CSE 421 Notes

Contents

1 Graphs

1.1 Graph Terminology

- Degree: the number of edges connected to the node
- Connected: there exists a path from any node to any other node in the graph
- Walk: a sequence of adjacent vertices
- Length: the number of edges in a walk
- Path: a walk that does not repeat a vertex
- Matching: a subset of edges of a graph such that each vertex appears in at most one edge

1.2 Storing Graphs

- Adjacency matrix
  - `A[i][j] = 1` if and only if there exists an edge (v_i, v_j)
  - $O(n^2)$ space complexity
- Adjacency list
  - `A[i]` represents the list of edges of vertex v_i
  - $O(n + m)$ space complexity
- Adjacency array
  - `A[i]` represents the index of vertex v_i in the array `B`
  - `B[k]`, where `k` $\in$ [`A[i]`,`A[i+1]`) represents the edges of the vertex v_i
  - $O(n + m)$ space complexity
- Implicit representation
  - Uses objects to store relationships between vertices

1.3 Undirected Graphs

An undirected graph is a graph where the edges do not have direction

- Every undirected graph with n vertices has at least $\frac{1}{2}n(n-1)$ edges

1.4 Tree Graphs

A tree is a graph that contains no cycles

- If a graph has no cycle, then there exists a vertex of degree at most 1

1.5 Bipartite Graphs

An undirected graph is bipartite if the vertex set can be partitioned into two disjoint sets such that no two graph vertices within the same set are adjacent

- If a graph is bipartite, then it can be 2-colored
- If a graph is bipartite, then it does not contain an odd cycle

1.6 Directed Acyclic Graphs

A directed acyclic graph is a directed graph with no directed cycles

- If a graph is a directed acyclic graph, then there exists a vertex with in-degree 0

1.7 Topological Sort

A topological sort of a directed graph is a linear ordering of its vertices such that all edges go from left to right

- A graph has a topological ordering if and only if the graph is a directed acyclic graph

1.8 Spanning Trees

A tree is a spanning tree of a graph if it spans all vertices in the graph and consists of only edges within the graph

- If $T = (V', E')$ is a spanning tree of $G = (V, E)$, then $V' = V$, $|E'| = |V'| - 1$, and $E' \in E$

1.9 Minimum Spanning Trees

A minimum spanning tree is the lowest-cost spanning tree of a graph

- A graph may have multiple minimum spanning trees

1.10 Cuts

A cut is any partition of the vertices in a graph into two disjoint sets of vertices, denoted (A, B)

- The vertices in each set are not necessarily connected to each other

1.11 Cut Property

The lightest edge connecting any two disjoint sets of vertices must be in every minimum spanning tree

- If there are multiple edges tied for the lowest weight, then every minimum spanning tree must contain at least one of them

1.12 Cycle Property

The heaviest edge in every cycle cannot be in any minimum spanning tree

- If there are multiple edges tied for the highest weight, then every minimum spanning tree can contain at most all but one of them

1.13 Stable Matching Problem Terminology

- A match (A, B) is unstable if there exists C such that A prefers C over B, and C prefers A
- A match (A, B) is stable if it is not unstable
- A matching is perfect if each man gets exactly one woman and each woman gets exactly one man
- A matching is stable if it is perfect with no unstable matches

1.14 Gale-Shapely Algorithm

```
function galeShapely(preferenceList) {
    <set each person to be unengaged>

    while (exists unengaged man) {
        <choose unengaged man m>
        <let w be the next woman in the preference list to whom m has not yet proposed>

        if (w is unengaged) {
            <assign m and w to be engaged>
        } else if (w prefers m to her current fiance f) {
            <assign m and w to be engaged>
            <set f to be unengaged>
        } else {
            <w rejects m>
        }
    }
}
```

