ALAGAPPA UNIVERSITY

[Accredited with 'A+' Grade by NAAC (CGPA:3.64) in the Third Cycle
and Graded as Category–I University by MHRD-UGC]

(A State University Established by the Government of Tamil Nadu)

KARAIKUDI – 630 003

Directorate of Distance Education

M.Sc. [Computer Science]

I - Semester

341 11

DESIGN AND ANALYSIS OF ALGORITHMS

Reviewer	
Dr. K. Kuppusamy	Professor and Head (i/c), Department of Computational Logistics, Alagappa University, Karaikudi

Authors

Dr Mudasir M Kirmani, *Assistant Professor-cum-Junior Scientist, Sher-e-Kashmir University of Sciences and Technology of Kashmir*

Dr Syed Mohsin Saif Andrabi , *Assistant Professor, Islamic University of Science & Technology, Awantipora, Jammu and Kashmir*

Units (1, 4-5, 7.2, 10.0-10.3, 11-12, 13.7)

S. Mohan Naidu, *Principal & Visiting Faculty, VRN College of Computer Science and Management, Tirupathi and Viswa Bharathi P.G. College of Engineering & Management, Hyderabad*

Unit (3)

Rohit Khurana, *CEO, ITL Education Solutions Ltd.*

Units (6, 7.3-7.4, 8.0-8.2, 9, 10.4-10.10, 13.0-13.6, 13.8-13.14, 14.3-14.8)

Sunita Tiwari, *Faculty in Computer Science & Information Technology Department at JSS Academy of Technical Education, Noida*

Shilpi Sengupta , *Lecturer of Computer Science and Engineering in JSS Academy of Technical Education, Noida*

Unit (14.0-14.2)

Vikas® Publishing House: Units (2, 7.0-7.1, 7.5-7.10, 8.2.1-8.7)

Vikas® is the registered trademark of Vikas® Publishing House Pvt. Ltd.

VIKAS® PUBLISHING HOUSE PVT. LTD.
E-28, Sector-8, Noida - 201301 (UP)
Phone: 0120-4078900 • Fax: 0120-4078999
Regd. Office: A-27, 2nd Floor, Mohan Co-operative Industrial Estate, New Delhi 1100 44
• Website: www.vikaspublishing.com • Email: helpline@vikaspublishing.com

Work Order No. AU/DDE/DE1-616/Printing of Course Materials/2019, Dated 19.11.2019 Copies - 200

SYLLABI-BOOK MAPPING TABLE

Design and Analysis of Algorithms

Syllabi	Mapping in Book
BLOCK 1: Introduction **Unit 1: Introduction:** notion of algorithm, fundamentals of algorithmic problem solving, important problem types, fundamentals of analysis of algorithm efficiency **Unit 2: Asymptotic notations:** Big-oh notation, omega notation, theta notation **Unit 3: Performance analysis:** space complexity, time complexity, pseudo code for Algorithms	**Unit 1:** Introduction to Algorithms **(Pages 1-12)** **Unit 2:** Asymptotic Notations **(Pages 13-23)** **Unit 3:** Performance Analysis **(Pages 24-34)**
BLOCK 2: Mathematical Analysis of Non Recursive Algorithms **Unit 4: Analysis of Recursive algorithms:** algorithms for computing Fibonacci Numbers **Unit 5: Empirical analysis of algorithms:** Brute force, selection sort, Bubble sort, sequential sort **Unit 6: Closest-pair and convex-hull problems:** Divide and conquer, merge sort, quick sort, Binary search, Strassens matrix multiplication	**Unit 4:** Analysis of Recursive Algorithms **(Pages 35-41)** **Unit 5:** Empirical Analysis of Algorithms **(Pages 42-49)** **Unit 6:** Closest Pair and Covex-Hull Problems **(Pages 50-67)**
BLOCK 3: Dynamic Programming and Search Binary Trees **Unit 7: General method:** computing a Binomial coefficient, warshalls and Floyds algorithms, optimal search Binary trees, knapsack problems **Unit 8: Greedy Technique:** General method **Unit 9: Applications:** prims algorithm, kruskals algorithm, dijikstras algorithm	**Unit 7:** General Method **(Pages 68-82)** **Unit 8:** Greedy Technique **(Pages 83-95)** **Unit 9:** Applications **(Pages 96-110)**
BLOCK 4: Sorting and Optimization Problem **Unit 10: Sort and Searching algorithms:** decrease and conquer, Insertion sort, Depth first search and Breadth first search, Topological sorting **Unit 11: Generating combinatorial objects:** Transform and Conquer, presorting, Heap and Heap sort **Unit 12: Optimization Problems:** Reductions, Reduction to Graph Problems	**Unit 10:** Sorting and Searching Algorithms **(Pages 111-131)** **Unit 11:** Generating Combinatorial Objects **(Pages 132-140)** **Unit 12:** Optimization Problems **(Pages 141-151)**
BLOCK 5: Backtracking and Graph Traversals **Unit 13: General method:** 8 queens problem, sum of subsets, Graph colouring, Hamiltonian cycle, Branch and Bound, assignment problem, knapsack problem, travelling salesman problems **Unit 14: Graph traversals:** connected components, spanning trees, NP hard and NP complete problems	**Unit 13:** General Method **(Pages 152-190)** **Unit 14:** Graph Traversals **(Pages 191-221)**

CONTENTS

BLOCK 2: MATHEMATICAL ANALYSIS OF NON RECURSIVE ALGORITHMS

BLOCK 3: DYNAMIC PROGRAMMING AND SEARCH BINARY TREES

BLOCK 4: SORTING AND OPTIMIZATION PROBLEM

INTRODUCTION

NOTES

An algorithm is an effective method for solving a problem using a finite sequence of instructions. Algorithms are used for calculation, data processing and many other fields. Each algorithm is a list of well-defined instructions for completing a task. Starting from an initial state, the instructions describe a computation that proceeds through a well-defined series of successive states, eventually terminating in a final ending state. The transition from one state to the next is not necessarily deterministic. Specific algorithms, known as randomized algorithms, incorporate randomness. Algorithms are essential to the way computers process information. Many computer programs contain algorithms that specify the specific instructions a computer should perform in a specific order to carry out a specified task. Thus, an algorithm can be considered to be any sequence of operations that can be simulated by a Turing complete system.

Algorithm analysis is an important part of a computational complexity theory, which provides theoretical estimates for the resources needed by any algorithm to solve a given computational problem. In theoretical analysis of algorithms, it is common to estimate their complexity in the asymptotic sense, i.e., to estimate the complexity function for arbitrarily large input. Big Oh notation, Omega notation, Theta notation, etc. are used for this. The binary search is performed in a number of steps proportional to the logarithm of the length of the list being searched or in O (log (n)), colloquially 'in logarithmic time'. Usually, asymptotic estimates are used because different implementations of the same algorithm may differ in efficiency. However, the efficiencies of any two reasonable implementations of a given algorithm are related by a constant multiplicative factor called a hidden constant.

Exact and not asymptotic measures of efficiency can sometimes be computed, but they usually require certain assumptions concerning the particular implementation of the algorithm called the model of computation. A model of computation may be defined in terms of an abstract computer, e.g., Turing machine and/or by postulating that certain operations are executed in unit time. Time efficiency estimates depend on what we define in an algorithm step. To analyse an algorithm is to determine the amount of resources, such as time and storage, necessary to execute it. Most algorithms are designed to work with inputs of arbitrary length. Usually, the efficiency or complexity of an algorithm is stated as a function relating the input length to the number of steps (time complexity) or storage locations (space complexity).

In programming language, an algorithm is a deterministic procedure that when followed yields a definite solution to a problem. It expresses steps for the solution in a way that is appropriate for computer processing, and produces a corresponding output and terminates in a fixed period of time.

This book, *Design and Analysis of Algorithms*, is aimed at providing the readers with knowledge in the concepts of algorithm and design analysis. The book follows the Self-Instruction Mode or SIM format wherein each unit begins with an 'Introduction' to the topic of the unit followed by an outline of the 'Objectives'. The detailed content is then presented in a simple and structured form interspersed with 'Check Your Progress' questions to facilitate a better understanding of the topics discussed. The 'Key Words' help the student revise what he/she has learnt. A 'Summary' along with a set of 'Self Assessment Questions and Exercises' is also provided at the end of each unit for effective recapitulation.

Self-Instructional Material

NOTES

BLOCK - I
INTRODUCTION

UNIT 1 INTRODUCTION TO ALGORITHMS

Structure

1.0 INTRODUCTION

An algorithm is a set of steps of operations to solve a problem performing calculation, data processing, and any automated reasoning tasks. It is an efficient method that can be expressed within finite amount of time and space and is the best way to represent the solution of a particular problem in a very simple and efficient manner. If we have an algorithm for a specific problem, then we can implement it in any programming language. This means that the algorithm is independent from any programming languages.

The important aspects of algorithm design include creating an efficient algorithm to solve a problem in an efficient way using minimum time and space. To solve a problem, different approaches can be followed. Some of them can be efficient with respect to time consumption, whereas other approaches may be memory efficient. In this unit, you will learn about how algorithms work.

1.1 OBJECTIVES

After going through this unit, you will be able to:

- Understand the notion of algorithm
- Discuss how algorithms work
- Analyze algorithm efficiency

1.2 NOTION OF ALGORITHM

NOTES

A computer accepts instructions from a user in order to execute the same on a machine. In order to instruct a computer to perform a task, a program is required which enables a computer to perform a specified task. These computer programs include set of steps which explain a computer what needs to be done, how to implement instructions and in case of any errors how to inform the end user about the same. A computer executes the set of statements also known as a program in order to achieve the desired objective. A program can be written in different languages like COBOL, FORTRAN, PASCAL, C, C++, JAVA, Python etc. The statements written in a program differ from one language to another. However, one of the easiest technique to write a program is to use algorithms. The computer algorithm is a structure which explains the set of instructions that need to be executed in order to accomplish the objective of a progam. These computer algorithms can be translated very easily into computer programs written using a programming language as per the requirements. For example a algorithm can be written to prepare a cup of tea and the same is given below:

Algorithm to prepare a cup of tea
Line 1: start
Line 2: collect ingredients vessel, water, tea leaves, sugar, milk
Line 3: switch on the Heater
Line 4: Put the empty vessel on heater
Line 5: pour water as per the requirement in the vessel
Line 6: wait till the water is boiled
Line 7: Add milk, tea leaves and sugar as per the requirement
Line 8: wait till the mixture is boiled
Line 9: pour the boiling mixture in a cup
Line 10: serve the tea

The algorithm given above to prepare a cup of tea is one of the methods to prepare tea. However, different methods are used to prepare a cup of tea and the same may vary from one place to another. Therefore, the algorithm written is not the only algorithm to prepare tea rather 'n' number of algorithms can be written to prepare tea. Similarly a computer program can have different methods to write a computer program.

The notion of algorithm

An algorithm is a systematic sequence of instructions which are required to be executed in order to achieve the objective of an algorithm. The basic characteristics of an algorithm are given below:

(i) Finite

(ii) Single Entry and Single Exit point

(iii) Achieve Desired Objective

(iv) Systematic Sequence

Finite: The algorithm should be finite in nature, for example, an algorithm with a condition which is always true will lead to execution of a program or an algorithm infinite number of times.

NOTES

```
void main()
{
     while(1)
     {
         printf("i  am  stuck  in  the  loop  as  the
condition will never terminate");
     }
}
```

The program given above will not terminate as the condition mentioned in loop is always true. The program will result in displaying the text "I am stuck in the loop as the condition will never terminate" continuously on screen. Therefore, the need of the hour in this case is to write a program or an algorithm which will allow a user to terminate a program as and when required.

Single Entry and Single Exit point: Every algorithm should have a single entry point and a single exit point. In case a program is having multiple entry and exit points will lead to improper execution of an algorithm.

Table 1.1 *Single entry and single exit point*

Algorithm	Program using C language

Line 1: start
Line 2: declare a,b,c
Line 3: read a,b
Line 4: c=a+b
Line 5: display c
Line 6: stop
	void main()
	{
	int a,b,c;
	printf("Enter two numbers:");
	scanf("%d,%d", a,b);
	c=a+b;
	printf("the sum of two numbers is %d",c);
	}

The algorithm shown above is written to automate the process of addition of two numbers. An equivalent program in C language is written for the same for the benefit of the reader. The algorithm should start from line No. 1 and should end at line number 6 only. In case the algorithm starts from line No. 4 will generate error as the information about the initialization of the variable is not available at line No. 4. Therefore, the algorithm needs to be executed from a single point of entry and exit from a single point in order to maintain the integrity and accuracy of an algorithm.

NOTES

Achieve Desired Objective: An algorithm should be able to achieve the desired objective for which the same has been designed. For example the algorithm given above is designed to add two numbers and display the result to the user. In case the algorithm is not generating the summation of two numbers then the same cannot be treated as an algorithm which is fulfilling the requirement of an end-user.

Systematic Sequence: An algorithm needs to written in a sequence of instructions which are executed one after another. The sequence will always be implemented from line number first till the line number last.

The algorithm written to add two numbers is not necessarily the only method which can be used to automate the process of adding two numbers. As an example a simple algorithm can be written in "n" number of ways and to explain the same an algorithm to add two numbers is written in three different ways and the same is given below:

Table 1.2 *An algorithm to add two numbers in three different ways*

Algorithm-I	Algorithm-II	Algorithm-III
Line 1: start	Line 1: start	Line 1: start
Line 2: declare a,b,c	Line 2: declare a,b	Line 2: declare a,b
Line 3: read a,b	Line 3: read a,b	Line 3: read a,b
Line 4: c=a+b	Line 4: a=a+b	Line 4: b=a+b
Line 5: display c	Line 5: display a	Line 5: display b
Line 6: stop	Line 6: stop	Line 6: stop

The number of variables used in the above given algorithms varies which results in optimal usage of resources. The same algorithm can be written in different ways in order to achieve the desired objective. This helps a user in differentiating between a good algorithm and a bad algorithm. The comparison of the algorithms based on the usage of resources like memory-usage, time-taken etc motivates the reader to study the domain of algorithm analysis. Keeping in mind the proper utilization of memory-usage and time taken to solve a problem a good algorithm will always efficiently use the resources of a system.

1.3 FUNDAMENTALS OF ALGORITHMIC PROBLEM SOLVING

The problem solving is the main initiator of writing algorithms or computer programs. Everyone around you has different problems or requirements on daily basis. Every individual would like to develop automated tools for specific problem solving methods or procedures. Therefore, a computer program or an algorithm is written for automating the process of developing a set of instructions to fulfill the requirement of a user by automating the procedure of a problem solving technique. The process of writing an algorithm for a problem solving methods includes different steps and the same are shown in Figure 1.1

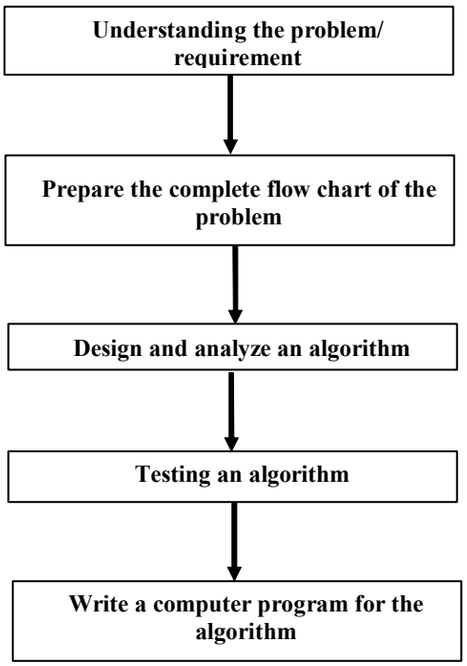


Fig. 1.1 *Steps of problem solving*

(i) Understanding the problem/requirement

Understand the requirement of a user in order to develop and design an algorithm. The requirement of a user can be an overview but as a designer of an algorithm the developer needs to understand all the processes and sub processes within the system. The problem understanding is one of the most important activity for designing an algorithm as if a developer skips even a small sub process within the problem definition will result is design of wrong algorithm. As discussed in the precious section one of the characteristics of an algorithm is to fulfill the desired objective which cannot be achieved if a developer has not understood the problem accurately and completely.

An algorithm developer needs to discuss all the confusions in order to understand the problem without any ambiguities. At the same time the developed should try to consider multiple alternatives which can enhance the process of achieving the final objective of a problem solution process. The evaluation of multiple alternatives will help a developer to prepare a plan for developing an algorithm which is designed to automate the process of problem solution effectively and efficiently.

(ii) Prepare the complete flow chart of the problem

Flow chart is a graphical representation to display the sequence of instruction within an algorithm or a computer program. The flow-chart tool helps the developer in preparing a graphical representation of the problem solving process which results in efficient design of an algorithm. This graphical representation helps the end-user

to understand the accuracy of the problem definition which is very essential before designing an algorithm.

(iii) Design and analyze an algorithm

NOTES

Based on the understanding of the problem solving process and graphical representation of the same an algorithm is written. The reader here can again get an idea of the importance of step (i) of the problem solving process.

Once the algorithm has been designed, the developer has to ensure that the resources like space and time of a computer are used by an algorithm effectively and efficiently. The process of analyzing an algorithm plays an important role while writing a program and execution of the same on a computer. A algorithm designed can develop alternative algorithms for the same problem solving process in order to analyze the same for selection of an optimal algorithm. The main objective of the analyzing an algorithm is to ensure that the algorithm uses the resources of a computer optimally.

(iv) Test the algorithm

Once the algorithm is designed the developer needs to ensure the correctness, completeness and accuracy of the algorithm. The process of checking the correctness of the algorithm is known as testing where the algorithm is tested for dummy situations and dummy data. This method of executing the algorithm is also known as dry run.

(v) Write a computer program for the algorithm

The algorithm written at step (iv) can be written in any programming language as per the requirement of the problem solving process in order to write a computer program for the problem solving process.

