Algorithms

Lecture : Complexity Theory

Imdad ullah Khan

Contents

1 Efficiently Solvable problems

So far in the course we have developed algorithms for sorting of n integers, finding closest pair of 2d points, shortest paths and minimum spanning trees in graphs, the best alignment of two sequences, maximum flows in networks, and so on. All these algorithms are "efficient", because in each case their time requirement grows as a polynomial function n^k, for some constant k, of the size of the input. We call them efficient algorithms because the search space (solution space) in many cases is exponential in size, and we could find the solution in polynomial time (such as $n + m$, $n \log n$, n^2, n^3).

To better appreciate such efficient algorithms, consider the alternative: In many of these problems we are searching for a solution (path, tree, permutation, matching, etc.) among exponentially many possibilities. The brute-force algorithms would check all possibilities and select the best or correct one. There are $n!$ factorial different orderings of n numbers, only one of them is ascending order sorted, there are n^{n-2} spanning trees of a graph on n vertices (out of which only a unique one could be the MST if all weights are unique), there could be exponentially many paths from s to t in general.

The efficient algorithms we designed bypass the process of exhaustively searching candidate solutions (hence avoid exponential time), using clues from the input in order to dramatically narrow down the search space. We employed algorithm design paradigms to avoid exponential time algorithms such as greedy algorithms, dynamic programming, network flow based techniques. Divide and conquer based strategies generally give us reduced running times than already polynomial time brute force algorithm. For example, such as closest pair can be found by the $O(n^2)$ brute-force algorithm of checking all pairwise distances, but we divide and conquer strategy resulted in $O(n \log n)$ algorithm.

If there is an **Polynomial-time algorithm** for a problem, we call the problem efficiently solvable. If there exists an algorithm whose worst-case running time is $O(n^k)$ (polynomial) for some constant k on input of size n, for a problem, then that problem is designated is efficiently solvable. This does not mean that n^{70} is OK, or there is no difference between n^2 and n^3. But generally, when polynomial time algorithms exist, we can do more theoretical analysis such as divide and conquer to design a new algorithm (such as the Karatsuba algorithm or the divide and conquer based algorithm for counting inversions and finding closest pair). Some we need to design better data structures to improve the running time of the same algorithm (such as what we did for the Dijkstra, Prim and Kruskal's algorithms)

2 Problems that we don't know if they are efficiently solvable

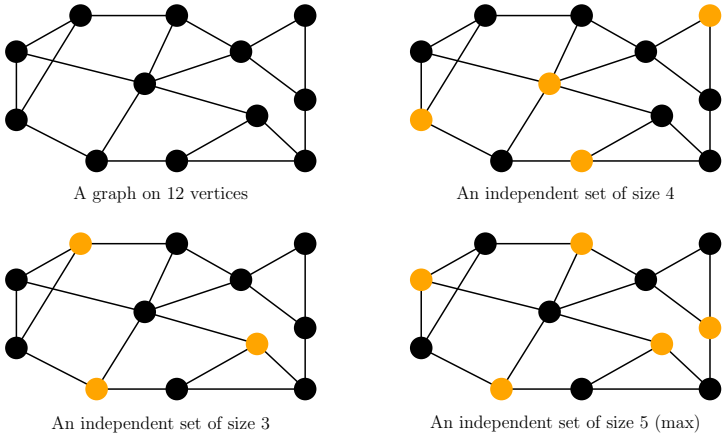
One might wonder whether all problems can be solved in polynomial time. In complexity theory we generally study negative results, i.e. we try to characterize problems about which we typically don't have a good news. **In many cases we cannot say they are not efficiently solvable (just that we do not know yet).** Once we characterize a problem to be "**Hard"** (not yet known to be efficiently solvable), we generally design approximation algorithm for them, or employ heuristic techniques or focus on special and more realistic cases of the problem.

A rough definition of "**Hard"** problems is as follows: the problems for which exponential time is enough to solve them (such as $O(n^n), O(n!), O(k^n)$) and for which no algorithm with runtime $O(n^k)$ is known. **Note that there are problems much harder than them, some are not even computable**. We shall see many problems which just like the efficiently solvable problems have exponential sized solution space but none of the above (or other known) strategies seem to be give us efficient algorithms. **For many of these problems we cannot say that they are not efficiently solvable, just that we don't an efficient algorithm "yet".**

We will give precise definitions of these hard problems and in an attempt to characterize these "hard" problems we establish that in some sense they are equivalent. First we briefly review some of the most well-known problems that are very useful in many practical situations but we do not yet know if they are efficiently solvable.

2.1 Independent Set in Graph

Definition 1 (Independent Set)**.** *Given graph $G = (V, E)$, a set of nodes $S \subseteq V$ is called an independent set if no two nodes in S are adjacent.*



A graph on 12 vertices

An independent set of size 4

An independent set of size 3

An independent set of size 5 (max)

The INDPENDENT-SET(G, k) problem is defined as follows.

PROBLEM 1 (INDPENDENT-SET(G, k) problem)**.** *Is there an independent set of size k in G?*

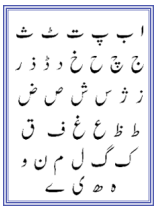
2.1.1 Applications of INDPENDENT-SET(G, k)

There are many practical applications of this classical problem. We give a few in very different areas.

- **Site Seleciton:** Suppose n potential sites are identified for opening up restaurants or franchises of a certain company. Some pairs of sites cannot have the franchises at both of them (perhaps they are too close to each other, result in unnecessary competitions, or due to some regulatory or operational constraints). Selecting k feasible sites for franchises is the problem of finding an independent set in a graph with vertices corresponding to sites and edges representing the pairwise constraints.
- **The SNP (Single Nucleotide Polymorphism) Assembly Problem:** In computational biology (biochemistry) given a set of sequences we want to resolve inter-sequential conflicts by excluding some sequences. Conflicts between two sequences are due to their biochemical properties and our goal is to select a large number of conflict-free sequences. This means in a graph with vertices representing sequences and edges representing conflicts, we want to find a large independent set.
- **Diversifying Investment Portfolio:** Different stocks are available in a market. Let $P_i(t)$ be the price for stock i at time t and let $R_i(t) = \log \frac{P_i(t)}{P_i(t-1)}$ be the return or trading volume of stock i at time t. Overtime one calculates the correlation between the returns of two stocks, essentially measuring the likelihood of both stocks simultaneously going up or down. We can represent each stock by a node

in a graph and two stocks have edge between them if the correlation of their returns is $\geq \theta$ for some threshold $-1 \leq \theta \leq 1$. The parameter θ is set depending on potential risk (degree of diversification). For example two adjacent vertices in $G_{\theta=.9}$ represent very high risk investment pair, i.e. if one invest in both of them the risk of losing is high (though the chances of earning more is also high). General θ is set depending on investment policy of an individual, a risk averse investor would for instance set $\theta < -0.5$. An independent set in G_θ represents a portfolio with "small" risk (diverse set of investments)

- **Shannon Capacity of a graph:** Suppose you are sending a message (a sequence of symbols from an alphabet) through a noisy channel.

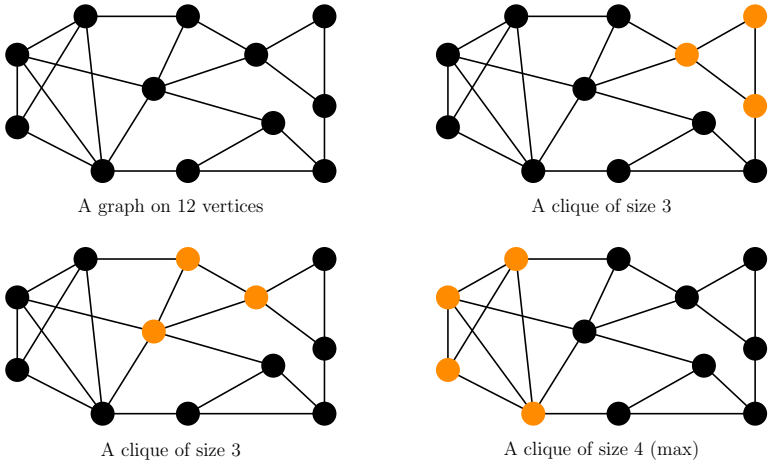


Because of noise some characters can be confused at the receiving end. We want to determine how many 1-length strings can be sent without confusion?

To answer this question, make each symbol a node in a graph and introduce an edge between a pair of nodes if and only if the corresponding symbols can be confused. This depends on the SNR of the channel and the similarity of codes for the symbols. The maximum number of messages (size 1) that can be sent without confusion is the size of maximum independent set in this graph. For the more practical scenario of finding the number of k-length strings that can be sent on this channel without confusion. We need to find the size of maximum independent set in G^k (strong product of graphs). Please read the wikipedia article for strong product of graphs and make a small example to understand the product and why an independent set in G^k correspond to the confusion free messages.

2.2 Clique in a Graph

Definition 2 (Clique)**.** *Given graph $G = (V, E)$, a set of nodes $S \subseteq V$ is called an independent set if every pair of nodes in S are adjacent.*



The CLIQUE(G, k) problem is defined as follows.

PROBLEM 2 (CLIQUE(G, k) problem)**.** *Is there a clique of size k in G?*

2.2.1 Applications of CLIQUE(G, k)

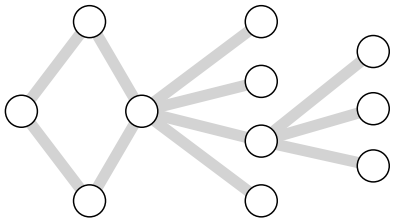
There are many practical applications of the clique problem. Think about it is in some sense the "opposite" of the INDEPENDENT-SET(G, k) problem. So all the application that we studied for that will work here too in an appropriately defined graph.

- **Cliques in Market Graphs:** A clique in the market graph defined above (e.g. $G_{\theta=.5}$ represents a high risk portfolio. Cliques in this graph can also be of interest to a regulatory body as it could represent some kind of collusion or market manipulation.
- **Organized Tax Fraud Detection by IRS:** Clustering similar objects is widely used in many applications. Ideally clusters are cliques in a graph. Generally, a community in a graph is characterized by high internal degrees, low internal distances, or large internal densities etc. so subgraphs that are close to cliques.

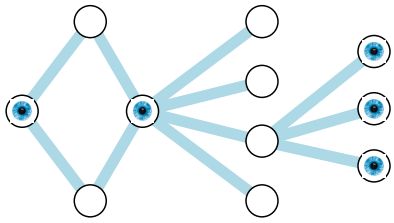
  In tax returns one can submit many phony tax returns (of small amounts) to get undeserved returns. The IRS constructed a graph, where each tax form is a vertex and edges between two vertices means the "similarity" between the two forms is above some threshold. A large clique in this graph points to a potential fraud
- To see some more problems of the cliques please search the keywords
- **Location Covering Using Clique Partition**
- **Protein Docking Problem**

2.3 Vertex Cover

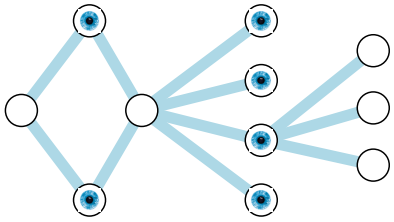
Definition 3 (Vertex Cover)**.** *Given a graph $G = (V, E)$, a set of nodes $S \subseteq V$ is called a vertex cover if every edge $e \in E$ has at least one endpoint in S.*



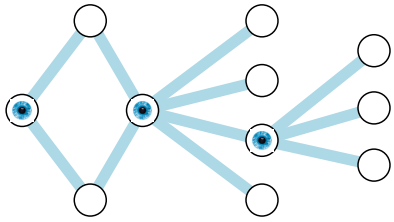
A graph on 11 vertices



A vertex cover of size 5



A vertex cover of size 6



A vertex cover of size 3

The VERTEX-COVER(G, k) problem is defined as follows.

PROBLEM 3 (VERTEX-COVER(G, k) problem)**.** *Is there a vertex cover of size k in G?*

2.3.1 Applications of VERTEX-COVER(G, k)

- **The Art Gallery Problem:** Suppose you want to assign traffic wardens to different intersections (or install cameras) in a city. Your goal is cover all street segments, so you can keep an eye on every single street. Construct a graph where intersections are nodes and streets between them are edges. You can find a vertex cover and install cameras on those intersection, this way you can keep an eye on all the streets.
- **Network Security- Rout Based Filtering:** In network design or administration, we would like to identify a small set of routers/AS, so that all packets can be monitored at those routers/switches (check if the source/destination addresses is valid given the routing table and network topology). This is important for Route-based distributed packet filtering or preventing distributed denial of service (DDOS) attacks. We would like to cover every single transmitted packet in the

network but have a limited budget (a set of k routers with enhanced capabilities). A vertex cover of size k in the naturally defined graph would do the job.

- See also vertex cover application in **Dynamic Detection of Race Condition in Shared Memory Parallel Processing**

2.4 Set Cover

Definition 4 (Set Cover)**.** *Given a set U of n elements and a collection $\mathcal{S}$ of m subsets $S_1, S_2, \ldots, S_m \subseteq U$. A Set Cover is a sub-collection $I \subset \{1, 2, \ldots, m\}$ such that $\bigcup_{i \in I} S_i = U$*

To see an example,

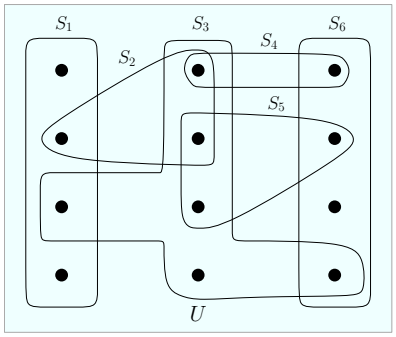
$U = \{1, 2, 3, 4, 5, 6\}$
Sets: $\{1, 2, 3\}, \{3, 4\}, \{1, 3, 4, 5\}, \{2, 4, 6\}, \{1, 3, 5, 6\}, \{1, 2, 4, 5, 6\}$

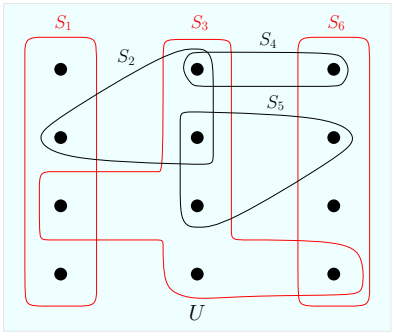
Cover $\{1, 2, 3\}, \{3, 4\}, \{1, 3, 4, 5\}, \{2, 4, 6\}$

Cover $\{1, 2, 3\}, \{3, 4\}, \{1, 3, 4, 5\}, \{2, 4, 6\}, \{1, 3, 5, 6\}, \{1, 2, 4, 5, 6\}$

Cover $\{1, 2, 3\}, \{3, 4\}, \{1, 3, 4, 5\}, \{1, 2, 4, 5, 6\}$

The first cover has size 3, the latter two have size 2 each





The SET-COVER$(U, \mathcal{S}, k)$ problem is defined as follows.

PROBLEM 4 (SET-COVER$(U, \mathcal{S}, k)$ problem)**.** *Is there a cover of size k for U?*

2.4.1 Applications of SET-COVER$(U, \mathcal{S}, k)$

- **Application Software with different capabilities:** Let U be the set of capabilities or functionalities we want in a software system. Let $\mathcal{S}$: be the set of available softwares in the market each providing a **subset** of capabilities in U. We would like to select a (small) subset of softwares for our system to provide all required functionalities.
- **IBM antivirus tool:** Suppose U is a set of (500) known viruses describe by their binaries. There is a set of 9000 strings of 20 bytes or more that occur in the binaries of viruses but not in "clean" codes. In order to determine whether a given binary is a virus, one would need to check a code for any of the 9000 strings. To do it more efficiently, we make a collection $\mathcal{S}$ of subsets of U as follows. For each string i, let S_i be the subsets of viruses containing this string. Find a set cover of size k in $U, \mathcal{S}$ to get k (small) strings to search for in codes to detect any virus.

2.5 Set Packing

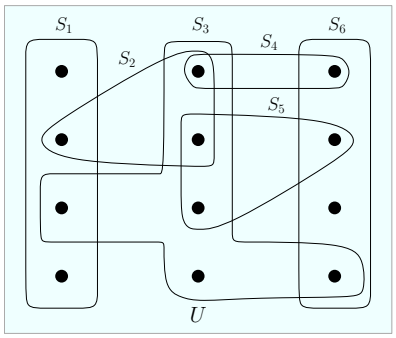
Definition 5 (Set Packing)**.** *Given a set U of n elements and a collection $\mathcal{S}$ of m subsets $S_1, S_2, \ldots, S_m \subseteq U$. A Packing is a sub-collection $I \subset \{1, 2, \ldots, m\}$ such that no two of them intersect, i.e. $\forall\, i \neq j \in I,\ S_i \cap S_j = \emptyset$*

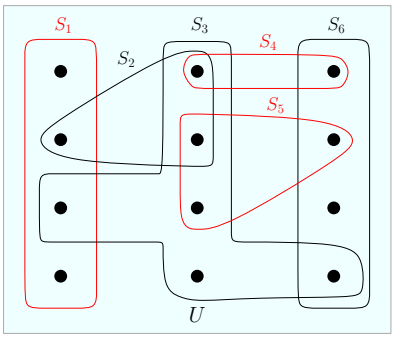
$U = \{1, 2, 3, 4, 5, 6\}$
Sets: $\{1, 2, 3\}, \{4, 5\}, \{4, 6\}, \{2, 3\}, \{1, 6\}, \{4, 5, 6\}$

Pack: $\{1, 2, 3\}, \{4, 5\}, \{4, 6\}, \{2, 3\}, \{1, 6\}, \{4, 5, 6\}$
Pack: $\{1, 2, 3\}$,$\{4, 5\}$, , $\{2, 3\}, \{1, 6\}$

Pack: $\{1, 2, 3\}, \{4, 5\}, \{4, 6\}, \{2, 3\}, \{1, 6\}$,$\{4, 5, 6\}$

The first and third pack has size 2 each, the second has size 3





The SET-PACKING$(U, \mathcal{S}, k)$ problem is defined as follows.

PROBLEM 5 (SET-PACKING$(U, \mathcal{S}, k)$ problem)**.** *Is there a packing of size k for U?*

2.5.1 Applications of Set Packing$(U, \mathcal{S}, k)$

- **Resource Sharing** U is the set of non shareable resources and each subset $S_i \in \mathcal{S}$ corresponds to the subset of resources required by the ith process. Two processes pack together if they do not have any common resource in their requirements. We want to select a (large) subset of k processes (requests) to allocate resources from U.
- **Airline Crew Scheduling:** In this U is the set of crew members (pilots, copilots, navigators,stewards and stewardesses etc.). Each set $S_i \in \mathcal{S}$ is a subset of crew members (one pilot, one copilot, and 10 steward(esses) for a flight) who are willing to work with each other, know each other languages, eligible to enter certain territories etc. The problem of scheduling k flights become that of finding a set packing of size k from $(U, \mathcal{S}$.

2.6 The Satisfiability Problem: SAT and $3-$SAT

Given n Boolean variables $x_1, \ldots, x_n$, (each can take a value of 0/1 (true/false)). A literal is a variable appearing in some formula (expression) as x_i or $\bar{x_i}$. A clause is an OR of one or more literals.

Definition 6 (CNF formula)**.** *A CNF formula (conjunctive normal form) is a Boolean expression that is* AND *of one or more clauses.*

A (CNF) formula is satisfiable, if there is an assignment of 0/1 values to the variables such that the formula evaluates to 1 (or true). Note that for a formula to satisfy every

clause has to evaluate to true which happens when at least one literal in every clause is set to true.

1. $f_1 = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \bar{x_2}) \wedge (x_2 \vee \bar{x_3})$ is a CNF formula, that is satisfiable (the assignment is $x_1 = 1, x_2 = 1, x_3 = 1$). $x_1 = 1, x_2 = 0, x_3 = 0$ is also a satisfying assignment.
2. $f_2 = (x_1 \vee \bar{x_2}) \wedge (x_1 \vee x_2) \wedge (\bar{x_1} \vee \bar{x_2}) \wedge (\bar{x_1} \vee x_2)$ is not satisfiable.

The SAT(f) problem is defined as follows.

PROBLEM 6 (SAT(f) problem:)**.** *Is there a satisfying assignment for f?*

Definition 7 (3-CNF formula)**.** *A 3-CNF formula is a CNF formula such that every clause has at most* 3 *literals.*

The 3-SAT(f) problem is defined as follows.

PROBLEM 7 (3-SAT(f) problem:)**.** *Is there a satisfying assignment for the 3-CNF formula f?*

2.6.1 Applications of Satisfiability Problems

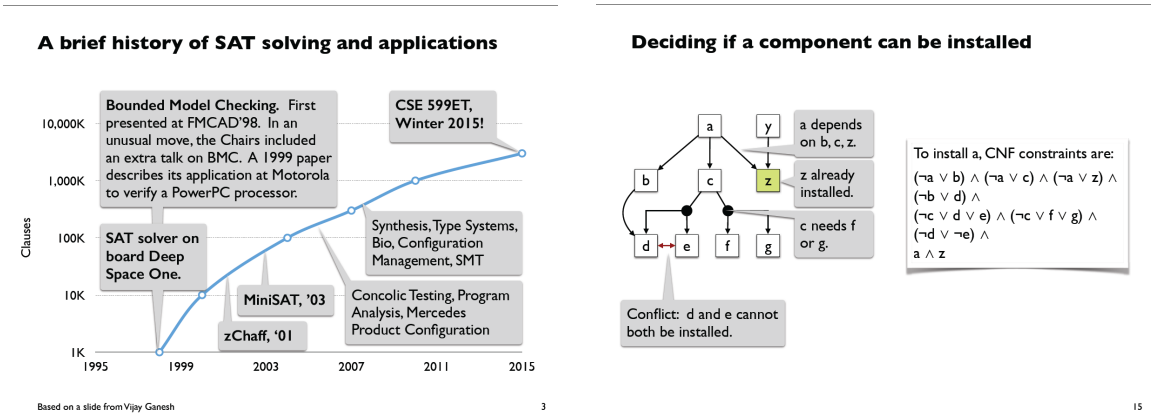


Figure 1: From slides of Emina Torlak (CS@ Uni. of Washington)

- Many applications in hardware/software verification
- Also in planning, partitioning, scheduling
- Model all kinds of constrained satisfaction problem
- Many hard problems can be stated in terms of SAT

A scheduling Problem as Constraint satisfaction problem

- Consider the following constraints:
- John can only meet either on Monday, Wednesday or Thursday
- Catherine cannot meet on Wednesday
- Anne cannot meet on Friday
- Peter cannot meet neither on Tuesday nor on Thursday
- Question: When can the meeting take place if at all?

