CS 180 Comprehensive Review: Introduction to Algorithms and Complexity

Einar Balan

Contents

1 | Greedy Algorithms

1.1 Famous Problem

We have a group of people and we want to determine which among them are famous. Considering person A, there are two criteria for being famous:

1. Every person in the group knows A
2. A doesn't know anyone in the group

We want an efficient algorithm to determine who in the group is famous.

Algorithm

- Pick 2 arbitrary people, A and B
- Ask A if they know B
  - if yes, eliminate A as a famous candidate
  - if no, eliminate B as a famous candidate
- Repeat until there is a single candidate remaining
- Finally, ask this candidate if they know everyone and ask every other (eliminated) person if they know the candidate
  - if the candidate doesn't know anyone and everyone knows the candidate, then the candidate is famous
  - otherwise, there is no famous person in the group

Proof

- Assume towards a contradiction that the algorithm came to an incorrect conclusion
- Two cases
  1. Non famous candidate chosen
     - For a candidate to be chosen they had to pass the final round of vetting
     - We know the candidate was chosen by the algorithm, so they must not know anyone and every other candidate knows them
     - Via problem definition, this candidate must be famous
     - Contradiction

2. No famous person declared when there is a famous person
   - If there was no famous person found, this means that the final candidate did not pass the final round of vetting
   - By assumption, there must be another person P that was eliminated in the first pass
   - For this person to have been eliminated while the final candidate was kept, either the final candidate doesn't know P or P knew the final candidate
   - In either case, P cannot be famous, so we have reached a contradiction

- In both cases we have a contradiction

■

Complexity Analysis

- We ask $n-1$ questions for elimination round
- We ask $2(n-1)$ questions for confirmation round
- Therefore O(n)

1.2 Stable Matching Problem

We have a set of n men and n women to be married, each of which has a preference list consisting of the other gender. We want a perfect stable matching (i.e. a matching in which every man is paired with exactly one woman and there is no pair that likes each other more than their current partners). The algorithm we will describe is called the Gale-Shapley algorithm.

Algorithm

- Until there are no free men
  - Pick an arbitrary man, m
  - Propose to the first unproposedto woman on his preference list (i.e. the first woman he has not yet proposed to)
  - If the woman is single, the manand woman become engaged
  - If the woman is engaged toanother man, m'
    - If m is higher onthepriority list than m', m and the woman become engaged and m' is freed
    - Else no change

Proof

- Assume towards a contradiction that the matching produced is not perfect
  - This means that each man is not married to exactly one woman
  - There must be a point at which there is a free man but every woman has been proposed to (and is engaged)
  - This means that n men must be engaged (since n woman are engaged), but this is a contradiction because we just stated that we have a free man

  - So every man must be engaged to exactly 1 woman at the termination of the algorithm
- Assume towards a contradiction that the matching produced is not stable
  - This means that there must exist pairs (m, w) and (m', w') s.t. m prefers w' over w and w' prefers m over m'
  - m's last proposal must have have been to w, because they ended up together
  - If m did not propose to w' earlier $\rightarrow w > w'$, contradiction
  - If m did propose to w' earlier
    * There must be some man m'' which is ranked above m on w''s preference list
    * $m'' > m$
    * However, $m' > m''$ because m' ends up with w'
    * By transitive property, $m' > m$, which is a contradiction

■

Complexity Analysis

- A man can propose to at worst n women in his priority list
- There are n men
- n * n
- $O(n^2)$

1.3 Interval Scheduling

We have a set of tasks, each of which have some duration (an interval of time). Only one task can be done at a time (that is, no two tasks can overlap). How can we schedule the maximum number of nonoverlapping tasks?

Algorithm

- Sort set of tasks by earliest finish time
- Iterate through sorted list and schedule earliest
  - When a task is scheduled, remove all overlapping tasks
- Continue until maximally scheduled

Proof

- Assume there is an optimal solution that matches output in the first i intervals, but after this our algorithm "falls behind"
- If the (i + 1)th interval in our algorithm is longer than the optimal, there is a contradiction because the optimal will end first (and our solution would have picked it). Therefore up to this interval, the optimal and the algorithm we choose have the same result
- If our algorithm (i+1)th interval is shorter than the optimal, then replace the optimal solution with our interval.

