have returned to the original array.
    - *Combination*: We can merge two subarrays into a new subarray by continuously picking the minimum of the next element for each subarray.

Time Complexity

Merging Step: Given two arrays of size p, q, each element of the output array is added in O(1) $\rightarrow$ total runtime: $O(p+q)$

Overall Runtime: Let $T(n)$ be the time required to sort n elements. Per the algorithm:

1. $T(n) = 2T(\frac{n}{2}) + O(n)$

2. $T(1) = \mathrm{O}(1)$

We can define the cost of merging n elements to be λn, for some λ.

$$\begin{aligned} T(n) &= 2T(\frac{n}{2}) + \lambda n \\ &= 2(2T(\frac{n}{2}) + \frac{1}{2}\lambda n) + \lambda n \\ &= 2^2 \cdot T(\frac{n}{2^2}) + 2 \cdot \lambda n \\ &= 2^i \cdot T(\frac{n}{2^i}) + i \cdot \lambda n \,(\text{by induction}) \end{aligned}$$

After $\log_2(n)$ recursions, $T(\frac{n}{2^i})$ becomes just $T(1)$:

$$\begin{aligned} \implies T(n) &= 2^{\log_2(n)} T(\frac{n}{2^{\log_2 n}}) + 2\lambda \cdot n \log_2(n) \\ &= n \cdot T(1) + 2\lambda \cdot n \log_2(n) \end{aligned}$$

$$\implies \mathrm{O}(n \log n)$$

Notes

- The ***master theorem*** is a general formula for finding the time complexities of recursive divide-and-conquer algorithms *à la* merge sort.
- It is not immediately obvious that $\mathrm{O}(n \log n)$ is optimal, since the most obvious lower bound for the time complexity of sorting is $\mathrm{O}(n)$; however, it has been proven that $\mathrm{O}(n \log n)$ *is*, in fact, optimal.

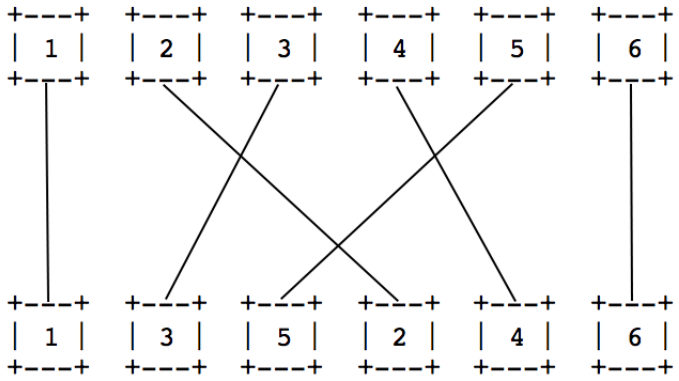
5.2 Counting Inversions

Problem (***Crossing Problem***)**.** Given two parallel lines between which there are total n lines (potentially overlapping), find an algorithm to determine the number of crossings.

- *Trivial solution*: compare every possible pair of lines [$O(n^2)$]

Observation. We can assign numerical indices $1, 2, \ldots$ to each line, in order of the coordinates where those lines touch one of the two parallel axes (e.g. the bottom axis). Then, two lines i, j cross iff i is before j on one axis and after j on the other.

- The number of crossings is equal to the number of out-of-order indices (*inversions*).



Algorithm (Counting Inversions)

1. Assign each line a label based on where that line meets the bottom axis, as described above. Create a counter [initially 0] to hold the number of crossings.

2. Take the sequence in which the lines meet the top axis, and perform merge sort.

   During every merge:

   (a) Case 1: The number on the right is larger than the number on the left; then the two lines did not cross.

   - We use "left" to denote the subsequence that occurred earlier in the original sequence (of the two subsequences being compared)

   (b) Case 2: The number on the right is smaller than the number on the left; then the number on the right crossed the current left number, and every number in the left array larger than the current left number.

   We increment the crossings counter accordingly.

$\Longrightarrow$ **Runtime**: $O(n \log n)$ for merge sort

5.3 Closest Pair Problem

Problem (*Closest Pair*). Given set of n points in $\mathbb{R}^i$, we want to find the closest pair of points, i.e. two points x, y $[x \neq y]$ such that $||x - y|| \leq ||x' - y'||$ for any other pair x', y'.

- *Trivial solution*: compare every possible pair of points $[\mathrm{O}(n^2)]$
- *Note*: There are a number of possible metrics for quantifying distance in $\mathbb{R}^i$ (e.g. the Manhattan/L1 norm); in this case, we consider only the *Euclidean/L2* norm.

In the case of $\mathbb{R}^1$ $[i = 1]$, we can obtain an $\mathrm{O}(n \log n)$ solution simply by sorting the points by coordinate. However, simple greedy approaches (e.g. projection onto an axis, taking polar coordinates) fail in general $\mathbb{R}^n$.

Observation. If we partition the set of points S into two subsets S_1, S_2, then the closest pair of points in S is either a closest pair in one of S_1, S_2, or is a pair $(x \in S_1, y \in S_2)$.

Namely, if we define δ to be the minimum of the distance between closest pairs in S_1, S_2, a pair $(x \in S_1, y \in S_2)$ is a closest pair in S only if $||x - y|| \leq \delta$.

- Suggests a divide-and-conquer approach: we can find the closest pair in S_1, S_2 recursively, then look for closest pairs between partitions during the merging step.

  $\rightarrow$ We can obtain an $\mathrm{O}(n \log n)$ runtime if we can merge in $\mathrm{O}(n)$.

The Merging Step

We can use $(i-1)$-dimensional subspaces of $\mathbb{R}^i$ (e.g. 2D planes in $\mathbb{R}^3$) as dividers between S_1, S_2. Then a pair of points between partitions can only be less than δ apart if both points are less than δ away from the divider.

