

Lecture #14:

0.0.1 NP-Completeness (Chapter 34 – Old Edition – Chapter 36)

Discussion here is from the old edition.

0.0.2 Preliminaries:

Definition 1 An abstract problem Q is a binary relations on a set I of problem instances and a set S of problem solutions.

Example 2 *SHORTEST-PATH* : An instance is a graph $G = [V, E]$ and two vertices. A solution is an ordered sequence of vertices (possibly the empty sequence if no path exists). A given instance may have more than one solution.

Definition 3 A decision problem is a problem having a yes/no (or 0/1} solution. So here the solution set $S = \{0, 1\}$.

Many problems are of the optimization variety but have a corresponding decision problem associated with them that come about very naturally. So we will only consider decision problems from now on.

Encodings: An encoding considered here is binary. A set S of abstract objects is mapped by a mapping $e : S \mapsto \{0, 1\}^*$. A problem whose instance set is the set of binary strings is called a **concrete problem**. We say that an algorithm solves a concrete problem in time $O(T(n))$ if, when it is provided a instance i of length $n = |i|$, it takes time at most $O(T(n))$. If $T(n) = O(n^k)$ then we say that the problem is **polynomial-time solvable**. This notion can be extended to abstract decision problem Q via a mapping e that maps an instance $i \in I$ to a string $Q(i)$ in $\{0, 1\}^*$. Meaningless strings are mapped to 0.

Complexity Classes: P is the class of concrete decision problems that are polynomial-time solvable.

0.0.3 Formal-Language Framework:

Σ is a finite set of symbols called an **alphabet**. A language L over Σ is any set of strings made up of symbols in Σ . ϵ represents the **empty string** and ϕ represents the **empty language**. So $L \subseteq \Sigma^*$ where Σ^* is the language of all strings on Σ .

Operations: **Union:** $L_1 \cup L_2$; **intersection:** $L_1 \cap L_2$; **complementation:** $\Sigma^* - L = \bar{L}$; **concatenation:** $L = \{x_1x_2 : x_1 \in L_1, x_2 \in L_2\}$; **Closure or Kleene Star:** $L^* = \{\epsilon\} \cup L \cup L^2 \cup L^3 \cup \dots$, where $L^k = L$ concatenated with itself k times.

Algorithm A **accepts (rejects)** a string $x \in \{0, 1\}^*$ if given input x , the algorithm outputs $A(x) = 1(0)$. The **language accepted** by an algorithm A is the set $L = \{x \in \{0, 1\}^* : A(x) = 1\}$. Such an algorithm that accepts

L , may not reject $x \notin L$. A **language** L is **decided** by an algorithm A iff $[x \in L] \Rightarrow [A(x) = 1]$ and $[x \notin L] \Rightarrow [A(x) = 0]$. So every string is either accepted or rejected. A language L is **accepted in polynomial time** by an algorithm A if for any n -length string $x \in L$, the algorithm accepts x in time $O(n^k)$ for some fixed k . A language L is **decided in polynomial time** by an algorithm A if for any n -length string x , the algorithm decides x in time $O(n^k)$ for some fixed k .

$P = \{L : \text{there is an algorithm } A \text{ that decides } L \text{ in polynomial time}\} = \{L : \text{there is an algorithm } A \text{ that accepts } L \text{ in polynomial time}\}$

0.0.4 NP:

Poly-time verification:

A **verification algorithm** is a two-argument algorithm A , where one argument is an input binary string x and the other is a binary string y called a **certificate**. A two-argument algorithm A **verifies** an input string x if there is a certificate y such that $A(x, y) = 1$. The **language verified** by a verification algorithm A is $L = \{x \in \{0, 1\}^* : \exists y \in \{0, 1\}^* \ni A(x, y) = 1\}$. So A verifies L if for each $x \in L$, there is a y that A can use in proving that $x \in L$. Also, for any $x \notin L$, there should be no such y proving that $x \in L$.

Complexity Class NP: is the class of languages that can be verified by a polynomial-time algorithm. So $L \in NP$ iff there is polynomial-time algorithm A and a constant c such that

$$L = \{x \in \{0, 1\}^* : \exists y \in \{0, 1\}^*; |y| = O(|x|^c) \ni A(x, y) = 1\}$$

Algorithm A is said to **verify language** L in poly-time. If $L \subseteq P$ then $L \subseteq NP$ since there is a poly-time algorithm to decide L , the verification algorithm simply ignores the second argument. Hence $P \subseteq NP$; whether equality holds is the famous problem.