- The number of intervals doesn't change
- Repeat this for the rest of the intervals
- After the the optimal solution has been completely transformed into the solution generated by our greedy algorithm, the number of intervals stays the same, therefore our solution is at least as good as this hypothetical optimal

■

Complexity Analysis

- O(nlogn) to sort by finish time
- O(n) to schedule

1.4 Moore's Voting Algorithm

Given a set of m candidates and a set of n voters, determine which candidate will win an election. A candidate can win an election if they have $> n/2$ votes. We want an O(n) solution.

Algorithm

- iterate through array of votes
  - set winner variable equal to first element in array and set count equal to 1
  - for each vote
    - if vote is for same as winner variable, increment count
    - if vote is not same, decrement count
    - if count reaches 0, set winner variable equal to candidate that caused decrement to 0 and increment count by 1
- when we reach the end of the array, the final winner variable is set to a potential majority winner
- do a second pass through the array to determine for sure that the potential winner truly does have a majority of votes

Proof

- Assume towards a contradiction that the algorithm does not determine a winner correctly
  1. Algorithm picks the wrong candidate as a winner (including the case where there is no majority)
     - this means that the final candidate left in the winner variable does not hold a majority
     - after the second pass, this candidate would not be picked b/c it does not have $n/2$ votes (assumed this was the wrong candidate)
     - contradiction bc the wrong candidate could not be chosen after the second pass
  2. Algorithm picks no winner (no majority) when there truly is a majority winner

- this means that the search for a majority candidate failed in the first pass (as the second pass correctly determined that the remaining candidate did not hold a majority, leading to no winner being picked)
- at the end, we can be certain that if there is a majority element, then since it appears more than n/2 times, there are not enough elements to cross out all of its occurrences, and thus it must be the element stored at variable m.

■

Complexity Analysis

- we look at each vote twice (first in the first pass, next in the second pass), non asymptotic term coefficient is dropped
- O(n)

2 | Graphs

Term	Definition
vertex	a node in the graph
edge	connects two nodes
connected graph	can reach any vertex in the graph starting at an arbitrary vertex
directed graph	can only travel in one direction across an edge; for instance for e(u,v) it is only possible to travel from u to v
cycle	a series of edges in which you start and end at the same node
tree	a connected graph with no cycles
DAG	a directed acyclic graph; the directed analog of a tree*

*Trees and DAG's differ in that in a DAG a node can have multiple parents and still not have a cycle (due to the directed edges) and in a tree a node can only have one parent (or it is guaranteed to have a cycle).

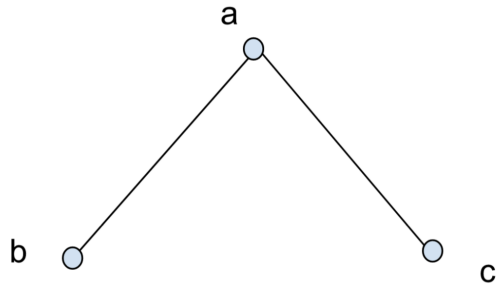

Figure 2.1: We will refer to this graph in the representations discussed.

2.1 Data Structure Representations

Adjacency Matrix

$$\begin{bmatrix} & a & b & c \\ a & 1 & 1 & 1 \\ b & 1 & 1 & 0 \\ c & 1 & 0 & 1 \end{bmatrix}$$

If nodes i and j are connected, then M[i][j] = 1 otherwise it is 0.

- O(1) to check if two vertices are neighbors.
- O(n) to traverse all neighbors (to find al ledges adjacent to a given vertex).
- Works better for dense graphs with many edges

Adjacency Lists

$$a \to b \to c$$

$$b \to a$$

$$c \to a$$

The head of each linked list points to all its neighbors.

- O(degree(n)) to check if a vertex is neighbors with n
- O(degree(n)) to traverse all neighbors
- Works better for sparse graphs

2.2 Breadth First Search

Explores all neighbors of a node and puts them into a "layer" before moving onto the next layer (neighbors of neighbors). Like "flooding" the graph. The layer number of a node marks the distance from the starting node. If some node does not appear in any layers, the graph is not connected.

BFS also generates a "BFS tree." That is, a representation of which edges were traveled along to discover all nodes. An edge is only added to the BFS tree if the node it connects has not been visited before. Non-tree edges all either connect nodes in the same layer or in adjacent layers.