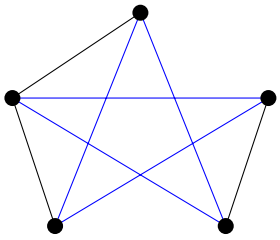
Encode then into the following Boolean formula: $(Mon \vee Wed \vee Thu) \wedge (\neg Wed) \wedge (\neg Fri) \wedge (\neg Tue \vee \neg Thu)$

The meeting must take place on Monday

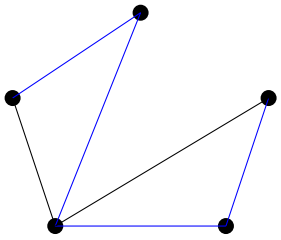
2.7 Hamiltonian Cycle and Paths in Graphs

Definition 8 (Hamiltonian Cycle (Path)). *A Hamiltonian Cycle in a graph $G = (V, E)$ is a simple cycle (path) that visits every vertex in V exactly once.*

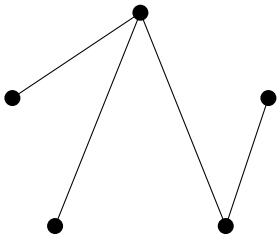
If the graph G is directed, then we refer to them as Directed Hamiltonian Cycle and Path, respectively.



Hamiltonian cycle in G



No Hamiltonian cycle in G
Hamiltonian path in blue



No Hamiltonian path in G
So no hamiltonian cycle

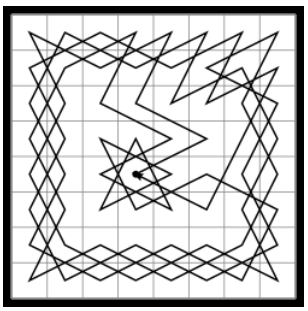
The HAM-CYCLE(G) problem is defined as follows.

PROBLEM 8 (HAM-CYCLE(G) problem:). *Is there a Hamiltonian cycle in G?*

The problems HAM-PATH(G), DIR-HAM-CYCLE(G), and DIR-HAM-PATH(G) are defined analogously.

2.7.1 Applications of Hamiltonian Cycle and related problems

- **Re-entrant Knight's Tours** Is there a sequence of moves that takes the knight to each square on a chessboard exactly once, returning to the original square. This problem was solved for 8×8 Abu Bakr Muhammad bin Yahya al-Suli; he discovered a tour in the 9th century. For $n \times n$ chessboard we construct a graph by defining a vertex for each position and connect vertex v_{ij} to vertex v_{kl} if there is a legal knight's move between the position i, j to position k, l on the board. A Hamiltonian Cycle in this graph correspond to a re-entrant knight's tour.



- **Mapping of Genomes** Scientists must combine many tiny fragments of genetic codes (call "reads"), into one single genomic sequence (a 'superstring'). If we consider each of the reads as a node in a graph and each overlap (place where the end of one read matches the beginning of another) is considered an edge. A Hamiltonian Cycle in this graph is a mapping of genomes.
- **Route for School Bus** We need to find a route for bus that passes each student's house exactly once to save fuel and time. Houses of school students in a district are considered nodes in graph and paths between them are edges. Find a Hamiltonian Cycle in the graph for a route for the bus (starting and ending at the school).

2.8 Longest Path Problem

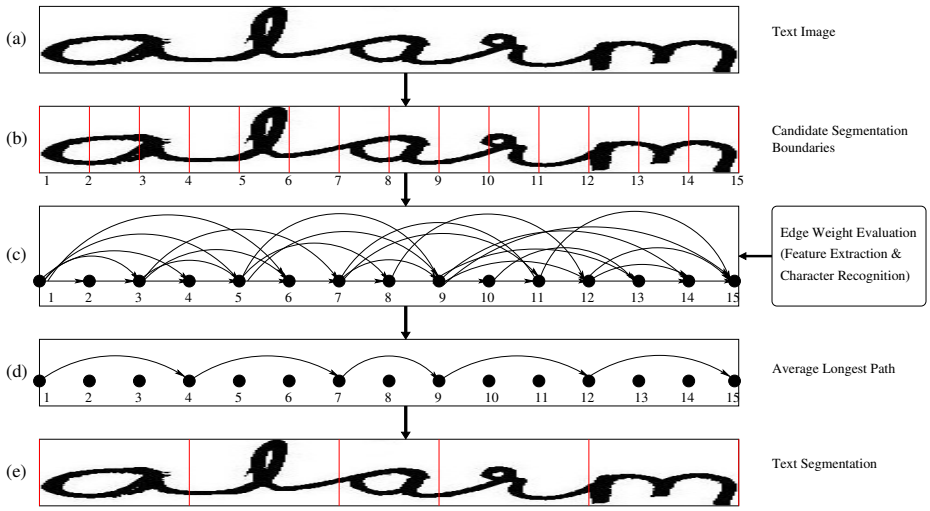
As in the shortest path problem, Given an edge-weighted graph $G = (V, E)$ with $w : E \to R$. Two vertices $s \neq t \in V$, called the source and target vertex, respectively are identified. The goal is to find a simple $s - t$ path P of maximum total weight, where weight of a path is the sum of weights of its edges, i.e. $w(P) = \sum_{e \in P} w(e)$.

The Longest-Path(G, s, t) problem is defined as follows.

PROBLEM 9 (Longest-Path(G, s, t, k) problem:). *Is there a $s - t$ path in G of weight at least k?*

2.8.1 Applications of Longest Path problem

- **Character Segmentation for OCR:**The first step in any OCR (Optical Character Recognition) system is that of character segmentation. That is given a handwritten text, we would like to isolate individual characters, that can be passed through a character recognition system. Salvi et.al. (2013). "Handwritten Text Segmentation using Average Longest Path Algorithm" proposed the following algorithm based on average longest paths.

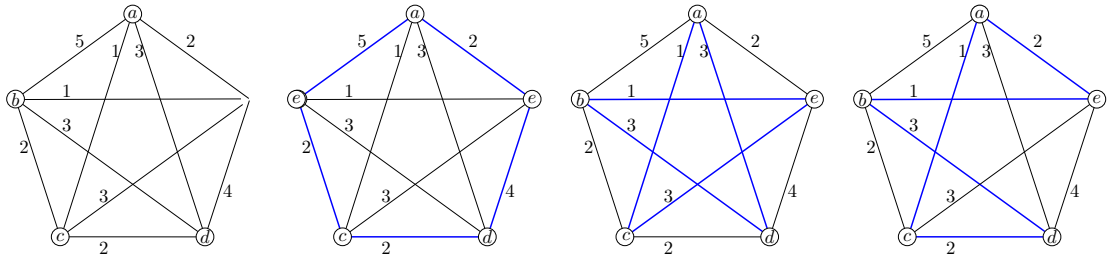


- **Static timing analysis (STA):** is a widely used method in circuit design and embedded systems. Static timing analysis (STA) is a simulation method of computing the expected timing of a digital circuit without requiring to simulate the full circuit. Longest path is used to identify critical paths in a digital integrated circuit (IC), or VLSI system and STA is performed only on these critical paths.

2.9 Traveling Salesman Problem

Given a graph with weights on edges, we want to find a Hamiltonian path (or cycle) in G, that has the least total weight. This path or cycle is called the *traveling salesman tour*. Note that in the Hamiltoninan cycle the problem is to find if there exist a tour that visits every city exactly once. Here we know that Hamiltonian Tour exists (because we assume the graph is complete, by adding non-existing edges with weights equal to ∞)

so in fact many such tours exist, the problem is to find a minimum weight Hamiltonian Cycle.



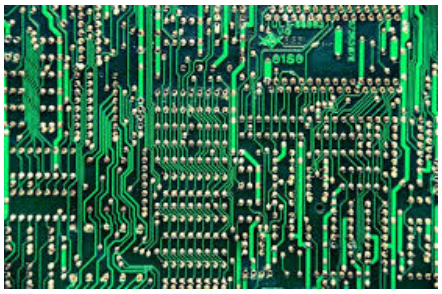
K_5 with edge weights | A TSP tour of length 15 | A TSP tour of length 11 | A TSP tour of length 9

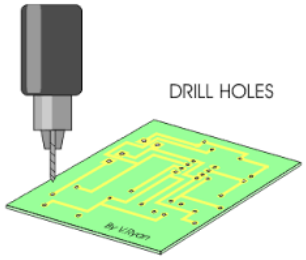
The TSP(G, k) problem is defined as follows.

PROBLEM 10 (TSP(G, k) problem:). *Is there a TSP tour in G of weight at most k?*

2.9.1 Applications of TSP

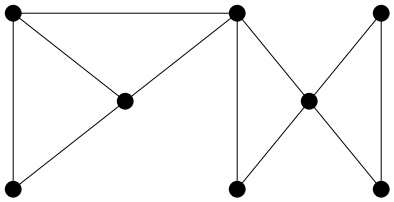
- **Transportation:** The obvious application is in logistics and transportation. Where a salesman (trucker, delivery guy) wants to visit all cities (houses or any other locations) with minimum total cost (or traveling the least total length).
- **Manufacturing Tool Optimization:** Suppose you have a tool that is used for manufacturing equipment. We would like to optimize the path of the tool so as it spends the least time in moving from one place to another.



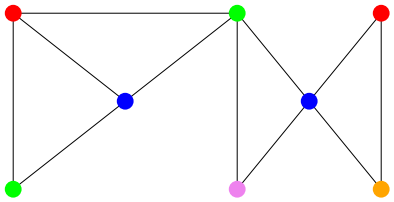


2.10 Graph Coloring

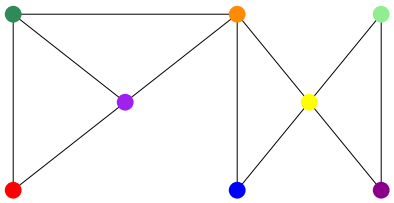
Definition 9 (Graph Coloring:). *A graph (vertex) coloring is to assign a color to each vertex such that no two adjacent vertices get the same color.*



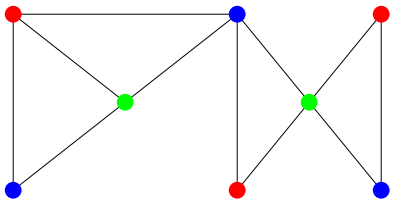
A graph G on 8 vertices



A coloring with 6 colors



A coloring with 8 colors



A coloring with (optimal) 3 colors

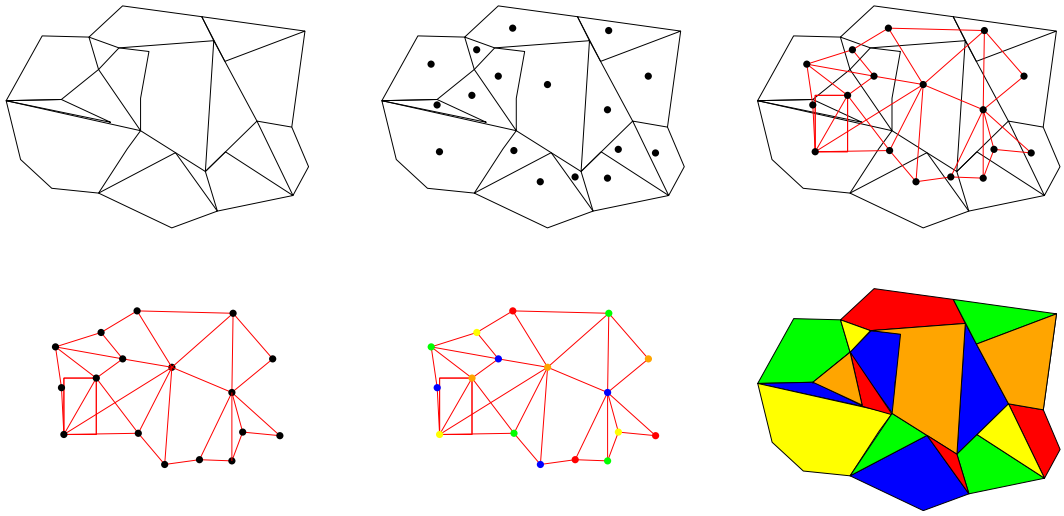
Clearly, any graph can be colored with $|V|$ colors, the goal in coloring is to use as few colors as possible, while satisfying the constraint. The Minimum number of colors needed to color a graph is called the ***chromatic number*** of G and this number is denoted by $\chi(G)$.

The k-Colring(G) problem is defined as follows.

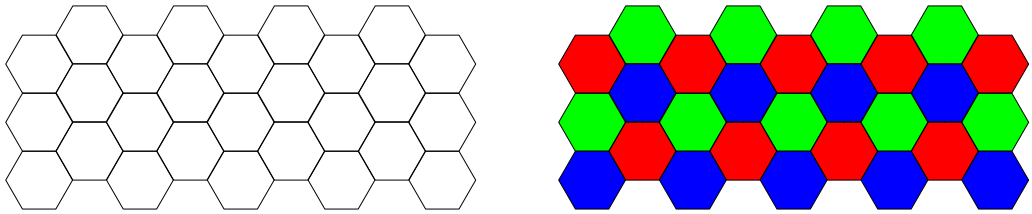
PROBLEM 11 (k-Coloring(G) problem:)**.** *Is there a coloring of G with k colors?*

2.10.1 Applications of Graph Coloring

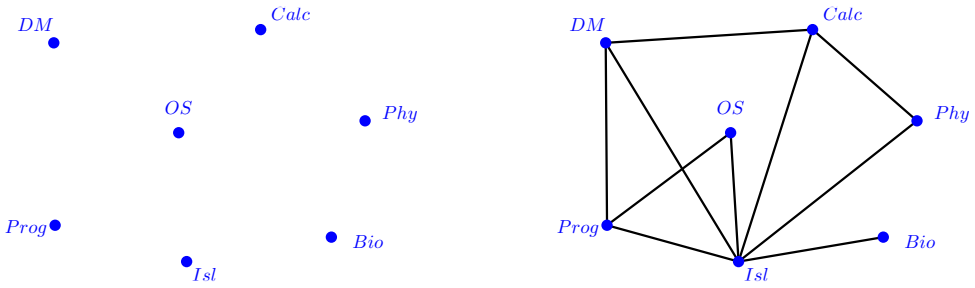
- **Map Coloring:** Given map of a certain area or country, we would like to color its region such that no two neighboring regions (countries, cities, towns, electoral constituencies) get the same color. Otherwise those region will not be distinguishable. This is accomplished by constructing a graph with vertices corresponding to regions and two vertices have an edge between them if the corresponding regions share a boundary. This graph is called the **the Dual Graph of the region**. A k-coloring of this graph yields a coloring of map satisfying the given constraint.



- **GSM Frequency Bands Assignment** In cellular networks (GSM) coverage area is divided into a hexagonal grid, each cell (a hexagon) is served by an antenna. Each cell uses a frequency band (one of 850, 900, 1800, 1900 MHz). The requirement is that frequency of a cell must be different from adjacent cells (hexagons sharing a line segment). We can achieve this by four coloring the vertices of the dual graph of the hexagonal grid.



- **Final Exam Scheduling:** Suppose we want to schedule exams for n courses. The obvious requirement is that no student should have two exams at the same time-slot. This correspond to scheduling exams for the courses that have common students in different time-slots. We want to determine how many time-slots are needed? This can be achieved by considering each course a vertex in a graph. In this graph two vertices are adjacent if the corresponding courses have a common student. Then we want to find the chromatic number of this graph or determine what is the minimum number of colors needed to color this graph.



2.11 Knapsack and Subset Sum Problem

We studied both of these problems in the class. In the following we describe a version of the problem that is consistent with all the above problems described above, i.e. a version of the problem, where the answer is **Yes** or **No**. In these problems we are given a set $U = \{a_1, a_2, \ldots, a_n\}$ of integers, a weight function $w : U \to \mathbb{Z}^+$, a value function $v : U \to \mathbb{R}^+$, and a positive integer C.

The KNAPSACK(U, w, v, C, k) problem is defined as follows.

PROBLEM 12 (KNAPSACK(U, w, v, C, k) problem:)**.** *Is there a subset $S \subset U$ such that $\sum_{a_i \in S} w_i \leq C$ and $\sum_{a_i \in S} v_i = k$.*

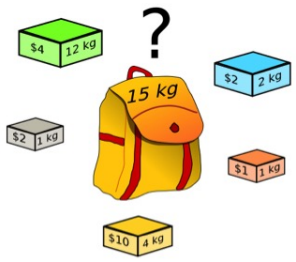
In the subset problem, which is a special case of the knapsack problem, we do not have a distinct value function (or value of each item is equal to its weight).

The SUBSET-SUM(U, w, C) problem is defined as follows.

PROBLEM 13 (SUBSET-SUM(U, w, C) problem:)**.** *Is there a subset $S \subset U$ such that $\sum_{a_i \in S} w_i = C$*

2.11.1 Applications of Knapsack and Subset Sum Problems

- Suppose we have borrowed a server that has capacity C MFLOPS (Mega Floating Point Operations Per Second). We have n processes (jobs) such that job i requires w_i MFLOPS. The question of whether there is a subset of jobs that can feasibly (total MFLOPS requirement at most C) be scheduled to run in parallel on the server and consumes at least k MFLOPS, is just the SUBSET-SUM(U, w, C, k), where U is the set of jobs.
- In many logistic or freights transportation problems we would like to allocate space in a fixed capacity container to items with certain volumes and values (e.g. rents). Such problems are modeled by the KNAPSACK(U, w, v, C, k) problem.



2.12 Partition Problem

Given a set of n positive integers $U = \{a_1, a_2, \ldots, a_n\}$. We would like to partition U into two subsets U_1 and U_2, such that the sum of values in each subset is equal (balanced partition). In other words, $\sum_{a \in U_1} a = \sum_{a \in U_2} a$. This problem is also called Number Partition Problem.

PROBLEM 14 (PARTITION(U, k) problem:)**.** *Is there a biparition of U into U_1 and U_2 such that $|\sum_{a \in U_1} a - \sum_{a \in U_2} a| = k$.*

2.12.1 Applications of the Partition Problem

This problem has application in minimizing VLSI circuit size and delay, assigning tasks to 2 machines such that the finishing time (the larger of the two) is as small as possible (or is equal). You might recall from your childhood, the method of choosing teams was actually an algorithm for this problem. In that method two captains (often self-appointed) will take rounds and alternatively pick a player in each round until all players are assigned to one of the sides. This method actually is a greedy algorithm to solve the bipartition or balanced bipartition problem.

2.13 Number Theoretic Problems: Prime, Composite and Factoring

In this subsection, I just list the problems, which are quite easy to understand and have many applications in public key cryptography.

PROBLEM 15 (PRIME(n))**.** *Is the given number n a prime?*

PROBLEM 16 (COMPOSITE(n))**.** *Is the given number n a composite number?*

PROBLEM 17 (FACTOR(n, k))**.** *Is there a factor d of n such that $2 \leq d \leq k$*

2.14 Circuit Satisfiability Problem

A combinatorial circuit is a generalization of logic gates, which takes n Boolean inputs and produces a single Boolean output. It is implemented with basic logic gates **AND**, **OR**, and **NOT**. We say that a combinatorial circuit is satisfiable if there is an input combination (an assignment of 0 and 1's to it's input) on which the circuit outputs 1.

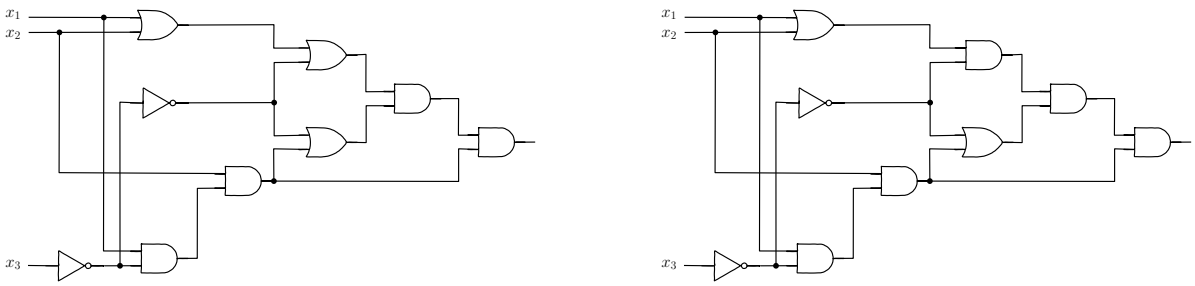
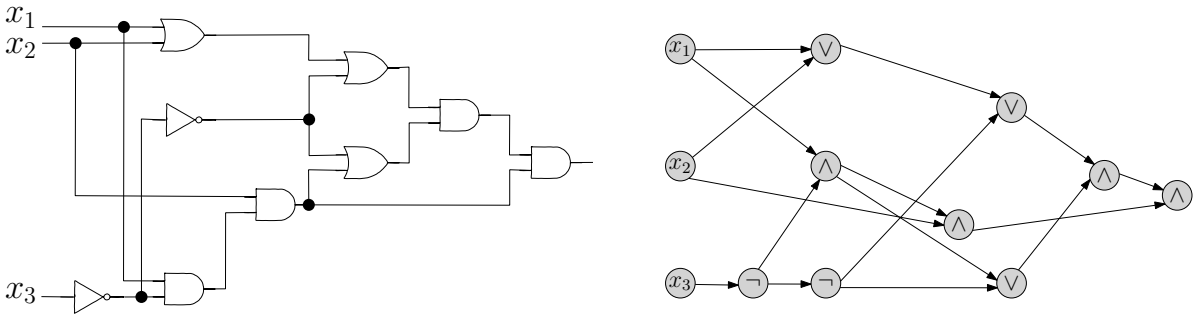


Figure 2: Two instances of the CIRCUIT-SAT problem. The circuit on the left is satisfiable with the assignment $(x_1, x_2, x_3) = (1, 1, 0)$, while the circuit on the right is not satisfiable. Figure adapted from CLRS Figure 34.8

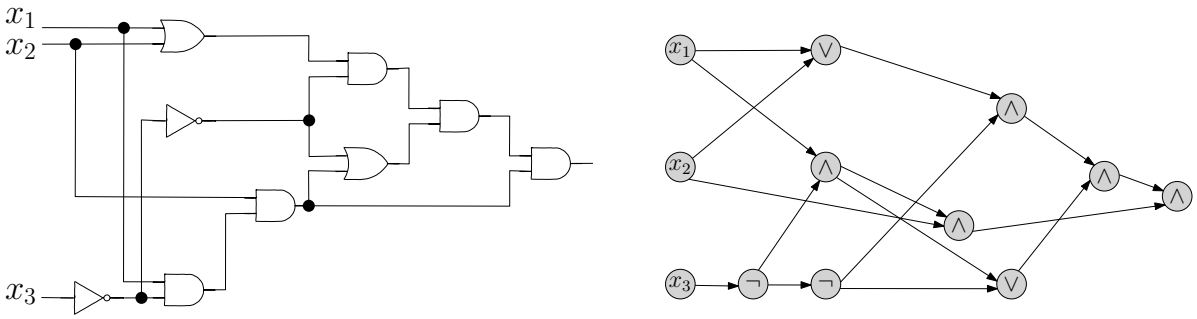
The circuit is encoded as a Directed Acyclic Graph (DAG), where nodes correspond to gates, the output wire and the input wires.