Taking $\mathbb{R}^2$ as an example: if our divider is a vertical line at $x = \lambda$, then a pair of points $a \in S_1, b \in S_2$ can only have $||a - b|| < \delta$ if $|a_x - \lambda|, |b_x - \lambda| < \delta$.

However, we also observe that a, b must also have that $|a_y - b_y| < \delta$.

Consequence: For any point $p \in S_1$, a point $p' \in S_2$ can have $||p' - p|| < \delta$ only if:

1. $p'_x \in (\lambda, \lambda + \delta)$
2. $p'_y \in (p_y - \delta, p_y + \delta)$

Then any closest pair partner for p must be within the $\delta \times 2\delta$ rectangular region $(\lambda, \lambda + \delta) \times (p_y - \delta, p_y + \delta)$, i.e. we need only look at the points in that region. Then we can find that the merging step is $\mathrm{O}(n)$ if we can show that the number of points within this rectangle is bounded above by a constant.

Proof. We observe that the rectangle is strictly contained in S_2.

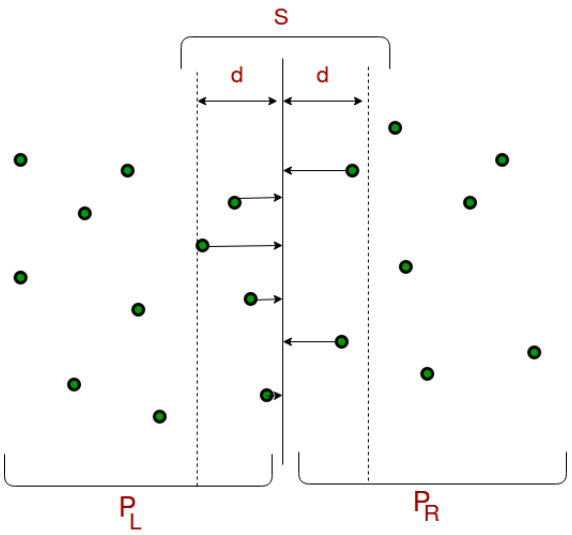
We can divide the rectangle into 8 squares of size $\frac{\delta}{2} \times \frac{\delta}{2}$. We see that all of these squares are themselves strictly in S_2; then there is at most one point in each square.

- Recall the definition for δ: if there were more than one point in a square, then we could find a pair in S_2 less than δ apart.

Since there is at most one point in each square, then there are at most 8 points in the rectangle; then for any point $p \in S_1$, we need only consider at most 8 points $p' \in S_2$ to identify all possible closest pairs. □

Then, if we can find a way to make sure we look at all said points (if they exist) in O(1), we can merge in O(n).

For simplicity, assume the points belong to $\mathbb{R}^2$, and that the divider is some vertical line $x = x_0$. Then the region of eligible points is the set of points within the x-interval $(x_0 - \delta, x_0 + \delta)$; and for any point $p = (x_1, y_1) \in S_1$, we need only look at the 8 points in S_2 with y-coordinates nearest to y_1 (that fall within the x-interval).



Then, if the points are sorted by y-coordinate, we can check our 8 points in O(1) by simply looking at the 8 points before/after p; then it remains to find a way to have all points by sorted by y-coordinate, while limiting our merges to O(n) runtime.

$\implies$ **Solution**: Sort all points by y-coordinate ahead of time, and merge in a way that preserves sortedness.

- During merging: assume left and right are sorted ahead of time, then use the same process as merge sort to merge.

Algorithm (Closest Pair - $\mathbb{R}^2$)

1. Sort the set of points by y-coordinate.
2. Keep using vertical lines to divide the set of points in half.
3. Keep dividing subsets into additional subsets, until subsets are of size 1 or 2.
4. Compute the closest pair within each subset (if a pair exists).
5. *Merging step*: Once the closest pair in each subset is found, then we can keep merging our subsets into larger subsets until we have returned to the original set.

   For each merge:

   (a) Merge points in sorted order (by y-coordinate).
   - During the merge: keep track of points of S_1, S_2 within the 2δ x-interval. For each point in S_1, look at the 8 points of S_2 above/below it and find the minimum distance. Keep track of the closest such pair.

   (b) Set the closest pair in the merged set to be the closest of:
   i. The closest pair in S_1.
   ii. The closest pair in S_2.
   iii. The closest pair between S_1, S_2.

Runtime: $O(n \log n)$

6 Dynamic Programming

Overview (Dynamic Programming)

Recall (***Divide & Conquer***): *"Given a problem, we can recursively divide the problem into distinct non-overlapping subproblems."*

Although divide & conquer is powerful, there may be instances in which a problem cannot be easily broken down into non-overlapping subproblems in a way that leads to a solution. However, we may still be able to use a similar approach via dynamic programming:

Principle (***Dynamic Programming***): Given a problem, we can divide it into multiple overlapping subproblems; if we can solve each subproblem optimally, then we can combine the solutions to subproblems to obtain a solution for the original problem.

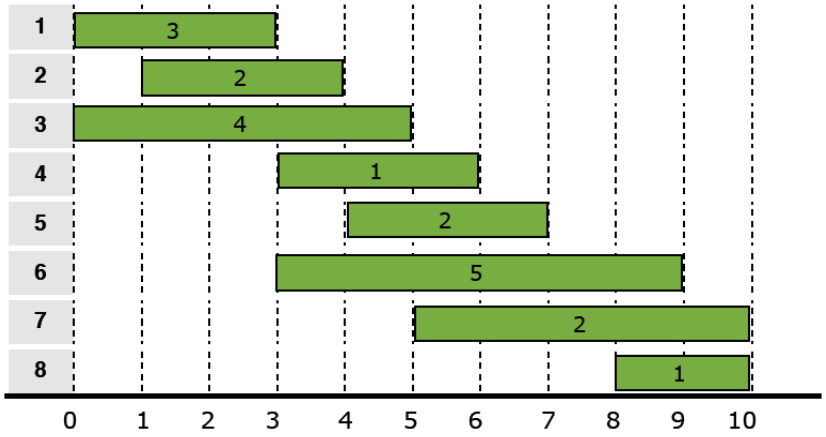
Dynamic programming is analogous to a controlled exhaustive search - looking through a large number of possibilities (helping us ensure the optimality of solutions), but in a way that avoids the typical runtime cost of blind exhaustive search.

