# Introduction to Computational Complexity Theory

Last Updated March 7th, 2024

## 1 Introduction

**Definition 1.1.** The **Computational complexity** of a computational problem refers to the minimum amount of resources (e.g. execution steps or memory) needed to solve an instance of the problem in relation to its size.

In other words, given problem $A$, we seek functions $s(n)$ and $t(n)$ such that, if $\mathcal{A}$ is an algorithm that solves $A$, then $\mathcal{A}$ will require $\Omega(s(n))$ amount of memory and $\Omega(t(n))$ number of steps to solve an instance of $A$ of size $n$, where $n$ is the (for simplicity, we're assuming only one) size parameter for $A$.

In this chapter we focus almost entirely on decision problems. One reason for this is that the vast majority of problems that are of interest to both computing practitioners and complexity theorists are either decision or optimization problems. And an optimization problem can be readily translated into a decision problem by introducing a nonnegative integer $k$ that represents a threshold for which it must be decided if the property that is being optimized for the problem instance can achieve the given threshold. For example, an instance of optimization problem `Max Clique` is a simple graph $G$, and the problem is to find the the largest clique in $G$. On the other hand, an instance of decision problem `Clique` is a pair $(G, k)$ and the problem is to decide if $G$ has a clique of size $k$. Notice that an algorithm for solving `Max Clique` immediately yields an algorithm for solving `Clique` (why?). Furthermore, if there is an algorithm for solving `Clique` in $\mathrm{O}(t(n))$ steps, then it can be shown that there is also one for solving `Max Clique` in $\mathrm{O}(\log(n)t(n))$ steps.

# 2 Problem Size and Size Parameters

**Definition 2.1.** Given a decision problem $A$ and an instance $x$ of $A$, $|x|$ denotes the **size** of $x$ and equals the number of bits needed to binary-encode $x$. The notation $|x|$ is often useful when speaking abstractly about a generic decision problem, and an instance $x$ of that problem.

**Definition 2.2.** Given a decision problem $A$, a **size parameter** for $A$ is a parameter that may be used to (approximately) represent the size of an instance of $A$. Given an algorithm $\mathcal{A}$ that decides $A$, its size parameters allow one to describe the number of steps (and/or amount of memory) required by $\mathcal{A}$ as a function of the size parameters.

**Example 2.3.** The following are some examples of problems, their size parameters, and examples of how those size parameters are used.

`Clique`
1. Instance: $(G = (V, E), k)$
2. Size parameters: $m = |E|$, $n = |V|$
3. Example: verifying that some $k$ vertices form a clique can be done in $O(n^2)$ steps.

`Subset Sum`
1. Instance: $(S, t)$
2. Size parameters: $n = |S|$, $b$ is the number of bits needed to write $t$ (we assume that $t \geq s$, for all $s \in S$.
3. Example: verifying that a subset of $S$ sums to $t$ can be done in $O(nb)$ steps.

`3SAT`
1. Instance: $\mathcal{C}$
2. Size parameters: $m = |\mathcal{C}|$, $n =$ number of variables of $\mathcal{C}$.
3. Example: verifying that an assignment $\alpha$ satisfies $\mathcal{C}$ can be done in $O(m)$ steps.

`Prime`
1. Instance: $n$
2. Size parameters: $b$ is the number of bits needed to write $n$ in binary. Note that $b = \lfloor \log n \rfloor + 1$.
3. Example: there is an algorithm that can decide `Prime` using $O(b^6)$ steps.

# 3 The Complexity Class P

**Definition 3.1.** A **complexity class** represents a set of decision problems, each of which can be decided by an algorithm that has one or more constraints placed on the resources (i.e. number of computational steps and/or amount of computer memory) that it may use when deciding the problem.

**Definition 3.2.** Decision problem $A$ is a member of complexity class P if there is an algorithm that decides $A$ in a polynomial number of steps with respect to the size parameters of $A$.

Complexity class P is considered robust in the sense that its members tend to remain the same from one model of computation to the next (granted, some models of computation are inherently inefficient, and are not appropriate for use in complexity theory).

**Example 3.3.** Here is a description of some important decision problems that are members of `P`, some of which required an algorithmic breakthrough before acquiring membership.

**Distance between Graph Vertices** Given weighted graph $G = (V, E, w)$, vertices $a, b \in V$, and integer $k \geq 0$, is it true that the distance from $a$ to $b$ does not exceed $k$? Dijkstra's algorithm solves this problem in $O(m \log n)$ steps.

**Primality Test** Given natural number $n \geq 2$ is $n$ prime? (see "PRIMES is in P", Annals of Mathematics, Pages 781-793 from Volume 160 (2004), Issue 2 by Manindra Agrawal, Neeraj Kayal, Nitin Saxena). The algorithm requires $O(\log^6 n)$ steps.

**Linear Optimization** Given i) function $f(x) = cx$, for some $1 \times n$-dimensional constant matrix $c$ and $n \times 1$ real-valued matrix $x$, ii) constant $k \in \mathcal{R}$, iii) $m \times n$ constant matrix $A$, and iv) $m \times 1$ constant matrix $b$, is it true that there is an $x$ for which

$$f(x) = cx \leq k,$$

subject to

$$Ax \geq b?$$

Karmarkar's algorithm solves this problem in $O(n^{3.5}L^2 \cdot \log L \cdot \log(\log L))$ steps, where $n$ is the number of problem variables, and $L$ is the number of bits needed to encode an problem instance.

**Maximum Flow** Given directed network $G = (V, E, c, s, t)$ and integer $k \geq 0$, is there a flow from $s$ to $t$ of size at least $k$? The Ford Fulkerson algorithm solves this problem in $O(n^3)$ steps.

**Perfect Matching** Given bipartite graph $G = (U, V, E)$, where $|U| = |V| = n$, does $G$ have a **perfect matching**, i.e. a set of edges $M = \{e_1, \ldots, e_n\} \subseteq E$ such that any two edges $e_i, e_j \in M$ neither share a vertex in $U$, nor share a vertex in $V$? The Ford Fulkerson algorithm solves this problem in $O(n^2 + mn)$ steps.

**2SAT** Given a set of Boolean formulas $\mathcal{C}$, where each formula (called a **clause**) has the form $a \vee b$, where $a$ and $b$ are literals, is there a truth assignment for the variables so that each clause has at least one literal that is assigned `true`? This problem can be solved in $O(m + n)$ steps.

**Bitonic Traveling Salesperson** given $n$ cities $c_1, \ldots, c_n$, where $c_i$ has grid coordinates $(x_i, y_i)$, and a cost matrix $C$, where entry $C_{ij}$ denotes the cost of traveling from city $i$ to city $j$, determine a left-to-right followed by right-to-left Hamilton-cycle tour of all the cities that minimizes the total traveling cost. In other words, the tour starts at the leftmost city, proceeds from left to right visiting a subset of the cities (including the rightmost city), and then concludes from right to left visiting the remaining cities. The problem can be solved in $O(n \log^2 n)$ steps.

Figure 1: Solving an NP problem can be like finding a needle in a haystack.

# 4 The Complexity Class NP

**Definition 4.1.** Decision problem $A$ is a member of complexity class NP if there is

1. a set Cert, called the **certificate set**,
2. a decision algorithm V, called the **verifier**, which has the following properties:
   (a) the inputs to V are i) an instance $x$ of $A$ and ii) a certificate $c \in$ Cert
   (b) the output is 1 iff $c$ is a **valid certificate** for $x$, meaning that $c$ proves that $x$ is a positive instance of $A$
   (c) V requires a polynomial number of steps with respect to the size parameters of $A$.

Although, for any given instance $x$ of $A$ and any certificate $c \in$ Cert, the verifier only requires a polynomial number of steps, what makes some NP problems very difficult to solve is that there are usually an exponential number of certificates, and finding a valid one is like finding a "needle-in-a-haystack" because there is no apparent way to avoid having to examine an exponential (in the size parameters of $A$) number of certificates.

**Example 4.2.** We show that `Clique` $\in$ `NP`. Let $(G = (V, E), k)$ be a problem instance for `Clique`.

**Step 1: define a certificate**. Certificate $C$ is a subset of $V$ where $|C| = k$.

**Step 2: provide a semi-formal verifier algorithm**.

For each $u \in C$,

For each $v \in C$ with $u \neq v$,

If $(u, v) \notin E$, then return 0.

Return 1.

**Step 3: size parameters for `Clique`**. $m = |E|$ and $n = |V|$.

**Step 4: provide the verifier's running time with an explanation**.

The nested `for`-loops require at most $k^2 = \mathrm{O}(n^2)$ query to determine if a pair of vertices $(u, v)$. Each query can be answered using a hash table that stores the graph edges. Building such a table takes $\Theta(m)$ steps. Thus, algorithm's total number of steps is $\mathrm{O}(m + n^2) = \mathrm{O}(n^2)$ steps, which is quadratic in the size of $(G, k)$.

**Example 4.3.** By repeating the steps of Example 4.2, prove that `Subset Sum` $\in$ `NP`. Let $(S, t)$ be an instance of `Subset Sum`.

**Step 1: define a certificate**.

**Step 2: provide a semi-formal verifier algorithm**.

**Step 3: provide size parameters for** `Subset Sum`.

**Step 4: provide the verifier's running time with an explanation**.

**Example 4.4.** The `3-Dimensional Matching (3DM)` decision problem takes as input three sets $A$, $B$, and $C$, each having size $n$, along with a set $S$ of triples of the form $(a, b, c)$ where $a \in A$, $b \in B$, and $c \in C$. We assume that $|S| = m \geq n$. The problem is to decide if there exists a subset $T \subseteq S$ of $n$ triples for which each member from $A \cup B \cup C$ belongs to exactly one of the triples.

Show that $(A, B, C, S)$ is a positive instance of `3DM`, where $A = \{a, b, c\}$, $B = \{1, 2, 3\}$, $C = \{x, y, z\}$, and

$$S = \{(a, 1, x), (a, 2, z), (a, 3, z), (b, 1, x), (b, 2, x), (b, 3, z), (c, 1, x), (c, 2, z), (c, 3, y)\}.$$

**Solution.**

**Example 4.5.** By repeating the steps of Example 4.2, prove that 3DM $\in$ NP. Let $(A, B, C, S)$ be an instance of 3DM.

**Step 1: define a certificate**.

**Step 2: provide a semi-formal verifier algorithm**.

**Step 3: provide size parameters for** 3DM.

**Step 4: provide the verifier's running time with an explanation**.

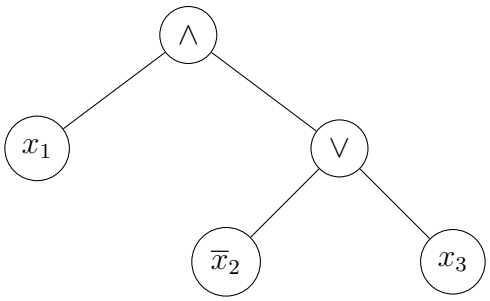



Figure 2: Parse tree for Boolean formula $x_1 \wedge (\overline{x}_2 \vee x_3)$

## 4.1 The Satisfiability (SAT) problem is in NP

A **Boolean formula** $F(x_1, \ldots, x_n)$ over variable set $V = \{x_1, \ldots, x_n\}$ may be represented with a **parse tree** for which i) each internal node is labeled either $\wedge$ or $\vee$, and ii) each leaf node is labeled either $x_i$ or $\overline{x}_i$, for some $i = 1, \ldots, n$. Leaf nodes are also called **literal** nodes, since a formula literal is any variable or its negation. For example, the Boolean formula

$$F(x_1, x_2, x_3) = x_1 \wedge (\overline{x}_2 \vee x_3)$$

may be represented by the **parse tree** shown in Figure 2.

