# Lecture Notes on Complexity and NP-completeness

## 1. Reductions

Let A and B be two problems whose instances require as an answer either a "yes" or a "no" (3SAT and Hamilton cycle are two good examples). A *reduction* from A to B is a polynomial-time algorithm $R$ which transforms inputs of A to equivalent inputs of B. That is, given an input $x$ to problem A, $R$ will produce an input $R(x)$ to problem B, such that $x$ is a "yes" input of A if and only if $R(x)$ is a "yes" input of B.

A reduction from A to B, together with a polynomial time algorithm for B, constitute a polynomial algorithm for A (see Figure). For any input $x$ of A of size $n$, the reduction $R$ takes time $p(n)$ —a polynomial— to produce an equivalent input $R(x)$ of B. Now, this input $R(x)$ can have size a most $p(n)$ —since this is the largest input $R$ can conceivably construct in $p(n)$ time. If we now submit this input to the assumed algorithm for B, running in time $q(m)$ on inputs of size $m$, where $q$ is another polynomial, then we get the right answer of $x$, within a total number of steps at most $p(n) + q(p(n))$ —also a polynomial!

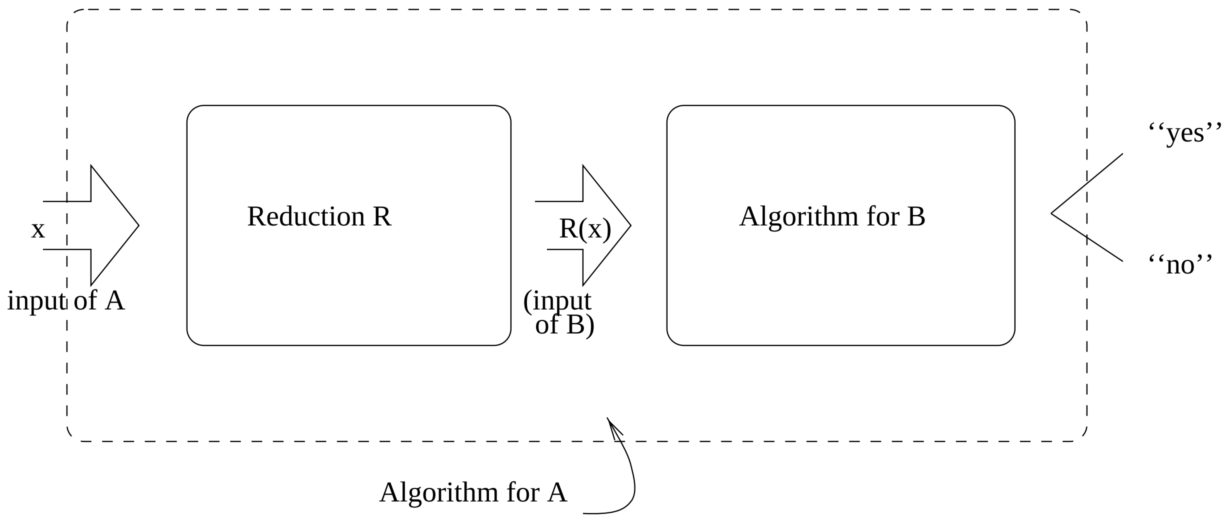


Figure 1: Reduction

We have seen many reductions so far, establishing that problems are easy (e.g., from matching to max-flow). In this part of the class we shall use reductions in a more sophisticated and counterintuitive context, in order to prove that certain problems are hard. If we reduce A to B, we are essentiually establishing that, *give or take a polynomial, A is no harder than B.* We could write this as

$$\text{A} \leq \text{B},$$

an inequality between the complexities of the two problems. If we know B is easy, this establishes that A is easy. If we know A is hard, this establishes B is hard. It is this latter implication that we shall be using soon.

