CSCE 629 Final Review Sheet

Graph Theory

DFS: aka Depth-First Search. See pseudocode below. (Normally recursive implementation)

Algorithm 1 DFS

```
function DFS(G, u)
    u.visited = true
    for each v ∈ G.Adj[u] do
        if v.visited == false then
            DFS(G,v)
function INIT( )
    for each vertex u ∈ G do
        u.visited = false
    for each vertex u ∈ G do
        DFS(G, u)
```

BFS: aka Breath-First Search. See pseudocode below. (Normally implemented using a queue instead of recursion)

Notice that a queue is open ended; first-in, first out structure (unlike DFS which uses a stack)

Algorithm 2 BFS

```
function BFS(G, u)
    Set all nodes ∈ G to "not visited";
    q = new Queue();
    q.enqueue(u);
    while q ≠ ∅ do
        x = q.dequeue();
        if x.visited == false then
            visited[x] = true;
            for every edge (x, y) do
                if y.visited == false then
                    q.enqueue(y);
```

de Bruijn graph: could be used to solve the *rotating drum* problem. (Frequently used in computational biology to solve the sequence assembly problem)

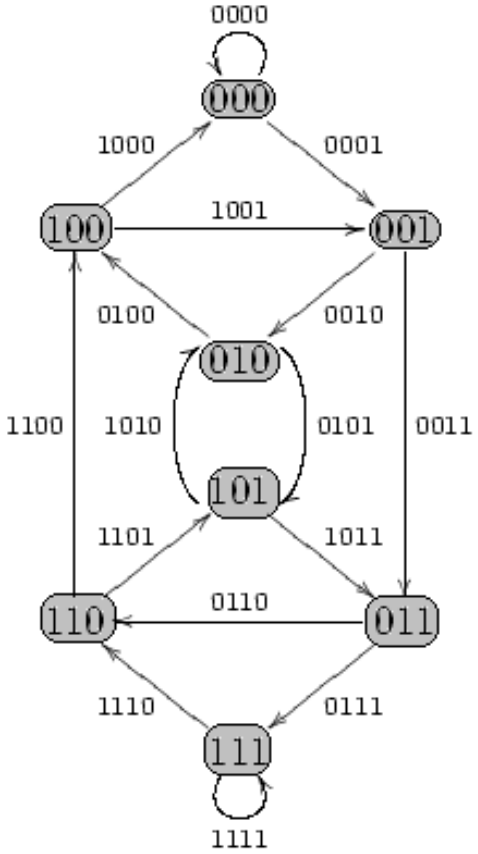
★ **Rotating drum problem**: Given a drum with 8 cells, put 0 or 1 in each cell so that if reading clockwise, each triple appears exactly once.

Definition: Given a parameter k, we construct a directed graph. 1) Each string of length k is a vertex. 2) Two vertices are connected by a directed edge if the $(k-1)$ suffix of the first vertex is the same as the $(k-1)$ prefix of the second vertex.

Property: A de Brujin graph with parameter k has exactly 2^{k+1} edges, with each $(k+1)$ tuple appearing exactly once.

→ An Eulerian path in the de Brujin graph with $k = 2$ solves the rotating drum problem.

Below is an example with $k = 3$.



Eulerian path: a trail in a finite graph which visits every edge exactly once.

Necessary condition: All vertices except START and FINISH have even degree. The START and FINISH both have odd or even degrees at the same time.

Claim: If all the vertices of a graph are of even degree, or all vertices except 2 are of even degree and the remaining two are of odd degrees, then there exists an Eulerian path.

Proof: see notes.

Algorithm: From Hierholzer's 1873 paper.

★ Choose any starting vertex v, and follow a trail of edges from that vertex until returning to v. It is not possible to get stuck at any vertex other than v, because the even degree of all vertices ensures that, when the trail enters another vertex w there must be an unused edge leaving w. The tour formed in this way is a closed tour, but may not cover all the vertices and edges of the initial graph.

★ As long as there exists a vertex u that belongs to the current tour but that has adjacent edges not part of the tour, start another trail from u, following unused edges until returning to u, and join the tour formed in this way to the previous tour.

Time complexity: $O(|E|)$. This problem is in P.

Connected components: a connected component of an *undirected* graph is a subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the supergraph.