**Definition 4.6. assignment** over variable set $V$ is a function $\alpha : V \to \{0, 1\}$ that assigns to each variable $x \in V$ a member of $\{0, 1\}$. We may represent $\alpha$ using function notation, or as a labeled tuple. Indeed, for the assignment $\alpha$ that assigns 1 to both $x_1$ and $x_2$, and 0 to $x_3$, we may use function notation and write $\alpha(x_1) = 1$, $\alpha(x_2) = 1$, and $\alpha(x_3) = 0$, or we may use tuple notation and write $\alpha = (x_1 = 1, x_2 = 1, x_3 = 0)$.

Given Boolean formula $F$ and assignment $\alpha$ over $V$ we may evaluate $F$ using the function $\text{eval}(F, \alpha)$ that returns a value in $\{0, 1\}$. We provide a recursive definition of $\text{eval}(F, \alpha)$ over the set of all Boolean formulas defined over $V$.

**Base Case** If $F$ consists of a leaf node labeled with literal $l$ (i.e., $x$ or $\overline{x}$ for some variable $x$), then

$$\text{eval}(F, \alpha) = \alpha(l).$$

**Recursive Case (And)** If the root of $F$ is labeled $\wedge$, and $C_1, \ldots, C_m$ are the root children, then

$$\text{eval}(F, \alpha) = \text{eval}(C_1, \alpha) \wedge \cdots \wedge \text{eval}(C_m, \alpha).$$

**Recursive Case (Or)** If the root of $F$ is labeled $\vee$, and $C_1, \ldots, C_m$ are the root children, then

$$\text{eval}(F, \alpha) = \text{eval}(C_1, \alpha) \vee \cdots \vee \text{eval}(C_m, \alpha).$$

It is worth noting that $\text{eval}(F, \alpha)$ may be computed in $O(|F|)$ steps, where $|F|$ denotes the number of nodes in formula $F$.

**Example 4.7.** Use the recursive definition of `eval` to evaluate the formula

$$F(x_1, x_2, x_3) = ((x_1 \vee x_3) \wedge (\overline{x}_1 \vee x_2)) \vee ((\overline{x}_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee x_3))$$

over the assignment $\alpha = (1, 0, 1)$. Demonstrate how the number of evaluation steps is proportional to the number of nodes in the tree representation of $F$.

**Example 4.8.** The **satisfiability problem (SAT)** is the problem of deciding if a Boolean formula $F(x_1, \ldots, x_n)$ evaluates to 1 on some assignment $\alpha$ over the Boolean variables $x_1, \ldots, x_n$. We show that $\texttt{SAT} \in \texttt{NP}$. Let $F(x_1, \ldots, x_n)$ be an instance of SAT.

**Step 1: define a certificate**. **Solution.** $\alpha$ is an assignment over the variables $x_1, \ldots, x_n$.

**Step 2: provide a semi-formal verifier algorithm**. **Solution.** Evaluate $F(x_1, \ldots, x_n)$ recursively.

//Base Case:

If $F = l$ is a single literal, then return $\alpha(l)$.

//Recursive Case 1

If $F = F_1 \wedge F_2 \wedge \cdots \wedge F_k$, then return

$$\text{eval}(F_1, \alpha) \wedge \text{eval}(F_2, \alpha) \wedge \cdots \wedge \text{eval}(F_k, \alpha).$$

//Recursive Case 2

If $F = F_1 \vee F_2 \vee \cdots \vee F_k$, then return

$$\text{eval}(F_1, \alpha) \vee \text{eval}(F_2, \alpha) \vee \cdots \vee \text{eval}(F_k, \alpha).$$

**Step 3: provide size parameters for SAT**. **Solution.** The size parameter is $|F|$, the number of nodes in $F$'s parse tree.

**Step 4: provide the verifier's running time with an explanation**. **Solution.** The verifier has running time $O(|F|)$, since evaluating $F$ can be done by evaluating each node of $F$'s parse tree exactly once.

Therefore, $\texttt{SAT} \in \texttt{NP}$. □

**Theorem 4.9.** $\mathtt{P} \subseteq \mathtt{NP}$.

**Proof.** Let $L \in \mathtt{P}$ be a decision problem that can be decided in polynomial time. Then there is a polynomial-time computable predicate function $D(x)$ that decides $L$. We now show that $L \in \mathtt{NP}$ by repeating the steps of Example 4.2.

**Step 1:** define a certificate. **Solution.** Since $L \in \mathtt{P}$, a verifier for $L$ does not require a certificate, since it can run predicate function $D(x)$ in polynomial time to determine if $L$-instance $x$ is positive. However we must follow the definition of being in $\mathtt{NP}$. So we make a "dummy" certificate set $C = \{1\}$ consisting of a single member which the verifier can ignore.

**Step 2:** provide a semi-formal verifier algorithm. **Solution.**

// A one line program:

Return $D(x)$.

**Step 3:** provide size parameters for $L$. **Solution.** Since $L$ is a generic problem, we let $|x|$ denote the size of $L$-instance $x$.

**Step 4:** provide the verifier's running time and defend your answer. **Solution.** Since $L \in \mathtt{P}$, predicate function $D(x)$ may be computed in $\mathrm{O}(p(|x|))$ steps, for some polynomial $p$.

Therefore, $L \in \mathtt{NP}$. □

Do there exist decision problems that are in NP but not in P? This would imply that some problems have solutions that can be verified in polynomial time, but not solved in polynomial time. Moreover, NP problems such as Clique, Subset Sum, and 3DM are candidates since, at this writing, polynomial-time algorithms for these problems have yet to be established. But at the same time a proof that such algorithms do not exist has yet to be found.

To better understand the difficulty faced with resolving the P =? NP question, consider the SAT problem. Several algorithms have been designed for solving instances of SAT and, although none of them appear to run in polynomial-time, for some there is no proof that it does *not* run in polynomial time. Moreover, it may be possible to design an "algorithm cocktail" that combines the best SAT algorithms in a way that solves each of the "hard" instances in polynomial time where the polynomial may have a very high degree. In other words, it's possible that two things could be true at the same time:

1. P = NP
2. but some instances of SAT require an exorbitant (albeit polynomial) amount of computational resources to solve, both now and in the forseeable future.

The P=?NP problem is considered one of the most challenging and important in all of computer science and mathematics. The Clay Mathematics Institute is awarding a prize of $1 million dollars to anyone who can resolve this problem.

# 5 Polynomial-time Mapping Reducibility

**Definition 5.1.** Decision problem $A$ is **polynomial-time mapping reducible** to decision problem $B$, written $A \leq_m^p B$, iff there exists a computable function $f : A \to B$ for which $x$ is a positive instance of $A$ iff $f(x)$ is a positive instancce of $B$ and $f(x)$ may computed in a number of steps that is bounded by a polynomial in the size parameters of $A$.

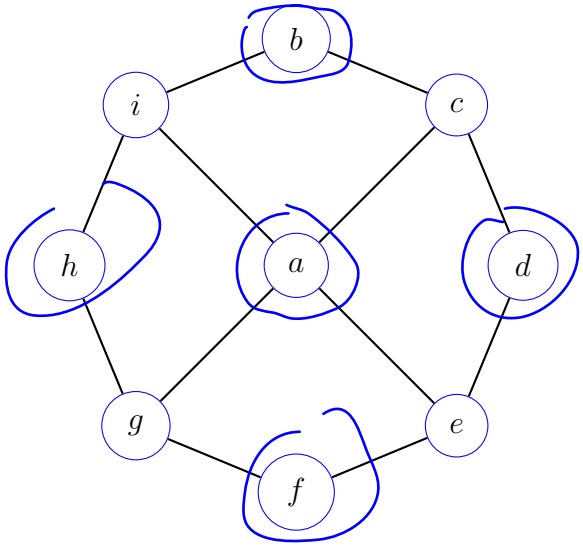



Figure 3: $I = \{a, b, d, f, h\}$ is an independent set for the above graph

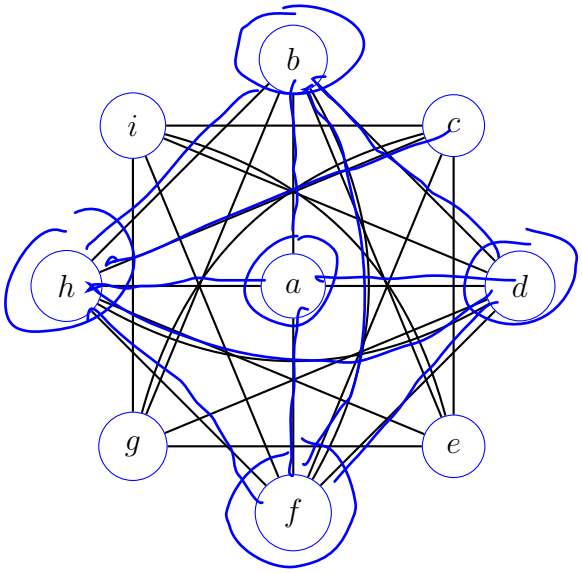



Figure 4: The complement graph $\overline{G}$ for the graph $G$ shown in Figure 3

**Example 5.2.** Recall the `Clique` decision problem defined in Example 2.3. A related problem is the `Independent Set (IS)` optimization problem for which a problem instance is a simple graph $G = (V, E)$ and a nonnegative integer $k \geq 0$, and the problem is to decide if $G$ has an **independent set**, i.e. a subset $I \subseteq V$ of $k$ vertices such that, for every $u \in I$ and $v \in I$, $(u, v) \notin E$. In other words, every pair of vertices $u, v \in I$ must be non-adjacent. Figure 3 shows the graph of a positive instance $(G, k = 5)$ of `IS`, since $G$ has the independent set $I = \{a, b, d, f, h\}$.

We now provide a map reduction from `IS` to `Clique`. Given a simple graph $G = (V, E)$, our reduction makes use of the **complement** of $G$, denoted $\overline{G}$, which is defined as $\overline{G} = (V, \overline{E})$, where, for any two vertices $u, v \in V$, $(u, v) \in \overline{E}$ iff $(u, v) \notin E$. In other words, $G$ and its complement $\overline{G}$ have the same vertex set, but the edges of $\overline{G}$ are exactly those edges that are *not* edges of $G$, and vice versa. For example, the Graph is Figure 4 shows the complement of the graph $G$ in Figure 3.

