Problems and Algorithms: Classes of problems

Nandini Mukherjee

Department of Computer Science and Engineering

Problems and Algorithms

- We know about complexity of algorithms.
- We could compare different algorithms (e.g. $\Theta(n^2)$ versus $\Theta(n \log n)$ for sorting)
- But are we sure that we had the best possible algorithm?
- We must look for the inherent complexity in problems.
  - how "hard" is a problem?
  - what are the minimum resources required to solve a problem?
- We shall consider three classes of problems
  - P: problems solvable in polynomial time.
  - NP: problems verifiable in polynomial time.
  - NPC: problems in NP and as hard as any problem in NP.

However, we shall study these classes in the context of *decision problems*

What are decision problems?

- The statement of a decision has two parts:
  - *The instance description part defines the information expected in the input*
  - *The question part states the actual yes-or-no question, which contains variables defined in the instance description*

e.g. given an undirected graph G and a positive integer k, is there a colouring of G using at most k colours?

But more often we are interested in optimisation problems.

So how can we apply the concept of NP-completeness?

Fortunately, there is a convenient relationship between optimisation problems and decision problems

Like the above problem is actually an optimisation problem stated as:

Given an undirected graph G = (V,E), produce an optimal colouring.

Optimisation and decision problems

- Example – Knapsack
  - Optimisation problem – *Find the largest total profit of any subset of the objects that fit in the knapsack*
  - Decision problem – *Given k, is there a subset of objects that fits in the knapsack and has total profit at least k?*
- Example – Shortest Path
  - Optimisation problem – *Given graph G, u,v, find a path from u to v with fewest edges*
  - Decision problem – *Given graph G, u,v, and k, whether there exists a path from u to v consisting of at most k edges?*
- Example – Traveling Salesperson
  - Optimisation problem – *Given a complete, weighted graph, find a minimum weight Hamiltonian cycle*
  - Decision problem – *Given a complete, weighted graph and integer k, is there a Hamiltonian cycle with total weight at most k?*

Optimisation and decision problems

- Decision is easier (i.e., no harder) than optimization
- If there is an algorithm for an optimization problem, the algorithm can be used to solve the corresponding decision problem
- Relationship between the optimisation problems and decision problems actually helps us –
  - If an optimisation problem is easy, its related decision problem is easy as well.
  - If we can provide evidence that a decision problem is hard, we also provide evidence that its related optimisation problem is hard.

The Class *P*

P is the class of decision problems that can be deterministically solved in polynomial time

- An algorithm is *polynomially bounded* if its worst-case complexity is bounded by a polynomial function of the input size.
- A problem is *polynomially bounded* if there exists a polynomially bounded algorithm for it.
- *P* is the class of decision problems that are polynomially bounded
- *i.e. if input size is n, then worst-case running time is $O(n^c)$ for some constant c.*

The Class *P*

Why do we use polynomial bound?

- If a problem is not in *P*, it will be extremely expensive and probably impossible to solve in practice
- Any algorithm built from several polynomially bounded algorithms in various natural ways (addition, multiplication and composition) is also polynomially bounded – nice closure properties
- It is independent of the particular formal model of the computation used

- There are problems for which
  - no polynomial algorithm is known
  - a superpolynomial algorithm is known
  - we have not (yet?) proved a superpolynomial lower bound.
  - we do not know whether or not those problems are in P.

$\Theta(n^{100})$ vs. $\Theta(2^n)$ – very few problems are actually has algorithms with complexity $\Theta(n^{100})$

The Class *NP*

- *NP* is the class of decision problems for which there is a polynomially bounded nondeterministic algorithm.
- Nondeterministic algorithms:
  - A *nondeterministic algorithm* has two phases and an output step:
    - The nondeterministic “guessing” phase, and
    - The deterministic “verifying” phase
    - The guessing phase generates an arbitrary string of characters *s* (a certificate) - is it a solution?
    - The verifying phase returns a value *true*, if *s* is a solution, else it returns a *false* value
    - If the verifying phase returns *true*, the algorithm outputs *yes*. Otherwise, there is no output.

