ITDO 5014 ADSA SYLLABUS

Module 6 : Advanced Algorithms and NP problems : (6 Hrs)

1. Optimization Algorithms: Genetic algorithm (GA),
2. Approximation Algorithms: Vertex-cover problem,
3. Parallel Computing Algorithms: Fast Fourier Transform,
4. Introduction to NP-Hard and NP-Complete Problems

Self-learning Topics: Implementation of Genetic algorithm and Vertex-cover problem.

4. NP-Hard and NP-Complete :

Polynomial Time	Exponential Time
Linear search -- n	Satisfiability -- 2^n
Binary Search -- log n	Travelling Sales Person -- 2^n
Insertion Sort – n^2	Sum of Subsets -- 2^n
Merge Sort --- n log n	Graph Colouring -- 2^n
MergeSort – n log n	Hamiltonian Cycle – 2^n
Matrix Multiplication – n^3	0/1 Knapksack -- 2^n

Exponential time is very very large compared to Polynomial time .
So, we should be able to convert exponential time algorithms into polynomial time algorithms.

If we are not able to find polynomial time algorithms for above exponential time algorithms, then, at least, we try to show the similarity among all above exponential problems so that , if one of these problems is solved , then all other problems will also be solved.

In order to do this ,

1. we must find the relationship that exists among all of these exponential time problems or the properties of these exponential time algorithms.

2. If we are not able to find polynomial time deterministic algorithms for these problems, then try to find polynomial time NON- Deterministic algorithms for them.

Deterministic Algorithms (P = Polynomial time algorithms) :

Behavior of the algorithm is known. Eg: Algorithms that we studied using D&C and Greedy such as

Binary search (log n), quick sort, merge sort, Fractional knapsack, Optimal merge pattern (n log n), single source shortest path, Minimum cost Spanning Tree, etc.

Non Deterministic Algorithms (NP = Non Deterministic Polynomial time algorithms) :

Behavior of the algorithm is not known. i.e.

some of the statements of the algorithm are deterministic and others are non-deterministic.

If we are able to find polynomial time algorithms for above exponential time algorithms,

=> NP problem is converted into P problem.

NP-Completeness :

We have seen efficient algorithms to solve complex problems, like shortest path, Euler graph, minimum spanning tree, etc.

Can all computational problems be solved by a computer?

There are computational problems that can not be solved by algorithms even with unlimited time.

For example Turing Halting problem (Given a program and an input, whether the program will eventually halt when run with that input, or will run forever).

Alan Turing proved that general algorithm to solve the halting problem for all possible program-input pairs cannot exist.

A key part of the proof is, Turing machine was used as a mathematical definition of a computer and program.

NP complete problems are problems whose status is unknown.

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

However, if any one of the NP complete problems can be solved in polynomial time, then all of them can be solved.

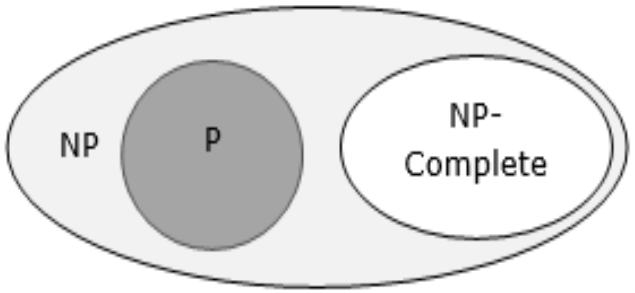
What are NP, P, NP-complete and NP-Hard problems?

P is set of problems that can be solved by a deterministic Turing machine in Polynomial time.

NP is set of decision problems that can be solved by a Non-deterministic Turing Machine in Polynomial time.

P is subset of NP (any problem that can be solved by deterministic machine in polynomial time can also be solved by non-deterministic machine in polynomial time).

Informally, NP is set of decision problems which can be solved in polynomial time via a "Lucky Algorithm", a magical algorithm that always makes a right guess among the given set of choices (Source Ref 1).



NP-complete problems are the hardest problems in NP set.