Notice that the independent set $I = \{a, b, d, f, h\}$ for $G$ now represents a 5-clique in $\overline{G}$. In fact, this is true for *any* independent set $I$ for $G$: $I$ is an independent set for $G$ iff $I$ is a clique for $\overline{G}$. Thus, we may provide a map reduction $f : \texttt{IS} \to \texttt{Clique}$ that is defined by $f(G, k) = (\overline{G}, k)$. Indeed, $G$ has an independent set of size $k$ iff $\overline{G}$ has a $k$-clique.

Finally, we have the following two facts.

1. the map reduction is a polynomial-time reduction, since constructing $\overline{G}$ from $G$ requires at most $\mathrm{O}(n^2)$ steps (why?), and
2. we could just as easily have defined the same reduction from `Clique` to `IS`.

## 5.1 Embeddings

**Definition 5.3.** An **embedding reduction** is a kind of mapping reduction for which a problem $A$ is map reduced to problem $B$, where $A$ is a special case of $B$.

As an example, consider a vehicle that has an air-conditioning system that includes the ability to cool and heat different parts of the vehicle, as well as control the flow of air entering and leaving the vehicle. Recall our friend Sam from the Turing Reducibility lecture. Sam and his friends drove to the Mohave desert where they encountered extreme heat, lots of wind, and some very dusty roads. While driving, they reduced the problem of staying cool and breathing clean air to setting the vehicle's air-conditioner to "cool" and allowing no outside air to enter the vehicle. In other words, the general air-conditioning system served the special purpose of cooling and maintaining air cleanliness.

**Definition 5.4.** The `Subset Sum (SS)` decision problem is a pair $(S, t)$, where $S$ is a set of nonnegative integers, and $t$ is a nonnegative integer. The problem is to decide if there is a subset $A \subseteq S$ whose members sum to $t$, i.e., a subset $A$ for which

$$\sum_{a \in A} a = t.$$

**Example 5.5.** `Subset Sum` instance $(S = \{3, 7, 13, 19, 22, 26, 35, 38, 41\}, t = 102)$ is a positive instance of `SS` since $A = \{3, 7, 13, 38, 41\} \subseteq S$ and

$$3 + 7 + 13 + 38 + 41 = 102.$$

**Definition 5.6.** An instance of decision problem `Set Partition (SP)` consists of a set $S$ of positive integers, and the problem is to decide if there are subsets $A, B \subseteq S$ for which

1. $A \cap B = \emptyset$,
2. $A \cup B = S$, and
3. $$\sum_{a \in A} a = \sum_{b \in B} b.$$

In other words, the members of $A$ must sum to the same value as the members of $B$.

**Example 5.7.** Verify that `Set Partition` instance

$$S = \{3, 14, 19, 26, 35, 37, 43, 49, 52\}$$

is a positive instance of `SP` via $A = \{3, 35, 49, 52\}$ and $B = \{14, 19, 26, 37, 43\}$.

A few moments of thought should convince you that `SP` is a special case of `SS`. Indeed given instance $S$ of `SP`, if we let $M$ denote the sum of all members of $S$, then we essentially are seeking a subset $A$ whose members sum to $M/2$ since, if such $A$ can be found, then by setting $B = S - A$, we have all three conditions met (check this!). Therefore, we may reduce `SP` to `SS` via the map

$$f(S) = (S, t = M/2), \tag{1}$$

where $M$ is defined as above.

Just as Sam and his friends don't care about the heating features of the air-conditioning, when solving an instance of `Set Partition` via `Subset Sum`, we don't care that `Subset Sum` can solve a wider variety of problems involving $t$ values that may not have any relationship with $S$.

**Example 5.8.** Given the instance $S = \{4, 8, 17, 25, 34, 46, 53, 59\}$ of `Set Partition`, provide the instance of `SS` to which it reduces via the mapping defined in equation 1.

**Solution.** $f(S, t = \sum_{s \in S} s / 2)$

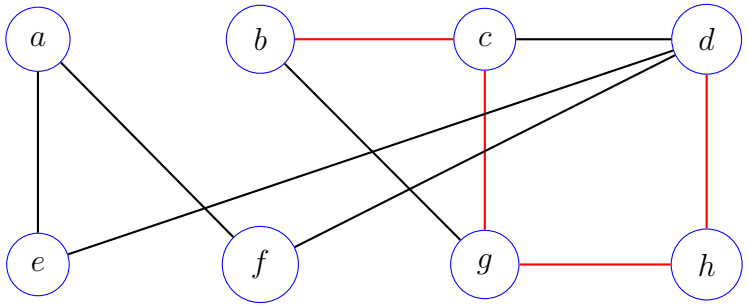
$$M = \sum_{s \in S} s = 246$$

$$A \cup B = S$$

$$\sum_{a \in A} a = \sum_{b \in B} b = \frac{246}{2} = 123$$

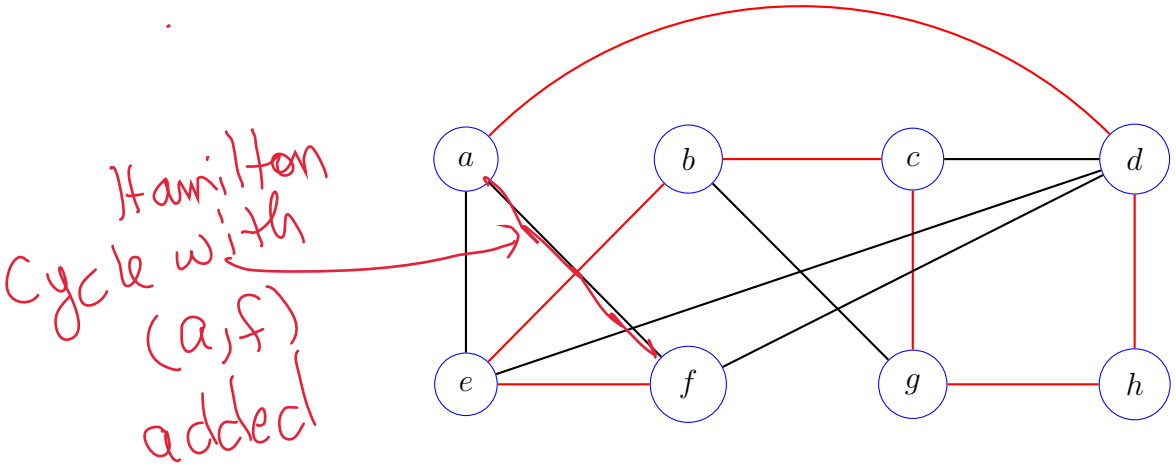
**Definition 5.9.** An instance of the `LPath` decision problem is a pair $(G, k)$, where $G = (V, E)$ is a simple graph, and $k \geq 0$ is a nonnegative integer. The problem is to decide if $G$ has a **simple path** of length $k$, i.e. a path that traverses $k$ edges and visits exactly $k + 1$ different vertices.

**Example 5.10.** The following graph, along with $k = 4$, shows a positive instance of LPath.



**Definition 5.11.** An instance of the `Hamilton Path (HP)` decision problem consists of a simple graph $G = (V, E)$ and the problem is to decide if $G$ has a **Hamilton path**, namely a path that visits every vertex in $G$ exactly once.

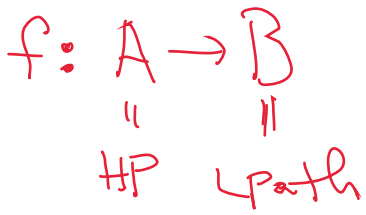
**Example 5.12.** The following graph represents a positive instance of `HP`, with the Hamilton Path shown in red.



Once again, it is hopefully evident that HP is a special case of LPath. To see this, note that a simple graph has a Hamilton path iff it has a simple path of length $n - 1 = |V| - 1$. Therefore, we may reduce HP to LPath via the map

$$f(G) = (G, |V| - 1).$$

$\square$



## 5.2 Contractions

As the examples in the previous section suggest, devising an embedding reduction from problem $A$ only requires knowledge of a problem $B$ that is more general than $A$ and includes $A$ as a special case. On the other hand, a **contraction reduction** is a mapping reduction for which problem $B$ is the special case of problem $A$. Defining a contraction usually involves some insight and imagination. Moreover, a contraction reduction demonstrates actual computing progress in the sense that a body $A$ of problem instances gets effectively reduced to a smaller set $B$ which may improve the chances of efficiently solving $A$ through the lens of $B$.

**Example 5.13.** Contractions are quite common in everyday computing. As an example, consider a legacy compiler $\mathcal{C}$ for some programming language $L$. Over the years $L$ gets extended with the addition of new programming constructs that improve the ease of programming in $L$. We'll call this extended version Turbo-$L$. Rather than write a new compiler for Turbo-$L$, we instead provide a contraction that is capable of translating any Turbo-$L$ program to an $L$ program, followed by compiling the $L$-code with the legacy compiler. □

$L\ Path \leq_m^P$ $SAT \leq_m^P \leq_m^P$

$3SAT \leq_m^P$

$DHP \leq_m^P HP$

Let's return to the `Subset Sum (SS)` and `Set Partition (SP)` problems defined in Section 5.1. There we showed a natural embedding from `SP` to `SS`. Now we show a contraction from `SS` to `SP`. To see how the contraction works, consider the set

$$S = \{7, 12, 15, 23, 37, 42, 48\}.$$

The sum of its members is $M = 184$. Therefore, if $(S, t)$ is an instance of `SS` with $t = 184/2 = 92$, then, by dropping $t$, this instance naturally maps to `SP`:

$$(S, t) \rightarrow S,$$

and $(S, t)$ is positive for `SS` iff $S$ is positive for `SP`.

What if $t$ is less than half of $M$? Then we may not simply drop $t$, but will also have to modify $S$. But how? As an example, suppose that $t = 64 < 92$. Then $(S, t)$ is a positive instance of `SS` via $A = \{12, 15, 37\}$. Notice also that the members of the complement $B = S - A$ sums to 120. Therefore, by adding to $S$ the extra number $J = 56$, we see that the members of

$$A \cup \{56\} = \{12, 15, 37, 56\},$$

also sum to 120 and thus $S' = S \cup \{56\}$ is a positive instance of `SP`.

Let's generalize the example from the last paragraph. Assume that $t < M/2$. Then we must add the number $J$ to $S$ for which

$$t + J = M - t \Rightarrow J = M - 2t.$$

Now suppose $(S, t)$ is positive for `SS` via $A \subseteq S$. Let $A' = A + \{M - 2t\}$ and $B' = S - A$. Then we have

1. $A' \cap B' = \emptyset$,
2. $A' \cup B' = S' = S + \{M - 2t\}$, and
3. the members of $A'$ sum to $t + (M - 2t) = M - t$ while the members of $B'$ also sum to $M - t$.