- AND gates and OR gates have indegree 2
- NOT gates have indegree 1
- Known input writes (constant inputs) have no incoming edges and are labeled with input values
- Unknown input writes have no incoming edges and are labeled with variable names
- The node corresponding to the output gate is designated as the sink in the DAG

Given an assignment of values to the unknown inputs, we can evaluate the gates of the circuit in topological order, using the rules of Boolean logic (such as $false$ OR $true = true$) until we obtain the value at the output gate. This is the value of the circuit for the particular assignment to the inputs. The DAGs corresponding to the above two circuits are depicted below.



Please verify that the assignment above does make the circuit output 1, by traversing the DAG in topological sorted order.



PROBLEM 18 (CIRCUIT-SAT(C)). *Is the given Boolean Circuit C satisfiable?*

2.14.1 Applications of CIRCUIT-SAT(C)

- **Computer-Aided Circuit (Hardware) Optimization:** If a digital circuit or one of its sub-circuits is not satisfiable (i.e. it never outputs 1), then that (sub)circuit is not doing any thing (it is useless), so we do not need to waste any gates on it and replace it with a constant output.
- **Complexity Theory** This a fundamental construct in complexity theory, it turns out that a huge number of problems (including all listed above) can be phrased in terms of CIRCUIT-SAT(C) for appropriately defined circuit C, hence it is worthwhile studying and analyzing this problem in detail.

3 Versions of Problems

You should have noticed that all of the above problems are phrased so as the answer to them is either **Yes** or **No**. Some problems like CIRCUIT-SAT(C) and SAT(f) are

inherently of this nature, others are artificially turned into this form. For instance we are interesting in finding a Hamiltonian cycle or computing the maximum independent set in a graph, yet we turned them into **Yes/No** problems. The reasons for this will soon be clear, once we study that the same problem can be phrased in different ways (called versions of the problem). We will also see that this transformation does not result in any loss and that different versions are essentially equivalent. Computational problems basically come in the following three versions.

3.1 Decision Problem

A decision problem is characterized by an algorithm, which returns **Yes** or **No** based on certain criteria being fulfilled by the given instance of the problem or not.

For example the INDEPENDENT-SET(G, k) problem described above is the decision version of the broader independent set problem. If there exists an independent set of size k, the algorithm will answer **Yes** otherwise the output will be **No**.

Similarly SAT(f) is inherently a decision problem, if the given formula is satisfiable the output will be **Yes** otherwise **No**.

All the problems we described above are decision problems (or decision versions of the problem).

Note that the algorithm is not supposed to compute a solution and it is not restricted on how it computes the binary answer.

3.2 Search Problem

A search problem or (the search version of a problem) are characterized by an algorithm that given an instance of the problem returns a structure satisfying certain property (ies) or it returns the NOT-FOUND flag in case there is no structure with the required property (ies).

For example, Search versions of SAT(f) and 3-SAT(f) ask for an assignment to the variables of the formula that satisfies f. Note that the output is a n-bit strings (specifying values for variables (ordered)). In case there is no satisfying assignment the algorithm is supposed to ouput NOT-FOUND = NF.

Search version of CLIQUE(G, k) asks for a clique in G of size k or NF. Again the answer is a subset of vertices that constitute a clique. Similarly, the output of SET-COVER$(U, \mathcal{S}, k)$ is a subcollection of $\mathcal{S}$ and in case no subcollection of size k exists that covers U, it will output NF.

Please go through all the problems above try to phrase their search versions, argue about the format or structure of the output.

3.3 Optimization Problem

An optimization problem is a bit different from search and decision problem as it is supposed to search for an optimal solution. These problems asks for a structure that satisfies certain property (feasibility) such that no other feasible structure have better "value". In a sense, this is a search problem but it searches for an optimal structure. For example, the optimization version of the CLIQUE(G, k) problem, does not specify the number k and can be stated as follows. Given a graph G, find the largest clique in G. Since we studied many optimization problems in the course so far. There could be in a sense two subverisons of an optimization problem, the $OPT - Val(\cdot)$ and the $Opt - Soln(\cdot)$ versions but this distinction generally is not very important.

Optimization version of CLIQUE(G, k), INDEPENDENT-SET(G, k), VERTEX-COVER$(U, \mathcal{S}, k)$, SET-PACKING$(U, \mathcal{S}, k)$ are naturally termed as MAX-CLIQUE(G), MAX-INDEPENDENT-SET(G), MIN-VERTEX-COVER$(U, \mathcal{S})$, MAX-SET-PACKING$(U, \mathcal{S})$. Notice the argument k is dropped. For other problems you should be able to phrase the optimization versions where possible.

In some cases the optimization version does not make much sense. For instance for the SAT(f) problem, one can define it's optimization as that of finding an assignment, which satisfies the maximum number of clauses in f. But the formula may still not be satisfiable.

One final point, for some problem the optimization version just does not make any sense, for instance the Hamiltonian Cycle problem. So is the 3-Coloring problem. Please spend some time on it, to make it clear.

In summary,

- Decision Problem: answer is **Yes/No**
- Search Problem: answer is a feasible structure of certain value or the flag NF.
- Optimization Problem: answer is a feasible structure of optimal "value" (where value is defined depending on the problem, e.g. for vertex cover it is the cardinality of the set, for TSP tour it is the sum of weights of edges in the tour).

Remark: Many authors only use decision problems and search problems. Where search problem there actually means the optimization problem. This is perhaps a better notion, as if you know value of the optimal solution (which is quite easy to find through decision

version of a problem), then one can use search problem (our notion) with the input value equal to the optimal value.

4 Polynomial Time Reduction

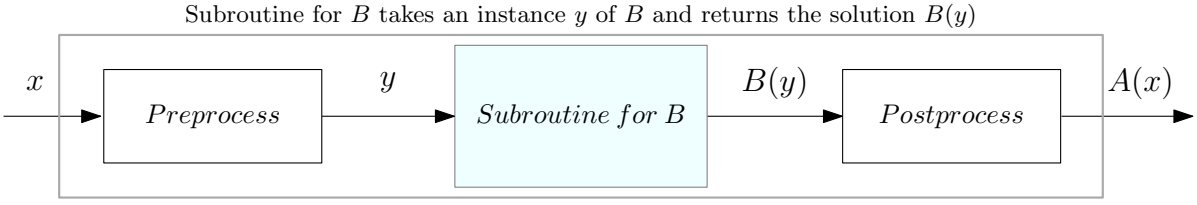
In an attempt to explore the class of computational hard problems, we first define a notion of comparing the hardness of two problems; the relative difficulty of a pair of problems. We would like to formally express "Problem A is at most as hard as problem B". This is formalized through the notion of reduction: we will show that problem A is at most as hard as another problem B by arguing that, if we had a "black box" capable of solving B, then we could solve A.

Suppose we had a black box for solving problem B; Given any instance of B, the black box will return the correct answer. If any instance of problem A can be solved using a polynomial amount of computation plus a polynomial number of calls to this solution of problem B, then we say that A is ploynomial time reducible to B. This is denoted by $A \leq_p B$.

We say that problem A reduces to problem B, if any subroutine (think of it as $C++$ function) for problem B can also be used (called (one or more times) with well thought of legal inputs) to solve any instance of problem A. The following diagram show it schematically,

Definition 10 ($A \leq_p B$)**.** *Problem A is polynomial time reducible to Problem B, $A \leq_p B$, if any instance of problem A can be solved using a polynomial amount of computation plus a polynomial number of calls to a solution of problem B*

In other words, $A \leq_p B$, If a subroutine (e.g a C++ function) for problem B can be used (called once or more times with clever legal inputs) to solve any instance of problem A.



Algorithm for A transforms an instance x of A to an instance y of B. Then transforms $B(y)$ to $A(x)$

4.1 Polynomial Time Reduction to Design Algorithm

An important consequence is that suppose $A \leq_p B$. If B is polynomial time solvable, then A can be solved in polynomial time. In practice we use the contrapositive of this statement, because we do not actually know whether the problem we are studying can be solved in polynomial time or not, we will be using $\leq_p$ to establish relative levels of difficulty among problems.

For now let us see what reducibility among easy problems that we have already studied in the course. We have actually used some of these reduction in the course. Please think about it and convince yourself that these are polynomial time reductions.

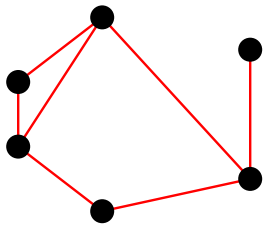
- FindMin $\leq_p$ Sorting
- Sorting $\leq_p$ FindMin
- Median $\leq_p$ Sorting
- Cycle Detection $\leq_p$ DFS
- All Pairs Shortest Paths $\leq_p$ Single Source Shortest Paths
- Single Source Shortest Paths $\leq_p$ All Pairs Shortest Paths

Next we see some polynomial time reductions among the "hard" problems. For now the purpose is establishing the positive results, i.e. using reduction as an algorithm design paradigm. The idea is to get you familiar with the idea of polynomial time reduction. Later we will see more polynomial time reduction and your problem set also contains exercise related to it.

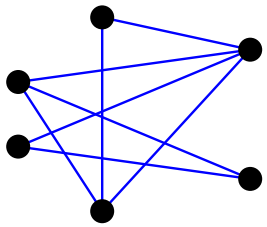
4.2 Reduction by (Complementary) Equivalence

Poly-time reduction of Clique to Independent Set Problem

Recall that for $G = (V, E)$, the complement of G is the graph $\overline{G} = (V, \overline{E})$, where $\overline{E} = \{(u, v) : (u, v) \notin E\}$. We first establish the so-called 'complementary equivalence' between cliques and independent set.



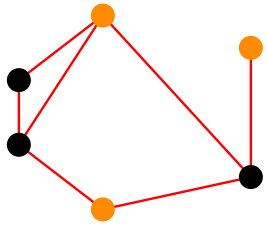
A graph G



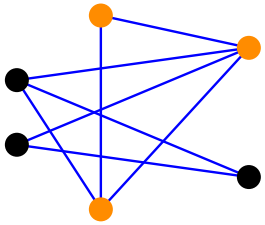
$\overline{G}$

It is not very hard to argue that

Theorem 11. *G has an independent set of size k if and only if $\overline{G}$ has a clique of size k*



An independent set of size 3



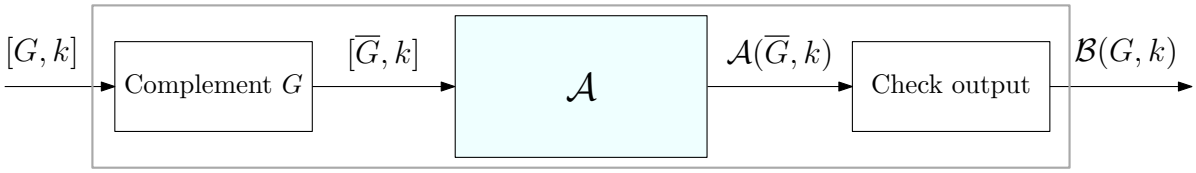
The same 3 vertices make a clique in $\overline{G}$

Using this theorem we prove that

Theorem 12. $\textsc{Clique}(G,k) \leq_p \textsc{Independent-Set}(G,k)$

Proof. Let $\mathcal{A}$ be an algorithm solving $\textsc{Independent-Set}(G,k)$ problem for any G and $k \in \mathbb{Z}$. Let $[G,k]$ be an instance of the $\textsc{Clique}(G,k)$ problem. Compute the complement $\overline{G}$ of G. Call algorithm $\mathcal{A}$ on $[\overline{G},k]$. If it outputs **YES**, output **YES** for the problem $\textsc{Clique}(G,k)$. Else output **NO**.

Algorithm $\mathcal{B}$ takes an instance $[G,k]$ of CLIQUE returns **Yes** if G has a clique of size k else returns **No**



Algorithm $\mathcal{B}$ solves CLIQUE(G,K) problem using a solution $\mathcal{A}$ for INDEPENDENT-SET problem

The schematic representation of the reduction algorithm is provided in the picture. This is clearly a polynomial time reduction, since the preprocessing can be done in linear time (by visiting every pair of vertices or cell in the adjacency matrix). □

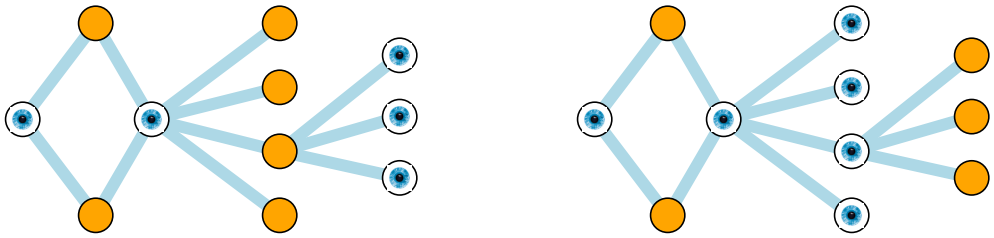
Poly-time reduction of Independent Set to Vertex Cover Problem

Here we first establish he dual relationship between independent set and vertex covers in graphs.

Theorem 13. *Let $G = (V,E)$ be a graph. Then $S \subset V$ is an independent set if and only if its complement $V \setminus S$ is a vertex cover.*

Proof. First, suppose that S is an independent set. Consider an edge $e = (u, v)$. Since S is an independent set, both u and v cannot be in S, at least one of them must be in $V \setminus S$. Thus every edge has at least one endpoint in $V \setminus S$ implying that $V \setminus S$ is a vertex cover.

Conversely, suppose that C is a vertex cover. Consider any two vertices u and v in $V \setminus C$. If they were joined by an edge e, then neither endpoint of e would lie in C, contradicting our assumption that C is a vertex cover. It follows that no two vertices in $V \setminus C$ are joined by an edge, hence $V \setminus C$ is an independent set.

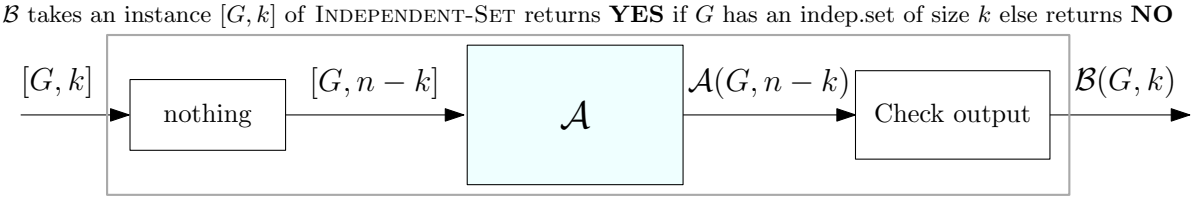


□

Next we use this duality between them to give a polynomial time reduction from INDEPENDENT SET(G, k) problem to the VERTEX COVER(G, k) problem.

Theorem 14. INDEPENDENT SET$(G, k) \leq_p$ VERTEX COVER(G, k)

Proof. Let $\mathcal{A}$ be an algorithm solving VERTEX-COVER(G, k) for any G and $k \in \mathbb{Z}$. Let $[G, t]$ be an instance of the INDEPENDENT SET problem. We call $\mathcal{A}$ on the instance (i.e. with input) $[G, n-t]$. If it outputs **Yes**, we also output **Yes** for INDEPENDENT-SET(G, t). If it outputs **No** we also output **No**.



Algorithm $\mathcal{B}$ solves INDEPENDENT-SET(G,K) problem using solution, $\mathcal{A}$ for VERTEX-COVER problem

□

Note that given this duality relation we can also prove that VERTEX COVER$(G, k) \leq_p$ INDEPENDENT SET(G, k).

4.3 Reduction from Special Case to General Case

Reduction from Vertex Cover to Set Cover

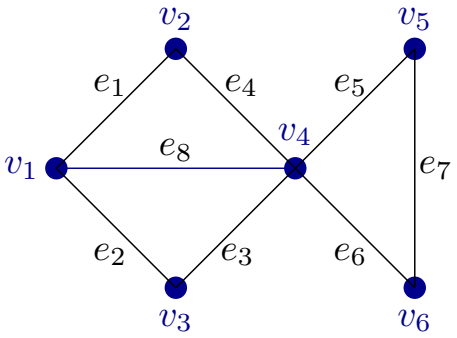
Theorem 15. VERTEX-COVER$(G, k) \leq_p$ SET-COVER$(U, \mathcal{S}, k)$

Proof. Let $\mathcal{A}$ be an algorithm solving SET-COVER$(U, \mathcal{S}, k)$. Suppose we have an instance $[G, k]$ VERTEX-COVER problem, let $G = (V, E)$ with $|V| = n$ and $|E| = m$. We will transform this instance to an instance of SET-COVER and use the algorithm $\mathcal{A}$ to solve it.

Make $U = E$ and construct the collection of subsets of U as follows:

$$\mathcal{S} = \{S_1, \ldots, S_n\}, \text{ where } S_i = \{e \in E \mid e \text{ is incident on } v_i\}$$

See the following figure for an example.



$$U = \{e_1, e_2, e_2, e_4, e_5, e_6, e_7, e_8\}$$

$$S_1 = \{e_1, e_2, e_8\}$$
$$S_3 = \{e_2, e_3\}$$
$$S_2 = \{e_1, e_4\}$$
$$S_4 = \{e_3, e_4, e_5, e_6, e_8\}$$
$$S_5 = \{e_5, e_7\}$$
$$S_6 = \{e_6, e_7\}$$

We argue the following property of the constructed SET-COVER instance.

Lemma 16. *$[U, \mathcal{S}]$ has a set cover of size k if and only if G has a vertex cover of size k*

Proof. if part: Suppose there is a vertex cover V' of size at most k. By definition of vertex cover for each $e = (x, y) \in E$, either $x \in V'$ or $y \in V'$. The collection of subsets $S_{i'}, (v_{i'} \in V')$ is a set cover of the instance $(U, \{S_1, \ldots, S_n\}$.

only if part: Suppose U can be covered with at most k sets in $S_1, \ldots, S_n$, then G has a vertex cover of size at most k. Let the set cover be $\{S_{i_1}, ..., S_{i_k}\} \subset \mathcal{S}$, then every edge in G is incident to one of the vertices $v_{i_1}, \ldots, v_{i_k}$ and so the set $v_{i_1}, \ldots, v_{i_k}$ is a vertex cover in G of size k. □

Using the lemma, we process the output of $\mathcal{A}$ as follows. If $\mathcal{A}(U, \mathcal{S}, k') =$ **YES**, then output **YES**, else output **NO**. □

Reduction from Independent Set to Set Packing

Theorem 17. INDEPENDENT SET$(G, k) \leq_p$ SET-PACKING$(U, \mathcal{S}, k)$

Proof. Given an instance of Independent Set, you should be able to create an instance of Set Packing (this is exactly the same construction as above for Set Cover). □

Another example of reduction from special case to general case is the following.

Theorem 18. 3-SAT$(f) \leq_p$ SAT(f)

Proof. 3-SAT is in Any 3-CNF formula is a CNF formula, so a solution for SAT(f) can trivially be used to solve an instance of 3-SAT(f). □

4.4 Reduction via encoding with "Gadgets"

The more challenging way of reduction and more versatile way is to reduction through encoding a problem instance with cleverly defined gadgets.

Theorem 19. SAT$(f) \leq_p$ 3-SAT(f')

Proof. Given an instance f of the SAT$(\cdot)$ problem, we will construct a 3-CNF formula f' as an instance of the 3-SAT$(\cdot)$ problem and show that f' is equisatisfiable with f. Since f may have longer clauses (containing more than 3 literals), we proceed clause by clause and make a collection of 3-CNF clauses that are together equisatifiable with the given clause.

Lemma 20. $C = (x_{i1} \vee x_{i2} \vee \underbrace{x_{i3} \vee x_{i4} \vee \ldots}_{y})$ *and* $C' = (x_{i1} \vee x_{i2} \vee d_i) \wedge (\overline{d_i} \vee \underbrace{x_{i3} \vee x_{i4} \vee \ldots}_{y})$ *are equisatisfiable.*

Proof. We will show that if there is a satisfying assignment for C, then there is also a satisfying assignment for C' and vice-versa.

1. *If C is satisfiable, then C' is also satisfiable:* Suppose $C = (x_{i1} \vee x_{i2} \vee \underbrace{x_{i3} \vee x_{i4} \vee \ldots}_{y})$ is satisfiable. In a satisfying assignment if $x_{i1} \vee x_{i2} = 1$, then setting $d_i = 0$, we get that both clauses of C' are also satisfiable. On the other hand if in the satisfying assignment for C, $x_{i1} \vee x_{i2} = 0$, hence $y = 1$. Setting $d_i = 1$ we again get that both clauses of C' are satisfiable. Therefore, if there is a satisfying assignment for C, we showed by construction that there is a satisfying assignment for C' too.

2. *If C' is satisfiable, then C is also satisfiable:* Suppose $C' = (x_{i1} \vee x_{i2} \vee d_i) \wedge (\overline{d_i} \vee \underbrace{x_{i3} \vee x_{i4} \vee \ldots}_{y})$ is satisfiable. In a satisfying assignment if $d_i = 1$ ($\overline{d_i} = 0$ and $y = 1$), the same values of $x_2 \ldots$ would satisfy C (in other words y). On the other hand, in the satisfying assignment for C' if $d_i = 0$, then $\overline{d_i} = 1$ and $x_{i1} \vee x_{i2} = 1$, therefore the values of x_{i1} and x_{i2} would satisfy C.

□

The reduction now is straight-forward. Let $X = \{x_1, \ldots, x_n\}$ be the set of variables in the formula f, let $D := \{d_1, d_2, \ldots\}$ be a set of new (dummy) variables that we will introduce. The 3-CNF formula will be on variables $X \cup D$.

Initialize $f' := f$. Let $C_i = (x_{i1} \vee x_{i2} \vee x_{i3} \vee x_{i4} \vee \ldots)$ be a long clause in f. Add two clauses $(x_{i1} \vee x_{i2} \vee d_i) \wedge (\overline{d_i} \vee x_{i3} \vee x_{i4} \vee \ldots)$ to f'. Note that the new (longer) clause is shorter than C_i. Repeat until there is no longer clause in f'.