- Gale-Shapely algorithm has $O(n^2)$ running time

1.15 Breadth First Search Algorithm

```
function breadthFirstSearch(graph, source) {
    toVisit.add(source);
    source.visited = true;
    source.level = 0;

    while (toVisit.size() > 0) {
        curr = toVisit.remove();
        for (vertex : curr.neighbors) {
            if (!vertex.visited) {
                toVisit.add(vertex);
                vertex.visited = true;
                vertex.level = curr.level + 1;
            }
        }
        finished.add(curr);
    }
}
```

- In breadth first search, `toVisit` is a FIFO queue
- Breadth first search algorithm has $O(|V| + |E|)$ running time

Applications of breadth first search algorithm

- Shortest path of unweighted graph
  - `vertex.level` represents the length of the shortest path from the source to the vertex
- Check if a graph is bipartite
  - If a graph is bipartite, then there is no edge connecting two vertices on the same level

1.16 Depth First Search Algorithm

```
function depthFirstSearch(graph, source) {
    toVisit.add(source);
    source.visited = true;

    while (toVisit.size() > 0) {
        curr = toVisit.remove();
        for (vertex : curr.neighbors) {
            if (!vertex.visited) {
                toVisit.add(vertex);
                vertex.visited = true;
            }
        }
        finished.add(curr);
    }
}
```

- In depth first search, `toVisit` is a LIFO stack
- Depth first search algorithm has $O(|V| + |E|)$ running time

Applications of depth first search algorithm

- Topological sort
  - Running depth first search on a graph generates a tree consisting of the edges taken by the algorithm
  - An edge (u, v) in the tree indicates that vertex u appears before vertex v

1.17 Dijkstra's Shortest Path Algorithm

```
function dijkstraShortestPath(graph, source) {
    for (vertex : graph.vertices) {
        vertex.dist = infinity;
    }
    source.dist = 0;
    toVisit.add(source, 0);

    while (toVisit.size() > 0) {
        curr = toVisit.removeMin();
        for ((curr, neighbor) : curr.outEdges) {
            if (curr.dist + weight(curr, neighbor) < neighbor.dist) {
                if (neighbor.dist == infinity) {
                    toVisit.insert(neighbor, curr.dist + weight(curr, neighbor));
                } else {
                    toVisit.decrKey(neighbor, curr.dist + weight(curr, neighbor));
                }
                neighbor.dist = curr.dist + weight(curr, neighbor);
                neighbor.predecessor = curr;
            }
        }
        finished.add(curr);
    }
}
```

- In Dijkstra's algorithm, `toVisit` is a minimum priority queue implemented using a Fibonacci heap
  - A Fibonacci heap has $O(1)$ `insert()` / `decrKey()` / `findMin()` time complexity and $O(\log n)$ `removeMin()` time complexity
- Dijkstra's algorithm has $O(|E| + |V| \log |V|)$ running time

1.18 Kruskal's Minimum Spanning Tree Algorithm

```
function kruskalMST(graph) {
    <initialize each vertex as a connected component>
    <sort edges by weight>

    for (edge (u, v) in sorted order) {
        if (u and v are in different components) {
            <add (u, v) to MST>
            <update u and v to be in the same component>
        }
    }
}
```

- At each iteration, Kruskal's algorithm adds the smallest edge that expands the currently reachable set
- Kruskal's algorithm has $O(|E| \log |V|)$ running time

1.19 Prim's Minimum Spanning Tree Algorithm

```
function primMST(graph) {
    for (vertex : graph.vertices) {
        vertex.dist = infinity;
    }
    source.dist = 0;
    source.bestEdge = (source, source);
    toVisit.add(source, 0);

    for ((source, neighbor) : source.outEdges) {
        neighbor.dist = weight(source, neighbor);
    }
    while (toVisit.size() > 0) {
        curr = toVisit.removeMin();
        spanning_tree.add(u.bestEdge);
        for ((curr, neighbor) : curr.outEdges) {
            if (weight(curr, neighbor) < neighbor.dist) {
                neighbor.dist = weight(curr, neighbor);
                neighbor.bestEdge = (curr, neighbor)
            }
        }
        finished.add(curr);
    }
}
```