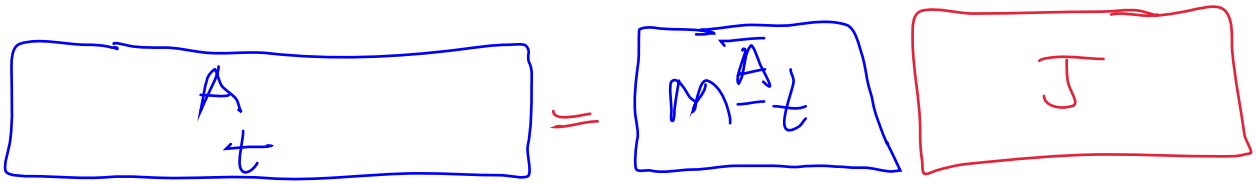
Therefore, $S' = S \cup \{M - 2t\}$ is a positive instance of `SP`.

We now show that the converse is also true: if $S' = S \cup \{M - 2t\}$ is positive for `SP`, then there are subsets $A'$ and $B'$ of $S'$ for which

1. $A' \cap B' = \emptyset$,
2. $A' \cup B' = S'$,
3. $(M - 2t) \in A'$, and
4. the members of $A = A' - \{M - 2t\}$ sum to
$$1/2(M + (M - 2t)) - (M - 2t) = (M - t) - (M - 2t) = t.$$

And since the members of $A = A' - \{M - 2t\}$ are all members of $S$, we see that $(S, t)$ is positive for `SS`.

**Example 5.14.** Derive the formula for $J$ in case $t > M/2$.




$$t = M - t + J \Rightarrow$$

$$J = 2t - M$$

**Example 5.15.** Use the above reduction to map the following instances of `Subset Sum` to `Set Partition`.

$S$ $\quad M = \sum_{s \in S} s = 264 \quad M/2 = 132$

1. $(\{7, 11, 23, 25, 37, 39, 49, 73\}, t = 51)$
2. $(\{7, 11, 23, 25, 37, 39, 49, 73\}, t = 132)$
3. $(\{7, 11, 23, 25, 37, 39, 49, 73\}, t = 200)$

1. $f(S, t=51) = S \cup \{J = M - 2t\} = S \cup \{162\}$

2. $t = \frac{M}{2} \Rightarrow f(S, t) = S$

3. $t > \frac{M}{2} \Rightarrow f(S, t) = S \cup \{J = 2t - M\} = S \cup \{136\}$

## 5.3 Implications of mapping reducibility on complexity

**Theorem 5.16.** If $A \leq_m^p B$, where $A$ and $B$ are two decision problems. Then the following statements hold.

1. If $B$ is decidable in $O(p(m))$ steps, for some polynomial $p$, where $m$ is the size parameter for $B$, then $A$ is decidable in $O(r(n))$ steps, for some polynomial $r(n)$, where $n$ is the size parameter for $A$.
2. If $A$ cannot be solved in $O(r(n))$ steps, for any polynomial $r(n)$, then $B$ cannot be solved in $O(p(m))$ steps for any polynomial $p(m)$.

**Proof.** Notice that the second statement is the contrapositive of the first, and so it suffices to prove the first. To this end, assume $B$ is decidable in $O(p(m))$ steps via some algorithm $\mathcal{B}$. Since $A \leq_m^p B$, there is a map $f : A \to B$ that is computable in $O(q(n))$ steps for some polynomial $q(n)$, where $n$ is the size parameter for $A$. Moreover, the decision for instance $a$ of $A$ equals the decision for instance $f(a)$ of $B$. Therefore, an algorithm for solving $A$ first computes instance $f(a)$ via the algorithm that computes $f$, and then applies algorithm $\mathcal{B}$ to instance $f(a)$. Now, since $f(a)$ is computable in $O(q(n))$ steps, we may assume that the size of $f(a)$ is $m = O(q(n))$ bits, i.e. each algorithm step can construct at most $O(1)$ bits of the output. Thus, $\mathcal{B}$ runs on input $f(a)$ whose size is $m = O(q(n))$ bits which means the algorithm's running time is $O(p(q(n)))$. Thus, the total running time for solving instance $a$ is

$$O(q(n)) + p(q(n))) = O(p(q(n))),$$

and so we may set $r(n) = q(p(n))$. □

**Example 5.17.** Suppose $f$ map reduces $A$ to $B$ and is computable in $O(n^3)$ steps. Furthermore, suppose algorithm $\mathcal{B}$ solves an instance of $B$ in $O(m^4)$ steps. Determine the running time of the following algorithm that solves $A$.

n: size parameter for A
m: " " B

**Algorithm for Solving** $A$

Input: instance $a$ of $A$.

Output: solution to $a$.

Compute $b = f(a)$ which is an instance of $B$. // $O(n^3)$

Return $\mathcal{B}(b)$. // $O((n^3)^4) = O(n^{12})$

Complexity of Algorithm

$O(n^3 + n^{12}) = O(n^{12})$

# 6 NP-Complete Decision Problems

Now that we have an idea about the type of decision problems that belong in NP, we would like a method for demonstrating that a decision problem is in some sense one of the *hardest* amongst all problems in NP, and thus is the best candidate for not belonging to class P. Furthermore, if we consider what might constitute a difficult problem amongst a class of problems, the most difficult would seem to be one to which every other problem in the class can be reduced.

**Definition 6.1.** A decision problem $B$ is said to be **NP-complete** iff

1. $B \in \mathtt{NP}$
2. for every other decision problem $A \in \mathtt{NP}$, $A \leq_m^p B$.

**Theorem 6.2.** (Cook's Theorem) SAT is NP-complete.

### Outline of a Proof of Cook's Theorem

1. Let $L \in \texttt{NP}$ be an arbitrary decision problem and $x$ an instance of $L$.
2. Let $v(x, c)$ be the verifier program associated with $L$.
3. Let $q(|x|)$ denote the running time for $v(x, c)$, where $q$ is a polynomial.
4. Let variables $y_1, \ldots, y_{l(|x|)}$ be a collection of Boolean variables that is capable of encoding any certificate $c$ for verifying $x$, where $l$ is a polynomial (one of the requirements of a certificate is that its size must be polynomial with respect to $|x|$).
5. It can be shown that any program that runs in a polynomial $q(|x|)$ number of steps and depends on a polynomial $l(|x|)$ number of Boolean variables, can be procedurally converted in polynomial time to a Boolean formula

   $$F_{v,x}(y_1, \ldots, y_{l(|x|)}),$$

   $y \in C$ is a certificate

   where

   (a) $F_{v,x}$ is satisfiable iff $v(x, c)$ evaluates to 1 for some certificate $c$, iff $x$ is a positive instance of $L$, and

   (b) $|F_{v,x}|$ is bounded in size by some polynomial in terms of $|x|$, the size of $x$.

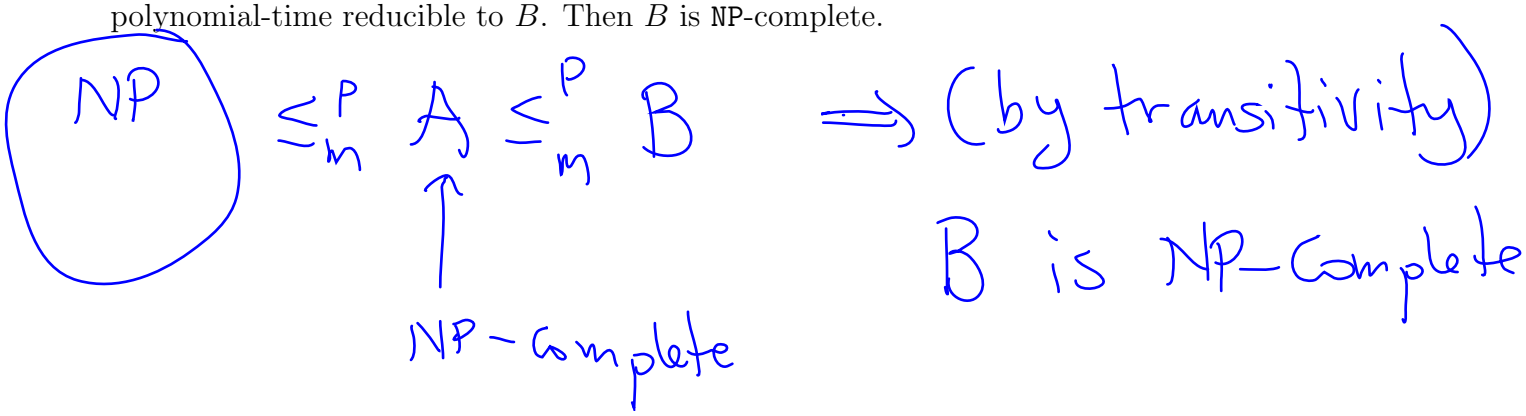
   Therefore,

   $$L \leq_m^p \texttt{SAT}.$$

# 7 More NP-Complete Problems

We now build on Cook's theorem to show that a host of other problems are NP-complete (to this date there are several thousand known NP-complete problems from several areas of computer science and mathematics). We do this with the help of the following lemma. Its proof relies on the fact that mapping reducibilities are transitive: if $A \leq_m^p B$ and $B \leq_m^p C$, then $A \leq_m^p C$.

**Lemma 7.1.** Suppose decision problems $A$ and $B$ are in NP, and $A$ is both NP-complete and polynomial-time reducible to $B$. Then $B$ is NP-complete.



## 7.1 The 3SAT Logic Problem

In this Section we introduce the 3SAT logic problem which will which represents one of the more important problems in all of Computer Science.

## 7.2 Boolean Variable Assignments

Before introducing the 3SAT decision problem, we need to understand the concept of a Boolean variable assignment.

**Boolean Variable** A variable is said to be **Boolean** iff its domain equal $\{0, 1\}$. We use lowercase letters, such as $x, y, z, x_1, x_2, \ldots$, etc., to denote a Boolean variable.

**Assignment** An **assignment** over a Boolean-variable set $V$ is a function $\alpha : V \to \{0, 1\}$ that assigns to each variable $x \in V$ a value in $\{0, 1\}$. We may represent $\alpha$ using function notation, or as a labeled tuple.

**Example:** for the assignment $\alpha$ that assigns 1 to both $x_1$ and $x_2$, and 0 to $x_3$, we may use function notation and write $\alpha(x_1) = 1$, $\alpha(x_2) = 1$, and $\alpha(x_3) = 0$, or we may use tuple notation and write

$$\alpha = (x_1 = 1, x_2 = 1, x_3 = 0),$$

or

$$\alpha = (1, 1, 0),$$

if the associated variables are understood.

**Variable Negation** If $x$ is a variable, then $\overline{x}$ is called its **negation**.

**Example:** Suppose assignment $\alpha$ satisfies $\alpha(x_1) = 0$. Then (extending $\alpha$ to include negation inputs) $\alpha(\overline{x}_1) = 1$.

**Literal** A **literal** is either a variable or the negation of a variable .

**Example:** $x_1$, $x_3$, $\overline{x}_3$, are $\overline{x}_5$ all examples of literals.

**Consistent** A set $R$ of literals is called **consistent** iff no variable and its negation are both in $R$. Otherwise, $R$ is said to be **inconsistent**.

