Randomized Algorithms

April 26, 2021

Radomized Algorithms

Definition

An algorithm is *randomized* if its behavior is determined not only by its inputs but also by values produced by a *random-number generator*

Randomized Algorithms

Benefits

- Alleviate the need to make assumptions about input distributions
  - e.g. all permutations of input are equally likely
- Protects from malicious adversarial attacks
  - e.g. who generate the input data in a form which takes too long to execute

Types of Randomized Algorithms

- Las Vegas Algorithms
  - Different running times on same input (but different inputs from random number generator)
  - Output is same (and correct) for same input (and different inputs from random number generator)
- Monte-Carlo Algorithms
  - Same running time on same input (but different inputs from random number generator)
  - Different output (sometimes correct and sometimes incorrect) for same input (and different inputs from random number generator)

Hiring Problem

Hire Assistant Problem

Problem Specification

- You want to hire an office assistant, and make use of employment agency
- Everyday you conduct single interview (involves cost)
  - if candidate is better than current assistant, hire him and replace current assistant
  - involves cost as employment agency should be paid
- Hiring cost much higher than interviewing cost
  - in both cases payment to employment agency

Hiring Problem

Algorithm

```
HIRE-ASSISTANT(n)
1  best ← 0          ▷ candidate 0 is a least-qualified dummy candidate
2  for i ← 1 to n
3        do interview candidate i
4           if candidate i is better than candidate best
5              then best ← i
6                   hire candidate i
```

Figure: Hire Assistant[1]

[1] Cormen et. al "Introduction to Algorithms"

Hiring Problem

Computation of cost

Let:

n be number of candidates interviewed
m be number of candidates hired m
c_i be cost of interview
c_h be cost of hiring

Then, cost is $nc_i + mc_h$
Worst-case cost nc_h occurs when every next candidate interviewed is hired (c_i may be ignored in comparison to c_h)

Hiring Problem

Average Case Analysis

- Each input is a sequence of ranks, with bigger rank indicating better candidate
- Average case analysis is a probabilistic analysis of average running times using probability distribution of inputs
- For this problem, it is assumed that all permutations of possible rank sequences are equally likely
  - i.e. candidates arrive in random order
- Above assumption may not hold in realistic scenario

Hiring Problem

Average Case Analysis

Let X be a random variable representing the number of times we hire a new office assistant. We want $E[X]$.

Let $X_i = I\{\text{candidate i is hired}\}$ where I is an indicator function. Then

$$X = X_1 + X_2 + \cdots + X_n \tag{1}$$

Hiring Problem

Average Case Analysis

To calculate $E[X]$ directly is not straightforward:

$$E[X] = \sum_{x=1}^{n} x \Pr\{X = x\}$$

Instead, we use linearity of expectation

$$\begin{aligned}
E[X] &= E[\sum_{i=1}^{n} X_i] && \text{(from Eq. 1)} && (2)\\
&= \sum_{i=1}^{n} E[X_i] && \text{(by linearity of expectation)} && (3)\\
&= \sum_{i=1}^{n} 1/i && && (4)\\
&= \ln n + O(1) && && (5)
\end{aligned}$$

Hiring Problem

Average Case Analysis - Continued

It may be noted that

$$E[X_i] = 1 \cdot \Pr[X_i = 1] + 0 \cdot \Pr[X_i = 0] \quad (6)$$
$$= \Pr[X_i = 1] \quad (7)$$
$$= 1/i \quad (8)$$

Where Eq. 8 happens because candidates arrive in random order, and so, among i candidates, each have equal probability of being the best candidate, including the i^{th} candidate.

Also, Eq. 5 arises because of the formula of the n^{th} harmonic number

Hiring Problem

Average Case Analysis - Continued

$\Pr[X_i = 1] = 1/i$

Consider a sequence of ranks 4, 2, 3, 8. The last rank is highest, so the candidate at position 4 is hired corresponding to $X_i = 1$. This highest ranked candidate could very well have been the first , second, or third candidate under the assumption of random permutations of ranks. So, $\Pr[X_4 = 1] = 1/4$

Hiring Problem

Analysis of Cost

- Average-case cost of hiring is $O(c_h \ln n)$
- However, average-case analysis assumes all permutations of ranks are equally possible (which may not be true)
- Worst-case cost is $O(c_h n)$
  - When every next candidate interviewed is the best candidate
- If the employing agency is malicious and decides to make more money, it can send candidates in increasing order of rank

Hiring Problem

Applications

- Finding maximum/minimum in a sequence of numbers
  - where, at every point in time there is a current winner

Hiring Problem

Activity

1. Consider a binary random variable X with probability of success (i.e. $X = 1$) p. Compute $E[X]$.
2. How would you compute the expected value of number of successes over n independent trials of the same experiment using $E[X]$ computed above? (Hint: use the concept of indicator functions and linearity of expectation)

Randomized Hiring Problem

Randomized Hiring Problem

Formulation

1. Assumes that complete list of candidates is given to us before hand
2. We then choose a candidate from the list in random order

- Note that cost now depends on the random number
- IMPORTANT: For randomized algorithms, we say *Expected cost* rather than average cost

Randomized Hiring Problem

Expected Cost

- Expected cost is the same - $O(c_h \ln n)$
  - Same discussion applies as for average cost
- Benefit
  - The assumption made by HIRE-ASSISTANT that all permutations are equally possible may not hold at all
    - e.g. because of malicious employment agency
  - Randomized version chooses the next candidate randomly
    - thereby overcoming malicious attempts by employment agency
  - Worst case behaviour is rarely exhibited by randomized version
    - but more frequently by non-randomized version because of malicious data sources

Randomized Quicksort

Randomized Quicksort

Quicksort Example (Non-randomized)

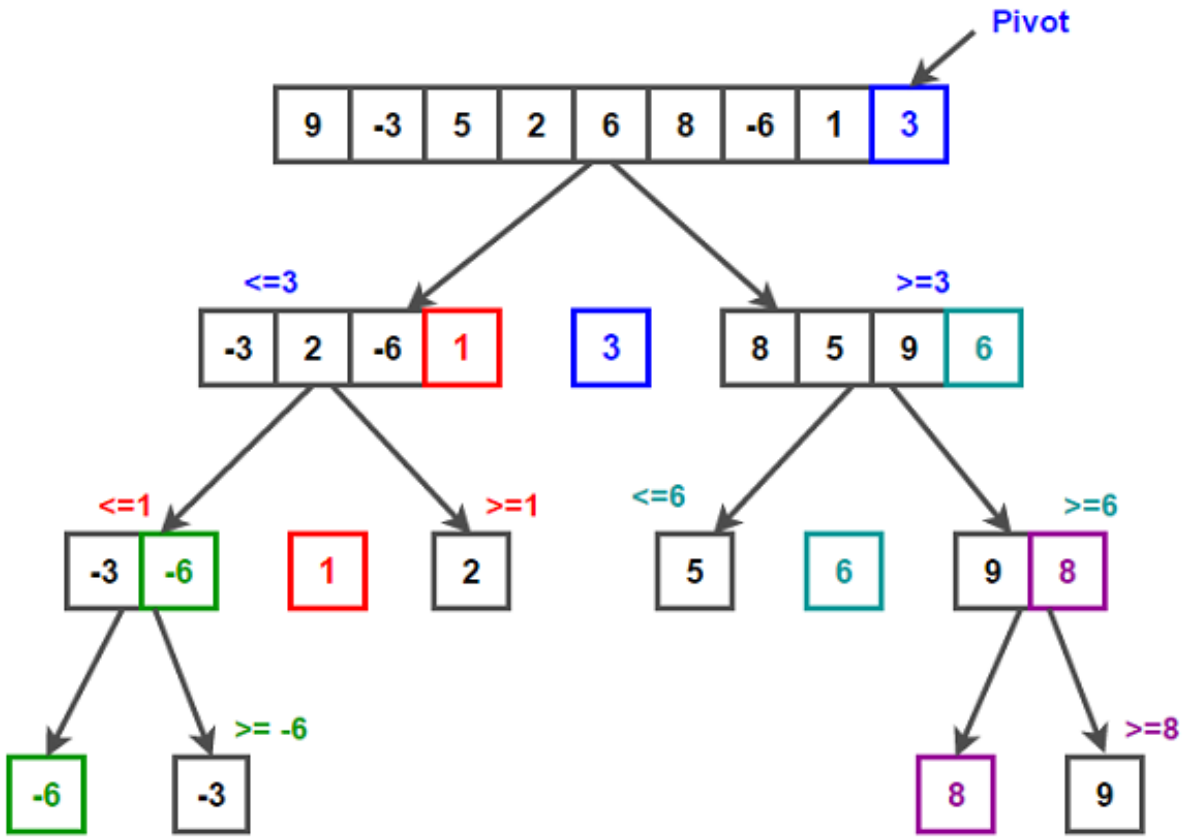


Figure: Quick Sort[1]

[1] Techie Delight

Randomized Quicksort

Quicksort Worst case

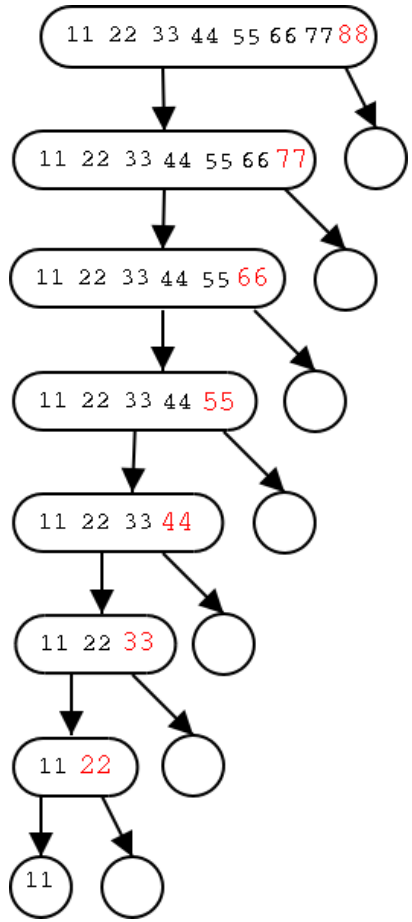


Figure: Quick Sort Worst Case[1]

[1] NYU Computer Science

Randomized Quicksort

Why We Need It

- Worst case running time of Quicksort is $\Theta(n^2)$
- Average case running time is $O(n \log n)$
  - assuming the input is in random order
  - all permutations of ranks are equally likely
  - above assumption is not always true
- Randomized Quicksort can be used to make the performance independent of the input

Randomized Quicksort

Algorithm

```
RANDOMIZED-PARTITION(A, p, r)
1  i = RANDOM(p, r)
2  exchange A[r] with A[i]
3  return PARTITION(A, p, r)
```

```
RANDOMIZED-QUICKSORT(A, p, r)
1  if p < r
2       q = RANDOMIZED-PARTITION(A, p, r)
3       RANDOMIZED-QUICKSORT(A, p, q − 1)
4       RANDOMIZED-QUICKSORT(A, q + 1, r)
```

Figure: Randomized Quicksort[1]

[1] Cormen et al., "Introduction to Algorithms"

Randomized Quicksort

Compared to Quicksort

- Only difference is that the pivot element is chosen randomly
  - rather than first/last element of subarray from p to r

Randomized Quicksort

Some Observations in Quicksort

- Pivot element is compared to every other element in subarray $A[p \ldots r]$ other than itself
  - Pivot is not compared to any other element after that
  - It remains in its place in the array
- Elements in different partitions are never compared with each other
  - only items within a partition are compared
- Every pair of elements is compared atmost once
  - one among them should be the pivot
  - after the pivot is used, it is never used for comparison again

Randomized Quicksort

Expected Running Time

Let us rewrite the array A as $z_1, z_2, \ldots z_n$ where z_i is the i^{th} smallest element in the array.