Algorithm

- start at a chosen node and mark it as visited
- add each neighboring node to a queue
- until the queue is empty
  - pick the first node in the queue
  - add its neighbors to the queue
  - mark the node as visited
  - remove node from the queue

Proof

- if distance is actually longer than what was found
  - this means that a node at a higher level allowed the destination node to be visited for the first time
  - this is a contradiction because we found a lower layer that allows the node to be visited for the first time
- if distance is actually shorter than what was found
  - this means there is a path length j $<$ i (the distance we found)
  - go to the first point where the paths branch away from each other
    - for each path, going to a new layer in the BFS tree is distance 1
    - the path j will reach the destination in the same number of edges as path i
    - contradiction
- more informally: BFS restricts the path traveled to be only along previously unvisited nodes. The shortest path between two nodes cannot revisit any so it is the shortest path.

■

Complexity Analysis

- assume adjacency list representation
- outer loop runs v times, for each vertex in the graph
  - there are n neighbors of each vertex, each of which is added to the queue
  - for each vertex, we mark the node as visited and remove it from the queue
- v * (O(n) + O(1))
- O(vn + v), but vn is just the number of edges in the graph
- O(v + e)

2.3 Depth First Search

Completely exhausts a path before moving on to new paths in the graph. Generates a DFS tree. Every edge not in the tree that is in the graph is an ancestor relationship (they are connected in the tree via a different path).

Algorithm

- start at a chosen node and mark it as visited
- add each neighboring node to a stack
- until the stack is empty
  - pick the first node in the stack
  - add its neighbors to the stack
  - mark the node as visited

- pop node from the stack

Proof

- exercise to the reader

■

Complexity Analysis

- see BFS

2.4 Graph Coloring

A bipartite graph is a graph in which the nodes can be split up into two groups where every node has an edge between the two groups. We say a graph is bipartite if it is possible to color its node red and blue such that every edge is between only a red and blue node.

We know if a graph contains an odd cycle, it is not bipartite.

Algorithm

- run BFS
- color even layers blue and odd layers red
- if the BFS tree is not the same as graph (i.e. a cycle was present), the graph is not bipartite
- otherwise, since there are no edges between nodes of the same color, the graph is bipartite
- OR
- scan adjacency lists to ensure no neighbors are of same color

Proof

- if there is no edge connecting nodes of the same layer
  - there is no cycle and the BFS tree will be the same as the graph
  - then every edge will be between adjacent layers which are colored two different colors (representing the 2 different groups)
  - so for every node in a group there are edges only to the other group
- if there is an edge connecting nodes of the same layer
  - consider a node n that is an ancestor of both nodes connected in the same layer
  - these 3 nodes form an odd cycle
  - an odd cycle shows the graph is not bipartite because there will always be an edge between two nodes of the same color (pigeonhole principle)

■

Complexity Analysis

- BFS is O(v + e)
- comparing the number of edges in the tree vs the graph is O(1)

2.5 Topological Sort

For a directed acyclic graph (DAG), we want to output a list such that the ordering in the list does not violate any precedence relationships established in the DAG. That is, if we have an edge from v to u, v comes before u in the topologically sorted list.

We assume that we know the in degree and out degree of each node (specifically, our graph representation is two lists for each node: one with nodes to which the node has an edge and one with nodes from which the node has an edge; the sizes of these two lists are the out and in degree of the node).

An important note is that there is no solution if our graph is not a DAG.

Algorithm

- find all source vertices in the graph
- pick an arbitrary source vertex
  - add it to a list L and remove it from the graph
  - repeat until there are no more vertices in the graph
- the list L is topologically sorted

Proof

- assume G is a DAG
- assume towards a contradiction that the list output by the graph violates a precedence relationship between two nodes u and v (v is directed towards u in the graph but comes after it in the list)
  - this means that at some point u must have been a source node before v became a source node and was output into the list
  - this is a contradiction because there is no way for u to not have any nodes pointing towards it (i.e. source node) while v is still in the graph
  - because of the way the algorithm is structured v will always be output before u
- thus the algorithm produces a valid topological sort

■

Complexity Analysis

- to find our sources is O(v), where v is the number of nodes
  - each vertex must be a source node that is output once throughout the algorithm
- O(e) to create new source nodes
  - we delete each edge when it becomes a source

- O(v + e)

2.6 Shortest Path Problem

Given a weighted graph, we want to find the shortest path between two vertices.