Check Your Progress

1. In what languages can a program be written?
2. What is the main initiator of writing algorithms or computer programs?

1.4 IMPORTANT PROBLEM TYPES

In the real world scenario the numbers of problems existing at present are infinite. However, in the domain of design and analysis of algorithms researchers and the scientific community have mainly focused on the following types of problems.

- Sorting
- Searching
- Graph problems

- Strings Processing
- Geometric Problems
- Combinatorial Problems
- Numerical Analysis Problems

NOTES

Sorting: The sorting problem is arranging a sequence of items either in ascending or descending order based on the requirements of a user. The sorted list of items can be prepared based on an integer or a character or a field depending on the need of the process. The sorted list of items is used for different purposes and one of the important usages is in searching for an item within a given list in lesser time. Different sorting algorithms are available for implementation, depending on the situation an appropriate sorting algorithm is selected for usage. Some of the sorting algorithms at present in user are given below:

- Selection sort
- Merge sort
- Bubble sort
- Insertion sort
- Quick sort
- Radix sort
- Heap sort

Searching: The searching problem is related to searching an item from a list of items. The item searched can be an integer or a character which helps an end-user in searching for an attribute as and when required based on the requirements. The searching algorithms are desingned using different methods like binary searching technique and binary searching technique. The sorting algorithms are also used in searching where first the list of items is sorted which makes the process of searching easier and faster.

Graph Problems: A graph is a collection of nodes also known as vertices and the vertices are connected with one another using edges. The graphs are one of the pivotal parts of analyses of algorithms as most of the algorithms use the graph theory concept in one or the other form. The graphs can be used in different real life problems like travelling salesman problem, optimal network path problem, shortest path searching algorithms etc. The graphs are also used in developing advances electronic chips where different cores are put in a single chip by simulating the concept prior to implementing the same in physical design.

Strings Processing: With the advent of technology in general and bio-informatics in particular the need for analyzing and processing text and strings has increased manifold. This has motivated researchers and Practitioners across the globe to develop algorithms for analyzing and processing of strings. The main objective of string processing algorithms is to analyze the presence of defined strings in order

NOTES

to analyze huge sized files effectively and efficiently. At present the need has been justified with the pattern recognition in the bio-informatics domain for genome sequence matching analysis and information retrieval.

Geometric Problems: In order to construct a geometric shape different processes are to be executed from plotting points and then connecting the points using lines based on the different parameters are automated using algorithms. These algorithms are known as geometric problem algorithms. The geometric problem algorithms are applied in different domain across horizontals and verticals in order to serve humanity in general and computer sciences in particular. The different domains like bio-medical equipments, robotics, graphics etc. are some of the field where the usage of geometric problem algorithms has become a necessity.

Combinatorial Problems: A problem will not necessarily have only one method of solution. Every problem solution method starts from an initial state and explores the possible outcomes in order to move from one state to another in order to evaluate the possible outcome for selection of an optimal option. The number of possible outcomes at a particular state is retrieved using permutations and combinations as well.

Numerical Analysis Problems: These algorithms are used mainly in mathematical problems where the solutions generated are continuous. Some numerical problems like roots between two given parameters using different methods is one of the best examples where the application of these algorithms helps the mathematicians in reaching higher levels of efficiency.

Check Your Progress

3. What is a flow chart?
4. What is a graph?

1.5 FUNDAMENTALS OF ANALYSIS OF ALGORITHM EFFICIENCY

The analysis of an algorithm for efficiency is a method to evaluate the efficiency of an algorithm for different parameters. The main objective of evaluating different algorithms for efficiency is to find out an algorithm which uses the system resources optimally and achieving the desired objective of an algorithm. As discussed earlier in this chapter for every problem the possible solutions are more than one which justifies the importance of evaluation of algorithms for finding out an optimal one based on utilization of resources. The basic definition of efficiency of an algorithm is which uses minimum memory and has less running time. The time efficiency of an algorithm demonstrates how fast an algorithm will be executed and space efficiency of an algorithm demonstrates the optimal units of memory required to execute an

algorithm. The two parameters memory and time are two basic parameters used to find the efficiency of an algorithm. However, some other parameters are also used to find out efficiency of an algorithm and the list of the parameters is given below:

- Size of input
- Running time
- Worst, Best and Average scenarios
- Asymptotic Notations

Size of input: The efficiency of algorithm is also evaluated based on the input size of an algorithm. For example the size of input in case of an word count algorithm will be the number of words and for a alphabet count algorithm the size of the input will be the number of alphabets given as input to an algorithm.

Running time: The efficiency of an algorithm based on running depends on the measuring unit used to measure running time. For example if the unit used for measurement in terms of second then the efficiency will be evaluated in seconds. In case the measuring unit is nano-seconds then the evaluation parameter unit will be in nano-seconds. The running time for an algorithm also is directly dependent on the speed of the computer, compiler used and quality of an algorithm. However, for uniformity the basic operations are identified within a algorithm and the time taken to complete the same is evaluated in order to analyze the efficiency of an algorithm.

Worst, Best and Average scenarios: The algorithm efficiency is also evaluated based on the different possibilities of the input size. In case the size of the input is worst fit where the size of the inputs is taken as extreme higher value and the efficiency of an algorithm based on the same is evaluated and the same is used to compare for a good algorithm. In case the size of the input is best fit where the size of the input is ideal and the efficiency of an algorithm is evaluated and the same criterion is used as one of the benchmarks in finding out a optimal algorithm. Similarly in case the size of the input is average then the efficiency of the algorithm is evaluated in order to find out the best selection of an algorithm.

Order of Growth: The efficiency of an algorithm is evaluated based on order of the growth in which an algorithm is performing. The order of growth is about how a algorithm is performing when the system used to execute the same very fast and what is the efficiency of an algorithm when the input size is doubled. The efficiency can be evaluated using a basic equation as given below:

$$T(n) = C_{op} \times C_n$$

where T(n) is running time of an algorithm

C_{op} is the time take for single basic operation

C_n is the number of basic operations within an algorithm.

NOTES

Asymptotic Notations: The efficiency of an algorithm is measure using asymptotic notations where three notations O, Ω, Θ are used. The notation O is known as big "Oh", the Ω notation is known as omega and the notation Θ is known as big theta.

For big O notation a function t(n) is said to be in O(g(n)), for big Ω notation a function t(n) is said to be in Ω (g(n)) and for big Θ notation a function t(n) is said to be in Θ (g(n)).

Check Your Progress

5. What is the analysis of an algorithm for efficiency?
6. What is the definition of efficiency of an algorithm?

1.6 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. A program can be written in different languages like COBOL, FORTRAN, PASCAL, C, C++, JAVA, Python etc.
2. The problem solving is the main initiator of writing algorithms or computer programs.
3. Flow chart is a graphical representation to display the sequence of instruction within an algorithm or a computer program.
4. A graph is a collection of nodes also known as vertices and the vertices are connected with one another using edges.
5. The analysis of an algorithm for efficiency is a method to evaluate the efficiency of an algorithm for different parameters
6. The basic definition of efficiency of an algorithm is which uses minimum memory and has less running time.

1.7 SUMMARY

- A computer executes the set of statements also known as a program in order to achieve the desired objective.
- A program can be written in different languages like COBOL, FORTRAN, PASCAL, C, C++, JAVA, Python etc.
- The statements written in a program differ from one language to another.
- An algorithm is a systematic sequence of instructions which are required to be executed in order to achieve the objective of an algorithm.
- An algorithm needs to written in a sequence of instructions which are executed one after another. The sequence will always be implemented from line number first till the line number last.

NOTES

- The algorithm written to add two numbers is not necessarily the only method which can be used to automate the process of adding two numbers.
- The number of variables used in the above given algorithms varies which results in optimal usage of resources.
- The comparison of the algorithms based on the usage of resources like memory-usage, time-taken etc motivates the reader to study the domain of algorithm analysis.
- The problem solving is the main initiator of writing algorithms or computer programs.
- Every individual would like to develop automated tools for specific problem solving methods or procedures.
- The process of writing an algorithm for a problem solving methods includes different steps.
- The requirement of a user can be an overview but as a designer of an algorithm the developer needs to understand all the processes and sub processes within the system.
- An algorithm developer needs to discuss all the confusions in order to understand the problem without any ambiguities.
- The evaluation of multiple alternatives will help a developer to prepare a plan for developing an algorithm which is designed to automate the process of problem solution effectively and efficiently.
- Flow chart is a graphical representation to display the sequence of instruction within an algorithm or a computer program.
- The searching problem is related to searching an item from a list of items. The item searched can be an integer or a character which helps an end-user in searching for an attribute as and when required based on the requirements.
- A graph is a collection of nodes also known as vertices and the vertices are connected with one another using edges.
- The analysis of an algorithm for efficiency is a method to evaluate the efficiency of an algorithm for different parameters
- The basic definition of efficiency of an algorithm is which uses minimum memory and has less running time.
- The basic definition of efficiency of an algorithm is which uses minimum memory and has less running time.

1.8 KEY WORDS

- **Algorithm:** An algorithm is a systematic sequence of instructions which are required to be executed in order to achieve the objective of an algorithm.

- **Searching Problem:** It is related to searching an item from a list of items. The item searched can be an integer or a character which helps an end-user in searching for an attribute as and when required based on the requirements.

NOTES

1.9 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. What is the complexity of algorithms?
2. How can one understand the problem?
3. What are the objectives of problem identification?

Long Answer Questions

1. List the different evaluation criterions used in evaluating the efficiency of an algorithm?
2. For a program to find out summation of 10 numbers find out the different methods to write an algorithm?
3. Write a note on single entry and single exit point.

1.10 FURTHER READINGS

Levitin, Anany. *Introduction to Design and Analysis of Algorithms*. Delhi: Pearson Education.

Ellis Horowitz, S. Sahani and Rajasekaran, *Fundamentals of Computer Algorithms.* Delhi: Galgotia Publications.

Goodrich, M T and R. Tomassia. *Algorithm Design: Foundations, Analysis and Internet Examples*. Delhi: John wiley and Sons.

UNIT 2 ASYMPTOTIC NOTATIONS

NOTES

Structure

2.0 INTRODUCTION

Asymptotic notations are the way to express time and space complexity. It represents the running time of an algorithm. If we have more than one algorithm with alternative steps then to choose among them, the algorithm with lesser complexity should be selected. To represents these complexities, asymptotic notations are used. Asymptotic notations are of three types Oh, Omega, and Theta. These are further classified as Big-oh, Small-oh, Big Omega, Small Omega, etc. This unit will introduce you with all of these.

2.1 OBJECTIVES

After going through this unit, you will be able to:

- Understand the Big-oh notation
- Discuss Omega notation
- Analyse theta notation

2.2 THETA (Θ) NOTATION

It is the method of expressing the tight bound of an algorithm's running time. For non-negative functions $f(n)$ and $g(n)$, if there exists an integer n_0 and positive constants c_1 and c_2; such that for all integers $n \geq n_0$,

$$\mathbf{0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)}$$

NOTES

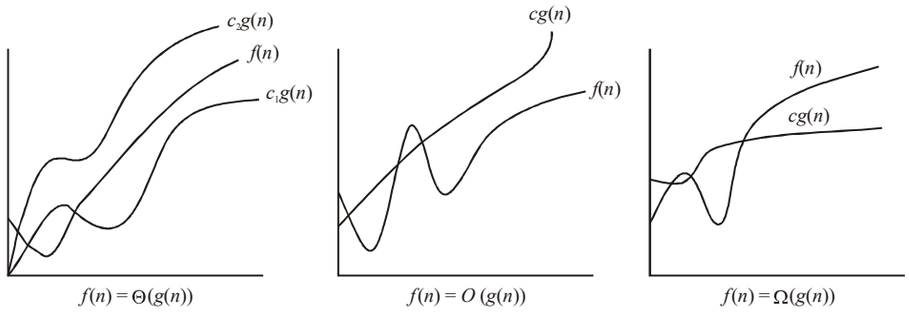


Fig. 2.1 *Functions of f(n) and g(n)*

$\Theta(g(n)) = \{f(n)$: There exist positive constants c_1, c_2 and $n_{0,}$ such that $0 \leq c_1g(n) \leq f(n) \leq c_2g(n)$ for all $n \geq n_0\}$.

A function $f(n)$ belongs to the set $\Theta(g(n))$ such that it can be 'sandwiched' between $c_1g(n)$ and $c_2g(n)$, for sufficiently large n.

Figure 2.1 shows the functions $f(n)$ and $g(n)$, where we have that $f(n) = \Theta(g(n))$. For all values of $n \geq n_0$, the value of $f(n)$ lies at or above $c_1g(n)$ and at or below $c_2g(n)$. In other words, for all $n \geq n_0$, the function $f(n)$ is equal to $g(n)$ to within a constant factor. We say that $g(n)$ is an **asymptotically tight bound** for $f(n)$.

Theta notation provides a tight bound on the running time of an algorithm. Consider the following examples:

1. $f(n) = 123$

   $122*1 \leq f(n) \leq 123*1$

   Here, $c_1 = 122$, $c_2 = 123$ and $n_0 = 0$

   So, $f(n) = \Theta(1)$

2. $f(n) = 3n + 5$

   $3n < 3n + 5 \leq 4n$

   Here, $c_1 = 3$, $c_2 = 4$ and $n_0 = 5$

   So, $f(n) = \Theta(n)$

3. $f(n) = 3n^2 + 5$

   $3n^2 < 3n^2 + 5 \leq 4n^2$

   Here, $c_1 = 3$, $c_2 = 4$ and $n_0 = 5$

   So, $f(n) = \Theta(n^2)$

4. $f(n) = 7n^2 + 5n$

   $7n^2 < 7n^2 + 5n$ for all n, $c_1 = 7$

   Also, $7n^2 + 5n \leq 8n^2$ for $n \geq n_0 = 5$, $c_2 = 8$

   So, $f(n) = \Theta(n^2)$

5. $f(n) = 2^n + 6n^2 + 3n$

   $2^n < 2^n + 6n^2 + 3n < 2^n + 6n^2 + 3n^2 < 2^n + 6.\ 2^n + 3.\ 2^n \leq 10.2^n$

   Here, $c_1 = 1$, $c_2 = 10$ and $n_0 = 1$

   So, $f(n) = \Theta(2^n)$

There can be infinite choices for c_1, c_2 and n_0.

Theta Ratio Theorem: Let $f(n)$ and $g(n)$ be such that $\lim_{n\to\infty} f(n) / g(n)$ exists, then $f(n) = \Theta(g(n))$

$\lim_{n\to\infty} f(n) / g(n) = c$ (some constant), where $0 < c < \infty$

Example, $f(n) = 2n + 4 = \Theta(n)$

$\lim_{n\to\infty} f(n) / g(n) = \lim_{n\to\infty} (2n + 4) / n = 2$ (constant)

So, $f(n) = \Theta(n)$

Some incorrect bounds are as follows:

$2n + 4 \neq \Theta(1)$

$2n + 5 \neq \Theta(n^2)$

$10n^2 + 3 \neq \Theta(1)$

2.3 BIG-OH (O) NOTATION

As any notation asymptotically bounds a function from above and below, Big-Oh is the formal method of expressing the upper bound of an algorithm's running time. It describe the limiting behaviour of a function if the argument tends towards a particular value or infinity. It is the measure of the longest amount of time it could possibly take for the algorithm to complete. For non-negative functions $f(n)$ and $g(n)$, if there exists an integer n_0 and constant $c > 0$, such that for all integers $n \geq n_0$,

$$\mathbf{0 \leq f(n) \leq cg(n)}$$

$O(g(n)) = \{f(n)$: There exist positive constants c and n_0, such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$.

Consider the following examples:

1. $f(n) = 13$

   $f(n) \leq 13*1$

   Here, $c = 13$ and $n_0 = 0$

   So, $f(n) = O(1)$

2. $f(n) = 3n + 5$

   $3n + 5 \leq 3n + 5n$

NOTES

NOTES

$3n + 5 \leq 8n$

Here, $c = 8$ and $n_0 = 1$

So, $f(n) = O(n)$

3. $f(n) = 3n^2 + 5$

$3n^2 + 5 \leq 3n^2 + 5n$

$3n^2 + 5 \leq 3n^2 + 5n^2$

$3n^2 + 5 \leq 8n^2$

Here, $c = 8$ and $n_0 = 1$

So, $f(n) = O(n^2)$

4. $f(n) = 7n^2 + 5n$

$7n^2 + 5n \leq 7n^2 + 5n^2$

$7n^2 + 5n \leq 12n^2$

Here, $c = 12$ and $n_0 = 1$

So, $f(n) = O(n^2)$

5. $f(n) = 2^n + 6n^2 + 3n$

$2^n + 6n^2 + 3n \leq 2^n + 6n^2 + 3n^2 \leq 2^n + 6.\ 2^n + 3.\ 2^n \leq 10.2^n$

Here, $c = 10$ and $n_0 = 1$

So, $f(n) = O(2^n)$

Big-Oh Ratio Theorem: Let $f(n)$ and $g(n)$ be, such that $\lim_{n\to\infty} f(n)/g(n)$ exists, then $f(n) = O(g(n))$,

$$\lim_{n\to\infty} f(n)/g(n) = c < \infty, \text{ also including the case in which limit is 0.}$$

$$\lim_{n\to\infty} f(n)/g(n) = \begin{cases} 0 & ; \quad f(n) \text{ grows slower than } g(n). \\ \infty & ; \quad f(n) \text{ grows faster than } g(n). \\ \text{otherwise} & ; \quad f(n) \text{ and } g(n) \text{ have same growth rates.} \end{cases}$$

Example, $f(n) = 3n^3 + 4n = O(n^3)$

$$\lim_{n\to\infty} f(n)/g(n) = \lim_{n\to\infty} (3n^3 + 4n) / n^3 = 3$$

So, $f(n) = O(n)$

Some incorrect bounds are as follows:

$2n + 4 \neq O(1)$

$2n^3 + 5 \neq O(n^2)$

$10n^2 + 3 \neq O(n)$

Some loose bounds are as follows:

$5n + 4 = O(n^2)$

$7n^3 + 5 = O(n^4)$

$105n^2 + 3 = O(n^3)$

NOTES

2.4 BIG OMEGA (Ω) NOTATION

Big Omega (Ω) is the method used for expressing the lower bound of an algorithm's running time. It is the measure of the smallest amount of time it could possibly take for the algorithm to complete.