After the above procedure terminates, we have a 3-CNF formula f', which is equisatisfiable as f. This is so because f is satisfiable if every clause of f is satisfiable, and by the Lemma each clause of f is satisfiable if and only if the clauses in f' constructed for the corresponding clause of f are satisfiable.

Now suppose $\mathcal{A}$ is an algorithm that decides the satisfiability of any 3-CNF formula. Then given an instance f of SAT(f), we transform it into 3-CNF formula f' and call $\mathcal{A}$ on f'. If $\mathcal{A}(f')$ returns **Yes** we also return **Yes**, else we return **No**.

Suppose f has n variables and m clauses. By construction f' has at most $O(m \times n)$ variables and clauses and the transformation takes times $O(m \times n)$. Hence this is a polynomial time reduction. □

The next is an important reduction, it is a bit tricky but not very difficult. It uses a very clever idea of reducing a problem about Boolean formula to a problem in graph theory.

Theorem 21. 3-SAT(f) $\leq_p$ INDEPENDENT-SET(G, k)

Proof. Suppose there is an algorithm $\mathcal{A}$ that takes as input a graph G and an integer k and decides whether or not G has an independent set of size k.

In the 3-SAT(f) instance we have a formula f and we want to decide whether f is satisfiable. Looking at f we will construct a graph G such that G has an independent set of size k if and only if f is satisfiable. Then we will input this G to $\mathcal{A}$ and return the answer.

Suppose f has n variables $(x_1, x_2, \ldots, x_n)$ and $m-$clauses (so $3m$ literals). In order to satisfy the formula, we need to set each of $x_1, x_2, \ldots, x_n$ to 0 or 1 so as f evaluates to 1. Alternatively, we need to pick a literal from each clause and set it to 1. But we cannot make conflicting setting, i.e. we cannot set a literal value to 1 for a clause and for another clause we set the same literal to 0.

Keeping this in mind we construct m triangles with vertices of ith triangle corresponding to literals in the ith clause.

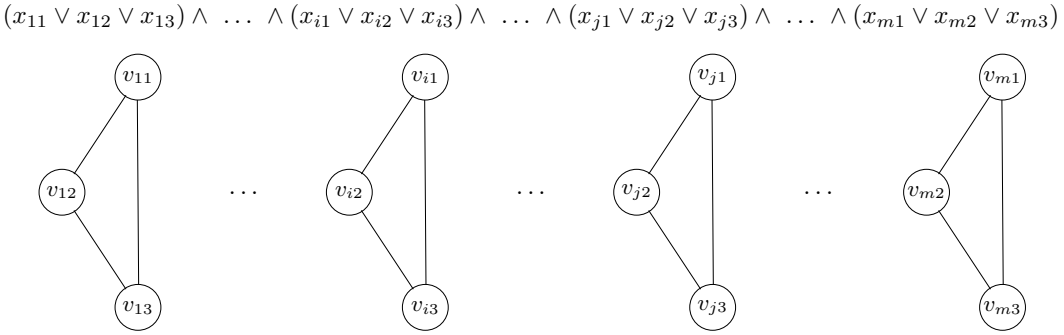


Figure 3: A generic 3-CNF formula with m clauses $(x_{i1} \vee x_{i2}, x_{i3})$

An independent set of size m in this graph (the set of disjoint triangles) would correspond to choosing exactly one literal from each clause (that could be set to 1 to get all clauses satisfied). However, the independent set may contain vertices corresponding to two literals that are negation of each other. To avoid that we add an edge between any two vertices (in different triangles) that correspond to two literals that are negation of each other. That is vertices where the corresponding literals represent the same variable in negation. The following diagram depicts that.

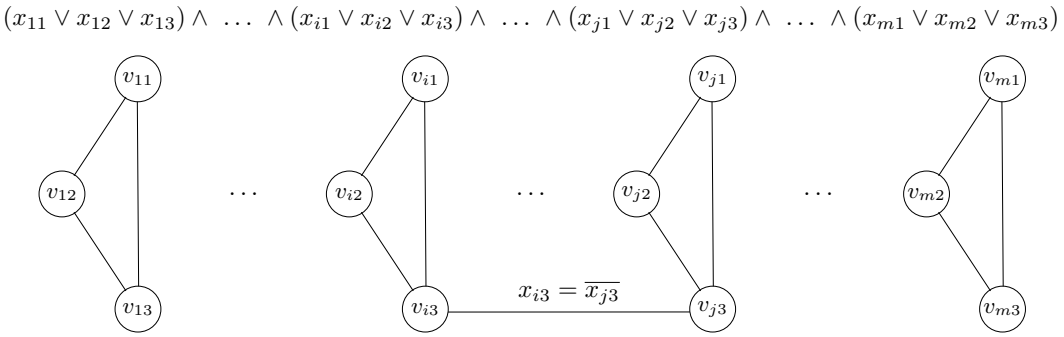


Figure 4: A generic 3-CNF formula with m clauses and the constructed graph on $3m$ vertices

Following is a concrete example of this gadget for a satisfiable formula

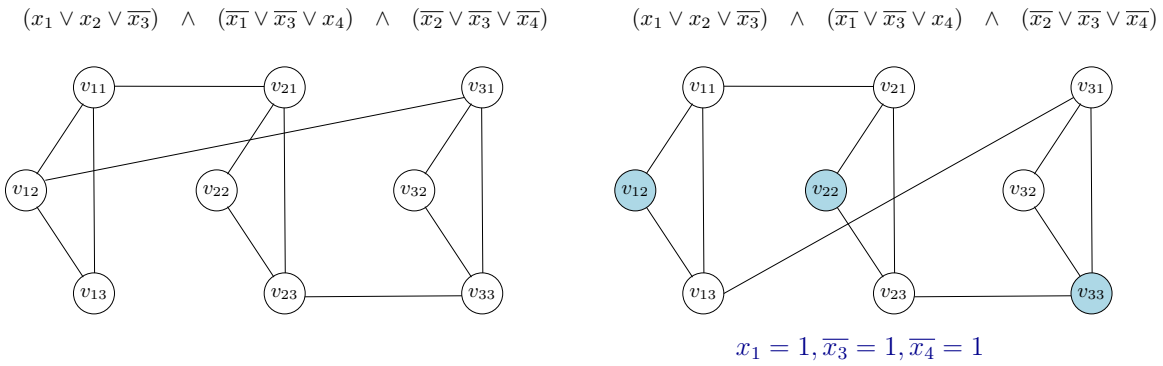
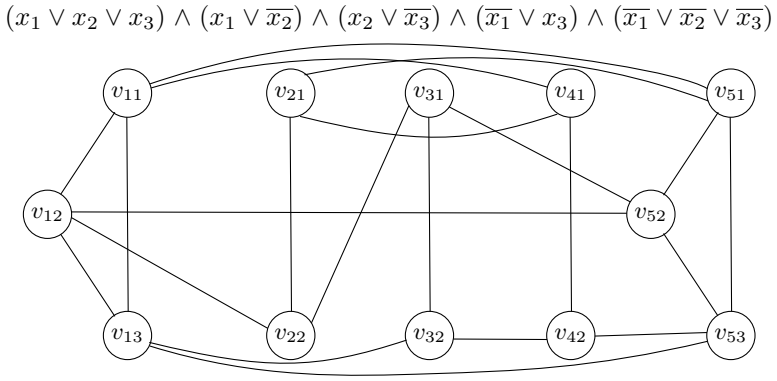


Figure 5: A 3-CNF formula and the constructed instance of INDEPENDENT-SET(G, k) problem. (b) An independent set of size 3 correspond to a satisfying assignment to the formula.

The diagram below gives a non-satisfiable formula and the corresponding graph with no indepenent set.



No satisfying assignmnet, No independent set of size 5

Lemma 22. *Let $G(f)$ be the graph constructed by the above procedure for a* 3-CNF *formula f. $G(f)$ has an independent set of size m if and only if f is satisfiable.*

Proof. First, if the f is satisfiable, then clause of f has at least one literal that is true in the satisfying assignment. Correspondingly, each triangle in $G(f)$ contains at least one node whose label evaluates to 1. Let S be a set consisting of one such node from each triangle. We claim that S is independent; for if there were an edge between two nodes

$u, v \in S$, then the labels of u and v would have to conflict(i.e. $u = \overline{v}$); but this is not possible, since they both evaluate to 1.

Conversely, suppose $G(f)$ has an independent set S of size at least m. Then the size of S is exactly m, and it must consist of exactly one node from each triangle. Now, we claim that there is a truth assignment $\mathbf{v}$ for the variables in f with the property that the labels of all nodes in S evaluate to 1. Here is how we could construct such an assignment $\mathbf{v}$. For each variable x_i, if neither x_i nor $\overline{x_i}$ appears as a label of a node in S, then we arbitrarily set $\mathbf{v}(x_i) = 1$. Otherwise, exactly one of x_i or $\overline{x_i}$ appears as a label of a node in S; for if one node in S were labeled x_i and another were labeled $\overline{x_i}$, then there would be an edge between these two nodes, contradicting our assumption that S is an independent set. Thus, if x_i appears as a label of a node in S, we set $\mathbf{v}(x_i) = 1$ and otherwise set $\mathbf{v}(x_i) = 0$. By constructing $\mathbf{v}$ in this way, all labels of nodes in S will evaluate to 1.

There is one side issue, the formula f also has clauses with one or two literals, convince yourself that the same construction works, but instead of triangles some clauses will be represented by edges and vertices. □

The polynomial time reduction: Given a formula f, we construct the graph $G(f)$ **in polynomial** time and declare f as satisfiable if and only $\mathcal{A}(G(f))$ outputs **Yes**. □

Reduction from Hamiltonian Cycle to Hamiltonian Path

Next we give the final reduction for this section, we will discuss more reductions later. All the reductions we discussed so far are sometimes referred to as the **"Karp Reducibility"**. Named after the great theoretical computer scientist Richard Karp. We will also introduce the notion of the so-called **"Cook Reducibility"** named after another great Stephen Cook. We will here more about their relevant work in complexity theory and why they deserve to get something important named after them.

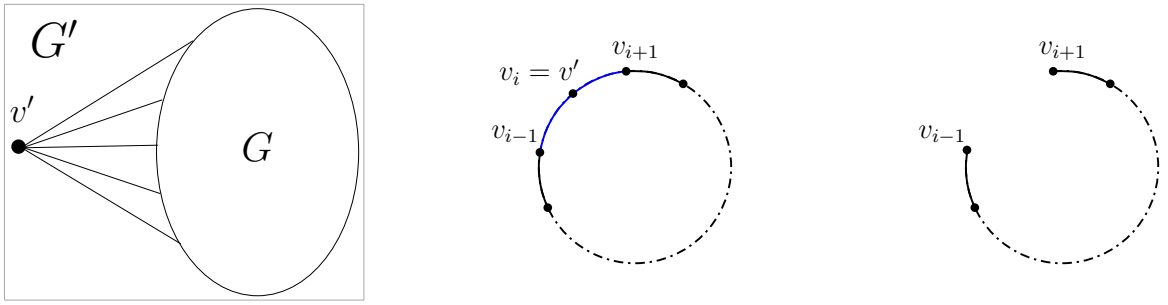
The following theorem uses the standard reduction (Karp reduction).

Theorem 23. Hamiltonian-Path(G) $\leq_p$ Hamiltonian-Cycle(G)

Proof. Let $\mathcal{A}$ be an algorithm for the Hamiltonian-Cycle(G') problem, that takes a graph G' and return **Yes** if and only if G' has a Hamiltonian cycle. We will use $\mathcal{A}$ to solve the Hamiltonian-Path(G) problem that asks for deciding whether G has a Hamiltonain path.

Given an instance $G = (V, E)$ of Hamiltonian-Path(G), we construct an instance $G' = (V', E')$ of the Hamiltonian-Cycle(G') problem as follows. $V' = V \cup \{v'\}$, where v' is dummy new vertex. E' is all edges in E and the new vertex v' is adjacent

to all other vertices in G (that is the original V). See the following figure (a). We first establish the following lemma.



(a) (b) (c)

Lemma 24. *G' has a Hamiltonian Cycle if and only if G has a Hamiltonian path.*

Proof. Suppose G' has a Hamiltonian cycle, then it contains the (dummy) vertex v' at (say at the ith index in some ordering). Removing v' from the cycle, we get a path $v_{i+1}, v_{i+2}, \ldots, v_{i-1}$. This path is a Hamiltonian path in G, as it uses all vertices of V and edges from E.

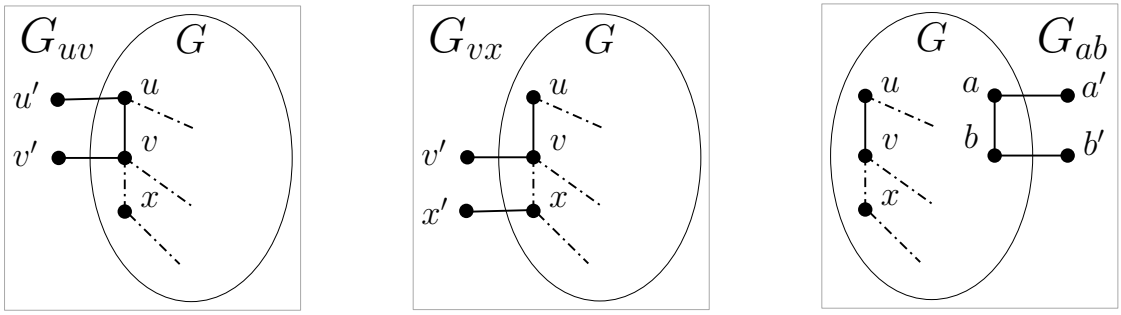
To see the other side, suppose there is a Hamiltonian path $v_1, \ldots, v_n$ in G. Then in G' since v' is adjacent to both v_1 and v_n, inserting v' gives a Hamiltonian cycle in G'. See the figure (b) and (c). □

The reduction is now straightforward and its correctness follows from the lemma. We call $\mathcal{A}$ on G' and if $\mathcal{A}$ outputs **YES** we output **YES** and vice-versa □

Theorem 25. Hamiltonian-Cycle$(G) \leq_p$ Hamiltonian-Path(G)

Proof. For this we use the so-called Cook reduction. So far in all the reduction, we made a single call to the supposed solution. Since we are allowed to do polynomial amount of extra work, we can make polynomial number of calls to the supposed algorithm. The overall reduction will still be a polynomial time reduction if the supposed algorithm takes polynomial amount of time. Following is how the reduction work.

Let $\mathcal{A}$ be a polynomial time algorithm for the Hamiltonian-Path(G') problem. Given an instance $G = (V, E)$ of the Hamiltonian-Cycle(G) problem. For each $(u, v) \in E(G)$, we make a new graph $G_{uv} = (V_{uv}, E_{uv})$. For G_{uv}, $V_{uv} = V \cup \{u', v'\}$ (two dummy vertices added) and $E_{uv} = E \cup \{(u, u'), (v, v')\}$ (two extra edges added).



Lemma 26. *G has a Hamiltonian cycle if and only if some G_{uv} has a Hamiltonian path*

Proof. Suppose G has a Hamiltonian cycle C. Pick any two consecutive vertices on C say a and b ((a, b) is an edge in G). In G_{ab} the edge (a', a), the (longer) part of C from a to b and the edge (b, b') is a Hamiltonian path.

To see the other side suppose the G_{ab} has a Hamiltonian path. This Hamiltonian path must start at the vertex a' and end at the vertex b' (as this is the unique Hamiltonian path possible if at all). Now by construction (a, b) is an edge in G. Hence using the edge (a, b) with the rest of the path except for the (a', a) and (b', b) edge gives us a Hamiltonian cycle in G. See the figure. □

We call the algorithm $\mathcal{A}$ on each of G_{uv}, if $\mathcal{A}$ outputs **Yes** on any G_{uv} we will output **Yes**, otherwise if on all G_{uv}'s $\mathcal{A}$ returns **No**, we return **No**. Constructing each G_{uv} takes $O(|V| + |E|)$ time (we might have to traverse the adjacency list of u and v etc.) And we have to construct $|E|$ such graphs, hence total time in this preprocessing step is $O(|E|^2)$. Constructing the cycle from the returned path takes $O(|V|)$ time (the post processing time). Hence in polynomial time extra work we can solve the Hamiltonian Cycle problem using a solution to the Hamiltonian Path problem. □

5 Transitivity of Reductions

In the previous section, we used the following techniques to reduce one problem to another.

- **Simple or Complementary Equivalence or Duality** We used Complementary equivalence to prove that Clique$(G, k) \leq_p$ Independent-Set(G, k) and used duality to prove Independent-Set$(G, k) \leq_p$ Vertex-Cover(G, k'). For easier problems, recall from the bipartite matching, we established a duality between the maximum bipartite matching and minimum vertex cover (both optimization version), that readily gives us a polynomial time reduction. Similarly, while studying

network flow we established duality between maximum s-t flow and minimum s-t cut.

- **Special Case to General Case** We used it to prove VERTEX-COVER$(G, k) \leq_p$ SET-COVER$(U, \mathcal{S}, k')$, INDEPENDENT-SET$(G, k) \leq_p$ SET-PACKING$(U, \mathcal{S}, k')$, 3-SAT$(f') \leq_p$ SAT(f). Earlier we studied the SUBSET SUM(U, v, C) problem was a special case of the KNAPSACK(U, v, w, C) problem. Thus quite easily we can reduce the SUBSET SUM$(U, v, C) \leq_p$ KNAPSACK(U, v, w, C)
- **Encoding with Gadgets** This was the most tricky one. It requires a lot of thoughts and is usually challenging, but with practice one get reduction using this method. We shall see some more examples of this. So far we used it to prove 3-SAT$(f) \leq_p$ INDEPENDENT-SET(G, k), SAT$(f) \leq_p$ 3-SAT(f') and HAMILTONIAN-PATH$(G) \leq_p$ HAMILTONIAN-CYCLE(G).
- **Cook Reducibility** Another method we used is the so-called Cook reducibility. This is not really a stand-alone method as can be used in any of the above method, but nonetheless to prove HAMILTONIAN-CYCLE$(G) \leq_p$ HAMILTONIAN-PATH(G) we used. We will see a few more applications of this shortly.

Another very powerful technique to reduce a problem to another is to exploit the following theorem about transitivity of reductions

Theorem 27 (Transitivity of Reductions)**.** *Given three problems X, Y, and Z,* **if** $X \leq_p Y$ **and** $Y \leq_p Z$**, then** $X \leq_p Z$

We first give a proof for this theorem and then discuss how to use it to derive more reductions.

Proof. Suppose $X \leq_p Y$ and $Y \leq_p Z$. We will prove that this implies that $X \leq_p Z$. In other words, let $\mathcal{A}_Z$ be an algorithm for Z. Given any instance I_X of X, we will solve X on I_X using $\mathcal{A}_Z^+$ (using one or more calls).

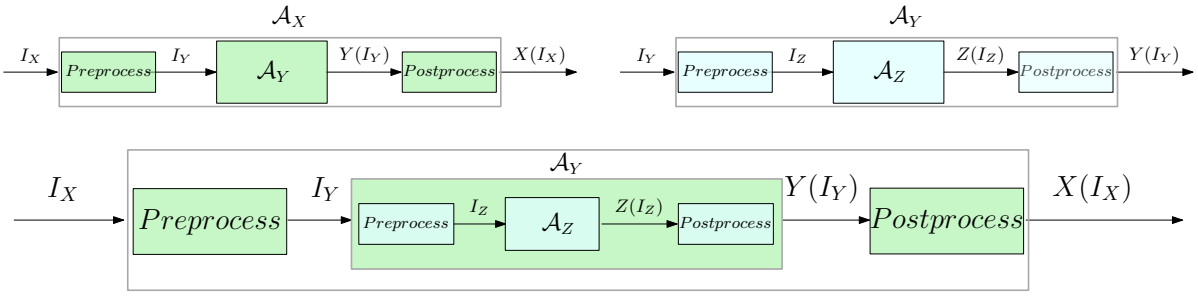
Since $Y \leq_p Z$, there is an algorithm $\mathcal{A}_Y$ for Y using $\mathcal{A}_Z^+$. There may be many other algorithms to solve Y, but we will use the one that is based on the polynomial time reduction implied by $Y \leq_p Z$.

Similarly, there is an algorithm $\mathcal{A}_X$ for X using $\mathcal{A}_Y^+$. Again $X \leq_p Y$ means that any solution for Y can be used to solve X with polynomial time extra work. But $\mathcal{A}_X$ is the one that uses $\mathcal{A}_Y^+$ (which is built upon $\mathcal{A}_Z$).

We design a new algorithm $\mathcal{B}_\mathcal{X}$ for X. This algorithm is just the $\mathcal{A}_X$ it uses the subroutine $\mathcal{A}_Y$ that is built upon $\mathcal{A}_Z$. In essence we essentially compose the two reductions

into one. It is a polynomial time reduction, because of the fact that sum of two polynomials is a polynomial. Hence $\mathcal{B}_{\mathcal{X}}$ uses polynomial amount of extra work (prepcocessing, postprocessing and calls to $\mathcal{A}_Y$) and calls to $\mathcal{A}_Z$ to solve X.

The following figure depicts the above reduction schematically.



□

Transitivity is an extremely useful property of reduction. Let's see what we get from it.

- We proved SAT$(f) \leq_p$ 3-SAT(f') and 3-SAT$(f) \leq_p$ Independent-Set(G, k). We conclude from these two using the theorem that SAT$(f) \leq_p$ Ind-Set(G, k)
- Similarly we proved that

  SAT$(f) \leq_p$ 3-SAT$(f') \leq_p$ Independent-Set$(G, k) \leq_p$ Vertex-Cover$(G, t) \leq_p$ Set-Cover$(U, \mathcal{S}$

- We can safely conclude that SAT$(f) \leq_p$ Set-Cover$(U, \mathcal{S}, l)$
- It is easy to see that even amongst the above mentioned reductions we get many because of transitivity

6 Self Reducibility

We introduced the decision, search and optimization versions of problems and focused on decision versions only. In this section we discuss the questions like did we lose any generality by focusing on decision versions, are all versions of a problem "equivalent"? are versions of a problem reducible to each other?.

An interesting fact that we elaborate on further later in this section is that for many problem their search and optimization versions are only polynomially more difficult than the corresponding decision versions. This is in the sense that any efficient algorithm for the decision problem can be used to solve the search problem efficiently. In other words

the optimization and search versions are polynomial time reducible to their decision versions. This is called ***self-reducibility***.

Next we discuss some examples of self-reducibility and problems that exhibit it. We add the the prefix DEC-, SRCH, or MAX/MIN/OPT to the problem name to identify the decision, search and optimization versions of a problem, respectively. Note that the original name was already decision version but for the sake of clarity we still add the prefix DEC for now. The argument to the problem will change accordingly.

