CS180: Algorithms

Professor Sarrafzadeh

Thilan Tran

Fall 2019

Contents

CS180: Algorithms

Time Complexity

The definition of the **time complexity** $O(f(n))$ for some function $T(n)$ is as follows:

$$T(n) = O(f(n))$$
$$\textit{if}\, \exists c \geq 0, n_0 \geq 0 \textit{ s.t. } T(n) \leq cf(n) \forall n \geq n_0$$

Using n_0 addresses the order only for large values of n.

- thus **order** notation denotes the *upper* bound of a function
  - Ω notation is the *lower* bound of a function (ie. the optimal bound)
  - Θ notation is the *exact* bound of a function

Greedy

- the **greedy** paradigm is a problem-solving approach where the solution set is *greedily* minimized by repeatedly eliminating a possibility from consideration *without global analysis*:
  - *pros*:
    - very fast, since there is no need for global analysis
  - *cons*:
    - more difficult to prove correctness for, since there is no global analysis

Famous Problem

Problem:

- define a **famous** person as someone who *everyone* else knows, but knows *no-one* else:
  - the **model of computation (MOC)** or basic set of permitted operations for this problem is asking a *pair* of people at a time if they know the other
  - note that there cannot be two famous people in a room, since they would have to know each other
  - in addition, there may *not* be a famous person in the room
- find if there is a famous person in a room of n people

Solution #1:

1. *repeatedly*, pick an arbitrary person p:
   - for every other person p', ask if p knows p'
     - if p knows p', p can't be famous
   - then, for every other person p', ask if p' knows p
     - if p' doesn't know p, p can't be famous

Analysis #1:

- for each candidate, $2 \times (n-1)$ questions are asked
- in the worst case, $2n \times (n-1)$ questions are asked
- thus algorithm #1 has a complexity of $O(n^2)$

Optimization:

- is it possible to improve on solution #1?:
  - when considering every pair, there are $\binom{n}{2} \approx n^2$

    - but not *every* pair is needed for an algorithm to be succesful
- use the greedy paradigm, and try to reduce the problem size by one repeatedly
    - ie. eliminate one person from being famous with every question

Solution #2:

1. *repeatedly*, pick two arbitrary people a and b, ask if a knows b:
    - if a knows b, then a *cannot* be famous
    - otherwise, if a does not know b, then b *cannot* be famous
    - whatever person is *eliminated* will not be picked again in this step
2. after $n - 1$ questions, only one candidate c remains:
    - check if c knows no-one else ($n - 1$ questions)
    - check if everyone else knows c ($n - 1$ more questions)

Analysis #2:

- $3 \times (n - 1)$ questions are asked
- thus solution #2 has a complexity of $O(n)$
- the *lower bound* for complexity is n since everybody must be asked a question once
    - thus the algorithm is *optimal*

Matching Problem

Problem:

- given 2 groups of n people each:
    - every person in each group has an ordered **preference list** containing every other person in the other group
        - no repeats, *strict* ordering
    - a person **prefers** another person p over another p' if p is ranked higher then p' in their preference list
        - ie. $p > p'$
    - in a **perfect matching**, there are n matches and every person is matched with a single other person
    - in a **unstable match**, $m1$ is paired with $w1$ and $m2$ is paired with $w2$, but $m1$ prefers $w2$ and $w2$ prefers $m1$
    - in a **stable matching**, there are no unstable matches for any two pairs in a perfect matching
- find a **stable matching**

Solution:

1. *repeatedly*, pick an arbitrary person p from the first group who is unmatched:

    - go down their preference list starting at the highest rank that hasn't yet been asked by p (this solution never goes *backwards* in the list)
    - for each person p' in the list:
        - if p' has not yet been engaged in a match, p and p' will become engaged to each other
        - otherwise, if p' is already engaged in a match, check the preference list of p':
            - if p is prefered over the current engaged match of p', p and p' will become engaged to each other
            - the previous match of p' becomes unmatched
            - otherwise, p' will refuse the engagement
    - algorithm terminates when every person in the first group is matched

Analysis:

- in the worst case, every person goes through n other potential matches
    - thus the algorithm has a complexity of $O(n^2)$

Proof:

- *properties* of the algorithm:
    1. in the first group, people can only become matched with *lower* ranked people as engagements break off
    2. in the second group, people can only become matched with *higher* ranked people as they are asked by different people
- without loss of generality, consider 2 arbitrary pairs from the first group, m with w and m' with w'
- assume by contradiction, this pair has an unstable matching and m prefers w' and w' prefers m
    - ie. for m, $w' > w$ and for w', $m > m'$
- did m propose to w' before w?
    - if *not*, then $w > w'$ in the preference ranking of m and this is a contradiction on the assumption
    - if so, then, the engagement between m and w' could only have been broken by some $m'' > m$ and then subsequently $m' > m''$
        - however, this implies that $m' > m$, which is a contradiction on the assumption
- thus no pairs can have an unstable matching and the solution is correct

Majority Problem

Problem:

- find who has a majority out of n votes for m possible candidates, where $n \geq$

m

  - there can be only 0 or 1 majority

Solution:

1. *repeatedly*, eliminate two *different* votes
2. continue until 1 or 0 votes are left
3. verify that the remaining candidate does have a majority by going through the original list

Analysis:

- to *implement*:
  - want to avoid a linear search for two different votes each time
  - want to avoid an overall $O(nlogn)$ solution, could then trivially sort the votes
- instead, maintain a *count* of votes for a current candidate:
  - linearly go through the votes, *incrementing* the count if the vote is for the current candidate, otherwise *decrementing*
  - when the count drops to 0, switch the current candidate to the candidate of the next vote
- again, after traversing the list, go back and verify that the number of votes for the candidate is $> \frac{n}{2}$
- overall, algorithm has a complexity of $O(n)$

Proof:

- *claim*: removing two different votes *maintains* a majority
  - initially, $\frac{n}{2} + 1$ out of n votes is a majority
  - if two different votes are removed and one vote for the majority was removed, the ratio becomes $\frac{n}{2}$ out of $n - 2$ votes
    * otherwise, if the removed votes were not for the majority, there are still $\frac{n}{2} + 1$ votes for the majority
  - the same majority is maintained in either case
- thus the algorithm is correct, as it never disrupts the majority

Interval Scheduling Problem

Problem:

- various tasks or **intervals** $\{I_1, ..., I_n\}$ with a start and end time $I_i = (l_i, r_i)$ overlap and conflict in time
- find a subset of intervals that do not overlap in time, and *maximize* the number of intervals selected

Solution #1:

1. *repeatedly*, pick the next *shortest* interval
    - eliminate any overlapping intervals
    - algorithm terminates when no intervals are left to consider