Nondeterministic graph colouring

- The string s can be interpreted as a list of integers $c_1, c_2, c_3, \ldots, c_q$
- Assign c_i to v_i
- In the second phase:
  - Check that there are n colours listed (i.e. $q=n$)
  - Check that each c_i is in the range $1, \ldots, k$
  - Scan the list of edges in the graph and for each edge $v_i v_j$ check that $c_i \neq c_j$

The Class *NP*

- A nondeterministic algorithm is said to be polynomially bounded if there is a fixed polynomial *p* such that for each input of size *n* for which the answer is *'yes'*, there is some execution of the algorithm that produces a *yes* output in at most *p(n)* steps.

In other words, *NP* is the class of problems that are "verifiable" in polynomial time, i.e. we can verify in $O(n^c)$ time whether a proposed solution is correct.

We can certainly say that P $\subseteq$ NP

Classes *P* and *NP*

$P \subseteq NP$

Does $NP \subseteq P$?

whether $P = NP$ or $P \neq NP$?

$P \neq NP$: one of the deepest, most perplexing open research problems in theoretical computer science since 1971.

Deciding if a problem is in *P*

- To show that a problem A is in *P*
  - We can prove a $O(n^c)$ bound for A directly (This is hard)
  - We can find a $O(n^c)$ algorithm for A
  - We can show that A is reducible to some problem already known to be in *P*.

What do we mean by 'reducible'? – discussed later.

- To show a problem A is not in *P*,
  - We can prove a $\Omega(f(n))$ bound for some superpolynomial $f(n)$ (e.g. $f(n) = 2^n$).
  - We can show some problem already known not to be in *P* is reducible to A.

NP-complete problems

- This is the class of problems for which it is *unknown* whether a polynomial-time algorithm does exist
  - No polynomial time algorithm has yet been discovered for an NPC problem
  - No one has proven that no polynomial time algorithm exist for any of them

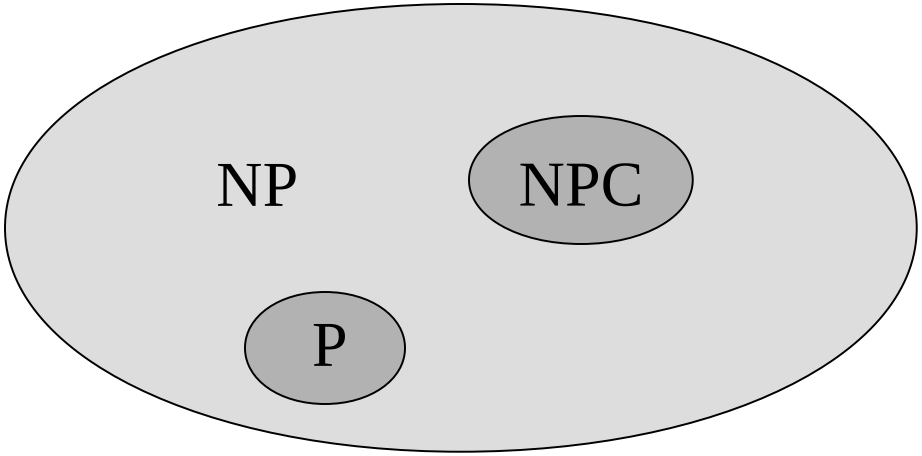
If any *NPC* problem can be solved in polynomial time then *every NPC* problem has polynomial time algorithm

Why study NPC problems?

  - If a problem is *NPC*, it is highly unlikely to find a P-time algorithm to solve it
  - Computer scientists believe that NPC is intractable (i.e., hard, and P ≠ NP)
  - Instead of wasting time on finding an efficient algorithm, try to design approximation algorithms to solve the problem
  - Find heuristics that give correct answer in many cases