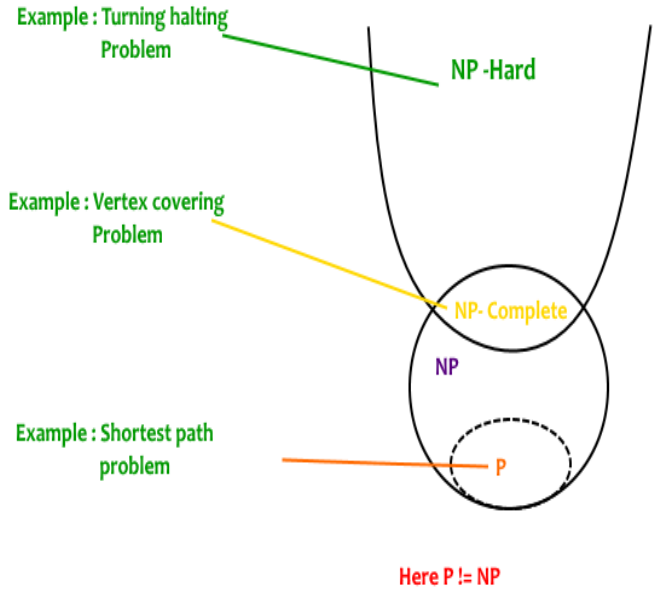
A decision problem L is NP-complete if:

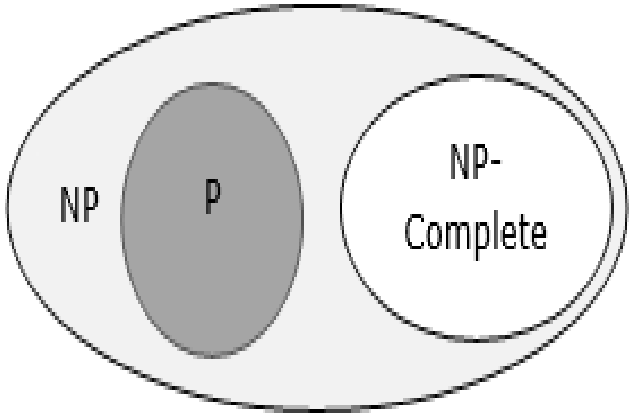
1) L is in NP (Any given solution for NP-complete problems can be verified quickly, but there is no efficient known solution).

2) Every problem in NP is reducible to L in polynomial time (Reduction is defined below).

A problem is NP-Hard if it follows property 2 mentioned above, doesn't need to follow property 1.

Therefore, NP-Complete set is also a subset of NP-Hard set.