## 2. Problems, problems...

We have seen many problems that we can solve in polynomial time —and we know this is good. The class of all problems that are so solvable is denoted **P**. We review several problems from **P**below; in each case we also list another problem, bearing a superficial similarity to the one in **P**. The similarity is, indeed, superficial: the second problem in each pair is not known (or believed, or expected) to be solved in polynomial time; only algorithms that are exponential in the worst case are known for these problems.

- `minimum spanning tree`: Given a weighted graph and an integer $K$, is there a tree that connects all nodes of the graph whose total weight is $K$ or less?
- `traveling salesman problem`: Given a weighted graph and an integer $K$, is there a cycle that visits all nodes of the graph whose total weight is $K$ or less?

Notice that we have converted each one of these familiar problems into a decision problem, a "yes-no" question, by supplying a goal $K$ and asking if the goal can be met. Any optimization problem can be so converted (we shall soon see more examples). If we can solve the optimization problem, we can certainly solve the decision version (actually, the converse is in general also true). Therefore, proving a negative complexity result about the decision problem (for example, proving that it cannot be solved in polynomial time) immediately implies the same negative result for the optimization problem. By considering the decision versions, we can study optimization problems side-by-side with decision problems (see the next examples), and consider reductions between them. This is a great convenience in the theory of complexity which we are about to develop.

- `Eulerian graph`: Given a directed graph, is there a closed path that visits each edge of the graph exactly once?
- `Hamilitonian graph`: Given a directed graph, is there a closed path that visits each *node* of the graph exactly once?

A graph is Eulerian if and only if it is strongly connected and each node has equal in-degree and out-degree; so the problem is squarely in **P**. There is no known such characterization —or algorithm— for the Hamilton problem (and notice its eerie similarity with the TSP).

- `circuit value`: Given a Boolean circuit, and its inputs, is the output T?
- `circuit SAT`: Given a Boolean circuit, and *some* of its inputs, is there a way to set the remaining inputs so that the output is T?

We know that `circuit value` is in **P**(we showed that it can be reduced to linear programming; also, the naïve algorithm for that evaluates all gates bottom-up is polynomial). How about `circuit SAT`? There is no obvious way to solve this problem, sort of trying all input combinations for the unset inputs —and this is an exponential algorithm.

- `2SAT`: Given a Boolean formula in conjunctive normal form and with at most two literals per clause, is there a satisfying truth assignment?
- `3SAT`: Given a Boolean formula in conjunctive normal form and with at most *three* literals per clause, is there a satisfying truth assignment?

We know from the homework that 2SAT can be solved by graph-theoretic techniques. For 3SAT, no such techniques are available, and the best algorithms known for this problems are exponential in the worst case.

- `matching`: Given a boys-girls compatibility graph, is there a complete matching??

- `3D matching`: Given a boys-girls-homes compatibility relation (that is, a set of triangles), is there a complete matching (a set of disjoint triangles that covers all boys, all girls, and all homes)?

We know that matching can be solved by a reduction to max-flow, and then to linear programming. For `3D matching` we shall see a reduction too. Unfortunately, the reduction is from 3SAT to `3D matching` —and this is bad news for `3D matching`...

- `unary knapsack`: Given integers $a_1, \ldots, a_n$, and another integer $K$ *in unary*, is there a subset of these integers that sum exactly to $K$?
- `knapsack`: Given integers $a_1, \ldots, a_n$, and another integer $K$ *in binary*, is there a subset of these integers that sum exactly to $K$?

`unary knapsack` is in **P**—simply because the input is represented so wastefully, with about $n+K$ bits, so that the $O(n^2K)$ dynamic programming algorithm, which would be exponential if $K$ were represented in binary, is bounded by a polynomial in the length of the input. There is no polynomial algorithm known for `knapsack`.

- `linear programming`: Given an $m \times n$ matrix $A$ and an $m$ vector $b$, are there *real numbers* $x_1, \ldots, x_n \geq 0$ satisfying $Ax \leq b$?
- `integer linear programming`: Given an $m \times n$ matrix $A$ and an $m$ vector $b$, are there *integers* $x_1, \ldots, x_n \geq 0$ satisfying $Ax \leq b$?