Algorithm: do either BFS or DFS starting from every unvisited vertex. Eventually all strongly connected components will be found. (Note: add a print statement at the DFS function, right after the recursive call)

Time complexity: $O(|V| + |E|)$ (same as DFS)

Strongly connected components: Given a directed graph G, a strongly connected component is a maximal set of vertices so that for each pair of vertices u and v, there is a path from u to v and a path from v to u. (The set is maximal; each vertex is reachable from every other vertex within a strongly connected component)

Algorithm: Kosarajus algorithm. Denote G^T as the directed graph of G with the direction of all its edges reversed.

★ Call DFS(G) and record the FINISH time $f(v)$ of each vertex v. (FINISH time of vertex v = the time when DFS(v) returns. Could be captured by the PRINT statement)

★ Call DFS(G^T) except that the vertices are considered in a decreasing order of FINISH time.

Property: Given 2 strongly connected component C and C', if there is an edge e from C to C', then the FINISH

time of vertices in C is larger than the FINISH time of vertices in C'. (See proof in notes)

Time complexity: $O(|V|+|E|)$ (same as DFS - called twice)

Topological sorting: Given a directed acylic graph (DAG), we want to impose an ordering on the vertices so that whenever there is an edge $u \to v$, vertex u is placed before vertex v in the ordering. Note that topological sorting only works for DAG.

Algorithm: use a modified DFS.

✯ Reverse all the arrows in the graph before applying DFS. (Alternatively, use a stack)

✯ Add a $print(u)$ statement in the function DFS(G,u).

Note: the print statement serves as a record of the FINISH time.

Time complexity: $O(|V|+|E|)$ (same as DFS)

Shortest Path: Given a connected directed graph G=(V,E) with positive weight on each edge (u,v), denoted as w(u,v), there could be **two** types of the shortest path problems. (w(u,v) = inf if there is no edge from u to v)

✰ Single-source shortest path: Given a vertex v_0, find the shortest path from v_0 to all other vertices.

✰ All-pairs shortest path: Find the shortest paths between all pairs of vertices.

Single-source shortest path algorithm: use Dijkstra. Essentially a greedy approach.

Note: $d(v)$ records the current best shortest path distance from v_0 to v. At the end, $d(v)$ is the best distance.

Algorithm 3 Dijkstra's algorithm

```
function SHORTEST_PATH(G, v_0)
    S ← {v_0}
    d(v_0) ← 0
    for each vertex v ∈ V − {v_0} do
        d(v) ← w(v_0, v)
    while S ≠ V do
        let u be the vertex V − S such that d(u) is minimized
        S ← S ∪ {u}
        for each vertex v ∈ V − S do
            d(v) ← min(d(v), d(u) + w(u, v)) ▷ Relaxation step
```

To get the actual shortest path, backtrace along the minimum comes from in the relaxation step.

Time complexity: $O(|V|^2)$ as there are $|V|$ iterations, each takes $O(|V|)$ time.

All pair shortest path algorithm

✰ One way to solve it is to use single-source shortest path repeatedly. Time complexity would be $O(|V|^3)$ as there are $|V|$ vertices to run in total.

✰ Another way to solve this is to use DP.

1) Label each vertex from 1 to n in an arbitrary way.

2) Let $d_{ij}^{(k)}$ be the length of the shortest path from i to j using only vertices 1...k

3) Base case: $d_{ij}^{(0)} = w_{ij}$, i.e. weight of the edge from i to j

Recurrence:

$$d_{ij}^{(k)} = min \begin{cases} d_{ij}^{(k-1)} & \text{if not using vertex } k \\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)} & \text{if using vertex } k \end{cases}$$

Note: DP would be a bottom up approach. This is more straightforward but still takes $O(n^3)$ time.

Longest Path: This problem is NP-hard. It doesn't have the optimal substructure property as the shortest path problem does. Nonetheless, it's possible to find a polynomial time solution if the given graph is DAG.

Algorithm: let G = graph to process, V = all vertice, and longest = -∞.

Algorithm 4 Longest path DAG

```
function LONGEST_PATH(v_0)
    d(v_0) = 0
    for each vertex v ∈ V − {v_0} do
        d(v) = −∞
    toplogical_sort(G);
    for each u in {V} in topological order do
        for v in {neighbor of u} do
            if d(v) < d(w) + w(u, v) then
                d(v) = d(w) + w(u, v)
```

To find the longest path: backtrack along the $d(v) = d(u) + w(u,v)$ step.