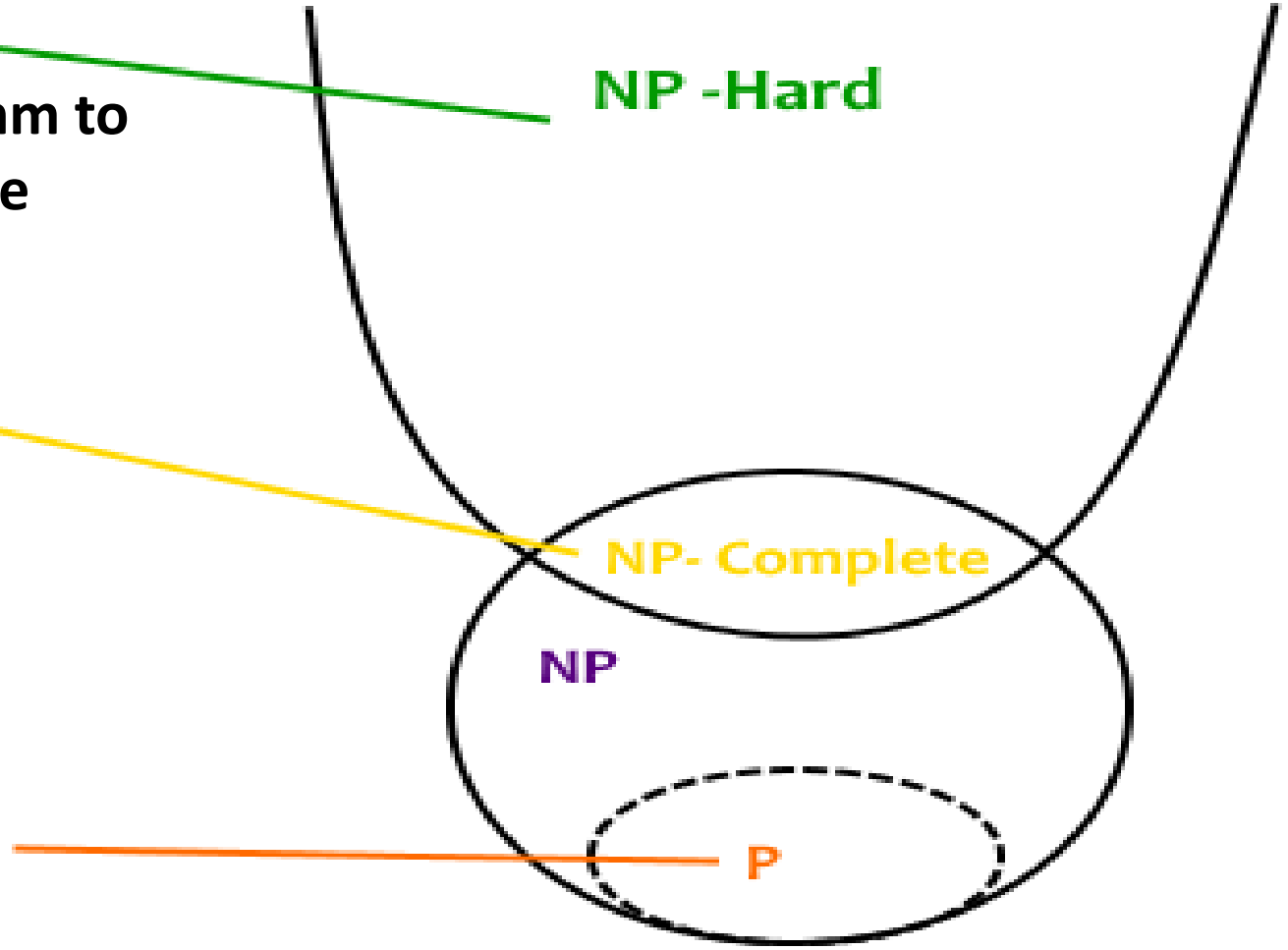


Example : Turning halting Problem

Alan Turing proved that general algorithm to solve the halting problem for all possible program-input pairs cannot exist.

Example : Vertex covering Problem

Example : Shortest path problem



Here P != NP

NP-Complete Problems :

Following are some NP-Complete problems, for which no polynomial time algorithm is known.

- **Determining whether a graph has a Hamiltonian cycle**
- **Determining whether a Boolean formula is satisfiable, etc.**

NP-Hard Problems :

The following problems are NP-Hard

- **The circuit-satisfiability problem**
- **Set Cover**
- **Vertex Cover**
- **Travelling Salesman Problem**

Satisfiability Problem is 2^n ➔ with 3 variables a,b,c
Formula = (a or b-bar or c) ^ (a-bar or b or c-bar)
3 variables ➔ 8 = 2^3 possible combinations

000
001
010

111

In general, for Boolean formula with n variables, No of combinations = 2^n = Exponential Time

Decision vs Optimization Problems :

NP-completeness applies to the realm of decision problems.
It was set up this way because it's easier to compare the difficulty of decision problems than that of optimization problems.
In reality, though, being able to solve a decision problem in polynomial time will often permit us to solve the corresponding optimization problem in polynomial time (using a polynomial number of calls to the decision problem).

So, discussing the difficulty of decision problems is often really equivalent to discussing the difficulty of optimization problems. (Source Ref 2).

For example, consider the vertex cover problem (Given a graph, find out the minimum sized vertex set that covers all edges).
It is an optimization problem.

Corresponding decision problem is, given undirected graph G and k, is there a vertex cover of size k?

How to prove that a given problem is NP complete?

From the definition of NP-complete, it appears impossible to prove that a problem L is NP-Complete.

By definition, it requires us to that show every problem in NP is polynomial time reducible to L. Fortunately, there is an alternate way to prove it.

The idea is to take a known NP-Complete problem and reduce it to L.

If polynomial time reduction is possible, we can prove that L is NP-Complete by transitivity of reduction (If a NP-Complete problem is reducible to L in polynomial time, then all problems are reducible to L in polynomial time).

References:

1. MIT Video Lecture on Computational Complexity
2. Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest
3. http://www.ics.uci.edu/~eppstein/161/960312.html
4. **www.geeksforgeeks.org/np-completeness-set-1/**

ITDO 5014 ADSA SYLLABUS

Module 6 : Advanced Algorithms and NP problems : (6 Hrs)

1. **Optimization Algorithms: Genetic algorithm (GA),**
2. **Approximation Algorithms: Vertex-cover problem,**
3. **Parallel Computing Algorithms: Fast Fourier Transform,**
4. **Introduction to NP-Hard and NP-Complete Problems**

Self-learning Topics: Implementation of Genetic algorithm and Vertex-cover problem.

1. Genetic Algorithms

1. Genetic Algorithms :

Genetic algorithms (GAs) are used mainly for optimization problems for which the exact algorithms are of very low efficiency.

GAs search for good solutions to a problem from among a (large) number of possible solutions.

The current set of possible solutions is used to generate a new set of possible solution.

They start with an initial set of possible solutions, and at each step they do the following:

Evaluate the current set of solutions (current generation).

Choose the best of them to serve as “parents” for the new generation, and construct the new generation.