Although there are algorithms that solve linear programming in polynomial time (simplex is exponential in the worst case, but certain more recently proposed alternative approaches work in polynomial time), the additional requirement that the solution consist of integers seems to make the problem impossible. Rounding the solutions of the linear program up or down is no help —even finding whether a rounding that remains feasible is possible is a hard problem (see the reduction from `3SAT` to `integer linear programming` later in these notes).

**3. Certificates and the Class NP**Although some of the problems we saw in the pre-

vious section (`TSP, 3SAT, circuit SAT, Hamilton cycle, 3D matching, knapsack`) are not known of believed to be solvable by polynomial algorithms, they all have a positive common property: *the certificate property.* In each case, if a given input of the problem is a "yes" input (a satisfiable Boolean formula, a graph with a Hamilton cycle), *then there is a short argument, a succinct certificate* that may convince one about the fact that the input is indeed a "yes" input. In the case of `3SAT`, the certificate would be a satisfying truth assignment. In the case of *Hamilton cycle*, it would be a closed path that visits each node once. In the case of the `TSP`, it would be a tour whose total cost is less than or equal to the given goal. And so on. These certificates have the following properties:

- They are *small.* In each case the certificate would never have to be longer than a polynomial in the length of the input.
- They are *easily checkable.* In each case there is a polynomial algorithm which takes as inputs the input of the problem and the alleged certificate, and checks whether the certificate is a valid one for this input. In the case of `3SAT`, the algorithm would just check that the truth assignment indeed satisfies all clauses. In the case of *Hamilton cycle* whether the given closed path indeed visits every node once. And so on.

- Every "yes" input to the problem has at least one certificate (possibly many), and each "no" input has none.

Not all problems have such certificates. Consider, for example, the problem `non-Hamiltonian graph`: Given a graph $G$, is it true that there is no Hamilton cycle in $G$? How would you prove to a suspicious person that a given large, dense, complex graph has *no* Hamilton cycle? Short of listing all cycles and pointing out that none visits all nodes once (a certificate that is certainly not succinct)?

The problems that have this positive property comprise a class known as **NP**[1]. A "yes-no" problem is in **NP**if and only if it has the certificate property. That is, every "yes" instance has at least one concise certificate of its "yes-ness", and all "no" instances have no such certificate; and furthermore a certificate can be tested efficiently for validity.

Notice also that **P**is a subset of **NP**. To see why, suppose that a problem is in **P**, that is, it has a polynomial-time algorithm. But then a *trace* of this algorithm run on a given input, and returning "yes", is a good certificate for this input: It is concise, can be tested fast, "yes" inputs have one, "no" inputs don't.

Let us next consider the problem `circuit SAT`, defined above. It is of course in **NP**: A setting of the unknown input gates s that makes the whole circuit T serves well as a certificate of any "yes" input. It turns out that `circuit SAT` plays a very special and important role within **NP**:

*A problem is in* **NP***if and only if it can be reduced to* `circuit SAT`

Let us argue why this statement (known as *Cook's theorem*, and considered as one of the most important results in Computer Science) is true. One direction is easy: If a problem A can be reduced to `circuit SAT` then of course it is in **NP**: A certificate of any "yes" input would be a running of the reduction on this input, together with a certificate for the resulting input of `circuit SAT` (a satisfying setting of the unknown input gates).