Theorem 28. DEC-INDEPENDENT-SET$(G, k) \leq_p$ MAX-INDEPENDENT-SET(G)

Proof. Suppose $\mathcal{A}$ is an algorithm for MAX-INDEPENDENT-SET(G). That is $\mathcal{A}$ takes as input a graph G and returns a largest independent set in it. We will use $\mathcal{A}$ to solve DEC-INDEPENDENT-SET(G, k).

Given an instance $[G, k]$ of the DEC-IND-SET(G, k) problem, call $\mathcal{A}$ on G. Check if independent set returned by $\mathcal{A}$ is of size $\geq k$ return **Yes** else return **No**. Computing the size of the returned set and comparing it with k can clearly be done in polynomial time. Hence the reduction is a polynomial time reduction. □

Theorem 29. DEC-INDEPENDENT-SET$(G, k) \leq_p$ SRCH-INDEPENDENT-SET(G)

Proof. Suppose $\mathcal{A}$ is an algorithm for SRCH-INDEPENDENT-SET(G, k). That is $\mathcal{A}$ takes as input a graph G and returns an independent set of size k in G. We will use $\mathcal{A}$ to solve the DEC-INDEPENDENT-SET(G, k).

Given an instance $[G, k]$ of the DEC-IND-SET(G, k) problem, call $\mathcal{A}$ on $[G, k]$. Check if $\mathcal{A}$ returns an independent, then return **YES**. In the other case if $\mathcal{A}$ returns **NF**, then return **No**. This clearly is a polynomial time reduction. □

Theorem 30. SRCH-INDEPENDENT-SET$(G, k) \leq_p$ MAX-INDEPENDENT-SET(G)

Proof. Suppose $\mathcal{A}$ is an algorithm for MAX-INDEPENDENT-SET(G). That is $\mathcal{A}$ takes as input a graph G and returns a largest independent set in it. We will use $\mathcal{A}$ to solve SRCH-INDEPENDENT-SET(G, k).

Given an instance $[G, k]$ of the SRCH-IND-SET(G, k) problem, call $\mathcal{A}$ on G. Check if independent set returned by $\mathcal{A}$ is of size $\geq k$, then return the same independent set. On the other hand if the size of the retuned indpendent set is less than k, then return **NF**. Computing the size of the returned set and comparing it with k can clearly be done in polynomial time. Hence the reduction is a polynomial time reduction. □

Theorem 31. SRCH-INDEPENDENT-SET$(G, k) \leq_p$ DEC-INDEPENDENT-SET(G, k)

Proof. This one is generally trickier, since the decision versions of problems gives **Yes/No** answers, while the search and optimization versions are required to output some kind of structure. Generally, we repeatedly call the supposed algorithm for the decision version (Cook reduction) to construct the structure using the successive **Yes/No** answers.

Here is an algorithm for SRCH-INDEPENDENT-SET(G, k) that uses an algo $\mathcal{A}$ for DEC-INDEPENDENT-SET(

Algorithm Algorithm for the SRCH-INDEPENDENT-SET(G, k) problem

```
1: I ← ∅                                          ▷ Initialize an empty independent set
2: t ← k
3: for v ∈ V(G) = {v_1, ..., v_n} do
4:     ans ← A(G \ {v}, t)
5:     if ans = yes then        ▷ Check if this vertex is needed for an independent set
6:         V(G) ← V(G) \ {v}
7:     else
8:         V(G) ← V(G) \ {v}
9:         I ← I ∪ {v}
10:        t ← t − 1
```

□

Theorem 32. MAX-INDEPENDENT-SET$(G) \leq_p$ SRCH-INDEPENDENT-SET(G, k)

Proof. This type of reductions also require a little care. The main difference between the optimization version and the search version of a problem is the size of the solution. The search version in this case for instance requires the parameter k, while for optimization version we do not know the size of largest independent set. We have to find the size of largest independent set too. Suppose $\mathcal{A}$ is an algorithm for SRCH-IND-SET(G, k). If we knew that t is the size of largest independent set, then we could just call on $[G, t]$ and return the independent set.

Therefore, we search for t also. The reduction works as follows. For $1 \leq k \leq t(?)$, call $\mathcal{A}$ on $[G, k]$. When we get to the first k on which $\mathcal{A}$ such that $[G, k]$ returns **NF** we know $t = k - 1$. Now we make one call again to $\mathcal{A}$ on $[G, t]$ and return the independent set. We know it is a maximum independent set, because $\mathcal{A}(G, t + 1) \to$ **NF**.

Note that this reduction critically uses the monotonicity of independent sets, i.e. if there is no independent set of size i, then there cannot be an independent set of sizer bigger than i.

One final point, this linear search might be a problem. This is the problem because k is an input to the SRCH-INDEPENDENT-SET(G, k) problem and the number of calls we made to $\mathcal{A}$ is t, which is value of the input the size of this input is logarithmic in the value. Think about it this way, we know $k \leq n$, for $n = |V(G)|$, and making k calls would be $O(2^{\log n})$ (exponential in the size of input).

Luckily there is an easy fix. We should run a binary search algorithm to find the first k on which the algorithm returns **NF**. i.e. start with $k = 1$ and iteratively double the value of k until we find one for which the answer is **NF**. Now we find the value of t between $k/2$ and k. □

Theorem 33. MAX-INDEPENDENT-SET$(G) \leq_p$ DEC-INDEPENDENT-SET(G, k)

Proof. Finally, this last of the six possible reduction. In this case we first find the size of maximum independent set (t) by doing a binary search as above. Once we know the size of a maximum independent set, say t.

Then we find an independent set of size t in G (which is just the search version). This can be accomplished as in the reduction of SRCH-INDEPENDENT-SET(G, t) to DEC-INDEPENDENT-SET(G, t). □

Next we give a few more examples of reduction from search version to decision version, mainly to emphasize the point we made that such reductions are trickier.

Theorem 34. SRCH-HAMILTONIAN-PATH$(G) \leq_p$ DEC-HAMILTONIAN-PATH(G)

Proof. Let $\mathcal{A}$ be an algorithm for DEC-HAMILTONIAN-PATH(G) Call $\mathcal{A}$ on G, if it returns **No**, then return **NF**. In case it returns **Yes**, then we need to find a Hamiltonian path in G. Suppose we do the following. For each vertex v, Call $\mathcal{A}$ on $G \setminus \{v\}$, what to do if it $\mathcal{A}$ returns **Yes/No** on $G \setminus \{v\}$. We cannot use this call to select or de-select v, like we did for independent set, as all vertices of G have to be in a Hamiltonian path. The correct way is as follows.

For each edge $e = (u, v)$, we call $\mathcal{A}$ on $G \setminus \{e\}$, if $\mathcal{A}$ returns **Yes**, then this means e is not needed for the Hamiltonian path since there is a Hamiltonian path even without e present. We delete the edge e from the remaining part of G and move to the next edge. On the other hand, if $\mathcal{A}$ returns **No**, this means that e is required for the Hamiltonian path. We keep the edge e in (remaining) G and move to next edge. In the end, since the original graph has a Hamiltonian Path (we tested initially), we will be left with $n - 1$ edges that make a Hamiltonian path.

□

Theorem 35. SRCH-VERTEX-COVER$(G, k) \leq_p$ DEC-VERTEX-COVER(G, k)

Proof. Suppose $\mathcal{A}$ is an algorithm for DEC-VERTEX-COVER(G, k). Given an instance $[G, k]$ of the search problem, we call $\mathcal{A}$ on $[G, k]$, if it returns **No**, we return **NF**. But if it returns **Yes**, we have to find a vertex cover of size k.

For each vertex v, we want to determine whether or not v is part of a vertex cover of size k. So we call $\mathcal{A}$ on $[G \setminus \{v\}, k]$, if it returns **Yes** we do not include v in the output, if it returns **No**, we include v in the output.

Careful: Since G has a vertex cover of size k (suppose there is a unique minimum cover called C.), $G \setminus \{v\}$ has a vertex cover of size k whether or not v is in the cover. Because if v is not in C, then the same C covers all edges of G, so it definitely cover all edges of $G \setminus \{v\}$. But if v is in C, then v only covers edges incident to v. When v is gone, the remaining vertices in C still cover all the remaining edges. So we can add some vertex $u \neq v$ to C and it still covers all edges in $G \setminus \{v\}$. Recall vertex cover is a minimization problem. Therefore, e will get **Yes** answer for every v.

The correct idea is to call $\mathcal{A}$ on $[G \setminus \{v\}, k-1]$, if it returns **Yes**, then v is in the k-sized cover. If it returns **No**, then v is not part of any k-sized cover.

□

6.1 Caution for Self Reducibility

CAUTION! Self reducibility **DOES NOT** mean that "any algorithm solving the decision version must use a solution of the search version". A solution to the search version is sufficient to solve the decision version, but it is not necessary. There are problems where we can solve the decision version without yet having a solution to the search version.

Our purpose of introducing the numeric problems above was to give you an example of this.

Recall the COMPOSITE(n) problem that decides whether or not the given integer n is a composite number. Note the problem PRIME(n) is the "complement" problem. Also the search version of the FACTOR(n) that asks for a factor of n, if n is composite, and **NF** if n is prime.

The famous Agarwal, Kayal, Saxena (AKS) (2004) theorem for "Primality Testing" uses involved number theory to solve the PRIME(n) or COMPOSITE(n) problems. But it does not solve the corresponding search version i.e. FACTOR(n) problem.

7 Polynomial-time verification

This point onward our main focus will be decision problems (or decision version of problems) and we will denote them by the names given initially (i.e. without the prefix DEC-).

So far we have been discussing how to solve problems using many different general techniques including greedy algorithms, dynamic programming and now polynomial time reduction. Now we will introduce another aspects of problems that is to certify and verify a given solution. We will discuss how to check (verify) a proposed solution (a certificate) to a problem.

It should be clear that the problem of computing an independent set in a graph or finding a satisfying assignment for a formula is very different than verifying a claim that a given subset is an independent set in this graph or checking the claim that this specific assignment to variables satisfies the given formula.

We will precisely formulate the verification problem but you should have some idea of this. Please keep the following points in mind this will hopefully your understanding of polynomial time verification more clear.

- In some cases both computing a solution and checking a proposed solution are both easy (efficiently solvable in polynomial time). For example computing the minimum spanning tree of a graph can be accomplished in polynomial time using Prim's or Kruskal's algorithm. On the other hand suppose we are given a "subgraph" (accompanied with the claim that this thing is a MST of the graph), how can we verify the claim. We would first check if what is given is indeed a subgraph, a tree, a spanning tree. Then we can compute a MST using Kruskal's algorithm and check it's total weight is equal to the total weight of the claimed spanning tree. All of these tasks can be done in polynomial time. You can similarly argue about Maximum s-t flow in networks, and many other problems.
- In some cases computing a solution is hard, while verifying a claimed solution is easy. For instance as we discussed earlier that we do not know of any efficient algorithm for the 3-SAT problem. But if one claims to have a solution, they can express the solution by just giving the values of each variable. And we can just verify that solution by evaluating each clause and checking if all of them are true. It requires only one scan over the formula (linear in number of variables and number of clauses.) Similarly computing an independent set is hard, but a claimed solution (a subset of vertices) can easily be verified by checking if all vertices in the subset are in the graph and that all pairs are non-edges.
- In some cases both computing the solution and verifying a claimed solution are

hard. Think of the problem $\overline{\text{3-SAT}}(f)$, which is supposed to output **Yes** if f is not satisfiable and **No** if there is a satisfying assignment. Suppose you want to claim that a given formula is a **Yes** instance of the $\overline{\text{3-SAT}}(\cdot)$ problem. How would you express a solution to this problem (how would you certify the claim), and then how would you verify the claim in polynomial time. It seems like the only way to express the solution is essentially encode the following. $\underbrace{000\ldots00}_{n \text{ bits}}$ does not satisfy f, $\underbrace{000\ldots01}_{n \text{ bits}}$ does not satisfy f, $\underbrace{000\ldots10}_{n \text{ bits}}$ does not satisfy f, all the way up to $\underbrace{111\ldots11}_{n \text{ bits}}$ does not satisfy f. This certificate is exponentially long and a verification algorithm will take exponential time in just reading the certificate (claimed solution). Similarly, think about what "evidence" could we show that would convince you, in polynomial time, that a given graph has no independent set of size k.

7.1 Verification algorithms

Definition 36 (Polynomial Time Verifiable Problems)**.** *A decision problem X is efficiently verifiable if*

- *The claim: "$\mathcal{I}$ is a* **Yes** *instance of X" can be made in polynomial bits, i.e. there exists a polynomial sized certificate C for* **Yes** *instances of X*
- *A given certificate can be verified in polynomial time. In other words, there exists a polynomial time algorithm $\mathcal{V}$ that takes the instance $\mathcal{I}$ and the certificate $\mathcal{C}$ such that $\mathcal{V}(\mathcal{I}, \mathcal{C}) =$* **Yes** *if and only $X(\mathcal{I}) =$* **Yes**

It takes some time to comprehend this, examples should make it clear

Theorem 37. $\text{MST}(G, k)$ *is polynomial time verifiable*

Proof. A certificate could be the spanning tree for G that is claimed to be a spanning tree of G with weight at most k. This "claimed MST" can be written in at most $O(n \log n)$ bits (vertices ids and order) and its adjacency matrix $O(n^2)$. Hence there is exists a polynomial sized certificate. A verifier can just check if the given vertices are in G, if all edges are actually from G, and the given subgraph is acyclic and connected. Finally, it can just compute the sum of weights of edges and check if it is at most k.

Alternatively, a certificate could be an empty string (0 bits). A verifier can run the Prim's algorithm to find a MST T of G. If $w(T) \leq k$, it verifies the claim otherwise rejects the claim □

Theorem 38. MAX-FLOW(G, s, t, k) *is polynomial time verifiable*

Proof. A certificate could be the "flow" on each edge $f : E \to \mathbb{R}$ expressed as $|V| \times |V|$ matrix. A verifier can just check capacity constraints and flow conservation constraints. If it is a valid flow, then it's size can be computed and verified if it is at least k. □

Theorem 39. 3-SAT(f) *is polynomial time verifiable*

Proof. As we discussed earlier, a certificate could be the assignment of each variable to 0 or 1. A verifier can evaluate f with the assignment and if the value of f is 1 it outputs **Yes** otherwise **No**. Please think about what is this **Yes** as in make the complete statement this **Yes** means. □

You can similarly prove that CLIQUE(G, k), VERTEX-COVER(G, k), HAMILTONIAN-CYCLE(G) and all the problems we discussed in the first section are polynomial time verifiable.

Important Remark: The following are few points that you should keep in mind. All these points are there in the definition above.

- We do not have to design a verifier and or technique for certifying. We only need to prove the existence of a certificate and a verifier.
- The certificate should be of polynomial size, as otherwise the verifier would take super-polynomial time just in reading the certificate. The certificate is also referred to as evidence, solution or hint.
- Verifier does not have to be unique
- Similarly, there can be many ways to certify. e.g. An independent set can be certified as the set of vertices, set of edges, complements thereof
- The verifier does not have to read the certificate. The requirement is that verifier has to say **Yes** if and only if the instance is a **Yes** instance.

It will help clarify the definition to think about the following questions.

Is $\overline{\text{3-SAT}}(f)$ polynomial time verifiable?

This problem is sometime referred to as UNSAT(f) which decides whether or not a given formula f is not satisfiable.

Is $\overline{\text{HAMILTONIAN}}(G)$ polynomial time verifiable? Similarly, the problem NO-INDEPENEDENT-SET(G, k): that requires **Yes** output, if the G does not have an independent set of size k.

Mostly-Long-Paths(G, s, t, k): Are majority of paths from s to to in G have length at least k.

8 Classes of Decision Problems: P, NP, co-NP, EXP

8.1 The Class P of problems:

P is the set of decision problems that can be solved (decided) in polynomial time, i.e. the set of decision problems for which there exists an algorithm that correctly outputs **Yes/No** on any input instance.

Examples of problems in P include the problems listed in the first column of the table below

P	NP
2-SAT(f)	3-SAT(f)
Euler-Tour(G)	Hamiltonian-Cycle(G)
MST(G, k)	TSP(G, k)
Shortest-Path(G, s, t, k)	Longest-Path(G, s, t, k)
Independent-Set-Tree(G, k)	Independent-Set(G, k)
Bipartite-Matching(G, k)	3d-Matching(P, k)
Bipartite-Vertex-Cover(G, k)	Vertex-Cover(G, k)
Linear Program	Integer Linear Program
Prime(n)	Factor(n)

8.2 The Class NP of problems:

NP is the set of decision problems that can be verified in polynomial time, i.e. the set of decision problems for which there exists a polynomial sized certificate for it's **Yes** instances and a polynomial time algorithm that takes an instance of it and a certificate and verify if the instance indeed is a **Yes** instance.

NP stands for "Non-deterministic Polynomial Time". Examples of problems in NP include the problems listed in the second column of the table above

The next very important result establishes a relationship between the two classes of problems discussed above.

Theorem 40. $\mathrm{P} \subseteq \mathrm{NP}$

Proof. Consider a decision problem $X \in \mathrm{P}$; by definition this means that there is a polynomial-time algorithm $\mathcal{A}$ that solves X. To show that $X \in \mathrm{NP}$, we must show that for any instance $\mathcal{I}$ of X, there is a polynomial time certificate C and a polynomial time verification algorithm $\mathcal{V}$ for X, that takes the instance $\mathcal{I}$ and C and return **Yes** if and only $X(\mathcal{I}) = \textbf{Yes}$.

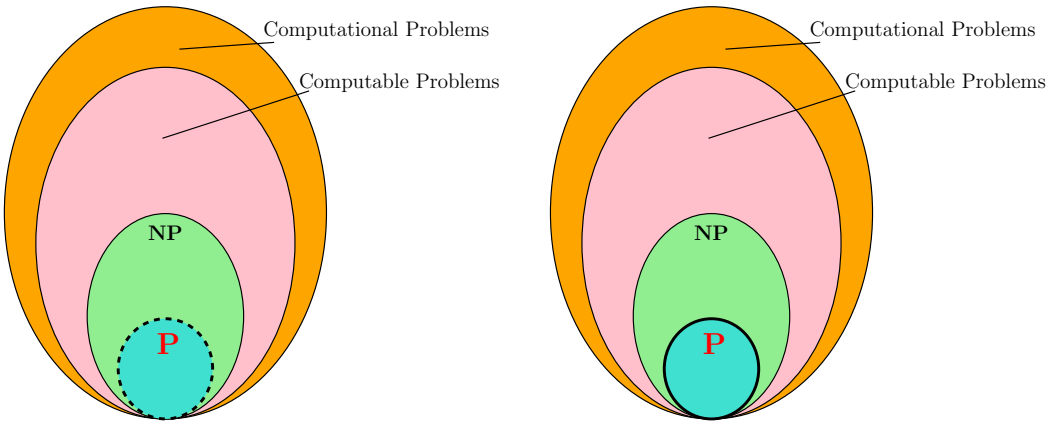
Given an instance $\mathcal{I}$ of X and the certificate C could be an empty bit string or an arbitrary bit string. There exists a verifier $\mathcal{V}$ which works as follows: $\mathcal{V}(\mathcal{I}, C) = \mathcal{A}(\mathcal{I})$. Essentially ignore the certificate C, solve the problem X on $\mathcal{I}$ using the algorithm $\mathcal{A}$ and declare the instance verified if $\mathcal{A}$ returns **Yes**, otherwise it does not verify it (i.e. return **No**). The verifier $\mathcal{V}$ takes polynomial time as it only makes a call to which is a polynomial time algorithm. □

8.3 The P vs NP Question

Many problems in computer science, mathematics, engineering, optimization and operations research and many other fields are polynomial time verifiable but have no known polynomial time algorithm. It is unknown whether P = NP. It is the biggest open problem in computer science. The question comes down to "Is verifying a candidate solution easier than finding a solution?" Although polynomial time verification seems like a weaker task than polynomial time solution, no one has been able to prove that it is actually weaker (describes a larger class of problems). Intuitively, one can check if any of possible candidate solutions verifies, finding a solution is difficult because candidate space can be exponential. For example, there are $n!$ possible Hamiltonian cycles are candidates for $\mathrm{TSP}(G, k)$ and $\binom{n}{k} = O(n^k)$ possible subsets for $\textsc{Clique}(G, k)$. Essentially, to prove there is no known better way for many problems in NP but there is no proof that there is no better way.

Let's see what will it take to prove that $\mathrm{P} \neq \mathrm{NP}$. We have proved that $\mathrm{P} \subset \mathrm{NP}$, to prove that $\mathrm{P} \neq \mathrm{NP}$, we need to show that $\mathrm{NP} \not\subset \mathrm{P}$. This can be achieved by picking any problem $X \in \mathrm{NP}$ and proving that there is no polynomial time algorithm, i.e. proving that $X \notin \mathrm{P}$. You have proved $\mathrm{P} \neq \mathrm{NP}$, You will get a million dollars from the Clay Institute (and an A in this course).

On the other hand to prove that $\mathrm{P} = \mathrm{NP}$, you will need to come up with a polynomial time algorithm for every single problem in NP. Well, using reduction you can save yourself quite some time. More on this later. But you will get a million dollars from the Clay Institute (and an A in this course).



Most researchers believe that P and NP are not the same class, actually there is almost consensus opinion that P $\neq$ NP. But nonetheless they are only opinion and no proof. [Prof. Scott Aaronson blog:] To say that "P vs NP is the central unsolved problem in computer science" is a comical understatement. P vs NP is one of the deepest questions that human beings have ever asked. There is a reason it one of 7 million-dollar prize problem of the Clay Mathematical Institute (now one of the 6). If P = NP, then mathematical creativity can be automated (the ability to verify a proof would be the same as the ability to find a proof). Since verification seems to be way easier, every verifier would have the reasoning power of Gauss and the like. By just programming your computer in polynomial time you can solve (perhaps) the other 5 Clay Institute problems.

Someone else made a statement to the effect, "just because I can appreciate good music, doesn't mean that I would be able to create good music". In the above statement, understanding (verifying) proofs (solutions) by Gauss is one thing and coming up with proofs is a totally different ballgame.

Then why isn't it obvious that P $\neq$ NP. It is generally believed that there is no general and significantly better method than the brute-force search to solve NP problems. It is said the great physicist Prof. Richard Feynman had trouble even being convinced that P vs NP was an open problem. Why can't we prove it? Well there are many (way too many) problems where we could avoid brute-force search. See the the list of "hard" problems and their easier "counterparts". Though not a decision problem, recall that we discussed that (to impress your boss) you can say that your SORTING algorithm finds that one unique permutation out of the $n!$ possible ones.

What we do in this course and all complexity theorists do is try to characterize these hard problems and say that almost all of them all essentially the same.