Time complexity: $O(|V|+|E|)$ as topological sorting takes $O(|V|+|E|)$ time, there are $O(|E|)$ adjacent vertices in total and the inner loop runs $O(|V|+|E|)$ time.

Network Flow, Matching, and Linear Programming

Network flow: Given a directed graph $G = (V, E)$ with positive edge weights called capacities (denoted by $c(u, v)$ for each edge (u, v)), define a flow in the graph (aka a network) as follow:

A flow satisfies 3 properties: (a value $f(u, v)$ is assigned for each edge (u, v))

1) Capacity constraint: $f(u, v) \leqslant c(u, v)$

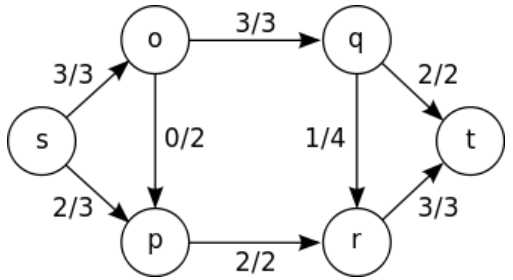
2) Skew symmetry: $f(u, v) = -f(v, u)$

3) Flow conservation: There are 2 distinguished vertices called s (source) and t (sink). Except for s and t, for each vertex u in $V - \{s, t\}$, $\sum_{v \in V} f(u, v) = 0$.

Goal: Find the maximum flow from the source s to the sink t, where the flow is defined as $\sum_{v \in V} f(s, v)$.

✲ If there is no edge from u to v, its capacity $c(u, v) = 0$.

Notation in the diagram: flow/capacity



Residual network: This graph has the same vertices as the original network, with positive edge weights $C_f(u, v) = c(u, v) - f(u, v)$. (Not represented here)

Augmenting path: a path that does not contain cycles through the graph, using only edges with positive capacity from the source to the sink.

Algorithm 1: Ford-Fulkerson.

1) Given a flow network, initialize flow to 0.

2) Construct the residual network with respect to the flow.

3) If there is no path from s to t in the residual network, we are done.

4) Otherwise, we can increase the flow by adding the minimum residual capacity along the path. This gives us a new flow network. Go back to step 2) and repeat until hitting the termination criterion in 3).

✲ the algorithm above won't loop forever if all the capacities are integers.

Time complexity: $O(|E||f|^*)$ since we can use DFS to find an augmenting path in $O(|E|)$ time. If the capacities

are integers, the maximum number of iterations is the value of the maximal flow $|f|^*$.

Algorithm 2: Edmond-Karp. Improvement on Ford-Fulkerson; instead of using an arbitrary augmenting path, we always use the shortest augmenting path in each iteration.

✳ "Shortest" in a sense of the actual shortest path *without considering the edge weights*. This could be found using BFS.

See the proof in note - each edge can be a bottleneck edge for a polynomial number of times (to be exact: $\frac{|V|}{2}$).

Time complexity: $O(|V||E|^2)$ since the maximum number of iterations is $O(|V||E|)$. Within each iteration, BFS takes $O(|E|)$ time.

Minimum cut: Given a flow network $G = (V, E)$, a cut is a partition of V into two parts S and T so that $s \in S$ and $t \in T$.

Define the capacity of the cut to the total capacity going from S to T: $\sum_{x \in S} \sum_{y \in T} c(x, y)$.

The **minimum cut** is the one with the lowest total capacity.

Max flow, min cut theorem: The value of the maximum flow is equal to the value of the minimum cut.

Proof: see notes. Two properties: 1) the maximum flow corresponds to some cut in the network. 2) the value of the maximum flow is at most the value of any cut in the network.

✶ There is no easy way to solve a minimum cut problem. One of the application of this theorem is that we can reduce to a maximum flow problem instead. (If nodes are accessible from the source node, then they belong to the set S. Otherwise they'll be in the set T)

Maximum bipartite matching: A matching in a Bipartite Graph is a set of the edges chosen in such a way that no two edges share an endpoint. A maximum matching is a matching of maximum size (maximum number of edges). In a maximum matching, if any edge is added to it, it is no longer a matching. There can be more than one maximum matchings for a given Bipartite Graph.

To solve this: reduce it to a network flow problem.