The other direction is much more complicated, but here is a plausible explanation: Suppose that we have a problem A in **NP**; we want to show that it has a reduction to `circuit SAT`. The fact that A is in **NP**means that there is a polynomial algorithm that checks inputs of A and certificates for validity. But an algorithm runs on a computer, and, after all, a computer is nothing more than a huge Boolean circuit supplying the rules whereby the next state is computed from the current state and the input. If we superpose enough (polynomially many) such circuits, we get a circuit that describes the full run of the validity algorithm on a certificate and an input, where the bits in the input gates stand for the input and the certificate. Suppose now that we are given an input $x$ of A. If we plug in the correct T/F values for $x$ in the apprpriate input gates of the circuit, and keep the input gates that correspond to the certificate unknown, we get an instance of `circuit SAT` that precisely captures the question whether a valid certificate for $x$ exists —that is to say, whether $x$ is a "yes" instance of A. Hence, the construction of the circuit we described is the sought reduction from A to `circuit SAT`!

If a "yes-no" problem has these two properties:

- it is in **NP**;
- all other problems in **NP**reduce to it;

[1]**NP**stands for *nondeterministic polynomial*, meaning that all problems in it can be "solved" in polynomial time by a nondeterministic "computer" that starts by *guessing* the right certificate, and then checking it.

then it is called **NP**-complete. The existence of such problems may seem *a priori* unlikely, but we already know that there is at least one: `circuit SAT`. In the next section we shall see many more examples of **NP**-complete problems..

## 4. NP-complete problems

To prove that a problem is **NP**-complete, we typically reduce a problem that is known to be **NP**-complete to it. Now that `circuit SAT` has provided a place to start, we shall prove many problems **NP**-complete by the reductions pictured in the figure.

*From* `circuit SAT` *to* `3SAT`. Suppose that we are given a circuit $C$ with some input gates

unknown, such as the one pictured in Figure. We must construct from it an equivalent input to `3SAT`, that is, a formula $R(C)$ that is satisfiable if and only if there is a satisgying setting of the unknown input gates of $C$. The construction is shown in the figure. $R(C)$ has a variable for each gate of $C$, and also for each gate of $C$ it has certain clauses. The precise set of the clauses depends on the nature of the gate considered.

- If $x$ is a T input gate, then we simply have the clause $(x)$.
- If $x$ is a F input gate, then we have the clause $(\overline{x})$.
- If $x$ is a F input gate, then we have the clause $(\overline{x})$.
- If $x$ is an unknown input gate, then no clauses are added for it (intuitively, they are free to be whatever they want, as long as they succeed in making the output gate T).

- If gate $x$ is the OR of the gates $y$ and $z$, then we add the clauses $(\overline{y} \vee x) \cdot (\overline{z} \vee x) \cdot (\overline{x} \vee y \vee z)$. It is easy to see that the conjunction of these clauses is equivalent to the statement $[x = (y \vee z)]$.
- Similarly, if gate $x$ is the AND of the gates $y$ and $z$, then we add the clauses $(\overline{x} \vee y) \cdot (\overline{x} \vee z) \cdot (\overline{y} \vee \overline{z} \vee x)$, which is $[x = (y \wedge z)]$
- Next, if gate $x$ is the NOT of gate $y$, then we add the clauses $(\overline{x} \vee \overline{y}) \cdot (x \vee y)$, which is $[x = (\overline{y})]$
- Finally, if gate $x$ also happens to be the output gate, then we add the clause $(x)$, expressing the condition that the output gate be T.

The conjunction of all these clauses is the sought formula $R(C)$. To show that the reduction $R$ works, we must establish the following statement:

*$C$ has a setting of the unknown input gates that makes the output variable T if and only if $R(C)$ is satisfiable.*

Suppose that $C$ does have such a setting of the unknown input gates. Then $R(C)$ can be satisfied by the following truth assignment: Set all variables correpsonding to unknown input gates to T or F, depending on their value in the given setting of the input gates. Compute now all values of all gates in the circuit; this is claimed the truth assignment that satisfies $R(C)$. This truth assignment must satisfy all clauses of $R(C)$, because it corresponds to a legal assignment of values to the gates of $C$, and all these clauses require is that the values be legal. Finally, since the given setting makes the output gate T, the last clause is also satisfied.