8.4 The Class coNP of problems:

coNP is the set of decision problems whose **No** instances can be verified in polynomial time. The **No** instances of a problem in coNP are the **Yes** instances of its complement problem. In other words, they are the complement of problems in NP. We focused on decision problem to be able to talk about the complement of problems.

For example $\overline{\text{3-SAT}}(f)$, $\overline{\text{Hamiltonian}}(G)$, $\overline{\text{Independent-Set}}(G,k)$ are in the class coNP. Please note that coNP **is not the complement of** NP. Given this definition of coNP we ask the question, is NP = coNP**?** Irrespective of the answer to P vs NP question can we certify in polynomial space that G has no Hamiltonian cycle and can we verify it in polynomial time. The following theorem states equality but only conditionally.

Theorem 41. *If* P = NP*, then* NP = coNP*.*

Proof. This can be proved quite easily, let X be a problem in coNP. We will prove that $X \in$ NP. Consider $\overline{X}$, by definition we know that $\overline{X} \in$ NP and we are given that P = NP. Hence given any instance I of X in polynomial time we can find whether $\overline{X}(I) =$ **Yes** or $\overline{X}(I) =$ **No**. Thus in polynomial time we can decide the problem X, thus $X \in$ P = NP. □

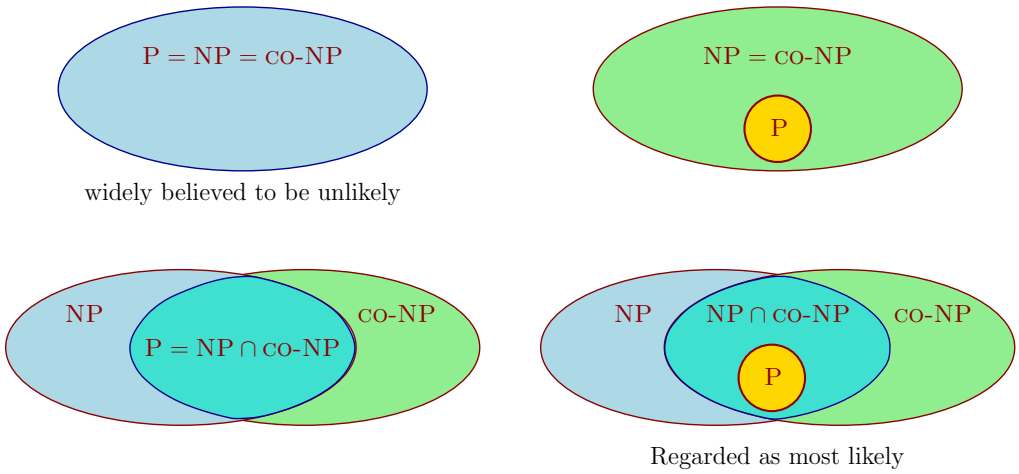
The converse of the statement is an open problem (not known to be true).

Note that it is clear (we used it in the above proof)

Theorem 42. P ⊂ coNP*. Thus* P ⊂ NP ∩ coNP

It is widely believed that P $\subsetneq$ NP ∩ coNP, meaning there are problems in NP ∩ coNP that are not in P. For example the factor(n). It is easy to easy to see that Factor$(n,k) \in$ NP. As a certificate could be the factor, and verifier can just check if the given factor is at most k and with one division verify if it indeed is a factor of n. To see that Factor$(n,k) \in$ coNP, a **No** instance of this problem, i.e. $[n,k]$ (such that n has no factor less than k) can be certified by providing the prime factorization of n. The verfier can check if every factor in the claimed factorization is prime (can be done in polynomial time as Prime$\cdot \in$ P) and check if everyone of them is greater than k. It is widely believed that Factor$(n,k) \notin$ P and this is the basis of belief in security of the RSA cryptosystem.

The following figure illustrate the possibilities of containment of P, NP, and coNP.



8.5 The Class EXP of problems:

EXP is the set of decision problems that can be solved in exponential time, i.e. the set of decision problems for which there exists an exponential time algorithm that that correctly output **Yes/No** on any instance. Clearly, P $\subseteq$ EXP as any problem which can be solved in polynomial time can also be solved in exponential time. Indeed, the following is true.

Theorem 43. NP $\subseteq$ EXP

Proof. The theorem states that any problem for which there exists a polynomial sized certificate and a polynomial time verification algorithm can be solved in exponential time. Consider any $X \in$ NP, let $\mathcal{V}$ be the poly-time verifier. We can solve X any instance I of X by considering all possible certificates C and running $\mathcal{V}(I, C)$. We return yes if $\mathcal{V}(I, C)$ return **Yes** on any of the certificates C. The runtime of this algorithm exponential as there are only exponentially many certificates of polynomial size 2^{n^k} and $\mathcal{V}$ on each of them takes polynomial time. □

It is not very hard to prove the following theorem

Theorem 44. coNP $\subseteq$ EXP

With this we get the following containment hierarchy which many people believe to be more likely. There is a very large hierarchy (called polynomial hierarchy) that complexity theorist have been studying and establish various containment.

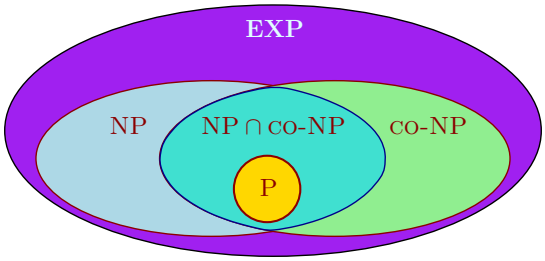


Figure 6: Relationships among complexity classes we discussed and their more likely hierarchy

9 NP-Complete Problem and NP-Hard Problems

Definition 45 (NP-Hard)**.** *A problem X is* NP-Hard*, if every problem in* NP *is polynomial time reducible to X, i.e.*

- $\forall\, Y \in \text{NP}\ Y \leq_p X$
- *In other words, the problem X is at least as hard as any problem in* NP

Definition 46 (NP-Complete)**.** *A problem $X \in$ NP is* NP-Complete*, if every problem in* NP *is polynomial time reducible to X, i.e.*

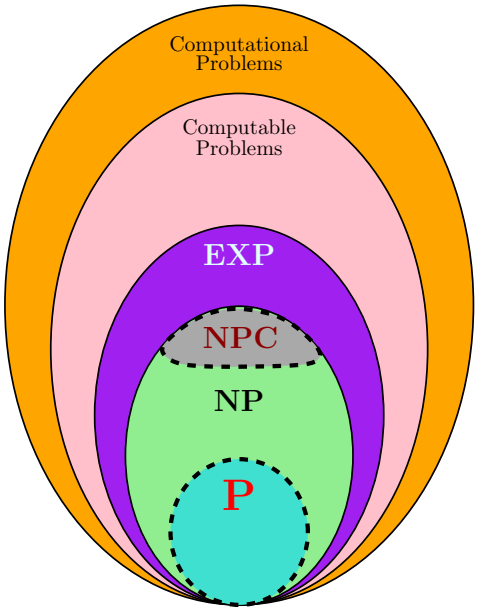
- $X \in \text{NP}$
- $\forall\, Y \in \text{NP}\ Y \leq_p X$
- *In other words, the problem X is at least as hard as any problem in* NP

9.1 The (sub)class NPC of problems

is the set of all problems in NPC that are NP-Complete.

To appreciate the above definition and the problems in NPC. Here is what it means.

- The class NPC is the set of hardest problems in the class NP.
- In the sense that, if **any** NP-Complete problem can be solved in polynomial time, then all problems in NP can be solved in polynomial time



Recall this hierarchy, the subclass NPC is also shown. The definition of NP-Complete implies the following.

- We already proved that P $\subseteq$ NP. By definition we have that NPC $\subseteq$ NP
- P $\subseteq$ NP means that if you take any $X \in$ NPC or NP and prove that it cannot be solved in polynomial time, then you have proved that P $\neq$ NP. You will win a million dollars and recall my promise you will get an A in this course.
- The definition of NP-Complete also means that if you take any $X \in$ NPC and prove that it is in P (by giving a polynomial time algorithm), then you have proved that P = NP and many more things, such as NP = coNP. The polynomial hierarchy collapses (almost). Again you will get a million dollars and get an $A+$ in this course (actually in both cases).
- Thus far no polynomial time algorithm is known for any NP-Complete problem. A lot of great researchers did and do try, as there are many interesting and practical problems in NPC
- Also thus far, there is no proof of impossibility of existence of a polynomial time algorithm for any NP-Complete problem
- Similarly, people did and do try on this direction more, as this is widely held belief and will prove that P $\neq$ NP

So what is the impact of defining yet a new class of problems. Recall that to prove

$P \neq NP$ all you had to do is to pick any problem $X \in NP$ and show that there is no polynomial time algorithm for X, i.e. show that $X \notin P$. Note you had to do it for just one problem in NP.

Now the other possibility is also made "easier". To prove that $P = NP$ all you have to do is pick any problem $X \in NPC$ and design a polynomial time algorithm for X, i.e. prove that $X \in P$. Note that any problem in NPC a problem that is not NP-Complete will not work. Before 2002-3, people didn't know if $\text{Prime}(n) \in P$. It was shown to be in P but didn't affect the P vs. NP question. the following theorem summarizes the above points. It's proof follows from definition of polynomial time reduction and that of NP-Complete.

Theorem 47. *Let X be any* NP-Complete *problem. X is polynomial time solvable if and only if* $P = NP$.

In addition to the above theorem, why should in practice one prove a problem to be NP-Complete. Suppose you fail to find an efficient solution to a problem, after many trials and with some experience you can sense similarity to some known hard problem, one should really give a try to prove to prove the problem to be NP-Hard. The following pictures summarize it very beautifully.

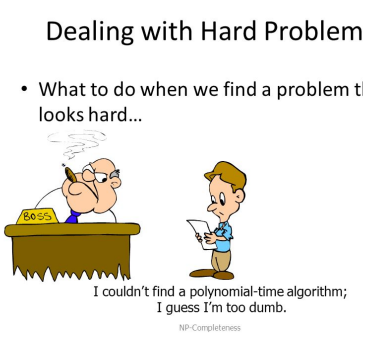


Figure 7: Honesty is the best policy! Now you are fired

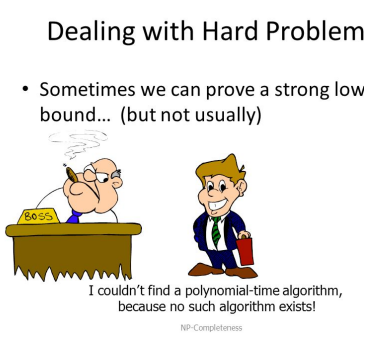


Figure 8: Careful! you might be claiming $P \neq NP$

Figure 9: You still need to prove that the problem is NP-Complete

When you prove that a problem is NP-Complete, there is a good evidence that it is very hard, at least by the almost consensus opinion. Unless your interest is proving $P = NP$, you should stop trying finding efficient algorithm, instead you can focus on efficient algorithm for the special cases that you are more likely going to encounter, employ some heuristic search techniques, design algorithm that almost all cases give the best algorithm, try using randomized algorithm that yield good result in expectation

(or with high probability), or design approximation algorithm that gives a solution with guaranteed quality.

Note that a NP-Complete problem single-handedly capture the essential difficulty of all problems in NP. Two fundamental questions, could there be any NP-Complete problem at all? How would one prove a problem to be NP-Complete. Can we do so many reductions. The next section is dedicated to the latter question. While we will give a somewhat formal proof for the affirmative answer to the first question, it is not very hard to imagine. Here is a rough idea of the argument for a problem to be NP-Complete.

Let A be any polynomial time algorithm working on bit-strings that outputs **Yes/No** based on some unknown but consistent logic. Meaning we don't know the functionality of the algorithm (the problem specs), but it's input can ultimately be transformed into bit strings and its output is either **Yes** or **No**. Consistent means that the same input will always give the same answer.

We define a (dummy) problem H(A) as follows: "Is there any polynomial sized bit-string S (input to A) on which A outputs **Yes**?". Clearly H(A) $\in$ NP, because there exists a polynomial time certificate (the string S), which can be verified by running A on S and verifying if A(S) = **Yes**. This verification takes polynomial time, because A is a polynomial time algorithm.

Now we claim that H is NP-Complete. We need to prove that H is NP-Hard, as H $\in$ NP. In other words, we need to prove that any problem $Y \in$ NP is reducible to H.

$Y \in$ NP means that for any instance I of Y, there exists polynomial sized certificate S and a polynomial time algorithm $\mathrm{V_Y}$ such that if $Y(I)$ = **Yes** if and only if $\mathrm{V_Y}(I, S)$ = **Yes**. Now we will show that any instance I of Y can be transformed to an instance of H($\cdot$) with the same answer. Namely, suppose there is an algorithm B to decide $H(\cdot)$. We will use B to solve Y. Given an instance I of Y, we call B with input $\mathrm{V_Y}$ and if it return **Yes**, we make $Y(I)$ = **Yes**, otherwise $Y(I)$ = **No**.

To understand this proof, just expand the statement $B(\mathrm{V_Y})$ = **Yes**. The problem $H((\mathrm{V_Y}))$ by definition asks "Is there a polynomial sized certificate C on which the algorithm $\mathrm{V_Y}$ outputs **Yes**?". Thus $B(\mathrm{V_Y})$ = **Yes** means "yes there is a polynomial sized certificate C on which ($\mathrm{V_Y}$) outputs **Yes**", in other words, it says "yes there is a solution to the problem Y on instance I. Hence we used the claimed solution B for H to solve the problem Y.

If you understand this, you understood the seminal Cook-Levin theorem and got your first NP-Complete problem, namely H.

10 The first NP-Complete Problem. Circuit-SAT

In this section we give a more formal proof of the above intuition how is it not so hard to prove a problem to be NP-Hard. We will prove many other problems NP-Hard after we establish a first one by the method outlined above.

Theorem 48. Circuit-SAT *is* NP-Complete

Proof. We first need to prove that **Circuit-SAT** $\in$ **NP**, in other words CIRCUIT-SAT is poly-time verifiable.

Given (an encoding of) a Circuit-SAT problem C (a DAG). A certificate for the claim is can be an assignment of Boolean values all input wires in C. The verifier works as follows: It evaluate the gates of C in topological order, just checks each gate and the output wire of C. If for every gate, the computed output value matches the value of the output wire given in the certificate and the output of the whole circuit is 1, then the verifier outputs **Yes**, otherwise **No**. The algorithm is executed in poly time (even linear time) (requires a topological sort, n is the number of gates, what is the maximum number of edges in this DAG?)

Next we need to prove that **Circuit-SAT is NP-Hard**, that is every problem in NP reduces to it. For this we critically have to use the definition of NP as we do not know anything else about the specific problem.

Let $A \in$ NP. We know that there exists a verification algorithm $\mathcal{V}$ that takes an instance I of A and a certificate S certifying that I is a **Yes** instance of A. The algorithm V works as follows $V(I, S) =$ **Yes** iff $A(I) =$ **Yes**. The runtime of $\mathcal{V}$ is polynomial in $|I|$ and also $|S|$ is polynomial in $|I|$.

The algorithm V can be implemented in a digital computer, that takes as input I and S and produce a **Yes**/**No** output ($= 1/0$) in a polynomial number of clock cycles. A digital computer has a state or configuration, represented by all of its internal registers, control registers, program counter etc. At each clock cycle with execution of an instruction, a huge (but fixed-size) set of combinational circuits map the current configuration to the next configuration. The computer hardware that accomplishes this mapping can be implemented by a combinatorial circuit. We eliminate the clock and replicate the combinatorial circuit that maps a configuration to the next. See the following two figures, that illustrates construction of such a combinatorial circuit.

The verification algorithm V for the problem A can be implemented on a digital computer, and from there, we can generate a combinational circuit C that "executes" V. If the output of the circuit C is 1 for a given input, then that input verifies the problem. Thus, if we could solve the Circuit-Sat(C) problem in polynomial time, then we

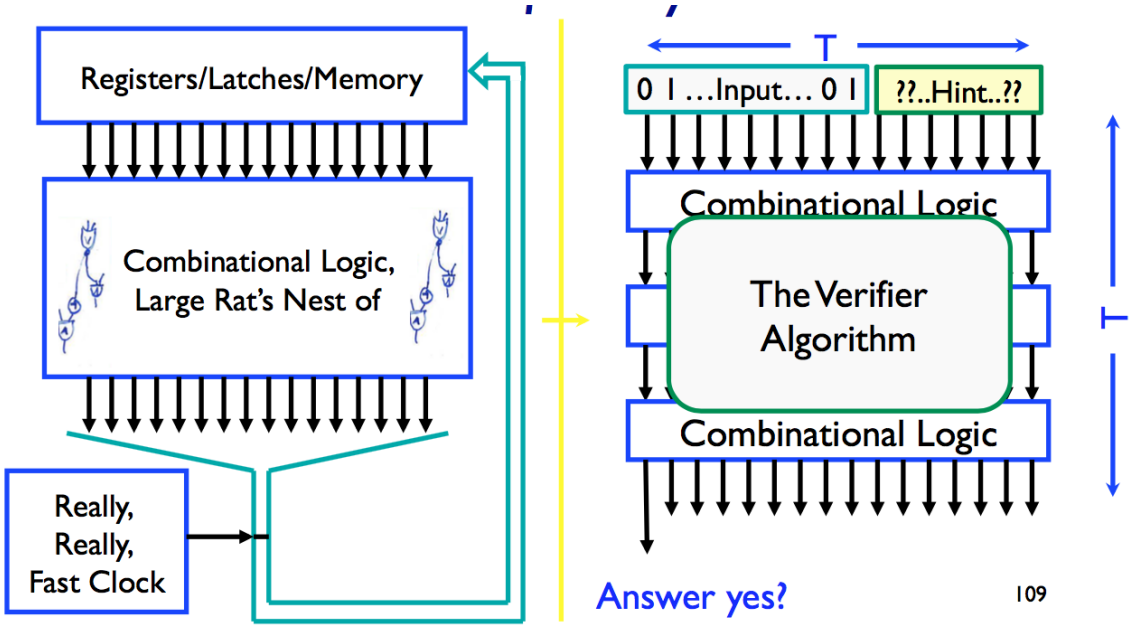


Figure 10: Figure Credit: https://courses.cs.washington.edu/courses/cse421/12wi/

would be able to find a solution to any problem for which a solution can be verified in polynomial time.

Recall that a solution to circuit-sat tells you whether there exists a combination of ones and zeros that produces a 1 as the output. But that output is the solution to the other NP problem.

An important point is that the number of stages in that the circuit C is equal to the the number of clock cycles that the algorithm takes — since $A \in$ NP, then that number of clock cycles is polynomial in $|I|$, and thus the number of logic gates in C will be polynomial in $|I|$

This construction shows that A reduces to the Circuit-Sat(C) problem. That means that Circuit-Sat(C) is at least as hard as the problem A. Since A is a generic problem in NP - Circuit-Sat(C) is at least as hard as any problem in NP. In other words $\forall\ A \in$ NP $A \leq_p$ Circuit-Sat(C). This is the definition of an NP-Hard problem:

Therefore, we conclude that Circuit-Sat(C) is NP-Hard, together with the first part above we get that Circuit-Sat(C) is NP-Complete. Keep in mind, that an NP-Hard problem does not have to be in NP (technically, it doesn't even need to be a decision problem!) □

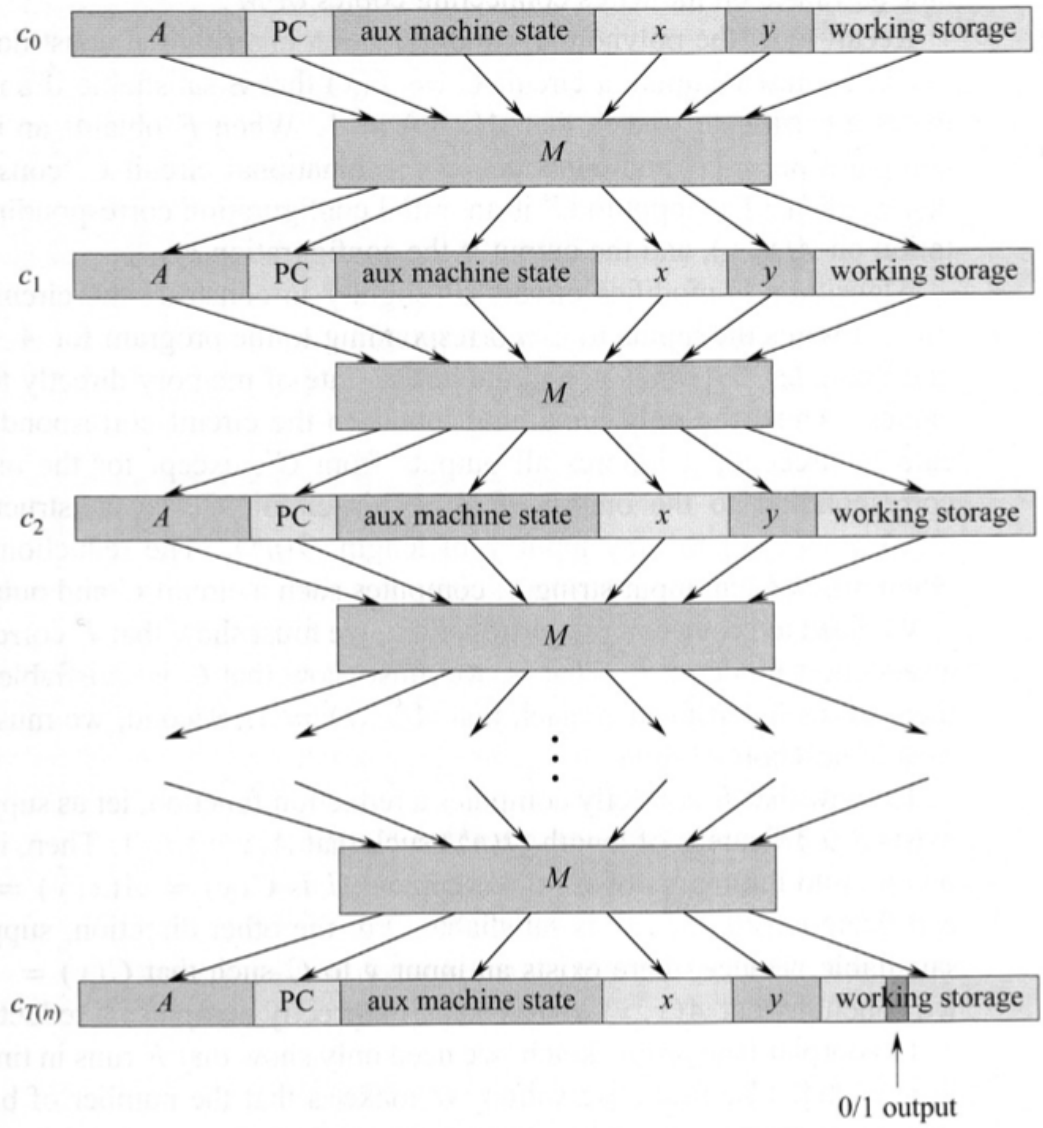


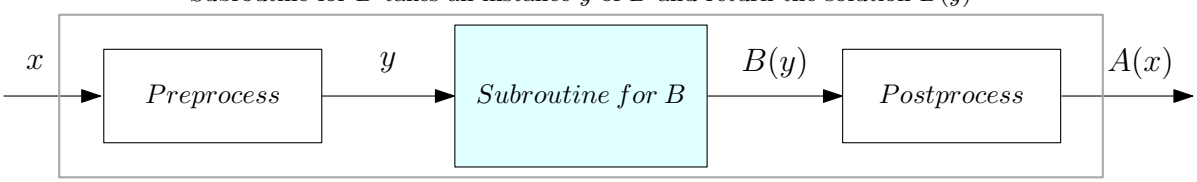
Figure 11: Figure Credit: Figure 34.9, Introduction to Algorithms, Cormen, Leiserson, Rivest, Stein

11 Proving other NP-Complete problems

Now that we have gotten our first NP-Complete problem, we address the other questions. How would one prove a problem to be NP-Complete. Can we do so many

reductions? It turns out we can do only one reduction but with a different perspective to prove a problem to be NP-Complete.