Open Question: Is the following true: $[L \in NP] \Rightarrow [\bar{L} \in NP]$?

If $\bar{L} \in NP$ then we say L is in the **complexity class co-NP**.

Note: $P \subseteq NP \cap co - NP$. Again we do not know if equality holds.

0.0.5 NP-completeness and Reducibility:

NPC is a subset of NP with the property that if any problem in NPC has (or does not have) a poly-time algorithm then every problem in NP does (does not). So these are the hardest problems in NP. Language L_1 is poly-time reducible to a language L_2 (written as $L_1 \leq_p L_2$) if there exists a poly-time computable function $f : \{0, 1\}^* \mapsto \{0, 1\}^*$ such that for all $x \in \{0, 1\}^*$

$$[x \in L_1] \Leftrightarrow [f(x) \in L_2]$$

f is called the reduction function and the poly-time algorithm that computes it is called a reduction algorithm.

Lemma 4 $[L_1 \leq_p L_2; L_2 \in P] \Rightarrow [L_1 \in P]$

0.0.6 NP-completeness:

A language L is **NP-complete** if

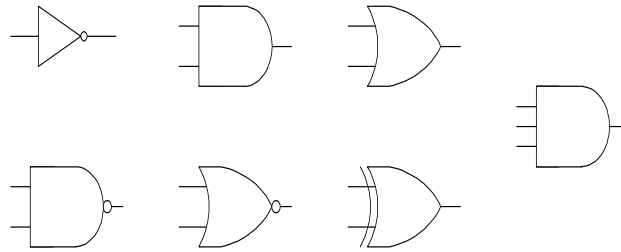
1. $L \in NP$, and
2. $L' \leq_p L$ for every $L' \in NP$.

If language satisfies only condition (2) it said to be **NP-hard**.

Theorem 5 *If any NP-complete problem is poly-time solvable, then $P = NP$. If any problem in NP is not poly-time solvable, then all NP-complete problems are not poly-time solvable.*

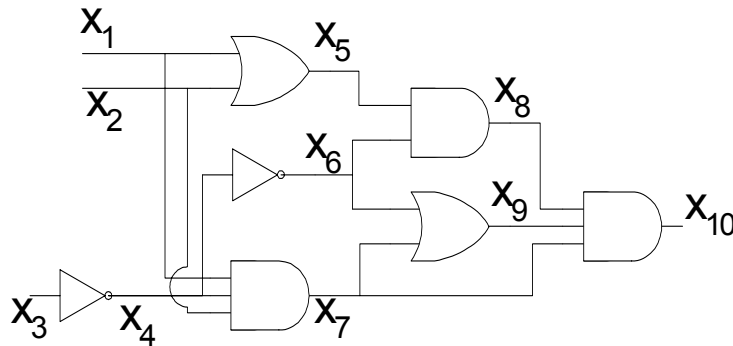
First Problem: **Circuit Satisfiability:** "Given a boolean combinational circuit composed of AND, OR, and NOT gates, (shown below), is it satisfiable?". We say that a one output boolean combinational circuit is satisfiable if has a truth assignment (= a set of boolean input values) that causes the output to be 1. For the circuit below: $x_1 = 1; x_2 = 1, x_3 = 0$ is such an assignment.

Combinational elements: (nodes of this graph):



NOT, AND, OR, NAND, NOR, XOR, AND(3)

Combinational circuit is made up of such elements in an acyclic manner. Example: all lines are directed from left to right.



Boolean Formula satisfiability:

An instance of SAT is a boolean formula ϕ composed of :

1. boolean variables
2. boolean connectives: one or two inputs and one output, such as AND (\wedge), OR (\vee), NOT (\neg), implication (\longrightarrow), if and only if (\longleftrightarrow); and
3. parenthesis

Example:

$$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \longleftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$

A **truth assignment** for a boolean formula ϕ is a set of values for the variables in ϕ and a **satisfying assignment** is a truth assignment that causes it to evaluate to 1. If a satisfying assignment exists for a formula then it is **satisfiable**. **SAT** (satisfiability problem) asks if a given formula is satisfiable. For the example formula, $x_1 = 0; x_2 = 0; x_3 = 1 = x_4$ is a satisfying assignment and hence ϕ above is satisfiable.

Theorem 6 $SAT \in NPC$

Proof.