6.1 Weighted Interval Scheduling

Recall (***Interval Scheduling Problem***): Given a number of overlapping intervals, we want to find the maximum number of non-overlapping intervals. (Solved using a greedy approach)

→ **Extension** (***Weighted Interval Scheduling***): Given a number of weighted overlapping intervals, we want to find the maximum set of non-overlapping intervals (maximum weight).

- No efficient greedy/divide-and-conquer solutions



Approach

We can denote weighted intervals as elements $I_j = [l_j, r_j]$ for start time l_j, end time r_j, with corresponding weights w_j.

We can describe optimal solutions for end times, where the optimal solution for end time r is the maximum weight set of intervals ending at (or before) r. Let opt(r) denote the *optimal solution for end time* r.

Observation. For any interval I_j: either I_j is in the optimal solution opt(r_j) for end time r_j, or it is not in the optimal solution for end time r_j. Looking at our two cases:

1. Case 1: I_j is in opt(r_j) $\implies$ opt(r_j)=opt(l_j)+w_j
2. Case 2: I_j is not in opt(r_j) $\implies$ opt(r_j)=opt(r_{j-1})

Since solutions opt(r) are specifically *optimal* solutions, we can thus find:

$$\text{opt}(r_j)=\max\{\text{opt}(l_j)+w_j,\ \text{opt}(r_{j-1})\}$$

We can take the problem of finding optimal solutions for various times, as the subproblems for our larger problem. Notably, we assume that when finding a value opt(r), we have already found all values opt(s) for times $s < r$.

Algorithm

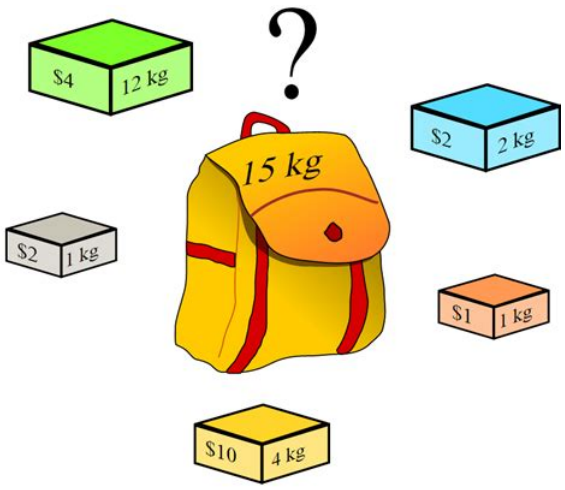
We can store the solutions to our subproblems via an array of length $2n$, where each index in the array (corresponding to some time r) maps to a cell containing the solution opt(r).

1. Create an array of length $2n$, where each index in the array represents either a start time l_j or an end time r_j.
   (a) Sort the array.
2. Set opt(l_1)=0.
3. For each index i of the array:
   (a) Case 1: The entry x_i at index i is a start time l_j; then we set $x_i = x_{i-1}$.
   (b) Case 2: The entry x_i is an end time r_j; then we set $x_i = \max\{x_{l_j} + w_j, x_{i-1}\}$
4. We take the value of the cell corresponding to end time r_j as the solution to our problem.

6.2 Knapsack Problem

Problem (***Knapsack***)**.** Given a knapsack with space capacity S and n items I_i each with value v_i and space s_i - what is the most valuable set of items that can fit in the knapsack?

- Assume multiple of each item is allowed
- Greedy approaches do not work



Approach

We can find solutions $\text{opt}(i, j)$ to subproblems of finding the most valuable set of items, when considering only the first i items, that can fit in a knapsack with space capacity j.

Observation. For each solution $\text{opt}(i, j)$:

1. Case 1: I_i is included in the set $\implies$ $\text{opt}(i, j) = \text{opt}(i, j - s_i) + v_i$
   - In the case where we choose not to allow duplicate items, this would instead become $\text{opt}(i, j) = \text{opt}(i - 1, j - s_i) + v_i$
2. Case 2: I_i is not included in the set $\implies$ $\text{opt}(i, j) = \text{opt}(i - 1, j)$

   $\implies$ $\underline{\text{opt}(i, j) = \max\{\text{opt}(i, j - s_i) + v_i, \text{opt}(i - 1, j)\}}$

Notably, we assume that when computing a value $\text{opt}(i, j)$, $\text{opt}(i', j')$ has already been found for all $i' \leq i$, $j' \leq j$ (excluding $i' = i, j' = j$).

Additionally, we can find *base cases*:

- opt$(0, j)$ is just 0, since no items are being considered
- opt$(i, 0)$ is just 0, since there is no space to put any items

Algorithm

1. Initialize a table of size $n + 1 \times S + 1$.
2. Set each entry opt$(0, j) = 0$
3. Set each entry opt$(i, 0) = 0$
4. Fill the table in row-major order: first fill row 1, then row 2, etc.
   (a) For each entry opt(i, j):
      i. If $s_i \leq j$: set opt(i, j)=max{opt$(i, j - s_i)+v_i$, opt$(i - 1, j)$}
      ii. If $s_i > j$: set opt(i, j)=opt$(i - 1, j)$
5. Take opt(n, S) as the final solution.

Time Complexity

Runtime: # cells $\cdot$ cost per cell $= (n \cdot S) \cdot \mathrm{O}(1) = \mathrm{O}(nS)$

We observe, however, that our input size (the amount of space needed to store the input) is only proportional to the number of items n, since S is just an integer (changing S does not affect the input size); then our runtime $\mathrm{O}(nS)$ is proportional to a factor S that is, in turn, *not* directly proportional to our input size.