The loop runs until :

-a specified condition becomes true, e.g. a solution is found that satisfies the criteria for a “good” solution, or

-the number of iterations exceeds a given value, or

-no improvement has been recorded when evaluating the solutions.

Key issues here are:

How large to be the size of a population so that there is sufficient diversity?

Usually the size is determined through trial-and-error experiments.

How to represent the solutions so that the representations can be manipulated to obtain a new solution?

One approach is to represent the solutions as strings of characters (could be binary strings) and to use various types of "crossover" (explained below) to obtain a new set of solutions.

The strings are usually called "chromosomes"

how to evaluate a solution?.

The function used to evaluate a solution is called “fitness function” and it depends on the nature of the problem.

How to manipulate the representations in order to construct a new solution?

The method that is characteristic of GAs is to combine two or more representations by taking substrings of each of them to construct a new solution.

This operation is called “crossover”.

How to choose the individual solutions that will serve as parents for the new generation?

The process of choosing parents is called “selection”.

Various methods have been experimented here.

It seems that the choice is dependent on the nature of the problem and the chosen representation.

One method is to choose parents with a probability proportional to their fitness.

This method is called “roulette wheel selection”.

How to avoid convergence to a set of equal solutions?

The approach here is to change randomly the representation of a solution.
If the representation is a bit string we can flip bits. This operation is called “mutation”.

Using the terminology above, we can outline the algorithms to obtain one generation:

compute the fitness of each chromosome
select parents based on the fitness value and a selection strategy
perform crossover to obtain new chromosomes
perform mutation on the new chromosomes (with fixed or random probability)

Examples:

The knapsack problem solved with GAs
The traveling salesman problem

ITDO 5014 ADSA SYLLABUS

Module 6 : Advanced Algorithms and NP problems : (6 Hrs)

1. Optimization Algorithms: Genetic algorithm (GA),
2. Approximation Algorithms: Vertex-cover problem,
3. Parallel Computing Algorithms: Fast Fourier Transform,
4. Introduction to NP-Hard and NP-Complete Problems

Self-learning Topics: Implementation of Genetic algorithm and Vertex-cover problem.

2. Approximation Algorithms: Vertex-cover problem :

Approximate Algorithms :

There are several optimization problems such as Minimum Spanning Tree (MST), Min-Cut etc. which can be solved exactly and efficiently in polynomial time.

However, there are many optimization problems that are NP-Hard, for which we are unlikely to find an algorithm that solves the problem exactly in polynomial time.

Examples of the standard NP-Hard problems are :

- **Traveling Salesman Problem (TSP) - finding a minimum cost tour of all cities**
- **Vertex Cover - find minimum set of vertex that covers all the edges in the graph**
- **Max Clique**
- **Set Cover - find a smallest size cover set that covers every vertex**
- **Shortest Superstring - given a set of string, find a smallest subset of strings that contain specified words**

These are NP-Hard problems, i.e., If we could solve any of these problems in polynomial time, then P = NP.

As of now, there is no known polynomial time exact algorithm for NP-Hard problems.

However, it may be possible to find a near-optimal solution for NP-Hard problems in polynomial time.

An algorithm that runs in polynomial time and outputs a solution that is close to the optimal solution is called as an approximation algorithm or heuristic algorithms.

Thus, an Approximate Algorithm technique does not guarantee the best solution.

The goal of an approximation algorithm is to come as close as possible to the optimum value in a reasonable amount of time i.e. polynomial time.

Examples :
For the traveling salesperson problem,
the optimization problem is to find the shortest cycle, and
the approximation problem is to find a short cycle.

For the vertex cover problem,
the optimization problem is to find the vertex cover with fewest vertices, and
the approximation problem is to find the vertex cover with few vertices.

[[Performance Ratios
Suppose we work on an optimization problem where every solution carries a cost.
An Approximate Algorithm returns a legal solution, but the cost of that legal solution may not be optimal.

For Example, suppose we are considering for a minimum size vertex-cover (VC).
An approximate algorithm returns a VC for us, but the size (cost) may not be minimized.

Another Example is we are considering for a maximum size Independent set (IS).
An approximate Algorithm returns an IS for us, but the size (cost) may not be maximum.
Let C be the cost of the solution returned by an approximate algorithm, and C* is the cost of the optimal solution.