- At each iteration, Prim's algorithm adds the closest vertex to the currently reachable set
- In Prim's algorithm, `toVisit` is a minimum priority queue
- Prim's algorithm has $O(|E| + |V| \log |V|)$ running time

2 Greedy Methods

2.1 Greedy Methods

Greedy methods build a solution part by part, always making the locally optimal choice at each stage. This approach never reconsiders the choices taken previously

2.2 Interval Scheduling Problem

Consider a set of tasks where each task i starts at s_i and ends at f_i. Two tasks are said to be compatible if they do not overlap. Find the maximum subset of compatible tasks

- Task u is compatible with task v if $s_u \geq f_v$ or $f_u \leq s_v$

2.3 Interval Scheduling Algorithm

```
function intervalScheduling(tasks) {
    <sort tasks in increasing order of finish times>
    scheduledTasks = [];

    for (task : tasks) {
        if (task is compatible with all tasks in scheduledTasks) {
            scheduledTasks.add(task);
        }
    }
}
```

- Interval scheduling algorithm has $O(n \log n)$ running time

2.4 Interval Partitioning Problem

Consider a set of tasks where each task i starts at s_i and ends at f_i. Two tasks are said to be compatible if they do not overlap, and each partition represents a subset of compatible tasks. Find the minimum number of partitions needed to encompass all tasks

2.5 Interval Partitioning Algorithm

```
function intervalPartitioning(tasks) {
    <sort tasks in increasing order of start times>
    partitions = <min priority queue of scheduledTasks based on finishing time>

    for (task : tasks) {
        if (exists scheduledTasks in people where task is compatible) {
            scheduledTasks.add(task);
        } else {
            <initialize new scheduledTasks>
            scheduledTasks.add(task);
            partitions.add(scheduledTasks);
        }
    }
}
```

- The depth of a set of time intervals is the maximum number of time intervals that contains any given time
  - The minimum number of partitions needed to complete all tasks is greater than or equal to the depth of the problem
- Interval partitioning algorithm has $O(n \log n)$ running time

2.6 Minimizing Lateness Problem

Consider a set of tasks where each task i takes t_i time to complete and has deadline d_i. Tasks may be scheduled into time intervals $[s_i, f_i]$ where $t_i = f_i - s_i$. The lateness of a task is defined as $L_i = \max(f_i - d_i, 0)$. Find a schedule that minimizes the maximum lateness $L = \max\limits_{i \in \Omega}(L_i)$

2.7 Minimizing Lateness Algorithm

```
function minimizingLateness(tasks) {
    <sort tasks in increasing order of deadlines>
    time = 0;

    for (task : tasks) {
        task.start = time;
        task.finish = task.start + task.time;
        time = task.finish;
    }
}
```

- Minimizing lateness algorithm has $O(n \log n)$ running time

2.8 Minimizing Lateness Exchange Argument

A pair of tasks (i, j) is valid if

- Task i is scheduled immediately after task j
- Deadline d_i is less than or equal to deadline d_j

The exchange argument states that swapping a pair of valid tasks does not increase the maximum lateness

- If task i is scheduled after task j and $d_i \leq d_j$, then task i is scheduled too late while task j is scheduled too early
- Swapping tasks i and j will not increase $\max(L_i, L_j)$ such that the maximum lateness does not increase

*A comprehensive proof on the exchange argument can be found in **CSE 421 HW 3**, page 4*

2.9 Optimal Caching Problem

Consider the following setup

- Main memory can store n data items
- Cache can store k data items, where $k < n$
- Initial cache is full
- A cache hit occurs when a requested item is already in the cache
- A cache miss occurs when a requested item is not in the cache
  - The requested item must be loaded into the cache
  - If the cache is full, some existing items are evicted