We have a special name for these cases:

Definition. An algorithm may be said to have ***pseudo-polynomial runtime*** if it can be either polynomial or not polynomial (relative to the input size), depending on a parameter that is independent of input size.

- Ex: If $S = 2^n$, then $\mathrm{O}(nS)$ would be exponential relative to input size n; if $S = n^4$, then $\mathrm{O}(nS)$ is polynomial.

Notes

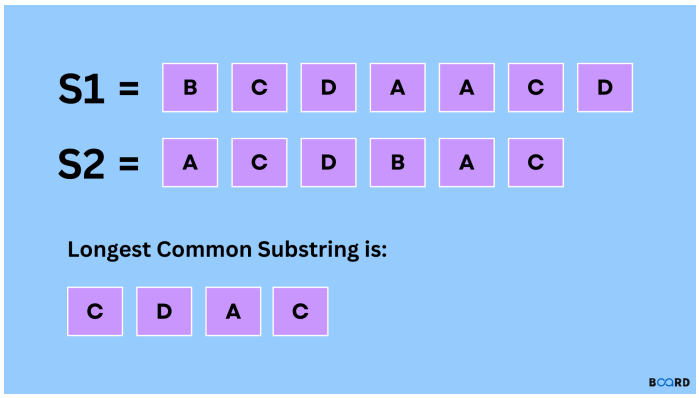
- Matrix representations are common in dynamic programming:
  - Weighted interval scheduling: 1 row, n columns
  - Knapsack problem: n rows, S columns
- Solutions to problems so far have only provided total values/weights, not actual items/intervals in the optimal set

- To find items/intervals, we would backtrack a pointer from the cell containing the overall solution to the first cell in the matrix
  - Jumps taken during the backtracking process would each correspond to items added - can be stored for each cell during the computation process
- Backtracking (if done) becomes an additional step with complexity O(nS)

6.3 Sequence Alignment

Problem (***Longest Common Subsequence***)**.** Given two sequences $L\&R$ with sizes $m\&n$, find the longest subsequence in common between both sequences.

- Also called: *Sequence Alignment*, *Maximum Subsequence*
  - "Sequence Alignment" may also be used to refer to more complex problems, such as RNA Sequencing and other extensions of Longest Common Subsequence
- Is equivalent to *aligning shared terms* between sequences



Approach

We can find solutions $\text{opt}(i, j)$, representing the longest subsequences in common between the first i characters of L and the first j characters of R.

For each pair i, j:

Case 1: $\underline{L_i = R_j}$. We claim that $\text{opt}(i, j) = \text{opt}(i-1, j-1)+1$.

Proof. We observe that if $L_i = R_j$, then the optimal solution $\text{opt}(i, j)$ includes at least one of L_i, R_j; otherwise, we could match up L_i, R_j and extend $\text{opt}(i, j)$ by 1 [a contradiction].

We use this result to show that matching L_i and R_j is an optimal solution. Assume, for instance, that L_i is included in the optimal solution, and that another optimal solution exists that matches L_i with some character R_k $(k \neq j)$; then we could swap L_i's match to R_j and obtain a solution of equivalent length. □

$$\implies \text{opt}(i, j) = \text{opt}(i-1, j-1)+1.$$

Case 2: $\underline{L_i \neq R_j}$. Then either L_i is matched with some element not R_j, R_j is matched with some element not L_i, or neither L_i, R_j are matched. Notably, L_i and R_j could not both be matched simultaneously.

$$\implies \text{opt}(i, j)=\max\{\text{opt}(i-1, j), \text{opt}(i, j-1)\}$$

Algorithm

1. Initialize a table of size $m + 1 \times n + 1$
2. Set each entry $\text{opt}(0, j) = 0$
3. Set each entry $\text{opt}(i, 0) = 0$
4. Fill the table in row-major order, using the recursive relations specified above
5. Backtrack from $\text{opt}(m, n)$ to opt(0,0) to obtain the final result

Time Complexity

Runtime: $O(m \cdot n)$ [polynomial relative to input size $O(m + n)$]

6.4 RNA Sequencing

Recall: RNA sequences are sequences composed of elements A, U, C, and G, where A can only match with U and C can only match with G (and vice versa).

Problem (*RNA Sequencing*). Given a sequence of AUCG, what is the largest possible number of matches between elements in the sequence?

- Also called *Sequence Alignment*
- **Constraints**:
  1. *No sharp corners*: any two elements must be at least 4 characters in between them in order to match
     - May be generalized to values other than 4
  2. No crossing/overlapping matches

Approach

We can find solutions $\text{opt}(i, j)$ to the subproblem of finding the most number of matches between element i, element j in the sequence.

Observation. Assume that when considering the first i elements, we find a match t for element i; then any other matches can only be within the range $(1, t-1)$ or within the range $(t+1, i-1)$.
Notably, if $\text{opt}(1, i)$ matches i with t, we find that:

$$\text{opt}(1, i) = \text{opt}(1, t-1) + \text{opt}(t+1, i-1) + 1$$

For each value $\text{opt}(1, i)$:

1. Case 1: i is matched with another element t in $\text{opt}(1, i)$. Then $\text{opt}(1, i)$ must be the maximum of $\text{opt}(1, t-1) + \text{opt}(t+1, i-1) + 1$ across all possible values of t:

   $$\implies \text{opt}(1, i) = \max_{t<i-4}\{\text{opt}(1, t-1) + \text{opt}(t+1, i-1) + 1\}$$

   (Note: the max is across all elements $t < i-4$ that are eligible to match with i)

2. Case 2: i is not matched in $\text{opt}(1, i)$

   $$\implies \text{opt}(1, i) = \text{opt}(1, i-1)$$

Combining our two cases:

$$\text{opt}(1, i) = \max\{\max_{t<i-4}\{\text{opt}(1, t-1) + \text{opt}(t+1, i-1) + 1\}, \text{opt}(1, i-1)\}$$

Implementation

We can fill out $\text{opt}(i, j)$ for all intervals (i, j) in order of interval length: filling out all intervals of length 5, then all intervals of length 6, and so on. (We can set $\text{opt}(i, j) = 0$ for all intervals where $i < j - 4$)

Runtime: $O(n^2)$ possible intervals , each seen at most n times $\implies$ $\underline{O(n^3)}$

[Alternatively: $O(n^2)$ possible intervals $\cdot$ $O(n)$ for computing $\text{opt}(i, j)$ $\implies$ $O(n^3)$]

6.5 (∗) Bellman-Ford

Recall (***Shortest Path Problem***): Given a graph with weighted edges, find the minimum weight path from a vertex a to another vertex b.