Conversely (this is the subtler direction in proofs of reductions), suppose that $R(C)$ has a satisfying truth assignment. Then consider the setting of the unknown input gates of $C$ suggested by this truth assignment. Since the truth assignment satisfies $R(C)$, it assigns to all other variables of $R(C)$ the value of the corresponding gate (since $R(C)$ requires that all gates be properly computed). But, since $R(C)$ also insists that the output be T, this means that the setting of the unknown inputs succeeds in satisfying $C$.

From `3SAT` to `integer linear programming`. This reduction is easy, since any clause such as $(x \vee \overline{y} \vee z)$ can be rewritten as the integer linear program $x + 1 - y + z \geq 1, 0 \leq x, y, z \leq 1$. Repeating for all clauses, we have an input of `integer linear programming` that is equivalent to the given input to `3SAT`.

From `3SAT` to `independent set`. In a typical input to `independent set` we are given a graph $G = (V, E)$ and an integer $K$. We are asked whether there is a set $I \subseteq V$ with $|I| \geq K$ such that if $u, v \in I$ then $[u, v] \notin E$.

We must reduce `3SAT` to `independent set`. That is, given any Boolean formula $\phi$ with at most 3 literals in each clause, we must produce a graph $G = (V, E)$ and an integer $K$ such that $G$ has an independent set of size $K$ or more if and only if $\phi$ is satisfiable.

The reduction is illustrated in the figure. $K$ is the number of clauses. For each clause we have a group of nodes, one for each literal in the clause, connected by edges in all possible ways. Also, any two nodes from different groups, correponding to contradictory literals (like $x$ and $\overline{x}$) are connected by an edge. This concludes the description of $G$, and of the reduction.

Suppose that $G$ has an independent set $I$ of size $K$ or more. Since there are $K$ groups, and all nodes in them are connected in all ways, $I$ cannot contain two nodes from any group. So, it must have one node from each group. Think of the node from a group as the literal that satisfies this clause. Since contradictory literals are connected with an edge, no nodes in $I$ are contradictory, and hence these literals together comprise a satisfying truth assignment for $\phi$. (If a particular variable was not used, this means that we can take it to be either T or F; both alternatives would satisfy $\phi$.)

Conversely, suppose that we have a satisfying truth assignment of $\phi$. Each clause must have at least one T literal; fix one of for each clause. Consider the corresponding set $I$ of $K$ nodes. Since these literals come from a single truth assignment, they are not contradictory, and so $I$ is an independent set, completing the proof.

*From* `independent set` *to* `vertex cover` *and* `clique`. Let $G = (V, E)$ be a graph. A *vertex cover* of $G$ is a set $C \subseteq V$ such that all edges in $E$ have at least one endpoint in $C$. The `vertex cover` problem is this: Given a graph $G$ and a number $K$, does $G$ have a vertex cover of size at most $K$?

The reduction from `independent set` to `vertex cover` is very easy, and based on this observation: $C$ is a vertex cover of $G = (V, E)$ if and only if $V - C$ is an independent set! This is because any two nodes not in a vertex cover cannot have an edge between them, because this edge would not have an endpoint in the vertex cover. So, here is the reduction: Given an instance $(G = (V, E), K)$ of `independent set`, we produce the instance $(G = (V, E), |V| - K)$ of `vertex cover`. There is an independent set with $K$ nodes or more if and only if there is a vertex cover of size $|V| - K$ or less.

The *clique* in a graph is a fully connected set of nodes. The `clique` problem asks whether there is a clique of size $K$ or larger in the graph. The reduction from `independent set` to `clique` is very simple: We go from the instance $(G, K)$ of `independent set` to the equivalent instance $(\overline{G}, K)$, where $\overline{G}$ is the *complement* of $G$, the graph with the same nodes as $G$, and with precisely all edges that are missing from $G$.