Find an eviction schedule that minimizes number of evictions

2.10 Optimal Caching Solution

Evict the item in the cache that is not requested until farthest in the future

2.11 Huffman Coding Algorithm

```
function huffman(nodeQueue) {
    while (nodeQueue.size() > 0) {
        x = nodeQueue.removeMin();
        y = nodeQueue.removeMin();
        z = combineNodes(x,y);
        nodeQueue.insert(z);
    }
}
```

- In the Huffman coding algorithm, `nodeQueue` is a minimum priority queue implemented using a Fibonacci heap
  - A Fibonacci heap has $O(1)$ `insert()` / `decrKey()` / `findMin()` time complexity and $O(\log n)$ `removeMin()` time complexity
- Huffman coding algorithm has $O(n \log n)$ running time

3 Divide & Conquer

3.1 Divide & Conquer

Divide & conquer algorithms recursively breaks down a problem into two or more subproblems, until these become small enough to be solved directly

3.2 Closest Pair Algorithm

```
// inv: points is an array of points (x,y) sorted in increasing order of x
function int closestPair(points, lo, hi) {
    if (hi - lo == 1) {
        return distance(points[lo], points[hi]);
    } else if (hi - lo == 0) {
        return null;
    }
    mid = floor((hi - lo) / 2 + lo);
    midPoint = points[mid];

    leftDist = closestPair(points, lo, mid);
    rightDist = closestPair(points, mid, hi);
    sepDist = min(leftDist, rightDist);

    sepPoints = <points with x-values within separationDistance of midPoint.x>
    <sort sepPoints in increasing order of y>

    for (int i = 0; i < sepPoints.size(); i++) {
        j = i + 1;

        while (j < sepPoints.size() && sepPoints[i].y - sepPoints[j].y < sepDist) {
            minDist = min(sepDist, distance(sepPoints[i], sepPoints[j]));
            j++;
        }
    }
    return minDist;
}
```

- Closest pair algorithm has $O(n \log^2 n)$ running time
  - This implementation does not have $O(n \log^2 n)$ running time

3.3 Find k^{th} Smallest Element Algorithm

```
function int findKthSmallest(array, k, lo, hi) {
    partitionElt = <select median element in array[lo:hi] inclusive>
    arrayPartition = partition(array, partitionElt);

    partitionIndex = arrayPartition.getIndex(partitionElt);

    if (k - 1 == partitionIndex) {
        return partitionElt;
    } else if (k - 1 < partitionIndex) {
        return findKthSmallest(array, k, lo, partitionIndex - 1);
    } else {
        return findKthSmallest(array, k - partitionIndex - 1, partitionIndex + 1, hi);
    }
}
```

- Find k^{th} smallest element algorithm has $O(n)$ running time
  - This implementation has $O(n \log n)$ average running time

4 Dynamic Programming

4.1 Dynamic Programming

Dynamic programming algorithms recursively break down a problem into simpler subproblems. They use the fact that the optimum solution to the overall problem depends upon the optimal solution of the individual subproblems

- Dynamic programs can be implemented linearly or recursively
- If the problem is very large, then the call stack size of recursive dynamic programs may exceed the available stack frame size

4.2 Single Objective Linear Dynamic Programming

```
function int[] optimal(weight, n) {
    memo = [0];
    solution = [[]];

    for (int i = 1; i < n + 1; i++) {
        if (memo[i - 1] + weight[i] > memo[i - 1]) {
            memo[i] = memo[i - 1] + weight[i];
            solution[i] = union(solution[i - 1], [i]);

        } else {
            memo[i] = memo[i - 1];
            solution[i] = solution[i - 1];
        }
    }
    return solution[n];
}
```

- Given a set of items $\Omega = \{1, ..., n\}$, this program finds a subset Λ that maximizes $\sum_{i \in \Lambda} \texttt{weight[i]}$
- This algorithm has $O(n)$ running time