**Example:** $\{x_1, \overline{x}_2, x_4, \overline{x}_7, \overline{x}_9\}$ is a consistent set, but $\{x_1, \overline{x}_2, x_4, \overline{x}_7, x_7\}$ is an inconsistent set.

**Induced Assignment** If $R = \{l_1, \ldots, l_n\}$ is a consistent set of literals, then $\alpha_R$ is called the **(partial) assignment induced by R** and is defined by $\alpha(l_i) = 1$ for all $l_i \in R$.

**Example:** the assignment induced by $R = \{x_1, \overline{x}_2, x_4, \overline{x}_7, \overline{x}_9\}$ is

$$\alpha = (x_1 = 1, x_2 = 0, x_4 = 1, x_7 = 0, x_9 = 0).$$

**Definition 7.2.** A **ternary disjunctive clause** is a Boolean formula of the form

$$l_1 \vee l_2 \vee l_3,$$

where $l_1$, $l_2$, and, $l_3$ are literals. The clause evaluates to 1 in case at least one of $l_1$, $l_2$, or $l_3$ is assigned 1. In this case we say the clause is **satisfied**. Otherwise it is **unsatisfied**.

**Definition 7.3.** An instance of the `3SAT` decision problem consists of a set $\mathcal{C}$ of ternary disjunctive clauses. The problem is to decide if there is an assignment $\alpha$ over the variables in $\mathcal{C}$, such that every clause $(l_1 \vee l_2 \vee l_3)$ in $\mathcal{C}$ evaluates to 1 under $\alpha$. If such an assignment $\alpha$ exists, then it is said to be a **satisfying assignment** and we say $\mathcal{C}$ is **satisfiable**. Otherwise, $\mathcal{C}$ is said to be **unsatisfiable**. Finally, the `3SAT` decision problem is the problem of deciding whether a set $\mathcal{C}$ of ternary clauses is sastisfiable.

**Simplified clause notation.** In what follows, we often simplify the clause notation by writing each clause $(l_1 \vee l_2 \vee l_3)$ as $(l_1, l_2, l_3)$.

**Example 7.4.** Provide a satisfying assignment for

$$\mathcal{C} = \{(x_1, x_2, x_3), (\overline{x}_2, x_3, \overline{x}_4), (x_1, x_2, \overline{x}_4), (\overline{x}_1, \overline{x}_3, \overline{x}_4), (\overline{x}_1, x_2, x_4), (\overline{x}_2, x_3, x_4), (x_1, \overline{x}_3, x_4),$$
$$(\overline{x}_2, \overline{x}_3, x_4), (\overline{x}_2, \overline{x}_3, \overline{x}_4)\}.$$

3SAT Notation

$\alpha = (0, 0, 1, 0)$

$\alpha = 1001$

**Theorem 7.5.** SAT $\leq_m^p$ 3SAT.

**Proof.** We prove the theorem by making use of what is referred to as the **Tseytin transformation**, named after the Russian mathematician Gregory Tseytin. It is a method for transforming an instance $F$ of `SAT` to an instance $\mathcal{C}$ of `3SAT` that is satisfiability-equivalent to $F$, meaning that $F$ is satisfiable iff $\mathcal{C}$ is satisfiable. Let $F(x_1, \ldots, x_n)$ be an instance of SAT. Without loss of generality, we may assume that $F$ has a binary parse tree $T$. Let $n_1, \ldots, n_m$ denote the internal nodes of $T$, where we assume $n_1$ correpsonds with the root. We assign a literal to each tree node. If $n$ is a leaf, then the literal assigned to $n$ is the literal $l$ for which $n$ is labeled. If $n = n_i$ is an internal node, then we introduce a new variable $y_i$ and associate it with $n_i$.

Thus, the reduction $f$ from `SAT` to `3SAT` is such that $f(F) = \mathcal{C}$, and the variables used in the clauses of $\mathcal{C}$ are precisely $x_1, \ldots, x_n, y_1, \ldots, y_m$. Moreover the clauses of $\mathcal{C}$ are obtained from each internal node. For example, let $n_i$ be an internal node, and suppose it's two children have associated literals $l_1$ and $l_2$. If $n_i$ is a $\wedge$-operation, then the goal is to replace the formula

$$y_i \leftrightarrow (l_1 \wedge l_2)$$

with a logically equivalent conjunction of disjunctive clauses. The same is true in the case that $n_i$ is a $\vee$-operation: we must replace

$$y_i \leftrightarrow (l_1 \vee l_2)$$

with a logically equivalent conjunction of disjunctive clauses.

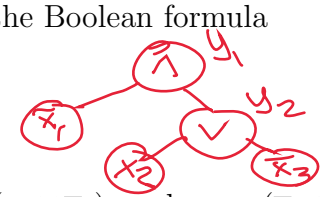
Finally, we add the clause $y_1$ to assert that formula $F$ evaluates to 1.

To see that $f(F) = \mathcal{C}$ is a polynomial-time reduction, we first note that $\mathcal{C}$ has a number of clauses and variables that is linear in $|F|$. Is is because each formula of the form $y_i \leftrightarrow (l_1 \wedge l_2)$ or $y_i \leftrightarrow (l_1 \vee l_2)$ yields up to six disjunctive formulas. Thus $f(F)$ can be constructed in a number of steps that is linear with respect to $|F|$.

Secondly, if $F$ is satisfiable, then there is an assignment $\alpha$ over $x_1, \ldots, x_n$ for which $F(\alpha) = 1$. Moreover, based on the recursive tree evaluation of $F(\alpha)$, this computation of $F(\alpha)$ also yields a correpsonding assignment $\beta$ over the internal $y$ variables in which $y_1$ is assigned 1. Hence, $\alpha \cup \beta$ is a satisfying assignment for $\mathcal{C}$. Conversely, if $\alpha \cup \beta$ is a satisfying assignment for $\mathcal{C}$, then, since the formulas of $\mathcal{C}$ represent a non-recursive representation for evaluating $F(x_1, \ldots, x_n)$, it follows that $\alpha$ must satisfy $F$, since it induces a computation of each node of $F$ in which the root node is evaluated to 1, since $\beta$ must assign $y_1 = 1$ in order to satisfy $\mathcal{C}$. $\square$

**Example 7.6.** Apply the reduction described in Theorem 7.5 to the Boolean formula

$$F(x_1, x_2, x_3) = \overline{x}_1 \wedge (x_2 \vee \overline{x}_3)).$$




**Solution.** We introduce Boolean variables $y_1$ and $y_2$, where $y_2 \leftrightarrow (x_2 \vee \overline{x}_3)$, and $y_1 \leftrightarrow (\overline{x}_1 \wedge y_2$. Then $F(x_1, x_2, x_3)$ is satisfiable iff

$$y_1 \wedge (y_1 \leftrightarrow (\overline{x}_1 \wedge y_2)) \wedge (y_2 \leftrightarrow (x_2 \vee \overline{x}_3))$$

is satisfiable. We now convert the latter to a logically-equivalent 3SAT-formula.

**Step 1:** replace $P \leftrightarrow Q$ with $(P \to Q) \wedge (Q \to P)$.

$$y_1 \wedge (y_1 \to (\overline{x}_1 \wedge y_2)) \wedge ((\overline{x}_1 \wedge y_2) \to y_1) \wedge (y_2 \to (x_2 \vee \overline{x}_3)) \wedge ((x_2 \vee \overline{x}_3) \to y_2).$$

**Step 2:** replace $P \to Q$ with $\overline{P} \vee Q$.

$$y_1 \wedge (\overline{y_1} \vee (\overline{x}_1 \wedge y_2)) \wedge (\overline{(\overline{x}_1 \wedge y_2)} \vee y_1) \wedge (\overline{y_2} \vee (x_2 \vee \overline{x}_3)) \wedge (\overline{(x_2 \vee \overline{x}_3)} \vee y_2).$$

**Step 3:** apply De Morgan's rule.

$$y_1 \wedge (\overline{y_1} \vee (\overline{x}_1 \wedge y_2)) \wedge (x_1 \vee \overline{y}_2 \vee y_1) \wedge (\overline{y_2} \vee (x_2 \vee \overline{x}_3)) \wedge ((\overline{x}_2 \wedge x_3) \vee y_2).$$

**Step 4:** distribute $\vee$ over $\wedge$.

$$y_1 \wedge ((\overline{y_1} \vee \overline{x}_1) \wedge (\overline{y_1} \vee y_2)) \wedge (x_1 \vee \overline{y}_2 \vee y_1) \wedge (\overline{y_2} \vee x_2 \vee \overline{x}_3) \wedge ((\overline{x}_2 \vee y_2) \wedge (x_3 \vee y_2)).$$

**Step 5:** Repeat last literal enough times to make three literals per clause and use clause notation.

$$\{(y_1, y_1, y_1), (\overline{y_1}, \overline{x}_1, \overline{x}_1), (\overline{y_1}, y_2, y_2), (x_1, \overline{y}_2, y_1), (\overline{y_2}, x_2, \overline{x}_3), (\overline{x}_2, y_2, y_2), (x_3, y_2, y_2)\}.$$

For the reduction from `SAT` to `3SAT`, it's fair to ask why it is necessary to add new $y$-variables and through so many steps to transform $F$ to a set of `3SAT` clauses. For example, why not just map $F$ to

$$\{(\overline{x}_1, \overline{x}_1, \overline{x}_1), (x_2, \overline{x}_3, \overline{x}_3)\}?$$

The problem is that not all formulas are this simple, and some relatively simple formulas may require an exponential number of steps if no new variables are introduced. As an example, consider the formula

$$F(x_1, \ldots, x_{2n}) = (x_1 \wedge x_2) \vee (x_3 \wedge x_4) \vee \cdots \vee (x_{2n-1} \wedge x_{2n}).$$

This formula is in what is called **disjunctive normal form (DNF)** since it is an OR of AND's. Moreover, to convert it to a logically equivalent formula in **conjunctive normal form (CNF)**, an AND of OR's, would require a number of steps that is exponential with respect to $n$. This is because it's logically equivalent CNF form has $2^n$ clauses! Verify this for $n = 2$ and $n = 3$ by repeatedly applying the distributive rule of $\vee$ over $\wedge$.

Although it is seems lengthy even for the simplest of formulas, the reduction method used in Example 7.6 has the advantage of requiring a maximum of $C > 0$ steps per logic operation of $F$, where $C$ is a constant. This is because all five steps of the procedure require only a constant number of operations. Therefore, the reduction can be completed in $O(|F|)$ steps which is a linear (and hence a polynomial) number of steps with respect to the size of $F$.

## 7.3 Interdomain reductions

In this section we look at **interdomain** mapping reductions that reduce a problem from one domain to a problem in a different domain. Some of the different mathematical and computer science domains include the following. include

**Mathematics** logic, graph theory, algebra and number theory, numerical optimization, geometry

**Computer Science** machine learning, network design and analysis, data storage and retrieval, cryptography/security, operating systems, automata and languages, programming languages and program optimization.