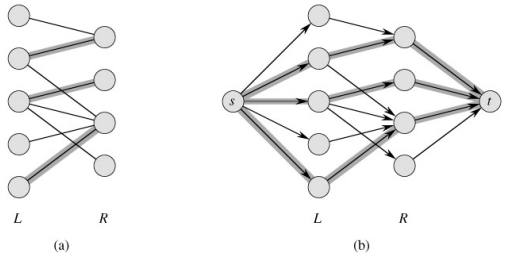
1) Add source node s and sink node t.

2) Add an directed edge from s to all vertices in L.

3) Add an directed edge from all vertices in R to t.

4) Add a direction for edges from L to R.

5) Put a capacity of 1 on each edge.



Maximum matching problem in general: cannot be easily reduced to a maximum flow network as the bipartite matching problem above. Nonetheless, a similar augmenting approach can be employed.

✶ the augmenting paths are of the form of an alternating path (some are in the matching set, some are not). It can be proved that when there is no augmenting path, we have a maximum matching.

Time complexity: ???

Linear programming: A linear program is of the following form:

max (or min) (a linear objective function) a set of linear constraints/inequalities

Remark: 1) The simplex method is mostly fast but in the worst case takes exponential time.

2) The ellipsoid method developed in the late 90's solves the problem in polynomial time but it's complicated.

Reduction to linear programming:

i) Network flow problem: In the linear program, $f(u, v)$ are variables for each pair of vertices $u, v \in V$.

Linear program formulation:

max $\sum_{v \in V} f(s, v)$, subject to: $f(u, v) \leqslant c(u, v)$ for $u, v \in V$ $f(u, v) = -f(v, u)$ for $u, v \in V$ $\sum_{v \in V} f(u, v) = 0$ for $u \in V - \{s, t\}$

ii) Maximum matching problem in a "general" undirected graph: assume that $m(u, v) = 1$ if (u, v) is in the matching, $m(u, v) = 0$ otherwise.

Linear program formulation:

max $\sum_{u \in V} \sum_{v \in V} m(u, v)$, subject to: $\sum_{v \in V} m(u, v) \leqslant 1$ for $u \in V$ $m(u, v) \geqslant 0$ for $u, v \in V$ $m(u, v) \leqslant 1$ for $u, v \in V$

! IMPORTANT: $m(u, v)$ has to be integer. Thus, integer linear programming has to be used here.

Note: no known polynomial time solution for integer linear programming problems, but heuristics existed to solve it (takes exponential time in the worse case but very fast in practice).

Randomized Algorithms

✲ **Monte Carlo algorithms**: have deterministic time complexity, but the result is not guaranteed.

✲ **Las Vegas algorithms**: time complexity is dependent on value of random variable, but the result is always correct. These algorithms are typically analysed for expected worst case.

Randomized quick sort: randomized version of the original quick sort algorithm. Instead of using the first element as the pivot, pick a random element.

Algorithm:

1) Pick a random number in the array as the pivot

2) Rearrange the numbers in the array so that numbers $\leqslant$ the pivot are to the left of it, and numbers $\geqslant$ the pivot are to the right of it.

3) Recursively call the randomized quick sort algorithm to the left and to the right.

Time complexity: $O(n * logn)$ independent of the distribution of numbers, no assumption is needed.

A little proof here: Let X_{ij} be the indicator variable so that $X_{ij} = 1$ if $S_{(i)}$ and $S_{(j)}$ are compared during randomized quicksort, $X_{ij} = 0$ otherwise.

$\therefore$ The expected time complexity $E(\sum_{i=1}^{n} \sum_{j=1}^{n} X_{ij}) = \sum_{i=1}^{n} \sum_{j=1}^{n} E(X_{ij})$ where $E(X_{ij}) = 1 * P_{ij} + 0 * (1 - P_{ij}) = P_{ij}$.

Probability that $S_{(i)}$ and $S_{(j)}$ are compared during randomized quicksort would be $P_{ij} = \frac{2}{j-i+1}$ since there are $j - i + 1$ items in total between $S_{(i)}$ and $S_{(j)}$, 2 comes from having either $S_{(i)}$ and $S_{(j)}$ before $S_{(l)}$ (assuming a sequence $S_{(i)}...S_{(l)}...S_{(j)}$ where $i < l < j$).