Analysis #1:

- the algorithm is incorrect
- consider the following counterexample:
    - three intervals where the middle one is the shortest, but overlaps with the other otherwise non-overlapping two
    - the algorithm would incorrectly select only the shortest middle interval, instead of the other two intervals

Solution #2:

1. *repeatedly*, pick the next *longest* interval
    - eliminate any overlapping intervals
    - algorithm terminates when no intervals are left to consider

Analysis #2:

- the algorithm is incorrect
- a similar counterexample to the first algorithm can be constructed

Solution #3:

1. *repeatedly*, pick the next *earliest* interval
    - eliminate any overlapping intervals
    - algorithm terminates when no intervals are left to consider

Analysis #3:

- the algorithm is incorrect
- consider the following counterexample:
    - three intervals where the earliest one overlaps with the other two otherwise non-overlapping intervals
    - the algorithm would incorrectly select only the earliest interval, instead of the other two intervals

Solution #4:

1. *repeatedly*, pick the next interval that *ends earliest*
    - eliminate any overlapping intervals
    - algorithm terminates when no intervals are left to consider

Analysis #4:

- the algorithm is optimal

- to *implement*:
  - sort the intervals by end time
  - linearly traverse through the sorted intervals and build a solution set:
    - ignore any intervals whose start times come before the end time of the current interval
    - add the first valid, unoverlapping interval to the solution set and treat it as the current interval
- thus the algorithm has a complexity of $O(nlogn + n) = O(nlogn)$

Proof:

- assume that the algorithm is not optimal, and returns the set of intervals $\{A_1, ..., A_k\}$
- then there exists a more optimal set of intervals $\{O_1, ..., O_m\}$ where $m > k$

Using an exchange argument:

- find the ith interval where the optimal solution diverges:
  - here we can use an *exchange* argument
  - since the algorithm always chooses the interval that ends earliest, we can *repeatedly* build a new solution as follows:
    - A_i will not overlap with any previous intervals in O, since the solutions have matched up to here
    - A_i will not overlap and invalidate any of the *next* intervals in O, since the algorithm always chooses the interval that ends the earliest
    - thus we can safely replace O_i with A_i
  - this exchange argument continues until we have converted the entirety of O into A
- the optimal solution O thus *cannot* have more intervals, since using our exchange argument, our algorithm would have selected the extra intervals as well

Alternatively, an argument for the algorithm *staying ahead* of the optimal solution can be used:

- *claim*: for all intervals in the algorithm's solution set, the algorithm *stays ahead* ie. will always end at the same time or earlier than the optimal solution's corresponding interval
  - this is evident from the function of the algorithm, since the earliest ending interval will always be selected, at any stage of the algorithm
- *claim*: O cannot have more intervals than A
  - using the previous claim, if O had more intervals, A would have chosen the extra intervals as well

Another proof method is finding a structural *bound* that each optimal solution would have.

Encoding Huffman Trees

Problem:

- when *encoding* data, a desirable encoding system is *uniquely* decodable, and doesn't use too much memory
- can use a **tree code** to encode the data in the leaves of the tree:
  - eg. going left in the tree is encoded as a 0, and going right is encoded as a 1
  - thus no encoding is a *prefix* of another, and all encodings are unique
  - additionally, if this tree is *balanced*, there will be an smaller number of 0's and 1's used
- if the *frequencies* of letters ie. encoded data is given, the tree should be further optimized
  - letters that have a higher frequency should have a *shorter* encoding so that less memory is used
- *minimize* the function $\sum_{i=1}^{n} l_i f_i$ where f_i is a letter's frequency and l_i is its encoded length in the tree code

Solution:

1. sort the elements by their frequencies
2. *repeatedly*, combine the lowest two elements and their frequencies
   - ie. coalesce the elements in a tree under a single parent
   - continue the algorithm with the parent element and their *added* frequencies
   - the algorithm terminates when the tree is complete ie. *rooted*
   - note that the resultant tree is not balanced, but does minimize the desired function

Analysis:

- sort followed by linear traversal
  - thus the algorithm has a complexity of $O(nlogn)$

Graphs

- a **graph** consists of a collection of **vertices** or **nodes**, with connections between vertices called **edges** or **links**:
  - notation:
    - $G = (V, E), V = \{a, b, c, ...\}, E = \{(a, b), (b, c), ...\}$
    - usually n or v nodes and m or e edges
  - edges may be **undirected** or **directed**
  - edges may be **unweighted** or **weighted**
- a **cycle** is a path where we start and end at the same vertex without repeating edges
- in a **connected** graph, every vertex can be reached through a path from every other vertex:
  - otherwise, it is **disconnected** and may have multiple connected **components** within
  - the minimum number of edges for a single connected component of n vertices is $n - 1$
    - such a graph is called a **tree**, and has no cycles
  - for a directed graph, **strong connectivity** is used instead
    - two vertices a, b are strongly connected if there is a path from a to b and vice versa
- the sum of degrees in a tree is $2(n - 1)$
  - can be proved by inductively building a tree
- graphs can be *represented* in different ways:
  - using a matrix or 2D-array:
    - n by n matrix, each element represents whether an edge exists between two vertices
    - *pros*:
      - better for denser graphs with many edges
    - *cons*:
      - more difficult to add vertices
      - more difficult to check *all* adjacencies of a particular vertex
  - using an **adjacency list**:
    - an n element array of linked lists, each list represents the adjacent vertices to a vertex
    - *pros*:
      - better for sparse graphs with few edges
      - less memory used
      - easy to add vertices
    - *cons*:
      - nonconstant access time to check if an edge exists between two

vertices

Breadth-First Search

- **Breadth-First Search (BFS)** algorithm:
  - search all nodes distance 1 away from an origin vertex, ie. adjacent nodes to origin
  - search all nodes distance 2 away ...
  - etc.
- a **BFS tree** can be formed out of the edges used to visit a vertex in a BFS for the *first* time
- every edge will be examined *twice* (checking adjacent edges to every vertex)
  - thus BFS has a complexity of $O(m)$ if a connected graph, and $O(m+n)$ if multiple components
- BFS tree *properties*:
  - *claim*: the distance between the source s and any other vertices in the BFS tree is the *minimum* distance between the same vertices in the original graph
    - ie. the **level** of a vertex in the BFS tree is its minimum distance from the origin
    - by contradiction, assume that there is a *shorter* path between s and some vertex v in the original graph
    - this is impossible, since v would have ended up with a *higher* or lesser level than it did in the BFS tree
  - note that the BFS tree only finds shortest paths *starting* from a given node
    - would have to rerun the algorithm n times to get *all* shortest paths
- BFS *implementation*:
  - use a FIFO queue
  - start with s in the queue
  - *repeatedly*, pop the first vertex from the queue, and add all its *unexplored* neighbors to the queue
    - need some sort of hash set to keep track of unexplored neighbors