Classes *P, NP and NPC*

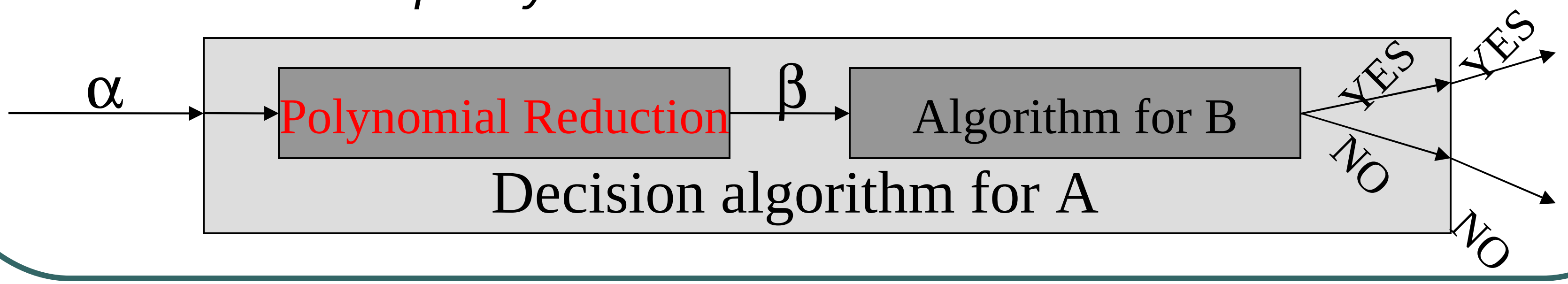
- View of Computer Scientists:



$$P \subset NP, NPC \subset NP, P \cap NPC = \emptyset$$

Polynomial reductions

- Consider problems A and B
- Consider a mapping from any instance α of A to an instance β of B
- This transformation is polynomial reduction
  - If it is "fast", i.e. takes place in *polynomial time*
  - If it is "answer-preserving"
    - *The answer of A for α is "yes" if and only if the answer of B for β is "yes"*



Polynomial reductions

- What is its implication?
  - If decision algorithm for B is polynomial, so does A
    - $B \in P$ implies $A \in P$
  - A is no harder than B (or B is no easier than A)
  - If A is hard (e.g. NPC), so does B
    - $A \notin P$ implies $B \notin P$

NP-Complete problems

- The term *NP-complete* describes the decision problems that are the hardest ones in *NP*
- A problem p is *NP-complete* if
  1. $p \in$ NP and
  2. $p' \leq_p p$ for every $p' \in$ NP

(if p satisfies 2, then p is said *NP-hard*)

- ***i.e.***
  - A problem **Q** in *NP-hard* if every problem **P** in *NP* is reducible to **Q**
    - *NP-hard* puts a lower-bound on the complexity of the problem
    - An *NP-hard* problem may itself not be an *NP* problem
  - A problem **Q** is *NP-complete* it is in *NP* and is *NP-hard*

NP-Complete problems

- How to prove a problem B to be NPC?
  - At first, prove B is in NP, which is generally easy
  - Find a already proved NPC problem A
  - Establish a polynomial reduction from A to B

Question: What is and how to prove the first NPC problem?

  - Circuit-satisfiability problem.
- Cook-Levin Theorem
  - The satisfiability problem in *NP-complete*

First NP-complete problem Circuit Satisfiability

- Boolean combinational circuit
  - Boolean combinational elements, wired together
  - Each element, inputs and outputs (binary)
  - Limit the number of outputs to 1
  - Called *logic gates*: NOT gate, AND gate, OR gate
  - *truth table*: giving the outputs for each setting of inputs
  - true assignment: a set of boolean inputs.
  - satisfying assignment: a true assignment causing the output to be 1.
  - A circuit is satisfiable if it has a satisfying assignment

Solving circuit-satisfiability problem

- Solution
  - For each possible assignment, check whether it generates 1.
- suppose the number of inputs is k
- total possible assignments are 2^k
- the running time is $\Omega(2^k)$
- Thus, the running time is not polynomial

Circuit-satisfiability problem is in NP

- Now we shall formally prove that circuit-satisfiability is NP-complete
- First let us prove that circuit-satisfiability belongs to NP
- i.e. there exists a non-deterministic algorithm A which is verifiable in polynomial time
  - Given (an encoding of) a boolean combinational circuit C and a certificate, which is an assignment of boolean values to (all) wires in C
  - The algorithm is constructed as follows:

  1. "Guess" the values of input nodes as well as the output value of each logic gate – a certificate
  2. Visit each logic gate *g* in *C* and check that the "guessed" value for the output of *g* is the correct value for *g*'s boolean function based on the given values for the inputs for *g*
  3. If any check for a gate fails, or if the guessed value for output is 0, then the algorithm outputs "no"
  4. If the check for every gate succeeds and the output is 1, the algorithm outputs "yes"

Circuit-satisfiability problem is in NP

- Step 2 is always performed in polynomial time
- Whenever a satisfiable circuit is input to the algorithm A, there is a certificate whose length is polynomial in the size of C and that causes A to output 1
- Whenever an unsatisfiable circuit is input, A will produce output 0 for every certificate
- The algorithm is executed in polynomial time
- Therefore, circuit satisfiability is in NP

Circuit-satisfiability problem is NP-hard

- To prove circuit satisfiability is NP-hard:
- Suppose X is *any problem* in NP
  - We shall construct a polynomial time algorithm F that maps every problem instance x in X to a circuit $C=f(x)$ such that the answer to x is YES if and only if C is satisfiable
  - Since $X \in NP$, there exists a polynomial time algorithm A which verifies X
  - Let $T(n)$ denote the worst-case running time of algorithm A on input strings of length n
  - Let k be the constant such that $T(n)=O(n^k)$ and the length of the certificate is $O(n^k)$

Circuit-satisfiability problem is NP-hard

- Any deterministic algorithm can be implemented on a simple computational model consisting of a CPU and a bank *M* of addressable memory cells
- A can be represented as a sequence of configurations, c_0, $c_1,\ldots,c_i,c_{i+1},\ldots,c_{T(n)}$, each c_i can be broken into
  - (program for A, program counter PC, auxiliary machine state, input *x*, certificate *y*, working storage) and
  - c_i is mapped to c_{i+1} by the combinational circuit M implementing the computer hardware.
  - The output of *A:* 0 or 1– is written to some designated location in working storage when A finishes executing
  - If the algorithm runs for at most $T(n)$ steps, the output appears as one bit in $c_{T(n)}$
  - Note: $A(x,y)$=1 or 0