For non-negative functions $f(n)$ and $g(n)$, if there exists an integer n_0 and constant $c > 0$, such that for all integers $n \geq n_0$,

$$\mathbf{0 \leq cg(n) \leq f(n)}$$

Consider the following examples:

1. $f(n) = 13$
   $f(n) \geq 12*1$
   where $c = 12$ and $n_0 = 0$
   So, $f(n) = \Omega(1)$
2. $f(n) = 3n + 5$
   $3n + 5 > 3n$
   where $c = 3$ and $n_0 = 1$
   So, $f(n) = \Omega(n)$
3. $f(n) = 3n^2 + 5$
   $3n^2 + 5 > 3n^2$
   where $c = 3$ and $n_0 = 1$
   So, $f(n) = \Omega(n^2)$
4. $f(n) = 7n^2 + 5n$
   $7n^2 + 5n > 7n^2$
   where $c = 7$ and $n_0 = 1$
   So, $f(n) = \Omega(n^2)$
5. $f(n) = 2^n + 6n^2 + 3n$
   $2^n + 6n^2 + 3n > 2^n$
   where $c = 1$ and $n_0 = 1$
   So, $f(n) = \Omega(2^n)$

Big Omega Ratio Theorem: Let $f(n)$ and $g(n)$ be such that $\lim_{n\to\infty} f(n) / g(n)$ exists, then $f(n) = \Omega(g(n))$,

$\lim_{n\to\infty} f(n) / g(n) > 0$, also including the case in which limit is ∞.

NOTES

Some incorrect bounds are as follows:

$2n + 4 \neq \Omega(n^2)$

$2n^3 + 5 \neq \Omega(n^4)$

$10n^2 + 3 \neq \Omega(n^3)$

Some loose bounds are as follows:

$5n + 4 = \Omega(1)$

$7n^3 + 5 = \Omega(n^2)$

$105n^2 + 3 = \Omega(n)$

2.5 LITTLE-OH (o) NOTATION

The asymptotic upper bound provided by Big-oh(O) notation may or may not be asymptotically tight. We use Little-Oh(o) notation to denote an upper bound that is not asymptotically tight.

$$\mathbf{0 \leq f(n) < cg(n)}$$

For Little-Oh notation: $\lim_{n \to \infty} f(n) / g(n) = 0$

Example, $3n + 9 = o(n^2)$

$$\lim_{n \to \infty} f(n) / g(n) = \lim_{n \to \infty} (3n + 9) / n^2 = 0$$

2.6 LITTLE OMEGA (ω) NOTATION

The asymptotic lower bound provided by Big Omega (Ω) notation may or may not be asymptotically tight. We use Little Omega (ω) notation to denote a lower bound that is not asymptotically tight.

$$\mathbf{0 \leq cg(n) < f(n)}$$

For Little Omega (ω) notation: $\lim_{n \to \infty} f(n) / g(n) = \infty$

Example, $7n^2 + 9n = w(n)$

$$\lim_{n \to \infty} f(n) / g(n) = \lim_{n \to \infty} (7n^2 + 9n) / n = \infty$$

Figure 2.2 shows the concept of asymptotic notation. Consider the function, say $f(n) = an^2 + bn + c$. Also, let us consider the band for n^2(maximum contributing term for large $\boldsymbol{n}$) and the lower and upper bounds are $c_1 g(n)$ and $c_2 g(n)$, respectively.

So, the function can be rewritten in terms of several asymptotic notations as:

$an^2 + bn + c = \omega(1)$

$an^2 + bn + c = \omega(n)$

$an^2 + bn + c = \Theta(n^2)$

$an^2 + bn + c = \Omega(1)$

$an^2 + bn + c = \Omega(n)$

$an^2 + bn + c = O(n^2)$

$an^2 + bn + c = O(n^6)$

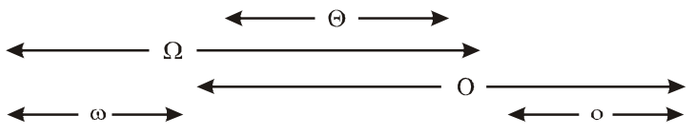
$an^2 + bn + c = O(n^{100})$

$an^2 + bn + c = o(n^{19})$

Also, $an^2 + bn + c \neq o(n^2)$ (Since it can never be tight bound)

$an^2 + bn + c \neq \omega(n^{19})$

$an^2 + bn + c \neq O(n)$



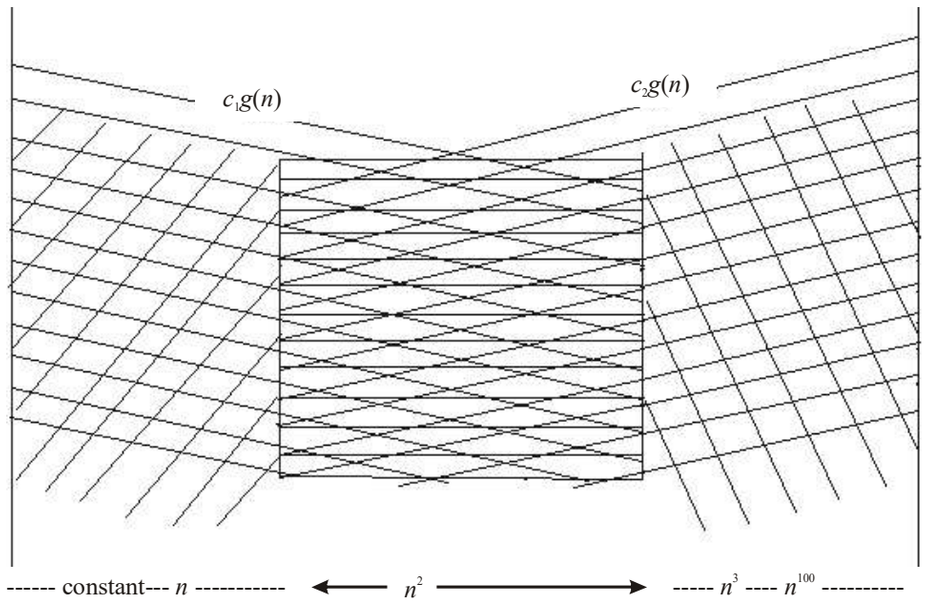


Fig. 2.2 *Asymptotic Notation*

NOTES

NOTES

Some more incorrect bounds are as follows:

$7n + 5 \neq O(1)$

$2n + 3 \neq O(1)$

$3n^2 + 16n + 2 \neq O(n)$

$5n^3 + n^2 + 3n + 2 \neq O(n^2)$

$7n + 5 \neq \Omega(n^2)$

$2n + 3 \neq \Omega(n^3)$

$10n^2 + 7 \neq \Omega(n^4)$

$7n + 5 \neq \Theta(n^2)$

$2n^2 + 3 \neq \Theta(n^3)$

Some more loose bounds are as follows:

$2n + 3 = O(n^2)$

$4n^2 + 5n + 6 = O(n^4)$

$5n^2 + 3 = \Omega(1)$

$2n^3 + 3n^2 + 2 = \Omega(n^2)$

Some correct bounds are as follows:

$2n + 8 = O(n)$

$2n + 8 = O(n^2)$

$2n + 8 = \Theta(n)$

$2n + 8 = \Omega(n)$

$2n + 8 = o(n^2)$

$2n + 8 \neq o(n)$

$2n + 8 \neq \omega(n)$

$4n^2 + 3n + 9 = O(n^2)$

$4n^2 + 3n + 9 = \Omega(n^2)$

$4n^2 + 3n + 9 = \Theta(n^2)$

$4n^2 + 3n + 9 = o(n^3)$

$4n^2 + 3n + 9 \neq o(n^2)$

$4n^2 + 3n + 9 \neq \omega(n^2)$

Correlation

- $f(n) = \Theta(g(n)) \approx f = g$
- $f(n) = O(g(n)) \approx f \leq g$

- $f(n) = \Omega(g(n)) \approx f \geq g$
- $f(n) = o(g(n)) \approx f < g$
- $f(n) = \omega(g(n)) \approx f > g$

NOTES

Properties of Asymptotic Notations

(i) Transitivity

(ii) Reflexivity

(iii) Symmetry

(iv) Transpose Symmetry

(i) **Transitivity**

- $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$ imply $f(n) = \Theta(h(n))$
- $f(n) = O(g(n))$ and $g(n) = O(h(n))$ imply $f(n) = O(h(n))$
- $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$ imply $f(n) = \Omega(h(n))$
- $f(n) = o(g(n))$ and $g(n) = o(h(n))$ imply $f(n) = o(h(n))$
- $f(n) = \omega(g(n))$ and $g(n) = \omega(h(n))$ imply $f(n) = \omega(h(n))$

(ii) **Reflexivity**

- $f(n) = \Theta(f(n))$
- $f(n) = O(f(n))$
- $f(n) = \Omega(f(n))$

(iii) **Symmetry**

- $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$

(iv) **Transpose Symmetry**

- $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$
- $f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$

Comparisons

- $f(n)$ is asymptotically smaller than $g(n)$ if $f(n) = o(g(n))$.
- $f(n)$ is asymptotically larger than $g(n)$ if $f(n) = \omega(g(n))$.

Check Your Progress

1. What is Big Omega?
2. Why do we use Little Omega?

2.7 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

NOTES

1. Big Omega (Ω) is the method used for expressing the lower bound of an algorithm's running time.
2. We use Little Omega (ω) notation to denote a lower bound that is not asymptotically tight.

2.8 SUMMARY

- As any notation asymptotically bounds a function from above and below, Big-Oh is the formal method of expressing the upper bound of an algorithm's running time.
- It describe the limiting behaviour of a function if the argument tends towards a particular value or infinity.
- Big Omega (Ω) is the method used for expressing the lower bound of an algorithm's running time.
- The asymptotic upper bound provided by Big-oh(O) notation may or may not be asymptotically tight. We use Little Omega (ω) notation to denote a lower bound that is not asymptotically tight.
- The asymptotic lower bound provided by Big Omega (Ω) notation may or may not be asymptotically tight.

2.9 KEY WORDS

- **Big-Oh:** It is a notation that is used to classify algorithms according to how their running time or space requirements grow as the input size grows.
- **Little Omega:** It is used to denote a lower bound that is not asymptotically tight.

2.10 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. Write a short note on Big-oh notation.
2. Describe Little Omega.
3. How does the Big Omega notation come into use?

Long Answer Questions

1. "Theta notation provides a tight bound on the running time of an algorithm." Discuss with an example.
2. Discuss the Big-Oh Ratio Theorem.
3. Discuss the concept of asymptotic notation. Support your answer with an illustration.

NOTES

2.11 FURTHER READINGS

Levitin, Anany. *Introduction to Design and Analysis of Algorithms*. Delhi: Pearson Education.

Ellis Horowitz, S. Sahani and Rajasekaran, *Fundamentals of Computer Algorithms.* Delhi: Galgotia Publications.

Goodrich, M T and R. Tomassia. *Algorithm Design: Foundations, Analysis and Internet Examples*. Delhi: John wiley and Sons.

UNIT 3 PERFORMANCE ANALYSIS

NOTES

Structure

3.0 INTRODUCTION

Performance analysis of an algorithm depends upon two factors- the amount of memory used and the amount of compute time consumed on any CPU. Formally they are notified as complexities in terms of: Space and Time Complexity. Space Complexity of an algorithm is the amount of memory it needs to run to completion i.e. from start of execution to its termination. Space need by any algorithm is the sum of fixed and variable components. This unit will explain about performance analysis.

3.1 OBJECTIVES

After going through this unit, you will be able to:

- Understand space complexity
- Discuss about time complexity
- Analyze the pseudo code for algorithms

3.2 SPACE COMPLEXITY

Various criteria can be used to judge an algorithm, such as:

- Is it doing what is wanted from it?
- Is it working correctly in accordance with the original specifications of the task?

- Is there a document describing how to use it and how it works?
- Are procedures structured in such a way that they are able to perform logical sub-functions?
- Is the code of algorithm readable?

NOTES

These criteria are very important as far as writing software is concerned, especially for large systems. Algorithms can also be judged using some other criteria having a more direct relationship with their performance. These have to do with their computing time and storage requirements.

Definitions

Profiling: Profiling is the process of the execution of a correct program on data sets and the measurement of the time and storage taken for computing the results. It is also known as performance profile. These timing figures can confirm and point out logical places for performing useful optimization and hence are very useful. Profiling can be done on programs that are devised, coded, proved correct and debugged on a computer.

Debugging: Debugging refers to the process of execution of program on sample data sets for determining if faulty results occur. In other words, debugging is concerned with conducting tests for uncovering errors and ensuring that the defined input will give the actual results which agree with the required results.

Debugging only points to the presence of errors; it does not point to their absence. Debugging is not testing but always occurs as a consequence of testing. Debugging begins with the execution of a test case. The debugging process attempts to match symptom with cause, thereby leading to error correction. Debugging has two outcomes. Either the error is detected and corrected or the error is not found.

Priori analysis: It is also known as machine-independent and programming language-independent analysis, is done to bind the algorithms computing time.

Posteriori testing: It is also known as machine-dependent and programming language-dependent analysis, is done to collect the actual statistics about the algorithms consumption of time and space while it is executing.

A priori analysis of algorithms is concerned chiefly with determination of order of magnitude/frequency count of the step/statement. This can be determined directly from the algorithm, independent of the machine it will be executed on and the programming language the algorithm is written in.

For example: Consider the three program segments a, b, c

```
a. for i ← 1 to n
     repeat x ← n + y
b. for i ← 1 to n
c. for j ← 1 to n
```

NOTES

```
                x ← x + y
            repeat
        repeat
            for segment a the frequency count is 1;
            for segment b the frequency count is n;
            for segment c the frequency count is n²
        These frequencies 1, n, n² are said to be different
increasing orders of magnitude.
```

3.2.1 Space Complexity

The space complexity of an algorithm indicates the quantity of temporary storage required for running the algorithm, i.e. the amount of memory needed by the algorithm to run to completion.

In most cases, we do not count the storage required for the inputs or the outputs as part of the space complexity. This is so because the space complexity is used to compare different algorithms for the same problem in which case the input/output requirements are fixed.

Also, we cannot do without the input or the output, and we want to count only the storage that may be served. We also do not count the storage required for the program itself since it is independent of the size of the input.

Like time complexity, space complexity refers to the worst case, and it is usually denoted as an asymptotic expression in the size of the input. Thus, a 0(n) – space algorithm requires a constant amount of space independent of the size of the input.

The amount of memory an algorithm needs to run to completion is called its space complexity. The space required by an algorithm consists of the following two components:

(i) Fixed or static part: Fixed or static part is not dependent on the characteristics (such as number size) of the inputs and outputs. It includes the various types spaces, such as instruction space (i.e., space for code), space for simple variables and fixed-size component variables, space for constants, etc.

(ii) Variable or dynamic part: Variable or dynamic part consists of the space required by component variables whose size is dependent on the particular problem instance at run-time being solved, the space needed by referenced variables and the recursion stack space (depends on instance characteristics).

The space requirements S(p) of an algorithm p is S(p) = c + Sp (instance characteristics), where 'c' is a constant.

We are supposed to concentrate on estimating SP (instance characteristics) since the first part is static.

Example 3.1: The problem instances for algorithm are characterized by n, the number of elements to be summed. The space needed by n is one word since it is of type integer. The space needed by a is the space needed by variables of type array of floating-point numbers.

This is at least n words since a must be large enough to hold the n elements to be summed. So, we obtain $S_s(n) = (n + 3)$ (n for a[ ], one each for n, i, and s).

Iterative function for sum

```
Algorithm RSum (a, n)
{
if (n 0) then     return 0.0;
else  return
RSum (a, n - 1) + a[n];
}
```

3.3 TIME COMPLEXITY

The time complexity of an algorithm may be defined as the amount of time the computer requires to run to completion.

The time T(P) consumed by a program P is the sum of the compile-time and the run- time (execution-time). The compile time is independent of the instance characteristics. Also, it may be assumed that a compiled program can be run many times without recompilation. As a result, we are more interested in the run-time of a program. This run-time is denoted by t_p (instance characteristics).

Many factors on which t_p depend are not known at the time a program is written; so it is always better to estimate t_p. If we happen to know the type of the compiler used, then we could proceed to find the number of additions, subtractions, multiplications, divisions, compare statements, loads, stores and so on that would be made by a program P.

So we can obtain an expression of the form.

$$t_p(n) = C_a\,ADD(n) + C_s\,SUB(n) + C_m\,MUL(n) + C_d\,DIV(n) + ...$$

where n denotes the instance characteristics, and C_a, C_s, C_m, C_d and so on denote the time needed for addition, subtraction, multiplication, division and so on.

But here we need to note that the exact amount of time needed for the operations mentioned here cannot be found exactly; so instead we could only count the number of program steps, which means that a program step is counted.

A program step is defined as a syntactically or semantically meaningful segment of a program that has an execution time that is independent of the instance characteristics.

For example, Consider the statement return a + b × c + d " e/f

where this can be regarded as a step since its execution time is independent of the instance characteristics.

NOTES

The number of steps any program statement is assigned depends on the type of statement. The comments do not count for the program step; a general assignment statement, which does not call another algorithm, is considered one step whereas in an iterative statement like for, while and repeat_until, we count the step only for the control part of the statement.

The general syntax for 'for' and 'while' statements is as follows:

for i = (exprl) to (expr2) do

while(expr) do

Each execution of the control part of a while statement is given step count equal to the number of step counts assignable to <expr>. The step count for each execution of the control part of a for statement is one, unless the counts attributable to <expr> and <exprl> are functions of the instance characteristics.