Depth-First Search

- **Depth-First Search (DFS)** algorithm:
  - go as *deep* as possible in a certain direction, instead of returning to previous adjacent nodes
  - on a *dead end*, backtrack to nodes with unvisited neighbors

- note that DFS can be useful for finding cycles:
  - if a previously visited neighbor is found while exploring new nodes, a cycle exists
- a **DFS tree** can be formed from all the edges used to *discover* a node in a DFS
- every edge will be examined at least once
  - thus DFS also has a complexity of $O(m+n)$
- DFS tree *properties*:
  - narrower and deeper than a BFS tree
  - *claim*: if the edge $(x, y) \in G$ but $(x, y) \notin$ the DFS tree, then one of x or y is an ancestor of the other
    - an edge will not be in the DFS tree if the incident nodes of the edge were already explored
    - any explored nodes must be *descendants* of the original node
- DFS *implementation*:
  1. analagous to BFS implementation, with a LIFO stack instead of a queue
  2. can naturally use recursion to start from a root and explore each neighboring node as long as it has not been visited

Coloring Problem

Problem:

- find the *minimum* number of colors to color the nodes on a graph so that no adjacent nodes are the same color

Solution #1:

1. choose an arbitrary node and give it the first color
2. *repeatedly*, for each of its neighbors, give them a different color than their neighbors
3. continue until all nodes are colored

Analysis #1:

- this algorithm is incorrect
  - the problem is actually NP-complete and *cannot* be solved in polynomial time

Revised Problem:

- detect if a graph is 2-colorable ie. can be colored in 2 colors
  - note that not all graphs are 2-colorable, since cycles of odd length *cannot* be two-colorable
  - note that finding if a graph is **bipartite** is the same as finding if a graph is 2-colorable

        * same as labelling each node to one of two groups, with no adjacent nodes in the same group

Solution:

1. run BFS on the graph from an arbitrary node, and create a BFS tree:
    - note that there are only two types of edges in the resultant tree:
        - edges in the same level
        - edges that go a single level above or below
    - if an edge spanned 2+ levels, the nodes in the later levels would have been discovered earlier
2. run through the BFS tree, and if an edge between two nodes in the same level is encountered, the graph is not 2-colorable

Analysis:

- only BFS and linear traversal through the graph
    - thus the algorithm has a complexity of $O(m+n)$

Proof:

- *claim*: an edge existing between two nodes in the same level indicates the existence of an odd cycle in the graph:
    - we can construct an odd cycle by following the adjacent nodes back up to the root node
    - the length of this cycle is $1+2\times k$, where k is the level of the two nodes
    - thus the cycle is of odd length, and the graph is not 2-colorable

Topological Sorting

- solving *precedence* relationships with directed graphs
- **topologically sorting** or **ordering** a directed graph is outputting a linear ordering of the nodes, where if $\exists$ *edge* (a,b), a *must* come before b in the ordering
    - note that the number of orderings on a graph is *non-polynomial*
- is there always a topological ordering for a directed graph (DG)?
    - no, if there is a cycle in the DG, no node can come first, so there is a contradiction on the topological ordering
    - thus not every DG has a topological ordering, but a **directed acyclic graph (DAG)** will *always* have a topological sort

Problem:

- generate a topological ordering for a DAG

Solution:

1. *repeatedly*, choose an arbitrary **source** node, output the node, and delete it (and any of its incident edges)
    - a source node has no incoming edges, and there can be multiple in a graph
    - the algorithm terminates when every node is output

Analysis:

- to efficiently implement this algorithm, we want to minimize searching for source nodes:
    1. count the number of all incident edges to all nodes, and note which nodes are sources - $O(m)$
    2. create a list of sources - $O(n)$
    3. at each iteration of the algorithm, the only nodes that can potentially become new sources are the ones adjacent to the current source:
        - each iteration should update the count of incident edges to the selected source
        - in addition, every edge will be only considered *once* when we choose its parent node as the current source - $O(m)$
- thus the algorithm has a complexity of $O(m+n)$

Dijkstra's Shortest Path Algorithm

Problem:

- in a weighted graph, find the shortest path between two vertices
    - the distance of a path is defined as the sum of the weights of the edges in the path

Dijkstra's Shortest Path Algorithm:

1. start at a root vertex s
    - place s in the set of vertices with *finalized* minimum distances from s, ie. the final minimum distance from s to s is 0
2. *repeatedly*, grow the set of finalized vertices:
    - find the *unfinalized* vertex v with minimum distance, and add it to the finalized set
        - then, *relax* the distance to each of its neighbors u as follows:
            * $d(u) = min(d(u), d(v) + l(e))$, where $e = (v, u)$ and $d(v)$ is the minimum distance so far from s to vertex v
    - ie. finding the next closest neighbor, finalizing its distance, and updating its neighbors' distances
    - the algorithm terminates when every vertex's distance has been finalized

Implementation:

- there are two types of nodes that have to be maintained throughout the algorithm:
  - *finalized* and *non-finalized* nodes
  - for non-finalized nodes, we want to keep track of any nodes that are *neighboring* a finalized node (they are a candidate to be finalized in the next iteration):
    - have to be able to efficiently find the next minimum distance to a neighbor
    - have to be able to efficiently update the neighbors throughout the algorithm
  - usually, the non-finalized nodes are kept in a data structure, and any other nodes are considered finalized

Finding the minimum linearly:

- at each iteration:
  - to find the next minimum neighbor, just check all non-finalized nodes - $O(n)$
  - then, update the distances for all adjacent vertices to the minimum neighbor - $O(n)$
- altogether, this version of the implementation has complexity $O(n^2)$
  - better for *dense* graphs

Using a priority queue / minimum heap:

- at each iteration:
  - to find the next minimum neighbor, just pop the min-heap - $O(1)$
    - have to *reheapify* the heap - $O(logn)$ for a balanced heap
  - then, update the distances to all adjacent vertices to the minimum neighbor - $O(logn)$ for a single update key
    - overall, an update will occur for a vertex on the tail of every edge - $O(mlogn) = O(mlogm)$
- altogether, this version of the implementation has complexity $O(mlogm)$
  - better for *sparse* graphs

Minimum Spanning Tree

- a **spanning tree** is a tree that connects all vertices:
  - will have $n - 1$ edges by the definition of a tree
  - can run a BFS or DFS traversal
- a **minimum spanning tree** is a spanning tree with the *minimum* sum of weighted edges