We define $Z_{ij} = \{z_i, z_{i+1}, \ldots z_j\}$

For z_i to be compared with z_j

- Either should become the pivot
  - elements in subarray are compared only with pivot
- No other element in Z_{ij} becomes a pivot before them
  - if that were the case, z_i and z_j would be placed in different partitions, and never compared at all
- As long as above two conditions hold, z_i is compared with z_j

Randomized Quicksort

Expected Running Time

Define $X_{ij} = I\{z_i \text{ is compared to } z_j\}$
Let X be the total number of comparisons over the entire execution of Quicksort. Then,

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij} \tag{9}$$

Average time $E[X]$ can be computed as

$$E[X] = E[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\,] \tag{10}$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] \text{ by linearity of expectation} \tag{11}$$

Randomized Quicksort

Expected Running Time

Now,

$$E[X_{ij}] = 1 \cdot \Pr\{z_i \text{ is compared to } z_j\} + 0 \cdot \Pr\{z_i \text{ is not compared to} z_j\} \tag{12}$$

$$= \Pr\{z_i \text{ is compared to } z_j\} \tag{13}$$

Randomized Quicksort

Expected Running Time

Substituting Eq. 13 in Eq. 11, we get

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \text{ is compared to } z_j\} \tag{14}$$

Now,

$$\Pr\{z_i \text{ is compared to } z_j\} = \Pr\{z_i \text{ or } z_j \text{ is the first pivot chosen from } Z_{ij}\} \tag{15}$$

$$= \Pr\{z_i \text{ is the first pivot chosen from } Z_{ij}\} + \tag{16}$$

$$\Pr\{z_j \text{ is the first pivot chosen from } Z_{ij}\}$$

$$= \frac{1}{j-i+1} + \frac{1}{j-i+1} = \frac{2}{j-i+1} \tag{17}$$

Randomized Quicksort

Expected Running Time

Substituting Eq. 17 in Eq. 14, we get

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \tag{18}$$

$$= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \qquad \text{putting } k = j-i \tag{19}$$

$$< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \tag{20}$$

$$= \sum_{i=1}^{n-1} O(\log n) \tag{21}$$

$$= O(n \log n) \tag{22}$$

Randomized Quicksort

Expected Running Time

$\Pr\{z_i \text{ is the first pivot chosen from } Z_{ij}\} = \frac{1}{j-i+1}$

As pivots are chosen randomly, there is equal probability that any among $z_i, z_{i+1}, \ldots z_j$ may be become the pivot first. So, $\Pr\{z_i \text{ is the first pivot chosen from } Z_{ij}\} = \frac{1}{j-i+1}$.

Randomized Quicksort

Benefits

- There is no assumption regarding the distribution of input (that all permutations are equally possible)
- Quicksort has the tendency to exhibit worst case cost when candidates are provided in ascending or descending order of ranks, but not the randomized version
- Quicksort is susceptible to malicious attacks but not randomized quicksort

Activity

Look at the quicksort example shown before (tree) and convince yourself that

1. Entries in one partition are never compared with entries in another partition.
2. A pair of entries in the array can be compared atmost once.

Thank You!

String Matching & Rabin-Karp Algorithm for String Matching

Prepared by

Dr.M.Ramprasath

Rabin-Karp Algorithm

String Matching - Introduction

- ***Problem*** **:** Finding all occurrences of a Pattern in a given Text
- ***Solution*** **:** All occurrences of the Pattern
- ***Applications***
  - Text Editing
  - Electronic Surveillance
  - Spam Identification
  - Screen Scraping

Rabin-Karp Algorithm

String Matching - Algorithms

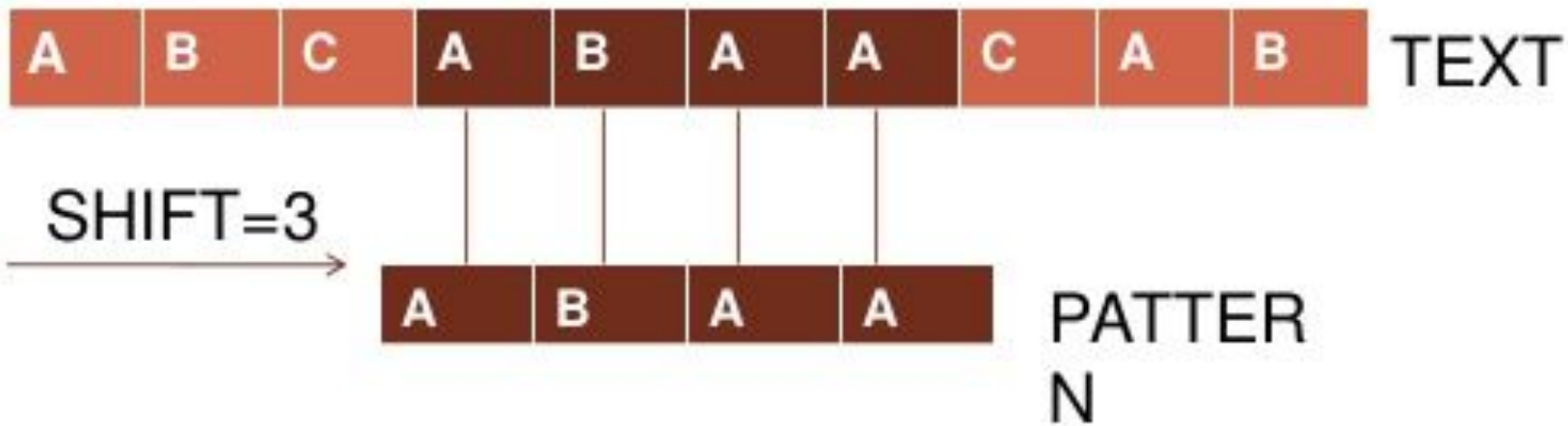
- Simple Text Searching
- Rabin-Karp Algorithm
- Knuth-Morris-Pratt Algorithm
- Boyer-Moore(-Horspool) Algorithm
- Approximate Matching
- Regular Expressions

WHAT IS STRING MATCHING

- In computer science, string searching algorithms, sometimes called string matching algorithms, that try to find a place where one or several string (also called pattern) are found within a larger string or text.

EXAMPLE

STRING MATCHING PROBLEM

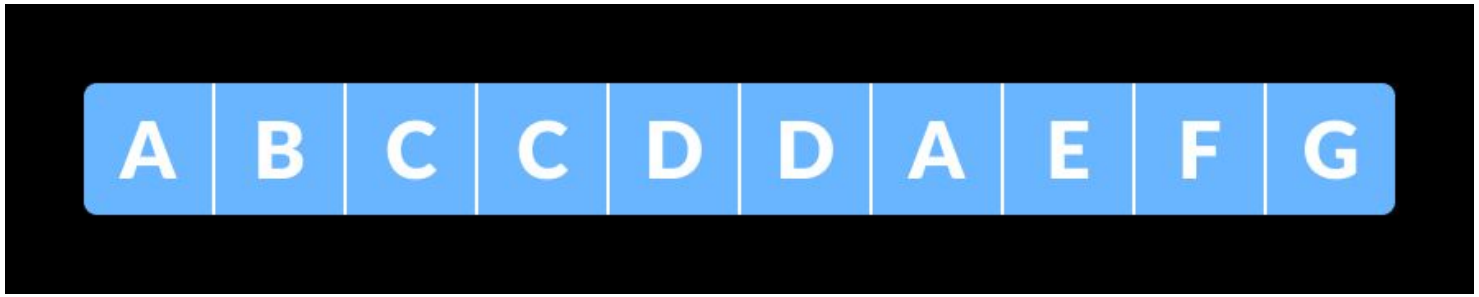


Rabin-Karp Algorithm

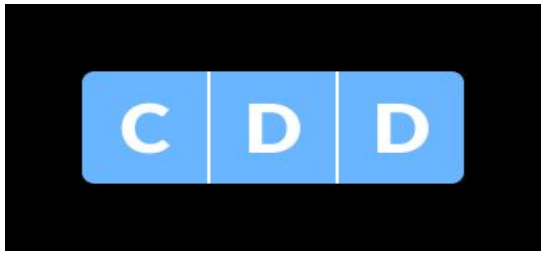
- Rabin-Karp algorithm is an algorithm used for searching/matching patterns in the text using a hash function.
- it does not travel through every character in the initial phase rather it filters the characters that do not match and then performs the comparison.
- A hash function is a tool to map a larger input value to a smaller output value. This output value is called the hash value

Working procedure

- A sequence of characters is taken and checked for the possibility of the presence of the required string. If the possibility is found then, character matching is performed.
- Let the text be:

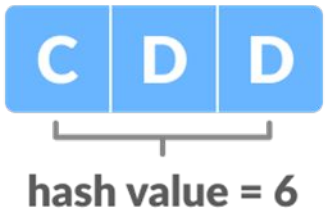


- String to be searched in the above text be:



- Let us assign a numerical value(v)/weight for the characters we will be using in the problem. Here, we have taken first ten alphabets only (i.e. A to J).
- ‘m’ be the length of the pattern and n be the length of the text.
- Here, m = 10 and n = 3.

- Let d be the number of characters in the input set. Here, we have taken input set {A, B, C, ..., J}. So, d = 10. You can assume any suitable value for d.



- hash value for pattern(p) = $\Sigma(v * dm\text{-}1) \bmod 13$

$= ((3 * 10^2) + (4 * 10^1) + (4 * 10^0)) \bmod 13$

$= 344 \bmod 13$

$= 6$

In the calculation above, choose a prime number (here, 13) in such a way that we can perform all the calculations with single-precision arithmetic.

- Calculate the hash value for the text-window of size m.

For the first window ABC,

hash value for text(t) = Σ(v * dn-1) mod 13

$= ((1 * 10^2) + (2 * 10^1) + (3 * 10^0)) \bmod 13$

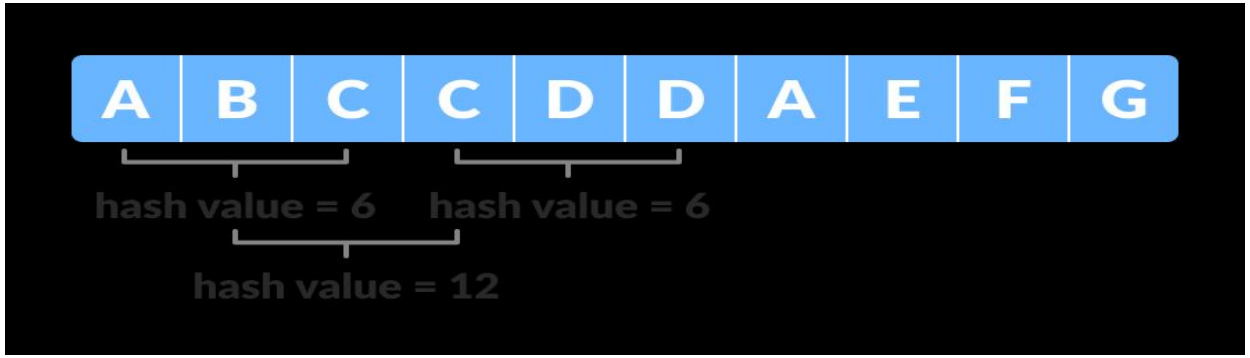
$= 123 \bmod 13$

$= 6$

- Compare the **hash value of the pattern** with the hash value of the text.
- If they match then, **character-matching is performed.**
- In the above examples, the hash value of the first window (i.e. t) matches with p so, go for character matching between ABC and CDD.
- Since they do not match so, go for the next window.
- We calculate the hash value of the next window by subtracting the first term and adding the next term as shown below.

- We calculate the hash value of the next window by subtracting the first term and adding the next term as shown below.
- $t = ((1 * 10^2) + ((2 * 10^1) + (3 * 10^0)) * 10 + (3 * 10^0)) \bmod 13$
- $= 233 \bmod 13$
- $= 12$
- In order to optimize this process, we make use of the previous hash value in the following way.

- t = ((d * (t - v[character to be removed] * h) + v[character to be added]) mod 13
- = ((10 * (6 - 1 * 9) + 3)mod 13
- = 12
- Where, h = dm-1 = 103-1 = 100.
- For BCC, t = 12 (≠6). Therefore, go for the next window.
- After a few searches, we will get the match for the window CDA in the text.



Limitations of Rabin-Karp Algorithm

- Spurious Hit
- When the hash value of the pattern matches with the hash value of a window of the text but the window is not the actual pattern then it is called a spurious hit.
- Spurious hit increases the time complexity of the algorithm. In order to minimize spurious hit, we use modulus. It greatly reduces the spurious hit.

Rabin-Karp Algorithm Complexity

- The average case and best case complexity of Rabin-Karp algorithm is O(m + n) and the worst case complexity is O(mn).

- The worst-case complexity occurs when spurious hits occur a number for all the windows.

Topics

- Introduction To NP Type Problems.
- NP Complete Problem Introduction.
- Hamiltonian Cycle Problem.

INTRODUCTION TO NP TYPE PROBLEMS
NP COMPLETE PROBLEM INTRODUCTION

SRM INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Session Learning Outcome – SLO

- To interpret solutions to evaluate P type, NP Type, and NPC problems.

INTRODUCTION TO NP TYPE PROBLEMS
NP COMPLETE PROBLEM INTRODUCTION

Motivation

It is always useful to know about NP-Completeness for engineers. Suppose you are asked to write an efficient algorithm to solve an extremely important problem for your company. After a lot of thinking, you can only come up exponential time approach which is impractical. If you don't know about NP-Completeness, you can only say that I could not come with an efficient algorithm. If you know about NP-Completeness and prove that the problem as NP-complete, you can proudly say that the polynomial time solution is unlikely to exist. If there is a polynomial time solution possible, then that solution solves a big problem of computer science many scientists have been trying for years.

Types of Problems

- Tractable Problems.
- Intractable Problems.
- Decision Problems.
- Optimization Problems.

Tractable Problems

- Problems that can be solvable in a reasonable (**polynomial**) time.
- Most searching and sorting algorithms.
- For example
    - Ordered search O(lg n), polynomial evaluation O(n), sorting O(n log n).

Intractable Problems

- Some problems are intractable, as they grow large, we are unable to solve them in reasonable (polynomial) time.

Tractability

- What constitutes reasonable time?
- Standard working definition: polynomial time
- On an input of size n the worst-case running time is $O(n^k)$ for some constant k
- $O(n^2)$, $O(n^3)$, $O(1)$, $O(n \lg n)$, $O(2^n)$, $O(n^n)$, $O(n!)$
  - Polynomial time: $O(n^2)$, $O(n^3)$, $O(1)$, $O(n \lg n)$
  - Not in polynomial time: $O(2^n)$, $O(n^n)$, $O(n!)$

Decision Problems

- There are many problems for which the answer is a Yes or a No. These types of problems are known as **decision problems**.
- For example,
  - Whether a given graph can be colored by only 4-colors.
  - Finding Hamiltonian cycle in a graph is not a decision problem, whereas checking a graph is Hamiltonian or not is a decision problem.
- An algorithm for a decision problem is termed as decision algorithm.

Optimization Problems

- Optimization problems are those for which the objective is to maximize or minimize some values.
- For example,
  - Finding the minimum number of colors needed to color a given graph.
  - Finding the shortest path between two vertices in a graph.
- An algorithm which solves an optimization problem is called as optimization algorithm.

Optimization/Decision Problems

- An optimization problem tries to find an optimal solution.
- A decision problem tries to answer a yes/no question.
- Many problems will have decision and optimization versions.
- Example: Traveling salesman problem.
    - optimization: find Hamiltonian cycle of minimum weight.
    - decision: is there a Hamiltonian cycle of weight <= k.

Deterministic Algorithms

- Algorithms with uniquely defined results.
- Predictable in terms of output for a certain input.
- Example: Deterministic Linear Search.

```
for i=0 to n-1 do
 if(x[i]=key) then
    return(i)
```

  - Time complexity T(n)=O(n).
  - Each time `x[i]` has one unique value to compare with key.

Nondeterministic Algorithms

- Nondeterministic algorithms are allowed to contain operations whose outcomes are limited to a given set of possibilities instead of being uniquely defined.
- Specified with the help of three new functions.

1. choice (S)
    - Arbitrarily chooses one of the elements of set S * x = choice(1,n) can result in x being assigned any of the integers in the range [1, n], in a completely arbitrary manner.
    - No rule to specify how this choice is to be made.

Nondeterministic Algorithms

2. failure()
    - Signals unsuccessful completion of a computation.
    - Cannot be used as a return value.
3. success()
    - Signals successful completion of a computation.
    - Cannot be used as a return value.
    - If there is a set of choices that leads to a successful completion, then one choice from this set must be made.

Nondeterministic Algorithms

- A nondeterministic algorithm terminates unsuccessfully iff there exist no set of choices leading to a success signal.
- A machine capable of executing a nondeterministic algorithm as above is called a nondeterministic machine.

Nondeterministic Algorithms

Example: Non-deterministic search

```
int j = choice(1,n)
if(a[j]==x) {
   write(j);
   success(); }
else {
   write(0);
   failure(); }
```

- There is no rule specifying how the choice is to be made for choice (1,n)
- A non-deterministic algorithm terminates unsuccessfully if and only if there exists no set of choices leading to a success signal.

Satisfiability SAT

- Let $x_1, x_2, \ldots$ denote Boolean variables (0 or 1).
- $\bar{x}$ denotes the negation of x.
- Formula:

  $(x_1 \wedge x_2) \vee (x_3 \wedge \overline{x_4})$ – conjunctive normal form CNF

  $(x_3 \wedge \overline{x_4}) \vee (x_1 \wedge \overline{x_2})$ – CNF
- The satisfiability problem is to determine whether a formula is true for some assignment of truth values to the variables.
- CNF satisfiability is the satisfiability problem for CNF formulas.

Satisfiability SAT

```
Eval (E, n) {
    x[size]
    //choose a truth value
    assignment
    for(i=1;i<=n;i++)
        x[i]=choice(0,1)
        if(E(x,n)
            Success();
        else
            failure();
}
```

- Time complexity
- T(n) =O(n)+length of E

The Class P (Tractable)

- P: the class of problems that have polynomial-time deterministic algorithms.
- That is, they are solvable in O(p(n)), where p(n) is a polynomial on n.
- P is just the set of tractable decision problems: the decision problems for which we have polynomial-time algorithms.
- Sample Problems in P:
  - Fractional Knapsack.
  - Minimum Spanning Tree (MST).
  - Sorting.

The Class NP (Intractable)

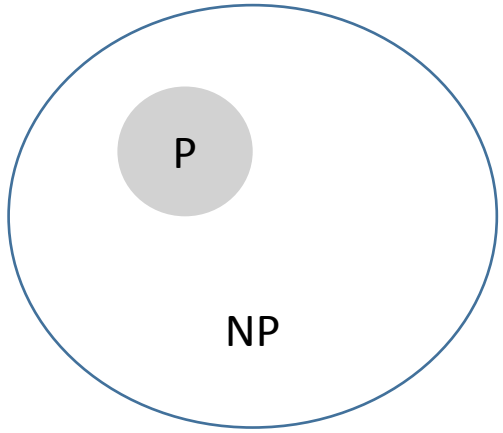
- NP: the class of decision problems that are solvable in polynomial time on a nondeterministic machine (or with a nondeterministic algorithm)
- (A deterministic computer is what we know)
- A nondeterministic computer is one that can “guess” the right answer or solution.
- Think of a nondeterministic computer as a parallel machine that can freely spawn an infinite number of processes.

The Class NP

- Thus NP can also be thought of as the class of problems “whose solutions can be verified in polynomial time”.
- Note that NP stands for “Nondeterministic Polynomial-time”.
- Sample Problems in NP:
  - Traveling Salesman.
  - Graph Coloring.
  - Satisfiability (SAT).
    - the problem of deciding whether a given Boolean formula is satisfiable.

Relationship between P and NP

- Since deterministic(P) algorithms are just a special case of NP.
- Therefore P $\subseteq$ NP.



- Commonly believed between P and NP.

Reducibility

- Let L_1 and L_2 be problems. Problem L_1 reduces to L_2 ($L_1 \propto L_2$) if and only if there is a way to solve L_1 by deterministic polynomial time algorithm using a deterministic algorithm that solves L_2 in polynomial time.
- Transitive Relation:
  - If $L_1 \propto L_2$ and $L_2 \propto L_3$, then $L_1 \propto L_3$.

Reduction

- A problem R can be reduced to another problem Q if any instance of R can be rephrased to an instance of Q, the solution to which provides a solution to the instance of R. This rephrasing is called a transformation.
- Intuitively: If R reduces in polynomial time to Q, R is “no harder to solve” than Q.
- Example: lcm(m, n) = m * n / gcd(m, n), lcm(m,n) problem is reduced to gcd(m, n) problem

NP-Hard

- A lot of times you can solve a problem by reducing it to a different problem. I can reduce Problem B to Problem A if, given a solution to Problem A, I can easily construct a solution to Problem B. (In this case, "easily" means "in polynomial time.“).
- A problem is NP-hard if all problems in NP are polynomial time reducible to it, ...
- Example:- Hamiltonian Cycle, Set Cover, Vertex Cover., Travelling Salesman.
- Every problem in NP is reducible to HC in polynomial time.
    - For Example:- TSP is reducible to HC.
    - Example: lcm(m, n) = m * n / gcd(m, n)

NP-Complete Problems

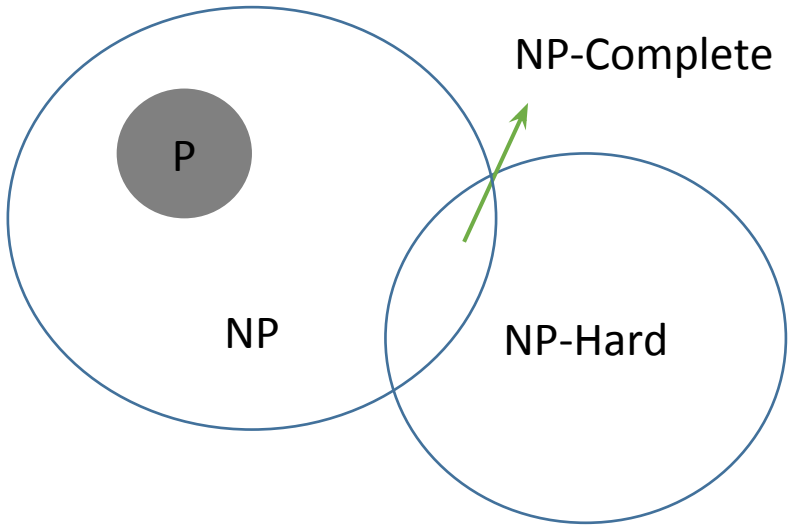
- A problem is NP-complete if the problem is both
  - NP-hard, and
  - NP.
- A decision problem C is NP-Complete if
  - C is in NP
  - Every problem in NP is reducible to C in polynomial time.
- Example:
  - Determining whether a graph has a Hamiltonian cycle.
  - Determining whether a Boolean formula is satisfiable.

NP-Complete Problems

- NP-Complete problems when expressed as decision problems are,
  - Knapsack problem.
  - Hamiltonian Path.
  - Travelling Salesman problem.
  - Subset sum problem.
  - Clique problem.
  - Graph Coloring Problem.

Relationship between P, NP, NP-Hard & NP-Complete

- All NP-Complete problems are NP-Hard.
- But some NP-Hard problems are not NP-Complete.



Summary

- P = set of problems that can be solved in polynomial time.
  - Examples: Fractional Knapsack, …
- NP = set of problems for which a solution can be verified in polynomial time.
  - Examples: Fractional Knapsack,…, TSP, CNF SAT, 3-CNF SAT
- Some problems can be translated into one another in such a way that a fast solution to one problem would automatically give us a fast solution to the other.

Summary

- There are some problems that every single problem in NP can be translated into, and a fast solution to such a problem would automatically give us a fast solution to every problem in NP. This group of problems are known as NP Complete. Example: Clique.
- A problem is NP-hard if an algorithm for solving it can be translated into one for solving any NP problem (nondeterministic polynomial time) problem. NP-hard therefore means "at least as hard as any NP-problem," although it might, in fact, be harder.

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

HAMILTONIAN CYCLE PROBLEM

Design and Analysis of Algorithms

Hamiltonian Cycle

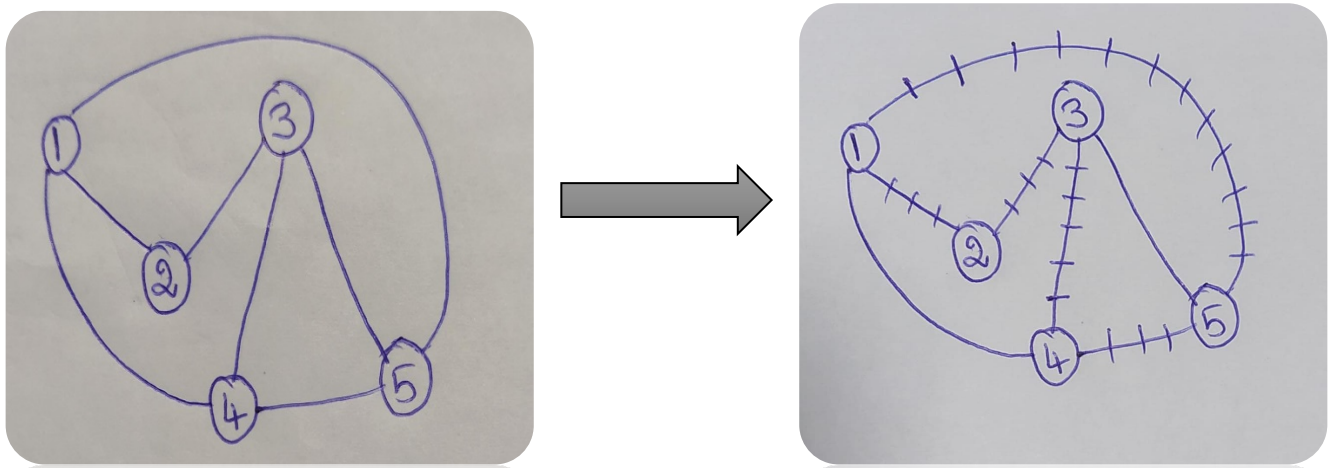
- In a connected graph, a path that visits every vertex of the graph G exactly once except the starting/ending vertices. It is Hamiltonian Cycle. (Start and end at same vertex)

Formal Definition

- Hamiltonian cycle begins at some vertex $V_1 \in G$ and the vertices of G are visited in the order $V_1, V_2, \ldots, V_{n+1}$, then the edges (V_i, V_{i+1}) are in E, $1 \leq i \leq n$, and the V_i are the distinct except for V_1 and V_{n+1} which are equal.

Hamiltonian Cycle - Example

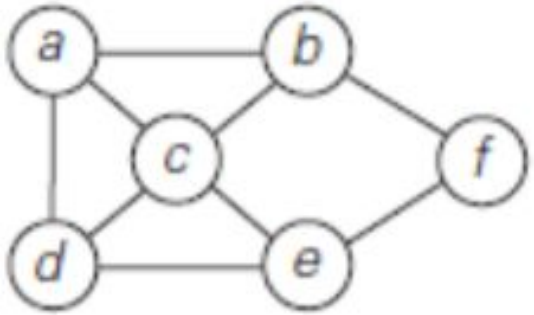
- This graph have Hamiltonian cycle.



Hamiltonian cycle: 1 – 2 – 3 – 4 – 5 - 1

Hamiltonian Cycle - Example

- Let us consider the problem of finding a Hamiltonian cycle in a graph.



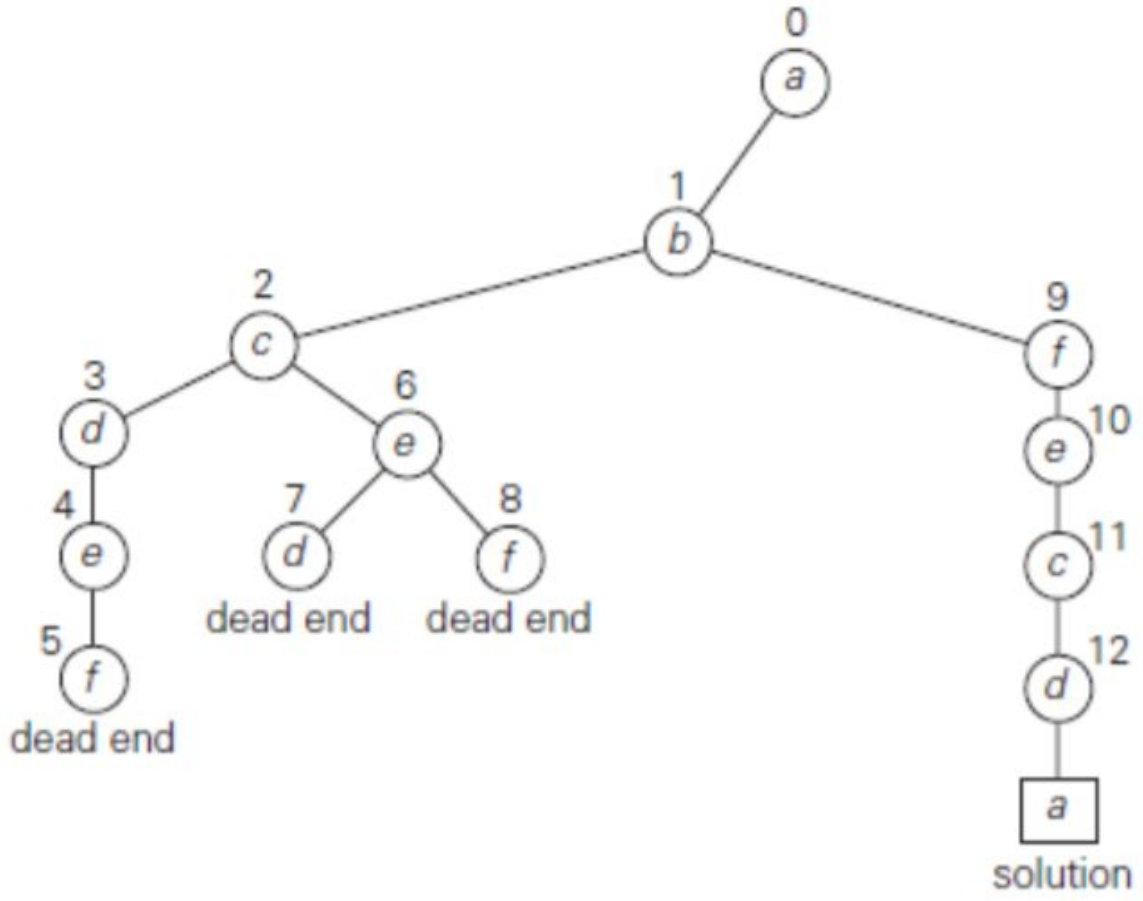
- Without loss of generality, we can assume that if a Hamiltonian cycle exists, it starts at vertex a.
- Accordingly, we make vertex a the root of the state-space.

Hamiltonian Cycle - Example

- The first component of our future solution, if it exists, is a first intermediate vertex of a Hamiltonian circuit to be constructed.
- Using the alphabet order to break the three-way tie among the vertices adjacent to a, we select vertex b.
- From b, the algorithm proceeds to c, then to d, then to e, and finally to f, which proves to be a dead end.
- So the algorithm backtracks from f to e, then to d, and then to c, which provides the first alternative for the algorithm to pursue.

Hamiltonian Cycle - Example

- The number above the nodes of tree indicate the order in which the nodes are generated.



Hamiltonian Cycle - Example

- Going from c to e eventually proves useless, and the algorithm has to backtrack from e to c and then to b.
- From there, it goes to the vertices f , e, c, and d, from which it can legitimately return to a, yielding the Hamiltonian circuit a, b,f , e, c, d, a.
- If we wanted to find another Hamiltonian circuit, we could continue this process by backtracking from the leaf of the solution found.

Hamiltonian Cycle is NP Hard problem

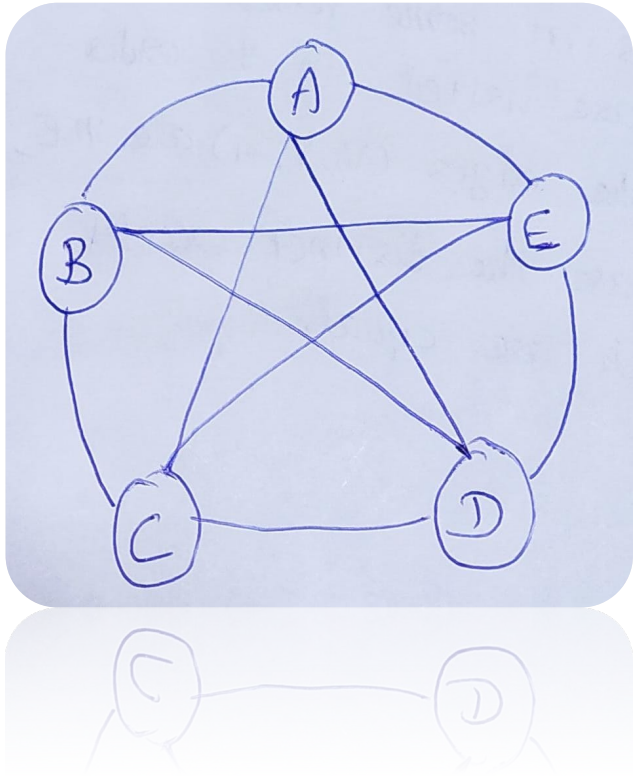
- Hamiltonian cycle is well known NP Hard problem.
- So the decision problem is here that the given graph does it have a Hamiltonian cycle?
- We would have multiple Hamiltonian cycles, Finding them is a NP hard problem as we need to consider all cases.
- So we can’t solve it in polynomial time.

Hamiltonian Cycle is NP Hard problem - Example

- This graph is a complete graph, since all nodes are connected to each other.
- Starting with any node, gives the result (Hamiltonian cycle).

A – B – C – D – E – A CYCLE

E – A – B – C – D – E CYCLE



Hamiltonian Cycle in Complete Graph

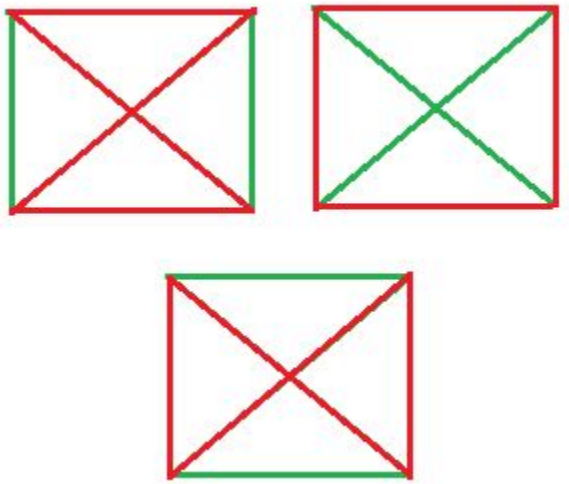
- Number of Hamiltonian cycle in complete graph = (N – 1)! /2,

  where N is number of vertices in the graph.

- When N=4,

  (N-1))!/2 3!/2 (3*2)/2 3

- Let us take the example of N = 4 complete undirected graph, The 3 different Hamiltonian cycle is as shown below:



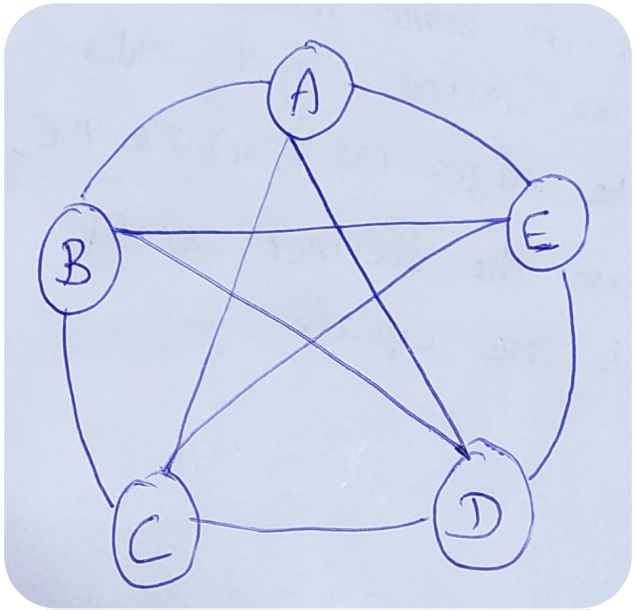
Hamiltonian Cycle in Complete Graph - Example

- **Input:** N = 6
- **Output:** Hamiltonian cycles = 60

- **Input:** N = 4
- **Output:** Hamiltonian cycles = 3

Hamiltonian Cycle - Activity

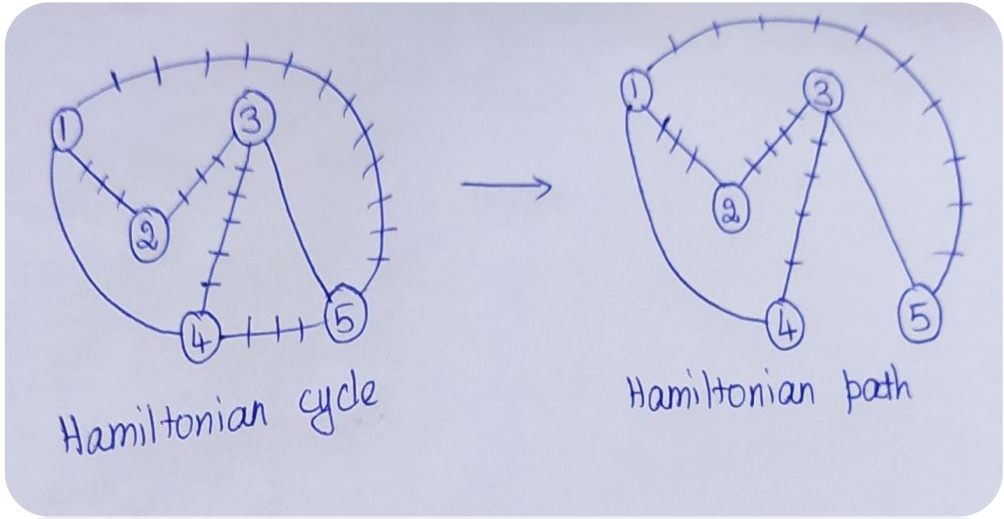
- How many Hamiltonian cycles in the given graph?



Hamiltonian Path

- Hamiltonian path is very similar problem. Instead of a cycle, you remove the requirement that you have to come back to the starting. You just start anywhere visit all the vertices and stop.

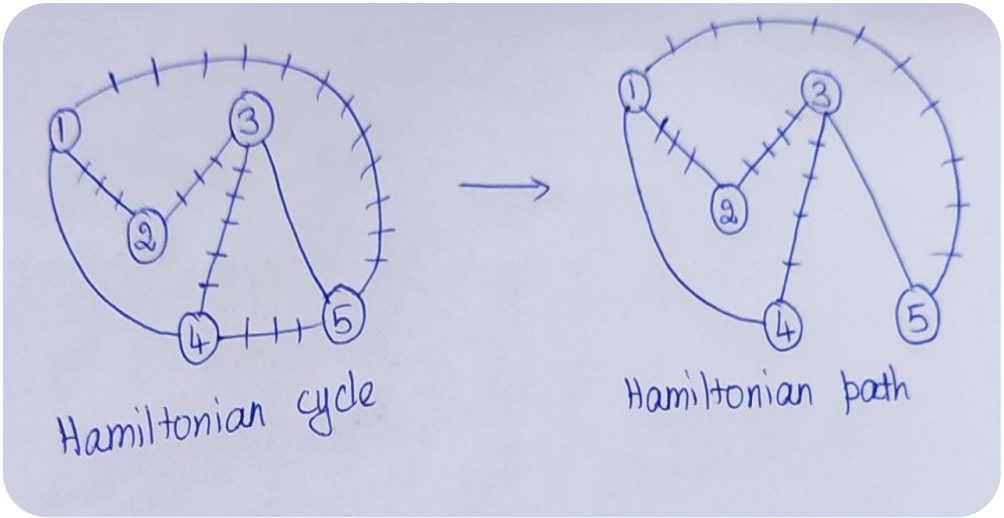
Hamiltonian Path - Example



- If you remove the edge between 4 and 5, this graph no longer has Hamiltonian cycle, but it has Hamiltonian path.

Hamiltonian Path - Example

- If you remove the edge between 4 and 5, this graph no longer has Hamiltonian cycle, but it has Hamiltonian path.
- So this is simple reduction because the problems are similar.
- Hamiltonian cycle □ Hamiltonian Path

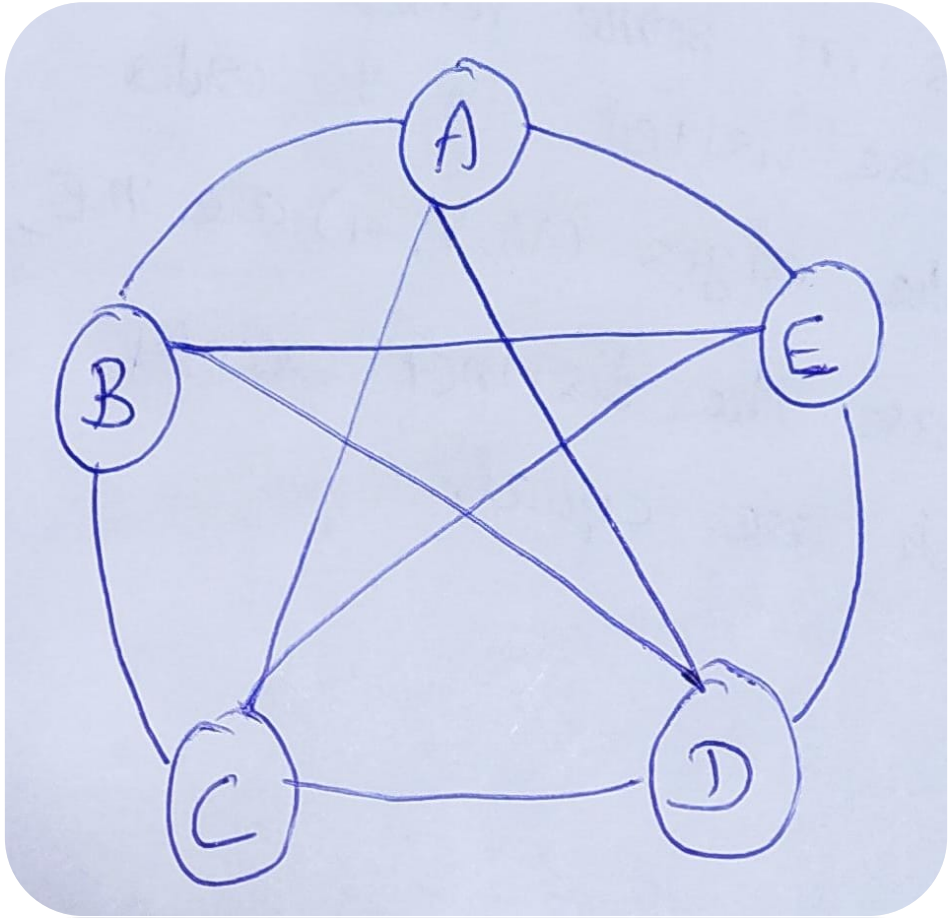


Number of Hamiltonian Path

- Possible number of Hamiltonian path = N(N-1),

  where N is the number of vertices.

- Possible number of Hamiltonian path in the above graph is N(N-1) = 5(5-1) = 5(4) = 20



Verifiable in Polynomial Time

- If someone says, OK I have solved the Hamiltonian path and this is the Hamiltonian path. And he gives you a certificate which is the actual path. So you can always look at the certificate, check the path and see if it is valid path. And then you can verify the answer in polynomial time.

Reduction

- Let us say input to Hamiltonian cycle is in the form of graph G

$$\text{Hamiltonian Cycle} - G = (V, E)$$

- Now you can transform this graph into G^1

$$\text{Hamiltonian Path} \; -- \; G^1 = (V^1, E^1)$$

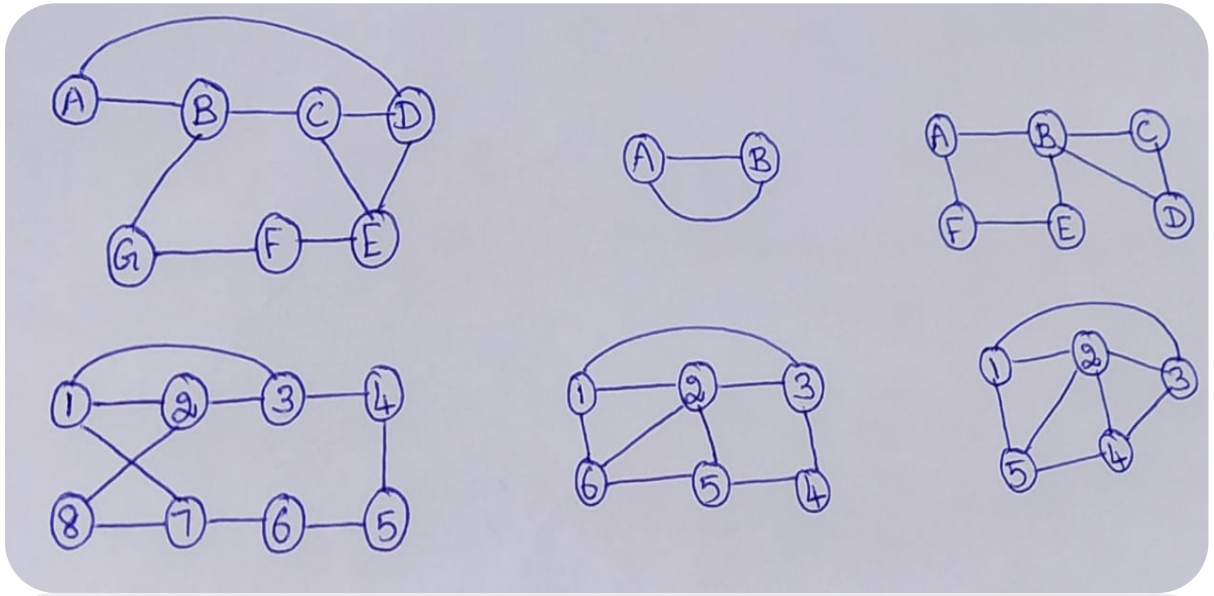
- This is transformation/Reduction R.

Terminologies

- A graph is **Hamiltonian**-connected if for every pair of vertices there is a **Hamiltonian path** between the two vertices.
- A **Hamiltonian cycle**, **Hamiltonian circuit**, vertex tour or **graph cycle** is a **cycle** that visits each vertex exactly once.
- A graph that contains a **Hamiltonian cycle** is called a **Hamiltonian graph**.

Activity

- Do Hamiltonian cycle exists in the following graph



References

- Thomas H Cormen, Charles E Leiserson, Ronald L Revest, Clifford Stein, Introduction to Algorithms, 3rd ed., The MIT Press Cambridge, 2014.
- Ellis Horowitz, Sartajsahni, Sanguthevar, Rajesekaran, Fundamentals of Computer Algorithms, Galgotia Publication, 2010.

Design and Analysis of Algorithms
Unit-5 topics

Topics to be Covered

- Approximation algorithm
- Vertex covering
- Introduction Complexity classes
- P type problems

Approximation algorithm

- Session Learning Outcome-SLO-1

  •The learners will be able to apply polynomial-time approximation algorithms for several NP-complete problems

- Examples

  - Travelling salesman problem
  - Set-cover problem
  - Subset-sum problem

Motivation of the topic

•How to find an optimal solution in polynomial time?

•Even if a problem is NP-complete, We have at least three ways to get around NP-completeness.

- •First, if the actual inputs are small, an algorithm with exponential running time may be perfectly satisfactory.
- •Second, we may be able to isolate important special cases that we can solve in polynomial time.
- •Third, we might come up with approaches to find *near-optimal solutions in* polynomial time (either in the worst case or the expected case).

•In practice, near-optimality is often good enough. We call an algorithm that returns near-optimal solutions an ***approximation algorithm.***

Performance ratios for approximation algorithms

•Suppose that we are working on an optimization problem in which each potential solution has a positive cost, and we wish to find a near-optimal solution.

•Depending on the problem, we may define an optimal solution as one with maximum possible cost or one with minimum possible cost. That is, the problem may be either a maximization or a minimization problem.

•We say that an algorithm for a problem has an ***approximation ratio of*** $\rho(n)$ ***if,*** for any input of size n, the cost C of the solution produced by the algorithm is within a factor of $\rho(n)$ of the cost C* of an optimal solution:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \le \rho(n) . \qquad (35.1)$$

Algorithm

•If an algorithm achieves an approximation ratio of $\rho(n)$ we call it a $\rho(n)$ - ***approximation algorithm.***

•The definitions of the approximation ratio and of a $\rho(n)$-approximation algorithm apply to both minimization and maximization problems.

•For a maximization problem, 0<C<=C*, and the ratio C*/C gives the factor by which the cost of an optimal solution is larger than the cost of the approximate solution.

Algorithm

•For a minimization problem, 0 < C*<= C, and the ratio C/C* gives the factor by which the cost of the approximate solution is larger than the cost of an optimal solution.

•Because we assume that all solutions have positive cost, these ratios are always well defined. The approximation ratio of an approximation algorithm is never less than 1, since C/C*<= 1 implies C*/C>= 1.

•Therefore, a 1-approximation algorithm produces an optimal solution, and an approximation algorithm with a large approximation ratio may return a solution that is much worse than optimal.

Examples

•For many problems, we have polynomial-time approximation algorithms with small constant approximation ratios, although for other problems, the best known polynomial-time approximation algorithms have approximation ratios that grow as functions of the input size n.

- •An example of such a problem is the set-cover problem (see Section 35.3 in book1).

•Some NP-complete problems allow polynomial-time approximation algorithms that can achieve increasingly better approximation ratios by using more and more computation time.

- •That is, we can trade computation time for the quality of the approximation.
- •An example is the subset-sum problem (see Section 35.5 in book1).

Approximation scheme

•An approximation scheme for an optimization problem is an approximation algorithm that takes as input not only an instance of the problem, but also a value $\varepsilon > 0$ such that for any fixed ε, the scheme is a $(1+\varepsilon)$-approximation algorithm.

•We say that an approximation scheme is a polynomial-time approximation scheme .

•if for any fixed $\varepsilon > 0$, the scheme runs in time polynomial in the size n of its input instance.

Approximation scheme

•The running time of a polynomial-time approximation scheme can increase very rapidly as ε decreases.

•For example, the running time of a polynomial-time approximation scheme might be $O(n^{2/\varepsilon})$.

•Ideally, if ε decreases by a constant factor, the running time to achieve the desired approximation should not increase by more than a constant factor (though not necessarily the same constant factor by which decreased).

Approximation scheme

•We say that an approximation scheme is a *fully polynomial-time approximation scheme if it is an approximation scheme and its running time is polynomial in* both $1/\varepsilon$ and the size n of the input instance.

•For example, the scheme might have a running time of $O((1/\varepsilon)^2 n^3)$. With such a scheme, any constant-factor decrease in ε comes with a corresponding constant-factor increase in the running time.

Summary

- Approximation algorithm gives near-optimal solution
- Approximation problem: minimization, maximization
- Performance ratio-: the cost C of the solution produced by the algorithm is within a factor of ρ(n) of the cost C* of an optimal solution.
- Approximation scheme: polynomial –time approximation, fully polynomial-time approximation.

Activity /Home assignment /Questions

- Choose anyone approximation problem, generate the mathematical solution to determine its cost. Justify why it is an approximate solution.

References

- Textbooks

Vertex covering

- Session Learning Outcome-SLO-2

  - To understand the vertex-cover problem and apply approximation scheme to find a vertex cover which is near-optimal.

- Example

- Given an undirected graph, finding a near-optimal vertex cover.

Motivation of the topic

•This problem is the optimization version of an NP-complete decision problem.

•Even though we don’t know how to find an optimal vertex cover in a graph G in polynomial time, we can efficiently find a vertex cover that is near-optimal.

The vertex cover problem

•A vertex cover of an undirected graph G = (V,E) is a subset V' $\subseteq$ V such that if (u, v) is an edge of G, then either u ϵ V' or v ϵ V' (or both). The size of a vertex cover is the number of vertices in it.

•The vertex-cover problem is to find a vertex cover of minimum size in a given undirected graph. We call such a vertex cover an optimal vertex cover.

•This problem is the optimization version of an NP-complete decision problem.

Solution

•This algorithm takes as input an undirected graph G and returns a vertex cover whose size is guaranteed to be no more than twice the size of an optimal vertex cover.

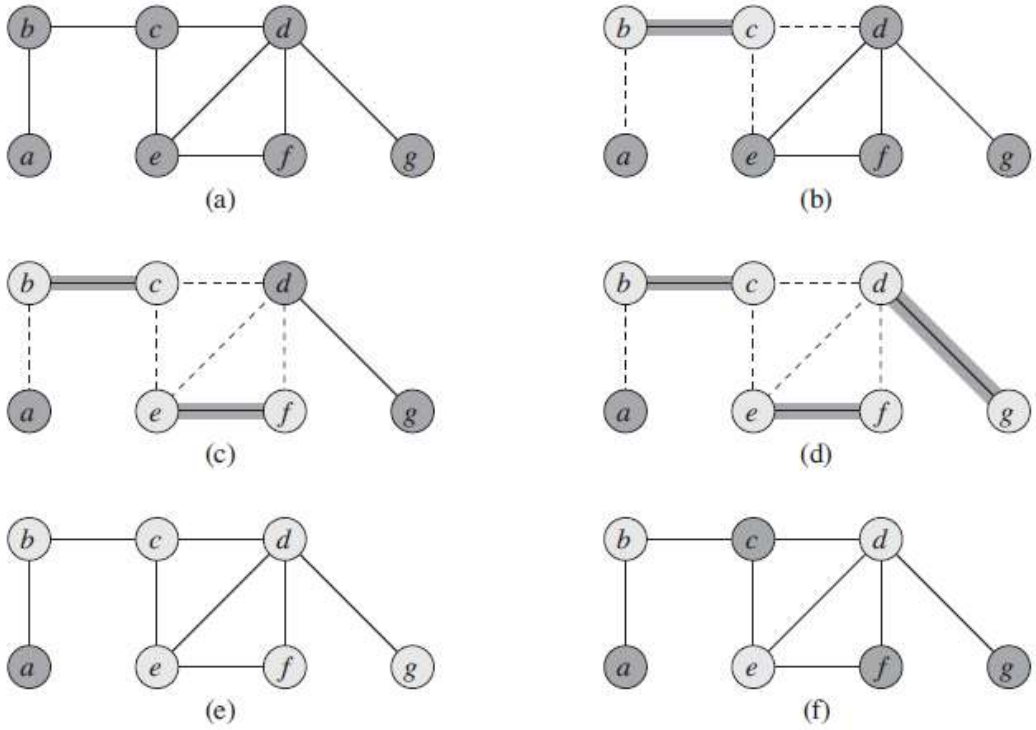


Figure 35.1 The operation of APPROX-VERTEX-COVER

Solution Description

(a) The input graph G, which has 7 vertices and 8 edges.

(b) The edge (b, c), shown heavy, is the first edge chosen by APPROX-VERTEX-COVER. Vertices b and c, shown lightly shaded, are added to the set C containing the vertex cover being created. Edges (a, b), (c, e), and (c, d), shown dashed, are removed since they are now covered by some vertex in C.

(c) Edge (e, f) is chosen; vertices e and f are added to C.

(d) Edge (d, g) is chosen; vertices d and g are added to C.

(e) The set C, which is the vertex cover produced by APPROX-VERTEX-COVER, contains the six vertices b, c, d, e, f, g.

(f) The optimal vertex cover for this problem contains only three vertices: b, d, and e.

Algorithm

- APPROX-VERTEX-COVER(G)

```
1   C=Ø
2   E‘ = G.E
3   while E’ ≠ Ø
4       let (u, v) be an arbitrary edge of E’
5       C = C ∪ {u, v}
6       remove from E’ every edge incident on either u or v
7   return C
```

Figure 35. 1 illustrates how APPROX-VERTEX-COVER operates on an example graph.

Algorithm

- The variable C contains the vertex cover being constructed.
- Line 1 initializes C to the empty set.
- Line 2 set E’ to be a copy of the edge set G.E of the graph.
- The loop of lines 3-6 repeatedly picks an edge (u, v) from E’, adds its endpoints u and v to C, and deletes all edges in E’ that are covered by either u or v.
- Finally line 7 returns the vertex cover C.
- The running time of this algorithm is O(V + E), using adjacency lists to represent E’.

Theorem 35.1

Theorem:

APPROX-VERTEX-COVER is a polynomial-time 2-approximation algorithm.

Proof:

• *We have already shown that APPROX-VERTEX-COVER runs in polynomial* time.

•The set C of vertices that is returned by APPROX-VERTEX-COVER is a vertex cover, since the algorithm loops until every edge in G.*E has been covered by some* vertex in C.

Theorem 35.1 cont.

- To see that APPROX-VERTEX-COVER returns a vertex cover that is at most twice the size of an optimal cover, let A denote the set of edges that line 4 of APPROXVERTEX-COVER picked.
- In order to cover the edges in A, any vertex cover—in particular, an optimal cover C*—must include at least one endpoint of each edge in A.
- No two edges in A share an endpoint, since once an edge is picked in line 4, all other edges that are incident on its endpoints are deleted from E' in line 6.
- Thus, no two edges in A are covered by the same vertex from C*, and we have the lower bound

$$|C^*| \geq |A| \qquad (35.2)$$

- On the size of an optimal vertex cover.

Theorem cont.

•Each execution of line 4 picks an edge for which neither of its endpoints is already in C, yielding an upper bound (an exact upper bound, in fact) on the size of the vertex cover returned:

$$|C| = 2|A| \ . \tag{35.3}$$

•Combining equations (35.2) and (35.3), we obtain

$$\begin{aligned} |C| &= 2|A| \\ &\leq 2|C^*| \ , \end{aligned}$$

•Thereby proving the theorem.

Theorem cont.

•Let us reflect on this proof. At first, you might wonder how we can possibly prove that the size of the vertex cover returned by APPROX-VERTEX-COVER is at most twice the size of an optimal vertex cover, when we do not even know the size of an optimal vertex cover.

•Instead of requiring that we know the exact size of an optimal vertex cover, we rely on a lower bound on the size.

•As Exercise 35.1-2 asks you to show, the set A of edges that line 4 of APPROX-VERTEX-COVER selects is actually a maximal matching in the graph G.

Theorem cont.

•A ***maximal matching is a matching*** that is not a proper subset of any other matching. The size of a maximal matching is, as we argued in the proof of Theorem 35.1, a lower bound on the size of an optimal vertex cover.

•The algorithm returns a vertex cover whose size is at most twice the size of the maximal matching A.

•By relating the size of the solution returned to the lower bound, we obtain our approximation ratio.

Real time applications

- Solving Optimization problem
- Keyword-based text summarization
- Computational biology

Summary

- Approximate vertex cover algorithm takes as input an undirected graph G and returns a vertex cover whose size is guaranteed to be no more than twice the size of an optimal vertex cover.

- Algorithm for APPROX-VERTEX-COVER.

- Discussed that an APPROX-VERTEX-COVER is a polynomial-time 2-approximation algorithm.

Activity /Home assignment /Questions

1. Give an example of a graph for which APPROX-VERTEX-COVER always yields a suboptimal solution.
2. Prove that the set of edges picked in line 4 of APPROX-VERTEX-COVER forms a maximal matching in the graph G.

References

- Textbooks

Introduction Complexity classes

- Session Learning Outcome-SLO

To understand whether *all problems can be solved in polynomial time.*

Example

•there are problems, such as Turing's famous "Halting Problem," that cannot be solved by any computer, no matter how much time we allow.

Motivation of the topic

•There are also problems that can be solved, but not in time $O(n^k)$ for any constant k.

•Generally, we think of problems that are solvable by polynomial-time algorithms as being tractable, or easy, and problems that require superpolynomial time as being intractable, or hard.

Algorithm and Analysis

•No polynomial-time algorithm has yet been discovered for an NP-complete problem, nor has anyone yet been able to prove that no polynomial-time algorithm can exist for any one of them.

•This so-called P ≠ NP question has been one of the deepest, most perplexing open research problems in theoretical computer science since it was first posed in 1971.

Real time applications

Shortest vs. longest simple paths:

•We have seen that even with negative edge weights, we can find *shortest paths from a single source in a directed* graph G=(V,E) in O(VE) time.

•Finding a *longest simple path between two* vertices is difficult, however. Merely determining whether a graph contains a simple path with at least a given number of edges is NP-complete.

Real time applications

Euler tour vs. hamiltonian cycle:

- An Euler tour of a connected, directed graph G = (V,E) is a cycle that traverses each edge of G exactly once, although it is allowed to visit each vertex more than once.
  - we can determine whether a graph has an Euler tour in only O(E) time and, in fact, we can find the edges of the Euler tour in O(E) time.

- A hamiltonian cycle of a directed graph G =(V,E) is a simple cycle that contains each vertex in V .
  - Determining whether a directed graph has a hamiltonian cycle is NP-complete.

Real time applications

CNF satisfiability vs. 3-CNF satisfiability:

•A boolean formula contains variables whose values are 0 or 1;

•boolean connectives such as ^ (AND), v (OR), and : (NOT); and parentheses.

•A boolean formula is *satisfiable if there exists* some assignment of the values 0 and 1 to its variables that causes it to evaluate to 1.

•Informally, 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.

NP-completeness and the classes P and NP

•Class P: The class P consists of those problems that are solvable in polynomial time.

- •More specifically, 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 NP: The 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.

Example

- Example 1: In the hamiltonian cycle problem, given a directed graph G = (V,E), a certificate would be a sequence $v_1, v_2, v_3, \ldots, V_{|v|}$ of |V| vertices.
  - We could easily check in polynomial time that $(v_i, v_{i+1}) \epsilon E$ for i = 1, 2, 3,…, |V|-1 and that $(V_{|v|}, v_1) \epsilon E$ as well.
- Example 2: for 3-CNF satisfiability, a certificate would be an assignment of values to variables.
  - 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.
  - We shall now believe that $\mathbf{P} \subseteq \mathbf{NP}$.
  - whether or not P is a proper subset of NP.?

NP-Complete

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.

We rely on three key concepts in showing a problem to be NP-complete:

1. Decision problems (DP) vs. optimization problems (OP)

•Many problems of interest are ***optimization problems,*** in which each feasible (i.e., "legal") solution has an associated value, and we wish to find a feasible solution with the best value.

NP-Complete

For example,

•in a problem that we call SHORTEST-PATH, we are given an undirected graph G and vertices u and , and we wish to find a path from u to v that uses the fewest edges. In other words, SHORTEST-PATH is the single-pair shortest-path problem in an unweighted, undirected graph.

•NP-Completeness applies directly not to optimization problems, however, but to decision problems, in which the answer is simply “yes” or “no” (or, more formally, “1” or “0”).

•Note: if an optimization problem is easy, its related decision problem is easy as well. Stated in a way that has more relevance to NP-completeness, if we can provide evidence that a decision problem is hard, we also provide evidence that its related optimization problem is hard.

NP-Complete

2. Reductions

•Let us consider a decision problem A, which we would like to solve in polynomial time.

•We call the input to a particular problem an instance of that problem*; for example, in PATH,* an instance would be a particular graph G, particular vertices u and v of G, and a particular integer k. Now suppose that we already know how to solve a different decision problem B in polynomial time.

•Finally, suppose that we have a procedure that transforms any instance α of A into some instance β of B with the following characteristics:

- 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."

NP-Complete

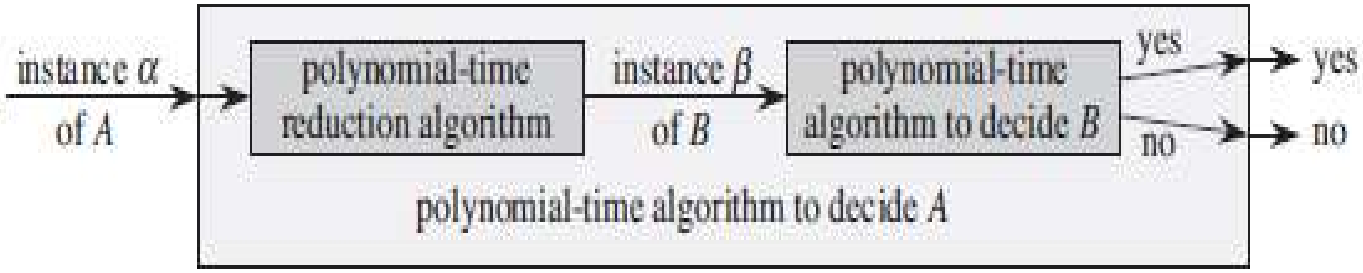


Figure 34.1 How to use a polynomial-time reduction algorithm to solve a decision problem A in polynomial time, given a polynomial-time decision algorithm for another problem B. In polynomial time, we transform an instance α of A into an instance β of B, we solve B in polynomial time, and we use the answer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

NP-Complete

•We call such a procedure a polynomial-time reduction algorithm and, as Figure 34.1 shows, it provides us a way to solve problem A in polynomial time:

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.

•by “reducing” solving problem A to solving problem B, we use the “easiness” of B to prove the “easiness” of A.

NP-Complete

3. A first NP-complete problem

•Because the technique of reduction relies on having a problem already known to be NP-complete in order to prove a different problem NP-complete, we need a “first” NP-complete problem.

•The problem we shall use is the 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.

NP-completeness and the classes P and NP

- Summary

•we formalized our notion of “problem” and defined the complexity class P of polynomial-time solvable decision problems. We have seen how these notions fit into the framework of formal-language theory.

•Defined the class NP of decision problems whose solutions are verifiable in polynomial time.

•Shown that we can relate problems via polynomial-time “reductions.” It defines NP-completeness.

- Activity /Home assignment /Questions
  - Identify any three problems which are in P
  - Identify any three problems which are in NP
  - Justify the relationship between P and NP
- References
  - Textbooks

P type problems

- Session Learning Outcome-SLO
  - To learn how to formalize our notion of polynomial time solvable problems.

•The polynomial-time computable problems encountered in practice typically require much less time.

•Experience has shown that once the first polynomial-time algorithm for a problem has been discovered, more efficient algorithms often follow.

•Even if the current best algorithm for a problem has a running time of $\theta(n^{100})$, an algorithm with a much better running time will likely soon be discovered.

P-type problems

•for many reasonable models of computation, a problem that can be solved in polynomial time in one model can be solved in polynomial time in another.

•For example, the class of problems solvable in polynomial time by the serial random-access machine is the same as the class of problems solvable in polynomial time on abstract Turing machines.

• It is also the same as the class of problems solvable in polynomial time on a parallel computer when the number of processors grows polynomially with the input size.

P-type problems

•the class of polynomial-time solvable problems has nice closure properties, since polynomials are closed under addition, multiplication, and composition.

•For example, if the output of one polynomial-time algorithm is fed into the input of another, the composite algorithm is polynomial.

P-type problems

•We define an *abstract problem Q to be a* binary relation on a set I of problem instances and a set *S* of problem solutions.

•For example, 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.

•The problem SHORTEST-PATH itself is the 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.

Real time applications

•An ***encoding of a set S*** of abstract objects is a mapping e from S to the set of binary strings.

For example,

we are all familiar with encoding the natural numbers N ={0, 1, 2,3,…} as the strings {0, 1, 10, 11,…|. Using this encoding, e(17) = 10001.

P-type problems

- Summary
  - Discussed the class of polynomial-time solvable problems
- Activity /Home assignment /Questions

  •Show that if an algorithm makes a constant number of calls to polynomial-time subroutines and performs an additional amount of work that also takes polynomial time, then the running time of the composite algorithm is polynomial.

- References
  - Textbooks

The Satisfiability Problem

Cook's Theorem: An NP-Complete Problem

Boolean Expressions

- Boolean, or propositional-logic expressions are built from variables and constants using the operators AND, OR, and NOT.
  - Constants are true and false, represented by 1 and 0, respectively.
  - We'll use concatenation (juxtaposition) for AND, + for OR, - for NOT, unlike the text.

Example: Boolean expression

- (x+y)(-x + -y) is true only when variables x and y have opposite truth values.
- Note: parentheses can be used at will, and are needed to modify the precedence order NOT (highest), AND, OR.

The Satisfiability Problem (*SAT*)

- Study of boolean functions generally is concerned with the set of *truth assignments* (assignments of 0 or 1 to each of the variables) that make the function true.
- NP-completeness needs only a simpler question (SAT): does there exist a truth assignment making the function true?

Example: SAT

- (x+y)(-x + -y) is satisfiable.
- There are, in fact, two satisfying truth assignments:
  1. x=0; y=1.
  2. x=1; y=0.
- x(-x) is not satisfiable.

SAT as a Language/Problem

- An instance of SAT is a boolean function.
- Must be coded in a finite alphabet.
- Use special symbols (,), +, - as themselves.
- Represent the i-th variable by symbol x followed by integer i in binary.

Example: Encoding for SAT

- (x+y)(-x + -y) would be encoded by the string `(x1+x10)(-x1+-x10)`

What is NP-completeness?

- Consider the circuit satisfiability problem
- Difficult to answer the decision problem in polynomial time with the classical deterministic algorithms

Nondeterministic algorithms

- A nondeterminstic algorithm consists of

  phase 1: guessing

  phase 2: checking

- If the checking stage of a nondeterministic algorithm is of polynomial time-complexity, then this algorithm is called an NP (nondeterministic polynomial) algorithm.

Nondeterministic searching algorithm

- Search for x in an array A
- Choice(S) : arbitrarily chooses one of the elements in set S
- Failure : an unsuccessful completion
- Success : a successful completion
- Nonderministic searching algorithm (which will be performed with unbounded parallelism):

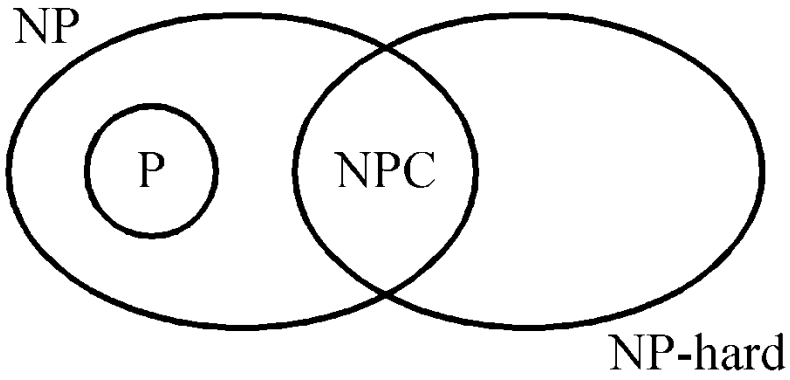
```
j ← choice(1 : n)    /* guessing */
if A(j) = x then success  /* checking */
            else failure
```

- A nondeterministic algorithm terminates unsuccessfully iff there exist not a set of choices leading to a success signal.
- A deterministic interpretation of a non-deterministic algorithm can be made by allowing unbounded parallelism in computation.
- The runtime required for *choice(1 : n)* is O(1).
- The runtime for nondeterministic searching algorithm is also O(1)

Nondeterministic sorting

```
B ← 0
/* guessing */
for i = 1 to n do
     j ← choice(1 : n)
     if B[j] ≠ 0 then failure
     B[j] = A[i]
/* checking */
for i = 1 to n-1 do
    if B[i] > B[i+1] then failure
    success
```

Perform the above with unbounded parallelism



- **NP** : the class of decision problem which can be solved by a non-deterministic polynomial algorithm.
- **P**: the class of problems which can be solved by a deterministic polynomial algorithm.
- **NP-hard**: the class of problems to which every NP problem reduces.
- **NP-complete (NPC)**: the class of problems which are NP-hard and belong to NP.

Some concepts of NP Complete

- 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. B is harder.
- 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.

SAT is in **NP**

- There is a multitape NTM that can decide if a Boolean formula of length n is satisfiable.
- The NTM takes $O(n^2)$ time along any path.
- Use nondeterminism to guess a truth assignment on a second tape.
- Replace all variables by guessed truth values.
- Evaluate the formula for this assignment.
- Accept if true.

Cook's Theorem

- SAT is NP-complete.
  - Really a stronger result: formulas may be in conjunctive normal form (CSAT) – later.
- To prove, we must show how to construct a polytime reduction from each language L in **NP** to SAT.
- Start by assuming the most resticted possible form of NTM for L (next slide).

Assumptions About NTM for L

1. One tape only.
2. Head never moves left of the initial position.
3. States and tape symbols are disjoint.

- Key Points: States can be named arbitrarily, and the constructions many-tapes-to-one and two-way-infinite-tape-to-one at most square the time.

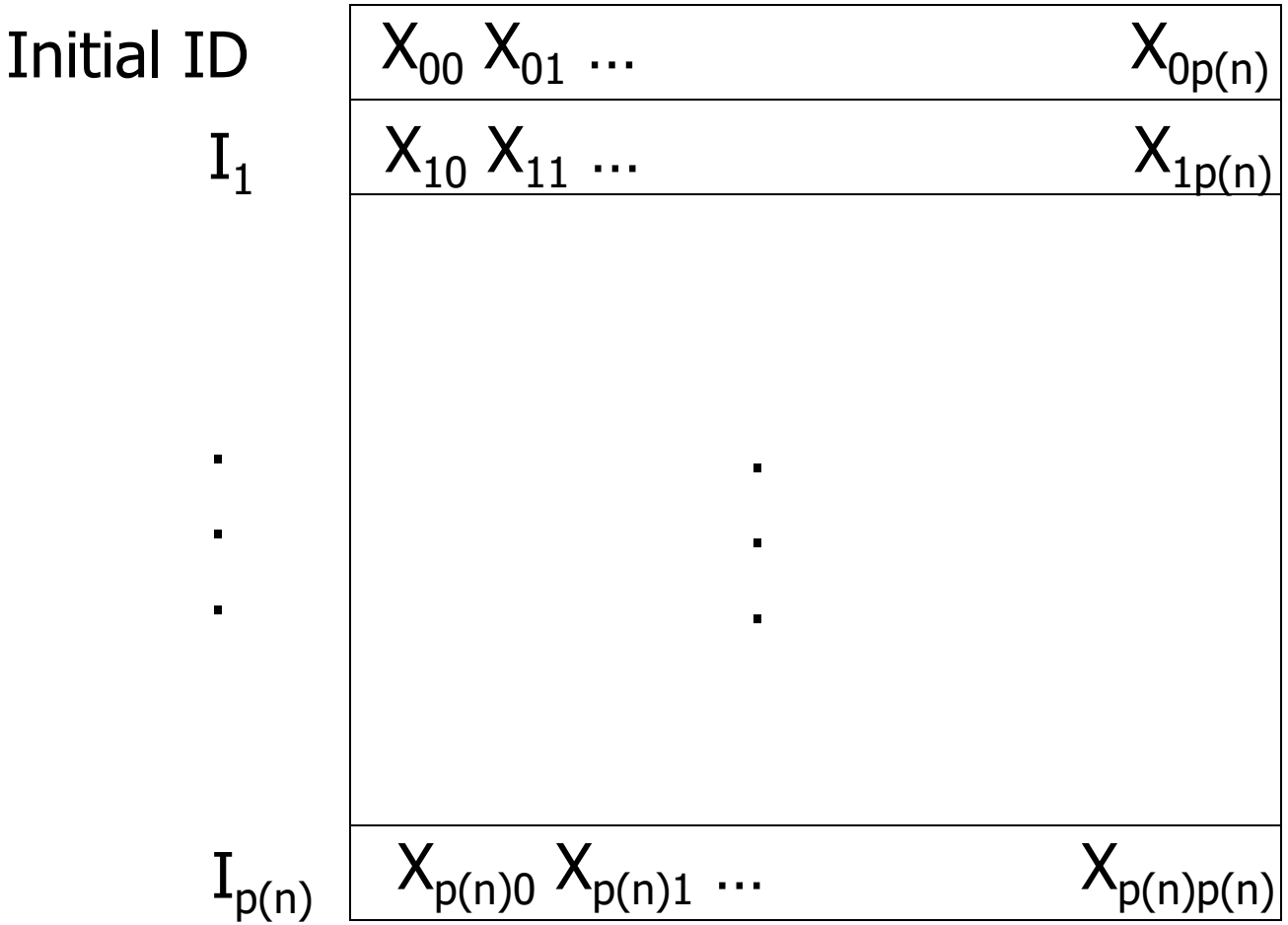
More About the NTM M for L

- Let p(n) be a polynomial time bound for M.
- Let w be an input of length n to M.
- If M accepts w, it does so through a sequence $I_0 \vdash I_1 \vdash \ldots \vdash I_{p(n)}$ of p(n)+1 ID's.
  - Assume trivial move from a final state.
- Each ID is of length at most p(n)+1, counting the state.

From ID Sequences to Boolean Functions

- The Boolean function that the transducer for L will construct from w will have $(p(n)+1)^2$ "*variables*."
- Let variable X_{ij} represent the j-th position of the i-th ID in the accepting sequence for w, if there is one.
  - i and j each range from 0 to p(n).

Picture of Computation as an Array



3SAT

- This problem is NP-complete.
- Clearly it is in **NP:**
  1. Guess an assignment of true and false to the variables
  2. Test whether the Boolean formula is true.

3SAT – (2)

- We need to reduce every CNF formula F to some 3-CNF formula that is satisfiable if and only if F is.
- Reduction involves introducing new variables into long clauses, so we can split them apart.

NP-Completeness of CSAT

- The proof of Cook's theorem can be modified to produce a formula in CNF.
- Unique is already the AND of clauses.
- Starts Right is the AND of clauses, each with one variable.
- Finishes Right is the OR of variables, i.e., a single clause.

NP-Completeness of CSAT – (2)

- Only Moves Right is a problem, and not much of a problem.
- It is the product of formulas for each i and j.
- Those formulas are fixed, independent of n.

NP-Completeness of CSAT – (3)

- You can convert any formula to CNF.
- It may exponentiate the size of the formula and therefore take time to write down that is exponential in the size of the original formula, but these numbers are all fixed for a given NTM M and independent of n.

CSAT & 3SAT

- CSAT: an intermediate step that is used to show that SAT can be reduced to 3SAT

- 3SAT: is a problem about satisfiability of boolean expressions, these expression have a very regular form:
  - they are the AND of “clauses”.
  - each clause is the OR of exactly three variables or negated variables.

Reduction of CSAT to 3SAT

- Let $(x_1+\ldots+x_n)$ be a clause in some CSAT instance, with $n \geq 4$.
  - Note: the x's are literals, not variables; any of them could be negated variables.
- Introduce new variables $y_1,\ldots,y_{n-3}$ that appear in no other clause.

CSAT to 3SAT – (2)

- Replace $(x_1+\ldots+x_n)$ by
  $(x_1+x_2+y_1)(x_3+y_2+ -y_1) \ldots (x_i+y_{i-1}+ -y_{i-2})$
  $\ldots (x_{n-2}+y_{n-3}+ -y_{n-4})(x_{n-1}+x_n+ -y_{n-3})$
- If there is a satisfying assignment of the x's for the CSAT instance, then one of the literals x_i must be made true.
- Assign y_j = true if $j < i-1$ and y_j = false for larger j.

CSAT to 3SAT – (3)

- We are not done.
- We also need to show that if the resulting 3SAT instance is satisfiable, then the original CSAT instance was satisfiable.

CSAT to 3SAT – (4)

- Suppose $(x_1+x_2+y_1)(x_3+y_2+ -y_1)$ … $(x_{n-2}+y_{n-3}+ -y_{n-4})(x_{n-1}+x_n+ -y_{n-3})$ is satisfiable, but none of the x's is true.
- The first clause forces y_1 = true.
- Then the second clause forces y_2 = true.
- And so on … all the y's must be true.
- But then the last clause is false.

CSAT to 3SAT – (5)

- There is a little more to the reduction, for handling clauses of 1 or 2 literals.
- Replace (x) by $(x+y_1+y_2)$ $(x+y_1+ -y_2)$ $(x+ -y_1+y_2)$ $(x+ -y_1+ -y_2)$.
- Replace (w+x) by (w+x+y)(w+x+ -y).
- Remember: the y's are different variables for each CNF clause.

CSAT to 3SAT Running Time

- This reduction is surely polynomial.
- In fact it is linear in the length of the CSAT instance.
- Thus, we have polytime-reduced CSAT to 3SAT.
- Since CSAT is NP-complete, so is 3SAT.

The NP-Hard Problem

P, NP, NP Hard, NP Complete

- Focus of this lecture = get the definitions across
- This will be a part of midterm2
- Only some small part of the next lecture will be part of the midterm
- Next lecture = some complexity examples (non NP complete)

The class P

- The class P consists of those problems that are solvable in polynomial time.
- More specifically, 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
- The key is that n is the **size of input**

NP

- **NP is not the same as non-polynomial complexity/running time. NP does not stand for not polynomial.**
- **NP = Non-Deterministic polynomial time**
- NP means verifiable in polynomial time
- Verifiable?
  - If we are somehow given a ‘certificate’ of a solution we can verify the legitimacy in polynomial time

NP problems

- Graph theory has these fascinating(annoying?) pairs of problems
  - Shortest path algorithms?
  - Longest path is NP complete (we'll define NP complete later)
  - Eulerian tours (visit every vertex but cover every edge only once, even degree etc). Solvable in polynomial time!
  - Hamiltonian tours (visit every vertex, no vertices can be repeated). NP complete

Hamiltonian cycles

- Determining whether a directed graph has a Hamiltonian cycle does not have a polynomial time algorithm (yet!)
- However if someone was to give you a sequence of vertices, determining whether or not that sequence forms a Hamiltonian cycle can be done in polynomial time
- Therefore Hamiltonian cycles are in NP

What is not in NP?

- Undecidable problems
  - Given a polynomial with integer coefficients, does it have integer roots
  - Hilbert's nth problem
  - Impossible to check for all the integers
  - Even a non-deterministic TM has to have a finite number of states!
  - More on decidability later
- Tautology
  - A boolean formula that is true for all possible assignments
  - Here just one 'verifier' will not work. You have to try all possible values

Reducibility

- a problem Q can be reduced to another problem Q' if any instance of Q can be "easily rephrased" as an instance of Q', the solution to which provides a solution to the instance of Q
- Is a linear equation reducible to a quadratic equation?
  - Sure! Let coefficient of the square term be 0

“Hard” and “easy’ Problems

- Sometimes the dividing line between “easy” and “hard” problems is a fine one. For example
  - Find the shortest path in a graph from X to Y. (easy)
  - Find the longest path in a graph from X to Y. (with no cycles) (hard)
- View another way – as “yes/no” problems
  - Is there a simple path from X to Y with weight <= M? (easy)
  - Is there a simple path from X to Y with weight >= M? (hard)
  - First problem can be solved in polynomial time.
  - All known algorithms for the second problem (could) take exponential time .

- Decision problem: The solution to the problem is "yes" or "no". Most optimization problems can be phrased as decision problems (still have the same time complexity).

  Example :
  Assume we have a decision algorithm X for 0/1 Knapsack problem with capacity M,
  i.e. Algorithm X returns “Yes” or “No” to the question
  “Is there a solution with profit $\geq P$ subject to knapsack capacity $\leq M$?”

NP-completeness and NP-hardness

NP-complete: class of problems **X** such that every problem from **NP** is polynomial-time reducible to **X**.

Optimization problems: problems where the answer is a number (maximum/minimum possible)

Each optimization problem has its decision version, e.g.,

- Find a **maximum** Independent Set
- Is there an Independent Set **of size** k?

NP-hard: class of optimization problems **X** such that its decision version is **NP-complete**.

Example: having solution for decision version of Independent Set problem, we can probe a parameter k, starting from $k = 1$, to find the size of the maximum independent set

NP - hard

- What are the hardest problems in NP?

$$L_1 \leq_p L_2$$

- That notation means that L_1 is reducible in polynomial time to L_2 .
- The less than symbol basically means that the time taken to solve L_1 is no worse that a polynomial factor away from the time taken to solve L_2.

NP-hard

- A problem (a language) is said to NP-hard if every problem in NP can be poly time reduced to it.

$$L' \leq_p L \text{ for every } L' \in NP$$

Knapsack problem

Input: set of n items, each represented by its weight w_i and value v_i ; thresholds W and V

Decision problem: is there a set of items of total weight at most W and total value V ?

Optimization problem: find a set of items with

- total weight at most W , and
- maximum possible value

Assumptions:

- weights and values are positive integers
- each weight is at most W

NP-hardness of knapsack

Knapsack is NP-hard problem, but

there exists **pseudo-polynomial** algorithm (complexity is polynomial in terms of values)

Typical numerical polynomial algorithm: polynomial in logarithm from the maximum values (longest representation)

Existence of pseudo-polynomial solution often yields very good approximation schemes

Dynamic pseudo-polynomial optimization algorithm

Let v^* be the maximum (integer) value of an item.

Consider any order of objects.

Let $OPT(i,v)$ denote the minimum possible total weight of a subset of items $1,2,\ldots,i$ which has total value v

Dynamic formula for $i = 0,1,\ldots,n-1$ and $v = 0,1,\ldots,nv^*$:

$OPT(i+1,v) =$

$= \min\{ OPT(i,v) , w_{i+1} + OPT(i,\max\{0,v-v_{i+1}\})\}$

Formula OPT does not provide direct solution for our problem, but can be easily adapted: maximum value of knapsack is the maximum value v such that $OPT(n,v) \leq W$

Dynamic algorithm

Initialize array $M[0\ldots n, 0\ldots nv^*]$ for storing $OPT(i,v)$

Fill positions $M[\cdot,0]$ and $M[0,\cdot]$ with zeros

For $i = 0,1,\ldots,n-1$

For $v = 0,1,\ldots,nv^*$

$M[i+1,v] :=$

$= \min\{ M[i,v] , w_{i+1} + M[i,\max\{0,v-v_{i+1}\}] \}$

Go through the whole array M and find the maximum value v such that $M[n,v] \leq W$

Complexities

Time: $O(n^2v^*)$

- Initializing array M : $O(n^2v^*)$
- Iterating loop: $O(n^2v^*)$
- Searching for maximum v : $O(n^2v^*)$

Memory: $O(n^2v^*)$

Polynomial approximation algorithm

Algorithm:

- Fix $b = (\varepsilon/(2n))\ v^*$
- Set (by rounding up) $x_i = [v_i/b]$
- Solve knapsack problem for values x_i and weights w_i using dynamic program
- Return set of computed items and its total value in terms of the sum of v_i's

Analysis

PTAS: polynomial time approximation scheme - for any fixed positive ε it produces $(1+\varepsilon)$-approximation in polynomial time (but ε is hidden in big Oh)

Time: $O(n^2 x^*) = O(n^3/\varepsilon)$

Approximation: $(1+\varepsilon)$

Analysis of approximation ratio

Recall notation:

- $b = (\varepsilon/(2n))\, v^*$
- $x_i = [v_i/b]$

Approximation: $(1+\varepsilon)$

Let S denote the set of items returned by the algorithm

- $v_i \le bx_i \le v_i + b \Rightarrow \sum_{i\in S} bx_i - b|S| \le \sum_{i\in S} v_i$

$$\sum_{i\in S} bx_i \ge v^* = 2nb/\varepsilon \Rightarrow (2/\varepsilon - 1)nb \le \sum_{i\in S} v_i$$

$$\sum_{i\in OPT} v_i \le \sum_{i\in OPT} bx_i \le \sum_{i\in S} bx_i \le b|S| + \sum_{i\in S} (bx_i - b) \le$$

$$b(2/\varepsilon - 1)n\varepsilon + \sum_{i\in S} v_i \le \varepsilon \sum_{i\in S} v_i + \sum_{i\in S} v_i = (1+\varepsilon) \sum_{i\in S} v_i$$

Weighted Independent Set

Optimization problem:

Weighted Independent Set: given graph G of *n* valued nodes, find an independent set of maximum value (set of nodes such that none two of them are connected by an edge)

Even for values 1 problem remains NP-hard, which is not the case for knapsack problem! WIS problem is an example of **strong NP-hard** problem, and no PTAS is known for it