Check Your Progress

1. What is space complexity?
2. On what does the number of steps any program statement is assigned depends on?
3. What is time complexity of an algorithm?

3.4 PSEUDO CODE FOR ALGORITHMS

A pseudo-code is neither an algorithm nor a program. It is an art of expressing a program in simple English that parallels the forms of a computer language. It is basically useful for working out the logic of a program. Once the logic seems right, you can attend to the details of translating the pseudo-code to the actual programming code. The advantage of pseudo-code is that it lets you concentrate on the logic and organization of the program while sparing you the efforts of simultaneously worrying how to express the ideas in a computer language.

A simple example of pseudo-code:

```
set highest to 100
set lowest to 1
ask user to choose a number
guess ( highest + lowest ) / 2
while guess is wrong, do the following:
   {
   if guess is high, set highest to old guess minus 1
```

```
if guess is low, set lowest to old guess plus 1
new guess is ( highest + lowest) / 2
}
```

3.4.1 Coding

NOTES

In the field of computer programming, the term **code** refers to instructions to a computer in a programming language. The terms **'code'** and **'to code'** have different meanings in computer programming. The noun **'code'** stands for source code or machine code. The verb **'to code' ,** on the other hand**,** means writing source code to a program. This usage seems to have originated at the time when the first symbolic languages evolved and were punched onto cards as 'codes'.

It is a common practice among engineers to use the word 'code' to mean a single program. They may say 'I wrote a code' or 'I have two codes'. This inspires wincing among the literate software engineer or computer scientists. They rather prefer to say 'I wrote some code' or 'I have two programs'. As in English it is possible to use virtually any word as a verb, a programmer/coder may also say 'coded a program'; however, since a code is applicable to various concepts, a coder or programmer may say 'hard-coded it right into the program' as opposed to the meta-programming model, which might allow multiple reuses of the same piece of code to achieve multiple goals. As compared to a hard-coded concept, a soft-coded concept has a longer lifespan. This is the reason of soft-coding of concept by the coder.

While writing your code, you need to remember the following key points:

- **Linearity:** If you are using a procedural language, you need to ensure that code is linear at the first executable statement and continues to a final return or end of block statement.
- **If constructs:** You would better use several simpler nested 'if' constructs rather than a complicated and compound 'if' constructs.
- **Layout:** Code layout should be formatted in such a way that it provides clues to the flow of the implementation. Layout is an important part of coding. Thus, before a project starts, there should be agreement on the various layout factors, such as indentation, location of brackets, length of lines, use of tabs or spaces, use of white space, line spacing, etc.
- **External constants:** You should define constant values outside the code. It ensures easy maintenance. Changing hard-coded constants takes too much time and is prone to human error.
- **Error handling:** Writing some form of error handling into your code is equally important.
- **Portability:** Portable code makes it possible for the source file to be compiled with any compiler. It also allows the source file to be executed on any machine and operating system. However, creating a portable code is a

fairly complex task. The machine-dependent and machine-independent codes should be kept in separate files.

3.4.2 Program Development Steps

NOTES

The following steps are required to develop a program:

- Statement of the problem
- Analysis
- Designing
- Implementation
- Testing
- Documentation
- Maintenance

Statement of the problem: A problem should be explained clearly with required input/output and objectives of the problem. It makes easy to understand the problem to be solved.

Analysis: Analysis is the first technical step in the program development process. To find a better solution for a problem, an analyst must understand the problem statement, objectives and required tools for it.

Designing: The design phase will begin after the software analysis process. It is a multi-step process. It mainly focuses on data, architecture, user interfaces and program components. The importance of the designing is to get the quality of the product.

Implementation: A new system will be implemented based on the designing part. It includes coding and building of new software using a programming language and software tools. Clear and detailed designing greatly helps in generating effective code with less implementing time.

Testing: Program testing begins after the implementation. The importance of the software testing is in finding the uncover errors, assuring software quality and reviewing the analysis, design and implementation phases.

Software testing will be performed in the following two technical ways:

- **Black box tests** or **Behavioral tests** (testing in the **large**): These types of techniques focus on the information domain of the software.

  Example: Graph-based testing, Equivalence partitioning, Boundary value analysis, Comparison testing and Orthogonal array testing.

- **White box tests** or **Glass box tests** (testing in the **small**): These types of techniques focus on the program control structure.

  Example: Basis path testing and Condition testing

- **Documentation**: Documentation is descriptive information that explains the usage as well as functionality of the software.

Documentation can be in several forms:

- o Documentation for programmers
- o Documentation for technical support
- o Documentation for end-users

NOTES

- **Maintenance**: Software maintenance starts after the software installation. This activity includes amendments, measurements and tests in the existing software. In this activity, problems are fixed and the software updated to make the system faster and better.

Programming is the process of devising programs in order to achieve the desired goals using computers. A good program has the following qualities:

- A program should be **correct** and designed in accordance with the specifications so that anyone can understand the design of the program.
- A program should be **easy to understand.** It should be designed that anyone can understand its logic.
- A program should be **easy to maintain and update.**
- It should be **efficient** in terms of the speed and use of computer resources such as primary storage.
- It should be **reliable.**
- It should be **flexible**; that is to say, it should be able to operate with a wide range of inputs.

Check Your Progress

4. What is the advantage of pseudo-code?
5. What does code refer to?

3.5 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. The amount of memory an algorithm needs to run to completion is called its space complexity.
2. The number of steps any program statement is assigned depends on the type of statement.
3. The time complexity of an algorithm may be defined as the amount of time the computer requires to run to completion.
4. The advantage of pseudo-code is that it lets you concentrate on the logic and organization of the program while sparing you the efforts of simultaneously worrying how to express the ideas in a computer language.

5. In the field of computer programming, the term code refers to instructions to a computer in a programming language.

NOTES

3.6 SUMMARY

- Algorithms can be judged using some other criteria having a more direct relationship with their performance.
- Debugging refers to the process of execution of program on sample data sets for determining if faulty results occur.
- Debugging only points to the presence of errors; it does not point to their absence. Debugging is not testing but always occurs as a consequence of testing.
- Debugging begins with the execution of a test case. The debugging process attempts to match symptom with cause, thereby leading to error correction.
- A priori analysis of algorithms is concerned chiefly with determination of order of magnitude/frequency count of the step/statement.
- The space complexity of an algorithm indicates the quantity of temporary storage required for running the algorithm, i.e. the amount of memory needed by the algorithm to run to completion.
- The amount of memory an algorithm needs to run to completion is called its space complexity.
- The time complexity of an algorithm may be defined as the amount of time the computer requires to run to completion.
- A pseudo-code is neither an algorithm nor a program. It is an art of expressing a program in simple English that parallels the forms of a computer language.
- In the field of computer programming, the term code refers to instructions to a computer in a programming language.
- If you are using a procedural language, you need to ensure that code is linear at the first executable statement and continues to a final return or end of block statement.
- Portable code makes it possible for the source file to be compiled with any compiler.
- Analysis is the first technical step in the program development process.
- The design phase will begin after the software analysis process. It is a multi-step process.
- Program testing begins after the implementation. The importance of the software testing is in finding the uncover errors, assuring software quality and reviewing the analysis, design and implementation phases.

- This activity includes amendments, measurements and tests in the existing software.
- A program should be correct and designed in accordance with the specifications so that anyone can understand the design of the program.

NOTES

3.7 KEY WORDS

- **Profiling:** Profiling is the process of the execution of a correct program on data sets and the measurement of the time and storage taken for computing the results.
- **Debugging:** Debugging refers to the process of execution of program on sample data sets for determining if faulty results occur.
- **Priori Analysis:** It is also known as machine-independent and programming language-independent analysis is done to bind the algorithms computing time.
- **Posteriori Testing:** It is also known as machine-dependent and programming language-dependent analysis is done to collect the actual statistics about the algorithms consumption of time and space while it is executing.

3.8 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. What do you understand by space complexity?
2. Discuss about time complexity.
3. Analyze the pseudo code for algorithms.

Long Answer Questions

1. Explain the following in detail:
   (i) Profiling
   (ii) Debugging
   (iii) Priori analysis
   (iv) Posteriori testing
   (v) Linearity
2. Discuss in detail about space complexity.
3. "The terms '**code**' and '**to code**' have different meanings in computer programming." Explain.

4. "The time T(P) consumed by a program P is the sum of the compile-time and the run- time (execution-time). The compile time is independent of the instance characteristics." Discuss.

NOTES

3.9 FURTHER READINGS

Levitin, Anany. *Introduction to Design and Analysis of Algorithms*. Delhi: Pearson Education.

Ellis Horowitz, S. Sahani and Rajasekaran, *Fundamentals of Computer Algorithms.* Delhi: Galgotia Publications.

Goodrich, M T and R. Tomassia. *Algorithm Design: Foundations, Analysis and Internet Examples*. Delhi: John wiley and Sons.

NOTES

BLOCK - II
MATHEMATICAL ANALYSIS OF NON-RECURSIVE ALGORITHMS

UNIT 4 ANALYSIS OF RECURSIVE ALGORITHMS

Structure

4.0 INTRODUCTION

The complexity of an algorithm is often analyzed to estimate the resources. The running time of non-recursive algorithms is quite straightforward. You count the lines of code, and if there are any loops, you multiply by the length. However, recursive algorithms are not that intuitive. The Fibonacci sequence is one relation that is defined by the recurrence relation. The Fibonacci sequence is probably one of the most famous and most widely written-about number sequences in mathematics. This unit will explain you about recursive algorithms and Fibonacci numbers.

4.1 OBJECTIVES

After going through this unit, you will be able to:

- Understand the analysis of recursive algorithms
- Analyse the algorithm to compute Fibonacci numbers
- Discuss the recursive algorithms for Fibonacci numbers

4.2 RECURSION

NOTES

Whenever there emerges a situation in which a programmer feels to perform a similar operation repeatedly to obtain the desired output by eliminating the coding complexity introduced after using looping constructs recursion is considered as best implementation choice.

Recursion is a situation in which a function is granted the special provision to call itself directly or indirectly. This repeated execution of same continues up to certain condition is met. The code or logical statements encapsulated within the recursion function execute themselves without user interruption. The function or process that is capable of repeating its functionality by calling itself again and again is called as recursion function. Let's understand this by a simple example that is preparing a coffee.

Case 1: Prepare a single cup of black coffee for your self

Case 2: Prepare same black coffee for ten friends

After implementing a recursive approach in while preparing tea it will be best to prepare and draft the methodology required to make a cup of black coffee and feed the same in the coffee maker. whenever you need to make more than one cup you only had to specify the quantity of ingredients the machine will repeatedly follow the way as prescribed for single cup till specified limit is not reached. This process of coffee making is a recursive approach and the recursive function is the set of instruction feed to machine to prepare coffee.

In more pragmatic scenario rincursión technique is used as problem solving approach wherein a major problem is branched into smaller but similar sub problems until a programer reaches an atomic problem sufficient enough to be solved trivially. Recursion empowers programmer to code programs more elegant, simple and clear. There is no doubt that recursion avoids code complexity in the programs but it reduces programs execution speed. Recursions use more memory and are generally slow.

Check Your Progress

1. What is recursion?
2. What is called as a recursion function?

NOTES

4.3 RECURSIVE ALGORITHMS

Recursive algorithms are those functions or algorithms specified to perform its operation recursively. If above example for preparation of black coffee is considered the actual set of statements that correspond for black coffee preparation including their systematic flow is considered as recursive algorithm.

Recursive algorithm to find positive integer

Even (positive integer *k*)

Input: *k* , a positive integer

Output: *k*-th even natural number (the first even being *0*)

Algorithm:

if *k* = *1*, **then** return *0*;

else return **Even(***k-1***)** + *2*.

4.4 ALGORITHMS FOR COMPUTING FIBONACCI NUMBERS

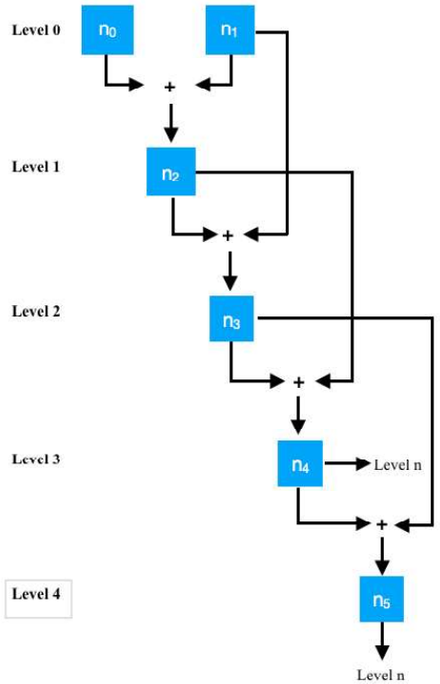
Fibonacci numbers is defined as a mathematical pattern of numbers (integers) arranged in a proper sequence. The arrangement of numbers in Fibonacci series beings with the 0 and 1 as the first and second numbers in series. The behavior of Fibonacci series depends on these first two numbers because the number which follows next in the series is the sum of two previous numbers in the generated series. Mathematically Fibonacci series can also be defined by establishing a recurrence relation between the individual but consecutive numbers in the series and the same relationship can be represented as $F_n = F_{n-1} + F_{n-2}$, n e – 2 with the base values $F_0 = 0$ and $F_1 = 1$. Therefore, the fibonacci series for six numbers starting with 0 and 1 will be 0,1,1,2,3,5.

4.4.1 Mathematical Representation

The mathematical logic behind the generation of Fibonacci series is very simple to understand. If a fibonacci series F with n elements is represented by Fn and the first number of the series F_1 is 0 and second element of the series $F_2 = 1$ therefore, the next element F_3 of Fibonacci series is equal to sum of F_1 and F_2 i.e. F3 = F1 + F2. The nth number in Fibonacci series can be obtained by adding Fn-1+Fn-2 numbers from the given series that is $F_n = (F_{n-1}+F_{n-2})$

NOTES

4.4.2 Graphical Representation



4.4.3 Algorithm to Generate Fibonacci Series

After understanding the definition, mathematical and graphical representation it is very easy to pen down the algorithm to generate Fibonacci series. On the basis of the above representations the simple algorithm for Fibonacci series is provided in Table 4.1.

Table 4.1 *A simple algorithm for Fibonacci series*

Step	Algorithm to generate Fibonacci Series	
1	**Fibonacci_Series(n)**	//Function Declaration with argument n to limit the series
2	**{**	
3	**if(n<=1) then**	//check if given limit is less or equal to 1
4	**Write (n)**	//if so, then simply write value of 1 which is less or equal to 1
5	**Else**	//If not then
6	**{**	
7	**fn1=0; fn2=1;**	//declare and initiate two variables n1 and n2
8	**for i=2 to n do**	//being loop to iterate till looping variable is less or equal to n
9	**{**	
10	**fn=fn0+fn1;**	//obtain number of Fibonacci series by adding fn0 and fn1(the previous consecutive numbers)
11	**f1=f2;**	//reinitialise the variables
12	**f2=fn;**	
13	**}**	
14	**write(fn)**	//after loop terminates write all values stores in fn
15	**}**	
16	**}**	

NOTES

Algorithm to generate Fibonacci Series using Recursion

```
Fibo(n)
Begin
 if n <= 1 then
      Return n;
 else
      Return Call Fibo(n-1) + Call Fibo(n-2);
 endif
End
```

Illustration

```
F(n) = 1                 when n <= 1
     = F(n-1) + F(n-2)   when n > 1
i.e.,
F(0) = 0
F(1) = 1
F(2) = F(2-1) + F(2-2)
     = F(1) + F(0)
     = 1 + 0
     = 2
```

Check Your Progress

3. What is a Fibonacci number?
4. How is a Fibonacci number defined mathematically?

4.5 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. Recursion is a situation in which a function is granted the special provision to call itself directly or indirectly.
2. The function or process that is capable of repeating its functionality by calling itself again and again is called as recursion function.
3. Fibonacci numbers is defined as a mathematical pattern of numbers (integers) arranged in a proper sequence.
4. Mathematically Fibonacci series can also be defined by establishing a recurrence relation between the individual.

NOTES

4.6 SUMMARY

- Recursion is a situation in which a function is granted the special provision to call itself directly or indirectly.
- The code or logical statements encapsulated within the recursion function execute themselves without user interruption.
- The function or process that is capable of repeating its functionality by calling itself again and again is called as recursion function.
- After implementing a recursive approach in while preparing tea it will be best to prepare and draft the methodology required to make a cup of black coffee and feed the same in the coffee maker.
- In more pragmatic scenario rincursión technique is used as problem solving approach wherein a major problem is branched into smaller but similar sub problems until a programer reaches an atomic problem sufficient enough to be solved trivially.
- There is no doubt that recursion avoids code complexity in the programs but it reduces programs execution speed.
- Fibonacci numbers is defined as a mathematical pattern of numbers (integers) arranged in a proper sequence.
- Mathematically Fibonacci series can also be defined by establishing a recurrence relation between the individual but consecutive numbers in the series and the same relationship can be represented as $F_n = F_{n-1} + F_{n-2}$, ne – 2 with the base values $F_0 = 0$ and $F_1 = 1$.
- The mathematical logic behind the generation of Fibonacci series is very simple to understand.
- After understanding the definition, mathematical and graphical representation it is very easy to pen down the algorithm to generate Fibonacci series.

4.7 KEY WORDS

- **Fibonacci Series:** It is a series of numbers in which each number (*Fibonacci number*) is the sum of the two preceding numbers. The simplest is the series 1, 1, 2, 3, 5, 8, etc.
- **Recursive Algorithms:** It is an algorithm which calls itself with smaller input values, and obtains the result for the current input by applying simple operations to the returned value for the smaller input.

