Design and Analysis of Algorithms

Theory of NP-Completeness & Undecidability

Dr. Soharab Hossain Shaikh
BML Munjal University

1

Nondeterministic Algorithms

- A nondeterminstic algorithm consists of
  phase 1: guessing (non-deterministic step)
  phase 2: checking (deterministic step)
- If the checking stage of a nondeterministic algorithm is of polynomial time-complexity, then this algorithm is called an NP (nondeterministic polynomial) algorithm.
- NP problems : (must be decision problems)
  - e.g. Satisfiability Problem (SAT)
    3-SAT, Clique, Vertex Cover
    Traveling Salesperson Problem (TSP)

2

Nondeterministic Operations & Functions

- Choice(S) : arbitrarily chooses one of the elements in set S
- Failure : an unsuccessful completion
- Success : a successful completion
- Nonderministic searching algorithm:

```
 j ← choice(1 : n)   /* guessing */
if A(j) = x then success  /* checking */
         else failure
```

3

Non-deterministic Search

Nondeterministic-Search(A, n, key)

```
{
    j=choice();    // non-deterministic guess
    if(key==A[j])  // deterministic verification
    {
      Write(j);
      Success();
    }
    Write(0);
    Failure();
}
```

3- 4

4

Nondeterministic Operations & Functions

- A nondeterministic algorithm terminates unsuccessfully if and only if there exists no set of choices leading to a success signal.
- The time required for *choice(1 : n)* is O(1).
- A deterministic interpretation of a non-deterministic algorithm can be made by allowing unbounded parallelism in computation.

5

Decision Problems

- The solution is simply "Yes" or "No".
- Optimization problems are more difficult.
- e.g. the traveling salesperson problem
  - Optimization version:
    Find the shortest tour
  - Decision version:
    Is there a tour whose total length is less than or equal to a constant *c* ?

6

Solving Optimization Problem using Decision Algorithm

- Solving TSP optimization problem by decision algorithm :
  - Give c_1 and test (decision algorithm)
    Give c_2 and test (decision algorithm)
    ⋮
    Give c_n and test (decision algorithm)
  - We can easily find the smallest c_i

7

General Definitions

Problems
- Decision problems (yes/no)
- Optimization problems (solution with best score)

P - Decision problems that can be solved by deterministic algorithm in polynomial time
- can be solved "efficiently"

NP - Decision problems which can be solved by nondeterministic algorithm in polynomial time. Whose "*YES*" answer can be verified in polynomial time. (SAT, 3-Colorable, Hamiltonian-Cycle)

co-NP - Decision problems whose "*NO*" answer can be verified in polynomial time. Set of complement of some known NP problems (UnSAT, No-3-Colorable, No-Hamiltonian-Cycle).

8

Different types of Problems

P : Tractable decision problems: a problem that is solvable by a polynomial time algorithm. The upper bound of running time is polynomial. (e.g. Sorting, Searching, MST, Shortest path). These problems have efficient solutions (worst case running time is polynomial). [**Polynomial time solvable by a deterministic Turing machine**]

NP: Decision problems that cannot be solved by a polynomial time algorithm (not tractable in deterministic machines for a large instance of the problem). The lower bound to running time is exponential. (e.g. TSP, Graph coloring, SAT etc.) [**Polynomial time solvable by a non-deterministic Turing machine. In other words, not polynomial time solvable but polynomial time verifiable by a deterministic Turing machine**]

Unsolvable: For some problems, there is no known algorithms (e.g. Halting problem of Turing machine). [**No Turing machine can be constructed for solving such problems**]

9

General Definitions

e.g. The **satisfiability problem (SAT)**

- Given a Boolean formula
- Is it possible to assign the input x1...x9, so that the formula evaluates to TRUE?
- If the answer is YES with a proof (i.e. an assignment of input value), then we can check the proof in polynomial time (SAT is in NP)

10

Satisfiability Problem

- In computer science, the Boolean satisfiability problem (sometimes called propositional satisfiability problem and abbreviated as SATISFIABILITY or SAT) is the problem of determining if there exists an interpretation that satisfies a given Boolean formula.
- In other words, it asks whether the variables of a given Boolean formula can be consistently replaced by the values TRUE or FALSE in such a way that the formula evaluates to TRUE.
- If this is the case, the formula is called satisfiable. On the other hand, if no such assignment exists, the function expressed by the formula is FALSE for all possible variable assignments and the formula is unsatisfiable.
- For example, the formula "a AND NOT b" is satisfiable because one can find the values a = TRUE and b = FALSE, which make (a AND NOT b) = TRUE. In contrast, "a AND NOT a" is unsatisfiable.

11

Satisfiability Problem

- A formula is in conjunctive normal form (CNF) if it is a conjunction(also denoted by ∧) of clauses (or a single clause)
- For example, x1 is a positive literal, ¬x2 is a negative literal, then x1 ∨ ¬x2 is a clause, and (x1 ∨ ¬x2) ∧ (¬x1 ∨ x2 ∨ x3) ∧ ¬x1 is a formula in conjunctive normal form
- The formula is satisfiable, choosing x1 = FALSE, x2 = FALSE, and x3 arbitrarily,
- (FALSE ∨ ¬FALSE) ∧ (¬FALSE ∨ FALSE ∨ x3) ∧ ¬FALSE evaluates to
- (FALSE ∨ TRUE) ∧ (TRUE ∨ FALSE ∨ x3) ∧ TRUE, and in turn to
- TRUE ∧ TRUE ∧ TRUE (i.e. to TRUE).
- In contrast, the CNF formula a ∧ ¬a, consisting of two clauses of one literal, is unsatisfiable, since for a=TRUE and a=FALSE it evaluates to TRUE ∧ ¬TRUE (i.e. to FALSE) and FALSE ∧ ¬FALSE (i.e. again to FALSE), respectively.

12

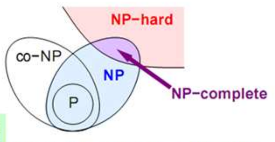
General Definitions

- NP-hard
  - A problem is NP-hard iff an polynomial-time algorithm for it implies a polynomial-time algorithm for every problem in NP
  - NP-hard problems are at least *as hard as* NP problems
- NP-complete
  - A problem is NP-complete if it is NP-hard, and is an element of NP (NP-easy)

Cook's Theorem
SAT is NP-hard

we knew SAT is in NP ⇩

SAT is NP-complete



More of what we *think* the world looks like.

13

Some Concepts of NPC

- Definition of reduction: Problem A reduces to problem B ($A \propto B$) iff A can be solved by a deterministic polynomial time algorithm using a deterministic algorithm that solves B in polynomial time.
- Up to now, none of the NPC problems can be solved by a deterministic polynomial time algorithm in the worst case.
- It does not seem to have any polynomial time algorithm to solve the NPC problems.

14

Some Concepts of NPC

- The theory of NP-completeness always considers the worst case.
- The lower bound of any NPC problem seems to be in the order of an exponential function.
- If $A, B \in$ NPC, then $A \propto B$ and $B \propto A$.
- Theory of NP-completeness

  If any NPC problem can be solved in polynomial time, then all NP problems can be solved in polynomial time. This implies NP = P

15

Problem Reductions

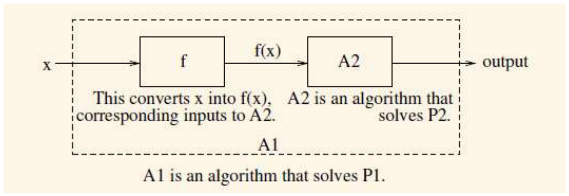
- Suppose you have a problem P2 which you know how to solve, e.g. by using algorithm A2.
- Suppose you have been given another problem P1 that seems similar to P2.
- How might you solve P1?
  - - You could try to solve P1 from scratch.
  - - You could try to borrow elements of A2.
  - - You could try to find a reduction from P1 to P2.

Reduction of P1 to P2:
- transforms inputs to P1 into inputs to P2
– run A2 (which solves P2) as a 'black-box'
– interpret the outputs from A2 as answers to P1

16

Problem Reductions

- A problem P1 is ***reducible*** to a problem P2 if there is a function f that takes any input x to P1 and transforms it to an input f(x) of P2, such that the solution to P2 on f(x) is the solution to P1 on x.


x
f
f(x)
A2
output
This converts x into f(x), corresponding inputs to A2.
A2 is an algorithm that solves P2.
A1
A1 is an algorithm that solves P1.

17

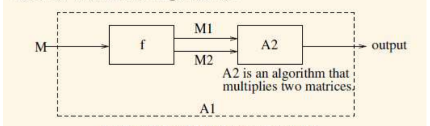
Problem Reductions

- Problem1: Squaring a Matrix
- Parameters: A matrix, M.
- Returns: The result of squaring M.

- Problem2: Matrix Multiplication
- Parameters: Two matrices, M1 and M2.
- Returns: The result of multiplying M1 and M2 together

- **Design a new algorithm A1 that solves Problem1 using algorithm A2 that solves Problem2**

Here's how we would do it using a reduction:

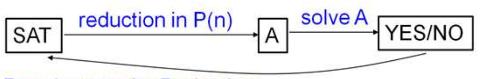


A1 is an algorithm that squares a matrix.

18

Polynomial Time Reduction

- How to know another problem, A, is NP-complete?
  - To prove that A is NP-complete, reduce a known NP-complete problem to A



- Requirement for Reduction
  - Polynomial time
  - YES to A also implies YES to SAT, while NO to A also implies No to SAT (Note that A must also have short proof for YES answer)

19

Polynomial Time Reduction

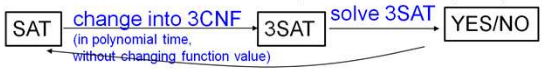
- An example of reduction

3-CNF: A Boolean formula in Conjunctive Normal Form having exactly 3 literals per clause.

$$\overbrace{(a\vee \underset{\text{literal}}{b}\vee c)}^{\text{clause}}\wedge(b\vee \bar{c}\vee d)\wedge(\bar{a}\vee c\vee \bar{d})$$

3SAT: is a Boolean formula in *3-CNF* has a feasible assignment of inputs so that it evaluates to TRUE?

- reduction from SAT to 3SAT (3SAT is NP-complete)



20

Cook-Levin Theorem

Cook–Levin theorem, also known as **Cook's theorem**, states that the Boolean satisfiability problem is NP-complete. That is, any problem in NP can be reduced in polynomial time by a deterministic Turing machine to the problem of determining whether a Boolean formula is satisfiable.

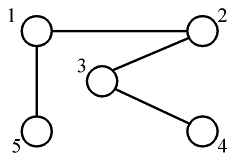
NP = P iff the satisfiability problem is a P problem.

- SAT is NP-complete.
- It is the first NP-complete problem.
- Every NP problem reduces to SAT.

21

Example of NPC Problem: Vertex Cover

- **Def**: Given a graph G=(V, E), S is the Vertex/Node Cover if $S \subseteq V$ and for every edge $(u, v) \in E$, either $u \in S$ or $v \in S$ or both.



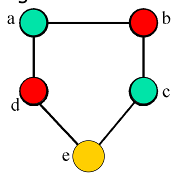
node cover :
{1, 3}
{5, 2, 4}

- Decision problem : $\exists$ S $|S| \le K$?

22

Example of NPC Problem: Chromatic Number (CN)

- **Def**: A coloring of a graph G=(V, E) is a function $f : V \rightarrow \{1, 2, 3, \ldots, k\}$ such that if $(u, v) \in E$, then $f(u) \neq f(v)$. The CN problem is to determine if G has a coloring for k.
- E.g.



3-colorable
f(a)=1, f(b)=2, f(c)=1
f(d)=2, f(e)=3

Theorem: Satisfiability with at most 3 literals per clause (3SAT) $\propto$ CN.

23

Example of NPC Problem: Sum of Subset

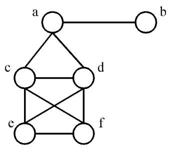
- **Def**: A set of positive numbers $A = \{ a_1, a_2, \ldots, a_n \}$
  a constant C
  Determine if $\exists A' \subseteq A$ $\sum_{a_i \in A'} a_i = C$
- e.g. A = { 7, 5, 19, 1, 12, 8, 14 }
  - C = 21, A′ = { 7, 14 }
  - C = 11, no solution

<Theorem> Exact Cover $\propto$ Sum of Subsets.

24

Example of NPC Problem: Max Clique

- **Def:** A complete subgraph of a graph G=(V,E) is a clique. The max (maximum) clique problem is to determine the size of a largest clique in G.
- e. g.



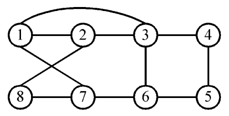
Different cliques :
{a, b}, {a, c, d}
{c, d, e, f}
Maximum clique :
(largest)
{c, d, e, f}

<Theorem> SAT ∝ Clique decision problem.

25

Example of NPC Problem: Hamiltonian Cycle

- **Def:** A Hamiltonian cycle is a round trip path along n edges of G which visits every vertex once and returns to its starting vertex.
- e.g.



Hamiltonian cycle : 1, 2, 8, 7, 6, 5, 4, 3, 1.

<Theorem> SAT ∝ Hamiltonian cycle (in a directed graph)

26

Traveling Salesperson Problem

- **Def:** A tour of a directed graph G=(V, E) is a directed cycle that includes every vertex in V. The problem is to find a tour of minimum cost.

<Theorem> Directed Hamiltonian cycle ∝ Traveling Salesperson decision problem.

27

Toward NP-Completeness

- Cook's Theorem

  The SAT problem is NP-complete.
- Once we have found an NP-complete problem, proving that other problems are also NP-complete becomes easier.
- Given a new problem *Y*, it is sufficient to prove that Cook's problem, or any other NP-complete problem, is ***polynomially reducible*** to *Y*.
- Known problem -> unknown problem

28

NP-Completeness Proof

The following problems are NP-complete:

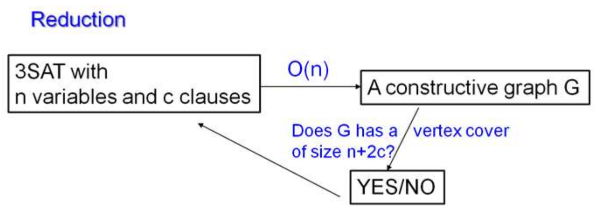
Vertex Cover(VC) and Clique.

Definition:

- A vertex cover of G=(V, E) is V′⊆V such that every edge in E is incident to some v∈V′.
- Vertex Cover(VC): Given undirected G=(V, E) and integer *k*, does G have a vertex cover with ≤*k* vertices?
- CLIQUE: A complete subgraph of a graph G=(V,E) is a clique.
- Does G contain a clique of size ≥*k*?

29

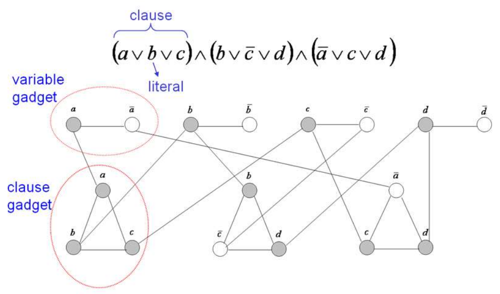
Examples of NPC Problems: Vertex Cover


Reduction
3SAT with n variables and c clauses
O(n)
A constructive graph G
Does G has a vertex cover of size n+2c?
YES/NO