$\therefore$ Expected time complexity $= \sum_{i=1}^{n} \sum_{j>i}^{n} P_{ij}$
$= \sum_{i=1}^{n} \sum_{j>i}^{n} \frac{2}{j-i+1} \leqslant \sum_{i=1}^{n} \sum_{k=1}^{n} \frac{2}{k} = 2 \sum_{i=1}^{n} \sum_{k=1}^{n} \frac{1}{k}$
(Harmonic sum) $= O(n * logn)$.

✸ Note: Randomized quick sort is a Las Vegas algorithm.

Randomized minimum cut: Given an undirected, unweighted graph, a minimum cut is the minimum number of edges to remove so as to split the graph into 2 or more parts (aka connected components).

Algorithm: Kargers Algorithm.

1) Initialize contracted graph CG as copy of original graph.

2) While there are more than 2 vertices.

a) Pick a random edge (u, v) in the contracted graph.

b) Merge (or contract) u and v into a single vertex (update the contracted graph).

c) Remove self-loops

3) Return cut represented by two vertices. (In the end, the number of edges between the remaining two vertices constitute a cut)

✸ Note: Randomized minimum cut is a Monte Carlo algorithm - that is, the result is not always a ***minimum*** cut.

To mitigate: Run the algorithm above for $\frac{n^2}{2}$ times. This is called **Amplification**. Eventually, after running it multiple time, the probability of obtaining a minimum cut will eventually greater than 1.

Proof: Let k be the size of the minimum cut. Then the graph has at least $\frac{kn}{2}$ edges, where n is the number of vertices in the graph (otherwise, there will be a vertex of degree less than k).

Probability that an edge in the minimum cut is chosen during the first contraction is $\leqslant \frac{k}{kn/2} \leqslant \frac{2}{n}$. It means that the probability that an edge in the minimum cut that is not chosen during the first contraction $\geqslant 1 - \frac{2}{n}$.

After the first contraction, only $n-1$ vertices are left.

$\therefore$ the probability that an edge in the minimum cut that is not chosen during the second contraction $\geqslant 1 - \frac{2}{n-1}$.

If no edge in the minimum cut is chosen during the contraction steps, then the correct minimum cut is returned. Probability that this is happening $\geqslant (1-\frac{2}{n})(1-\frac{2}{n-1})...(1-\frac{2}{3}) = \frac{2}{n(n-1)}$.

If the algorithm is run only once, prob. that ta correct minimum cut is returned $\geqslant \frac{2}{n^2}$. Repeatedly running the algorithm for $\frac{n^2}{2}$ times will lead to the probability increased to $\geqslant 1 - (1-\frac{2}{n^2})^{n^2/2} \geqslant 1 - \frac{1}{e} \approx 0.632$.

NP Completeness

Turing machine, formal formulation: won't be in the test, but just to set up the concept here.

A Turing machine T is a 7-tuple consists of the following:

1. an alphabet S called the state alphabet,
2. an element $s \in S$ called the start state,
3. an element $t \in S$ called the accept state, and
4. an element $r \in S$ such that $r \neq t$, called the reject state,
5. an alphabet Σ called the input alphabet,
6. a symbol called the blank symbol B not in Σ,
7. a function $\delta : S \times \Sigma \to S \times \Sigma \times L, R$ called the transition function, such that $\Sigma(t,a) = \Sigma(t,b,X)$ and $\Sigma(r,c) = \Sigma(r,d,Y)$, where $a,b,c,d \in \Sigma \cup \{B\}$ and $X, Y \in L, R$.

For more details: refer to the note.

Turing machine, informal description, examples:

1) Language $L = \{0^n 1^n | n \geqslant 0\} = \{\varepsilon, 01, 0011, 000111, ...\}$

Informal definition: Keep matching the leftmost 0 with the rightmost 1, change both of them to be B. If at the end, all positions become B, then accept.

2) Language
$L = \{ww^R | w \in \Sigma^*\} = \{\varepsilon, 00, 11, 0000, 0110, 1001, ...\}$
(note: w^R is the reversed string of w)

Informal definition: Iteratively match the leftmost letter with the rightmost letter, changing both of them to be B. If all positions are B at the end, accept.

3) Language
$L = \{w\#w | w \in \Sigma^*\} = \{\#, 0\#0, 1\#1, 00\#00, ...\}$

Informal definition: Iteratively match the leftmost letter in the first half (changing it to B) and the leftmost letter in the second half (changing it to X). If at the end, we are left with # followed by some Xs, we accept.