4.3 Single Objective Recursive Dynamic Programming

```
function (int, int[]) optimal(weight, n) {
    value, solution = optimal(weight, n - 1);

    if (value + weight[n] > value) {
        return ((value + weight[n]), union(solution, [n]));
    } else {
        return (value, solution);
    }
}
```

- Given a set of items $\Omega = \{1, ..., n\}$, this program finds a subset Λ that maximizes $\sum_{i \in \Lambda} \texttt{weight[i]}$
- This algorithm has $O(n)$ running time

4.4 Dual Objective Linear Dynamic Programming

```
function (int, int[]) optimal(value, weight, n, w) {
    memo = (int, int[])[n + 1, w + 1];

    for (int j = 0; j < w + 1; j++) {
        memo[0, j] = (0, []);
    }
    for (int i = 1; i < n + 1; i++) {
        for (int j = 1; j < w + 1; j++) {
            if (weight[i] > j) {
                memo[i, j] = memo[i - 1, j];

            } else {
                valueSelectI, solutionSelectI = memo[i - 1, j - weight[i]];
                valueNoSelectI, solutionNoSelectI = memo[i - 1, j];

                if (valueSelectI + value[i] > valueNoSelectI) {
                    memo[i, j] = (valueSelectI + value[i], union(solutionSelectI, [i]));
                } else {
                    memo[i, j] = (valueNoSelectI, solutionNoSelectI);
                }
            }
        }
    }
    return memo[n, w];
}

```

- Given a set of items $\Omega = \{1, ..., n\}$, this program finds a subset Λ that maximizes $\sum_{i \in \Lambda} \texttt{value[i]}$ while maintaining $\sum_{i \in \Lambda} \texttt{weight[i]} \leq w$
- This algorithm has $O(nw)$ running time

4.5 Dual Objective Recursive Dynamic Programming

```
function (int, int[]) optimal(value, weight, n, w) {
    if (n == 0) {
        return (0, []);
    } else if (weight[n] > w) {
        return optimal(n - 1, w);
    } else {
        valueSelectI, solutionSelectI = optimal(n - 1, w - weight[n]);
        valueNoSelectI, solutionNoSelectI = optimal(n - 1, w);

        if (valueSelectI + value[n] > valueNoSelectI) {
            return (valueSelectI + value[n], union(solutionSelectI, [n]));
        } else {
            return (valueNoSelectI, solutionNoSelectI);
        }
    }
}

```

- Given a set of items $\Omega = \{1, ..., n\}$, this program finds a subset Λ that maximizes $\sum_{i \in \Lambda} \texttt{value[i]}$ while maintaining $\sum_{i \in \Lambda} \texttt{weight[i]} \leq w$
- This algorithm has $O(nw)$ running time

4.6 Bellman-Ford Algorithm

```
function void bellmanFord(s) {
    for (int v = 1; i < n + 1; i++) {
        if (v != s) {
            memo[v, 0] = infinity;
        }
    }
    memo[s, 0] = 0;

    for (int i = 1; i < n; i++) {
        for (int v = 1; v < n + 1; v++) {
            memo[v, i] = memo[v, i-1];

            for <every vertex u with edge (u, v)> {
                memo[v, i] = min(memo[v, i], memo[u, i - 1] + weight(u, v));
            }
        }
    }
}
```

- Bellman-Ford outputs the shortest distance to all vertices from the source
- Bellman-Ford is a shortest path algorithm that works on graphs with negative edge weights
- Bellman-Ford has $O(nm)$ running time, where m is the number of edges

5 Network Flow

5.1 Network Flow

Network flow problems are a class of problems where the input is a directed graph with weights representing edge capacities. The goal is to construct a maximal flow from the source to the sink that respects edge capacities and has incoming flow equal to outgoing flow at each of the vertices

