

Module_4

Complexity Theory & Parallel Algorithms

- Computational Complexity:
 - Non Deterministic algorithms (Searching, sorting, knapsack, clique, sum of subsets),
 - The classes: P, NP, NP Complete, NP Hard,
 - Satisfiability problem
 - Proofs for NP Complete Problems: Clique, Vertex Cover.
 - Parallel Algorithms:
 - Basics of approximation
 - Randomised algorithm
 - Parallel Algorithms
- Reference Books: Sami

Basic Concepts

Many of the problems that we know and study, can be clustered into **two groups** based on the computing times for solving using best algorithms.

•Polynomial :

- Problems whose solution times are bounded by polynomials of small degree

Searching: $(\log n)$,

Sorting (heap) : $n \log n$

•Non-polynomial:

- Problems whose best known algorithms are non-polynomial.

Travelling salesman problem $O(n^2 2^n)$

knap-sack $O(2^{n/2})$

Deterministic Algorithms

- Deterministic Algorithms :
 - They have the property that the result of every operation is **uniquely defined**
 - These algorithms agree the way the programs are executed on

Non-Deterministic Algorithms

- Non-Deterministic Algorithms :
 - They have the property that the result of every operation is not uniquely defined but they are limited to specified sets of possibilities
 - Three new functions are used to specify such algorithms.
 - Choice(S) : arbitrarily chooses one of the elements of set S
 - Failure(): signals unsuccessful termination
 - Success(): signals successful termination
 - Failure() and Success() cannot be used to effect a return
- Computing times of Choice, Failure, Success are taken to be $O(1)$

Non-Deterministic Algorithms

- Whenever there is a set of choices that leads to successful termination, one such set of choices is always made and algorithm terminates successfully.
- A non-deterministic algorithm terminates unsuccessfully if and only if there exists no set of choices leading to a **success signal**.
- Machine capable of executing non-deterministic algorithms is said to be nondeterministic machine.
- Nondeterministic machines do not exist in practice.
- They provide strong intuitive reasons to conclude that certain problems cannot be solved by fast deterministic algorithms



Non Deterministic search

1. if $a[j] = x$ then { write (j); Success(); }
2. $j = \text{Choice}(1, n)$
3. write(0); Failure();

Non-deterministic complexity $O(1)$ A is non ordered, every deterministic search algorithm is of complexity $\Omega(n)$

Non-deterministic sorting

```
Algorithm Nsort ( A, n)
// Sort n positive integers
{
  for i= 1 to do B[i] = 0; // initialize B[]
  for j= 1 to n do
    {
      j= Choice(1,n);
      if( B[j] !=0) then Failure();
      B[j] = A[i];
    }
  for i= 1 to n-1 do // verify order
    if( B[i] > B[i+1] then Failure();
  write( B[1..n]);
  Success();
}
```

Complexity Class P

- **Deterministic** in nature
- Solved by conventional computers in **polynomial time**

➤ $O(1)$	Constant
➤ $O(\log n)$	Sub-linear
➤ $O(n)$	Linear
➤ $O(n \log n)$	Nearly Linear
➤ $O(n^2)$	Quadratic

- Polynomial upper and lower bounds

Complexity Class NP

- Non-deterministic part as well
- choose(b): choose a bit in a non-deterministic way and assign to b
- If someone tells us the solution to a problem, we can verify it in polynomial time
- Two Properties: non-deterministic method to generate possible solutions, deterministic method to verify in polynomial time that the solution is correct.

Polynomial-Time Reducibility

- Language L is polynomial-time reducible to language M if there is a function computable in polynomial time that takes an input x of L and transforms it to an input $f(x)$ of M , such that x is a member of L if and only if $f(x)$ is a member of M .
- Shorthand, $L_{\text{poly}}M$ means L is polynomial-time reducible to M

NP-Hard and NP-Complete

- Language M is NP-hard if every other language L in NP is polynomial-time reducible to M
- For every L that is a member of NP, $L \leq_{\text{poly}} M$
- If language M is NP-hard and also in the class of NP itself, then M is NP-complete

NP-Hard and NP-Complete

- **Restriction:** A known NP-complete problem M is actually just a special case of L
- **Local replacement:** reduce a known NP-complete problem M to L by dividing instances of M and L into “basic units” then showing each unit of M can be converted to a unit of L
- **Component design:** reduce a known NP-complete problem M to L by building components for an instance of L that enforce important structural functions for instances of M .

Non-deterministic polynomial time

- **Deterministic Polynomial Time:**
 - The TM takes at most $O(nc)$ steps to accept a string of length n
- **Non-deterministic Polynomial Time:**
 - The TM takes at most $O(nc)$ steps on each computation path to accept a string of length n

The Class P and the Class NP

- P = Set of deterministic problems that can be solved in polynomial time.
- NP = Set of non-deterministic problems that can be solved in polynomial time.

$P = NP?$

- No one knows if this is true
- How can we make progress on this problem?

$P = NP?$

- No one knows if this is true
- How can we make progress on this problem?

Progress

- $P = NP$ if every NP problem has a deterministic polynomial algorithm
- We could find an algorithm for every NP problem
- Seems... hard...
- We could use polynomial time reductions to find the “hardest” problems and just work on those

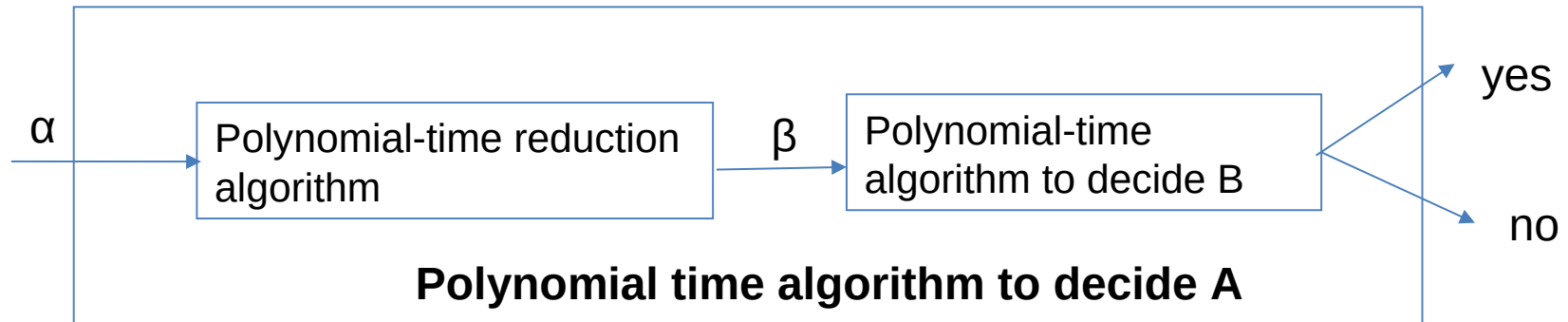
Reductions

- Let us consider a decision problem A which has to be solve in polynomial time
- We call input to that problem an instance of that problem
- Now suppose there is different decision problem say B that we already know how to solve in polynomial time.
- Finally suppose that we have a procedure that transforms any instance α of A into some instance β of B

Reductions

- The transformation takes polynomial time
- The answers are the same. That is the answer for α is “yes” if and only if the answer for β is also “yes”
- We call such a procedure a polynomial time **reduction algorithm**

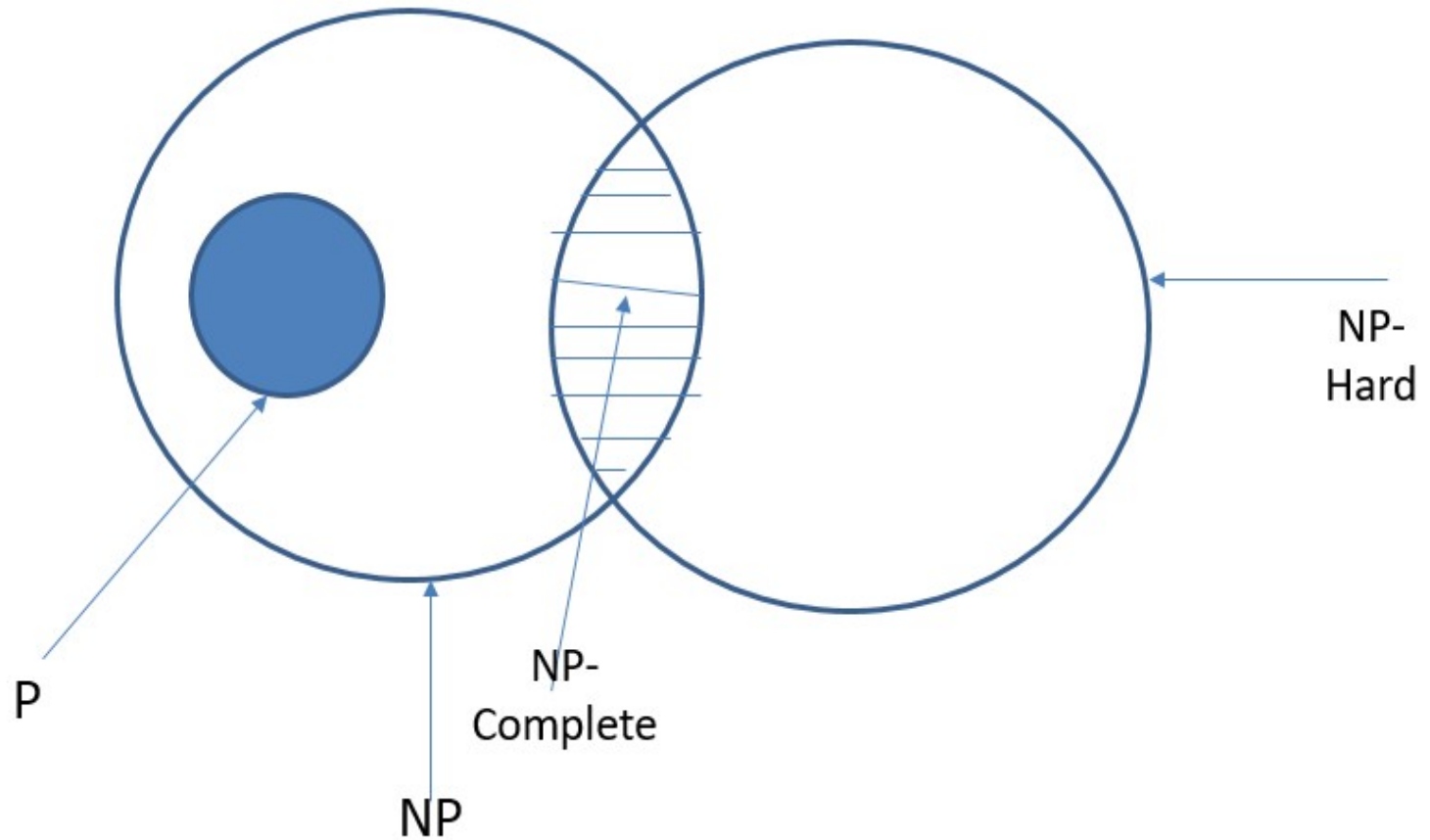
Reductions



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

Relation-ship of P, NP, NP-hard & NP-Complete

Reference Books: Cormen



Definition of NP-Complete

- Q is an NP-Complete problem if:
 - 1) Q is in NP
 - 2) every other NP problem polynomial time reducible to Q

Example

- SAT is NP-Complete (already Proved using **Cook-Levin Theorem**)

3-SAT

- $3\text{-SAT} = \{ f \mid f \text{ is in Conjunctive Normal Form, each clause has exactly 3 literals and } f \text{ is satisfiable} \}$
- 3-SAT is NP-Complete
- (2-SAT is in P)

NP-Complete

- To prove a problem is NP-Complete show a polynomial time reduction from 3-SAT
- Other NP-Complete Problems:
 - CLIQUE
 - VERTEX COVER
 - HAMILTONIAN PATH (TSP)
 - GRAPH COLORING
 - PARTITION
 - SUBSET-SUM
 - MINESWEEPER (and many more)

NP-Completeness Proof Method

- To show that Q is NP-Complete:
 - 1) Show that Q is in NP
 - 2) Pick an instance, R, of your favorite NP-Complete problem (ex: Φ in 3-SAT)
 - 3) Show a polynomial algorithm to transform R into an instance of Q

The SAT Problem

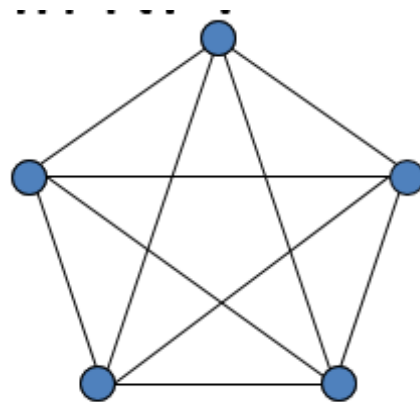
- One of the first problems to be proved NP-Complete was satisfiability (SAT):
- Given a Boolean expression on n variables, can we assign values such that the expression is TRUE?
 - Ex: $((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$
- Cook's Theorem: The satisfiability problem is NP-Complete

3-CNF: Clique

- What is a clique of a graph G ?
- A: a subset of vertices fully connected to each other, i.e. a complete subgraph of G
- The clique problem: how large is the maximum-size clique in a graph?
- Can we turn this into a decision problem?
- A: Yes, we call this the k -clique problem
- Is the k -clique problem within NP?

Example: Clique

- $\text{CLIQUE} = \{ \langle G, k \rangle \mid G \text{ is a graph with a clique of size } k \}$
- A clique is a subset of vertices that are all connected
- Why is CLIQUE in NP?



CLIQUE is NP-complete

- Proof:
- CLIQUE \in NP: given $G=(V,E)$ and a set $V' \subseteq V$ as a certificate for G . The verifying algorithm checks for each pair of $u,v \in V'$, whether $\langle u,v \rangle \in E$. time: $O(|V'|^2|E|)$.
- CLIQUE is NP-hard:
- show 3-CNF-SAT \leq_p CLIQUE.
- The result is surprising, since from Boolean formula to graph.

CLIQUE is NP-complete

Reduction from 3-CNF-SAT to CLIQUE.

Suppose $\phi = C_1 \wedge C_2 \dots \wedge C_k$ be a boolean formula in 3-CNF with k clauses.

We construct a graph $G=(V,E)$ as follows:

For each clause $C_r = (l_{1r} \vee l_{2r} \vee l_{3r})$, place a triple of v_{1r}, v_{2r}, v_{3r} into V

Put the edge between two vertices v_{ir} and v_{js} when:

$r \neq s$, that is v_{ir} and v_{js} are in different triples, and

Their corresponding literals are consistent, i.e, l_{ir} is not negation of l_{js} .

Then ϕ is satisfiable if and only if G has a clique of size k .

CLIQUE is NP-complete

Prove the above reduction is correct:

If ϕ is satisfiable, then there exists a satisfying assignment, which makes at least one literal in each clause to evaluate to 1.

Pick one this kind of literal in each clause. Then consider the subgraph V' consisting of the corresponding vertex of each such literal. For each pair $v_{ir}, v_{js} \in V'$, where $r \neq s$. Since l_{ir}, l_{js} are both evaluated to 1, so l_{ir} is not negation of l_{js} , thus there is an edge between v_{ir} and v_{js} . So V' is a clique of size k .

If G has a clique V' of size k , then V' contains exact one vertex from each triple. Assign all the literals corresponding to the vertices in V' to 1, and other literals to 1 or 0, then each clause will be evaluated to 1. So ϕ is satisfiable.

It is easy to see the reduction is in poly time.

The reduction of an instance of one problem to a specific instance of the other problem

$\phi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$ and its reduced graph G

Reference Books: Cormen

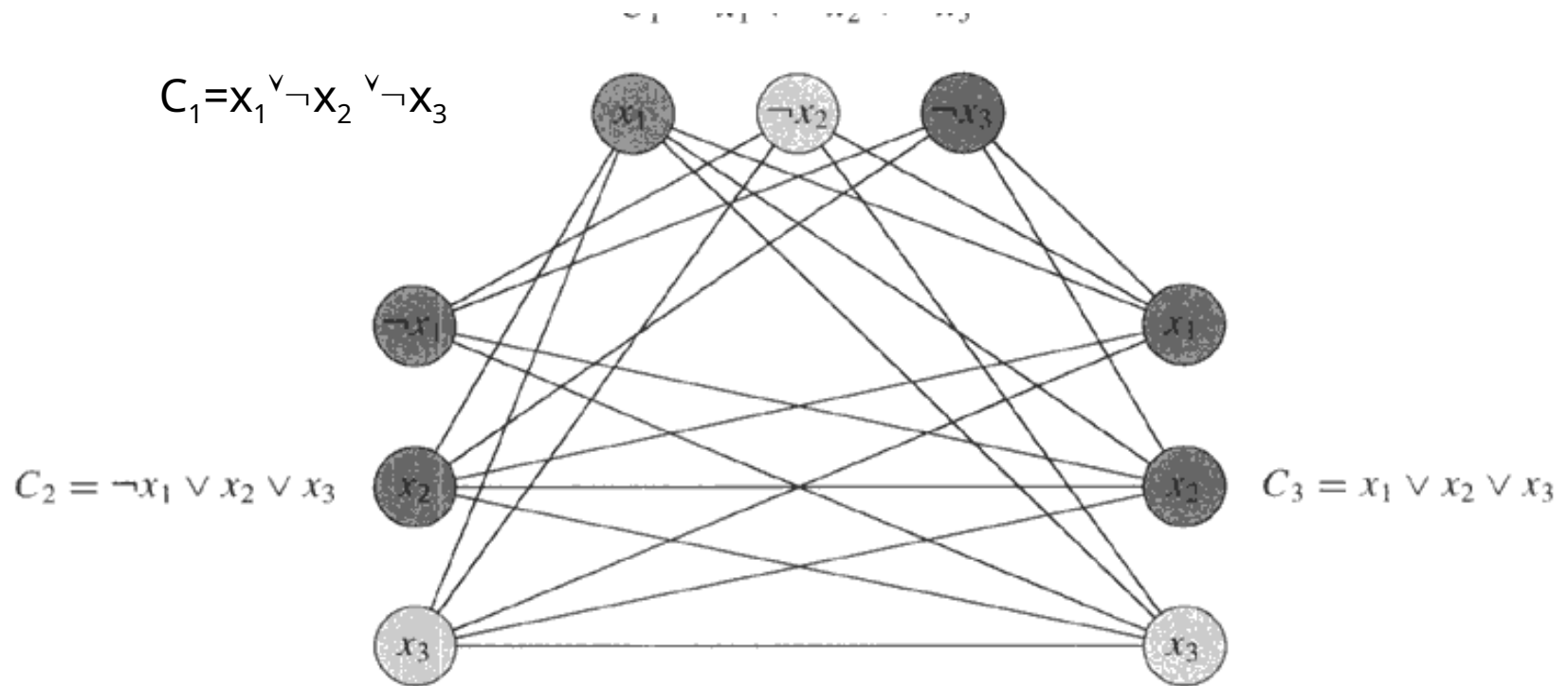


Figure 34.14 The graph G derived from the 3-CNF formula $\phi = C_1 \wedge C_2 \wedge C_3$, where $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$, $C_2 = (\neg x_1 \vee x_2 \vee x_3)$, and $C_3 = (x_1 \vee x_2 \vee x_3)$, in reducing 3-CNF-SAT to CLIQUE. A satisfying assignment of the formula has $x_2 = 0$, $x_3 = 1$, and x_1 may be either 0 or 1. This assignment satisfies C_1 with $\neg x_2$, and it satisfies C_2 and C_3 with x_3 , corresponding to the clique with lightly shaded vertices.

Vertex Cover

- What is a vertex-cover?
- Given a undirected graph $G=(V, E)$, **vertex-cover** V' :
 - $V' \subseteq V$
 - for each edge (u, v) in E , either $u \in V'$ or $v \in V'$ (or both)
- The size of a vertex-cover is $|V'|$

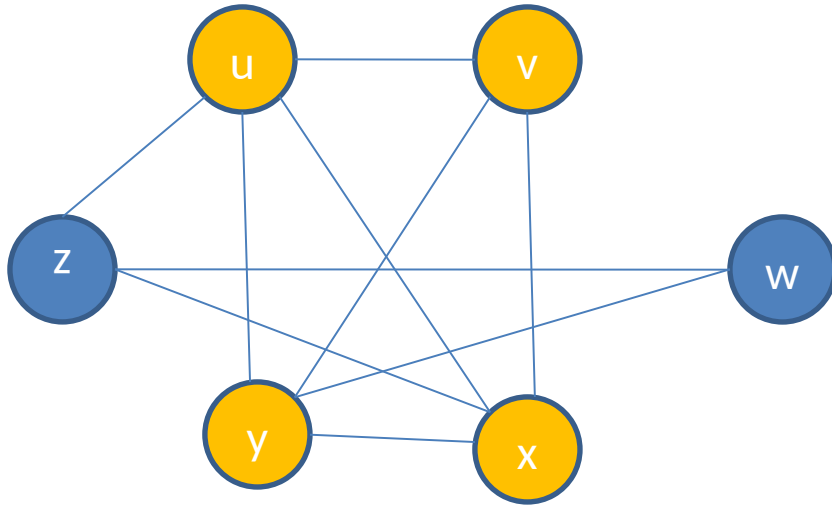
Clique \rightarrow Vertex Cover

- A vertex cover for a graph G is a set of vertices incident to every edge in G
- The vertex cover problem: what is the minimum size vertex cover in G ?
- Restated as a decision problem: does a vertex cover of size k exist in G ?
- Vertex cover is NP-Complete

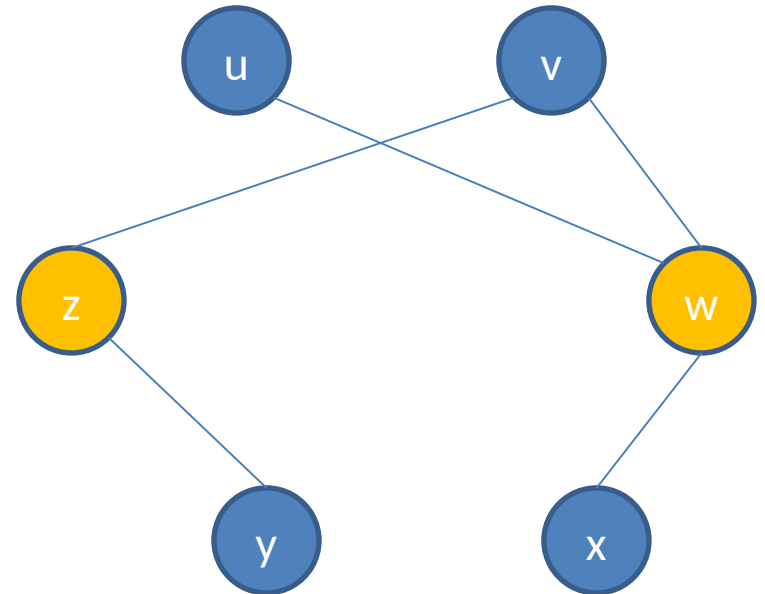
Clique \rightarrow Vertex Cover

- First, show vertex cover in NP (How?)
- The certificate we choose is the vertex cover V' which is subset of V itself. Check
- $|V'| = k$ and then for each edge $\langle u, v \rangle \in E$,
- $u \in V'$ or $v \in V'$

- Next, reduce k-clique to vertex cover
 - Reduction algorithm takes as input an instance $\langle G, k \rangle$ of the clique problem.
 - It computes complement of G as GC in polynomial time
 - The complement GC of a graph G contains exactly those edges not in G
 - G has a clique of size k iff GC has a vertex cover of size $|V| - k$



Undirected graph with
Clique = $V' = \{u, v, x, y\}$



Graph G_c produced by
reduction algorithm that has
vertex cover
 $V - V' = \{$

Clique \rightarrow Vertex Cover

- Claim: If G has a clique of size k , GC has a vertex cover of size $|V| - k$
 - Let V' be the k -clique
 - Then $V - V'$ is a vertex cover in GC
 - Let (u,v) be any edge in GC
 - Then u and v cannot both be in V' (Why?)
 - Thus at least one of u or v is in $V - V'$ (why?), so edge (u, v) is covered by $V - V'$
 - Since true for any edge in GC , $V - V'$ is a vertex cover

Clique \rightarrow Vertex Cover

- Claim: If GC has a vertex cover $V' \subseteq V$, with $|V'| = |V| - k$, then G has a clique of size k
 - For all $u, v \in V$, if $(u, v) \in GC$ then $u \in V'$ or $v \in V'$ or both (Why?)
 - Contrapositive: if $u \notin V'$ and $v \notin V'$, then $(u, v) \in E$
 - In other words, all vertices in $V - V'$ are connected by an edge, thus $V - V'$ is a clique
 - Since $|V| - |V'| = k$, the size of the clique is k

Approximation Algorithms

Introduction

- Approximation algorithms are used to get a solution close to the (optimal) solution of an optimization problem in polynomial time
- For example, Traveling Salesman Problem (TSP) is an optimization problem (the route taken has to have minimum cost).
- Since TSP is NP-Complete, approximation algorithms allow for getting a solution close to the solution of an NP problem in polynomial time.

Definition

- An algorithm is an *α -approximation algorithm* for an optimization problem Π if
 1. The algorithm runs in polynomial time
 2. The algorithm always produces a solution that is within a factor of α of the optimal solution
- Minimization – $\alpha < 1$
- Maximization – $\alpha > 1$

Set Cover (SC)

- A set cover of a set T is any collection of subsets of T whose union is T .
- The set cover problem: given a weight for each subset, find the set cover which minimizes the total weight

Example of Vertex Cover

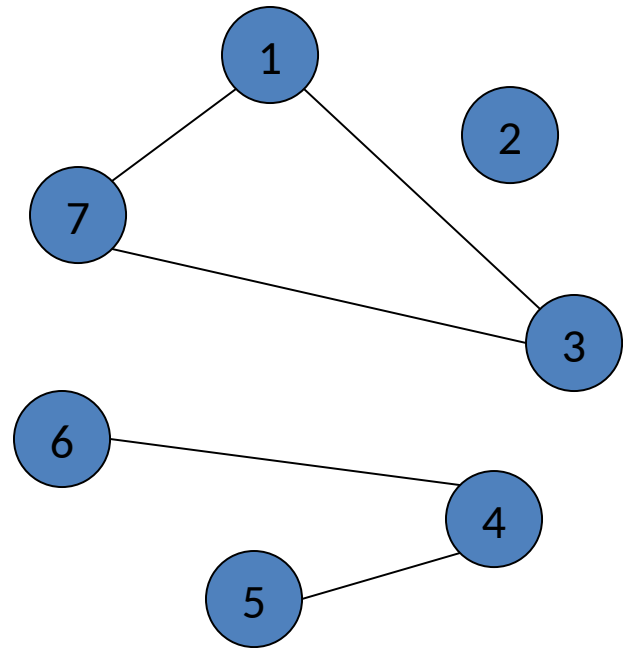
- Possible Solutions are:

1. $\{ 1, 3, 4 \}$

2. $\{ 1, 7, 5 \}$

- If the weights are the vertex numbers, then the optimal solution is: $\{1, 3, 4\}$

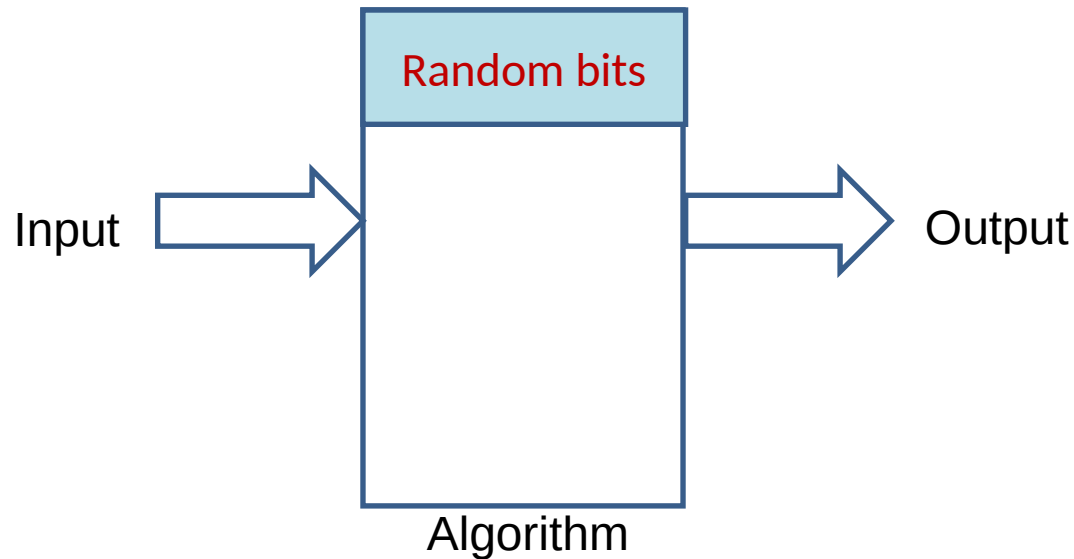
The Graph



Vertex Cover as Set Cover

- Vertex Cover is a special case of Set Cover
- To convert a problem from vertex cover to set cover:
 1. Make T = the set of edges E
 2. Make the subsets correspond to each vertex, with each subset containing the set of edges that touch the corresponding vertex

Randomized Algorithm



- The output or the running time are functions of the input and random bits chosen .

Randomized QuickSort(S)

When the input S is stored in an array A

QuickSort(A, l, r)

{ If ($l < r$)

$x \leftarrow$ an element selected **randomly** uniformly from $A[l..r]$;

$i \leftarrow$ Partition(A, l, r, x);

 QuickSort($A, l, i - 1$);

 QuickSort($A, i + 1, r$)

}

- **Distribution** insensitive: Time taken does not depend on initial permutation of A .
- Time taken **depends** upon the **random** choices of pivot elements.
 1. For a given input, Expected(**average**) running time: $O(n \log n)$
 2. **Worst** case running time: $O(n^2)$

Types of Randomized Algorithms

- Randomized **Las Vegas** Algorithms:

- Output is always correct
- Running time is a random variable
- Example: Randomized Quick Sort

- Randomized **Monte Carlo** Algorithms:

- Output may be incorrect with some probability
- Running time is deterministic.
- Example: Randomized algorithm for approximate median

Motivation for randomized Algorithms

“Randomized algorithm for a problem is usually **simpler** and **more efficient** than its deterministic counterpart.”

Example 1: Sorting

Deterministic Algorithms:

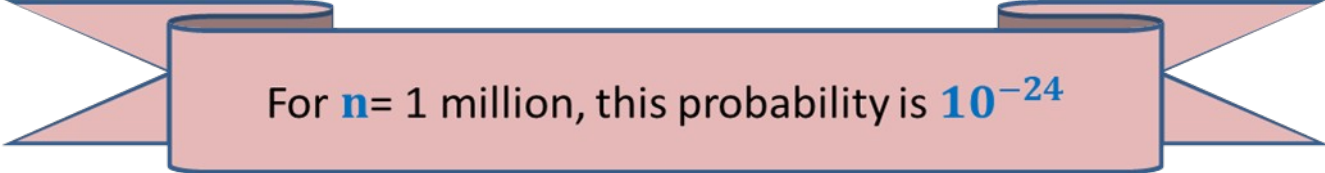
- Heap sort
- Merge Sort

Randomized Las Vegas algorithm:

- Randomized Quick sort

Randomized Quick sort almost always outperforms heap sort and merge sort.

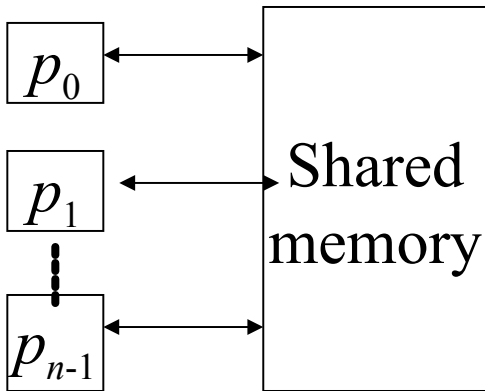
Probability[running time of quick sort exceeds twice its expected time] $< n^{-\log \log n}$



For $n = 1$ million, this probability is 10^{-24}

Parallel Algorithm

- Parallel: perform more than one operation at a time.
- PRAM model: Parallel Random Access Model.



- Multiple processors connected to a shared memory.
- Each processor access any location in unit time.
- All processors can access memory in parallel.
- All processors can perform operations in parallel.

Concurrent vs. Exclusive Access

- Four models
 - EREW: exclusive read and exclusive write
 - CREW: concurrent read and exclusive write
 - ERCW: exclusive read and concurrent write
 - CRCW: concurrent read and concurrent write
- Handling write conflicts
 - Common-write model: only if they write the same value.
 - Arbitrary-write model: an arbitrary one succeeds.
 - Priority-write model: the one with smallest index succeeds.
- EREW and CRCW are most popular.

Synchronization and Control

- Synchronization:
 - A most important and complicated issue
 - Suppose all processors are inherently tightly synchronized:
 - All processors execute the same statements at the same time
 - No race among processors, i.e, same pace.
- Termination control of a parallel loop:
 - Depend on the state of all processors
 - Can be tested in $O(1)$ time.

Parallel Computation

- To describe parallel algorithm , use statements of general form as :
for $x \in S$ in parallel do statement(x);
Eg.
For $1 \leq i \leq 10$ in parallel do statement(i);

Parallel Computation

- Parallel algorithms are designed to improve the computation speed of a computer. For analyzing a Parallel Algorithm, we normally consider the following parameters –
 - Time complexity (Execution Time),
 - Total number of processors used, and
 - Total cost.

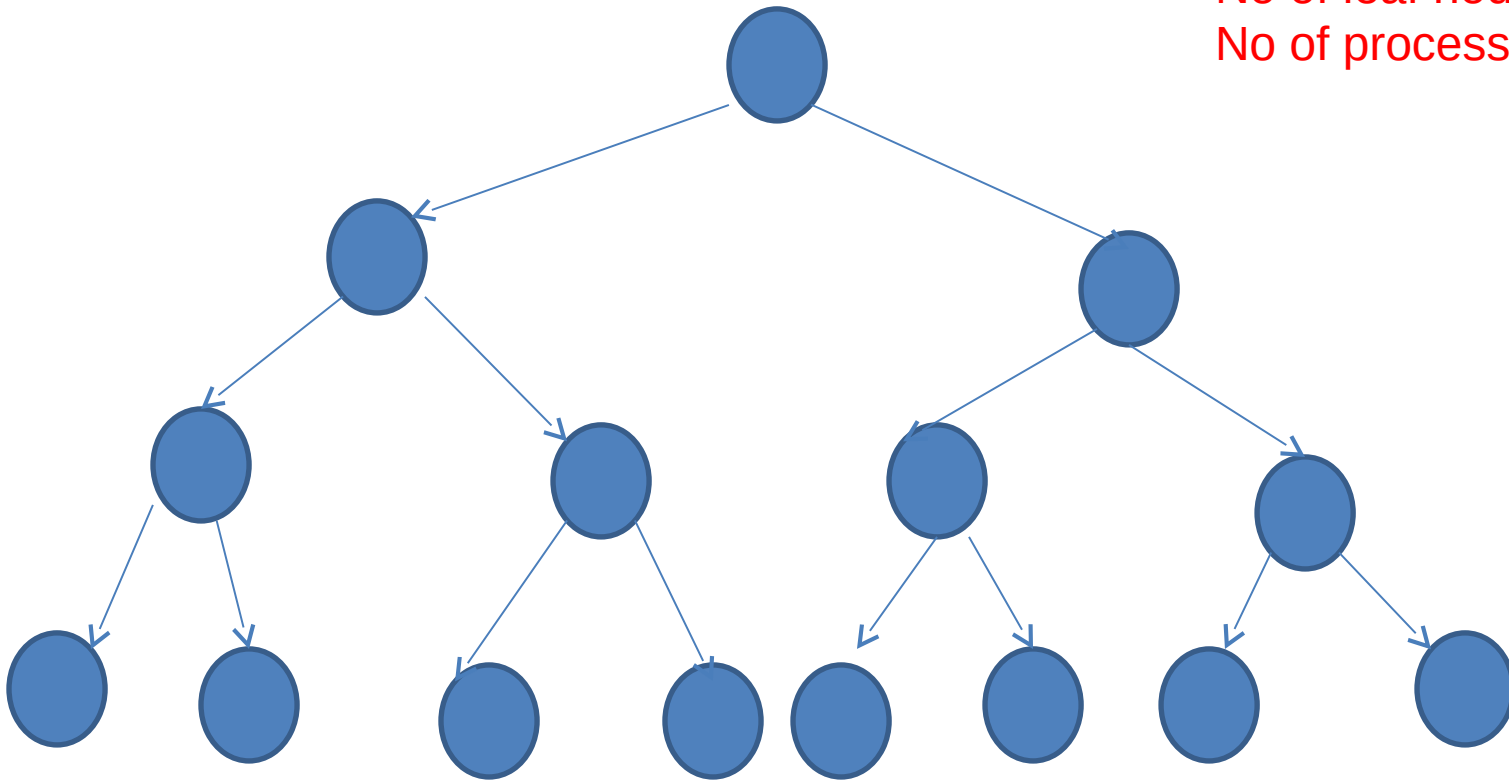
Parallel Computation

- Total cost of a parallel algorithm is the product of time complexity and the number of processors used in that particular algorithm.
- Total Cost = Time complexity \times Number of processors used
- Therefore, the **efficiency** of a parallel algorithm is –
- Efficiency = Worst case execution time of sequential algorithm / Worst case execution time of the parallel algorithm

Basic Techniques and algorithms

- Complete Binary Tree

No of leaf nodes=08
No of processors= $2^n - 1$



Example

- Sum of n numbers is calculated using binary tree

Algorithm:

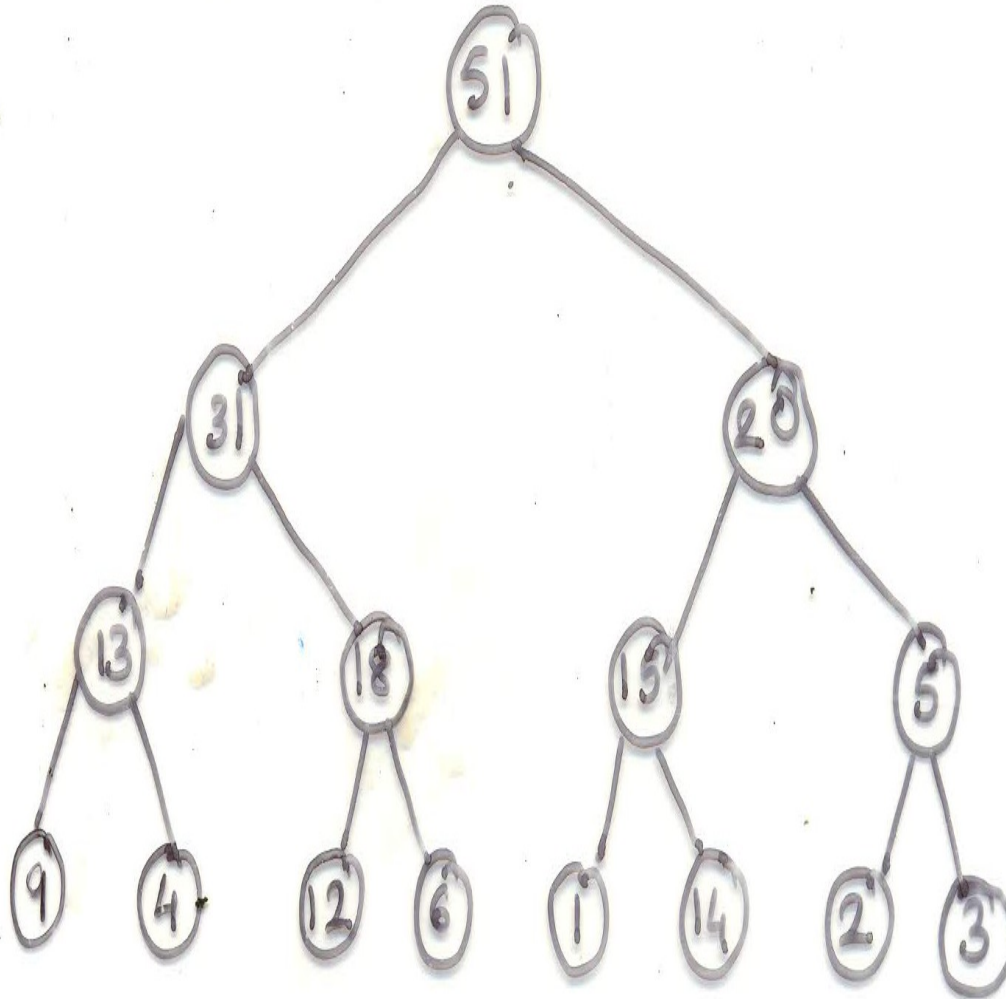
1. Each processor at leaf sends the number to its parent to start with.
2. On receipt of two numbers, an internal processor finds sum and send result to father
3. The process stops when root processor calculates sum.

Example

Step 3

Step 2

Step 1



$(\lg n)$ steps

$O(\log n)$

Algorithm

```
function parsum (T, n)
{
    Calculates the sum  $T[n] + \dots + T[2n-1]$ 
    for  $i \leftarrow \lg n - 1$  down to 0 do
        for  $2^i \leq j \leq 2^{i+1} - 1$  in parallel do
             $T[j] \leftarrow T[2j] + T[2j+1]$ 
    return  $T[1]$ .
```

→ The only syncⁿ required is that all the parallel computations for a particular value of i (one phase) should be completed before computations for next value of i .