30

3-SAT to Vertex Cover

Vertex cover - an example of the constructive graph

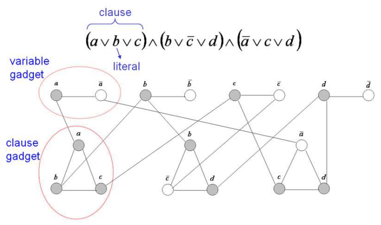

clause
$(a \vee b \vee c) \wedge (b \vee \bar{c} \vee d) \wedge (\bar{a} \vee c \vee d)$
literal
variable gadget
clause gadget
a
ā
b
b̄
c
c̄
d
d̄

31

3-SAT to Vertex Cover

- Vertex cover
  - we must prove:
    the graph has a n+2c vertex cover, if and only if the 3SAT is satisfiable (to make the two problem has the same YES/NO answer)

Vertex cover - an example of the constructive graph


clause
$(a \vee b \vee c) \wedge (b \vee \bar{c} \vee d) \wedge (\bar{a} \vee c \vee d)$
literal
variable gadget
clause gadget

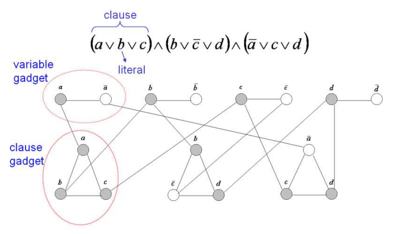
32

3-SAT to Vertex Cover

- Vertex cover
  - if the graph has a n+2c vertex cover
    1) there must be 1 vertex per variable gadget, and 2 per clause gadget
    2) in each clause gadget, set the remaining one literal to be true

Vertex cover - an example of the constructive graph

$$\underbrace{(a\vee b\vee c)}_{\text{clause}}\wedge(b\vee\bar{c}\vee d)\wedge(\bar{a}\vee c\vee d)$$



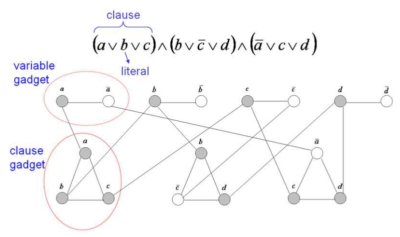
33

3-SAT to Vertex Cover

- Vertex cover
  - if the 3SAT is satisfiable
    1) choose the TURE literal in each variable gadget
    2) choose the remaining two literal in each clause gadget

Vertex cover - an example of the constructive graph

$$\underbrace{(a\vee b\vee c)}_{\text{clause}}\wedge(b\vee\bar{c}\vee d)\wedge(\bar{a}\vee c\vee d)$$



34

Proof: Clique is NP-Complete

Proof: To show CLIQUE is NP-complete.

Two things to be done:

- Show CLIQUE is in NP.
- Show that every language in NP is polynomial time reducible to CLIQUE.

Sufficient to give polynomial time reduction from some NP-complete language to CLIQUE.

35

Proof: Clique is NP-Complete

Let us try to reduce SAT to CLIQUE:

Let F be a CNF-formula.
Let $C_1, C_2, \ldots, C_k$ be the clauses in F.
Let $x_{j,1}$, $x_{j,2}$, $x_{j,m}$ be the literals of C_j.

Hint: Construct a graph G such that
F is satisfiable ⇔ G has a k-clique

36

Proof: Clique is NP-Complete

We construct a graph G as follows:

1. For each literal $x_{j,q}$, we create a distinct vertex in G representing it.

2. G contains all edges, **except** those

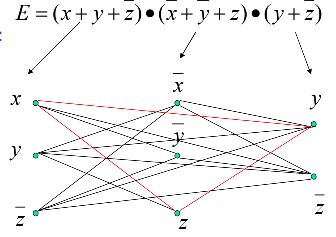
(i) joining two vertices in same clause.
(ii) joining two vertices whose literals is the negation of the others.

37

SAT to Clique

- Example: $E = (x + y + \overline{z}) \bullet (\overline{x} + \overline{y} + z) \bullet (y + \overline{z})$

G =



- Make "column" for each of k clauses.
  - No edge within a column.
  - All other edges present except between x and x'.