- Solved by Djikstra's algorithm for graphs with no negative edges

Q: What about graphs with no negative edges?

A: A solution exists only if there are **no negative cycles**.

- If there is a negative cycle, then we could go around the cycle infinitely to obtain paths with infinitely negative weight.

  $\implies$ We assume there are no negative cycles.

Observation (1). Under the assumption of negative cycles, a minimum-weight path $a \to b$ will contain at most $(n-1)$ edges.

Proof. Let P be a path $a \to b$ with more than $(n-1)$ edges. Since P has more than $(n-1)$ edges, therefore P contains a cycle. Per our assumption of no negative cycles, this cycle will have non-negative weight. Assume, WLOG, that the cycle has positive weight; then we could obtain a path of less weight by removing the cycle, therefore P is not a minimum-weight path. □

Observation (2). Let x be a vertex adjacent to vertices $u_1, \ldots, u_i$. Then the shortest path $x \to b$ containing at most $\lambda + 1$ vertices will either be the shortest path $x \to b$ containing at most λ vertices, or a shortest path $u_j \to b$ containing λ vertices plus the edge (x, u_j).

Namely, let $\text{opt}(x, \lambda)$ be the minimum-weight path $x \to b$ containing at most λ vertices:

$$\text{opt}(x, \lambda + 1) = \min \left\{ \min_{1 \le j \le i} \left\{ \text{opt}(u_j, \lambda) + |(x, u_j)| \right\}, \text{opt}(x, \lambda) \right\}$$

Algorithm (Bellman-Ford)

Let $a = v_0, v_1, \ldots, b = v_n$ be the vertices of the graph.

1. Create a table of size $n + 1 \times n - 1$.
2. Set $(0, j) = +\infty \; \forall \; j \neq n$; set $(0, n) = 0$.
3. For each entry $(1, j)$:
   (a) Case 1: $\exists$ an edge (v_j, b); then we set $(1, j)$ to be the weight of that edge.
   (b) Case 2: $\nexists$ an edge (v_j, b); then we set $(1, j)$ to be $+\infty$.
4. For $1 \leq i \leq n$:
   - Let $v_{\lambda_1}, \ldots, v_{\lambda_k}$ be the vertices adjacent to v_i. Set:

$$(i, j) = \min \left\{ \min_{1 \leq t \leq k} \{(i - 1, \lambda_t) + |(v_i, v_{\lambda_t})|\}, (i - 1, j) \right\}$$

5. Output $(n, n - 1)$.

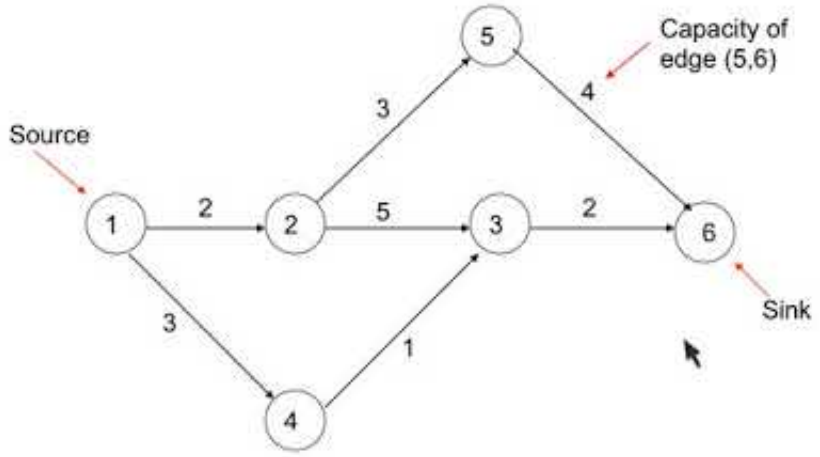
Runtime: $\mathrm{O}(V \cdot E)$

7 Network Flow

Introduction to Networks

Definition (***Network***)**.** A **network** is a weighted directed graph where each edge has an associated ***capacity*** [weight].

- An **S-T network** is a network with a specified starting vertex S, end vertex T.
- The capacity of an edge may be used to indicate the number of items that can be sent through that edge.
- Cycles are allowed.



Definition (***Flow***)**.** Given an S-T network, a **flow** is a function mapping each edge to a non-negative number. A ***legal flow*** is a flow such that the number assigned to each edge is less than or equal to the capacity of the edge.

Definition (***Max Flow***)**.** Given a flow F on an S-T network, we defined the ***capacity*** |F| of F to be the sum of the flow to T. We define a **max flow** on F to be a legal flow with maximum capacity.

7.1 Max Flow Problem

Problem (***Max Flow/Min Cut***)**.** Given an S-T network, find a max flow on the network.

Rules:

1. The flow through any edge is bounded by the capacity of the edge, i.e. $f_i \leq c_i \; \forall \; i$.
2. Edge capacities are *positive nonzero integers*.

3. *Conservation of flow*: The flow into any vertex needs to be equal to the flow out of it ($\sum_{\to v_i} f_j = \sum_{v_i \to} f_j$)
    - Exceptions: S, T

Solutions to Max Flow

Simple case: if the network is a single path, then the max flow is simply the minimum of capacities in the network.