    - not as simple as repeatedly removing the maximum weight edge
- **MST Theorem**:
    - for a graph G, let w_{min} be the minimum weight edge between two partitions of G, L and R
    - for an MST T on G, $w_{min} \in T \forall$ bipartitions of G
    - *proof*:
        * take an arbitrary partition with w_{min}
        * assume by contradiction that there is an MST without w_{min}
        * consider the MST has another edge w_x ($w_{min} < w_x$ by definition) connecting the two partitions
        * create a new MST using w_{min} isntead of w_x
        * this is a contradiction since this new MST has a smaller sum of edges
    - analagous *cut property*: for any cycle c and its most expensive edge e, then e does not belong to any MST
        * similar proof by contradiction

Problem:

- find an MST for a graph

Prim's Algorithm:

- an adaptation of Dijkstra's algorithm:

1. start at an arbitrary node s and greedily grow a tree
2. *repeatedly*, at each step, add the node that can be attached the cheapest (different *relax* function that only cares about edge weight instead of accumulated edges)

- the algorithm has complexity $O(mlogm)$

Kruskall's Algorithm:

1. *repeatedly*, insert edges in *increasing* cost, as long as increasing the edge would not create a cycle

- actual implementation is more complex than just reusing Dijkstra's
- have to maintain connected components so that it is easy to check for cycles when inserting edges:
    - need a data structure to efficiently *merge* components and find the *identity* of a component
        * called the **union-find** data structure, complexity of Kruskall's depends on the operation of this structure
    - at each step, use union-find to check that two nodes connected by an edge are *not* in the same group
        * if so, a cycle would exist, so skip this edge

        - if not, add the edge and union the two nodes' groups
    - continue until $n - 1$ edges in the tree
- complexity breakdown:
    - sort edges by cost - $O(mlogm)$
    - m finds - $O(m \times logm)$
    - m unions - $O(m \times 1)$
    - overall, the algorithm has complexity $O(mlogm)$

Union-Find Problem

- desired operations:
    - **make**: make all groups
        - every element starts as a single group
    - **find** or query: are two elements part of the same group
    - **union**: combining groups, ie. take the union of groups
        - groups cannot be broken
        - elements only remember their latest group
    - naming convention is that every group's *name* is one of the members of the group (arbitrarily)

Array / linear implementation:

- maintain an array of every element and their current group
- also maintain an array for every group of all of its elements to more easily update on unions
- unions only take the name of the *larger* set so that there are fewer updates
- operation complexities:
    - make - $O(n)$
    - find - $O(1)$
    - union - $O(n)$ on worst case
        - but when amortized, approaches $O(klogk)$
- still too slow, we want to optimize for unions and finds

Tree implementation:

- for every group, build a directed, rooted tree with a root as the group leader
    - initially, all elements point to themselves
- on a union, update the root of the *smaller* height tree and point it to the root of the other tree
- on a find, have to follow the sequence of pointers up the tree:
    - *claim*: the height of every tree is always $logn$ for n nodes in the tree or better
        - assume the claim holds

        * have to make sure that operations do not *invalidate* the assumption
        * on a union, the lower height tree is pointed to the root of the larger height tree
        * thus the assumption holds *unless* the trees are the same height h with 2^h nodes each:
            · then, we still point one tree to the other arbitrarily
            · the new tree has height $h + 1$ with $2(2^h)$ nodes, and the assumption still holds

- operation complexities:
    - make - $O(n)$
    - find - $O(logn)$
    - union - $O(1)$
        * the usage pattern is usually a find operation, then a union operation having found the root ie. name and height of groups
    - improved complexities for Kruskall's

Clustering

- putting *similar* data points in the same group, and *disimilar* data in different groups

Problem:

- given graph G and k desired clusters, place nodes close to each other in the same cluster and nodes far away in different clusters
    - every cluster must have ≥ 1 node
    - the **distance** between two clusters is the shortest distance (or weight of edge) between two nodes in each cluster, ie. the closest pair
- *maximize* the minimum distance, ie. spread out clusters as much as possible

Solution:

- another algorithm that utilized a form of clustering was Kruskall's

1. run Kruskall's algorithm until k clusters are formed
    - ie. equivalently, until k groups remain in the union-find structure

Proof:

- assume that Kruskall's returns clusters $\{c_1, c_2, ..., c_k\}$:
    - must stop at k clusters, as controlled by the algorithm
    - let $d*$ be the weight of the edge between the *closest* pair of clusters
        * this is the minimum distance we are trying to maximize
    - note that the next edge that *would have* been added by Kruskall's must be $d*$, since the list of edges is sorted by weight

        * additionally, any edge *not* in a cluster must be greater than every other edge *already considered* by Kruskall's
- *claim*: $\{c_1, c_2, ..., c_k\}$ is an optimal clustering
    - suppose by contradiction, a more optimal clustering exists $\{c'_1, c'_2, ..., c'_3\}$ with a $d*' > d*$
    - note that there must be a cluster c_r that is composed of parts of c'_i and c'_j, because otherwise, the clusterings would be identical
        * ie. there must be some difference between the optimal and algorithm's clusters
    - thus the optimal clustering is *cutting* one of the Kruskall clusters c_r:
        * however, all edges in c_r must have been already considered by Kruskall's before $d*$
        * thus all edges in c_r have a lower weight than $d*$
        * thus the optimal clustering has a $d*' < d*$, and is not actually optimal
    - thus the algorithm is correct

Analysis:

- same runtime as Kruskall's with union-find
    - thus the algorithm has a complexity of $O(mlogm)$

Other Graph Problems

Problem:

- if a DAG G has a longest path of length k, partition G's vertices into $k + 1$ groups such that there are no edges and no paths between each group

Solution:

- similar to a topological sorting, except we want to *pack* as many source vertices as possible together
    - at each step, instead of outputting an ordering one vertex at a time, create a *group* of vertices by picking *many* sources at once

Analysis:

- the algorithm is just a modified topological sort
    - thus the algorithm has a complexity of $O(m + n)$

Proof:

- by the definition of a source, there are no edges between sources, so the condition of independent groups is satisfied

  - in addition, if k is the longest path, removing all sources at a time would reduce the path by 1 each iteration and allow for $k + 1$ groups

Problem:

- **articulation points** are points whose removal disconnects the graph
- find articulation points of a graph
  - note that to find a non-articulation point, simple solution:
    - run BFS and remove *leaves*

Solution (sketch):

- *rules* for articulation points (AP):
  - if a root has ≥ 2 children, the root is an AP
  - if a vertex v has a subtree that does *not* have an edge that climbs higher than v (ie. a backedge), then v is an AP
- use an adapted DFS that keeps track of $Low(v)$, the lowest value of a subtree that is accessible by a backedge