- G has *k*-clique (*k* is the number of clauses in E), **iff** E is satisfiable. (Assign value 1 to all literals in clique)

38

Proof: Clique is NP-Complete

We now show that

G has a k-clique ⇔ F is satisfiable

(=>) If G has a k-clique,

1. The k-clique must have a vertex from each clause.

2. Also, no vertex will be the negation of the others in the clique.

Thus, by setting the corresponding literal to TRUE, F will be satisfied.

39

Proof: Clique is NP-Complete

(<=) If F is satisfiable, at least a literal in each clause is set to TRUE in the satisfying assignment.
So, the corresponding vertices forms a clique
Thus, G has a k-clique.

Finally, since G can be constructed from F in polynomial time, so we have a polynomial time reduction from 3SAT to CLIQUE.

Thus, CLIQUE is NP-complete

40

Undecidable Problem

There are some problems for which no algorithm (or Turing machine) exists.

e.g. The **Halting Problem** of Turing Machine.

Can we design a Turing machine $\mathbf{M_U}$ that takes an arbitrary Turing machine ***M*** and an input ***w*** to it, as its input and will be able to decide whether ***M*** will halt on input ***w***?

This problem is not decidable. No such Turing machine $\mathbf{M_U}$ exists.

41

Halting Problem

> **For every computer program a Turing machine can be constructed that performs the task of the program**.

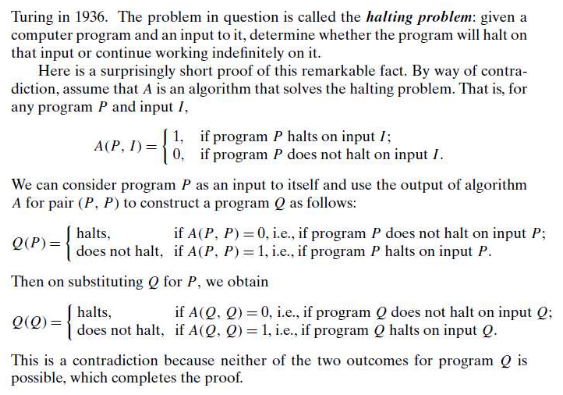
> Thus, the question of whether a program halts for a given input is nothing but the **halting problem**.

> Thus, one implication of the halting problem is that there can be NO computer program/algorithm (Turing machine) that checks whether any **arbitrary computer program stops for a given input**.

The problem can also be stated as follows:
We can never write an algorithm A (or program P) that will take another arbitrary algorithm A1 (or program P1) and an input I to A1(or P1) as input, and can decide whether A1 (or P1) will halt on input I.

42

Proof: Halting Problem is Undecidable



Turing in 1936. The problem in question is called the ***halting problem***: given a computer program and an input to it, determine whether the program will halt on that input or continue working indefinitely on it.

Here is a surprisingly short proof of this remarkable fact. By way of contradiction, assume that A is an algorithm that solves the halting problem. That is, for any program P and input I,

$$A(P, I) = \begin{cases} 1, & \text{if program } P \text{ halts on input } I; \\ 0, & \text{if program } P \text{ does not halt on input } I. \end{cases}$$

We can consider program P as an input to itself and use the output of algorithm A for pair (P, P) to construct a program Q as follows:

$$Q(P) = \begin{cases} \text{halts,} & \text{if } A(P, P) = 0, \text{ i.e., if program } P \text{ does not halt on input } P; \\ \text{does not halt,} & \text{if } A(P, P) = 1, \text{ i.e., if program } P \text{ halts on input } P. \end{cases}$$

Then on substituting Q for P, we obtain

$$Q(Q) = \begin{cases} \text{halts,} & \text{if } A(Q, Q) = 0, \text{ i.e., if program } Q \text{ does not halt on input } Q; \\ \text{does not halt,} & \text{if } A(Q, Q) = 1, \text{ i.e., if program } Q \text{ halts on input } Q. \end{cases}$$

This is a contradiction because neither of the two outcomes for program Q is possible, which completes the proof.

43

End of Lecture

44