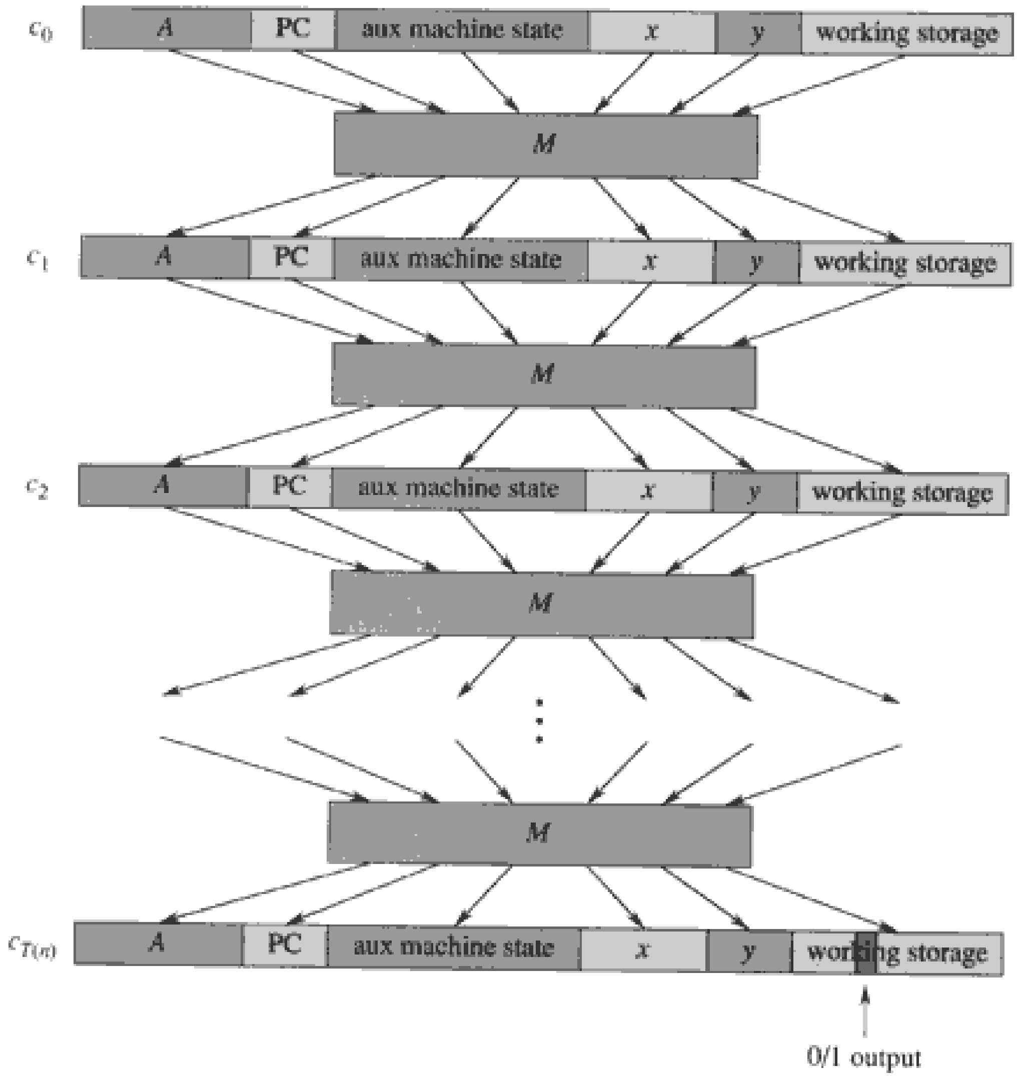


Figure 34.9 The sequence of configurations produced by an algorithm A running on an input x and certificate y. Each configuration represents the state of the computer for one step of the computation and, besides A, x, and y, includes the program counter (PC), auxiliary machine state, and working storage.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

Circuit-satisfiability problem is NP-hard

- The reduction algorithm F constructs a single combinational circuit C as follows:
  - Paste together all $T(n)$ copies of the circuit M.
  - The output of the ith circuit, which produces c_i, is directly fed into the input of the $(i+1)$th circuit.
  - All items in the initial configuration, except the bits corresponding to certificate y, are wired directly to their known values.
  - The bits corresponding to y are the inputs to C.
  - All the outputs to the circuit are ignored, except the one bit of $c_{T(n)}$ corresponding to the output of A.

Circuit-satisfiability problem is NP-hard

- Two properties remain to be proven:
  - F correctly constructs the reduction, i.e., C is satisfiable if and only if there exists a certificate *y*, such that *A*(*x*,*y*)=1.
    - Suppose there is a certificate *y*, such that *A*(*x*,*y*)=1. Then if we apply the bits of *y* to the inputs of C, the output of C is the bit of *A*(*x*,*y*), that is C(*y*)= *A*(*x*,*y*) =1, so C is satisfiable.
    - Suppose C is satisfiable, then there is a *y* such that C(*y*)=1. So, A(*x*,*y*)=1.
  - F runs in polynomial time.

Circuit-satisfiability problem is NP-hard

- F runs in polynomial time.
  - Polynomial space:
    - Size of x is n.
    - Size of A is constant, independent of x.
    - Size of y is $O(n^k)$.
    - Amount of working storage is polynomial in n since A runs at most $O(n^k)$.
    - M has size polynomial in length of configuration, which is polynomial in $O(n^k)$, and hence is polynomial in n.
    - C consists of at most $O(n^k)$ copies of M, and hence is polynomial in n.
    - Thus, the C has polynomial space.
  - The construction of C takes at most $O(n^k)$ steps and each step takes polynomial time, so F takes polynomial time to construct C from x.