4.8 SELF ASSESSMENT QUESTIONS AND EXERCISES

NOTES

Short Answer Questions

1. What is recursion and recursive algorithm?
2. Describe recursive algorithm for checking whether number is odd or even?
3. Write a short note on recursion function.

Long Answer Questions

1. Explain the generation of Fibonacci series.
2. Write and explain the algorithmic steps for generation of Fibonacci series.
3. "The behavior of Fibonacci series depends on the first two numbers." Explain why?

4.9 FURTHER READINGS

Levitin, Anany. *Introduction to Design and Analysis of Algorithms*. Delhi: Pearson Education.

Ellis Horowitz, S. Sahani and Rajasekaran, *Fundamentals of Computer Algorithms.* Delhi: Galgotia Publications.

Goodrich, M T and R. Tomassia. *Algorithm Design: Foundations, Analysis and Internet Examples*. Delhi: John wiley and Sons.

NOTES

UNIT 5 EMPIRICAL ANALYSIS OF ALGORITHMS

Structure

5.0 INTRODUCTION

Algorithm describes the stepwise solution required to solve a particular problem in a specific problem domain. There can be number of alternative approaches to solve any problem, what matters is the efficient execution of that selected approach technically known as an algorithm. In other words we can say every designed algorithm is associated with space and time complexity constraints. In order to ensure smooth and efficient algorithm that very algorithm needs to address space and time complexity properly. Empirical analysis of any algorithm deals with the critical analysis of an algorithm that is how fast it works, how much memory it is going to occupy and what attributes are incurred therein that causes increased lagging in programme execution. This unit will explain about empirical analysis of algorithms.

Sorting a problem is an example to understand empirical analysis of an algorithm designed to perform sort. Sorting is understood as a process wherein the items in a list are compared, rearranged or swapped till one arrives at the solution that is a sorted list of items or generally numbers. The complexity and the behaviour of a particular sorting technique depends on the degree or extent of comparison, rearrangement or swapping involved before one arrives at solution. If the extent of swap/comparison operations or degree of rearrangement is more and it varies across different available sorting techniques than one can distinguish these algorithms from one other by performing a critical empirical analysis on their operational behaviour. On the basis of the computational complexity associated with the sorting techniques are categorized as $O(n \log n)$, $O(\log^2 n)$ and $O(n^2)$, where n represents the size of data. Sorting can be either comparison based or

NOTES

non-comparison based. Comparison based sorting approach sorts data by comparing the data values while as non-comparison algorithm sorts without using pair-wise comparison of data values. Few popular comparison based sorting techniques are Brute force, selection sort, merge sort, bubble sort, insertion sort, sequential sort, etc. The empirical analysis of few popular sorting techniques are discussed in this unit.

5.1 OBJECTIVES

After going through this unit, you will be able to:

- Understand the nature of Brute force
- Discuss the meaning and function of selection sort
- Differentiate between bubble sort and selection sort
- Analyze what sequential sort is

5.2 BRUTE FORCE

Brute force is defined as a type of problem solving approach wherein a problem solution is directly based on the problem statement or the problem definition that is provided. It is considered as the easiest approach to adopt and is also very useful when problem domain is not that much complex. For example,, computing a factorial of a number, Multiplication of matrices, searching and sorting.

Let's consider an example to compute a^n (where, $a > 0$ and n is any non-integer).

Based on the definition of exponentiation $a^n = a*a*a*\ldots*a$

Implies using brute force to solve the above exponentiation problem it requires (n-1) repetitive multiplications.

Instead of using brute force approach to solve above problem if recursive approach is used the complexity of the problem is reduced to O(log(n)) because $a^n = a^{n/2} * a^{n/2}$.

5.2.1 Selection Sort using Brute Force approach

The selection sort using brute force approach will work by performing following steps:

- First scan to locate the smallest item in the list and swap the said element with first element in the list.
- Now, start with 2nd element in the list and scan for next smallest in the list and swap it with 2nd element in the list or array.
- Continue the same approach till (n-1)th element is placed at its proper location in the list.

Therefore, the complexity of the algorithm is O(n^2) and the total complexity introduced by swapping is O(n). Similarly, in case of bubble sort if the size of the array is n the total number of comparisons sought to is O(n^2), the number of the swaps depends on the behaviour of the given array and in worst case it is O(n^2) and the best case complexity can be O(n).

It needs to be mentioned that for any sorting or searching approach one can use brute force approach to arrive at the solution, however if the size of the array is very large then to prefer the best suited algorithm is choice to solve that very problem.

5.3 SELECTION SORT

Selection sort works on the basic principle concept to sort the given array of data by either selecting largest or smallest item/number from the given array. The selection of the smallest/ largest number is carried out by scanning all the length of the array that means after scanning all the elements of the array to locate the desired element to be notified as sorted. The selection of a given number is validated by performing comparisons of the current element with its adjacent elements in the list thereafter swapping the proper element and placing it at its perfect index in a sorted array.

Let's consider an array A[n] with 'n' elements to be sorted. Sorting on the basis of selection of largest or smallest number in the list requires scanning of all the 'n' elements in an array A[n] consuming n-1 comparisons. Similarly, finding next largest or smallest element from the unsorted array requires n-1, here n is the effective size of the array A[n]. After every single sort the effective size of the array is reduced by same value. For example if initial size n=5, in first case n-1= 5-1= 4 comparisons will happen and the effective size will be now n-1=4. In order to find second sorted elements n-1=4-1=3 comparisons will happen because effective size of the array has been reduced from 5 to 4. Comparisons and swapping will conclude once effective size will become 1. That means total comparisons for n=5 is 4+3+2+1= 10.

Algorithm

```
SELECTION SORT (DATA, N)
1.   Repeat  for K=1 to N-1.
2.    Set Max = DATA[K] and LOC=K
3.      Repeat For J=1 to N-K
    i.    If (MAX< DATA[J+1]) then:
              a) MAX=DATA[J+1]
              b) LOC= J+1
     [End of If in step i]
   [End of For loop in step 3]
4.    Set TEMP=DATA[J+1]
5.    Set  DATA [j+1]=MAX
6.    Set  DATA[LOC] = TEMP
     [end of For loop in step 1]
7.    Exit
```

Example

Elements	0	1	2	3	4
Data	25	17	14	16	9
1st Pass	25	17	14	16	9
	25	17	14	16	9
	25	17	14	16	9
	25	17	14	16	9
	9	17	14	16	25
2nd Pass	9	17	14	16	25
	9	17	14	16	25
	9	17	14	16	25
	9	16	14	17	25
3rd Pass	9	16	14	17	25
	9	16	14	17	25
	9	14	16	17	25
4th Pass	9	14	16	17	25
Sorted	9	14	16	17	25

Analysis:

- Let A[n] is a given array with random elements.
- Selecting either smallest/largest elements requires scanning of n elements.
- It performs n-1 comparisons.
- Therefore, the next elements are selected by following pattern of comparisons:

  (n-1), (n-2), (n-3) if n is static the original size of an array otherwise it will be (n-1) but n will get reduced at each element sort.
- Therefore, total comparisons(C) made is equal to,

  C=(n-1)+(n-2)+(n-3)+…+2+1

Best case complexity for selection sort is O(n log n), Average case complexity is O(n^2) and worst case complexity is O(n^2).

5.4 BUBBLE SORT

Bubble sort is treated as one of the simple and oldest sorting approach. Bubble sort almost performs sorting processes more or less in a similar fashion as selection sort. That is comparing each element in a list with immediate elements within the list and if the sort criteria is meet then swapping of the items is carried out. This particular technique is also called as sinking sort because here the smallest element in the sorted array lies at the bottom of array that is at index 0 and the largest element bubbles at the top of the array. The bubble sort differs from the selection sort in a way that in bubble sort the comparisons is carried out within the adjacent pairs of an array. The element that is result of the first iteration is probably not the sorted element in context with all elements of an array. Because in bubble sort the

sorting process begins by comparing first two elements (i and i +1) in a list, if the element on lower index is smaller than the other element then no swapping is done otherwise swap the elements. In next iteration the comparison is performed between i+2 and i+3 elements of the array. Similarly, next comparison pair will be i+3 and i+4 and will continue up to last element of array. The same pattern will repeat recursively till final sorted array is arrived.

Let's consider an array A[n] with 'n' elements to be sorted. Sorting on the basis of bubble sort, that is finding the smallest element from within the array. The array takes n-1 pass to bring sorted array. In first pass there are n-1 comparisons, in pass two n-2 and similarly in subsequent passes n-3,…2 and 1.

That means to find a smallest number in the list it requires scanning of all the 'n' elements in an array A[n] but in pairs and pair consists of adjacent array elements. However, it is not possible that after every scan the array will get any elements to be placed at its exact sorting location. The operational behavior of bubble sort is described with the help of an example provided as under.

Example

Elements	0	1	2	3	4
Data	25	11	16	19	9
1st Pass	25	11	16	19	9
	11	25	16	19	9
	11	16	25	19	9
	11	16	19	25	9
	11	16	19	9	25
2nd Pass	11	16	19	9	25
	11	16	19	9	25
	11	16	19	9	25
	11	16	9	19	25
3rd Pass	11	16	9	19	25
	11	16	9	19	25
	11	9	16	19	25
4th Pass	11	9	16	19	25
Sorted	9	11	16	19	25

Algorithm

BUBBLE (DATA, N)

1. Repeat Steps 2 and 3 for K=1 to N-1
2. Set PTR = 1 [Initializes the pass pointer PTR]
3. Repeat while PTR<= N-K [Execute pass]
   a) If DATA[PTR] > DATA[PTR+1] then
      i. Interchange DATA[PTR] and DATA[PTR+1]
      [End of If structure]
   b) Set PTR = PTR+1
   [End of inner loop]
   [End of step 1 outer loop]
4. Exit

Analysis: It is considered as an algorithm that is data sensitive sorting approach. The number of iterations that is needed to sort the given list of items may be any value between 1 and (n-1) and the extent of comparisons

NOTES

required is (n-1). The bubble sort will encounter its worst case scenario when the list to be sorted is in reverse order.

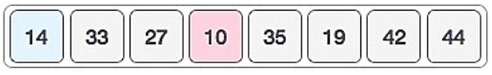
Best case complexity for bubble sort is O(n), Average case complexity is O(n^2) and worst case complexity is O(n^2).

5.5 SEQUENTIAL SORTING

A sorting algorithm is referred as an algorithm that puts the elements of a list in a certain specific order. Principally, the term 'Sequential Sorting' can be defined as a process of arranging elements in a group in a particular order, i.e., ascending order, descending order, numerical order, lexicographical order, alphabetic order, etc. The most frequently used orders are the numerical order and the lexicographical order. Further, in the sequential sorting, the input data is generally stored in the form of an array, which allows random access rather than a list, specifically it only allows sequential access to the data. The following example illustrates the sequential sorting.

For example, consider the following depicted array.

For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.



So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.

For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

After two iterations, two least values are positioned at the beginning in the list in a sequential sorted ascending manner.

The same process or methodology is applied to the rest of the items in the array.

NOTES

Check Your Progress

1. How does comparison based sorting approach sort data?
2. What is Brute force defined as?
3. How is the selection of a given number validated?

5.6 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. Comparison based sorting approach sorts data by comparing the data values
2. Brute force is defined as a type of problem solving approach wherein a problem is solution is directly based on the problem statement or the problem definition that is provided.
3. The selection of a given number is validated by performing comparisons of the current element with its adjacent elements in the list.

5.7 SUMMARY

- Algorithm describes the stepwise solution required to solve a particular problem in a specific problem domain.
- Empirical analysis of any algorithm deals with the critical analysis of an algorithm that is how fast it works, how much memory it is going to occupy and what attributes are incurred therein that causes increased lagging in programme execution.
- Sorting a problem is an example to understand empirical analysis of an algorithm designed to perform sort.
- Brute force is defined as a type of problem solving approach wherein a problem is solution is directly based on the problem statement or the problem definition that is provided. It is considered as the easiest approach to adopt and is also very useful when problem domain is not that much complex.
- Selection sort works on the basic principle concept to sort the given array of data by either selecting largest or smallest item/number from the given array.
- The selection of the smallest/largest number is carried out by scanning all the length of the array that means after scanning all the elements of the array to locate the desired element to be notified as sorted.
- Bubble sort is treated as one of the simple and oldest sorting approach. Bubble sort almost performs sorting processes more or less in a similar fashion as selection sort.
- That is comparing each element in a list with immediate elements within the list and if the sort criteria is meet then swapping of the items is carried out.

- A sorting algorithm is referred as an algorithm that puts the elements of a list in a certain specific order. Principally, the term 'Sequential Sorting' can be defined as a process of arranging elements in a group in a particular order, i.e., ascending order, descending order, numerical order, lexicographical order, alphabetic order, etc.

5.8 KEY WORDS

- **Brute Force:** It is defined as a type of problem solving approach wherein a problem is solution is directly based on the problem statement or the problem definition that is provided.
- **Bubble Sort:** It is treated as one of the simple and oldest sorting approach. Bubble sort almost performs sorting processes more or less in a similar fashion as selection sort.

5.9 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. What is the nature of Brute force?
2. Discuss the meaning and function of selection sort.
3. What is sequential sort?

Long Answer Questions

1. Differentiate between bubble sort and selection sort in detail.
2. "Instead of using brute force approach to solve above problem if recursive approach is used the complexity of the problem is reduced to O(log(n)) because $a^n = a^{n/2} * a^{n/2}$." Explain.
3. If in case of bubble sort if the size of the array is n the total number of comparisons sought to is O(n^2), then on what factors does the number of swaps depends on? Discuss in detail.

5.10 FURTHER READINGS

Levitin, Anany. *Introduction to Design and Analysis of Algorithms*. Delhi: Pearson Education.

Ellis Horowitz, S. Sahani and Rajasekaran, *Fundamentals of Computer Algorithms.* Delhi: Galgotia Publications.

Goodrich, M T and R. Tomassia. *Algorithm Design: Foundations, Analysis and Internet Examples*. Delhi: John wiley and Sons.

NOTES

UNIT 6 CLOSEST PAIR AND COVEX-HULL PROBLEMS

Structure

6.0 INTRODUCTION

In the field of computer science and mathematics, we often come across various problems which are quite complex, and solving such problems is a difficult task. Designing a solution for those problems which theoretically can be solved algorithmically is quite tough. Hence, in order to solve such problems, many new techniques and algorithms have been developed, out of which divide-and-conquer is an efficient one.

The divide-and-conquer technique solves a problem by breaking a large problem that is difficult to solve into sub-problems, solve these sub-problems recursively and then combine the answers. This unit discusses algorithms such as binary search, modular exponentiation, quick sort, and merge sort which are based on the divide-and-conquer technique. It also gives a brief comparison of various algorithms in terms of their time complexities.

6.1 OBJECTIVES

After going through this unit, you will be able to:

- Discuss divide and conquer
- Understand Strassens Matrix Multiplication
- Analyze the concept of exponentiation

NOTES

6.2 DIVIDE AND CONQUER

The **divide-and-conquer technique** is one of the widely used technique to develop algorithms for problems which can be divided into sub-problems (smaller in size but similar to the actual problem) so that they can be solved efficiently. The technique follows a top-down approach. To solve the problem it recursively divides the problem into number of sub-problems, to the extent where they cannot be sub-divided any further into more sub-problems. It then solves the sub-problems to find solutions that are then combined together to form a solution to the actual problem.

6.2.1 General Strategy

Some of the algorithms based on the divide and conquer technique are sorting, multiplying large numbers, syntactic analysis, etc. For example, consider the merge sort algorithm that uses the divide and conquer technique. The algorithm comprises three steps, which are as follows:

Step 1: Divides the n-element list, into two sub-lists of n/2 elements each, such that both the sub-lists hold half of the element in the list.

Step 2: Recursively sort the sub-lists using merge sort.

Step 3: Merge the sorted sub-lists to generate the sorted list.

Note that merging of sub-lists starts, only when the length of sorted sub-lists (through recursive application) reaches to 1. At this point, two sub-lists each of length 1 are merged (combined) by placing all the elements of the list in sorted order.

6.3 EXPONENTIATION

One of the simplest examples of divide-and-conquer technique is the **exponentiation algorithm**. This algorithm is used for fast computation of large powers of a given number. It works recursively and computes x^n for a positive integer n, as follows:

$$\begin{cases} x^n = x, & \text{if } n = 1 \\ (x^2)^{n/2}, & \text{if } n \text{ is even} \\ x.(x^2)^{(n-1)/2} & \text{if } n \text{ is odd} \end{cases}$$

In this algorithm only $O(\log n)$ multiplications are used; therefore, the computation of x^n becomes faster.

The modular exponentiation, that is, $x^b \bmod n$, for very large b and n, can be computed using same technique. Modular exponentiation is useful in computer science, especially in the field of cryptography. Let the binary representation of b be $(b_m, b_{m-1}, b_{m-2},..., b_1, b_0)$ where b_m is the most significant

bit and b_0 is the least significant bit. Then $x^b \bmod n$ can be computed by the Algorithm 6.1.

Algorithm 6.1: Modular Exponentiation

NOTES

```
modular_exponentiation(x,b,n)
 1.   Set c=0
 2.   Set res=1
      //let(bm, bm-1, bm-2…..,b1, b0) be the binary
      //representation of b
 3.   for i=m downto 0
 4.   {
 5.       Set c=2*c
 6.      Set res=(res*res) mod n
 7.      if (bi=1)
 8.       {
 9.         Set c=c+1
10.         Set res=(res*x) mod n
11.       }
12.  }
13.  return res
```

This algorithm computes $x^c \bmod n$ where c is repeatedly increased by squaring in each iteration (from b = 0 to m) and finally, we get $x^b \bmod n$. Therefore, this algorithm is also named as **repeated squaring** algorithm.