Of course, as is witnessed by interdomain reductions, different domains are often related in several ways, and thus there is some subjectivity regarding the classifications of problems. Nevertheless, the above mentioned domains are considered separate and vast areas of research. As we'll see in the following examples, interdomain reductions are often the more surprising and clever of all reductions.

The reductions we study in this section both reduce from the `3SAT` logic problem. Because of the ability to reduce `3SAT` to other problem domains, `3SAT` plays a crucial role in the study of NP-completness which we cover in the next lecture.

The following theorem uses refers to `Clique`, the decsion-problem version of `Max Clique`. In this case, an instance of the `Clique` is a simple graph $G = (V, E)$ and an integer $k \geq 0$. The problem is to decide if there is a subset $C \subseteq V$ of $k$ vertices that are pairwise adjacent.

**Theorem 7.7.** `3SAT` $\leq_m^p$ `Clique`.

**Proof.** Let $\mathcal{C}$ be a collection of $m$ clauses, where clause $c_i$, $1 \leq i \leq m$, has the form $c_i = l_{i1} \vee l_{i2} \vee l_{i3}$. We now define a mapping $f(\mathcal{C}) = (G, k = m)$ for which $G$ has an $m$-clique if and only if $\mathcal{C}$ is satisfiable. $G = (V, E)$ is defined as follows. $V$ consists of $3m$ vertices, one for each literal $l_{ij}$, $1 \leq i \leq m$ and $1 \leq j \leq 3$. Then $(l_{ij}, l_{rs}) \in E$ iff i) $i \neq r$ and ii) $l_{ij}$ is not the negation of $l_{rs}$ (i.e. the two literals are logically consistent).

First assume that $\mathcal{C}$ is satisfiable. Given a satisfying assignment $\alpha$ for $\mathcal{C}$, let $l_{ij_i}$, $1 \leq i \leq m$, denote a literal from clause $i$ that is satisfied by $\alpha$, i.e. $\alpha(l_{ij_i}) = 1$. Then, since each pair of these literals is consistent, $(l_{ij_i}, l_{rj_r}) \in E$ for all $i \neq r$. In other words,

$$C = \{l_{1j_1}, l_{2j_2}, \ldots, l_{mj_m}\}$$

is an $m$-clique for $G$.

Conversely, assume $G$ has an $m$-clique. Then by the way $G$ is defined the clique must have the form

$$C = \{l_{1j_1}, l_{2j_2}, \ldots, l_{mj_m}\}$$

where $l_{ij_i}$ is a literal in $c_i$. This is true since no two literal vertices from the same clause can be adjacent and so each literal vertex in $C$ must come from a different clause. Moreover, by the definition of $G$, $C$ is a consistent set of literals. Hence the assignment $a_C$ induced by $C$ satisifes every clause of $\mathcal{C}$, since $\alpha_C(l_{ij_i}) = 1$ satisfies $c_i$, for each $i = 1, \ldots, m$. Therefore, $\mathcal{C}$ is satisfiable.

Finally, we must show is that $f(\mathcal{C})$ may be computed via an algorithm whose running time is polynomial in $m$ and $n$. But this can be done via two nested `for`-loops that iterate through each pair of clauses $i$ and $r$, $i \neq r$, and identify all consistent pairs of literals $(l_{ij_i}, l_{rj_r})$. This require $\mathrm{O}(m^2)$ steps. □

**Example 7.8.** Show the reduction provided in the proof of Theorem 7.7 for input instance

$$\mathcal{C} = \{(x_1, x_1, x_2), (\overline{x}_1, \overline{x}_2, \overline{x}_2), (\overline{x}_1, x_2, x_2)\}.$$

$C_1$ $C_2$ $C_3$

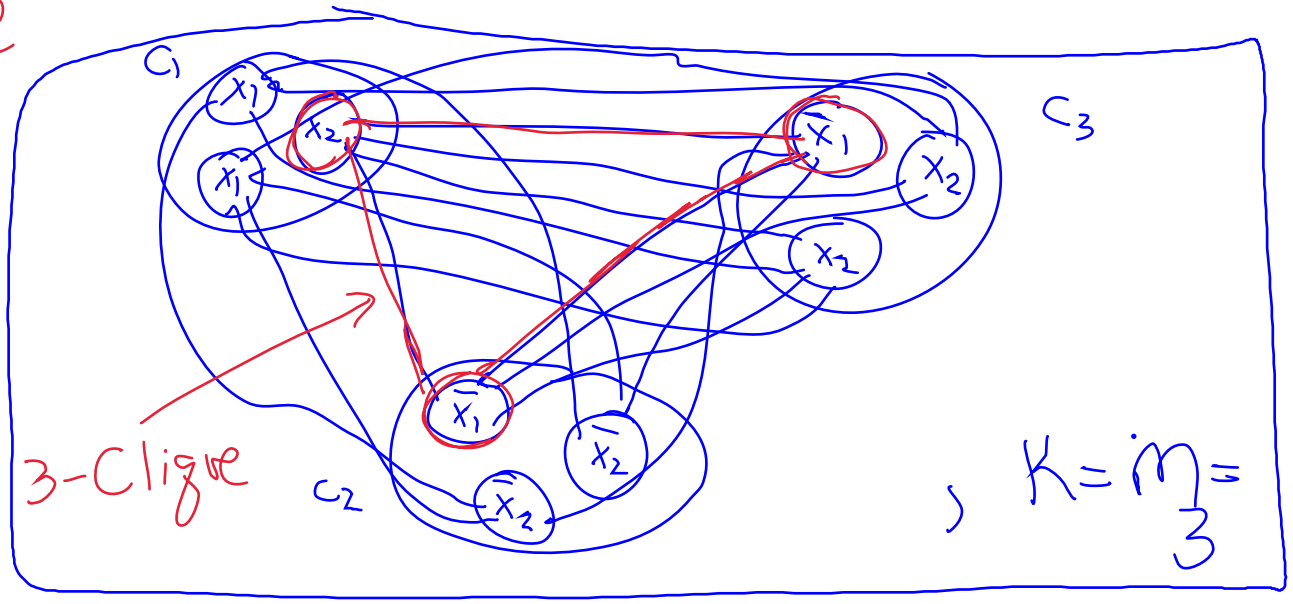
$m = 3$ clauses

$\alpha = (x_1 = 0, x_2 = 1)$ satisfies $\mathcal{C}$

$\alpha(\overline{x}_1) = 1$

$\alpha(x_2) = 1$

$f(\mathcal{C}) =$




, $k = m = 3$

Conversely, if $C = \{l_1, \ldots, l_m\}$ is an $m$-clique in $G$, then we assume $l_i \in C_i$ for each $i = 1, \ldots, m$. Also, $\{l_1, \ldots, l_m\}$ is a consistent set of literals.

So, $\alpha_C$ defined by $\alpha_C(l_i) = 1$

$\{l_1, \ldots\}$ $\{l_2, \ldots\}$ $\ldots$ $\{l_m, \ldots\}$

$C_1$ $C_2$ $C_m$

$\Rightarrow \mathcal{C}$ is satisfiable

**Theorem 7.9.** 3SAT $\leq_m^p$ Subset Sum.

Let $\mathcal{C}$ be a collection of $m$ ternary clauses over $n$ variables. The following table provides the reduction $f(\mathcal{C}) = (S, t)$. The table rows correspond to the $(n+m)$-digit integers comprising set $S$, while $t$ is the integer at the bottom whose first $n$ digits are 1's and whose final $m$ digits are 3's.

| | 1 | 2 | 3 | 4 | $\cdots$ | $n$ | $c_1$ | $c_2$ | $\cdots$ | $c_m$ |
|---|---|---|---|---|---|---|---|---|---|---|
| $y_1$ | 1 | 0 | 0 | 0 | $\cdots$ | 0 | 1 | 0 | $\cdots$ | 0 |
| $z_1$ | 1 | 0 | 0 | 0 | $\cdots$ | 0 | 0 | 0 | $\cdots$ | 0 |
| $y_2$ | | 1 | 0 | 0 | $\cdots$ | 0 | 1 | 0 | $\cdots$ | 0 |
| $z_2$ | | 1 | 0 | 0 | $\cdots$ | 0 | 0 | 0 | $\cdots$ | 0 |
| $y_3$ | | | 1 | 0 | $\cdots$ | 0 | 1 | 1 | $\cdots$ | 0 |
| $z_3$ | | | 1 | 0 | $\cdots$ | 0 | 0 | 0 | $\cdots$ | 1 |
| $\vdots$ | | | | | | $\vdots$ | $\vdots$ | | $\vdots$ | $\vdots$ |
| $y_n$ | | | | | | 1 | 0 | 0 | $\cdots$ | 0 |
| $z_n$ | | | | | | 1 | 0 | 0 | $\cdots$ | 0 |
| $g_1$ | | | | | | | 1 | 0 | $\cdots$ | 0 |
| $h_1$ | | | | | | | 1 | 0 | $\cdots$ | 0 |
| $g_2$ | | | | | | | | 1 | $\cdots$ | 0 |
| $h_2$ | | | | | | | | 1 | $\cdots$ | 0 |
| $\vdots$ | | | | | | | | | | $\vdots$ |
| $g_m$ | | | | | | | | | $\cdots$ | 1 |
| $h_m$ | | | | | | | | | $\cdots$ | 1 |
| $t$ | 1 | 1 | 1 | 1 | $\cdots$ | 1 | 3 | 3 | $\cdots$ | 3 |

Number $y_i$ correpsonds to literal $x_i$, while $z_i$ corresponds to literal $\overline{x}_i$. Thus, since the first $n$ digits of $t$ are 1's, we see that, to construct a subset $A$ whose members sum to $t$, it must either contain $y_i$ or $z_i$, but not both. Also, the final $m$ digits of $y_i$ (respectively, $z_i$) indicate which clauses $c_i$ are satisfied by $x_i$ (respectively $\overline{x}_i$).

Now suppose $\mathcal{C}$ is satisfiable via some assignment $\alpha$. Then the subset $A$ needed to sum to $t$ includes the following numbers. For $1 \leq i \leq n$, if $\alpha(x_i) = 1$, then add $y_i$ to $A$; otherwise add $z_i$ to $A$. For $1 \leq j \leq m$, to determine if $g_j$ and/or $h_j$ should be added to $A$, consider clause $c_j = \{l_{j1}, l_{j2}, l_{j3}\}$ and the sum

$$\sigma_j = \alpha(l_{j1}) + \alpha(l_{j2}) + \alpha(l_{j3}).$$

Since $\alpha$ satisfies $\mathcal{C}$, we must have $\sigma_j \geq 1$.

Case 1: $\sigma_j = 1$. In this case add both $g_j$ and $h_j$ for the $c_j$-column to sum to 3.

Case 2: $\sigma_j = 2$. In this case add only $g_j$ for the $c_j$-column to sum to 3.

Case 3: $\sigma_j = 3$. In this case neither $g_j$ nor $h_j$ need to be added since the $c_j$-column already sums to 3.

. Therefore, it is always possible to find a subset $A$ whose members sum to $t$.