4) Language $L = \{ww | w \in \Sigma^*\}$ (harder than 3) but can be solved using NTM)

Informal definition, DTM: Find the middle by iteratively matching the leftmost letter with the rightmost letter, changing 0 to X and 1 to y. At the end, the tape head is in the middle. Insert # at the middle, pushing all letters in the second half one cell to the right. Solve the $w\#w$ problem as above.

Informal definition, NTM: Guess the midpoint by moving the tape head some number of times. Insert # at the middle. Solve the problem as $w\#w$ then.

Nondeterministic Turning machine (NTM): everything is the same as deterministic Turing machine, except that the transition function is of the form $\delta(q,a) = \{(p_1,b_1,D_1),(p_2,b_2,D_2),...(p_k,b_k,D_k)\}$. For each pair q and a, there are k possibilities where $k \geqslant 0$.

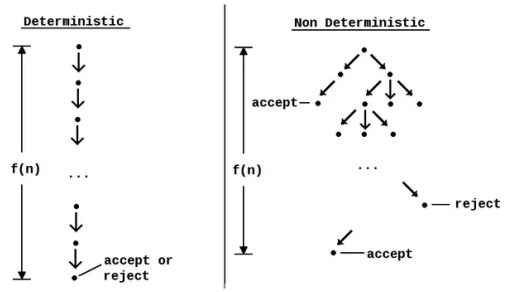
P = polynomial time = $U_i TIME(n^i)$, where $TIME(f(n))$ is the deterministic time complexity class that includes all the problems that can be solved by a deterministic Turing machine in time $O(f(n))$.

✶ Given an input size n, the time complexity of a deterministic Turing machine is *the maximum number of steps that a TM uses to reach a final state over all input size n.*

NP = nondeterministic polynomial time = $U_i NTIME(n^i)$, where $NTIME(f(n))$ is the nondeterministic time complexity class that includes all the problems that can be solved by a nondeterministic Turing machine in time $O(f(n))$.

✶ Given an input size n, the time complexity of a nondeterministic Turing machine is *the maximum number of steps that a NTM uses over all branches of computation and over all input of size n.*

Below is a visualization between P and NP. Note: **NP** $\neq$ **Non Polynomial!!!**



Polynomial time computable function: a function $f : \Sigma^* \to \Sigma^*$ so that there exists a polynomial time TM that on input w, halts with $f(w)$ on its tape.

Polynomial reducible: If a problem A can be reduced to a problem B in polynomial time, then we say that $A \leqslant_P B$. (In other words, there exists a polynomial time computatble function f such that $w \in A$ iff $f(w) \in B$)

NP-hard: a problem B is called NP-hard if for all A in NP, $A \leqslant_P B$. (i.e. every problem that can be solved in NP can be reduced to B)

NP-complete: if B is NP-hard and also in NP, then we say that B is NP-complete.

Properties: 1) if $B \in P$ and $A \leqslant_P B$, then $A \in P$.

2) If A is NP-complete and $A \leqslant_P B$ with $B \in NP$, then B is also NP-complete.

✸ First NP-complete problem: SAT, proved from scratch. Based on the definition of Turing machines.

SAT: aka boolean satisfaction problem. Suppose that there exists a boolean formula in conjunctive normal form that consists of clauses connected by $\wedge$, in which each clause consists of literals connected by $\vee$, with each literal being a variable or a negated variable. The question then becomes: is it possible to find a truth assignment for this boolean formula?

$\rightarrow$ Proof of NP-completeness of SAT: Using the Cook - Levin theorem.

✺: Starting from SAT, the problem can be further reduced to 3-SAT, then vertex cover, etc. (See notes to see how these reductions are done - will not be asked in exam)

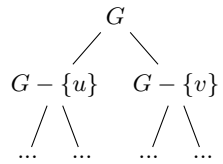
Standard procedure to prove NP completeness:

1) Show that the problem $L \in NP$.
2) Using a known NP complete problem $A \leqslant_P L$:
   i) Show a mapping from A to L.
   ii) Show the mapping takes polynomial time.
   iii) Show mapping preserves YES or NO answer.