Recall that we used polynomial time reduction as an algorithm design paradigm. It was the view guided by this familiar diagram. The following result immediately follows from



Algorithm for A transform an instance x of A to an instance y of B. Then transform $B(y)$ to $A(y)$

the definition of polynomial time reduction, we discussed it earlier and used it to design algorithms.

Lemma 49. *Suppose $A \leq_p B$. If B is polynomial time solvable, then A can be solved in polynomial time.*

In this section, we use another consequence of the definition of polynomial time reduction, (actually this is in essence some kind of contrapositive of the above lemma).

Theorem 50. *Suppose $A \leq_p B$. If A is* NP-Complete*, then B is* NP-Complete.

This follows just by the transitivity of polynomial time reduction. This is one of those concepts, that I always advise you to understand it by any means. It must be clear, if it is not clear, then think about it and keep thinking until it is clear. If it is still not clear do the water trick with it, if that doesn't work, please use combinations of languages with the water trick. If even that doesn't work, then use the drilling technique or anything that could work, but it must be clear.

Actually it is not very difficult to understand. For example Suppose we have the theorem that Clique(G, k) is NPComplete. It should not be very difficult to conclude that Independent-Set(G, k) is NP-Complete. Recall the reduction Clique$(G, k) \leq_p$ Independent-Set(G, k)

In general, to prove X to be NP-Complete, we **reduce some known NP-Complete problem Z to X**. Many many people do the reduction in the opposite direction, which is wrong, so make sure you have understood it so clearly, that you never make that mistake.

Theorem 51. *If Z is* NP-Complete*, and*

1. $X \in$ NP

2. $Z \leq_p X$

then X *is* NP-Complete

1. $X \in$ NP is explicitly proved.
2. $\forall\, Y \in$ NP $Y \leq_p X$ follows by the transitivity of polynomial time reduction $\forall\, Y \in$ NP $Y \leq_p Z$ is true as Z is NP-Complete. i.e. $[Y \leq_p Z \wedge Z \leq_p X] \implies Y \leq_p X$

11.1 The Cook-Levin theorem: SAT is NP-Complete

The original first NP-Complete problems was the SAT(f) problem proved by Stephen Cook in 1971, though a few years earlier Leonid Levin had proved it to be NP-Complete Indeed he gave six NP-Complete problems (in addition) to some other results, but those results became known a little later than the Cook's results. Thus the following theorem is called the Cook-Levin theorem. We prove it by reducing the Circuit-Sat(C) problem to it (the original proofs were different and did not start from a known NP-Complete problem.)

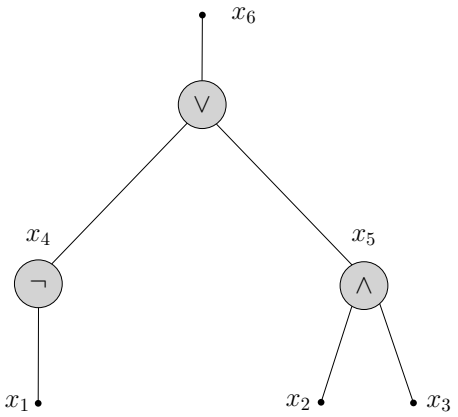
Theorem 52 (Cook-Levin Theorem). SAT(f) *is* NP-Complete.

Proof. First we prove that SAT $\in NP$. A certificate could be just an assignment to each of the formula's input n bits. Verification can be done simply by traversing the formula from left to right and evaluating the ANDs and ORs along the way. Total runtime is $O(n+m)$, where n is the number of variables and m is the number of clauses. To prove NP-Hardness, we reduce the Circuit-Sat(C) problem to SAT(f).

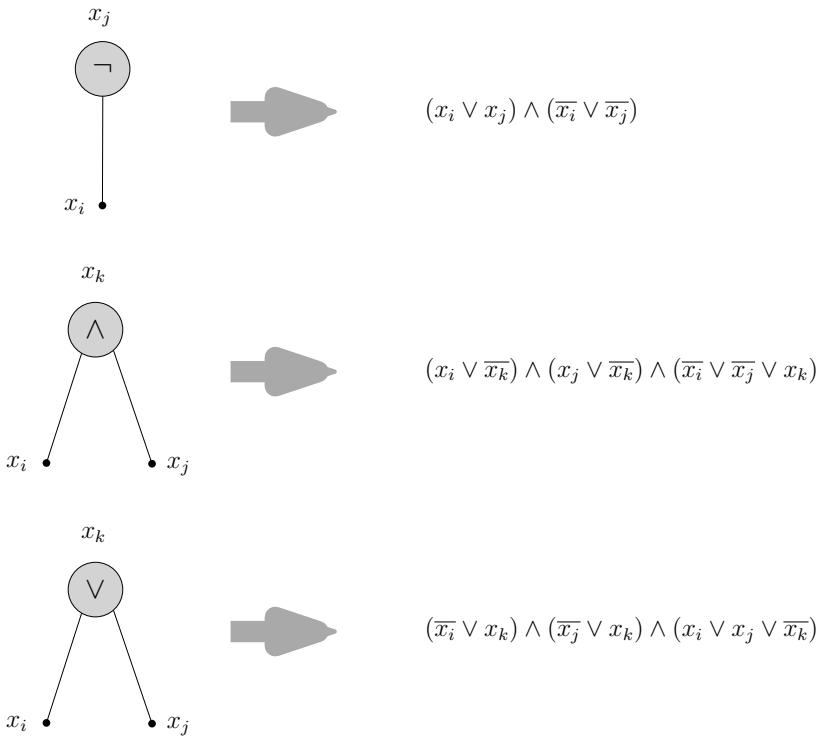
Theorem 53. Circuit-Sat(C) $\leq_p$ **SAT**(f)

Proof. The Intuitive idea for reduction is quite simple, as a combinatorial circuit is just a generalization of cnf formula. Suppose $\mathcal{A}$ is an algorithm to decide SAT(f). Given an instance C of the Circuit-SAT(C) problem, we will design an algorithm that perform polynomial time work to transform C into a cnf formula f and make a call $\mathcal{A}(f)$ to decide whether or not Circuit-SAT(C) = **Yes**.

Introduce a Boolean variable for each input and output of each gate of C, see Figure below.



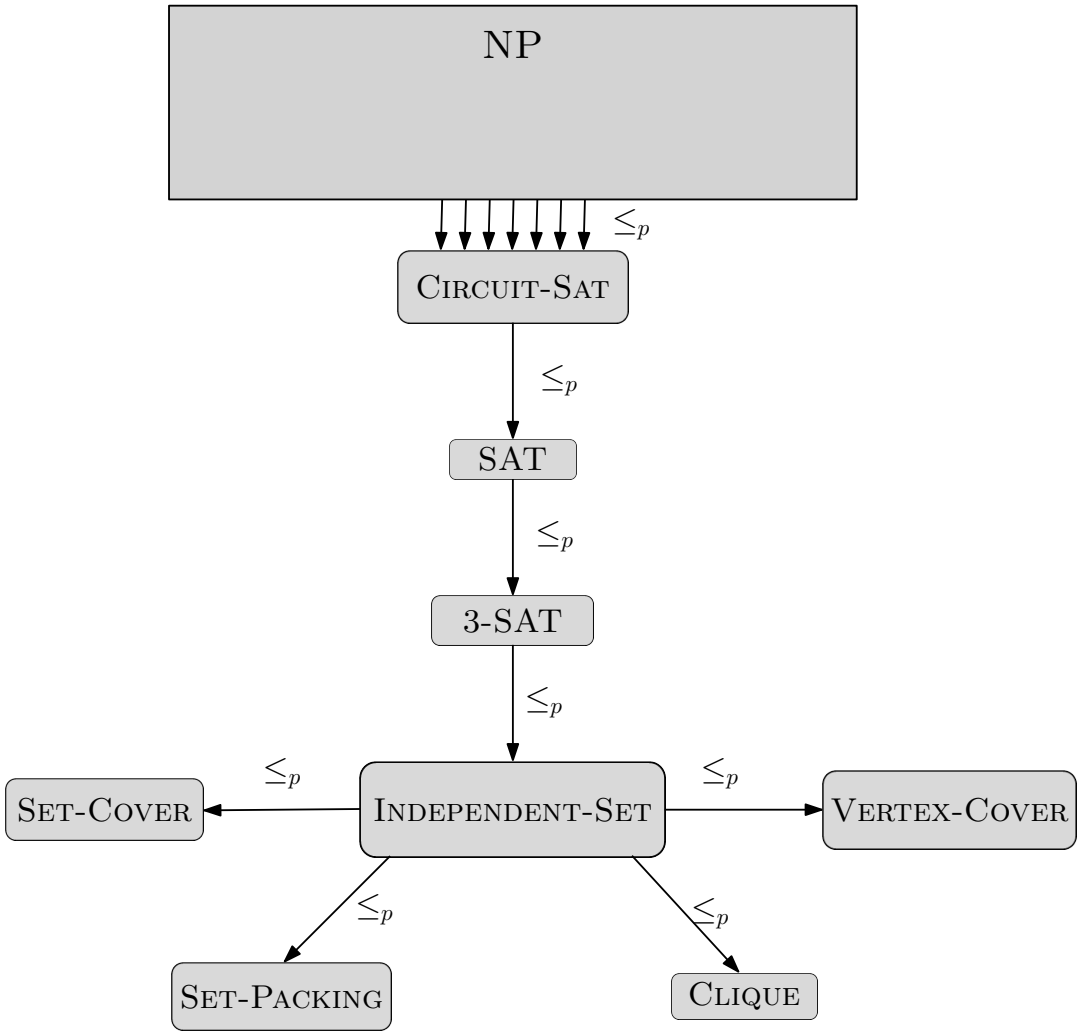
For each gate we will write a few clauses that encode the relationship between the gate output and its inputs (the direct predecessors of nodes in the DAG). In other words, these clauses will evaluate to true if and only if the gate output is true. The clauses introduced for each type of gates is show in the following figure.



It is easy to verify that the gates and corresponding formula are equal. The output gate value is encoded by a clause containing just the corresponding variable. The final

formula f is a conjunction of clauses made for all gates. f is "equal" to the circuit C, i.e. the value of f for any assignment to the variables is equal to the value of the output gate for the same assignment to the inputs of C. Thus f is equisatisfiable with C. Therefore, we call the algorithm CIRCUIT-SAT$(C) =$ **Yes** if and only if $\mathcal{A}(f) =$ **Yes**. The reduction takes polynomial time, as we only have to traverse the circuit (DAG) only once and write the formula. □

□



We will show a few problems to be NP-COMPLETE, but given that SAT is NP-COMPLETE, we already know many problems to be NP-COMPLETE from the reductions that we have proved and by transitivity.

With the already known reductions, we know the following problems to be NP-Complete

- Recall our discussion of them being in NP
- Write all the reductions statements from and show which implies what to be NP-Complete
- May be show the gadgets or condensed slides of the reductions
- SAT $\leq_p$ 3-SAT
- 3-SAT $\leq_p$ Independent-Set
- Independent-Set $\leq_p$ Clique
- Independent-Set $\leq_p$ Vertex-Cover
- Vertex-Cover $\leq_p$ Set-Cover
- Independent-Set $\leq_p$ Set-Packing

11.2 Directed Hamiltonian Cycle is NP-Complete

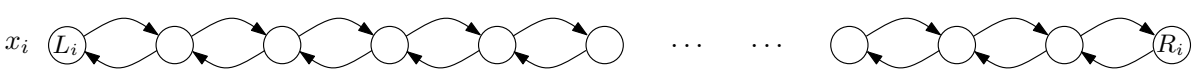
Theorem 54. Dir-Ham-Cycle(G) *is* NP-Complete.

Proof. First we show that Dir-Ham-Cycle(G) $\in$ NP. A certificate could be the claimed cycle in G, which is of size n (a sequence of vertices). This certificate can be verified by checking if every vertex is listed exactly once and each successive pair is an edge in G (including the last vertex and the first one.)

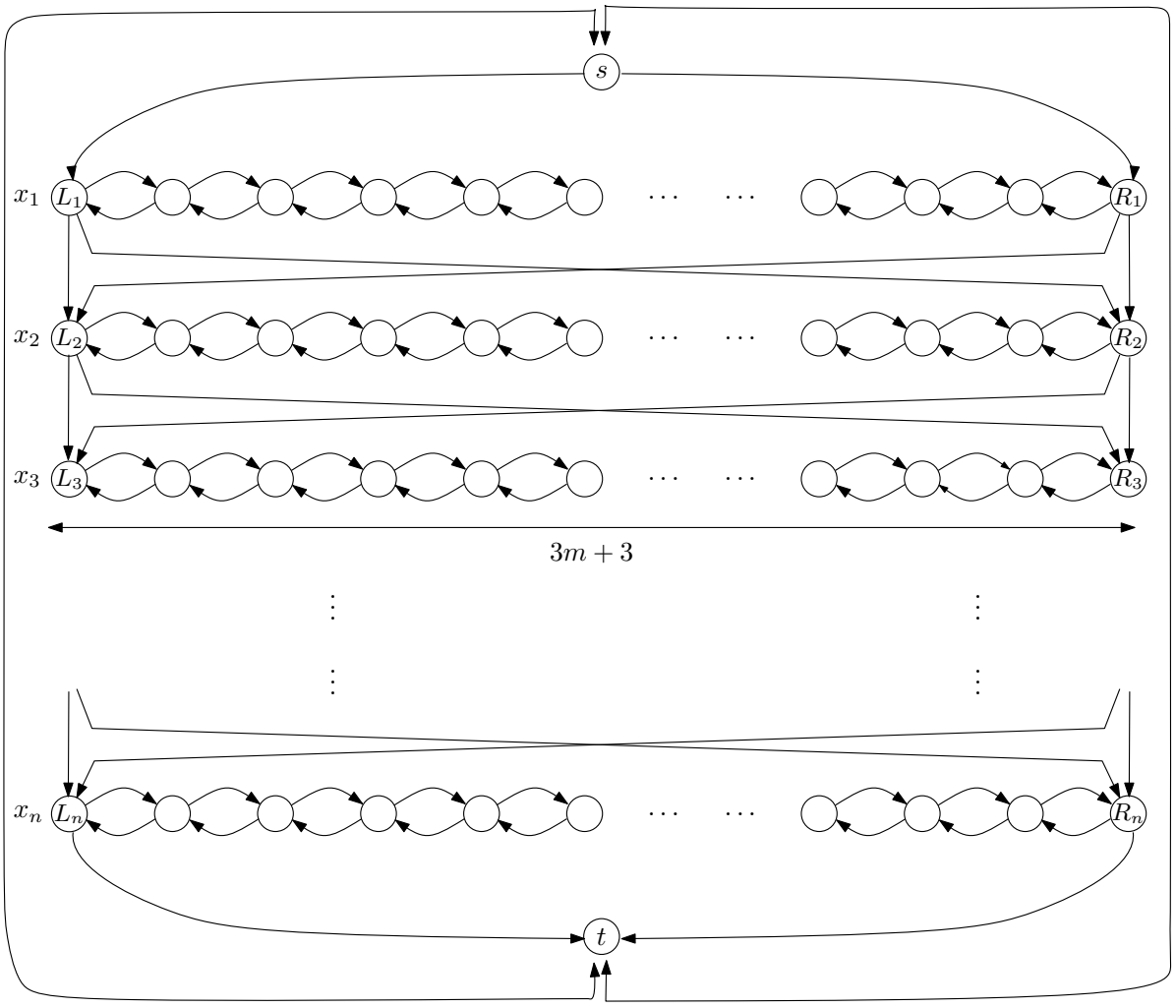
To show that Dir-Ham-Cycle(G) is NP-Hard, we reduce the already known NP-Complete problem 3-SAT(f) to it. Given an instance f of the 3-SAT(f) problem. Suppose f has n variables $x_1, \ldots, x_n$ and m clauses $C_1, \ldots, C_m$.

In this graph there will be 2^n Hamiltonian cycles corresponding to the 2^n possible assignments to variables $x_1, \ldots, x_n$. We will create a graph in which there will be a gadget to represent each variable. We will introduce some structure for each clause such that the graph has a Hamiltonian cycle if and only if the formula is satisfiable.

For each variable x_i we make the following structure. A sequence of $3(m+1)$ vertices successively adjacent to each other in both direction.



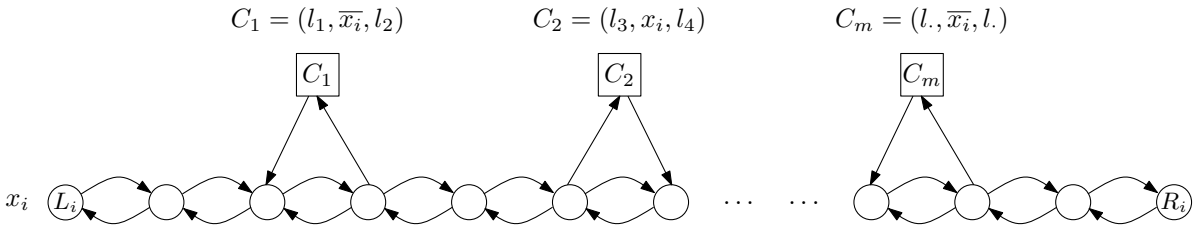
In the final graph we would traverse these vertices corresponding to x_i in either direction depending on the value of x_i. If x_i is true or we want it to be true, we will traverse these vertices from left to right (from L_i to R_i) and vice-versa. Consider the following graph



This graph has 2^n directed Hamiltonian cycles that traversing each of the gadgets corresponding to variables in exactly one direction. These 2^n directed Hamiltonian cycles correspond to the 2^n possible assignments to the n variables. We want to be able to restrict availability of a directed Hamiltonian cycle only when there is a satisfying assignment. For notice that restrictions on satisfying assignments comes from clauses only, which we have not used so far in the graph. We incorporate them as follows.

For each clause we add another vertex, called clause nodes. Each clause node will be hooked to three gadgets corresponding to the variables represented by the literals in

$C = (l_1, l_2, l_3)$. Let x_i, x_j, x_k be the variables represented by literals l_1, l_2, l_3, respectively. A clause node will be adjacent to two distinct vertices each in the gadget of x_i, x_j, and x_k. The following figure shows how to hookup a clause to gadgets.

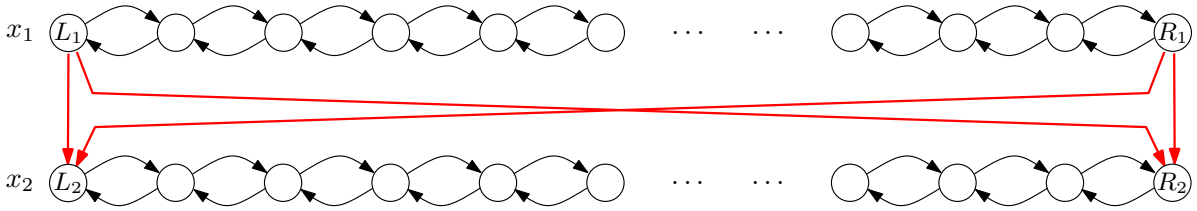


In other words, each clause node is adjacent to two distinct consecutive vertices in the gadgets of x_i, x_j and x_k. If the variable x_i appears in a clause as x_i, then clause node is hooked up to the gadget so as it can be inserted between two consecutive vertices while traversing from left to right. If x_i appears as $\overline{x_i}$ in a clause, then the clause node can be inserted between two consecutive vertices in a right to left traversal.

Each clause node has in-degree and out-degree equal to 3. Since a variable could appear in all the clauses, there are enough vertices in its gadgets so each clause can be hooked up.

Lemma 55. *Let G be the graph constructed as above from a* 3-CNF *formula f. Then G has a directed Hamiltonian cycle if and only if there is a satisfying assignment for f.*

Proof. If f is satisfiable, then G has a directed Hamiltonian cycle: Let $(x_1, x_2, \ldots, x_n) = (b_1, b_2, \ldots, b_n)$ be a satisfying assignment for f. . We show a directed Hamiltonian cycle, start traversal from s, if $b_1 = F$ [resp. $b_1 = T$], take the (s, R_1) edge [resp. (s, L_1) edge], and traverse all the vertices in gadget of x_1. If there is a clause containing $\overline{x_1}$ [resp. x_1] and the clause node is not already visited we take the detour to the clause node and continue traversing, until we reach L_1 [resp. R_1]. If $b_2 = F$ we go from the current vertex L_1 or R_1 to R_2, else we go to R_2 and continue traversal, until we reach t and then back to s.



This traversal visits every node exactly once, because all vertices within a gadget and s and t are clearly visited. All the clause nodes are also visited because all clauses are satisfied by this assignment, there is at least one literal satisfying it, and while traversing the first such literal, we will take that detour and visit the clause node.