5.2 Ford-Fulkerson Algorithm

```
function int fordFulkerson(graph, source, sink) {
    for (edge : graph.edges) {
        edge.flow = 0;
    }
    path = findPath(graph, source, sink);
    while (path != null) {
        minFlow = 0;
        for (edge : path.edges) {
            minFlow = min(minFlow, edge.capacity);
        }
        for (edge : path.edges) {
            edge.flow += minFlow;
            edge.capacity -= minFlow;
        }
        path = findPath(graph, source, sink);
    }
    maxFlow = 0;
    for (edge : source.edges) {
        maxFlow += edge.flow;
    }
    return maxFlow;
}
```

- Ford-Fulkerson outputs the maximum flow of a graph with integer edge weights
- Ford-Fulkerson has $O(m \cdot \mathrm{OPT})$ running time, where m is the number of edges and OPT is the max flow
- Ford-Fulkerson is pseudo-polynomial since $O(m \cdot \mathrm{OPT}) = O(mnU) = (mU)^{O(1)}$
- `findPath()` can be implemented using any algorithm that finds a path from s to t
  - Common choices include breadth first search and depth first search

5.3 Edmonds-Karp Algorithm

The Edmonds-Karp algorithm is implemented the same way as the Ford-Fulkerson algorithm, except that `findPath()` is implemented using breadth first search

- Edmonds-Karp has $O(m^2n)$ running time, where m is the number of edges and n is the number of vertices
- Edmonds-Karp is strongly polynomial since $O(m^2n) = m^{O(1)}$

5.4 Residual Graph

The residual graph is an auxiliary graph generated by the Ford-Fulkerson algorithm that indicates the residual capacity and current flow of each edge. Given two adjacent vertices u and v in the residual graph, $\mathrm{weight}(u, v)$ denotes the residual capacity and $\mathrm{weight}(v, u)$ denotes the current flow of the edge (u, v)

5.5 Minimum s-t Cut

Given a directed graph with a source and a sink, find a cut that minimizes the total weight of the edges going across the partition from the s-subset to the t-subset

- The max-flow min-cut theorem states that the value of the max s-t flow is equal to the value of the min s-t cut

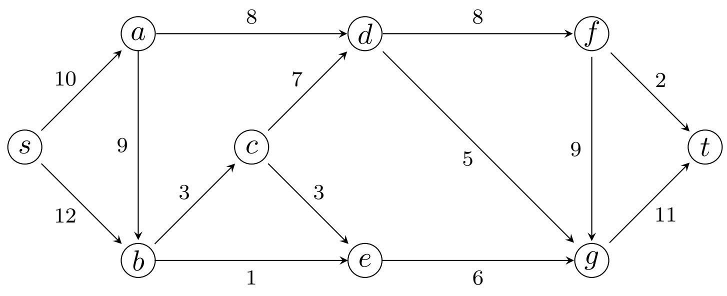
5.6 Network Flow Reductions

Network flow reductions are a class of problems that can be reduced to a network flow problem and solved using the Ford-Fulkerson algorithm

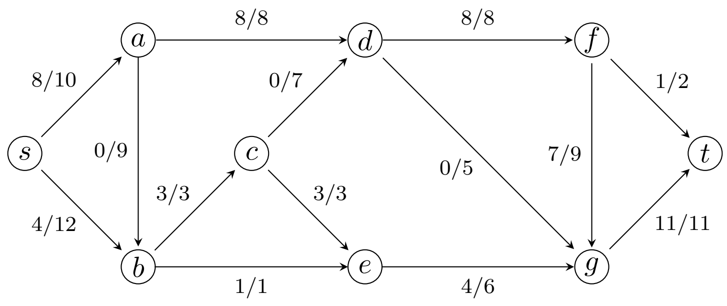
- Bipartite Matching Problem
  Given an undirected bipartite graph, find a matching with maximal cardinality
  - Given an undirected bipartite graph (A, B), construct a directed graph as follows
    - Assign infinite capacity to all edges from A to B
    - Add a source s and unit weight edges from s to each node in A
    - Add a sink t and unit weight edges from each node in B to t
  - Run Ford-Fulkerson on the constructed graph
  - The max-flow of the constructed graph corresponds to the maximal matching cardinality of the original graph
- Edge Disjoint Paths Problem
  Given a directed graph with a source and a sink, find the maximum number of s-t paths such that none of them share common edges
  - Given a directed graph, construct a directed graph such that all edges have unit weight
  - Run Ford-Fulkerson on the constructed graph
  - The max-flow of the constructed graph corresponds to the maximum number of edge disjoint paths in the original graph