Conversely, suppose there is a subset $A \subseteq S$ whose members sum to $t$. Then for each $i = 1, \ldots, n$, either $y_i \in A$ or $z_i \in A$, but not both. This is true since $t$'s first $n$ digits are 1's. Let $\alpha$ be an assignment over the variables of $\mathcal{C}$ such that, for each $i = 1, \ldots, n$ $\alpha(x_i) = 1$ iff $y_i \in A$. Now consider clause $c_j$, $j = 1, \ldots, m$. We know that the digits of the members of $A$ in the $c_j$-column sum to 3. Thus, one of the members must either be $y_k$ or $z_k$ for some $k = 1, \ldots, n$. In other words, either $y_k \in A$, $x_k \in c_j$, and $\alpha(x_k) = 1$ or $z_k \in A$, $\overline{x}_k \in c_j$ and $\alpha(\overline{x}_k) = 1$. In either case, we see that $\alpha$ satisfies $c_j$. Therefore, since $j = 1, \ldots, n$ was arbitrary, $\alpha$ satisfies $\mathcal{C}$.

Finally, notice that each of the $2m + 2n$ integers in $S$ can be constructed in $\mathrm{O}(m + n)$ steps, and so $f(\mathcal{C}) = (S, t)$ can be computed in $\mathrm{O}(m^2 + n^2)$ steps. □

**Example 7.10.** Show the reduction given in Theorem 7.9 using input instance

$$\mathcal{C} = \{c_1 = (x_1, x_2, x_3), c_2 = (\overline{x}_1, \overline{x}_2, \overline{x}_3), c_3 = (\overline{x}_1, x_2, x_3), c_4 = (x_1, \overline{x}_2, \overline{x}_3)\}.$$

**Solution.**

| | 1 | 2 | 3 | $c_1$ | $c_2$ | $c_3$ | $c_4$ |
|---|---|---|---|---|---|---|---|
| $y_1$ | 1 | 0 | 0 | 1 | 0 | 0 | 1 |
| $z_1$ | 1 | 0 | 0 | 0 | 1 | 1 | 0 |
| $y_2$ | | 1 | 0 | 1 | 0 | 1 | 0 |
| $z_2$ | | 1 | 0 | 0 | 1 | 0 | 1 |
| $y_3$ | | | 1 | 1 | 0 | 1 | 0 |
| $z_3$ | | | 1 | 0 | 1 | 0 | 1 |
| $g_1$ | | | | 1 | 0 | 0 | 0 |
| $h_1$ | | | | 1 | 0 | 0 | 0 |
| $g_2$ | | | | | 1 | 0 | 0 |
| $h_2$ | | | | | 1 | 0 | 0 |
| $g_3$ | | | | | | 1 | 0 |
| $h_3$ | | | | | | 1 | 0 |
| $g_4$ | | | | | | 0 | 1 |
| $h_4$ | | | | | | 0 | 1 |
| $t$ | 1 | 1 | 1 | 3 | 3 | 3 | 3 |

**Theorem 7.11.** The following decision problems are all NP-complete: Clique, Vertex Cover, Independent Set, Half Vertex Cover, Half Clique, Subset Sum, and Set Partition.

**Proof.** All of the following mapping reductions were proved in the Mapping Reducibility lecture.

$$\texttt{3SAT} \leq_m^p \texttt{Clique} \leq_m^p \texttt{Half Clique},$$

$$\texttt{Clique} \leq_m^p \texttt{Independent Set},$$

and

$$\texttt{3SAT} \leq_m^p \texttt{Subset Sum} \leq_m^p \texttt{Set Partition}.$$

Finally, since

$$\texttt{SAT} \leq_m^p \texttt{3SAT}$$

by Theorem 7.5, Lemma 7.1 implies that all of the above problems are NP-complete. □

**Theorem 7.12.** An instance of the `Directed Hamilton Path (DHP)` decision problem is a directed graph $G = (V, E)$ and two vertices $a, b \in V$. The problem is to decide of $G$ possesses a directed simple path from $a$ to $b$ and having length $n - 1$. Such a path is called a **(Directed) Hamilton Path (DHP)**. Then `DHP` is `NP` complete.

**Proof.** The fact that `DHP` is in `NP` is left as an exercise. We show a polynomial-time mapping reduction from `3SAT`. Let $\mathcal{C}$ be a collection of $m$ ternary clauses over $n$ variables. We proceed to define $f(\mathcal{C}) = (G = (V, E), a, b)$, a directed graph $G = (V, E)$ along with two vertices $a, b \in V$ so that $G$ has a Hamilton path from $a$ to $b$ iff $\mathcal{C}$ is satisfiable.

$G$ is defined as follows (see the graph in Example 7.13 for a specific image of the following general description). $G$ has $m$ clause vertices $c_1, \ldots, c_m$ and $n$ *diamond subgraphs*, one corresponding to each variable $x_i$, $1 \leq i \leq n$. Diamond subgraph $D_i$ consists of a top vertex $t_i$, bottom vertex $b_i$, left vertex $l_i$, and right vertex $r_i$, along with edges

$$(t_i, l_i), (t_i, r_i), (l_i, b_i), (r_i, b_i).$$

In addition, there is a row of $3m - 1$ vertices that connect $l_i$ with $r_i$:

$$lc_{i1}, rc_{i1}, s_{i1}, lc_{i2}, rc_{i2}, s_{i2} \ldots, lc_{im}, rc_{im}.$$

The $s_{ij}$ vertices, $j = 1, \ldots, m - 1$, are called *separators*, while the $lc_{ij}$ and $rc_{ij}$ pairs, $j = 1, \ldots, m$, correspond with each of the $m$ clauses and are used for making round-trip excursions to each of the clause vertices. Every vertex in the row is bidirectionally adjacent to both its left and right neighbor, i.e.,

$$(l_i, lc_{i1}), (lc_{i1}, rc_{i1}), (rc_{i1}, s_{i1}), \ldots, (s_{i(m-1)}, lc_{im}), (lc_{im}, rc_{im}), (rc_{im}, r_i) \in E,$$

as well as the reversals of each of these edges. Finally, if $x_i$ is a literal of clause $c_j$, then edges $(rc_{ij}, c_j), (c_j, lc_{ij})$ are added. On the other hand, if $\overline{x}_i$ is a literal of clause $c_j$, then edges $(lc_{ij}, c_j), (c_j, rc_{ij})$ are added.

Finally, for $1 \leq i \leq n - 1$ the edges $(b_i, t_{i+1})$ are added to connect the $n$ diamond subgraphs, and $a = t_1$, while $b = b_n$. We leave it as an exercise to show that $f(\mathcal{C}) = (G = (V, E), a, b)$ can be constructed in time that is polynomial with respect $m$ and $n$. It remains to prove that $\mathcal{C}$ is satisfiable iff $f(\mathcal{C}) = (G = (V, E), a, b)$ has a DHP from $a$ to $b$.

**Claim.** Suppose $P$ is a DHP from $a$ to $b$. Then, for all $i = 1, \ldots, n - 1$, $P$ must visit every vertex in $D_i$ before moving to a later diamond $D_j$, $j > i$.

**Proof of Claim.** Suppose by way of contradiction that $P$ is a DHP from $a$ to $b$, and let $D_i$ be the first diamond where the path moves from $D_i$ to $D_j$, for some $j > i$, without having visited every vertex in $D_i$. The only way this can happen is if $P$ moves from either vertex $lc_{ik}$ or $rc_{ik}$ in $D_i$ to clause vertex $c_k$, and then from there moves to either vertex $lc_{jk}$ or $rc_{jk}$ in $D_j$. In other words, the clause vertex $c_k$ acts as a bridge between the two diamonds. Without loss of generality, assume that $P$ is moving from left to right through $D_i$, then moves from $lc_{ik}$ to $c_k$, followed by moving to either $lc_{jk}$ or $rc_{jk}$. Now consider vertex $rc_{ik}$. The only vertex that has yet to be visited and can reach $rc_{ik}$

is separator vertex $s_{ik}$. Thus, $rc_{ik}$ must immediately follow $s_{ik}$ in $P$. But then there are no other vertices that can be visited after $rc_{ik}$ since both $lc_{ik}$ and $s_{ik}$ have been visited, which contradicts that $P$ is a DHP from $a = t_1$ to $b = b_n$. A similar argument holds if instead the path moves from $rc_{ik}$ to $c_k$.

By the above claim, we see that, when forming a DHP, there is at most one direction (left-to-right or right-to-left) that a path can move through a diamond $D_i$ and be able to visit some clause vertex $c$. Moreover, based on how $G$ was defined, the direction is left-to-right (respectively, right-to-left) iff $\overline{x}_i$ (respectively, $x_i$) is a literal of $c$. For some path $P$ that traverses through all the diamonds (and perhaps some of the clause vertices), starting at $a$ and finishing at $b$, let $\Delta(P) = (\delta_1, \ldots, \delta_n)$ denote a binary vector, where $\delta_i$ denotes the direction that $P$ takes (0 = left-to-right, 1 = right-to-left) through diamond $D_i$. We'll call $\Delta(P)$ the *signature* of $P$. As an example, consider the path $P$ shown in Example 7.13 that is highlighted in green. Then its signature is $\Delta(P) = (0, 1)$ since it moves left-to-right in the $x_1$ diamond, and right-to-left in the $x_2$ diamond.