To understand this algorithm, consider the modular expression $5^{650} \bmod 765$. Here, x = 5, b = 650, and n = 765. If we first calculate 5^{650} and then take the remainder when divided by 765, the calculation would take a lot of time. Even using a more effective method, that is, to square 5 and then take the remainder when divided by 765, will also take too much time. If the values of b and n are smaller, then it is possible to calculate the result with less difficulty but if the values of b and n are too large, as in the case of cryptography then it is efficient to compute the result using the repeated squaring algorithm. Computing the result using repeated squaring algorithm will take less time and less storage space.

The binary representation of b = 650 is 1010001010, that is, 10 bits, which means m = 9. Figure 6.1 depicts the sequence of values modulo 765 during each iteration. c given in the table depicts the sequence of exponents used.

i	9	8	7	6	5	4	3	2	1	0
b_i	1	0	1	0	0	0	1	0	1	0
c	1	2	5	10	20	40	81	162	325	650
res	5	25	65	400	115	220	260	280	320	655

Fig. 6.1 *Steps in Computing Modular Exponentiation*

Step 1: Initially, the value of c is 0 and the value of res is 1 according to step 1 and 2 of the algorithm. During the first iteration (that is, i = 9), we get c = 0 and res = 1 from Step 6 and 7 of algorithm. The condition in the Step 8 holds true (as $b_9 = 1$), which results in:

`c` = 0 + 1 = 1 (from Step 10 of algorithm)

`res` = (1*5) mod 765 = 5 (from Step 11 of algorithm)

Step 2: In the second iteration (that is, `i` = 8), we get `c` = 1 * 2 = 2, and `res` = (5 * 5) mod 765 = 25 from Step 6 and 7 of algorithm, respectively. Now, since the condition in the Step 8 of the algorithm evaluates to false (as b_8 = 0), the value of `c` and `res` remains 2 and 25, respectively.

Step 3: In the third iteration (that is, `i` = 7), we get `c` = 2 * 2 = 4, and `res` = (25 * 25) mod 765 = 625 from Step 6 and 7 of algorithm, respectively. As b_7 = 1, the condition in Step 8 of algorithm evaluates to true. This makes the value of `c` = 4 + 1 = 5 and `res` = (625 * 5) mod 765 = 65

Proceeding in this manner through each iteration, we get the final value of `res` = 655. That is, 5^{650} mod 765 = 655.

6.4 BINARY SEARCH

The binary search technique is used to search for a particular data item in a sorted (in ascending or descending order) array. In this technique, the value to be searched (say, item) is compared with the middle element of the array. If item is equal to the middle element, then search is successful. If item is smaller than the middle value, item is searched in the segment of the array before the middle element. However, if item is greater than the middle value, item is searched in the array segment after the middle element. This process is repeated until the value is found or the array segment is reduced to a single element that is not equal to item.

At every stage of the binary search technique, the array is reduced to a smaller segment. It searches a particular data item in the lowest possible number of comparisons. Hence, the binary search technique is used for larger and sorted arrays, as it is faster as compared to linear search. For example, consider an array `ARR` shown in Figure 6.2.

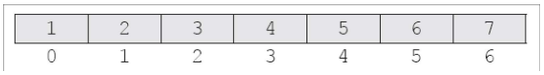
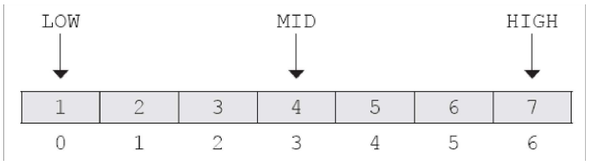


Fig. 6.2 *The Array ARR*

To search an `item` (say, 7) using binary search in the array `ARR` with `size`=7, these steps are performed.

1. Initially, set `LOW`=0 and `HIGH=size-1`. The middle of the array is determined using the formula `MID=(LOW+ HIGH)/2`, that is, `MID=(0+6)/2`, which is equal to 3. Thus, `ARR [MID]`=4.



NOTES

2. Since the value stored at `ARR[3]` is less than the value to be searched, that is 7, the search process is now restricted from `ARR[4]` to `ARR[6]`. Now `LOW` is 4 and `HIGH` is 6. The middle element of this segment of the array is calculated as `MID`=(4+6)/2, that is, 5. Thus, `ARR[MID]`=6.

3. The value stored at `ARR[5]` is less than the value to be searched, hence the search process begins from the subscript 6. As `ARR[6]` is the last element, the `item` to be searched is compared with this value. Since `ARR[6]` is the value to be searched, the search is successful.

Algorithm 6.2: Binary Search

```
binary_search(ARR,size,item)
//ARR is the list in which the element is to be searched
 1. Set LOW=0
 2. Set HIGH=size-1
 3. while (LOW = HIGH)
 4. {
 5.       Set MID=(LOW + HIGH)/2
 6.       If (item=ARR[MID])
 7.          return MID
 8.       Else If (item<ARR[MID])
 9.             Set HIGH=MID-1
10.       Else
11.          Set LOW=MID+1
12. }
13. return  -1    //item not found in the list
```

Analysis of Binary Search

In each iteration, binary search algorithm reduces the array to one half. Therefore, for an array containing n elements, there will be $\log_2 n$ iterations. Thus, the complexity of binary search algorithm is $O(\log_2 n)$. This complexity will be same irrespective of the position of the element, even if the element is not present in the list.

Check Your Progress

1. Which technique is used to develop algorithms for problems which can be divided into sub-problems?
2. Name a few algorithms based on the divide and conquer technique.
3. What is the binary search technique is used for?

6.5 QUICK SORT

NOTES

algorithm is based on the fact that it is easier and faster to sort two smaller arrays than one larger array. Thus, it follows the principle of *divide-and-conquer*. Quick sort algorithm first picks up a partitioning element, called **pivot**, that divides the list into two sub lists such that all the elements in the left sub list are smaller than the pivot, and all the elements in the right sub list are greater than the pivot. Once the given list is partitioned into two sub lists, these two sub lists are sorted separately. The same process is applied to sort the elements of left and right sub lists. This process is repeated recursively until each sub list contains not more than one element.

As we have discussed, the main task in quick sort is to find the pivot that partitions the given list into two halves so that the pivot is placed at its appropriate location in the array. The choice of pivot has a significant effect on the efficiency of quick sort algorithm. The simplest way is to choose the first element as pivot. However, first element is not always a good choice, especially if the given list is already or nearly ordered. For better efficiency, the middle element is chosen as pivot. For simplicity, we will take the first element as pivot.

The steps involved in quick sort algorithm are as follows:

1. Initially, three variables `pivot`, `beg` and `end` are taken, such that both `pivot` and `beg` refer to the 0^{th} position, and `end` refers to $(n-1)^{th}$ position in the list.
2. Starting with the element referred to by `end`, the array is scanned from right to left, and each element on the way is compared with the element referred to by `pivot`. If the element referred to by `pivot` is greater than the element referred to by `end`, they are swapped and Step 3 is performed. Otherwise, `end` is decremented by 1 and Step 2 is continued.
3. Starting with the element referred to by `beg`, the array is scanned from left to right, and each element on the way is compared with the element referred to by `pivot`. If the element referred to by `pivot` is smaller than the element referred to by `end`, they are swapped and Step 2 is performed. Otherwise, `beg` is incremented by 1 and Step 3 is continued.

The first pass terminates when `pivot`, `beg` and `end` all refer to the same array element. This indicates that the element referred to by `pivot` is placed at its final position. The elements to the left of this element are smaller than this element and elements to its right are greater.

To understand the quick sort algorithm, consider an unsorted array shown in Figure 6.3. The steps to sort the values stored in the array in ascending order using quick sort are given here.

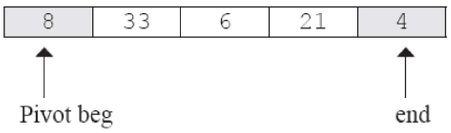
8	33	6	21	4

Fig. 6.3 *Unsorted Array*

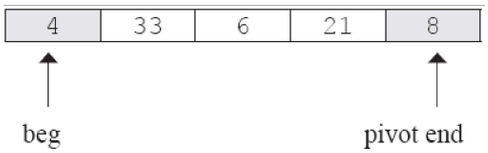
NOTES

First Pass:

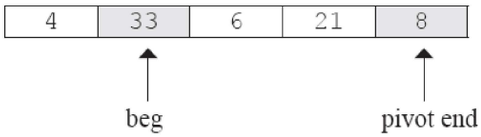
1. Initially, the index 0 in the list is chosen as the `pivot`, and the index variables `beg` and `end` are initialised with index 0 and `n-1`, respectively.



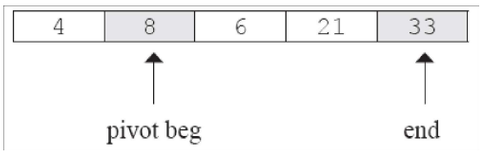
2. The scanning of elements is started from the end of the list. `ARR[pivot]` (that is, 8) is greater than `ARR[end]` (that is, 4). Therefore, they are swapped.



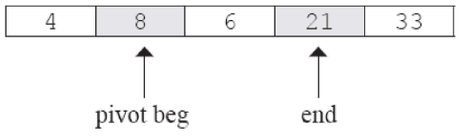
3. Now, the scanning of elements is started from the beginning of the list. Since `ARR[pivot]` (that is, 8) is greater than `ARR[beg]` (that is 33), therefore `beg` is incremented by 1, and the list remains unchanged.



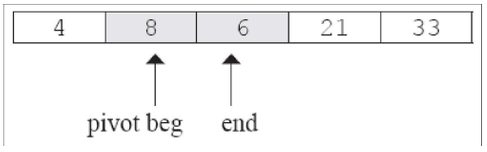
4. Next, the element `ARR[pivot]` is smaller than `ARR[beg]`, they are swapped.



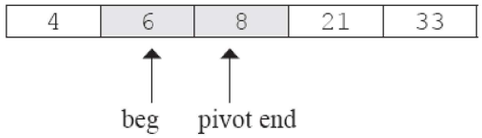
5. Again, the list is scanned from right to left. Since, `ARR[pivot]` is smaller than `ARR[end]`, therefore the value of `end` is decremented by 1, and the list remains unchanged.



6. Next, the element `ARR[pivot]` is smaller than `ARR[end]`, the value of `end` is decremented by 1, and the list remains unchanged.

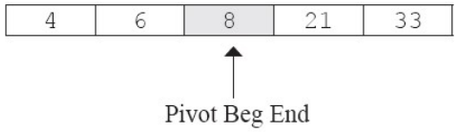


7. Now, `ARR[pivot]` is greater than `ARR[end]`, they are swapped.



NOTES

8. Now, the list is scanned from left to right. Since, `ARR[pivot]` is greater than `ARR[beg]`, value of `beg` is incremented by 1, and the list remains unchanged.



At this point, since the variables `pivot`, `beg` and `end` all refer to the same element, the first pass is terminated and the value 8 is placed at its appropriate position. The elements to its left are smaller than 8, and elements to its right are greater than 8. These two sub lists are again sorted using the same procedure.

Algorithm 6.3: Quick Sort

```
quick_sort(ARR,size,lb,ub)
 1. Set i=1     //i is a static integer variable
 2. If (lb<ub)
 3. {
 4.       Call splitarray(ARR,lb,ub) //returning an
                                     //integer value pivot
 5.       Print ARR after ith pass
 6.       Set i=i+1
 7.       Call quick_sort(ARR,size,lb,pivot - 1)
                  //recursive call to quick_sort() to
                  //sort left sub list
 8.       Call quick_sort(ARR,size,pivot + 1,ub);
                //recursive call to quick_sort()
                //to sort right sub list
 9. }
10. Else If (ub=size-1)
11.       Print "No. of passes: ", i
```

```
splitarray(ARR,lb,ub)
//spiltarray partitions the list into two sub lists such
//that the elements in left sub list are smaller than
//ARR[pivot], and elements in the right sub list are
//greater than ARR[pivot]
 1. Set flag=0
 2. Set beg=pivot=lb
 3. Set end=ub
 4. while (flag != 1)
 5. {
 6.   while (ARR[pivot] = ARR[end] AND pivot != end)
 7.          Set end=end-1
 8.   If (pivot=end)
 9.       Set flag=1
10.   Else
11.   {
12.          Set temp=ARR[pivot]
13.          Set ARR[pivot]=ARR[end]
14.          Set ARR[end]=temp
15.          Set pivot=end
```

NOTES

```
16.   }
17.   If (flag != 1)
18.   {
19.          while (ARR[pivot] = ARR[beg] AND pivot != beg)
20.             Set beg=beg+1
21.          If (pivot=beg)
22.             Set flag=1
23.          Else
24.          {
25.             Set temp=ARR[pivot]
26.             Set ARR[pivot]=ARR[beg]
27.             Set ARR[beg]=temp
28.             Set pivot=beg
29.          }
30.       }
31. }
32. return pivot
```

Analysis of Quick Sort

The quick sort algorithm gives worst case performance when the list is already sorted. In this case, the first element requires `n` comparisons to determine that it remains in the first position, second element requires `n-1` comparisons to determine that it remains in the second position, and so on. Therefore, total number of comparisons in this case is:

$$f(n) = n + (n-1) + \ldots + 3 + 2 + 1$$
$$= n(n+1)/2$$
$$= O(n^2)$$

Note that in the worst case the complexity of quick sort algorithm is equal to the complexity of bubble sort algorithm. In the best case when pivot is chosen in such a way that it partitions the list approximately in half, then there will be `log n` partitions. Each pass does at most `n` comparisons. Therefore, complexity of quick sort algorithm in this case is:

$$f(n) = n * \log n$$
$$= O(n \log n)$$

Randomized Quick Sort

The complexity of quick sort algorithm in worst case is $O(n^2)$ which is observed when the list is already sorted. In this case, the list is partitioned in such a way that only one element lies in one portion of the list and the remaining elements lie in the other portion of the list. As we are aware of the fact that the divide-and-conquer algorithm exhibits better performance when the splits are well balanced, the quick sort algorithm can be modified by using a randomizer in order to obtain good average-case performance for all inputs (see Algorithm 6.4).

The randomized version of quicksort is a better option to opt, when the inputs are large. In this algorithm, instead of choosing the first element as pivot, an element is selected randomly from the list. This element is also called as **randomizer**. Randomized algorithm uses random numbers, in addition to the inputs to solve a

NOTES

given problem. The random numbers are generated by a random number generator. Since the randomizer will generate different values with each execution, the output of the algorithm may vary for the same input data. The complexity of this algorithm is not affected by any input but is affected greatly by the random number chosen.

Algorithm 6.4: Randomized Quick Sort

```
randomized_quick_sort(ARR,size,lb,ub)
 1. Set i=1     //i is a static integer variable
 2. If (lb < ub)
 3. {
 4.       Call randomized_splitarray(ARR,lb,ub)
 5.       Print ARR after ith pass
 6.       Set i=i+1
 7.       Call randomized_quick_sort(ARR,size,lb,pivot-1)
          //recursive call to randomized quick sort()
 8.       Call randomized_quick_sort(ARR,size,pivot+1,ub)
             //recursive call to randomized_quick_sort()
 9.}
10.Else If (ub=size-1)
11.   Print "No. of passes: ", i

randomized_splitarray(ARR,lb,ub)
//randomized-_splitarray  ()  randomly  chooses  an  element
from
//the list and exchanges it with the first element and then
//calls the splitarray
 1. Set i=Random(lb,ub)
 2. exchange ARR[lb]=ARR[i]
 3. return splitarray(ARR,lb,ub)
```

```
splitarray (ARR,lb,ub)
 1. Set flag=0
 2. Set beg=pivot=lb
 3. Set end=ub
 4. while (flag != 1)
 5. {
 6.       while (ARR[pivot] = ARR[end] AND pivot != end)
 7.          Set end=end-1
 8.      If (pivot=end)
 9.       Set flag=1
10.       Else
11.       {
12.          Set temp=ARR[pivot]
13.          Set ARR[pivot]=ARR[end]
14.          Set ARR[end]=temp
15.          Set pivot=end
16.       }
17.       If (flag != 1)
18.       {
19.          while (ARR[pivot] = ARR[beg] AND pivot != beg)
20.             Set beg=beg+1
21.          If (pivot=beg)
22.             Set flag=1
```

NOTES

```
23.          Else
24.          {
25.             Set temp=ARR[pivot]
26.             Set ARR[pivot]=ARR[beg]
27.             Set ARR[beg]=temp
28.             Set pivot=beg
29.          }
30.       }
31. }
32. return pivot
```

6.6 MERGE SORT

Like quick sort, merge sort algorithm also follows the principle of *divide-and-conquer*. In this sorting, the list is first divided into two halves. The left and right sub lists obtained are recursively divided into two sub lists until each sub list contains not more than one element. The sub lists containing only one element do not require any sorting. Therefore, we start merging the sub lists of size one to obtain the sorted sub list of size two. Similarly, the sub lists of size two are then merged to obtain the sorted sub list of size four. This process is repeated until we get the final sorted array.

To understand the merge sort algorithm, consider an unsorted array shown in Figure 6.4. The steps to sort the values stored in the array in ascending order using merge sort are given here.