NP-complete Problems

- How do we prove a problem to be NP-complete?
  - Given a new problem *L*, we first prove that *L* is in NP
  - Then we reduce a known NP-complete problem to *L* in polynomial time showing *L* to be NP-hard
  - In doing so, we use the following lemma:
    - If $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$, then $L_1 \leq_p L_3$
  - The reductions generally take one of three forms:
    - **Restrictions:** Show that a problem *L* is NP-hard by noting that a known NPC problem *M* is actually just a special case of *L.*
    - **Local Replacement:** Reduce a known NPC problem *M* to *L* by dividing instances of *M* and *L* into "basic units" and then showing how each basic unit of *M* can be locally converted into a basic unit of *L*.
    - **Component design:** Reduce a known NPC problem *M* to *L* by building components for an instance of *L* that will enforce important structural functions for instances of *M*.
      - **Most difficult to construct. Used in the Cook-Levin theorem**

CNF-SAT is NP-complete

- Takes a boolean formula in conjunctive normal form (CNF) as input and asks if there is an assignment of boolean values to its variables so that the formula evaluates to 1.
  - CNF is formed as a collection of subexpressions, called clauses, that are combined using AND, with each clause formed as the OR of boolean variables or their negation, called literals.
- *CNF-SAT is NP*
  - for a given boolean formula *S,* we can construct a nondeterministic algorithm that first “guesses” as assignment of boolean values for the variables in *S* and then evaluates each clause of *S* in turn. If all the clauses evaluate to 1, then *S* is satisfied; otherwise it is not.
- *CNF-SAT is NP-hard*

CNF-SAT is NP-complete

- *CNF-SAT is NP-hard*
- *We shall reduce the circuit satisfiability (CIRCUIT-SAT) problem to CNF-SAT*
  - Assume that each AND and OR gate has two inputs and each NOT gate has one input
  - Create a variable x_i for each input for the entire circuit *C*
  - (Don't start constructing a formula just using these x_is, because this won't give you polynomial time reduction)
  - Create a variable y_i for each output of a gate in *C*
  - Create a formula B_g that corresponds to each gate *g*

CNF-SAT is NP-complete

- If *g* is an AND gate with inputs *a* and *b* (could be either x_is or y_is) and output *c*, then $B_g = (c \leftrightarrow (a.b))$
- If *g* is an OR gate with inputs *a* and *b* and output *c*, then $B_g = (c \leftrightarrow (a+b))$
- If *g* is a NOT gate with input *a* and output *b*, then $B_g = (b \leftrightarrow \bar{a})$

- Convert each B_g to be in CNF
- Combine all transformed B_gs by AND operations to define the CNF formula *S'* that corresponds exactly to each input and logic gate in the circuit *C*
- Final boolean formula *S* is then given by *S = S` . y*, where y is the variable that is associated with the output of the gate that defines the value of *C* itself
- Thus, *C* is satisfiable if and only if *S* is satisfiable
- Construction from *C* to *S* builds a constant-sized subexpression for each input and gate of *C*
- This construction runs in polynomial time

- *Thus CNF-SAT is NP-complete*

3-CNF Satisfiability Problem

- A boolean formula which is in CNF form is in 3-CNF form if each clause has exactly three distinct literals
- In 3-CNF satisfilability (or 3-SAT), we are asked whether a given boolean formula *S* in 3-CNF form is satisfiable.
- We shall prove that 3-SAT is NP-complete
- (2-SAT problem, where every clause has exactly two literals, can be solved in polynomial time)
- 3-SAT is in NP, because we can construct a nondeterministic algorithm that takes a CNF formula *S* with 3-literals per clause, guesses an assignment of boolean values for *S*, and then evaluates *S* to see if it is equal to 1

3-SAT is NP-complete

- Now to show that 3-SAT is NP-hard
  - Reduce CNF-SAT to 3-SAT in polynomial time
  - Let *C* be a boolean formula in CNF
  - Perform local replacements for each clauses C_i in *C* to generate a formula S_i
  - Values assigned to the newly introduced variables are irrelevant. No matter what we assign to them, the clause C_i is 1 if and only if S_i is 1
  - Thus, *C* is 1 if and only if *S* is 1
  - Moreover, each clause increases in size by at most a constant factor and that the computations involved are simple substitutions
- Thus, 3-SAT is NP-complete