A side effect of running the algorithm is that we will have the shortest path to every node from the starting node as well as the path to get there. We will discuss Dijkstra's Algorithm.

Algorithm

- label distance from starting node to every other node in min heap
  - self distance is 0
  - every other is ∞
- consider each neighboring node of starting node that is unvisited
  - if the weight + the distance leading to that node is $<$ the current labeled distance then relabel the distance with this value
  - mark node as visited
  - travel to the next closest node from starting node and repeat analysis
- finish when every node has been visited
- return the distance between the two nodes

Proof

- exercise to the reader

■

Complexity Analysis

- for each vertex, we examine all n connected edges
- insertion from min heap (and deletion is) O(logv)
- so O(vnlogv) = O(elogv)

3 | Minimum Spanning Trees

3.1 Prim's Algorithm

Given a weighted graph, we want to find a minimum spanning tree. That is, we want to find a subset of all the edges of the graph such that the graph is a tree (connected) and the sum of all the included edges is the minimum possible.

Prim's algorithm is a vertex centric approach.

MST Theorem: Given an arbitrary bipartition of a graph s.t. there is at least one node in each partition, the minimum cut edge between the partitions belongs in every MST.

Algorithm

- select an arbitrary root node S
- until there are no vertices left that have not been added
  - consider all vertices connected to the tree that have not yet been added
  - add the vertex that is connected by the lowest weight edge
- return the MST

Proof

- by MST theorem, for each step of the algorithm we add an edge that is contained in the MST
- the final result must be a spanning tree because it contains all vertices and no cycles (since the only edges added must be part of the MST, therefore no added edges can create a cycle)

■

Complexity Analysis

- for each edge, insert/delete from min heap which takes logv time
- O(elogv)

3.2 Kruskall's Algorithm

An edge centric approach to finding the MST of a graph.

Algorithm

- sort edges in non decreasing order
- until all vertices are included
  - consider e, the minimum edge that has not yet been added
  - if e will not create a cycle in the current tree, add e to the tree
- return the MST

Proof

- consider an edge e(v, w) about to be added by Kruskall's algorithm
- let V be the set of all nodes and S be the set of all nodes to which a node v has a path to during algorithm execution before e is added
- e is the cheapest edge between the partitions S and V - S, thus by MST theorem Kruskall's algorithm produces an MST

■

Complexity Analysis

- O(elogv) from Union Find operations
- O(eloge) for sorting

3.3 Union Find

A data structure that is used to implement Kruskall's algorithm efficiently (in particular the cycle checking).

In general in Kruskall's Algorithm, if two nodes are already in the same connected component, then adding another edge between them will create a cycle. Union Find allows us to efficiently track these connected components and determine if two nodes are in the same one.

Operations

- Find(u) - returns the set containing u in O(logn) time
- Union(a, b) - merge sets a and b O(1)
- MakeUnionFind(s) - creates a new Union Find using a set s in which all the items in s are disjoint in O(|s|) time

Description

- each node v is contained in a struct that contains a pointer to the set it belongs to
- Union involves taking node for which a set was named and pointing it towards the name node for another set

Complexity Analysis

- for each edge, we execute 2 Find(u) to see if new edge will create a cycle (if it won't we won't execute a union operation)
- therefore total O(elogv)

3.4 Clustering

Given a set of points where distance between points represents similarity, generate k clusters of similar points. Essentially, we want to create k groups of the most similar points.