We say the approximate algorithm has an approximate ratio P (n) for an input size n, where $\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq P(n)$

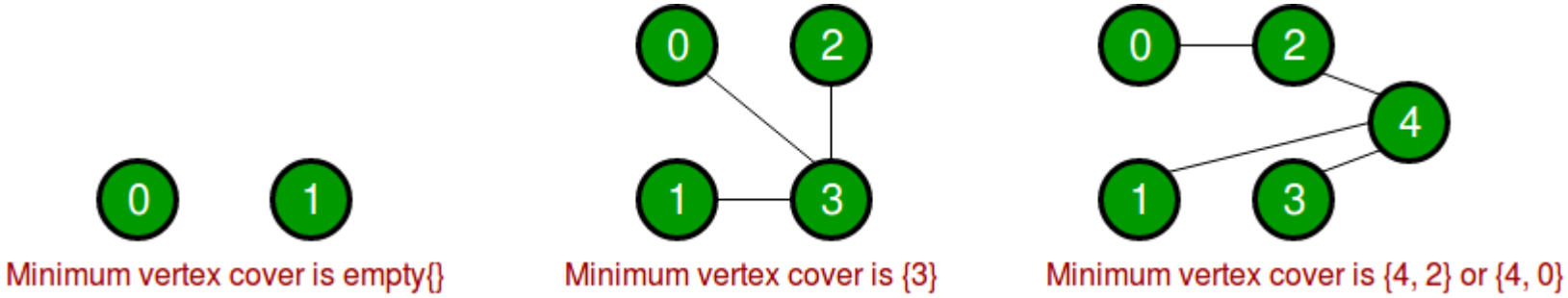
Approximate Algorithm
Intuitively, the approximation ratio measures how bad the approximate solution is distinguished with the optimal solution.
A large approximation ratio measures the solution as much worse than an optimal solution.
A small approximation ratio measures the solution as more or less the same as an optimal solution.

P (n) is always ≥ 1, if the ratio does not depend on n, we may write P.
Therefore, a 1-approximation algorithm gives an optimal solution.
Some problems have polynomial-time approximation algorithm with small constant approximate ratios, while others have best-known polynomial time approximation algorithms whose approximate ratios grow with n.]]

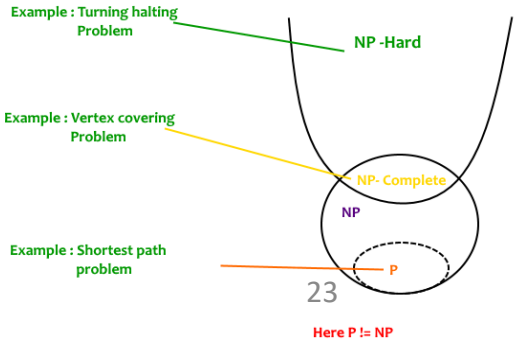
Approximate Algorithm – Eg. Vertex Cover Problem :

A vertex cover of an undirected graph is a subset of its vertices such that for every edge (u, v) of the graph, either ‘u’ or ‘v’ is in the vertex cover.

Although the name is Vertex Cover, the set covers all edges of the given graph.
Given an undirected graph, the vertex cover problem is to find minimum size vertex cover.
For example:



<u>Vertex Cover Problem</u> is a known <u>NP Complete problem</u>, i.e., there is no polynomial-time solution for this unless P = NP.
There are approximate polynomial-time algorithms to solve the problem.



An approximate algorithm for Vertex Cover Problem : [Ref: Cormen]

Naive Approach:

Consider all the subset of vertices one by one and find out whether it covers all edges of the graph.

For eg. in a graph consisting only 3 vertices the set consisting of the combination of vertices are: {0,1,2,{0,1},{0,2},{1,2},{0,1,2}} .