Greedy principle: We can keep looking for paths $S \to T$ that do not yet contain any saturated edges [edges where flow = capacity] and saturating them, until no more paths exist.

Issue: Random path selection does not always provide a max flow, but $\nexists$ any criteria for path selection that would provide a max flow on every network.
Solution: We can modify greedy to allow it to "change its mind".

- *Mechanism*: Anytime the greedy algorithm picks an edge $(u \to v)$ and runs x amount of flow through it, we can create a *back edge* $(v \to u)$ with capacity x.
    - Graph with back edges called an *augmented graph/network* or *residual network*.
- Can repeat the greedy process on the augmented graph to find "augmented paths" $S \to T$, stopping when no more augmented paths exist.
    - *Interpretation*: Sending flow through a back edge $(v \to u)$ is equivalent to decreasing the flow sent through the original edge $(u \to v)$.

Algorithm (Ford-Fulkerson)

1. Begin by finding any path $S \to T$ in the graph, e.g. via BFS/DFS.
2. "Augment" the graph:
    (a) For each edge $(u \to v)$ on the path, increment the flow on that edge by 1 and create a *back edge* $(v \to u)$ with capacity 1.
        - If the edge $(v \to u)$ already exists, increase its capacity by 1.
    (b) "Delete" any saturated edges from the graph, to remove them from consideration when looking for additional paths $S \to T$.
3. Repeat the process on the new *augmented graph* until no more paths $S \to T$ exist. Then a max flow can be found by, for each edge $(v \to u)$ in the original network, assigning it the value $|(v \to u)| - |(u \to v)|$ from the augmented network.

Proof of Correctness

Definition. Given an S-T network, we define a **cut** [partition] of the network to be a bipartition of the vertices of the network into two sets S_1, S_2 such that $S \in S_1$ and $T \in S_2$.

- Given a cut (S, T), we define its ***capacity*** $C(S, T)$ to be the sum of the capacities of all edges $(x \to y)$ such that $x \in S_1$, $y \in S_2$.

  $\to$ A ***minimum cut*** is a cut with minimum capacity.

  - Notably, $C(S, T)$ does not include edges $(y \to x)$ s.t. $x \in S_1$, $y \in S_2$.

Observation. Given a network with max flow F: for any cut (S, T), we find that $|F| \leq C(S, T)$. In particular, $|F|$ is bounded above by the capacity of the minimum cut.

- *Reasoning*: All flow in a max flow F must pass from S_1 to S_2 at some point, but it is (by definition) not possible to send more than $C(S, T)$ flow between S_1, S_2.

Claim: The augmented greedy approach is optimal.

Proof. Let N be a network, f a flow in N. We want to show that the following are equivalent:

1. f is a max flow in N
2. The augmented network N_f (N with flow f) contains no unsaturated augmented paths.
3. $|f| = C(S, T)$ for some cut (S, T) of N

$1 \to 2$: Assume (1). Assume, for the sake of contradiction, that N_f contains an unsaturated augmented path; then we could saturate that path and obtain a larger flow [contradiction].

$2 \to 3$: Assume (2). Remove (from N_f) all saturated edges. Then S and T will become disconnected in the resulting graph.

- *Reasoning*: Since N_f contains no unsaturated augmented paths, then every path $S \to T$ in N_f must contain at least one saturated edge.

We can take the connected components containing S and T, respectively, as a cut (S, T) of N; then all edges between partitions are saturated in N_f, therefore $|f| = C(S, T)$.

$3 \to 1$: Assume (3). Since the capacity of any flow is bounded above by the capacity of any cut, then no flow f' can have $f' > |f| = C(S, T)$; therefore f is a max flow. □

Time Complexity

Let $N(V, E)$ be a network with max flow f. Then Ford-Fulkerson will need to find at most $|f|$ augmenting paths, where finding an augmenting path via BFS/DFS is done in $O(V + E)$ and augmenting a graph is, worst-case, $O(E)$.

$\to$ **Runtime**: $O(|f| \cdot E)$ (*pseudo-polynomial*)

7.2 Extensions of Max Flow

Cell Tower Problem

Problem (***Cell Tower Problem***)**.** Given a set of m cell phones and n cell towers (where each cell tower has a certain capacity), find a way to assign a maximum amount of cell phones to cell towers.

Assumptions:

1. Each cell tower has the same capacity x.
2. Each cell phone can be assigned to any tower within a certain distance R.

Approach: Can reframe as a network flow problem:

1. Represent cell phones, cell towers as individual nodes
    - One phone-tower connection $\Rightarrow$ one unit of flow
    - Can use edge capacities to enforce cell tower capacities

Algorithm

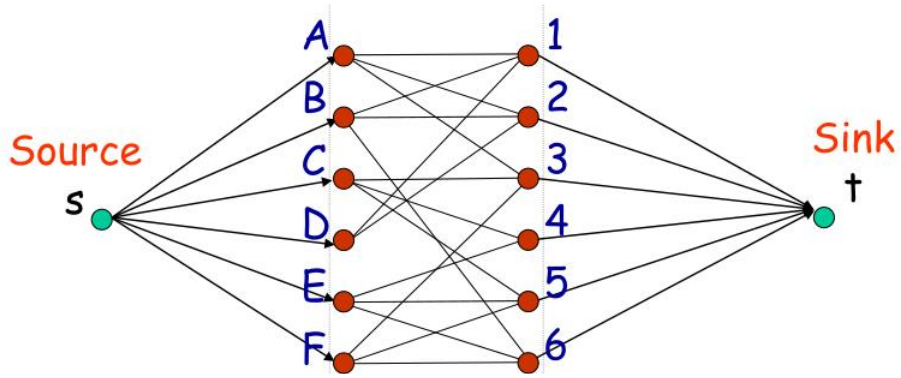
1. For each cell tower, create a corresponding node y in the graph
2. For each cell phone, create a corresponding node x in the graph
    - For each cell tower y_i within a distance R from x, create an edge $(x \to y_i)$ with capacity 1
3. Create a *virtual source* S
    - For each phone a_i, create an edge $(a_i \to S)$ with capacity 1
4. Create a *virtual sink* T
    - For each tower b_j, create an edge $(b_j \to T)$ with capacity x
5. Run Ford-Fulkerson from $S \to T$ on the graph.