Algorithm

- assume we have a graph G, where each point is represented by a node and nodes are connected by an edge of weight d_{ij}
- use Kruskall's algorithm to construct a minimum spanning tree, but halt the algorithm when there are k connected components remaining
- return the clusters

Proof

- an optimal clustering maximizes min dij, or the distance between 2 clusters i and j
- we have clusters c_1, c_2, ..., c_k produced by Krukall's algorithm
- by contradiction, assume we have another set of clusters c'_1, c'_2, ..., c'_k produced by a better solution
- there must be at least 1 cluster that is not a subset between of any 2 clusterings in c' clustering
- therefore, there must be 2 vertices in the same c' cluster that are in different c clusters w/ distance dx
- dx must be smaller than or equal to our min d_{ij} (since kruskall added that edge before d_{ij} edge) therefore we either have the same solution or the prime solution doesn't maximize d_{ij}
- contradiction because our solution produces a larger min d_{ij}

■

Complexity Analysis

- assume graph given we use Kruskall's algorithm and Union Find
- data structure so O(eloge)

4 | Divide and Conquer

Divide and conquer involves dividing the problem into subproblems and merging the solutions. Recurrence relations can be useful in order to determine the complexity of the algorithms.

Some common recurrences

- T(n) = qT(n/2) + cn
  - for n > 2 each step of the recursion is split into q subproblems of size n/2
  - division and merging are both O(n)
  - q = 2 → O(nlogn)
  - q > 2 → O($n^{\log q}$)
  - q = 1 → O(n)
- T(n) = 2T(n/2) + cn2
  - for n > 2 each step of the recursion is split into 2 subproblems of size n/2
  - division and merging are both O(n^2)
  - O(n^2)

4.1 Mergesort

Algorithm

- split the list into two approximately equally sized lists
- via recursion, sort these two lists (if the list size is two, simply compare the two elements and order appropriately)
- merge the result into one large sorted list
  - merge performed by maintaining two pointers starting at the first element in each list
  - we add the smaller of the two elements to the merged list and advance its pointer
  - continue until all elements have been added
- return sorted list

Proof

- by contradiction, assume mergesort does not produce a properly sorted list
- this means that there is some a[i] $>$ a[j] for i $<$ j
- consider the base case of a list of size 2
- mergesort will never allow this list to not be properly sorted since it is based on a simple comparison (putting the larger element 2nd)
- now consider we have 2 lists of size two (originally split from a larger list), each of which is sorted by mergesort
- during the merge operation, we always add the smaller elements first (meaning the resulting merged list will have smaller elements before larger)
- this base case can be extended to lists of arbitrary size
- we have reached a contradiction since it is impossible for a[i] $>$ a[j] for i $<$ j based on these facts

■

Complexity Analysis

- T(n) = 2T(n/2) + cn
- this resolves to O(nlogn)

4.2 Inverted Pairs Problem

Given a list of n integers, we want to find how many inversions are in the list. An inversion can be defined as the case where i $<$ j and ai $>$ aj. Essentially, it means that a larger number comes before a smaller number. We want to count these instances efficiently.

Algorithm

- divide the list into two lists, A and B
- sort each list and count the inversions within them recursively
- maintain a pointer to the beginning of each sorted list
- until one of the lists is empty
  - compare the elements referenced by the pointers
  - if $B_j < A_i$
    - add |A| - i to the number of inversions (number elements remaining after element i)
    - add B_j to sorted list
    - advance j pointer
  - else
    - add A_j to the sorted list
    - advance i pointer

- append the remaining elements from the non empty list to the sorted list
- return the number of inversions counted (in A and B and between A and B)

Proof

- similar to mergesort argument

■

Complexity Analysis

- T(n) = 2T(n/2) + cn
- we split the problem into two subproblems half the size of the original in O(n) time and then merge the results in O(n) time
- this recursion resolves to O(nlogn)

4.3 Closest Points Problem

Given n points in a plane, we want to efficiently find the pair of points that is closest together.

Algorithm

- sort points by non decreasing x and y coordinate
- via recursion, we know the closest points on left side of plane and right side of plane (call the min of these two d_R and store this pair as the min distance pair)
- let L be the line dividing the left and right sides of the plane
- let S_y be the set of all points within distance d_R of L
- for all points in S_y
  - compute the distance between this point and the next 15 points in S_y
  - if this distance is $< d_R$ set this distance as the new d_R and store points somewhere
- return the min distance pair

Proof

- exercise to the reader

■

Complexity Analysis

- T(n) = 2T(n/2) + cn
- splitting the problem into 2 half the size and then mergin in O(n) time (15 operations to merge)
- O(nlogn)

5 | Dynamic Programming

5.1 Weighted Interval Scheduling

We are given a set of n intervals, each with an associated starting time s(i), end time f(i), and weight w(i). We want to determine the maximum compatible set, that is the set of intervals that don't overlap and have the maximum possible weight.

Define p(j) to be the last interval that does not overlap interval j (0 if there are none). Define OPT(j) to be the size of the optimal set of intervals up to interval j.

Algorithm

- sort intervals by non decreasing f(i)
- OPT(0) = 0
- for i to n
  - OPT(i) = max(OPT(p(i)) + w(i), OPT(i - 1))

Proof

- by induction
- we know base case OPT(0) = 0
- assume that out algorithm is correct for all i $<$ j
- let our algorithm be ALG
- we know that ALG(p(i)) = OPT(p(i)) and
- ALG(i - 1) = OPT(i - 1) via induction hypothesis
- so OPT(i) = max(ALG(p(i)) + w(i), ALG(i - 1)) = ALG(i)

■

Complexity Analysis

- O(nlogn) to sort
- for each interval, we perform a constant time operation → O(n)

5.2 Knapsack Problem

Given a knapsack of limited size S and n items each with value vi and size si, we ant to fit the maximum possible value within our knapsack.

Define OPT(i, j) to be the optimal solution's value for items 1 - i and knapsack size j.

Algorithm

- for i to n
  - for j to S
    - OPT(i, j) = max(OPT(i - 1, j - s_i) + v_i, OPT(i - 1, j))

Proof

- exercise to the reader

■

Complexity Analysis

- we do a constant time operation n * S times
- O(nS)

5.3 Largest Common Subsequence

Given 2 sequences from an alphabet of size n and m, we want to determine the largest common subsequence (with holes allowed).

Define OPT(i, j) = i letters from first sequence and j letters from second.

Algorithm

- OPT(0, j), OPT(i, 0) = 0
- for i to n
  - for j to m
    - if L[i] == R[j]
      - OPT(i, j) = OPT(i - 1, j - 1) + 1
    - else
      - OPT(i, j) = max(OPT(i - 1, j), OPT(i, j - 1))

Proof

- exercise to the reader

■

Complexity Analysis

- O(nm)

5.4 Shortest Path with Negative Weight

We want to find the shortest paths in a graph to all nodes from a starting node, given that some weights are negative. We describe the Bellman-Ford Algorithm.

Define OPT(s, v) as the shortest distance starting from node s to node v. ci is the weight of an edge.

Algorithm

- for i to n -1
  - for v_i in V
    - OPT(s, v_i) = min(OPT(s, w_i) + c_i), for all neighbors w_i of v_i

Proof

- exercise to the reader

■

Complexity Analysis

- for up to v - 1 iterations, for v vertices, for up to n neighbors of a vertex, we perform a constant time operation
- O((v - 1) * v * n) = O((v - 1) * e) = O(ve)
- O(ve)

5.5 Curve Fitting

Given an error function and a set of points, we want to determine an optimal set of lines that minimize the error function.

Define OPT(i) to be the optimal set of lines up to point i and ee,j to be the min error of any line from point i to point j. n is the number of points. C is an arbitrary cost for adding a line.

Algorithm

- assume we have the error for all (i, j) pairs
- for j to n
  - for all i <= j, OPT(j) = min($e_{i,j}$ + C + OPT(i - 1))

Proof

- exercise to the reader

■

Complexity Analysis

- n * worst case n
- O(n^2)

6 | Networks

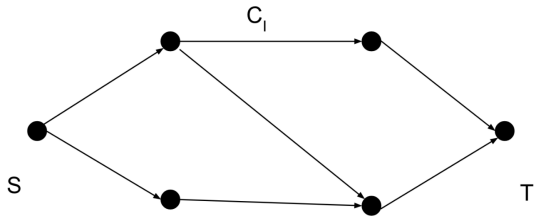

Figure 6.1: A network from source S to sink T that has an edge of capacity C_i

A **network** is a weighted directed graph with a source node and sink node. Each edge weight corresponds to a **capacity**, or a maximum allowable flow of "stuff" along that edge.

All edges point away from the source and towards the sink.

Flow refers to "stuff" being sent along the network. In order to send f_i goods along an edge, $f_i \leq c_i$ for all i. Flow must also be conserved at all points along the network. That is, the sum of flow into a node is equal to the sum of flow out of a node (except S and T). The flow on an edge must be $\leq$ the capacity of the edge and non-negative.

We are interested in the max flow, of which there may be multiple solutions.

A **bottleneck** is an edge that limits the max flow of a network. This is also referred to as the min cut.

A **residual graph** for a flow f_i contains the same node set but modifies forward edges with capacity c_i - f_i and adds backwards edges with capacity f_i. An **augmenting path** is a path along forward or backwards edges that has leftover capacity (more flow can be added to it).

6.1 Max Flow Problem

Given a network, we want to find the max possible flow along that network. We describe the Ford-Fulkerson Algorithm.

Algorithm

- initially the flow is 0 on all edges in G
- f is the overall flow

- while there is an s-t path, P, in the residual graph G_f
  - set f to be the resulting flow value after augmenting P
  - augmenting is when we take every edge and P and add bottleneck flow to it if it is a forward path and subtract bottleneck flow if it is backwards
  - set G_f to the resulting graph after augmentation
- return f

Proof

- WTS: if 1 then 2, if 2 then 2, if 3 then 1. If true, then our algorithm produces an optimal solution.
  1. Flow f is a max flow
  2. There is no augmenting path in the residual graph G_f
  3. |f| = capacity of some (S, T) cut of network
- if 1 then 2:
  - by contradiction, assume there is an augmenting path
  - take the flow f along this path and add 1
  - we have reached a contradiction, because f is not a max flow as stated in 1
- if 2 then 3:
  - an edge is saturated if $f_i = c_i$
  - remove saturated edges
  - by 2, we know that there is no augmenting path
  - treat the 2 disconnected components as 2 partitions
  - any cut edge therefore must be saturated
  - therefore |f| = capacity of (S, T) cut since $f_i = c_i$ (definition of edge saturation)
- if 3 then 4:
  - given cut with capacity C(ST)
  - |f| <= C(ST) for all ST cuts
  - no flow can be greater than a cut capacity
  - therefore if |f| = C(ST) for any ST cut, we know it must be a max flow

■

Complexity Analysis

- O((v + e) * |f|)
- O(v + e) operation (DFS to find s-t path in residual graph), f times

6.2 Applications

Ford-Fulkerson can be applied in a variety of situations. Typically if we want to match one group to another group without violating any capacity requirements, Ford-Fulkerson is a good choice.

Finding min cut in a network

- run Ford-Fulkerson and return resulting residual graph
- let A be the set of all nodes reachable from S and B be the remaining nodes
- the cut (A, B) is the min cut
- this can be determined in O(f *(v + e)) time

Bipartite Matching Problem

- generate a network based on problem statement (add source and sink)
- run Ford-Fulkerson
- every edge with 1 flow represents a match
- source edge capacity represents how many matches for each object on the left
- sink edge capacity represents how many matches for each object on the right

7 | NP Completeness

We know of many computationally difficult problems. For example:

1. Traveling Salesman - given a set of points, minimize the total distance traveled while traveled to all points
2. Satisfiability - given a product of sums, is there an assignment of variables to 0 and 1 s.t. in each clause at least 1 variable is 1 (resulting in an overall true evaluation)
3. Finding Hamiltonian Paths - given a directed graph, we find a path that starts from a source and passes through every vertex in the graph

Many of these problems can be made far easier if certain restrictions are applied to them. For example, the finding Hamiltonian Paths in a DAG can be done in O(v + e) time through an application of topological sort.

7.1 Polynomial Time Reductions

We can use problem transformation to prove NP completeness of a problem, given another problem that we know is NP complete.

We say that Y $\leq_p$ X if Y is polynomial time transformable to X, that is it takes polynomial time to transform the input of Y so that it is able to be fed into X and likewise for output. This can also be read as "X is at least as hard as Y."

If Y cannot be solved in polynomial time (it is NP complete), then X must also be NP complete, since it is at least as hard as Y.

Proving NP completeness

- we know Y is NP complete we want to show X is NP complete
- if Y $\leq_p$ X, then X must be NP complete
- so if we can show that the problem Y can be transformed in polynomial time to become X, then X is NP complete

7.2 Lower Bound Arguments (Sorting)

Given n numbers, we will have n! leaves in our decision tree.

The height of a tree is log(q) where q is the number of leaves. So we have log(n!) height. approximates to nlogn.