*From* `vertex cover` *to* `dominating set`. A set of nodes $D$ id a *dominating set* if each node either is in $D$, or is adjacent to a node in $D$. This twist on `vertex cover` is also **NP**-complete. To reduce an input $(G, K)$ of `vertex cover` to it, we simply add to $G$, for each edge $[a, b] \in E$, anew node $ab$, and two new edges $[a, ab], [b, ab]$. It is clear that any vertex cover of $G$ is a dominating set of the new graph. And any dominating set of the new graph can be made into a vertex cover of $G$ by replacing any new vertex by one of its adjacent vertices.

## 5. An Epilogue: Undecidability

Are there problems that are not even in **NP**? The answer is, "yes, but they rarely appear to come up in practice." In fact, there are problems for which there are *no algorithms at all!*

Consider the following situation. You want to write a Boolean function `term(P,X)` which takes two inputs, P and X. P is a program in the same language, and X is a data file. `term(P,X)` returns true if program P with input file X eventually terminates. If program P on file X loops forever, then `term(P,X)` returns false.

It can be proved that *such program is impossible to write!* That is to say, there is no algorithm, however ineficient, that solves the problem "given a program P and its input X, will P terminate on X?" Here is the proof:

Suppose, for the sake of contradiction, that we have written such a Boolean function `term(P,X)`. Using it, we can write the following simple program:

```
Boolean function diag(P): if term(P,P) then loop forever
```

And now the contradiction: Does `diag(diag)` terminate? It is easy to see that it does if and only if it does not! This is a contradiction, to whic we were led by assuming that `term(P,X)` can be written. We must conclude that there is no program that can be written to solve the termination (or *halting* problem above.