Bipartite Matching

Problem (***Bipartite Matching***)**.** Given a graph partitioned into two sets $S_{\text{left}}, S_{\text{right}}$, find the maximum number of possible matches $S_{\text{left}} \to S_{\text{right}}$ such that each node is matched with at most one other node.

Algorithm

1. Create a *virtual source* S
    - For each node $x \in S_{\text{left}}$, create an edge $(S \to x)$ with capacity 1
2. Create a *virtual sink* T
    - For each node $y \in S_{\text{right}}$, create an edge $(y \to T)$ with capacity 1
3. Run Ford-Fulkerson from $S \to T$ on the graph.



"Gadgets"

If we want to create a restriction that at most n flow passes through a given node x in a network without modifying the capacities of adjacent edges, we can do so by splitting x into two nodes x_1, x_2 such that:

1. All edges $(v \to x)$ become edges $(v \to x_1)$
2. All edges $(x \to u)$ become edges $(x_2 \to u)$
3. x_1, x_2 are connected by an edge with capacity n

8 Complexity Theory

8.1 Problem Transformation

Principle (***Algorithmic Transformation***): We can compare the difficulty of different problems via the notion of transforming/reducing certain problems into other problems.

Definition. Let X, Y be two distinct problems. If we can convert an arbitrary instance of Y into an instance of X in polynomial time, then we say that Y is ***polynomial-time reducible*** to X, or $Y \leq_p X$.

- "Reduction" also encompasses conversions into a polynomial number of instances of X in polynomial time.
- *Notation*: "$Y \leq_p X$", "$Y \alpha_p X$" $\Rightarrow$ "there exists a polynomial-time reduction $Y \to X$"

Reasoning: If we have a polynomial-time solution to X, then the best solution to Y is no worse than polynomial-time [Y is no harder than X].

- *Alt*: If we cannot solve Y in polynomial time, then it must be the case that we cannot solve X in polynomial time either.

Note: We may say $Y \leq_p X$ in cases where Y is an *instance* or *subproblem* of X.

- Ex. (*Element uniqueness*): Let X be determining if there are any duplicates within a list; and let Y be determining if there are any duplicates within a sorted list. $Y \leq_p X$.
  - In this case: X is $O(N \log N)$; Y is $O(N)$
- Ex. (*Sorting*): Let Y be finding the lowest element in a list of size n $[O(N)]$; let X be sorting a list of size n $[O(N \log N)]$. Since we can solve Y by solving X, then Y is reducible to X: $Y \leq_p X$

Applications for Problem Transformation

Case 1. We have a problem we already know how to solve; then, given a new problem, we want to transform it to an already-solved problem to obtain a solution.

- Ex: Since bipartite matching can be reduced to network flow, therefore having a solution to network flow gives us a solution to bipartite matching.

Case 2. We have a problem we already know is hard [intractable]. Then, given a new problem: if we can reduce the old problem to the new problem, then the new problem must also be hard.

- Argument [by contradiction]: If the new problem were easy and we could reduce the old problem to the new one; then the old problem must also be easy.

Note: Polynomial-time reductions rely on the transformation itself being polynomial-time.

- Ex: If $X \leq_{exp} Y$ (X is exponential-time transformable to Y), then even if Y can be solved in polynomial time, it may not be the case that X is solvable in polynomial time.

Max-Clique & Independent Set

Definition. Given a graph $G(V, E)$:

- A ***clique*** in G is a set $S \subset V$ that is pairwise connected (every vertex in the set is connected to every other vertex)
- An ***independent subset*** in G is a set $S \subset V$ such that no two vertices in the set are connected to each other.

Problem (***Max Clique***)**.** Given a graph G, find the largest clique in G.

Problem (***Max Independent Set***)**.** Given a graph G, find the largest independent set in G.

Claim: Max Clique $\leq_p$ *Max Independent Set*

Proof. Given an arbitrary instance of Max Clique, we want to show that we can transform it into an instance of Max Independent Set in polynomial time.

1. Start with an arbitrary instance of max-clique on a graph G.
2. *Transformation*: We can construct a complement $\bar{G}(\overline{V}, \overline{E})$ of $G(V, E)$:
   (a) Set $\overline{V} = V$
   (b) For every possible pair of vertices $x, y \in V$:
      - If $(x, y) \notin E$, then we add it to $\overline{E}$
      - If $(x, y) \in E$, then we omit it from $\overline{E}$
3. *Result*: Finding a max clique in G is equivalent to finding a max independent set in $\bar{G}$.
   - *Argument*: Given a max clique S in $G(V, E)$, for every pair of vertices $x, y \in S$, $(x, y) \in E$; then $(x, y) \notin \overline{E}$ for any $x, y \in S$, therefore S is an independent set in $\bar{G}$.
     – This shows that a max clique in G is an independent set in $\bar{G}$; we can show that a max independent set in $\bar{G}$ is a clique in G via similar argument.

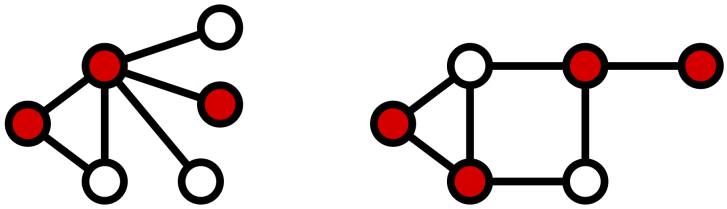
We observe that the transformation step is $O(V^2)$; then we can convert an arbitrary instance of Max Clique into an instance of Max Independent Set in polynomial time. □

Consequence: If maximum clique is not solvable in polynomial time, then maximum independent set cannot be either.