1. To show that $SAT \in NP$: Certificate is the truth assignment that satisfies the formula. It is easy to check that the size of the certificate and the process of checking that $\phi = 1$ are poly-time.
2. For the second part we show that $CIRCUIT-SAT \leq_p SAT$:

■

Proof. For each wire x_i in the circuit C , we have a variable x_i in SAT. The proper operation of a gate can be expressed as a formula involving both the input and output variables of this gate. We do this for each gate. The formula for SAT is the AND of the circuit output and the conjunction of clauses describing the operation of each gate. For the circuit above the formula is:

$$\begin{aligned} \phi &= x_{10} \wedge (x_4 \longleftrightarrow \neg x_3) \\ &\wedge (x_5 \longleftrightarrow (x_1 \vee x_2)) \\ &\wedge (x_6 \longleftrightarrow \neg x_4) \\ &\wedge (x_7 \longleftrightarrow (x_1 \wedge x_2 \wedge x_4)) \\ &\wedge (x_8 \longleftrightarrow (x_5 \vee x_6)) \\ &\wedge (x_9 \longleftrightarrow (x_6 \vee x_7)) \\ &\wedge (x_{10} \longleftrightarrow (x_7 \wedge x_8 \wedge x_9)) \end{aligned}$$

Given a circuit, it is straight forward to produce the formula in polynomial time. That the formula is satisfiable iff the circuit is satisfiable is easy to see. ■

3-CNF satisfiability (3-SAT):

A **literal** in a boolean formula is a variable or its negation. A boolean formula is in **conjunctive (disjunctive) normal form** denoted by **CNF (DNF)**

if it is expressed as an AND (OR) of clauses, each of which is the OR (AND) of one or more literals. It is known that any boolean formula can be written in either form. **3-CNF** is a CNF in which each clause has exactly 3 literals. **3-CNF-SAT (or 3-SAT)** asks if a given 3-CNF is satisfiable.

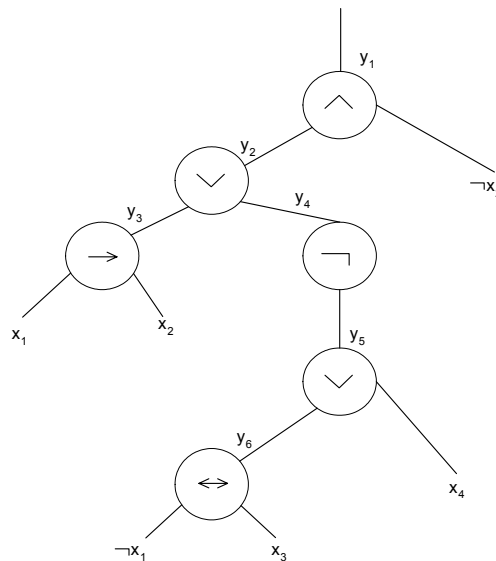
Theorem 7 $3\text{-SAT} \in \text{NPC}$

Proof. The first part is the same as that for SAT. The second part is by showing $\text{SAT} \leq_p 3\text{-SAT}$.

1. First we construct a binary parse tree for the input formula ϕ with literals as leaves and connectives as internal nodes. For example for the formula

$$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \wedge x_4)) \neg x_2$$

we get the tree:



If the formula contains a clause with several literals associativity can be used to parenthesize the expression fully so that every internal node in the resulting tree has 1 or 2 children. This tree is now viewed as a circuit for computing the function. Output of an internal node is denoted by $y_i; i = 1, 2, \dots$. Now we can rewrite the formula as the AND of the root variable and a conjunction of clauses describing the operation of each node.

For the above formula we get:

$$\begin{aligned}
 \phi' &= y_1 \\
 \wedge(y_1 &\longleftrightarrow (y_2 \wedge \neg x_2)) \\
 \wedge(y_2 &\longleftrightarrow (y_3 \vee y_4)) \\
 \wedge(y_3 &\longleftrightarrow (x_1 \rightarrow x_2)) \\
 \wedge(y_4 &\longleftrightarrow \neg y_5)) \\
 \wedge(y_5 &\longleftrightarrow (y_6 \vee x_4)) \\
 \wedge(y_6 &\longleftrightarrow (\neg x_1 \longleftrightarrow x_3))
 \end{aligned}$$

This is a conjunction of clauses each of which has at most three literals.

- In this step, we make each clause an OR of literals. For this purpose, think of ϕ' as the conjunction of ϕ'_i where this is one of the clauses in ϕ . Construct a truth table for ϕ'_i and build a disjunctive normal form for ϕ'_i using the entries that evaluate to 0. For the example, consider $\phi'_1 = (y_1 \longleftrightarrow (y_2 \wedge \neg x_2))$. It's truth table is :

y_1	y_2	x_2	$(y_1 \longleftrightarrow (y_2 \wedge \neg x_2))$
1	1	1	0
1	1	0	1
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	1
0	0	0	1

The DNF for $\neg\phi'_1 = (y_1 \wedge y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge \neg x_2) \vee (\neg y_1 \wedge y_2 \wedge \neg x_2)$. Applying DeMorgan's formula we get $\phi'_1 = (\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2)$ which is in conjunctive normal form. Thus, we get ϕ'' which is equivalent to ϕ'_1 and is in CNF. This has at most three literals for each clause. We need to have exactly three for which we do step 3.

- If a clause has exactly three, we leave it as is.
- If a clause $C_i = (l_1 \vee l_2)$, write it as: $((l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p))$ where p is a new variable.
- If a clause $C_i = (l_1)$ rewrite it as : $((l_1 \vee q \vee p) \wedge (l_1 \vee q \vee \neg p) \wedge (l_1 \vee \neg q \vee p) \wedge (l_1 \vee \neg q \vee \neg p))$.

Reduction is now complete. ■

CLIQUE:

Given an undirected graph $G = [V, E]$, a subset $V' \subseteq V$ is a clique if $[u, v \in V'; u \neq v] \Rightarrow (u, v) \in E$. The clique problem is to find a maximum size clique in a graph.

CLIQUE = $\{(G, k) : G \text{ is a graph with a clique of size } k\}$.

Theorem 8 *CLIQUE* \in *NPC*

Proof. Certificate is V' itself. It is easy to check that V' forms a clique in G in poly-time.

For the rest we show that $3\text{-SAT} \leq_p \text{CLIQUE}$. Let the clauses in an instance of 3-SAT be C_1, C_2, \dots, C_k and let the variables be x_1, x_2, \dots, x_n . The graph $G = [V, E]$ is constructed as follows: For each clause $C_r = (l_1^r \vee l_2^r \vee l_3^r)$ we have three vertices v_1^r, v_2^r, v_3^r in V . There is an edge in E between two vertices v_i^r and v_j^s if the corresponding literals l_i^r and l_j^s are not contradictory and $r \neq s$. G can be computed in poly-time from ϕ .

For each satisfiable assignment to ϕ , there is a literal in each clause that is true and these nodes form a clique in G . Conversely, if a set of nodes in G form a clique, then these come from different clauses and setting these variables to true yields a satisfying assignment. ■

Vertex Cover:

Given an undirected graph $G = [V, E]$, a subset $V' \subseteq V$ is a vertex cover if $[(u, v) \in E] \Rightarrow [(u \in V') \vee (v \in V')]$. The vertex cover problem is to find a vertex cover of minimum size.

VERTEX-COVER = $\{(G, k) : \text{graph } G \text{ has a vertex cover of size } k\}$.

Theorem 9 *VERTEX-COVER* \in *NPC*

Proof. Certificate is the set V' itself. It is easy to check that this a vertex cover in poly-time. make sure that each edge has at least one end in V' .

To show the rest, we show $\text{CLIQUE} \leq_p \text{VERTEX-COVER}$.

An instance $\langle G, k \rangle$ of the clique problem reduces to an instance $\langle \bar{G}, |V| - k \rangle$ of the vertex cover problem. Here $G = [V, E]$ and $\bar{G} = [V, \bar{E}]$ with $[(u, v) \in \bar{E}] \iff [(u, v) \notin E]$. It is easy to check that this reduction works and is obtained in poly-time. ■

SUBSET-SUM:

Given a set S of natural numbers (positive integers) and a target value t (also a natural number/positive integer), is there a subset $S' \subseteq S$ such that $\sum_{x \in S'} x = t$?

SUBSET-SUM: $\{\langle S, t \rangle : \exists S' \subseteq S \ni t = \sum_{x \in S'} x\}$

We assume that numbers are encoded in binary.

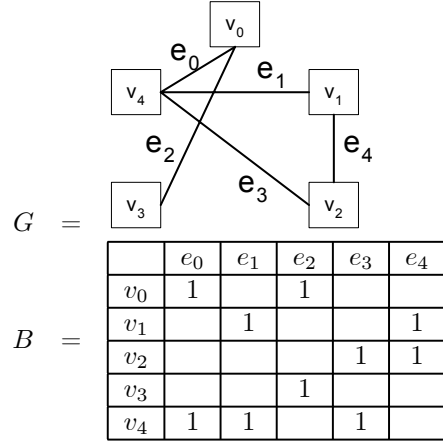
Theorem 10 *SUBSET-SUM* \in *NPC*

Proof. To show that $\text{SUBSET-SUM} \in \text{NP}$: The certificate used is the set S' itself. Checking whether $t = \sum_{x \in S'} x$ can be done in poly-time.

To show the rest we show: $\text{VERTEX-COVER} \leq_p \text{SUBSET-SUM}$.

An instance of VERTEX-COVER is given by a graph G and a number k . We produce an instance $\langle S, t \rangle$ of SUBSET-SUM such that G has a vertex cover of size k iff there is a subset S' of S whose sum is t as follows: Let B be the

vertex-edge incidence matrix of G . For example shown below is an instance of G and its B :



The set S of numbers has one for each vertex of G (these are the x_i), and one for each edge of G (these are the y_j) and these numbers are given as follows:

x_0	1	1		1		
x_1	1		1			1
x_2	1				1	1
x_3	1			1		
x_4	1	1	1		1	
y_0	0					1
y_1	0				1	
y_2	0			1		
y_3	0		1			
y_4	0	1				
t	k	2	2	2	2	2

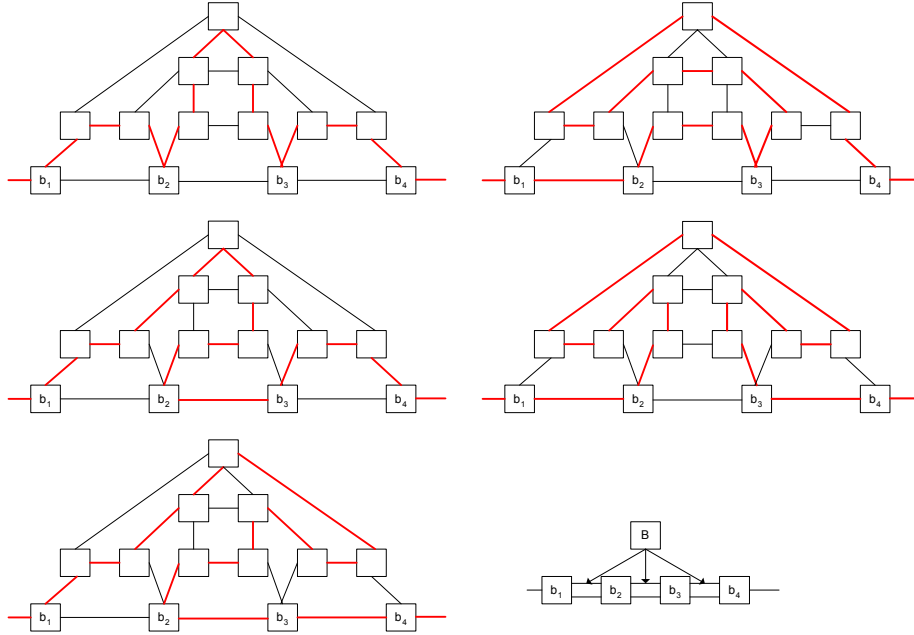
$x_i = r^{|E|} + \sum_{j=0}^{|E|-1} b_{i,j} r^j$; $y_j = r^j$; $t = kr^{|E|} + 2 \sum_{j=0}^{|E|-1} r^j$ where r is any base higher than 2 (say 10 for example). All these numbers have size that is polynomially bounded (in binary encoding) with respect to the size of G . To show that G has a vertex cover of size k iff subset of value t exists:

Suppose $V' \subseteq V$ is a vertex cover of size k . Let $V' = \{v_{i_1}, v_{i_2}, \dots, v_{i_k}\}$. Our choice for S' is

$$S' = \{x_{i_1}, x_{i_2}, \dots, x_{i_k}\} \cup \{y_j : e_j \text{ is incident on precisely one vertex in } V'\}.$$

The leading digit in $\sum_{x \in S'} x$ is clearly k . The next set of digits correspond one each to an edge of G . If both ends are in V' we get a 2 from the sum of x_i and a 0 from the y_j . If only one end is in V' we get a 1 from each part. So the total is same as in t . Thus if V' is a cover, then $\sum_{x \in S'} x = t$.