Fig. 6.4 *Unsorted Array*

1. Initially, `low`=0 and `high`=7, therefore, `mid`=(0+7)/2=3. Thus, the given list is divided into two halves from the 4th element. The sub lists are as follows:

2. The left sub list is considered first, and it is again divided into two sub lists. Now, `low`=0 and `high`=3, therefore, `mid`=(0+3)/2=1. Thus, the left sub list is divided into two halves from the 2nd element. The sub lists are as follows:

3. These two sub lists are again divided into sub lists such that all of them contain one element. Now the sub lists are as follows:

4. Since each sub list now contains one element, they are first merged to produce the two arrays of size 2. First, the sub lists containing the elements 18 and 13 are merged to give one sorted sub array, and the sub lists containing

the elements 5 and 20 are merged to give another sorted sub array. The two sorted sub arrays are as follows:

| 13 | 18 |

| 5 | 20 |

5. Now these two sub arrays are again merged to give the following sorted sub array of size 4.

| 5 | 13 | 18 | 20 |

6. After sorting the left half of the array, we perform the same steps for the right half. The sorted right half of the array is given below:

| 4 | 14 | 55 | 89 |

7. Finally, the left and right halves of the array are merged to give the sorted array as shown in Figure 6.5.

| 4 | 5 | 13 | 14 | 18 | 20 | 55 | 89 |

Fig. 6.5 *Final Sorted Array*

Algorithm 6.5: Merge Sort

```
merge_sort(ARR,low,high)
 1. If (low < high)
 2. {
 3.   Set mid=(low+high)/2
 4.   Call merge_sort(ARR,low,mid)//calling merge_sort
                                //recursively for left
                                //sub list
 5.   Call merge_sort(ARR,mid+1,high) //calling merge_sort
                                      //for right sub list
 6.   Call merging(ARR,low,mid,mid+1,high)
 7. }
merging(ARR,ll,lr,ul,ur)
//merging() merges the two sub arrays to produce a sorted
//array named merged. ll and ul are the lower bounds of
//left and right sub list, respectively. ul and ur the upper
//bounds of left and right sub list,respectively.
 1. Set i=ll
 2. Set j=ul
 3. Set k=ll
 4. while(i = lr AND j = ur)
5. {
 6.       If(ARR[i] = ARR[j])
 7.      {
 8.          Set merged[k]=ARR[i]
 9.          Set i=i+1
10.       }
11.       Else
12.       {
13.          Set merged[k]=ARR[j]
14.          Set j=j+1
15.       Set k= k + 1
16.       }
17. }
18. If(i = lr)
19. {
20.       while(i = lr)
21.       {
```

NOTES

```
22.       Set merged[k]=ARR[i]
23.       Set i=i+1
24.       Set k=k+1
25.    }
26. }
27. If(j = ur)
28. {
29.       while(j = ur)
30.    {
31.       Set merged[k]=ARR[j]
32.       Set j=j+1
33.       Set k=k+1
34.    }
35. }
36. Set k=ll
37. while (k = ur)
38. {
39.    Set ARR[k]=merged[k]
40.    Set k=k+1
41. }
```

Analysis of Merge Sort

In the first pass of merge sort algorithm, the given array is divided into two halves and each half is sorted separately. In each of the recursive calls to the `merge_sort()`, one for left half and one for right half, the array is further divided into two halves, thereby, resulting in four segments of the array. Thus, in each pass, the number of segments of the array gets doubled until each segment contains not more than one element. Therefore, the total number of divisions is `log n`. Moreover, in any pass, at most `n` comparisons are required. Hence, the complexity of the merge sort algorithm is `O(n log n)`.

6.7 STRASSENS MATRIX MULTIPLICATION

Volker Strassen is a German mathematician born in 1936. His algorithm for matrix multiplication is still one of the main methods that outperforms the general matrix multiplication algorithm.

Assume that `X` and `Y` are two `n x n` matrices. We need to determine the matrix `Z` as the product of matrix `X` and `Y`, that is `Z=X x Y`, and `Z` is also an `n x n` matrix. The conventional method to computer the element at position `Z[i, j]` is as follows:

$$Z(i,j) = \sum_{1 \le k \le n} X(i,k)Y(k,j) \qquad \ldots (6.1)$$

for all `i` and `j` between 1 and `n`.

Since `Z` has n^2 elements and each element is computed using n multiplications, the time for the resulting matrix multiplication algorithm is $\theta(n^3)$.

Another method to calculate the product of two n x n matrices is given to use the divide-and-conquer technique. For simplicity, we assume that $n=2^k$ where `k` is a nonnegative integer. Using the divide-and-conquer technique, the `X` and `Y`

matrices of size $n \times n$ are recursively divided into sub-matrices of size $\frac{n}{2} \times \frac{n}{2}$ until each matrix becomes a 2×2 matrix. For example, Figure 6.6 shows the partitioning of 4×4 matrices into four blocks.

$$X = \left[\begin{array}{cc|cc} x_{11} & x_{12} & x_{13} & x_{14} \\ x_{21} & x_{22} & x_{23} & x_{24} \\ \hline x_{31} & x_{32} & x_{33} & x_{34} \\ x_{41} & x_{42} & x_{43} & x_{44} \end{array}\right] \qquad Y = \left[\begin{array}{cc|cc} y_{11} & y_{12} & y_{13} & y_{14} \\ y_{21} & y_{22} & y_{23} & y_{24} \\ \hline y_{31} & y_{32} & y_{33} & y_{34} \\ y_{41} & y_{42} & y_{43} & y_{44} \end{array}\right]$$

Fig. 6.6 *Partitioning 4×4 Matrices into Four Submatrices*

Now we can write matrices X and Y each with elements as follows:

$$X = \begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix} \qquad Y = \begin{bmatrix} Y_{11} & Y_{12} \\ Y_{21} & Y_{22} \end{bmatrix}$$

Where,

$$X_{11} = \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix} \qquad Y_{11} = \begin{bmatrix} y_{11} & y_{12} \\ y_{21} & y_{22} \end{bmatrix}$$

$$X_{12} = \begin{bmatrix} x_{13} & x_{14} \\ x_{23} & x_{24} \end{bmatrix} \qquad Y_{12} = \begin{bmatrix} y_{13} & y_{14} \\ y_{23} & y_{24} \end{bmatrix}$$

$$X_{21} = \begin{bmatrix} x_{31} & x_{32} \\ x_{41} & x_{42} \end{bmatrix} \qquad Y_{21} = \begin{bmatrix} y_{31} & y_{32} \\ y_{32} & y_{42} \end{bmatrix}$$

$$X_{22} = \begin{bmatrix} x_{33} & x_{34} \\ x_{43} & x_{44} \end{bmatrix} \qquad Y_{22} = \begin{bmatrix} y_{33} & y_{34} \\ y_{43} & y_{44} \end{bmatrix}$$

Now, the product of XY can be obtained by the using Equation 6.1 for the product of 2×2 matrices as follows:

$$\begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix} \begin{bmatrix} Y_{11} & Y_{12} \\ Y_{21} & Y_{22} \end{bmatrix} = \begin{bmatrix} Z_{11} & Z_{12} \\ Z_{21} & Z_{22} \end{bmatrix} \qquad \ldots (6.2)$$

where,

$$\begin{aligned} Z_{11} &= X_{11}Y_{11} + X_{12}Y_{21} \\ Z_{12} &= X_{11}Y_{12} + X_{12}Y_{22} \\ Z_{21} &= X_{21}Y_{11} + X_{22}Y_{21} \\ Z_{22} &= X_{21}Y_{12} + X_{22}Y_{22} \end{aligned} \qquad \ldots (6.3)$$

Note: If n is not a power of two we can make it a power of two by adding rows and columns of zeros to both X and Y matrices.

NOTES

For n=2, the matrix Z is obtained by directly multiplying the elements of X and Y. However, for n>2, the matrices are recursively divided into $\frac{n}{2} \times \frac{n}{2}$ sub-matrices, and multiplication and addition operations are applied to them.

If the matrices are of size 4x4, then to compute XY using Equation 6.3, we need eight multiplications and four additions of $\frac{n}{2} \times \frac{n}{2}$ matrices. Since two matrices of size $\frac{n}{2} \times \frac{n}{2}$ can be added in cn^2 time, where c is a constant, the overall computing time T(n) of the divide-and-conquer technique is as follows:

$$T(n) = \begin{cases} 8T(n/2) + cn^2 & n > 2 \\ b & n \geq 2 \end{cases}$$

Here, b is also a constant.

Thus, the time complexity of divide-and-conquer technique is also $O(n^3)$, which is same as the conventional approach of matrix multiplication. Thus, we need another approach which has lesser time complexity. As we know that matrix addition $O(n^2)$ is less expensive as compared to the matrix multiplication $O(n^3)$ therefore we can have more addition operations and fewer multiplication operations by reformulating the equations for Z_{ij}.

Volker Strassen gave the Strassen's matrix multiplication algorithm in 1969. This algorithm uses 18 additions and 7 multiplications to compute Z_{ij}. In this method, initially the seven $\frac{n}{2} \times \frac{n}{2}$ matrices $P_1, P_2, P_3, P_4, P_5, P_6, P_7$ are computed using the formulas given in the Equation 2.4 followed by computing the Z_{ij} using the formulas given in Equation 2.5.

$$\begin{aligned}
P_1 &= (X_{11} + X_{22})(X_{11}+Y_{22}) \\
P_2 &= (X_{21} + X_{22})Y_{11} \\
P_3 &= X_{11}(Y_{12}-Y_{22}) \\
P_4 &= X_{22}(Y_{21}+Y_{11}) \\
P_5 &= (X_{11} + X_{12})Y_{22} \qquad \ldots (6.4) \\
P_6 &= (X_{21} - X_{11})(Y_{11}+Y_{12}) \\
P_7 &= (X_{12} + X_{22})(Y_{21}+Y_{22})
\end{aligned}$$

$$\begin{aligned}
Z_{11} &= P_1 + P_4 - P_5 + P_7 \\
Z_{12} &= P_3 + P_5 \qquad \ldots (6.5) \\
Z_{21} &= P_2 + P_4 \\
Z_{22} &= P_1 + P_3 - P_2 + P_6
\end{aligned}$$

As we can see, to compute P_1, P_2, P_3, P_4, P_5, P_6, and P_7 seven matrix multiplications and 10 matrix additions or subtractions are required, and to compute Z_{ij} 8 additions or subtractions are required. The time complexity T(n) of this technique is as follows:

$$T(n) = \begin{cases} 7T(n/2) + an^2 & n > 2 \\ b & n \geq 2 \end{cases} \quad \ldots(6.6)$$

Where, a and b are constants. Operating on this formula gives:

$$T(n) = an^2[1 + 7/4 + (7/4)^2 + \ldots + (7/4)^{k-1}] + 7^k T(1)$$

$$\leq cn^2(7/4)^{\log_2 n} + 7^{\log_2 n} \qquad c \text{ is a constant}$$

$$= cn^{\log_2 4 + \log_2 7 - \log_2 4} + n^{\log_2 7}$$

$$= \alpha(n^{\log_2 7}) \approx O(n^{2.81}), \text{ which is less than } O(n^3)$$

NOTES

Check Your Progress

4. What is the complexity of quick soft algorithm in worst case?
5. What principle does merge sort algorithm follows?

6.8 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. The divide-and-conquer technique is one of the widely used technique to develop algorithms for problems which can be divided into sub-problems.
2. Some of the algorithms based on the divide and conquer technique are sorting, multiplying large numbers, syntactic analysis, etc.
3. The binary search technique is used to search for a particular data item in a sorted array.
4. The complexity of quick sort algorithm in worst case is $O(n^2)$.
5. Like quick sort, merge sort algorithm also follows the principle of divide-and- conquer.

6.9 SUMMARY

- The divide-and-conquer technique is one of the widely used technique to develop algorithms for problems which can be divided into sub-problems.
- Some of the algorithms based on the divide and conquer technique are sorting, multiplying large numbers, syntactic analysis, etc.
- One of the simplest examples of divide-and-conquer technique is the exponentiation algorithm.
- The binary search technique is used to search for a particular data item in a sorted (in ascending or descending order) array.

NOTES

- At every stage of the binary search technique, the array is reduced to a smaller segment.
- In each iteration, binary search algorithm reduces the array to one half.
- The main task in quick sort is to find the pivot that partitions the given list into two halves so that the pivot is placed at its appropriate location in the array.
- The quick sort algorithm gives worst case performance when the list is already sorted.
- The complexity of quick sort algorithm in worst case is $O(n^2)$ which is observed when the list is already sorted.
- The randomized version of quicksort is a better option to opt, when the inputs are large.
- Like quick sort, merge sort algorithm also follows the principle of *divide-and- conquer*.
- In the first pass of merge sort algorithm, the given array is divided into two halves and each half is sorted separately.
- Volker Strassen is a German mathematician born in 1936. His algorithm for matrix multiplication is still one of the main methods that outperforms the general matrix multiplication algorithm.
- Thus, the time complexity of divide-and-conquer technique is also $O(n^3)$, which is same as the conventional approach of matrix multiplication.

6.10 KEY WORDS

- **Quick Sort:** It is an efficient sorting algorithm, serving as a systematic method for placing the elements of an array in order.
- **Merge Sort:** It is an efficient, general-purpose, comparison-based sorting algorithm.

6.11 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Question

1. What do you mean by divide and conquer?
2. Write a short note on Strassens matrix multiplication.
3. Analyze the concept of exponentiation.

NOTES

Long Answer Questions

1. "The **divide-and-conquer technique** is one of the widely used technique to develop algorithms for problems which can be divided into sub-problems (smaller in size but similar to the actual problem) so that they can be solved efficiently." Explain with the help of an example.
2. "The binary search technique is used to search for a particular data item in a sorted (in ascending or descending order) array." Discuss.
3. What is general strategy? Discuss the steps of general strategy.

6.12 FURTHER READINGS

Levitin, Anany. *Introduction to Design and Analysis of Algorithms*. Delhi: Pearson Education.

Ellis Horowitz, S. Sahani and Rajasekaran, *Fundamentals of Computer Algorithms.* Delhi: Galgotia Publications.

Goodrich, M T and R. Tomassia. *Algorithm Design: Foundations, Analysis and Internet Examples*. Delhi: John wiley and Sons.

UNIT 7 GENERAL METHOD

NOTES

Structure

7.0 INTRODUCTION

In mathematics, any of the positive integers that occurs as a coefficient in the binomial theorem is called a binomial coefficient. It can also be said as the number of ways of picking unordered outcomes from possibilities, also known as a combination or combinatorial number. If there are short paths in a weighted graph, the Floyd-Warshall algorithm is used to find the shortest paths with positive or negative edge weights.

This unit will discuss about how to compute a binomial coefficient, warshalls and Floyds, and knapsack problems.

7.1 OBJECTIVES

After going through this unit, you will be able to:

- Understand how to compute a binomial coefficient
- Analyze Floyd and Warshall algorithms
- Discuss optimal binary search trees
- Explain knapsack problems

7.2 COMPUTING A BINOMIAL COEFFICIENT

A **coefficient** is defined as a multiplicative factor that is associated with any individual token of a mathematical expression. This expression can be simply a term or binomial, polynomial, or any series. The nature or state of the coefficient

can be either a number or expression like in case of binomial expressions. Let's take an example to understand this by putting the following mathematical expression:

$2x^2 + 3x^3 + 4xy + 10 + z$

In the above expression 2, 3 and 4 are coefficients. In addition to them 10 is also a coefficient known as constant coefficient. Variable z has no coefficient or 1.

Let's consider the following binomial theorem:

$(X + Y)^2 = x^2 + 2xy + y^3$

The coefficient in above expression for x^2 or 2xy or y^3 that is 1, 2 and 1 respectively are called as binomial coefficients. In general the coefficient 'C' in any term with form as Cx^by^c is known as the binomial coefficient.

The binomial coefficients for any polynomial expression in the form $(x+y)^n$ can be obtained by expanding the said expression using binomial theorem. The expansion will be performed as follows:

$$(x+y)^n = \binom{n}{0}x^n y^0 + \binom{n}{1}x^{n-1}y^1 + \binom{n}{2}x^{n-2}y^2 + \cdots + \binom{n}{n-1}x^1 y^{n-1} + \binom{n}{n}x^0 y^n,$$

Where, each $\binom{n}{k}$ is a particular positive integer value and is also called as binomial coefficient. In order to compute binomial coefficient of any term in an expression the following formula is used:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

In a mathematical representation or expression any positive integer/number that occurs as a coefficient in the binomial theorem is called as binomial coefficient.

Computing Binomial Coefficients

In order to compute binomial coefficients of any expression say $(x+y)^n$, the main expression is divided into sub problems and the solution of the main problem is expressed in terms of the solutions obtained for small sub problems. The most favorable approach used to compute binomial coefficients of any expression is dynamic programming. Dynamic programming is best suited approach for optimization problems. In dynamic programming the solution of the problem is arrived by using multistage optimized decisions. In dynamic programming unlike divide and conquer approach where each sub-problem is solved recursively are solved only once and the obtained solution is preserved in a table. In general the dynamic programming requires the following steps to arrive at the solution of problem:

1. Problem is divided into overlapping sub-problems.
2. Sub-problem can be represented by a table.

NOTES

3. Principle of optimality, recursive relation between smaller and larger problems.

Computing binomial coefficients is non optimization problem but can be solved using dynamic programming.

As mentioned above that binomial coefficients can be represented by $\binom{n}{k}$ or $C(n, k)$ and are used to denote the coefficients of binomial expression $(a + b)^n$ as:

$$(a + b)^n = C(n, 0)a^n + ... + C(n, k)a^{n-k}b^k + ... + C(n, n)b^n$$

The recursive relation is defined by the prior power

$$C(n, k) = C(n\text{-}1, k\text{-}1) + C(n\text{-}1, k) \text{ for } n > k > 0$$

IC $C(n, 0) = C(n, n) = 1$

By using Dynamic algorithm a table with nxk dimensions(rows, colums), with the first column and diagonal filled out using the IC.

Construct the table

		0	1	2	...	k-1	k
	0	1					
	1	1	1				
	2	1	2	1			
n	.						
	.						
	.						
	k	1					1
	.						
	.						
	.						
	n-1	1				$C(n\text{-}1, k\text{-}1)$	
	n	1					$C(n, k)$

Each every iteration the particular entry in the table is filled out row by row using a recursive approach.