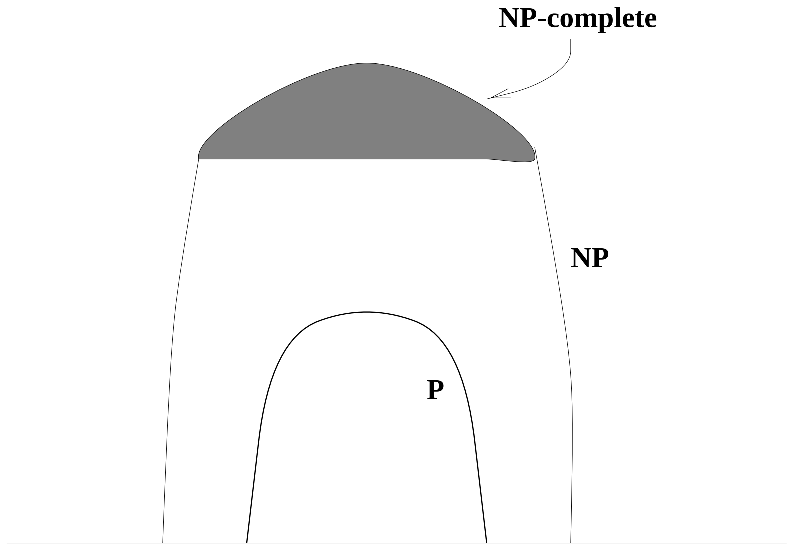


Figure 2: **P**and **NP**

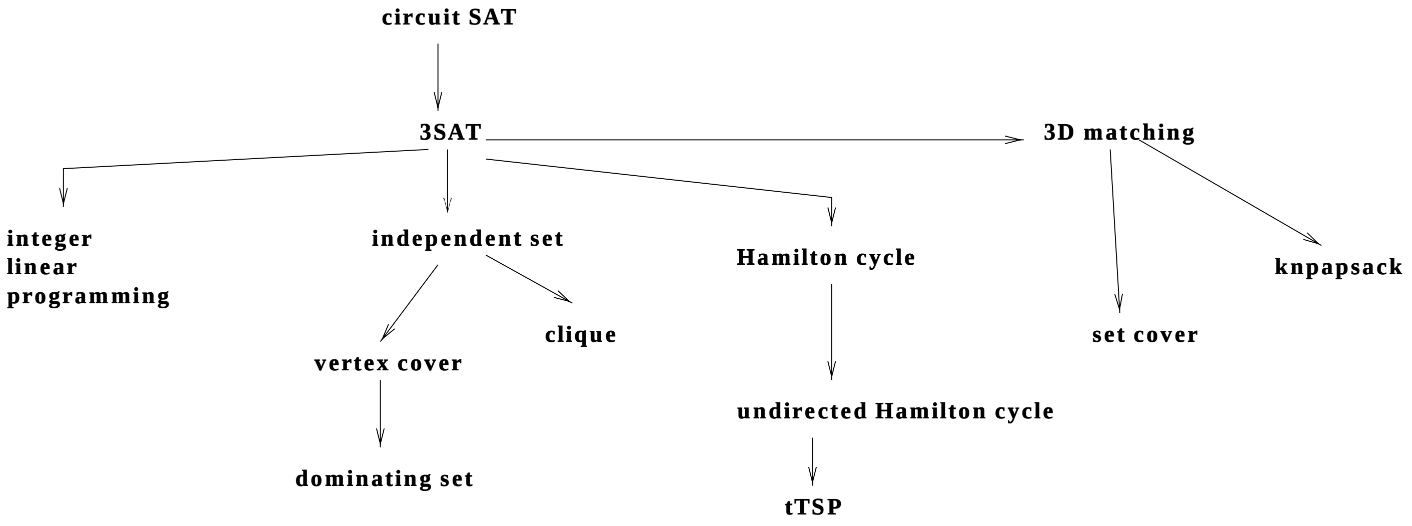
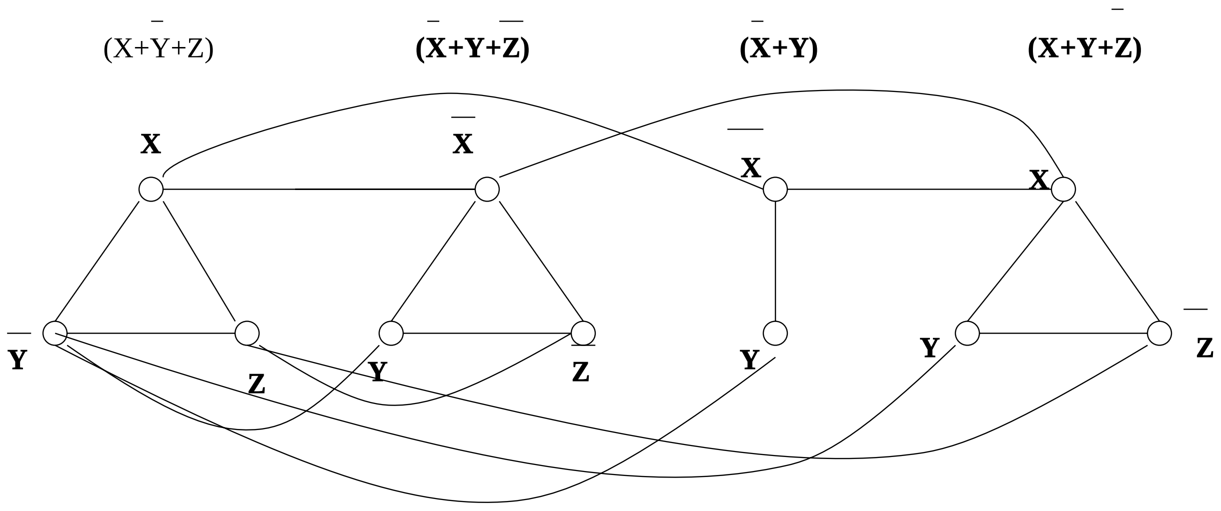


Figure 3:



$(X+\bar{Y}+Z)$
$(\bar{X}+Y+\bar{Z})$
$(\bar{X}+Y)$
$(X+Y+\bar{Z})$
X
$\bar{X}$
$\bar{X}$
X
$\bar{Y}$
Z
Y
$\bar{Z}$
Y
Y
$\bar{Z}$

Figure 4: