Intractability: Checking Algorithms

Time Complexity

- $\log n$
- n
- $n \log n$
- n^2
- n^3
- 2^n
- $n!$

Typical Functions T(n)

Input	log n	n	n log n	n^2	n^3	2^n	n!
10	3.3	10	33	100	1000	1000	10^6
100	6.6	100	66	10^4	10^6	10^{30}	10^{157}
1000	10	1000	10^4	10^6	10^9		
10^4	13	10^4	10^5	10^8	10^{12}		
10^5	17	10^5	10^6	10^{10}			
10^6	20	10^6	10^7				
10^7	23	10^7	10^8				
10^8	27	10^8	10^9				
10^9	30	10^9	10^{10}				
10^{10}	33	10^{10}					

10^9 operations is considered to be fine

– 10 secs

10^{10} is too long

Intractability

- Till now we were discussing about identifying efficient solution to problems
- But for some problems, no known efficient solutions exits
- We should be able to recognize this
- So that we do not fruitlessly try to look for solutions
- those problems are known to be hard to solve
- Let us look at some issues involving intractability

Intractability

- Problems that are solvable by polynomial–time algorithms as being tractable, or easy, and problems that require superpolynomial time as being intractable, or hard

Exponential Search Space

- Many problems that we have seen, we are trying to search through a number of possibilities to arrive as some kind of optimum combination
- actual search space is exponential
- For shortest paths, there are an exponential number of paths
- For a minimum cost spanning tree, there are an exponential number of such spanning trees to search through

Exponential Search Space

- For maximum flow, we have many different ways of adding and subtracting flow along edges
- Looking at all possible flows, all possible paths, all possible spanning trees and then choose the optimum – Brute force solution – would take exponential time
- If we have a polynomial time algorithm, we are actually cutting through exponentials and in some very drastic way
- Reducing search space from an exponential to polynomial

Exponential to Polynomial

- We work hard to find efficient short cut, where we can cut through the exponential space of possibilities and quickly narrow down the space to a polynomial number of realistic
- There are many problems, for which efficient algorithm do not exist or are not known to exist
- Many of these problems are extremely important practical problems

Generating a solution Vs Checking a solution

- A school Math's teacher assigns a homework
- take a large number which is known to be product of two prime numbers and find these two prime numbers
- From the student point of view obviously, the problems is to generate the solution
- So, given the large number N, the student is expected to find two prime numbers p and q, such that p times q is equal to N

Generating a solution Vs Checking a solution

- student submits their solutions to teacher for evaluation
- Teacher does not have to keep generating p and q
- Teacher has does not even need to know the answer
- Teacher can just take answer given by student,
  - Check if p and q are prime
  - Multiply p times q and determine whether p times q is equal to N or not
- teacher is checking solution that student was trying to generate

Class P Vs NP

- Class P consists of those problems that are solvable in polynomial time
- they are problems that can be solved in time $O(n^k)$ for some constant k, where n is the size of the input to the problem.

Class P Vs NP

- class NP consists of those problems that are “verifiable” in polynomial time
- What do we mean by a problem being verifiable?
- If we were somehow given a “certificate” of a solution, then we could verify that the certificate is correct in time polynomial in the size of the input to the problem

Class P Vs NP

- For example, in the hamiltonian–cycle problem, given a directed graph G =(V, E), a certificate would be a sequence $<v_1, v_2, v_3, \ldots, v_{|V|}>$i of |V| vertices
- We could easily check in polynomial time that $(v_i, v_{i+1}) \in E$ as well
- As another example, for 3–CNF satisfiability, a certificate would be an assignment of values to variables.

Class P Vs NP

- We could check in polynomial time that this assignment satisfies the boolean formula
- Any problem in P is also in NP
- Since if a problem is in P then we can solve it in polynomial time without even being supplied a certificate

Class NPC

- Informally, a problem is in the class NPC—and we refer to it as being NP–complete—if it is in NP and is as “hard” as any problem in NP

Overview of showing problems to be NP–complete

- Informally, a problem is in the class NPC—and we refer to it as being NP–complete—if it is in NP and is as “hard” as any problem in NP
- When we demonstrate that a problem is NP–complete, we are making a statement about how hard it is
- We are not trying to prove the existence of an efficient algorithm, but instead that no efficient algorithm is likely to exist

Key concepts in showing a problem to be NP–complete

- Decision problems vs. optimization problems:
- Optimization problems – Each feasible (i.e., “legal”) solution has an associated value, and we wish to find solution with best value
- Eg: SHORTEST–PATH – given an undirected graph G and vertices u and v, and we wish to find a path from u to v that uses the fewest edges

Key concepts in showing a problem to be NP-complete

- NP-completeness applies directly not to optimization problems, however, but to decision problems, in which the answer is simply "yes" or "no"
- We usually can cast a given optimization problem as a related decision problem by imposing a bound on the value to be optimized

Key concepts in showing a problem to be NP–complete

- For example, a decision problem related to SHORTEST–PATH is PATH: given a directed graph G, vertices u and v, and an integer k, does a path exist from u to v consisting of at most k edges?

Key concepts in showing a problem to be NP–complete

- The relationship between an optimization problem and its related decision problem works in our favor when we try to show that the optimization problem is “hard.”
- That is because the decision problem is in a sense “easier,” or at least “no harder.
- If decision problem is easy then is optimization problem and if decision problem is hard then is optimization problem

Reductions

- Idea of showing that one problem is no harder or no easier than another applies even when both problems are decision problems
- We use this idea in almost every NP–completeness proof
- Consider a decision problem A, which we would like to solve in polynomial time
- Input to a particular problem is an instance of that problem

Reductions

- In PATH, an instance would be a particular graph G, particular vertices u and v of G, and a particular integer k
- We already know how to solve a different decision problem B in polynomial time
- We have a procedure that transforms any instance α of A into some instance β of B with the following characteristics:

Reductions

- The transformation takes polynomial time.
- The answers are the same. That is, the answer for α is "yes" if and only if the answer for β is also "yes."

Reductions

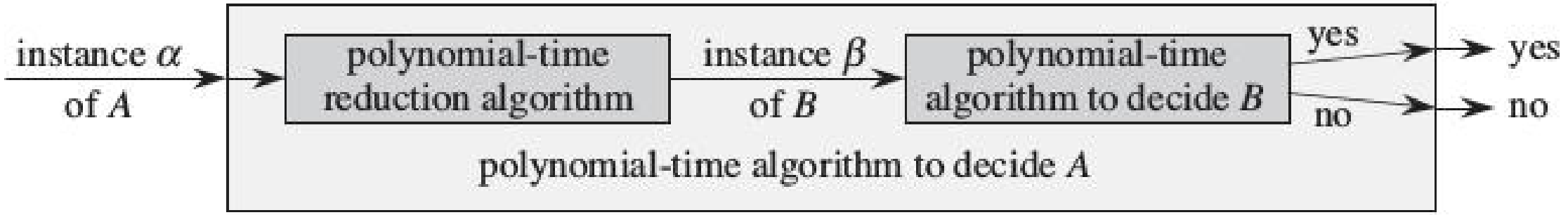


Figure 34.1 How to use a polynomial-time reduction algorithm to solve a decision problem A in polynomial time, given a polynomial-time decision algorithm for another problem B. In polynomial time, we transform an instance α of A into an instance β of B, we solve B in polynomial time, and we use the answer for β as the answer for α.

Reductions

1. Given an instance α of problem A, use a polynomial–time reduction algorithm to transform it to an instance β of problem B
2. Run the polynomial–time decision algorithm for B on the instance β
3. Use the answer for β as the answer for α

NP – completness – Use reduction in opposite way

- We have a polynomial–time reduction transforming instances of A to instances of B
- Use a simple proof by contradiction to show that no polynomial–time algorithm can exist for B
- Assume that B has a polynomial–time algorithm
- Then we would have a way to solve problem A in polynomial time, which contradicts our assumption that there is no polynomial–time algorithm for A

NP – completness – Use reduction in opposite way

- We cannot assume that there is absolutely no polynomial–time algorithm for problem A
- The proof methodology is similar, however, in that we prove that problem B is NP–complete on the assumption that problem A is also NP–complete

A first NP–complete problem

- We need a problem already known to be NP–complete in order to prove a different problem NP–complete
- Circuit–satisfiability problem, in which we are given a boolean combinational circuit composed of AND, OR, and NOT gates, and we wish to know whether there exists some set of boolean inputs to this circuit that causes its output to be 1

Abstract problems

- An abstract problem Q is a binary relation on a set I of problem instances and a set S of problem solutions
- Eg: an instance for SHORTEST–PATH is a triple consisting of a graph and two vertices
- A solution is a sequence of vertices in the graph, with perhaps the empty sequence denoting that no path exists

Abstract problems

- SHORTEST–PATH problem is a relation that associates each instance of a graph and two vertices with a shortest path in the graph that connects the two vertices
- Since shortest paths are not necessarily unique, a given problem instance may have more than one solution
- theory of NP–completeness restricts attention to decision problems: those having a yes/no solution

Abstract problems

- we can view an abstract decision problem as a function that maps the instance set I to the solution set {0, 1}
- For example, a decision problem related to SHORTEST–PATH is the problem PATH
- If i = <G, u, v, k> is an instance of the decision problem PATH, then PATH(i) = 1 (yes) if a shortest path from u to v has at most k edges, and PATH(i) = 0 (no) otherwise

Abstract problems

- Many abstract problems are not decision problems, but rather optimization problems, which require some value to be minimized or maximized
- However we can usually recast an optimization problem as a decision problem that is no harder

NP – Completeness

A language $L \subseteq \{0, 1\}^*$ is NP–complete if

1. $L \in NP$, and

2. $L' \leqslant_P L$ for every $L' \in NP$

If a language L satisfies property 2, but not necessarily property 1, we say that L is NP–hard

NP – Completeness

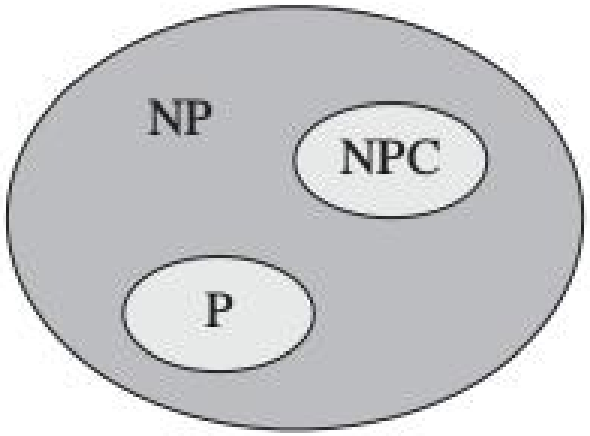


Figure 34.6 How most theoretical computer scientists view the relationships among P, NP, and NPC. Both P and NPC are wholly contained within NP, and $P \cap NPC = \emptyset$.

Circuit satisfiability

- Boolean combinational circuits are built from boolean combinational elements that are interconnected by wires
- A boolean combinational element is any circuit element that has a constant number of boolean inputs and outputs and that performs a well–defined function
- Boolean values are drawn from the set {0, 1}, where 0 represents FALSE and 1 represents TRUE

NP – Completeness

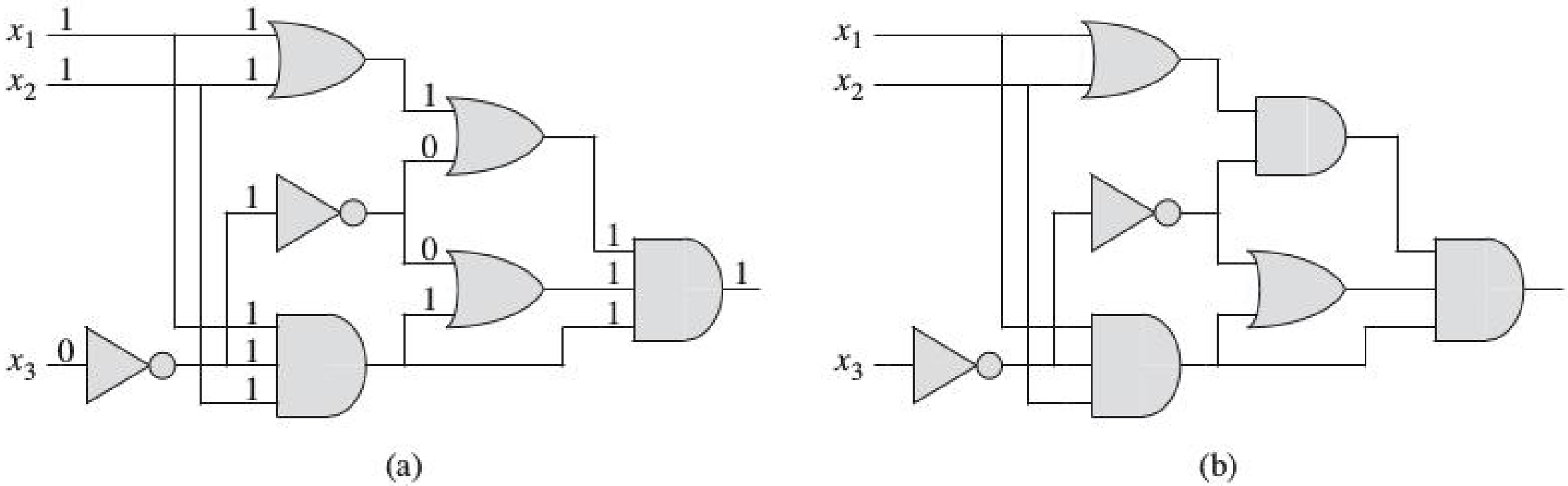


Figure 34.8 Two instances of the circuit-satisfiability problem. **(a)** The assignment $\langle x_1 = 1, x_2 = 1, x_3 = 0 \rangle$ to the inputs of this circuit causes the output of the circuit to be 1. The circuit is therefore satisfiable. **(b)** No assignment to the inputs of this circuit can cause the output of the circuit to be 1. The circuit is therefore unsatisfiable.

Circuit satisfiability

- CIRCUIT-SAT = {<C>: C is a satisfiable boolean combinational circuit}
- arises in the area of computer-aided hardware optimization
- If a subcircuit always produces 0, that subcircuit is unnecessary;
- the designer can replace it by a simpler subcircuit that omits all logic gates and provides the constant 0 value as its output

Circuit–satisfiability problem belongs to the class NP

Two–inputs for Algorithm:

- (i) A standard encoding of a boolean combinational circuit C
- (ii) Certificate corresponding to an assignment of boolean values to the wires in C

Circuit–satisfiability Algorithm

For each logic gate in the circuit:

- Check value provided by certificate on output wire is correctly computed as a function of the values on the input wires
- If output of entire circuit is 1, the algorithm outputs 1
- Otherwise, A outputs 0

Circuit–satisfiability Algorithm

- Whenever a satisfiable circuit C is input to algorithm A, there exists a certificate whose length is polynomial in size of C and causes A to output a 1.
- Whenever an unsatisfiable circuit is input, no certificate can fool A into believing that the circuit is satisfiable
- Algorithm A runs in polynomial time: with a good implementation, linear time suffices.
- Thus CIRCUIT–SAT $\in$ NP

CIRCUIT–SAT is NP–complete

- Show that the language is NP–hard
- Must show that every language in NP is polynomial–time reducible to CIRCUIT–SAT
- Outline of proof only given

CIRCUIT–SAT is NP–complete

- Idea of proof is to represent computation of A as a sequence of configurations
- Break each configuration into parts consisting of the program for A, the program counter and auxiliary machine state, the input x, the certificate y, and working storage
- Combinational circuit M, which implements computer hardware, maps each configuration c_i to the next configuration c_{i+1}, starting from the initial configuration c_0

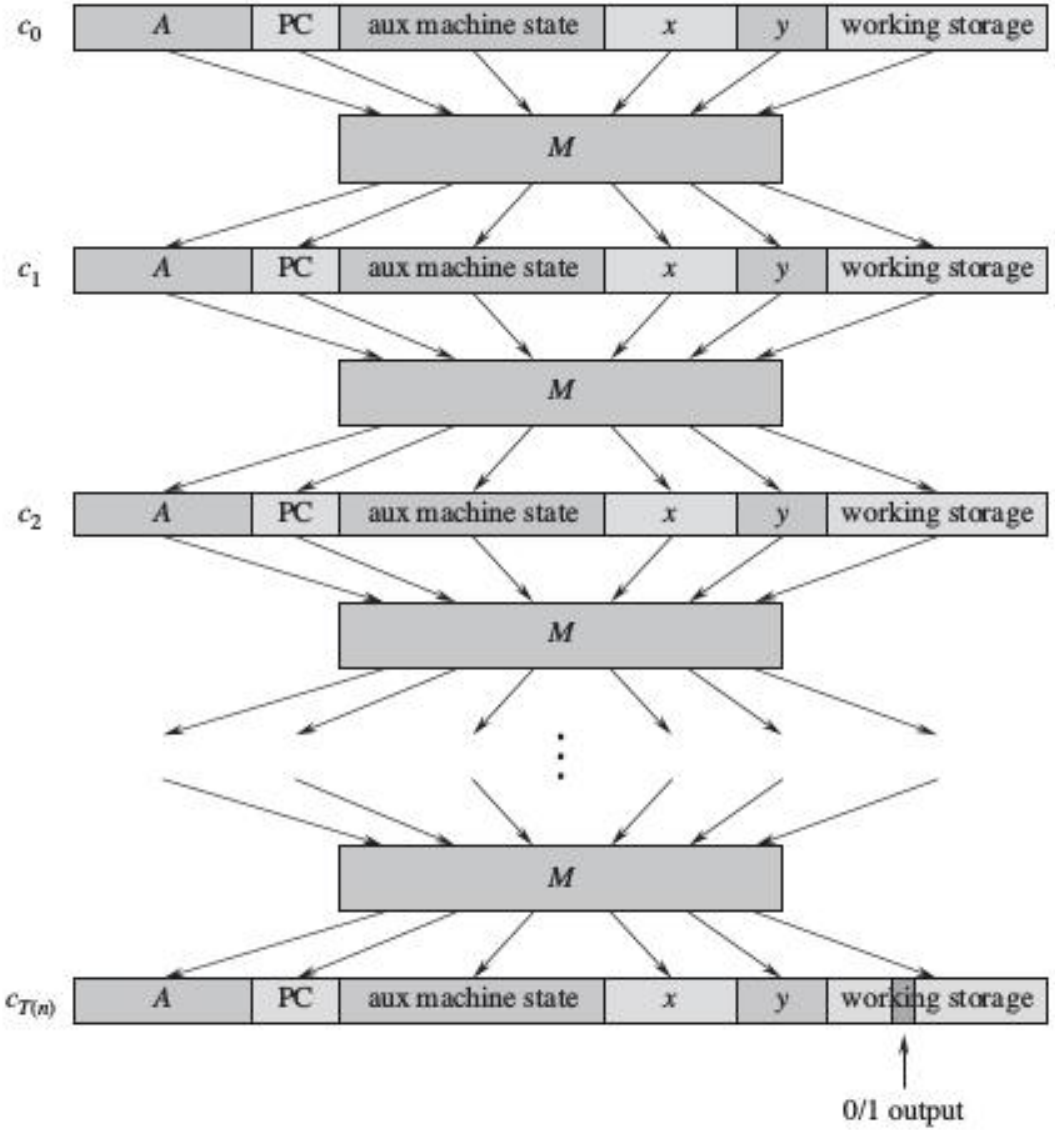


Figure 34.9 The sequence of configurations produced by an algorithm A running on an input x and certificate y. Each configuration represents the state of the computer for one step of the computation and, besides A, x, and y, includes the program counter (PC), auxiliary machine state, and working storage. Except for the certificate y, the initial configuration c_0 is constant. A boolean combinational circuit M maps each configuration to the next configuration. The output is a distinguished bit in the

CIRCUIT–SAT is NP–complete

- Algorithm A writes its output—0 or 1—to some designated location by the time it finishes executing, and if we assume that thereafter A halts, the value never changes
- Reduction algorithm F constructs a single combinational circuit that computes all configurations produced by a given initial configuration

CIRCUIT–SAT is NP–complete

- Idea is to paste together T (n) copies of the circuit M
- Output of the ith circuit, which produces configuration c_i, feeds directly into the input of the (i +1)st circuit
- configurations, rather than being stored in the computer's memory, simply reside as values on the wires connecting copies of M

CIRCUIT–SAT is NP–complete

- Polynomial–time reduction algorithm F must do
- Given an input x, it must compute a circuit C = f(x) that is satisfiable if and only if there exists a certificate y such that A(x, y) = 1
- When F obtains an input x, it first computes n = |x| and constructs a combinational circuit C' consisting of T (n) copies of M

CIRCUIT–SAT is NP–complete

- Input to C' is an initial configuration corresponding to a computation on A(x, y), and the output is the configuration $c_{T(n)}$
- Algorithm F modifies circuit C' slightly to construct the circuit C = f (x). First, it wires the inputs to C' corresponding to the program for A, the initial program counter, the input x, and the initial state of memory directly to these known values.
- Thus, the only remaining inputs to the circuit correspond to the certificate y

CIRCUIT–SAT is NP–complete

- Second, it ignores all outputs from C' , except for the one bit of $c_{T(n)}$ corresponding to the output of A
- This circuit C , so constructed, computes C(y) = A(x, y) for any input y of length $O(n^k)$
- The reduction algorithm F , when provided an input string x, computes such a circuit C and outputs it.

CIRCUIT–SAT is NP–complete

- Show that F runs in time polynomial in n = |x|
- First observation we make is number of bits required to represent a configuration is polynomial in n.
- Program for A itself has constant size, independent of the length of its input x
- Length of the input x is n, and the length of the certificate y is $O(n^k)$

CIRCUIT–SAT is NP–complete

- Since the algorithm runs for at most $O(n^k)$ steps, the amount of working storage required by A is polynomial in n as well.
- We assume that this memory is contiguous

Satisfiability Problem (SAT)

- Very canonical problem which has a checking algorithm called Boolean satisfactory
- we have some Boolean variables x, y and z.
- Boolean variables take true or false
- We have a standard operations on Boolean variables, negation takes a value of true and transit to false and vice versa

Satisfiability Problem (SAT)

- !x – negation of x
- x || y – x or y
- x & y – x and y
- Lierals – x, !x, ...
- Clause – Disjunction of literals
- formula C of the form – (x || y || z || ... || w)
- Conjunction of clauses – Boolean formula

Satisfiability Problem (SAT)

- Problem – to find suitable values (True or False) for variables x, y, z so that the formula evaluates to true
- (x || y || z) & (x || !y) & (y || !z) & (!x || !y || !z)
- x = true, y = true, z = false makes this true
- (x || y || z) & (x || !y) & (y || !z) & (z || !x) & (!x || !y || !z)
- Now there is no satisfying assignment

Satisfiability Problem (SAT)

- Generating solutions
- Try each possible assignment to x, y, z, ...
- N variables – 2^N possible assignment
- Is there a better algorithm? Not known
- Checking a solution – Given a formula F and a valuation V(x) for each x, substitute into formula and evaluate

Satisfiability Problem (SAT)

- Other way of writing boolean formula is disjunction of clauses
- Each clause is conjunction of literals
- (x & !y & z & ... & w) – Each clause forces a unique solution

2–CNF satisfiability vs. 3–CNF satisfiability

- a boolean formula is in k–conjunctive normal form, or k–CNF, if it is the AND of clauses of ORs of exactly k variables or their negations
- (x || y) & (!x || z) & (!y || !z) – 2 – CNF
- x = true, y = false, z = true
- This is a satisfying assignment

3 CNF is Complete

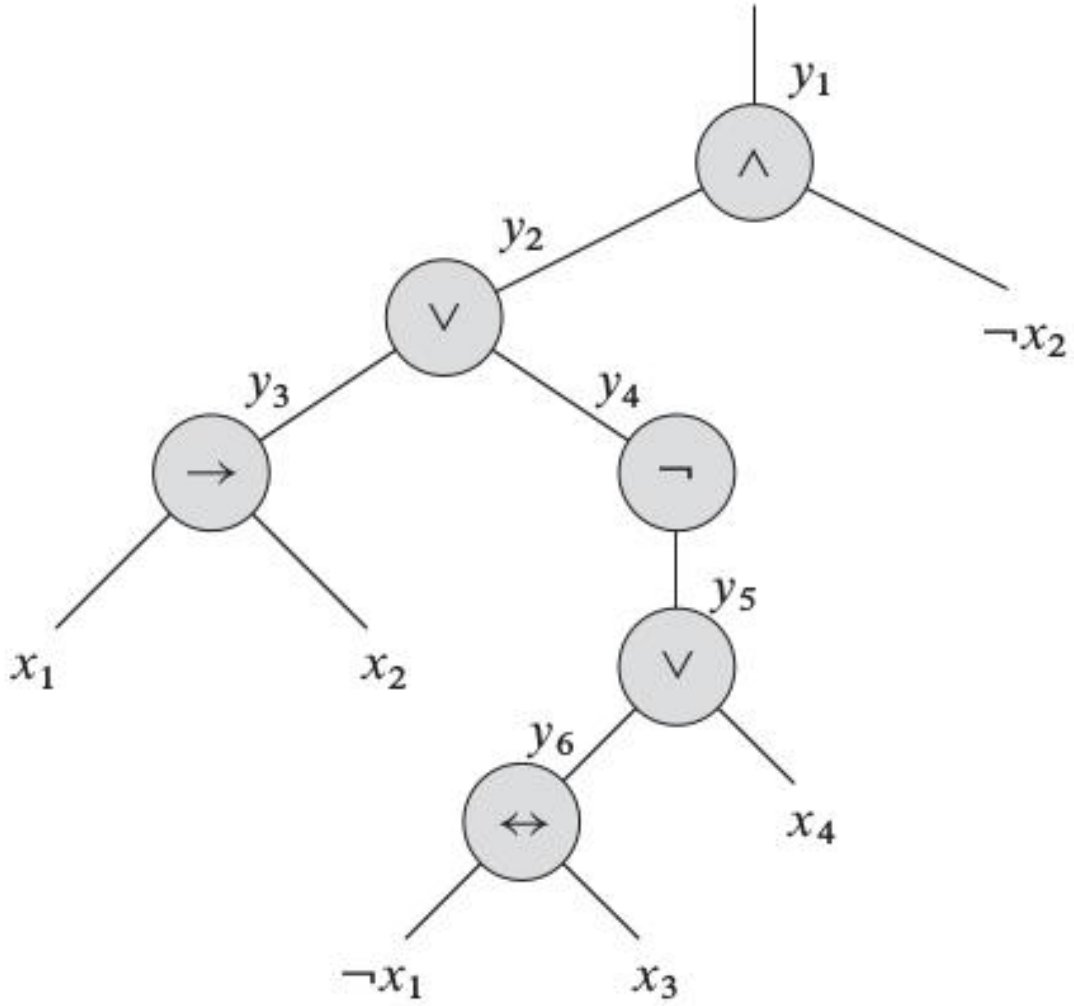
- Satisfiability of boolean formulas in 3–conjunctive normal form is NP–complete
- Step 1: Consruct a binary parse tree for input formula
- Step 2: Converts boolean formula to conjunction of clauses – DNF
- Step 3: Transforms formula so that each clause has exactly 3 distinct literals

3 CNF is Complete – Step 1

- Construct a binary "parse" tree for the input formula Φ, with literals as leaves and connectives as internal nodes

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

3 CNF is Complete – Step 1



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

Figure 34.11 The tree corresponding to the formula $\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$.

3 CNF is Complete – Step 2

- converts each clause Φ' into conjunctive normal form
- We construct a truth table for Φ' by evaluating all possible assignments to its variables
- Using the truth–table entries that evaluate to 0, we build a formula in disjunctive normal form (or DNF)—an OR of ANDs—that is equivalent to : $\neg\Phi'$

3 CNF is Complete – Step 2

- We then negate this formula and convert it into a CNF formula Φ" by using DeMorgan's laws for propositional logic,
- ¬(a ^ b) = ¬a ⋁ ¬b
- ¬(a ⋁ b) = ¬a ^ ¬b
- to complement all literals, change ORs into ANDs, and change ANDs into ORs

3 CNF is Complete – Step 2

- We then negate this formula and convert it into a CNF formula Φ" by using DeMorgan's laws for propositional logic,
- ¬(a ^ b) = ¬a ⋁ ¬b
- ¬(a ⋁ b) = ¬a ^ ¬b
- to complement all literals, change ORs into ANDs, and change ANDs into ORs

3 CNF is Complete – Step 2

- convert the clause $\phi'_1 = (y_1 \leftrightarrow (y_2 \wedge \neg x_2))$ into CNF as follows

y_1	y_2	x_2	$(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$
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

Figure 34.12 The truth table for the clause $(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$.

- DNF formula equivalent to $\neg \Phi'$ is

$$(y_1 \wedge y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge \neg x_2) \vee (\neg y_1 \wedge y_2 \wedge \neg x_2)$$

3 CNF is Complete – Step 2

- Negating and applying DeMorgan's laws, we get the CNF formula

$$\begin{aligned}\phi_1'' \quad = \quad & (\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \\ & \wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2) \ ,\end{aligned}$$

- which is equivalent to the original clause Φ_1'

3 CNF is Complete – Step 2

- At this point, we have converted each clause Φ_i' of the formula Φ' into a CNF formula Φ_i'', and thus Φ' is equivalent to the CNF formula Φ'' consisting of the conjunction of the Φ_i''
- Moreover, each clause of Φ'' has at most 3 literals

3 CNF is Complete – Step 3

- Transforms the formula so that each clause has exactly 3 distinct literals
- We construct the final 3–CNF formula Φ''' from the clauses of the CNF formula Φ''
- formula Φ''' also uses two auxiliary variables that we shall call p and q

3 CNF is Complete – Step 3

- For each clause C_i of Φ'' , we include the following clauses in Φ''':
- If C_i has 3 distinct literals, then simply include C_i as a clause of Φ'''
- If C_i has 2 distinct literals, that is, if $C_i = (l_1 \vee l_2)$, where l_1 and l_2 are literals, then include $(l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$ as clauses of Φ'''

3 CNF is Complete – Step 3

- If C_i has just 1 distinct literal l, then include (l ∨ p ∨ q) ^ (l ∨ p ∨ ¬q) ^ (l ∨ ¬p ∨ q) ^ (l ∨ ¬p ∨ ¬q) as clauses of Φ'''
- 3–CNF formula Φ''' is satisfiable if and only if Φ is satisfiable

3 CNF is Complete Reduction happens in Polynomial time

- Constructing Φ'' from Φ' introduces at most 1 variable and 1 clause per connective in Φ
- Constructing Φ'' from Φ' can introduce at most 8 clauses into Φ'' for each clause from Φ' , since each clause of Φ' has at most 3 variables, and the truth table for each clause has at most $2^3 = 8$ rows.

3 CNF is Complete Reduction happens in Polynomial time

- The construction of Φ'' from Φ' introduces at most 4 clauses into Φ''' for each clause of Φ''
- Thus, the size of the resulting formula Φ''' is polynomial in the length of the original formula
- Each of the constructions can easily be accomplished in polynomial time.

3 CNF is Complete – Step 1

- Construct a binary "parse" tree for the input formula Φ, with literals as leaves and connectives as internal nodes

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

Traveling Salesman Problem (TSP)

- Salesman is supposed to visit a network of cities and between each pair of cities, we have a distance
- we can think of it is a complete graph, every city connected to every other city
- each edge between two cities, there is a weight indicating the distance or time or some quantity which the sales man has to use in order to travel from the one city to next city

Traveling Salesman Problem (TSP)

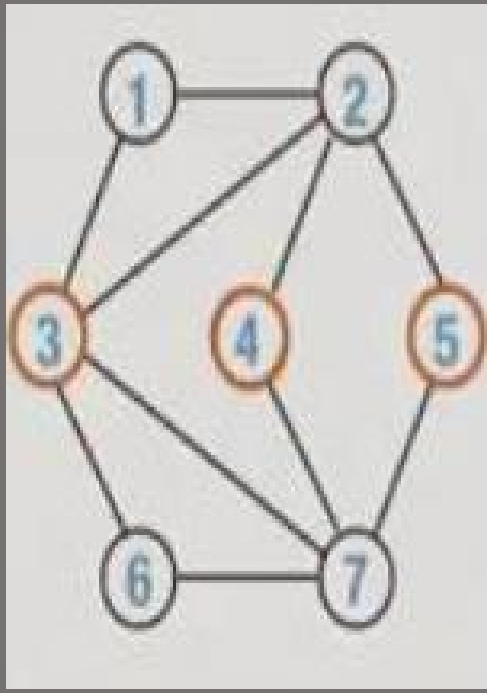
- Goal is to visit every city on this map
- Salesman wants to find the shortest tour that visits each city exactly once
- no simple way to write a generative an algorithm which will actually analyzes and find a good solution
- Is there a checking solution, is there a checking algorithms

Traveling Salesman Problem (TSP)

- Given graph and cycle, check if the cycle is valid and compute cost of cycle
- How to check it is of least cost?
- Is there a tour with a cost at most k?
- For original problem, cost is at most sum of all edges
- Find optimum k, test different values using binary search
- Initially lower bound is 0 and upper bound is sum of all edges

Independent Set

- Edge u, v are independent if there is no edge (u,v)
- U is an independent set if each pair (u,v) in U is independent
- Eg: Form a neutral committee where no member knows each other
- Find the largest independent set in a given graph



Independent Set

- Checking version – Is there an independent set of size k?
- Is there a independent set of size 4?