5.7 Network Flow Example

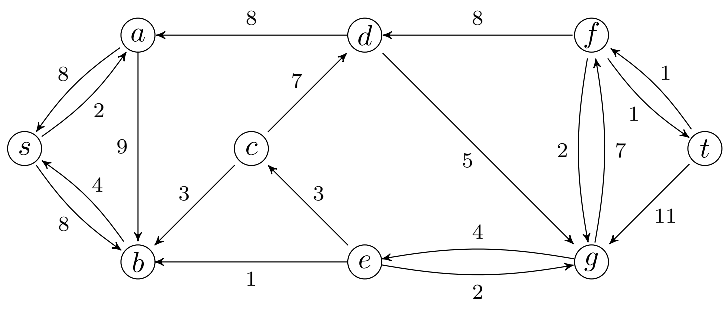
Suppose we are given the following graph



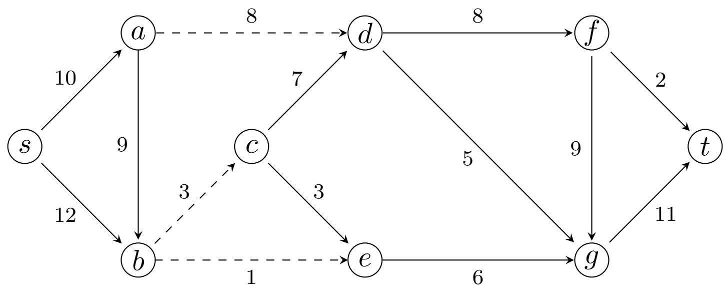
The maximum s-t flow for the graph is as follows



The corresponding residual graph for this maximum flow is as follows

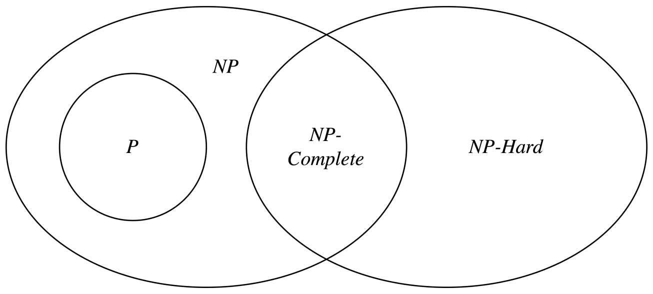


The minimum cut for this maximum flow is as follows



6 Complexity Classes

6.1 Complexity Classes



6.2 Polynomial

P is the set of all decision problems that have an algorithm that runs in time $O(n^k)$ for some constant k

- All P problems are NP problems

6.3 Non-Deterministic Polynomial

NP is the set of all decision problems such that if the answer is yes, there is a proof which can be verified in polynomial time

- NP is a superset of P

6.4 Polynomial Time Reducible

Problem A reduces to problem B in polynomial time if there exists an algorithm that, using a hypothetical polynomial time algorithm for B, solves A in polynomial time

- Problem A can be translated to problem B and solved using a polynomial time algorithm for B
- This means that problem A is easier than problem B

6.5 NP-Complete

A problem A is NP-complete if A is in NP and all problems in NP reduce to A in polynomial time