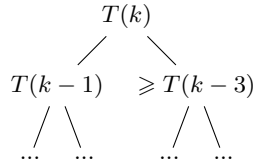
NP complete proof matters since it determines how much time we should put into effort in looking for a polynomial time solution.

Approximation Algorithms

Minimum vertex cover: given an undirected graph $G = (V, E)$, a vertex cover C is a subset of vertices so that for each edge $(u, v) \in E$, at least one of u or v is in C. We want to find a vertex cover of the minimum size.
✷ Note: this problem is NP-hard, but we can find an approximation algorithm for it.
Algorithm: Consider the maximal matching problem. Recall that a matching is a set of edges that don't touch each other.
1) Find a maximal matching M. Let $|M|$ be the number of edges in the matching.
2) Return all the verties that are incident to the edges in M.
Claim: we get a vertex cover that is at most twice the size of the optimum.
Let $|C|$ be the size of the minimum vertex cover.
Proof: ✱ 1) $|C| \geqslant |M|$. Since the edges in M don't touch each other, for each edge in M, one of its incident vertices has to be in C.
✱ 2) $|C| \leqslant 2|M|$. Suppose that one of the edges in G is not covered by any of the $2|M|$ vertices. This edge can be added to the matching M to extend the set, which is a contradiction (since the matching is already the maximal matching, meaning that no more edge can be added to the set).
Note: using the **maximum** matching doesn't help. Maximum matching = maximum # of edges not incident with each other → increased the number of vertices returned. Using a **maximal** matching is enough to approximate.
Traveling salesman problem with triangle inequality: Given a complete undirected graph $G = (V, E)$ (i.e. all vertices are connected) with positive edge weights. The goal is to find a Hamiltonian cycle with minimum total edge weights.
Algorithm: convert to minimum spanning tree.
1) Find a minimum spanning tree.
2) Consider a DFS of the minimum spanning tree. Consider the path that is traversed by DFS that includes all vertices that are encountered in the reverse call. All vertices in the graph are visited at least once.
3) Convert this path to a Hamiltonian cycle by keeping only the first visit to each vertex (except the last and first vertex).
Claim: The total edge weight of this Hamiltonion cycle is at most twice of the optimum.
Proof: ✱ i) The path we use in step 2) use each edge exactly twice.
✱ ii) The optimum Hamiltonian cycle has its total edge weight at least as big as the total weight of the edges in the traversal (when each edge is used once).
✱ iii) For algorithm step 2) to 3), the total edge weight can only decrease due to triangle inequality. Keeping the first visit is the same as removing the other visits using more directed paths.
✱ iv) The traversal uses each edge exactly twice. The Hamiltonian cycle obtained in 3) cannot have a bigger total edge weights. The total edge weights in 3) is at most twice the total weights of the edges in traversal, when each edge is included once.
Combining ii) and iv), we get a bound of 2.
Fixed parameter tractability (FPT): A problem is called FPT if it can be solved in time $O(f(k)n^c)$ where n is the size of the problem, k is the parameter of the problem.
✲ Note: this is different from the approximation algorithm above. FPT manages to find an exact solution, despite taking longer time.
✺ $f(k)$ is arbitary, it could be exponential or super-exponential. n^c is polynomial, where c is a constant independent of n. We need to identify a parameter k that is small. If k is small, this problem can be considered as tractable.
✺ in FPT, $O(f(k) * n^c)$ is usually written as $O^*(f(k))$, ignoring the polynomial part.
Vertex cover revisited: Suppose that k is the size of the minimum vertex cover C.
Algorithm:
1) For each edge (u, v), one of u or v needs to be covered (or both). We can divide an edge into 2 branches, with one branch having u in C and the other having v in C.
2) For each of these branches, e.g. vertex u branch, remove u and its incident edges and recursively solve the problem.
3) At the start of each recursion, an edge (u, v) can be picked at random.



Termination criterion: whenever the graph becomes empty, all edges are covered and we have a vertex cover.
Time complexity: if k is not known in advance, a BFS strategy is used to expand the search tree, leading a time complexity of $O^*(2^k)$.
Improvement: instead of picking an arbitary edge each time, we pick a vertex v of maximum degree in G. If the maximum degree is 2, the problem can be solved in polynomial time. Otherweise, we consider two branches:
1) Put v into the vertex cover.
2) Put all vertices adjacent to v into the vertex cover.



Time complexity: can be improved to $O^*(1.46^k)$. See proof in the note.