Conversely suppose $\sum_{x \in S'} x = t$ for some $S' \subseteq S$. Let $S' = \{x_{i_1}, x_{i_2}, \dots, x_{i_m}\} \cup \{y_{j_1}, y_{j_2}, \dots, y_{j_p}\}$. Then clearly $m = k$. Let $V' = \{v_{i_1}, v_{i_2}, \dots, v_{i_m}\}$. If both ends

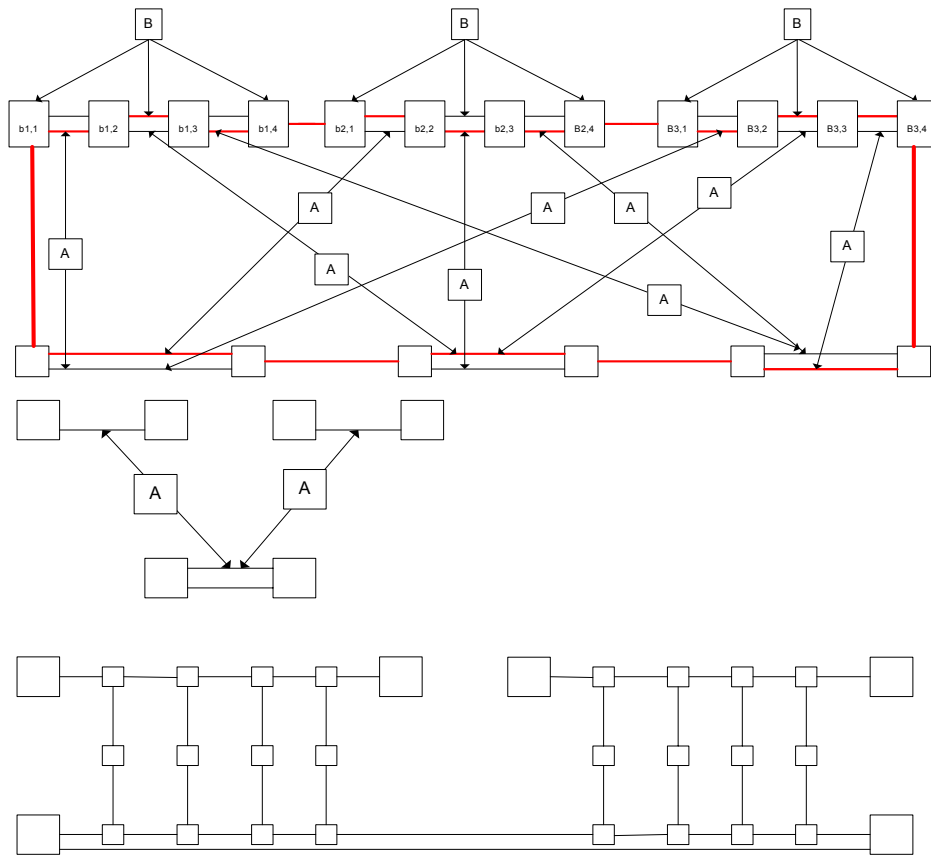


Starting from a 3-CNF ϕ , we construct the graph G as follows: For each of the k clauses in ϕ , we have a widget type B and these are joined in series as shown in the picture below. Then for each variable x_m in ϕ , we include two vertices x'_m and x''_m . These are connected by two edges e_m and \bar{e}_m . Each of these two-edge-loops is connected in series by adding edges (x''_m, x'_m) for $m = 1, 2, \dots, n - 1$. We connect the top (clause) side of the graph to the bottom (variable) side by means of two edges $(b_{1,1}, x'_1)$ and $(b_{k,4}, x''_n)$ which are the bold edges in the figure shown.

If the j^{th} literal of clause C_i is x_m (resp. $\neg x_m$), then we use the A-widget to connect $(b_{i,j}, b_{i,j+1})$ with the edge e_m (resp. \bar{e}_m). A given literal l_m may appear in several clauses, and thus an edge e_m or \bar{e}_m may be influenced by several A-widgets. In this case, we connect the A-widgets in series as shown, effectively replacing the edge e_m or \bar{e}_m by a series of edges. This completes the description of the construction of G .

The figure below shows the construction for

$$\phi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3)$$



Now we need to verify that the two problems have the same outcomes. ■