- An NP-complete problem is the hardest problem in NP
- A polynomial time algorithm for an NP-complete problem can be used to solve every problem in NP in polynomial time via reduction

6.6 NP-Hard

A problem A is NP-hard if all problems in NP reduce to A in polynomial time

- All NP-complete problems are NP-hard
- An NP-hard problem is not necessarily NP

6.7 Relative Complexity of Problems

A problem A is less complex than a problem B if there exists an algorithm that solves A using an algorithm that solves B

- If $A \leq_p B$, then problem A is less complex than problem B

6.8 Karp Reductions

A Karp reduction is a polynomial-time algorithm for transforming inputs to one problem into inputs to another problem, such that the transformed problem has the same output as the original

- If $A \leq_p^1 B$, then there exists a Karp reduction for problem A to problem B

6.9 Certificate and Verifier

- A certificate is an input to a decision problem that is supposed to represent a solution
- A verifier is a polynomial time algorithm that checks whether the provided certificate is a solution to the decision problem

6.10 Proving NP-Completeness

In order to show a problem B is NP-complete, we must do the following

- Show that problem B is NP-hard
  - Choose an NP-hard problem A we want to solve using problem B
  - Define a reduction map f from A to B
  - Prove that f takes polynomial time
  - Prove that $f(A)$ returns yes if and only if A returns yes
- Show that problem B is NP
  - Choose a certificate for problem B
  - Construct a verifier for the certificate
  - Prove that the verifier checks that the certificate satisfies the problem restrictions
  - Prove that the verifier takes polynomial time

7 Approximation Algorithms

7.1 Approximation Algorithms

Approximation algorithms are efficient algorithms that find approximate solutions to difficult optimization problems

7.2 Approximation Ratio

The approximation ratio of an algorithm is the ratio between the result obtained by the algorithm and the optimal solution. An algorithm has approximation ratio $\alpha(n)$ if

$$\frac{\text{cost of computed solution}}{\text{cost of optimum solution}} \leq \alpha(n)$$

for any input of length n

8 Asymptotic Analysis

8.1 Big-O Notation

$f(n)$ is $O(g(n))$ if there exists positive constants c, n_0 such that for all $n \geq n_0$, $f(n) \leq c \cdot g(n)$

- Big-O represents an upper bound for the algorithm run time

8.2 Big-O Theorems

- $O(\log_a n) = O(\log_b n)$
- $O(\log^k n) < O(n)$ for all $k \geq 0$
- If $a \neq b$, then $O\left(a^n\right) \neq O\left(b^n\right)$
- If $a \neq b$, then $O\left(e^{an}\right) \neq O\left(e^{bn}\right)$

8.3 Master Theorem

Suppose $T(n) = aT\left(\frac{n}{b}\right) + \Theta\left(n^k \log^p n\right)$ for all $a \geq 1,\ b > 1,\ k \geq 0$, and $p \in \mathbb{R}$. Then

- If $a < b^k$, then
  - If $p < 0$, then $T(n) = O(n^k)$
  - If $p \geq 0$, then $T(n) = \Theta\left(n^k \log^p n\right)$
- If $a = b^k$, then
  - If $p < -1$, then $T(n) = \Theta\left(n^k\right)$
  - If $p = -1$, then $T(n) = \Theta\left(n^k \log^2 n\right)$
  - If $p > -1$, then $T(n) = \Theta\left(n^k \log^{p+1} n\right)$
- If $a > b^k$, then $T(n) = \Theta\left(n^{\log_b a}\right)$

8.4 Directed Graph Algorithm Runtime

Given a directed graph with integral capacity $0 \leq c_e \leq U$, the input size is $O(m \log U)$

- A pseudo-polynomial algorithm has runtime $(mU)^{O(1)}$
- A weakly polynomial algorithm has runtime $(m \log U)^{O(1)}$
- A strongly polynomial algorithm has runtime $m^{O(1)}$