Divide and Conquer

- the **divide and conquer** paradigm is a problem-solving approach where we break a problem into multiple pieces, and reform the pieces after:
  - dividing should be *easy* and done recursively
    - should *eventually* reduce the problem to a trivial solution
  - merging pieces together should be *easy* as well

Merge Sort

Problem:

- sort a list of elements

Solution:

1. divide list of elements in half recursively
   - this reduces the problem set to a trivial base case, ie. a list with one element is already sorted
2. merge all the sorted lists together

Merging algorithm:

- to merge together two sorted lists:

1. initialize pointers to the start of each list
2. *repeatedly*, compare the pointers, put the smaller element into the output list, and increment that pointer

Analysis:

- with the merging algorithm, every element in each list is considered once - $O(m+n)$ where m and n are the sizes of each list
- we can calculate the overall recursive time complexity as follows: (assuming the merging is done linearly, in time cn)

$$\begin{aligned}
T(n) &= T(\frac{n}{2}) + T(\frac{n}{2}) + cn && (1)\\
&= 2T(\frac{n}{2}) + cn && (2)\\
&= 2[2T(\frac{n}{2 \cdot 2}) + \frac{cn}{2}] + cn && (3)\\
&= 2^i T(\frac{n}{2^i}) + icn && (4)\\
&= 2^{logn} T(\frac{n}{2^{logn}}) + cnlogn && (5)\\
&= nT(1) + cnlogn && (6)\\
&= O(nlogn) && (7)
\end{aligned}$$

- thus merge sort has a complexity of $O(nlogn)$

Binary Search complexity:

- to *binary search* for a target in a sorted list of elements:

1. check if the target is the same, above, or below the middle element
    - if the same, search is succesful
2. recurse to the corresponding half of the list

- the time to *merge* divided solutions is constant, ie. performing a comparison of the middle element
    - thus we can calculate the overall recursive time complexity as follows:

$$\begin{aligned}
T(n) &= T(\frac{n}{2}) + c && (8)\\
&= [T(\frac{n}{2 \cdot 2}) + c] + c && (9)\\
&= T(\frac{n}{2^i}) + ic && (10)\\
&= T(\frac{n}{2^{logn}}) + clogn && (11)\\
&= T(1) + clogn && (12)\\
&= O(logn) && (13)
\end{aligned}$$

- thus binary search has a complexity of $O(logn)$

Inversion Count Problem

Problem:

- count the number of inversions in n ordered numbers
- *alternatively*, find the number of line segment crossings given n line segments and their positions
  - ie. check the ordering of lines on the bottom vs. top
  - eg. 12345 vs. 21534 has 3 crossings or inversions

Solution:

- try a divide and conquer approach:
  - if we split the numbers in half, and know the number of crossings in each half, we only have to calculate crossings *between* the halves
  - if we can merge in linear time as well, can achieve $O(nlogn)$ complexity

1. divide list of elements in half recursively
   - this reduces the problem set to a trivial base case, ie. there are no inversions in a list with only one number
2. merge the solutions together, while *sorting* the lists

Modified merging algorithm to count crossings:

- to merge together two sorted lists L and R *and* return a count of their crossings:

1. initialize pointers to the start of each list, and initialize the total number of crossings to the sum of crossings in each half
2. *repeatedly*:
   - compare the pointers L_i and R_j
   - put the smaller element into the output list
   - whenever $R_j < L_i$, we know there are a number of crossings equal to S, the *remaining* elements in L after L_i
     - ie. *because* since L and R are sorted, we can add S to the count in one go
     - increment the number of crossings accordingly
   - increment the pointer of the smaller element

Analysis:

- the merging algorithm has the same complexity as the merge-sort merging algorithm, since each element is considered once
  - thus the overall time complexity of the inversion count algorithm is $O(nlogn)$

Closest Pair

Problem:

- find the *closest* pair among a set of coordinates:
  - an $O(n^2)$ solution is trivial, simply consider all $\binom{n}{2}$ pairs
  - is an $O(n)$ or $O(nlogn)$ solution achievable?
  - $O(n)$ - have to consider at least every coordinate, can't be sublinear
    - can't consider only one axis either
  - $O(nlogn)$ - can *sort* the points by an axis, or try divide and conquer with a linear merging algorithm

Solution:

- if we knew the closest pairs on the left and right halves of the problem set, only have to calculate the possible pairs between the halves:
  - however, the number of pairs between halves is still $\frac{n}{2} \times \frac{n}{2} \approx n^2$
  - somehow, if we only check a *constant* number of points against every coordinate on *one* side, we can achieve $O(nlogn)$ complexity
- assume that we have halves L with coordinates x_i and R with coordinates y_j, and that δ_L and δ_R are the closest distances in each half:
  - say we will check every x against a constant number of y's
  - note that there cannot be *too many* possible candidates among the y's to compare, because then the smallest distance would have to be one of the pairs in R as it becomes more densely populated
    - ie. too many close coordinates to consider would imply that some of them are closer together than δ_R
- to begin with, we can limit along the x-axis which elements we are checking:
  - let $\delta = \mathit{min}(\delta_L, \delta_R)$
  - we only have to check coordinates within δ on either side of the midpoint, since any other coordinates will be farther away the current minimum distance and can be disregarded
- then, we can additionally limit along the y-axis for each x we are comparing with y's:
  - can check coordinates a distance δ above and below x's y-coordinate
  - this creates 16 square boxes total, each with length $\frac{\delta}{2}$
- note that the number of points in each box is *at most* 1:
  - if there were two points, they would have a maximum possible distance from each other along the diagonal of the box of $\frac{\sqrt{2}}{2}\delta$
    - this distance is less than δ, which breaks the assumption since each box is entirely in either L or R
- thus there are *at most* 16 total y's within the specfied bounds to compare with each x to check for potentially closer pairs

Thus we can achieve an $O(nlogn)$ solution as follows:

1. sort all coordinates according to the x-axis and y-axis
2. recursively split the x-axis in half to divide up coordinates
3. merge the solutions together:

- to calculate the minimum distance between halves:
    - for all points in one half, compare each with at most 16 vertically-neighboring points in the other half using the points sorted by y-axis
- return the minimum of the minimum distance of pairs in the left half, right half, and between halves

Analysis:

- the merge operation is performed in linear time
    - thus the closest pair algorithm has a time complexity of $O(nlogn)$