The other direction is almost symmetric. Consider the Hamiltonian cycle, it must visit each of the variable gadget either from left to right or vice versa (also clause nodes must be visited). Depending on which case it is, we will set the values of the variables accordingly. □

□

For more details please read notes at https://courses.engr.illinois.edu/cs573/fa2013/lec/slides/04_notes.pdf

11.3 Dir-Ham-Path is NP-Complete

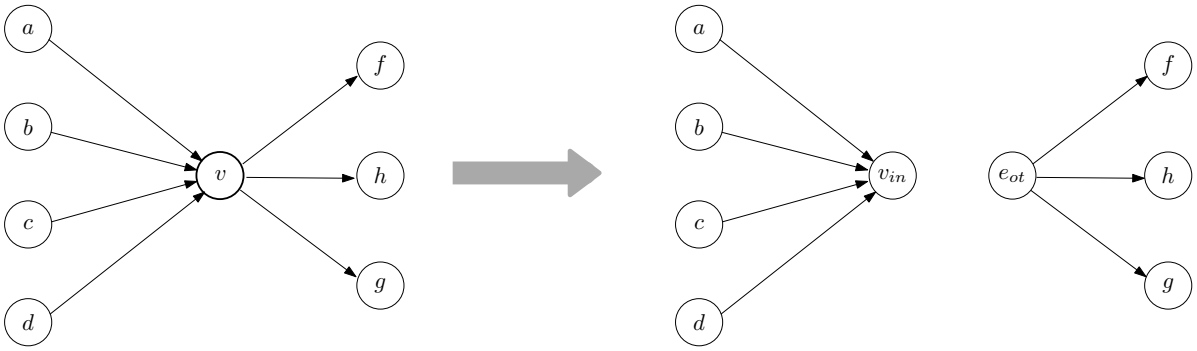
Theorem 56. Dir-Ham-Path *is* NP-Complete

Proof. The Dir-Ham-Path(G) is clearly in NP. To prove that it is NP-Hard we reduce the Dir-Ham-Cycle(G) problem to it.

Given an instance $G = (V, E)$ of the Dir-Ham-Cycle(G) problem, consider any vertex $v \in V(G)$, we make a new directed graph G' as follows: $V(G') = V(G) \setminus \{v\} \cup \{v_{in}, v_{out}\}$.

The new vertex v_{in} has an incoming edge from all the in-neighbors of v in G and the the vertex v_{out} has an outgoing to edge to all the out-neighbors of v in G. In other words

$E(G') = E \setminus \{(u, v) : (u, v) \in E\} \setminus \{(v, w) : (v, w) \in E\} \cup \{(u, v_{in}) : (u, v) \in E\} \cup \{(v_{out}, w) : (v, w) \in E\}$



It is easy to see that G has a directed Hamiltonian cycle if and only G' has a directed Hamiltonian path. Because a Hamiltonian cycle in G must contain v by splitting the vertex v open, we get a Hamiltonian path from in G' from v_{out} to v_{in}. Similarly, any Hamiltonian path in G' must begin at v_{out} and end at v_{in}, by joining the two end points of this path, we get a Hamiltonian cycle in G. □

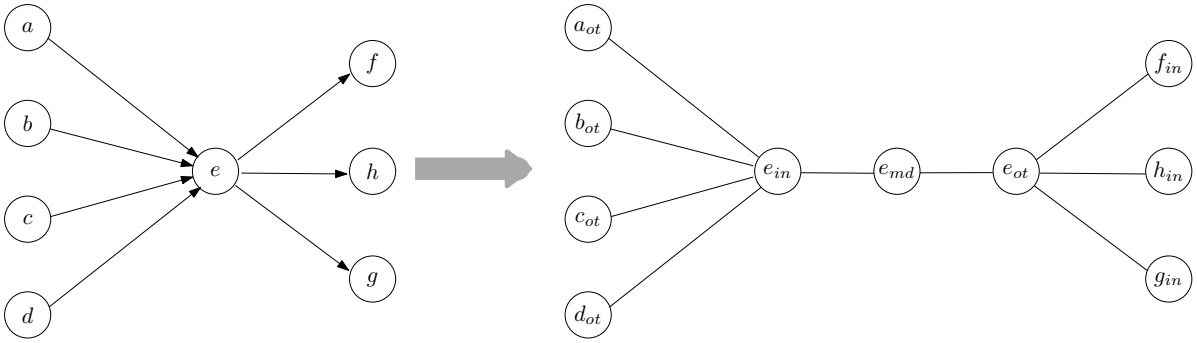
Please make one full example of a small graph using the above reduction.

11.4 Ham-Cycle is NP-Complete

Theorem 57. Ham-Cycle *is* NP-Complete

We will reduce Dir-Ham-Cycle(G) to the Ham-Cycle(G) problem, as we already proved that to be NP-Complete.

Given an instance $G = (V, E)$ of Dir-Ham-Cycle(G) problem with $|V| = n$ and $|E| = m$. We want to get rid of edge directions. We construct an undirected graph $G_u = (V_u, E_u)$ on $3n$ vertices and $m + 2n$ edges as follows. We split every vertex v inV into three v_{in}, v_{md}, v_{ot} (in, mid and out version of the vertex v) and add these three vertices to V_u. We create an edge between the in and mid version and an edge between the mid and in version of each vertex, i.e for each $v \in V$, the edges (v_{in}, v_{md}) and (v_{md}, v_{ot}) are in E_u. For each directed edge $(x, y) \in E$, we make the edge (x_{ot}, y_{in}) in E_u. Thus each directed edge in E introduce exactly one edge in E_u.



Lemma 58. *G has a directed Hamiltonian cycle if and only if G_u has an (undirected) Hamiltonian cycle.*

Proof. Suppose G has a directed Hamiltonian Cycle, C. Let $C = v^1, v^2, \ldots, v^n, v^1$. We argue that the $C_u = v^1_{in}, v^1_{md}, v^1_{ot}, v^2_{in}, v^2_{md}, v^2_{ot}, \ldots, v^n_{in}, v^n_{md}, v^n_{ot} v^1_{in}$ is a Hamiltonian cycle in G_u. In other words if we replace every vertex v^i in C with it's in, mid, and out versions in this order, then we get a cycle. This is very clear by construction.

If G_u has an undirected Hamiltonian cycle C_u, then G has a directed Hamiltonian cycle. Any Hamiltonian cycle in G_u must all vertices in V_u in either of the following two orders.

$\ldots, \ldots, v^i_{in}, v^i_{md}, v^i_{ot}, v^j_{in}, v^j_{md}, v^j_{ot}, v^k_{in}, v^k_{md}, v^k_{ot} \ldots, \ldots$ or

$\ldots, \ldots, v^i_{ot}, v^i_{md}, v^i_{in}, v^j_{ot}, v^j_{md}, v^j_{in}, v^k_{ot}, v^k_{md}, v^k_{in} \ldots, \ldots$

This is so because any other order will make two vertices consecutive which do not have an edge between them. Now If we merge these three versions of each vertex into one vertex, we get a directed Hamiltonian cycle in G.

□

11.5 TSP is NP-Complete

Theorem 59. TSP *is* NP-Complete

Proof. We will reduce the Hamiltonian-Cycle(G) problem to the TSP(G, k), since we have proved that Hamiltonian-Cycle(G) is NP-Complete.

The reduction is quite straightforward, since the only difference in TSP(G, k) problem is that the input graph is weighted and we need the number k. Let $G = (V, E)$ be an instance of the Hamiltonian-Cycle(G) problem, with $|V| = n$. We construct a complete graph G' on the vertex V and with weights on edges as follows: For a pair of vertices v_i and v_j in $V(G')$

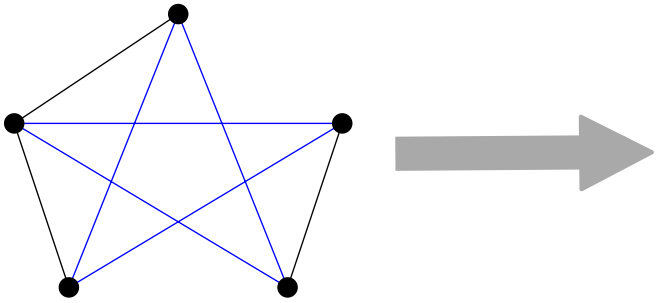
$$w(v_i, v_j) = \begin{cases} 1 & \text{if } (v_i, v_j) \in E(G) \\ 2 & \text{else} \end{cases}.$$

Note that G' is a complete graph, as every pair of distinct vertices have an edge with weight 1 or 2.

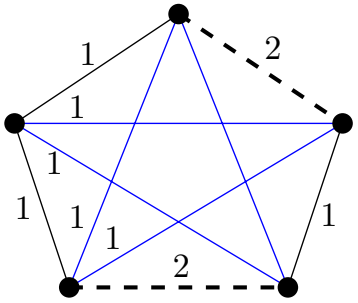
Lemma 60. *G has a Hamiltonian cycle if and only G' has a TSP tour of length n.*

Proof. If G has a Hamiltonian cycle, then the same cycle exists in G' and each edge in the cycle has weight 1 in G'. Thus this tour is of length n.

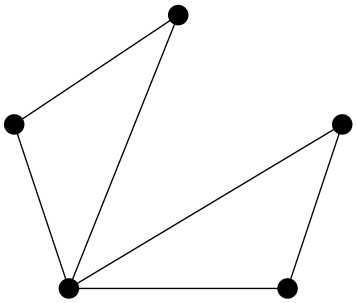
Similarly, if G' has a TSP tour of length n, then in that tour since it contains exactly n edges, there cannot be any edge with weight 2. The edges corresponding to those in the tour gives a Hamiltonian cycle in G. □



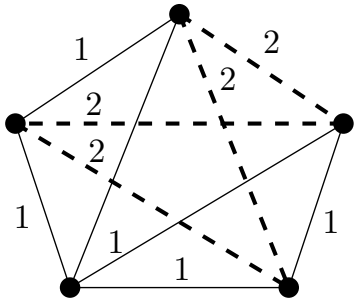
Hamiltonian cycle in G shown in blue



TSP tour in G' of length 5 shown in blue



No Hamiltonian cycle in G



No TSP tour of length 5 in G'

The reduction is clear now and is clearly a polynomial time reduction. □

11.6 Subset-Sum is NP-Complete

Theorem 61. SUBSET-SUM *is* NP-COMPLETE

Proof. Recall the subset problem, given a set $U = \{a_1, a_2, \ldots, a_n\}$ of integers, a weight function $w : U \to \mathbb{Z}^+$, and a positive integer C. The SUBSET-SUM(U, w, C) asks "is there a subset $S \subset U$ such that $\sum_{a_i \in S} w_i = C$?"

Note that the inputs w and C are input with their binary (base 2) encodings. If they were encoded in unary, then the dynamic programming based solution we discussed is a polynomial time algorithm (in the size of the input) and that will make SUBSET-SUM to be in P.

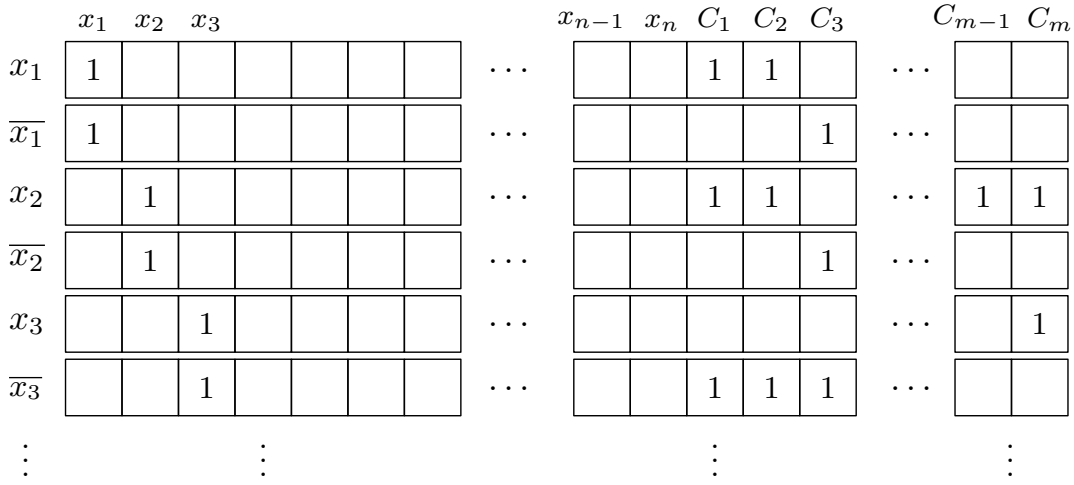
Again it is easy to see that SUBSET-SUM $\in$ NP. To show that it is NP-HARD we will reduce the 3-SAT(f) problem to it. i.e. we will prove that 3-SAT$(f) \leq_p$

SUBSET-SUM$(\cdot, \cdot, \cdot)$

Given an instance of the 3-SAT(f) problem - a 3-CNF formula f- we will construct an instance $[U, w, C]$ of the SUBSET-SUM problem, such that f is satisfiable if and only SUBSET-SUM$(U, w, C) =$ **Yes**.

Suppose f has n variables and m clauses. We will make a SUBSET-SUM instance with $|U| = 2n + 2m$ weight of each objects will be a $n + m$-digits long integer (decimal).

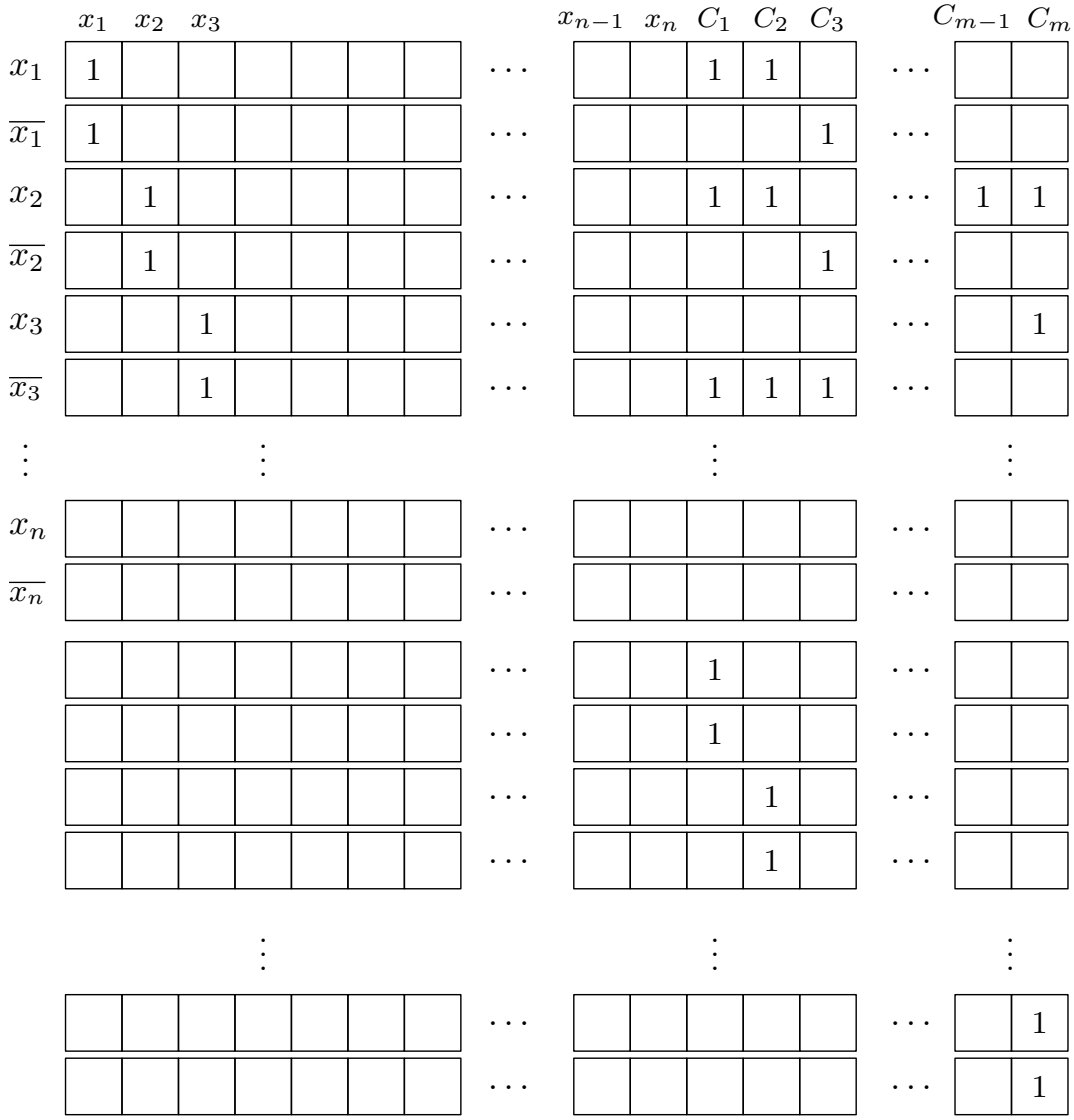
For each of the $2n$ literals the $n+m$-digits number will have a coordinate corresponding to each of the n variables and one coordinate corresponding to each of the m clauses. The literal x_i and $\overline{x_i}$ will have 1 at the digit corresponding to the variable x_i. The digit corresponding to clause C_j will be 1 if and only the literal appears in clause C_j. See the following diagram.



$C_1 = (x_1 \vee x_2 \vee \overline{x_3})$, $C_2 = (x_1 \vee x_2 \vee \overline{x_3})$, $C_3 = (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})$

$C_{m-1} = (x_2 \vee x_8 \vee x_9)$, $C_m = (x_2 \vee x_3 \vee x_5)$

These are $2n$ objects or weights. The remaining $2m$ weights are given as follows. In all of them the first n digits are 0, for each column corresponding to clause C_i we choose two distinct weights and make the corresponding digits equal to 1. See the diagram



Notice that if we consider these weights stacked on top of each other as a table, the first n columns of this table add up to 2 while the last n columns add up to 5 (assuming the formula is a strict 3-CNF formula, i.e. all clauses have exactly 3 literals.)

Let C be the number $\overbrace{111\ldots,11}^{n}\overbrace{333\ldots33}^{m}$

Lemma 62. *The instance of the* SUBSET-SUM *problem with $2n+2m$ objects and weights in decimal as shown above has a subset that sums to $C = \overbrace{111\ldots,11}^{n}\overbrace{333\ldots33}^{m}$ if and*

only if the given formula f is satisfiable.

Proof. If f is satisfiable, select the objects corresponding to the literals that are true in the satisfying assignment. We know that the satisfying assignment must have at least one literal true in each clause. If the satisfying assignment has $1 \leq k \leq 3$ literals set to true for a clause, then we select the $3 - k \leq 2$ other objects corresponding to this clause so as for each. The total weight of select objects is $\overbrace{111\ldots,11}^{n}\overbrace{333\ldots33}^{m}$

The other direction: If there is a selection of objects whose total weight is equal to $C = \overbrace{111\ldots,11}^{n}\overbrace{333\ldots33}^{m}$, then it must select exactly one object out of the two corresponding to each variable x_i, hence we get an assignment by setting the selected literals equal to true. This assignment is satisfying, as this object selection must select 3 rows so that the digits sum corresponding to each clause is 3. This can happen only, if at least one literal in each clause is selected, as the slack objects can only make the column sum equal to 2 □

□

11.7 PARTITION is NP-Complete

Theorem 63. Partition(U, k) *is* NP-Complete

Again first convince yourself, that the Partition(U, k) problem is in NP.

The following lemma prove it NP-Hard.

Lemma 64. Subset-Sum$(U, w, C) \leq_p$ Partition(U, k)

Proof. Let $U = \{a_1, \ldots, a_n\}$ with $w : U \to \mathbb{Z}^+$ and $C \in \mathbb{Z}^+$ be an instance of the Subset-Sum problem. We will make an instance of the partition problem as follows:

Let $U' = \{w(a_1), w(a_2), \ldots, w(a_n), w_{n+1}, w_{n+2}\}$, where $w_{n+1} = 2\left[\sum_{i=1}^{n} w(a_i)\right] - C$ and $w_{n+2} = \left[\sum_{i=1}^{n} w(a_i)\right] + C$.

We claim that Subset-Sum$(U, w, C) =$ **Yes** if and only if Partition$(U', \mathbf{0}) =$ **Yes** (i.e. there is a balanced bipartition of U')

Note that the sum of numbers in U' is

$$\sum_{x \in U'} x = \sum_{a_i \in U} w(a_i) + \underbrace{2\left[\sum_{i=1}^{n} w(a_i)\right] - C}_{w_{n+1}} + \underbrace{\left[\sum_{i=1}^{n} w(a_i)\right] + C}_{w_{n+2}} = 4 \sum_{a_i \in U} w(a_i)$$

If there is a balanced bipartition of U' into P_1 and P_2, then $\sum_{x \in P_1} x = \sum_{x \in P_2} x = 2 \sum_{a_i \in U} w_i$. This implies that both w_{n+1} and w_{n+2} cannot be in the same part, because $w_{n+1} + w_{n+2} = 3 \sum_{a_i \in U}$. WLOG assume that $w_{n+1} \in P_1$ and $w_{n+2} \in P_2$. It is also clear that both P_1 and P_2 cannot contain only one element. By definition, we get that

$$\sum_{x \in P_1 \setminus \{w_{n+1}\}} = C$$

Therefore, there exists a subset of elements of U with their sum equal to C.

P_1		P_2	
$w_{n+1} = 2\sum_i w_i - C$	C	$w_{n+2} = \sum_i w_i + C$	$\sum_i w_i - C$

Similarly, if there is an subset of objects in U such that their total weight is equal to C, then adding w_{n+1} to that subset and keeping all the remaining weights in another set plus w_{n+2} we get a balanced bipartition of U'. □

12 Genres of Problems and other applied Hard Problems

Six basic genres of NP-complete problems and paradigmatic examples.

- Packing problems: Set-Packing, Independent-Set
- Covering problems: Set-Cover, Vertex-Cover
- Constraint satisfaction problems: SAT, 3-SAT
- Sequencing problems: Hamiltonian-Cycle, TSP
- Numerical problems: Subset-Sum, Knapsack, Partition
- Partitioning problems: 3d-Matching 3-Coloring (We didn't do this in lecture slides, but all books have it, please read it there)

- Number Theory problems: FACTOR

There are many other hard problems that are very practical and important in different fields. We list some of them taken from slides for the Kleinberg and Tardos textbook.

- **Aerospace engineering:** optimal mesh partitioning for finite elements.
- **Biology:** protein folding.
- **Chemical engineering:** heat exchanger network synthesis.
- **Civil engineering:** equilibrium of urban traffic flow.
- **Economics:** computation of arbitrage in financial markets with friction.
- **Electrical engineering:** VLSI layout.
- **Environmental engineering:** optimal placement of contaminant sensors.
- **Financial engineering:** find minimum risk portfolio of given return.
- **Game theory:** find Nash equilibrium that maximizes social welfare.
- **Genomics:** phylogeny reconstruction.
- **Mechanical engineering:** structure of turbulence in sheared flows.
- **Medicine:** reconstructing 3-D shape from biplane angiocardiogram.
- **Operations research:** optimal resource allocation.
- **Physics:** partition function of 3-D Ising model in statistical mechanics.
- **Politics:** Shapley-Shubik voting power.
- **Popular culture:** Minesweeper consistency.
- **Statistics:** optimal experimental design.