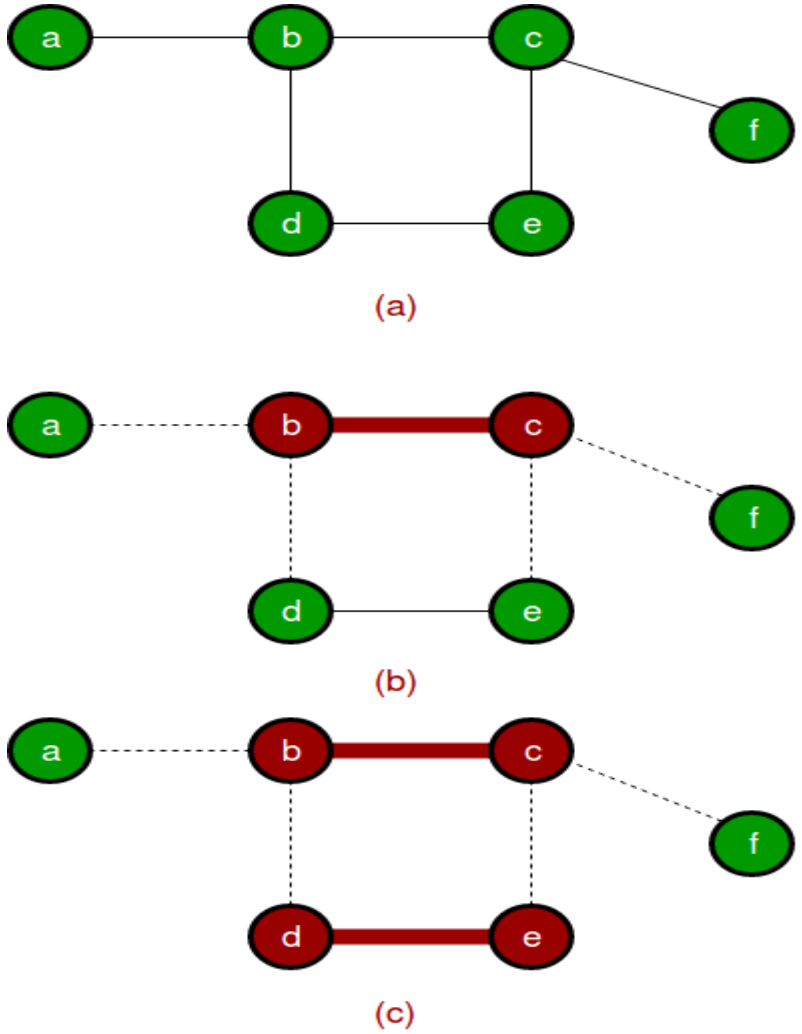
Using each element of this set check whether these vertices cover all the edges of the graph and update the optimal answer.

Then, Print the subset having minimum number of vertices which also covers all the edges of the graph.

Approximate Algorithm for Vertex Cover:

1) Initialize the result as {}
2) Consider a set of all edges in given graph.
   Let the set be E.
3) Do following while E is not empty
   a) Pick an arbitrary edge (u, v) from set E and
      add 'u' and 'v' to result
   b) Remove all edges from E which are either incident
      on u or v.
4) Return result

The diagram shows the execution of the above approximate algorithm:



Minimum Vertex Cover is {b, c, d} or {b, c, e}

How well the above algorithm perform?

It can be proved that the above approximate algorithm never finds a vertex cover whose size is more than twice the size of the minimum possible vertex cover.

The Time Complexity of the above algorithm is O(V + E).

Exact Algorithms:

Although the problem is NP complete, it can be solved in polynomial time for Bipartite Graph and Tree Graph.

The (decision) problem to check whether there is a vertex cover of size smaller than or equal to a given number k can also be solved in polynomial time if k is bounded by O(LogV)

ITDO 5014 ADSA SYLLABUS

Module 6 : Advanced Algorithms and NP problems : (6 Hrs)

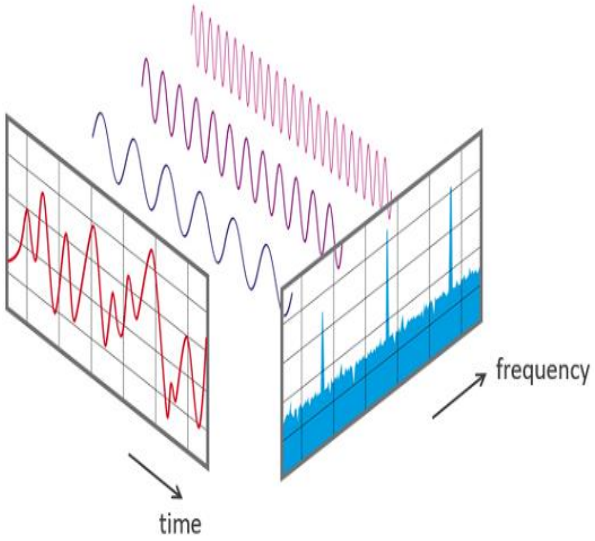
1. Optimization Algorithms: Genetic algorithm (GA),
2. Approximation Algorithms: Vertex-cover problem,
3. Parallel Computing Algorithms: Fast Fourier Transform,
4. Introduction to NP-Hard and NP-Complete Problems

Self-learning Topics: Implementation of Genetic algorithm and Vertex-cover problem.

3. Parallel Computing Algorithms - Fast Fourier Transform : [Ref: Cormen]

Fourier Transform plays an important role in signal processing, image processing, voice recognition etc.

Fourier analysis converts a signal from its original domain (often time or space) to a representation in the frequency domain and vice versa.



The Discrete Fourier Transform is a specific kind of Fourier Transform.