Dynamic Programming

- the **dynamic programming** paradigm is a problem-solving approach where subproblems *cannot* be ignored:
  - unlike greedy algorithms that greedily disregard certain solutions
  - often required when greedy algorithms fail
  - has an *inductive* flair
  - similar to an *ordered* exhaustive search
- two implementations:
  - using **memoized** recursion
    - more expensive in regards to stack space
  - using iteration
    - typically preferred
- *pros*:
  - easy to prove (argue inductively, exhaustive search)
  - relatively easy to implement
- *cons*:
  - easily mistaken with other algorithm paradigms
  - a larger space complexity than other paradigms
    - but this space is necessary, cannot discard subproblems

Interval Scheduling Problem with Weights

Problem:

- various tasks or **intervals** $\{I_1, ..., I_n\}$ with a start and end time $I_i = (l_i, r_i)$ overlap and conflict in time
  - each has an associated weight W_i
- finding a subset of intervals that do not overlap in time, but *maximize* the total weight of all selected intervals
  - note that a greedy solution that picks the highest weights fails

Solution:

- let $\mathit{OPT}[t]$ represent the total weight of the optimal set of intervals that can be selected between time 0 and t

1. *repeatedly*, grow OPT using the following **transition**, ie. relationship between iterations:
   - $\mathit{OPT}[r] = \mathit{max}(\mathit{OPT}[l-1] + W_i, \mathit{OPT}[r-1])$
     - where W_i is the weight of current interval we are considering, sorted by endpoint, with start and end times l and r

- we can either *include* or *exclude* each interval i from our solution:
  - *OPT*$[l - 1] + W_i$ is the total weight possible from including the interval
  - *OPT*$[r - 1]$ is the total weight possible from excluding the interval
- the algorithm terminates when *OPT*$[R]$ is found, where R is the ending time of the latest interval
- note that the order intervals are considered is important so that the elements of OPT used to compute the current solution have already been finalized
  - in this case, that means considering the intervals sorted by endpoint

Analysis:

- a proof of correctness can be easily found by arguing inductively using the transition relationship
- note that the algorithm above only saves the maximum total weight, not the solution itself of intervals:
  - to output the solution, pointers need to be mmaintained to each previous interval in the construction of *OPT*
  - then, traverse through the pointers of *OPT* backwards to extract the solution set of intervals
- the algorithm has three stages:
  - sorting intervals by endpoint - $O(nlogn)$
  - plane sweep using transition - $O(n)$
    - simple `for` loop with transition
  - retrieve full solution using pointers - $O(n)$
- thus the algorithm has a time complexity of $O(nlogn)$

Knapsack Problem

Problem:

- given a **knapsack** of capacity S and a set of n items each with size s_i and value v_i with total weight $> S$
  - maximize the total value of items fit into the knapsack
- considerations:
  1. if all items are of the same size, trivial solution is to take items with the greatest v_i until knapsack is full
  2. if all items are not of the same size:
     - *fractional* knapsack: can take a fraction of an item for a fraction of the value
       - then, simply break all items down to a unit size $\frac{v_i}{s_i}$

        * reduced to the trivial first case
3. *integer* knapsack:
    - items have different sizes, and items cannot be split
    - note that a greedy solution that selects the greatest $\frac{v_i}{s_i}$ fails

Solution:

- need a **parameter** for DP to transition over, similar to the time axis in interval scheduling:
    - eg. the number of items considered in a subsolution or the capacity S
    - can use both in a 2D *OPT*
- let *OPT*$[i, j]$ represent the maximum value subsolution for i items considered and capacity j
- the order in which we transition over *OPT* should be consistent
    - eg. traverse through *OPT* in row-major order until *OPT*$[n, S]$, our desired solution, is found
        * could also use column-major order

1. *repeatedly*, grow *OPT* using the following transition:
    - $\mathit{OPT}[i, j] = \mathit{max}(\mathit{OPT}[i, j - s_i] + v_i, \mathit{OPT}[i - 1, j])$
        - where s_i and v_i are the size and value of the ith item
    - we can either *include* or *exclude* each item i from our solution:
        - *OPT*$[i, j - s_i] + v_i$ is the total value possible from including the item
            * s_i less space in the knapsack, but add v_i value
        - *OPT*$[i - 1, j]$ is the total value possible from excluding the item
            * same as not considering the current item at all
        - DP fills out the rest of the knapsack from a previous subsolution
    - the algorithm terminates when *OPT*$[n, S]$ is found

Analysis:

- the algorithm would use a nested for-loop to fill out *OPT*
- thus the algorithm has a time complexity of $O(nS)$:
    - this is *not* a polynomial time algortihm, since the capacity S is a value that is *independent* of the input size
    - input is just size $2n + 1$, the n items' size and value, and a numerical value for S
    - example of a **pseudo-polynomial** time algorithm

Longest Common Subsequence

Problem:

- a **subsequence** is a subset of a string in the same given order (but not necce-

sarily *contiguous*)
  - number of subsequences is 2^n
- find the longest *common* subsequence between two strings x and y
  - note that neither a greedy nor a divide and conquer solution will work

Solution:

- use $|x|$ and $|y|$ as parameters
- let *OPT*$[i, j]$ represent the longest subsequence for the first i characters of X and the first j characters of Y

1. *repeatedly*, grow *OPT* using the following transition:
   - if $X_i = Y_j$:
     - $\mathit{OPT}[i, j] = \mathit{OPT}[i-1, j-1] + 1$
   - otherwise, if $X_i \neq Y_j$:
     - $\mathit{OPT}[i, j] = \mathit{max}(\mathit{OPT}[i-1, j], \mathit{OPT}[i, j-1])$
   - if the last character of both substrings is the same, we can *extend* the length of the previous solution by one
   - otherwise, we can try to match with a smaller set of characters of X or Y
   - the algorithm terminates when $\mathit{OPT}[|X|, |Y|]$ is found

Analysis:

- the algorithm would use a nested for-loop to fill out *OPT*
- thus the algorithm has a time complexity of $O(|X||Y|)$
  - this a *polynomial* solution that avoids finding an *exponentional* number of subsequences
- although it seems that the storage space complexity is also $O(|X||Y|)$, we do not need to save the entire array
  - instead, only the last row and column is needed to continue the transition
  - thus the algorithm has a space complexity of $O(\mathit{max}(|X|, |Y|))$

Bellman-Ford Algorithm

Problem:

- although Dijkstra's algorithm exists for finding shortest paths in a graph, it fails with *negative* edges
  - with negative edges, a node's weight can never be finalized in Dijsktra's since there may be a negative edge that reduces its cost later in the algorithm

  - want a solution that works for negative edges, as long as there are no **negative cycles**
    * negative cycles would allow for a shortest path of $-\infty$

Solution:

- assume *inductively* that we know the shortest paths from s to any other node within distance L:
  - ie. L is the number of edges allowed to use from node s
  - need to find a transition in efficient time from L to $L+1$
  - note that the maximum value L can be is n, the number of nodes in the graph, since if there are $> n-1$ edges between s and another node and there are no cycles in the graph, a node is being repeated along the way
- let *OPT*$[x, L]$ be the length of the shortest path from the root s to node x, of maximum length L

1. *repeatedly*, grow *OPT* on L using the following transition:
   - for each of the neighbors of node x, $\{y_1, ..., y_d\}$:
     - *OPT*$[x, L] = min_{1 \leq j \leq d}($*OPT*$[y_j, L-1] + w_j)$
       * where w_j represents the weight of the edge (y_j, x)
   - any path to x must pass through one of its neighbors y_j, and thus build on a previous subsolution
   - the algorithm terminates when *OPT*$[x, L]$ has been filled out for every node x and distance L up to n
   - note that if *detection* of negative cycles is desired, one more iteration can be run (ie. essentially for length $L+1$), and if *OPT* can still be updated to a shorter path, a negative cycle exists

Analysis:

- the transition consists of several parameters:
  - iterating over all nodes x in the graph - $O(n)$
  - iterating over all possible neighbors y_j of a node - $O(n)$
    * alternatively, instead of accounting for each node and each of its neighbors, the algorithm considers each edge exactly once
  - iterating over distance L up to n - $O(n)$
- thus the algorithm has a time complexity of $O(n^3)$ or equivalently $O(nm)$
- note that the algorithm also has a space complexity of $O(n)$, since the transition only needs the last row of *OPT* corresponding to $L-1$

Computational Biology

Problem:

- an **RNA sequence** consists of the characters A, U, C, G
  - match together pairs
- given such a sequence S of n bases, *maximize* the number of matches of bases, with certain restrictions:
  1. no *sharp* turns, ie. matches must be at least d characters apart
  2. matching the correct pairs A $\leftrightarrow$ U and C $\leftrightarrow$ G
  3. each base can only be matched with one other base
  4. no *crossing* of matches
- very similar to interval scheduling, except that matches can be *nested* within one another

Solution:

- let $\mathit{OPT}[i, j]$ represent the optimal number of matches for the RNA sequence subinterval between i and j
  - need i as another parameter to track matches *starting* at indices that are not 0, which allows matches to be nested

1. *repeatedly*, grow *OPT* using the following transition:
   - $\mathit{OPT}[i, j] = \mathit{max}(\mathit{OPT}_{\mathit{include}}, \mathit{OPT}_{\mathit{exclude}})$
     - where $\mathit{OPT}_{\mathit{include}} = \mathit{max}(\mathit{OPT}[i, k-1] + \mathit{OPT}[k+1, j-1] + 1)$
       * $\forall k$ *s.t.* S_k is a base match with S_j *and* $j - k > d$
     - and where $\mathit{OPT}_{\mathit{exclude}} = \mathit{OPT}[i, j-1]$
   - when considering the jth base, we can either include this base or exclude it from our solution:
     - if included, we can match it with a previous matching base pair and *nest* subsolutions on either side of the match
     - otherwise, same solution as only considering up to the $j-1$th base
   - the algorithm terminates when $\mathit{OPT}[0, n]$ has been found

Analysis:

- the transition consists of several parameters:
  - iterating over all possible start positions for the subinterval - $O(n)$
  - iterating over all possible ending positions for the subinterval - $O(n)$
  - iterating over all possible matches at position k with a single base - $O(n)$
- thus the algorithm has a time complexity of $O(n^3)$

Network Flow

- **network flow** is a unique problem-solving approach that revolves around allowing an algorithm to *regret* or *backtrack* on the decisions it has made
  - since there is really just one main algorithm that solves problems of this type, the challenge is in *transforming* a valid problem so that its input is compatible with the algorithm

Problem:

- given a directed graph $G = (V, E)$ where each edge e is associated with some capacity $c(e) > 0$, and nodes source s and sink t:
  - find the **maximum flow** from s to t
  - where the overall flow of a graph is equal to the sum of the flow along each edge in the graph
    - the flow along an edge e *cannot* exceed $c(e)$
    - all *incoming* flows to a node musts equal the *outgoing* flows from that node
- other definitions and related theorems:
  - **saturated** nodes have been assigned flow equal to the maximum capacity of all incoming edges
  - an **augmenting path** is a path from s to t s.t. at least 1 unit of flow can be *pushed* from s to t
    - ie. the minimum remaining capacity along this path is at least 1
  - a **residual graph** G_f from a graph G and a flow f is defined as follows:
    - for each edge $e = (u, v)$ in G where $f(e) < c(e)$, include the edge $e' = (u, v)$ in G_f with capacity $c(e) - f(e)$
      - this edge represents a **forward edge** that can still be taken in G_f
    - for each edge $e = (u, v)$ in G where $f(e) > 0$, include the edge $e' = (v, u)$ in G_f with capacity $f(e)$
      - this edge represents a **backward edge** that can be taken to *backtrack* from the flow
  - *theorem*: the maximum flow on a graph is *equal* to the **minimum cut** of the graph:
    - a **cut** is a partition of G into two sets A and B s.t. $s \in A$ and $t \in B$
    - the capacity of a cut (A, B) is the sum of all capacities of *all* edges out of A
    - the minimum cut is thus the cut of minimum capacity
    - note that every flow in G is $\leq$ the capacity on *any* cut
      - otherwise, pushing more flow than the capacity of some edges allow!

Ford-Fulkerson Algorithm

Solution:

1. let $f_{total} = 0$
2. *repeatedly*, while there is an $s \to t$ augmenting path:
    - run DFS from s to find a flow path to t
        - note that the DFS cannot traverse through edges with capacity $= 0$
    - let f be the *minimum* capacity along that path, and add f to f_{total}
    - for each edge $e = (u, v)$ in the path:
        - *decrease* $c(u \to v)$ by f
        - *increase* $c(v \to u)$ by f
        - essentially converting the graph into its residual graph
3. return f_{total}

Analysis:

- each iteration consists of a DFS and then a linear operation on the path found
    - the flow is also *strictly* increased on each iteration, since every new path contributes a new flow out of s of at least 1 unit
    - the *upper bound* for the max flow is C, the sum of the capacity of all edges leaving s
    - therefore, the algorithm *must* terminate after at most C iterations
- thus the algorithm has a time complexity of $O((m + n)C)$

Proof:

- *claim*: the following three statements are *equivalent*:
    1. f is a max flow in network ie. graph N
    2. the residual network N_f has no augmenting paths
    3. $|f| = c(S, T)$ for some cut (S, T) of N
    - to prove Ford-Fulkerson, we just need to show that $(2) \to (1)$, but the proof is simpler using the three-part equivalence version
- $(1) \to (2)$:
    - by contradiction, assume that N_f *has* an augmenting path
    - if an augmenting path exists, the max flow can still be increased by at least 1 unit of flow
    - this is a contradiction
- $(2) \to (3)$:
    - remove all saturated edges from N_f
    - note that thus s and t will become disconnected, since otherwise it violates the condition that there are no more augmenting paths
    - this creates a cut if we consider all reachable nodes from s to be in set S and all the remaining nodes in set T

    - then $|f| = c(S, T)$, since flow must leave S through each of the cut edges e, and the flow over each e is equal to $c(e)$ because e was saturated to begin with
- $(3) \to (1)$:
    - since $|f| \leq$ the capacity on any cut and $|f| = c(S, T)$, this is the *maximum* flow that can be achieved
    - as a side effect, the cut created in $(2) \to (3)$ is exactly the min-cut on N
- thus $(2) \to (1)$, and the Ford-Fulkerson algorithm is correct

Modelling Network Flow

- to model a problem as a network flow problem:
    - ensure the problem has the same constraints as network flow
        - eg. inflow to a node = outflow from a node
    - if there are multiple sources and sinks, create a **supersource** and **supersink** connected to the sources and sinks where the capacity on each edge is ∞
    - can use **vertex splitting** to split a vertex so that the flow that can come out of a vertex is *limited*
    - the min-cut acts as the *bottleneck* to network flow problems
        - can try and optimize the bottleneck

Non-Polynomial Problems

- for a certain class of problems, a polynomial time solution is impossible
  - finding a polynomial time algorithm for any one of these problems would solve all of them in polynomial time
  - these are **non-polynomial (NP)** problems

NP-Complete

Problem:

- a **vertex cover** is a set of vertices that touch *every* edge
  - find the *minimum* vertex cover with the least number of vertices

Attempt #1:

1. *repeatedly*, take the vertex of maximum degree
   - delete all adjacent edges
   - continue until no vertices remain

- this algorithm is incorrect
  - consider a simple three-pronged tree with legs of length two

Attempt #2:

- solve the simpler case where the graph is a tree

1. *repeatedly*, delete all leaves
   - place parents in vertex cover
   - continue until no vertices remain

- this specific version of the solution is correct, and runs in $O(n)$

NP-completeness:

- the original problem is atually **NP-complete**
  - cannot be solved in polynomial time
  - NP-complete problems include 5 original non-polynomial problems:
    - historically the original problems that coudn't be solved in polynomial time
    - establishing equivalence between problems

1. vertex cover problem
2. **travelling salesman problem (TSP)**:
   - given a collection of cities with costs on the edges connecting cities

  - *minimize* the cost to travel to every city
  - can be solved in $O(n!)$
3. **3-satisfiability problem (3SAT)**:
   - given a *boolean* equation, can we assign 0 and 1's such that the equation $= 1$
     - eg. $F = (x_1 + \overline{x_2} + x_3) \cdot (x_1 + x_4 + \overline{x_5}) \cdot \ldots$
       * $+$ represents the or operation, $\cdot$ represents the and operation, and $\overline{x}$ represents the inverse operation
       * equation is a series of and's or or's of literals
   - for 3SAT, each clause has 3 literals
     - note that 2SAT does have a polynomial time solution
   - for n variables, there are 2^n possible assignments
4. **max clique (MC)**:
   - a **clique** is a set of pairwise connected vertices, ie. each vertex is connected to all other vertices in the clique
   - find the clique with the *maximum* number of vertices
5. **max independent set (MIS)**:
   - an **independent set** is a set of vertices where the vertices share no edges
   - find the independent set with the *maximum* number of vertices

NP-Hard

- **NP-hard** problems use a **polynomial transformation** to prove that they are NP
  - a lower-bound transformation to establish the difficulty of a problem
- $Y \leq_p X$ indicates that problem Y is polynomial transformable to X:
  - the input of Y can be transformed into a format taken by X
  - correspondingly, the output of X can be transformed into the output of Y
  - the transformation must be able to be completed in polynomial time
- suppose $Y \leq_p X$:
  1. if X can be solved in polynomial time, then Y can be solved in polynomial time
     - runtime of $Y \leq$ runtime of X
     - however, if Y can be solved in polynomial time, we cannot make any conclusions since X could have any time
  2. if Y can't be solved in polynomial time, then X can't be solved in polynomial time
     - by contradiction, assume that X could be solved in polynomial time
       * then, Y could be solved polynomially by simply polynomially transforming to X and solving X

- – however, if X can't be solved in polynomial time, we cannot make any conclusions since Y could still be solvable

Max Clique to Max Independent Set

- assume that MC is not solvable in polynomial time
  - prove that MIS is also not solvable in polynomial time

Proof:

- take the graph's *complement* to pass to MIS
  - add edges where there are none and remove any edges
  - input transformation is $O(m)$
- the number returned is the same MC for the original graph
  - output transformation is $O(1)$
- thus, *MC* $\leq_p$ *MIS*
  - since MC is not polynomial solvable, then neither is MIS

Vertex Cover to Set Cover

Set Cover Problem:

- given subsets from an overall set of elements
  - find a group of these subsets whose union gives the entire set, with the *minimum* number of subsets
- show that set cover is NP-hard by transforming it to vertex cover

Proof:

- for every vertex, create a subset of its incident edges
  - these subsets become the input for set cover, with the overall set being the set of all edges
  - input transformation is polynomial in $O(nm)$
    - at worst, have to consider all incident edges for every vertex
- the outputted sets from set cover correspond exactly to each vertex in vertex cover
  - output transformation is $O(1)$
- thus, *vertex cover* $\leq_p$ *set cover*
  - since vertex cover is not polynomial solvable, then neither is set cover

Satisfiabiliy to Max Clique

- assume that SAT is not solvable in polynomial time
  - prove that MC is also not solvable in polynomial time

Proof:

- construct a graph G for MC as follows:
  - each clause in SAT corresponds to a *column* in G
    - each literal is a vertex in the column
  - there exists an edge between any two columns only if the two vertices are *not* each other's complement
  - input transformation is $O(kn)$ where k is the number of literals in a clause and n is the number of clauses
- if there is a MC equal to the number of clauses, then the equation is satisfiable:
  - note that the vertices selected by MC represent the literals that should be chosen as true in the boolean equation
    - there are no edges *within* a column and x and $\overline{x}$ will never both be selected by the construction of the input graph
  - output transformation is $O(1)$
  - note that the variant of MC that we are solving is the whether or not $MC \geq m$, the number of clauses
    - same as solving MC
- thus, *SAT* $\leq_p$ *MC*
  - since SAT is not polynomial solvable, then neither is MC