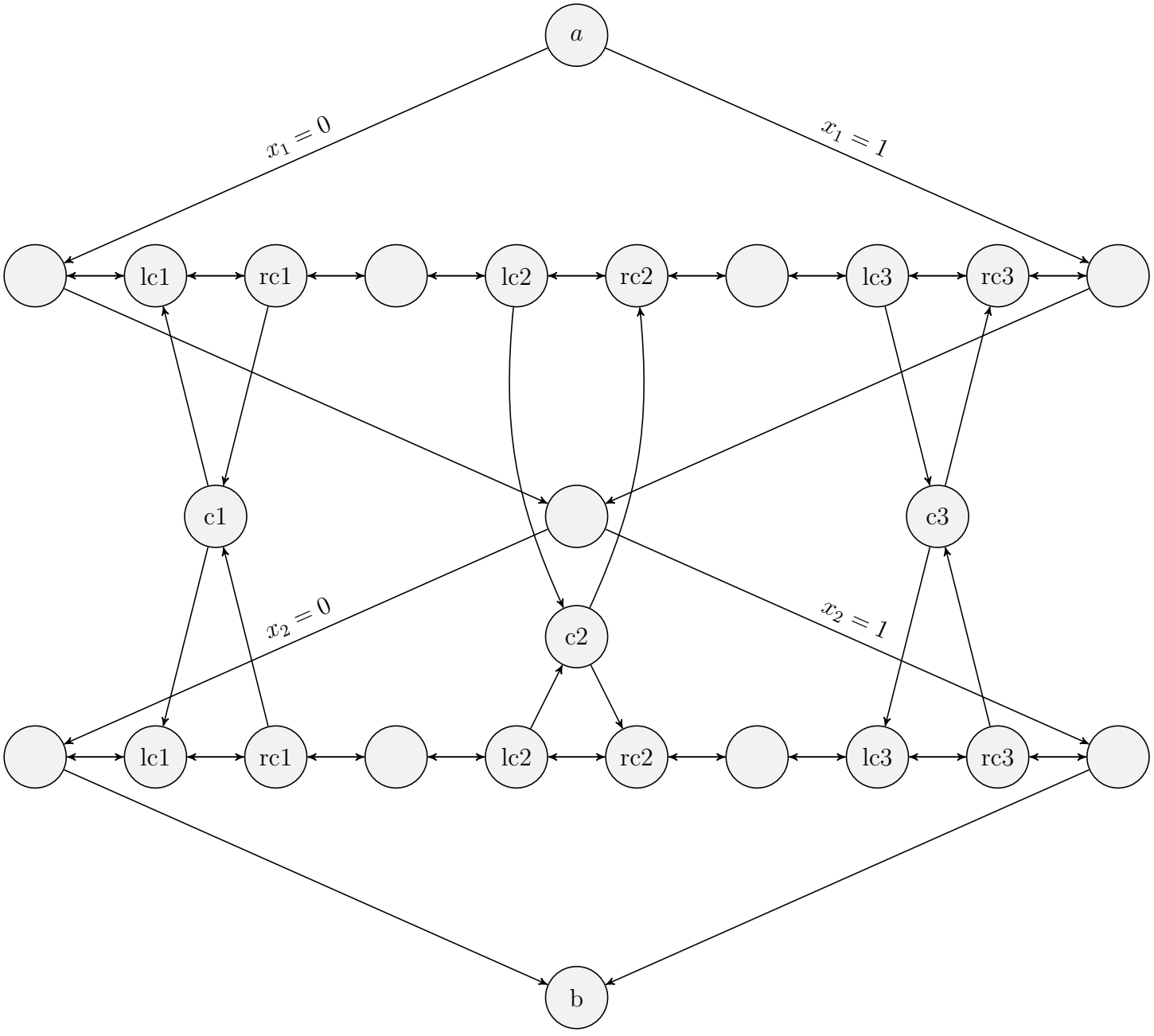
Now suppose $\mathcal{C}$ is satisfiable via satisfying assignment $\alpha$. Then there is a path $P$ for which $P$ i) has signature $\Delta(P) = (\alpha(x_1), \ldots, \alpha(x_n))$, ii) visits every clause vertex exactly once, and iii) is a DHP from $a$ to $b$. To see this, consider a clause $c_j$ and let $i$ be the least index for which $\alpha(x_i)$ satisfies $c_j$. Then if, for example, $\alpha(x_i) = 1$, then $x_i$ is a literal of $c_j$ and, by the way in which $G$ was defined, $P$ may move from right to left in $D_i$ and visit $c_j$ via the sequence $rc_{ij}, c_j, lc_{ij}$. Therefore, every clause clause vertex gets visited exactly once and $P$ is a DHP from $a$ to $b$.

Conversely, suppose $P$ is a DHP in $G$ and let $\Delta(P) = (\delta_1, \ldots, \delta_n)$ be its signature. Then the variable assignment $\alpha$ defined by $\alpha(x_i) = \delta_i$, $i = 1, \ldots, n$, satisfies $\mathcal{C}$. To see this, consider a clause $c_j$ and let $D_i$ be the diamond from where $P$ visits $c_j$. Then by the way $G$ was defined, $P$ can either visit $c_j$ by moving left-to-right, in which case $\overline{x}_i$ is a literal of $c_j$ and $\alpha(x_i) = \delta_i = 0$ satisfies $c_j$, or by moving right to left, in which case $x_i$ is a literal of $c_j$ and $\alpha(x_i) = \delta_i = 1$ satisfies $c_j$. In either case $\alpha$ satisfies $c_j$ and, since $j$ was arbitrary, we see that $\alpha$ satisfies $\mathcal{C}$. □

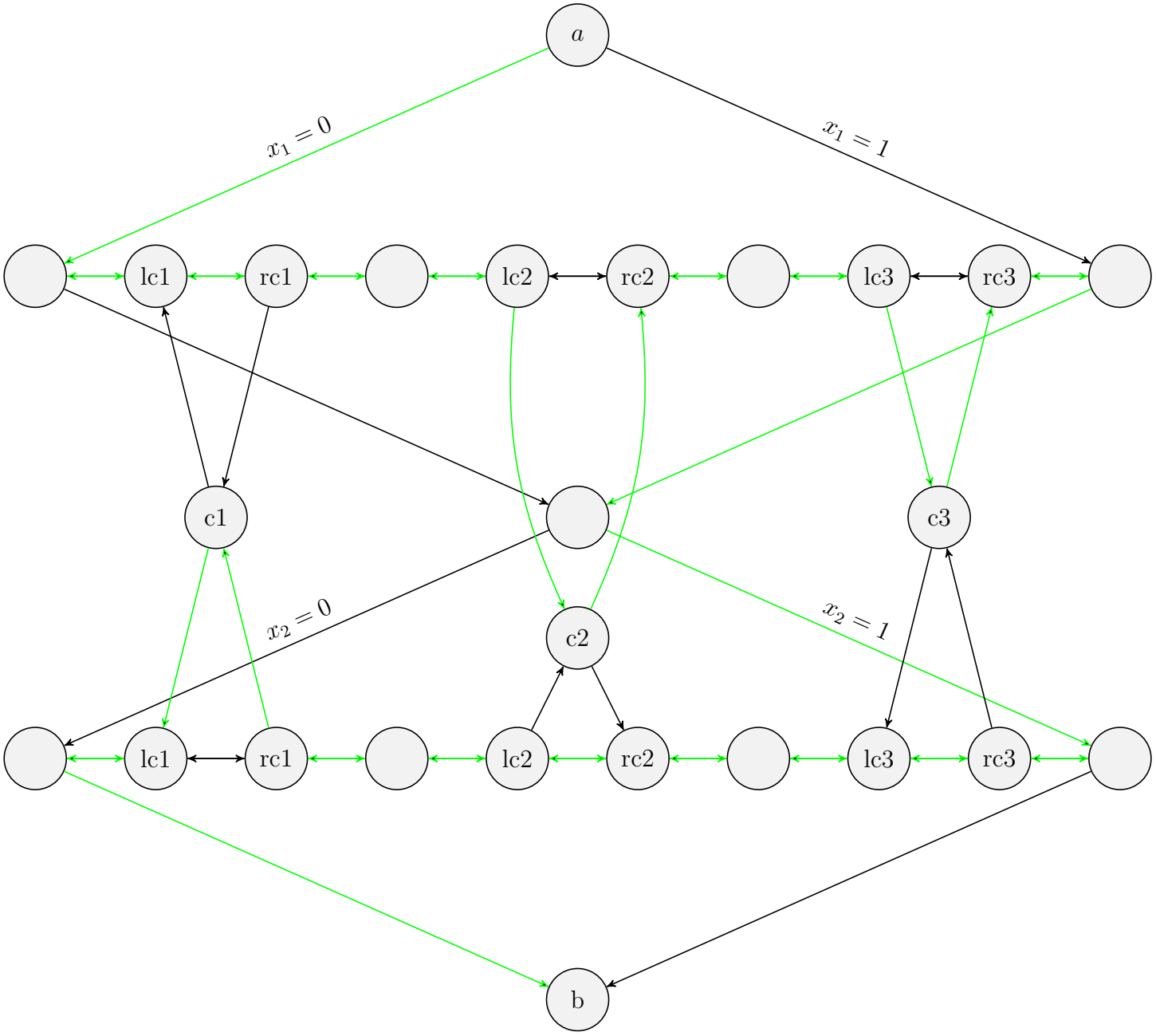
**Example 7.13.** The following graph shows $f(\mathcal{C})$, where

$$\mathcal{C} = \{c_1 = (x_1, x_2, x_2), c_2 = (\overline{x}_1, \overline{x}_2, \overline{x}_2), c_3 = (\overline{x}_1, x_2, x_2)\}$$

is an instance of `3SAT` and $f$ is the reduction reduction given in Theorem 7.12.



Notice that $\mathcal{C}$ is satisfiable via assignment $\alpha = (x_1 = 0, x_2 = 1)$. Therefore, $f(\mathcal{C})$ must have a DHP from $a$ to $b$. In fact, $\alpha$ gives directions for the path: go left in the $x_1$-diamond, right in the $x_2$-diamond, and visit a clause vertex if i) it has yet to be visited and ii) the clause is satisified by the direction of movement through the diamond. The figure below shows such a DHP in green.



# 8 The Complexity Class co-NP

Given a decision problem $L$, the **complement** of $L$, denoted $\overline{L}$, is that decision problem for which a positive (respectively, negative) instance $x$ of $\overline{L}$ is a negative (respectively, positive) instance of $L$.

**Example 8.1.** If $L$ is the problem of deciding if a positive integer is prime, then define its complement $\overline{L}$.

**Solution.**

**Example 8.2.** Define the complement of the SAT decision problem.

**Solution.**

## 8.1 A Logical definition of Co-NP

It is left as an exercise to show that if $L \in \texttt{P}$ then $\overline{L} \in \texttt{P}$. On the other hand, it is believed that NP and co-NP are different complexity classes because each has its own distinct predicate-logic definition.

For example, consider a problem $L \in \texttt{NP}$ which has certificate set $C$ and verifier function $v(x, c)$. Then we may logically write that $x$ is a positive instance of $L$ iff

$$\underset{c \in C}{\exists} v(x, c)$$

evaluates to 1.

Now consider its complement $\overline{L} \in \texttt{co-NP}$. Then we may logically write that $x$ is a positive instance of $\overline{L}$ iff it is a negative instance of $L$ iff

$$\neg \underset{c \in C}{\exists} v(x, c) \Leftrightarrow$$

$$\underset{c \in C}{\forall} (\neg v(x, c)) \Leftrightarrow$$

$$\underset{c \in C}{\forall} v'(x, c).$$

evaluates to 1, where $v'(x, c) = \neg v(x, c)$. In other words, co-NP problems are logically defined with a universal predicate-logic statement, while an NP problem is logically defined with an existential predicate-logic statement. Thus, logically speaking, these classes seem different in that their problems are complementary to one another.

**Example 8.3.** For each of the following problem definitions, provide the complexity class (`P`, `NP`, or `co-NP`) that best fits the problem.

`Tautology` Given a Boolean formula $F(x_1, \ldots, x_n)$ does $F$ evaluate to 1 on all possible $2^n$ binary input vectors?

`Reachability` Given a simple graph $G = (V, E)$ and two vertices $a, b \in V$, does there exist a path in $G$ starting at $a$ and ending at $b$?

`Dominating Set` Given a simple graph $G = (V, E)$ and an integer $k \geq 0$, does there exist a set $D$ of $k$ vertices for which every vertex in $V - D$ is adjacent to some vertex in $D$?

`Bounded Cliques` Given a simple graph $G = (V, E)$ and an integer $k \geq 0$, it true that $G$ has no $k$-cliques?