It maps a sequence over time to another sequence over frequency.

However, the straightforward implementation of Discrete Fourier Transform has the time complexity of $O(n^2)$ and hence, it is not a better way to be used in practice.

Alternatively, we can use the Fast Fourier Transform algorithm, which takes just O(n log n) time to perform the Discrete Fourier Transform.

The advantage of using FFT for implementation of DFT is that, it can be easily parallelized to execute even in lesser time on multiple processors/cores.

Thus, FFT is an optimized algorithm for the implementation of the "Discrete Fourier Transformation" (DFT).

A fast Fourier transform (FFT) is an algorithm that computes the discrete Fourier transform (DFT) of a sequence

3. Parallel Computing Algorithms - Fast Fourier Transform : [Ref: Cormen]

The straightforward method of adding two polynomials of degree n takes $\theta(n)$ time, but the straightforward method of multiplying them takes $\theta(n^2)$ time.

The fast Fourier transform, or FFT, can reduce the time to multiply polynomials to $\theta(n \log n)$.

Given below is a Recursive divide-and-conquer algorithm for computing DFT called as Fast Fourier Transform, or FFT.

FFT is a particular way of computing DFT efficiently.

Fast Fourier Transform (FFT) : A Recursive divide-and-conquer algorithm for computing DFT.

FFT is particular way of computing DFT efficiently.

```
procedure fft(x, y, n, ω)
    if n = 1 then
        y[0] = x[0]
    else
        for k = 0 to (n/2) – 1
            p[k] = x[2k]
            s[k] = x[2k + 1]
        end
    fft(p, q, n/2, ω2)
    fft(s, t, n/2, ω2)
    for k = 0 to n – 1
        y[k] = q[k mod (n/2)] + ω^k t[k mod (n/2)]
    end
end
```

There are log n levels of recursion,
each of which involves $\Theta(n)$ arithmetic operations,
so total cost is $\Theta(n \log n)$
By contrast, straightforward evaluation of matrix-vector product defining DFT requires $\Theta(n^2)$ arithmetic operations,
which is enormously greater for long sequences.

n	n log n	n^2
64	384	4096
128	896	16384
256	2048	65536
512	4608	262144
1024	10240	1048576

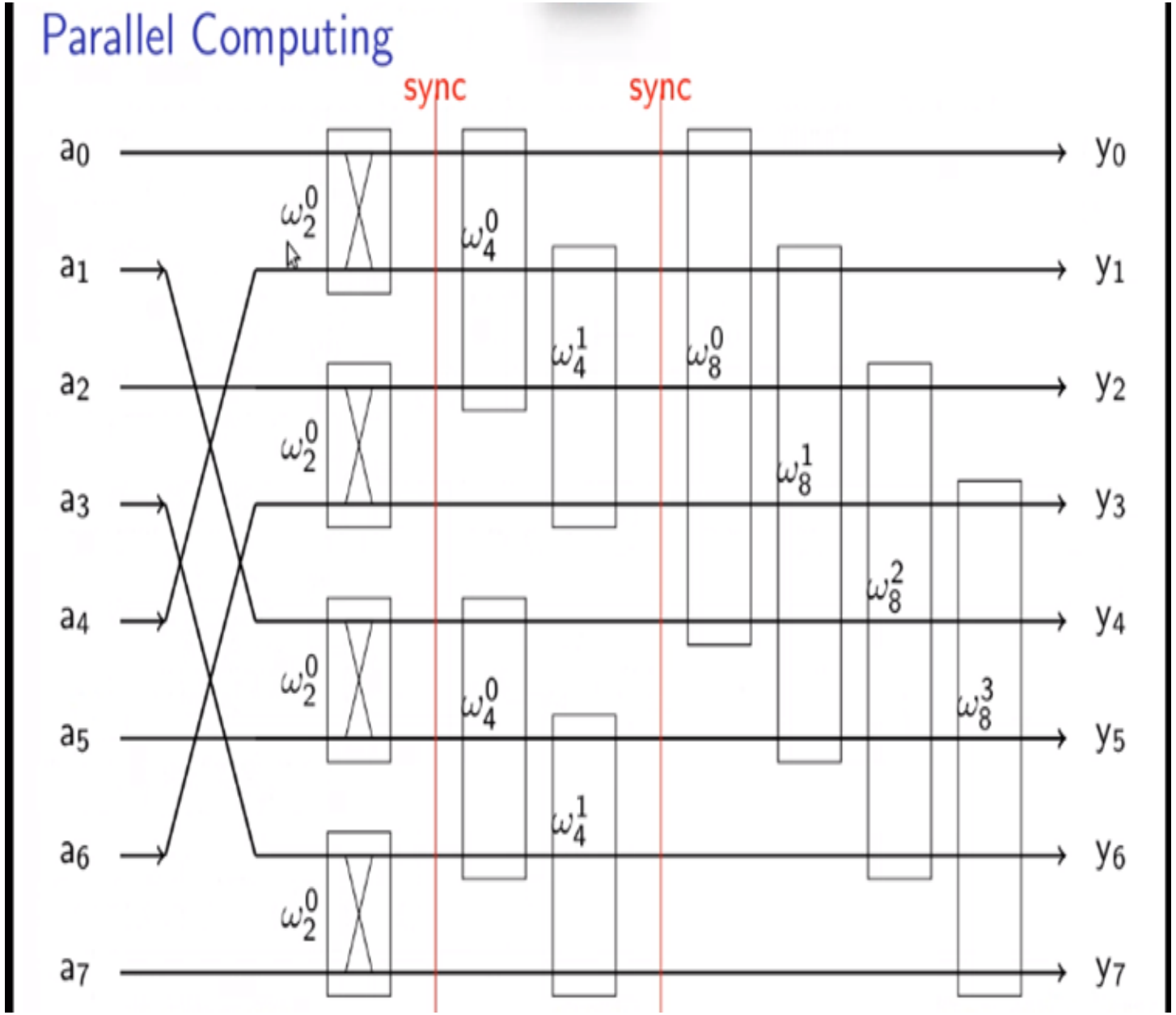
Efficient (Parallel) FFT Implementation : [Ref: Cormen]

ITERATIVE-FFT :

```
ITERATIVE-FFT(a)
 1  BIT-REVERSE-COPY(a, A)
 2  n = a.length                          // n is a power of 2
 3  for s = 1 to lg n
 4      m = 2^s
 5      ω_m = e^(2πi/m)
 6      for k = 0 to n − 1 by m
 7          ω = 1
 8          for j = 0 to m/2 − 1
 9              t = ω A[k + j + m/2]
10              u = A[k + j]
11              A[k + j] = u + t
12              A[k + j + m/2] = u − t
13              ω = ω ω_m
14  return A
```

Efficient (Parallel) FFT Implementation :

Figure 30.5 A circuit that computes the FFT in parallel, here shown on n = 8 inputs.
Each butterfly operation takes as input the values on two wires, along with a twiddle factor, and it produces as outputs the values on two wires.
The stages of butterflies are labeled to correspond to iterations of the outermost loop of the ITERATIVE-FFT procedure.
Only the top and bottom wires passing through a butterfly interact with it; wires that pass through the middle of a butterfly do not affect that butterfly, nor are their values changed by that butterfly.
For example, the top butterfly in stage 2 has nothing to do with wire 1 (the wire whose output is labeled y1); its inputs and outputs are only on wires 0 and 2 (labeled y0 and y2, respectively).
This circuit has depth Θ(lg n) and performs Θ(n lg n) butterfly operations altogether.



A parallel FFT circuit :

We can exploit many of the properties that allowed us to implement an efficient iterative FFT algorithm to produce an efficient parallel algorithm for the FFT.

We will express the parallel FFT algorithm as a circuit.

Figure 30.5 shows a parallel FFT circuit, which computes the FFT on n inputs, for n = 8.

The circuit begins with a bit-reverse permutation of the inputs, followed by lg n stages, each stage consisting of n/2 butterflies executed in parallel.

The depth of the circuit—the maximum number of computational elements between any output and any input that can reach it—is therefore $\Theta(\lg n)$.

The leftmost part of the parallel FFT circuit performs the bit-reverse permutation, and the remainder mimics the iterative ITERATIVE-FFT procedure.

Because each iteration of the outermost for loop performs n/2 independent butterfly operations, the circuit performs them in parallel.

The value of s in each iteration within ITERATIVE-FFT corresponds to a stage of butterflies shown in Figure 30.5.
For s = 1; 2; 3; 4; …… lg n, stage s consists of $n/2^s$ groups of butterflies (corresponding to each value of k in ITERATIVE-FFT), with 2^{s-1} butterflies per group (corresponding to each value of j in ITERATIVE-FFT).

The butterflies shown in Figure 30.5 correspond to the butterfly operations of the innermost loop (lines 9–12 of ITERATIVE-FFT).

Note also that the twiddle factors used in the butterflies correspond to those used in ITERATIVE-FFT: in stage s, we use

$$\omega_m^0, \omega_m^1, \ldots, \omega_m^{m/2-1}, \text{ where } m = 2^s.$$

Thank You