- ***Note***: Both Max Clique, Max Independent Set are actually NP-complete.

Set Cover & Vertex Cover

Definition (***Vertex Cover***)**.** Given a graph $G(V, E)$, a ***vertex cover*** of G is a set $S \subset V$ such that for all edges $(x, y) \in E$, at least one of x, $y \in S$.



Problem (***Vertex Cover***)**.** Given graph G, find the vertex cover of minimal cardinality in G.

Definition (***Set Cover***)**.** Given a set of sets $S = \{A, B, \ldots\}$ with $U = \bigcup_{X \in S} X$: a ***set cover*** is a set $C \subset S$ such that $\bigcup_{X \in C} X = U$.

Problem (***Set Cover***)**.** Given a set of sets S, find the set cover of minimal cardinality in S.

Claim: Vertex Cover $\leq_p$ *Set Cover*

Proof. Given an arbitrary instance of Vertex Cover, we want to show that we can transform it into an instance of Set Cover in polynomial time.

1. Start with an arbitrary instance of vertex cover on a graph G with vertices $A, B, C, \ldots$
2. *Transformation*: For each vertex $x \in G$:
   (a) Create a corresponding set $X' = \{e_1, e_2, \ldots, e_i\}$, where $e_1, e_2, \ldots, e_i$ are the edges in G adjacent to x
3. *Result*: Finding a vertex cover for G' is equivalent to finding a set cover for the sets $A', B', C', \ldots$

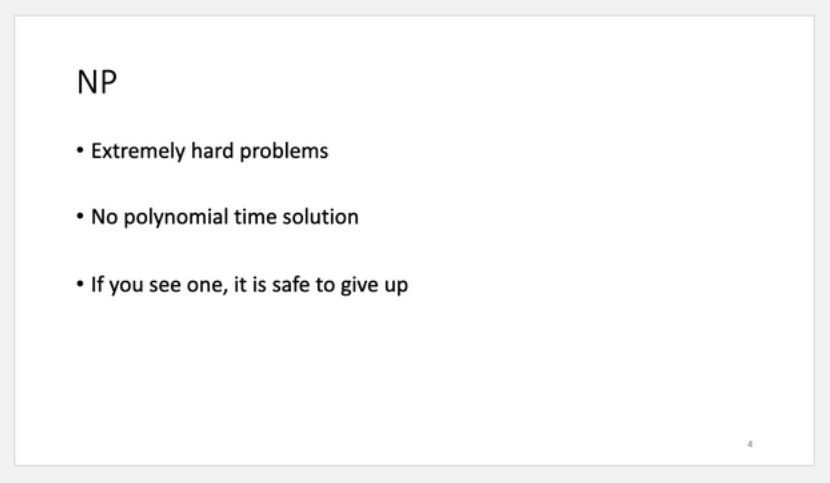
□

Consequence: If Vertex Cover is not solvable in polynomial time, then Set Cover cannot be either, since solving vertex cover is equivalent to solving set cover.

8.2 *NP*

Complexity Classes

- ***P***: Problems that can be solved in polynomial time
- ***NP***: Problems for which solutions can be verified in polynomial time
  - ***NP*-Complete**: The hardest problems in NP
- ***NP*-Hard**: Problems at least as hard as the hardest problems in NP
  - A problem X is considered NP**-Hard** if all problems $Y \in NP$ can be reduced to X.



Definition (***NP*-Complete**)**.** A problem $X \in NP$ is considered ***NP*-Complete** if all problems $Y \in NP$ can be reduced to X. Namely, $Y \leq_p X$.

- **Consequence**: If any NP-Complete problem can be solved in polynomial time, then it must be the case that all problems in NP can be solved in polynomial time.
- **Observation**: All NP-Complete problems are also NP-Hard.

Observation: $P \subseteq NP$

- **Reasoning**: If we can solve a problem in polynomial time, then any solution can be verified in polynomial time simply by resolving the problem.
- **Problem (*unsolved*)**: Is P = NP?
  - General consensus: Probably not

Note: "NP" = "Non-deterministic, polynomial-time Turing machine"

***NP*-Completeness**

Principle: As an alternative to solving a problem, we can instead prove that it is "difficult"; one way to do this is to show that the problem is equivalent to a known NP-Complete problem.

- *Ex*: Let X be a new problem, and let Y be a known *NP*-Complete problem. If it is possible to polynomial-time transform Y into X [$Y \leq_p X$], then X must itself be *NP*-Complete.

Problem (***Traveling Salesman***)**.** Given a starting city and a number of cities with weighted edges in between, find the shortest path that visits every city.

- This is doable in exponential time (n cities $\rightarrow n!$ permutations), but not polynomial time.

Problem (***Satisfiability***)**.** Given a set of Boolean clauses $C_1, \ldots, C_n$ involving a set of Boolean variables $x_1, \ldots, x_n$, is there a set of values for $x_1, \ldots, x_n$ satisfying all clauses?

- We say that a set of variables *satisfies* a function if it causes the function to return True.
- **Problem** (***3-Satisfiability***). Identical to Satisfiability, but with the added constraint that the Boolean clauses must be of length 3.
  - Ex (Boolean function): $F = (x_1 \,||\, \bar{x_2} \,||\, x_3)$ && $(\bar{x_1} \,||\, x_4 \,||\, x_5)$

Additional Problems (*NP*-Complete):

- Independent Set, Clique
- Set Cover, Vertex Cover

Note: The *"decision versions"* of the problems (ex: given $k \in \mathbb{N}$, does there exist a clique of size $\geq k$?) are considered *NP*-Complete; the *"optimization versions"* (find the maximum clique) are *NP*-Hard.