Algorithm *Binomial*(n, k)

for i ? 0 **to** n **do** // fill out the table row wise

for i = 0 **to** min(i, k) **do**

if j==0 or j==i **then** $C[i, j]$? 1 // IC

else $C[i, j]$? $C[i\text{-}1, j\text{-}1] + C[i\text{-}1, j]$ // recursive relation

return $C[n, k]$

7.3 FLOYD-WARSHALL ALGORITHM

NOTES

All pairs-shortest path problem is to find the shortest path between all pairs of vertices in a graph G=(V,E). For example, if we are given a graph consisting of five cities, say A, B, C, D, and E, then the aim is to find the shortest path between all pairs of vertices such as from A to B, B to C, A to E, E to D, and so on. The problem is efficiently solved by the Floyd-Warshall algorithm that we will discuss in this section. Another way is to apply the single source shortest path algorithm on all vertices, which is explained in the next section.

7.3.1 The Floyd-Warshall Algorithm

The Floyd-Warshall algorithm is used to find the shortest path between all pairs of vertices in a directed graph G=(V,E). This algorithm uses the dynamic programming approach in a different manner. This algorithm defines the structure of the shortest path by considering the 'intermediate' vertices of the shortest path, where the intermediate vertex of the path $p=\{v_1, v_2, v_3, \ldots, v_1\}$ can be any vertex other than v_1 and v_k.

Let G be the graph and V be the vertex set of graph, where V={1, 2, 3, 4, …, n}. For any pair of vertices i, jõV, while considering all paths from i to j which have a number of intermediate vertices, say belongs to the subset of V, which is {1, 2, 3, 4, …, k} for some vertex k and let us consider p be the minimum weighted path among all. The Floyd-Warshall algorithm establishes a notable relationship between path p and other shortest paths from i to j with the intermediate vertices from set {1, 2, 3, 4, …, k-1}.

If k is not an intermediate vertex of path p, then all the intermediate vertices of path p belongs to the set {1, 2, 3, 4, …, k-1}. Hence, the shortest path from vertex i to j having all intermediate vertices in set {1, 2, 3, 4, …, k}. If k is an intermediate vertex of path p, then we will break path p into two ways, say p_1 and p_2, such that all intermediate vertices in set {1, 2, 3, 4, …, k-1}.

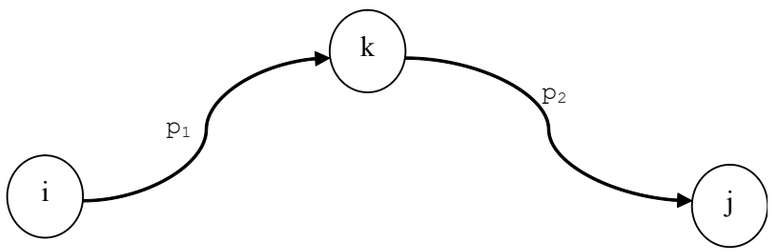


Fig. 7.1 *Intermediate Vertex*

From the above discussion, a recursive solution for the estimation of the shortest path can be made as follows. Let $d_{ij}^{(k)}$ be the weight of the shortest path from vertex i to j with all intermediate vertices belonging to set {1, 2, 3, 4, …,k}.

NOTES

If k is zero, that is no intermediate vertex exists between i and j, then there will be a single edge from i to j, and hence $d_{ij}^{(0)} = W_{ij}$. For this, first we need to define W_{ij}. Actually, W is the weight matrix defined as:

$$W_{ij} = \begin{cases} 0 & ; \text{if } i = j \\ \text{the weight of directed edge } (i, j) & ; \text{if } i \neq j \text{ and } (i, j) \in E \\ \infty & ; \text{if } i \neq j \text{ and } (i, j) \notin E \end{cases}$$

On the basis of the above definition, the recursive definition can be given as:

$$d_{ij} = \begin{cases} W_{ij} & ; \text{if } k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & ; \text{if } k \geq 1 \end{cases}$$

The algorithm given below is used to compute the all pairs shortest path using the above recurrence relation.

Algorithm 7.1 All Pairs Shortest Path

```
FLOYD-WARSHALL(W)
//consider set of vertices V={1, 2, 3,..n)
1. Set n = rows in matrix W
2. Set d(0) = W
3. for k = 1 to n do
4.    for i = 1 to n do
5.       for j = 1 to n do
6.          Set dij(k) = min(dij(k-1), dik(k-1)+dkj(k-1))
7.  return d(n)
```

Example 7.1: Given the directed graph shown in Figure 7.2. Design the initial n×n matrix W, then compute the values of $d_{ij}^{(k)}$ for increasing values of k, till it returns the matrix $d^{(n)}$ of shortest path weight.

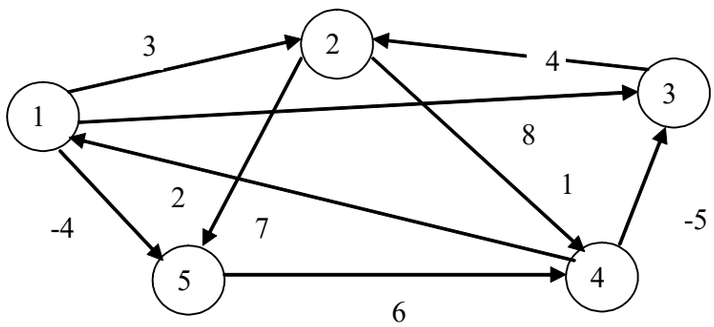


Fig. 7.2 *Directed Graph*

Solution: The initial weight matrix is:

$$W = \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$

NOTES

Here, we have considered the direct path between all the vertices, so $d^{(0)}=W$, that is,

$$d^{(0)} = \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$

Now, after applying algorithm we get the following matrices:

$$d^{(1)} = \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$

$$d^{(2)} = \begin{bmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$

$$d^{(3)} = \begin{bmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$

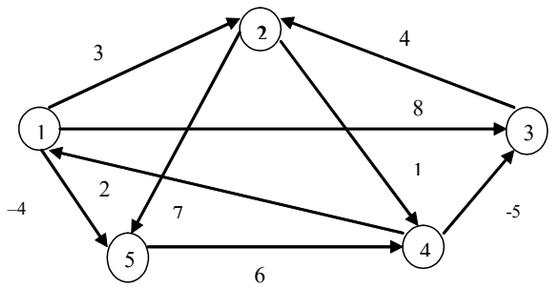
NOTES

$$d^{(4)} = \begin{bmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix}$$

$$d^{(5)} = \begin{bmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix}$$

This is the required matrix that shows all pairs shortest path of the graph.

Example 7.2: Consider the following directed graph:



The initial weight matrix,

$$W = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

Since in this matrix we have considered the direct path between all the vertices,

so $d^{(0)} = W$; i.e.,

$$d^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

After applying the algorithm, the following matrices will be formed.

NOTES

$$d^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$d^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$d^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$d^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

$$d^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

This is the required matrix which shows all pairs shortest path of the graph.

Check Your Progress

1. What is a Coefficient?
4. What is all Pairs-Shortest Path problem?

7.4 OPTIMAL BINARY SEARCH TREES

An **Optimal Binary Search Tree** (**OBST**) is a binary search tree in which nodes are arranged in such a way that the cost of searching any value in this tree is

NOTES

minimum. Let us consider a given set of n distinct key values $A=\{a_1, a_2, \ldots, a_n\}$, where $a_1<a_2<\ldots<a_n$ and it is required to construct binary search tree from these key values. The search for any of these key values will be successful. However, there may be many searches for the key values which are not part of the set A, and thus will always be unsuccessful. To represent the key values that are not part of A, **dummy** (**external**) nodes are added in the tree. If there are n key values then there will be n+1 dummy nodes. Let $d_0, d_1, \ldots, d_n$ represent dummy nodes not present in set A. Here, d_0 represents all key values less than a_1, d_i (for i= 1 to n) represents all the key values between a_i and a_{i+1} and d_n represents all the key values greater than a_n.

Figure 7.3 shows a binary search tree with dummy nodes (represented by square) added. Here, the internal nodes (shown by circle) represent the key values (a_1 to a_6) which are actually stored in the tree, while the external nodes represent the key values (d_0 to d_6) which are not present in the tree.

Let p_i be probability for each a_i with which the search will be for the key value a_i. Let q_i be the probability that the search will be unsuccessful and will end up at dummy node (say d_i). This implies,

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$$

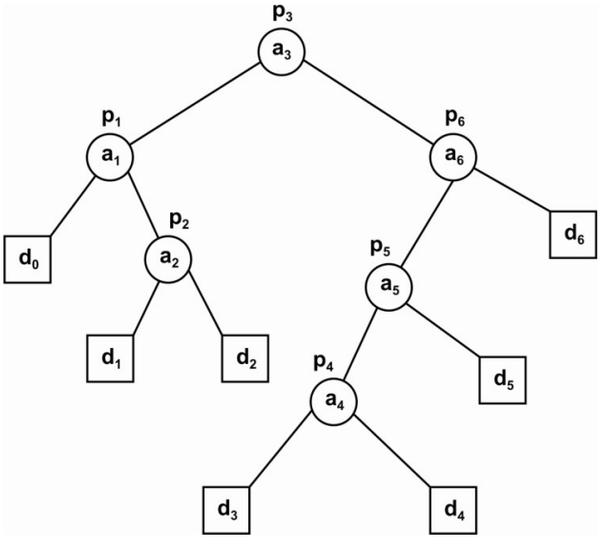


Fig. 7.3 *Binary Search Tree*

Using the probabilities of searches for key values represented by both internal and dummy nodes, the expected cost of a search in a binary search tree T can be determined. Let the cost of a search of a particular node is the number of nodes visited which is equal to one more than the depth of the node to be searched in tree T. Then, c the expected cost of a search in binary tree T is given by the following equation.

$$c = \sum_{i=1}^{n} (depth_T(k_i) + 1 .p_i) + \sum_{i=0}^{n} (depth_T(k_i) + 1 .q_i)$$

$$= 1 + \sum_{i=1}^{n} depth_T(k_i) .p_i + \sum_{i=0}^{n} depth_T(k_i) .q_i$$

NOTES

The aim is to create a binary search tree with minimum estimated search cost, that is, an optimal binary search tree. Unfortunately, the binary search tree obtained using the greedy technique may or may not be optimal This is because it always creates the tree in the decreasing order of the probabilities, that is, by taking the highest probability first, then the second highest, and so on. The resulting binary search tree may not always be the best solution of the problem. A guaranteed optimal binary search tree can be obtained using the dynamic programming technique, which is discussed in next unit.

7.5 KNAPSACK PROBLEMS

Knapsack problems can be explained considering the given example. A thief wants to rob a store which contains n items. Item i has weight w_i and is worth v_i dollars, profit earned for ith item is $p_i = (v_i/w_i)$, the capacity of the Knapsack or bag is W. If a fraction x_i, $0 \le x_i \le 1$, of item i is placed in the bag, then a profit of $p_i x_i$ is earned. The main objective of the problem is for the thief to take as valuable a load as possible but without exceeding W, i.e., maximize the total profit earned.

So, we have to maximize

$$\sum_{i=1}^{n} p_i x_i$$

Such that, $$\sum_{i=1}^{n} w_i x_i \le W$$

Where $0 \le x_i \le 1$ and $1 \le i \le n$

The following are two versions of Knapsack problem:

0-1 Knapsack: In 0-1 Knapsack, an item either must be taken or left but cannot be taken as a fractional amount, i.e., the value of x_i for ith item is either 0 or 1. If an item is taken then $x_i = 0$ and if an item is left then $x_i = 0$. 0-1 Knapsack problem can be solved using the dynamic programming method.

Fractional Knapsack: In fractional Knapsack, the thief can fraction the items. Fractional Knapsack problem can be solved using the greedy method.

Consider the following algorithm for fractional Knapsack. Here, we consider three arrays for weights, values and profits, respectively. W denotes the capacity of the Knapsack. This algorithm provides the fractions of items taken.

NOTES

FRACTIONAL-KNAPSACK (*w*, *v*, W)

1. **for** $i \leftarrow 1$ to n
2. **do** p[i] $\leftarrow$ v[i]/w[i]
3. Arrange the profit array in descending order using any linear sorting algorithm
4. **for** $i \leftarrow 1$ to n
5. **do** $x[i] \leftarrow 0.0$
6. U $\leftarrow$ W
7. **for** $i \leftarrow 1$ to n
8. **do if** ($w[i] >$ U)
9. **then** break
10. **else** $x[i] \leftarrow 1.0$
11. U $\leftarrow$ U – $w[i]$
12. **if** ($i \leq n$)
13. **then** $x[i] \leftarrow$ U/$w[i]$

The Knapsack algorithm takes O($n$log n) time if arranging profits array uses either merge sort or heap sort; otherwise only O(n) time will be required.

Example 7.3: Consider the following instance of the Knapsack problem.

$n = 3$, W = 20, $(v_1, v_2, v_3) = (25, 24, 15)$ and $(w_1, w_2, w_3) = (18, 15, 10)$.

Find the optimal solution to the Knapsack problem.

Solution:

i	**Items**	**Weight (*w*[*i*])**	**Value (*v*[*i*])**	**Profit *p*[*i*]= *v*[*i*]/*w*[*i*]**
1	I_1	18	25	1.38
2	I_2	15	24	1.6
3	I_3	10	15	1.5

Now arrange the items in the decreasing order of their profit.

i	**Items**	**Weight (*w*[*i*])**	**Value (*v*[*i*])**	**Profit *p*[*i*]= *v*[*i*]/*w*[*i*]**
1	I_2	15	24	1.6
2	I_3	10	15	1.5
3	I_1	18	25	1.38

In **for** loop of steps 4–5, $x[1] = x[2] = x[3] = 0$, i.e., we have not taken any item yet.

i **U** ***w*[*i*] > U** ***x*[*i*]**

1	20	15 > 20, false	1	(Originally it is item I_2 so $x[2] = 1$)
2	5	10 > 5, true	5/10 = 1/2	
				(Originally it is item I_3 so $x[3] = 1/2$)

So, the solution according to the original item is,

$x[1] = 0$, $x[2] = 1$ and $x[3] = 1/2$

The optimal solution is (0, 1, 1/2).

Total profit = $p[1]x[1] + p[2]x[2] + p[3]x[3]$

$= 1.38 \times 0 + 1.6 \times 1 + 1.5 \times ½$

$= 0 + 1.6 + 0.75$

$= 2.35$ units

NOTES

Example 7.4: Find an optimal solution to the Knapsack instance $n = 7$, W = 15, $(v_1, v_2, v_3, v_4, v_5, v_6, v_7) = (10, 5, 15, 7, 6, 18, 3)$ and $(w_1, w_2, w_3, w_4, w_5, w_6, w_7) = (2, 3, 5, 7, 1, 4, 1)$.

Solution:

i	Items	Weight ($w[i]$)	Value ($v[i]$)	Profit $p[i] = v[i]/w[i]$
1	I_1	2	10	5
2	I_2	3	5	1.67
3	I_3	5	15	3
4	I_4	7	7	1
5	I_5	1	6	6
6	I_6	4	18	4.5
7	I_7	1	3	3

Now arrange the items in the decreasing order of their profit.

i	Items	Weight ($w[i]$)	Value ($v[i]$)	Profit $p[i] = v[i]/w[i]$
1	I_5	1	6	6
2	I_1	2	10	5
3	I_6	4	18	4.5
4	I_7	1	3	3
5	I_3	5	15	3
6	I_2	3	5	1.67
7	I_4	7	7	1

NOTES

i	U	*w*[*i*] > U	*x*[*i*]	
1	15	1 > 15, false	1	(Originally it is item I_5 so $x[5] = 1$)
2	14	2 > 14, false	1	(Originally it is item I_1 so $x[1] = 1$)
3	12	4 > 12, false	1	(Originally it is item I_6 so $x[6] = 1$)
4	8	1 > 8, false	1	(Originally it is item I_7 so $x[7] = 1$)
5	7	5 > 7, false	1	(Originally it is item I_3 so $x[3] = 1$)
6	2	3 > 2, true	2/3	(Originally it is item I_2 so $x[2] = 2/3$)

So, the solution according to the original items is,

$x[1] = 1, x[2] = 2/3, x[3] = 1, x[4] = 0, x[5] = 1, x[6] = 1$ and $x[7] = 1$

The optimal solution is (1, 2/3, 1, 0, 1, 1, 1).

Total profit = $p[1]x[1] + p[2]x[2] + p[3]x[3] + p[4]x[4] + p[5]x[5] + p[6]x[6] + p[7]x[7]$

$= (5 \times 1) + (1.67 \times 2/3) + (3 \times 1) + (1 \times 0) + (6 \times 1) + (4.5 \times 1) + (3 \times 1)$

$= 5 + 1.11 + 3 + 0 + 6 + 4.5 + 3$

$= 22.61$ units

Check Your Progress

3. What is an Optimal Binary Search Tree?
4. What is a dummy?

7.6 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. A coefficient is defined as a multiplicative factor that is associated with any individual token of a mathematical expression.
2. All pairs-shortest path problem is to find the shortest path between all pairs of vertices in a graph G = (V,E).
3. An Optimal Binary Search Tree (OBST) is a binary search tree in which nodes are arranged in such a way that the cost of searching any value in this tree is minimum.
4. A dummy is an external node.

7.7 SUMMARY

- A coefficient is defined as a multiplicative factor that is associated with any individual token of a mathematical expression.
- The binomial coefficients for any polynomial expression in the form $(x+y)^n$ can be obtained by expanding the said expression using binomial theorem.

- In order to compute binomial coefficients of any expression say $(x+y)^n$, the main expression is divided into sub problems and the solution of the main problem is expressed in terms of the solutions obtained for small sub problems.
- The most favorable approach used to compute binomial coefficients of any expression is dynamic programming.
- Dynamic programming is best suited approach for optimization problems.
- In dynamic programming the solution of the problem is arrived by using multistage optimized decisions.
- All pairs-shortest path problem is to find the shortest path between all pairs of vertices in a graph G = (V,E).
- The Floyd-Warshall algorithm is used to find the shortest path between all pairs of vertices in a directed graph G = (V,E).
- An Optimal Binary Search Tree (OBST) is a binary search tree in which nodes are arranged in such a way that the cost of searching any value in this tree is minimum.

7.8 KEY WORDS

- **Fractional Knapsack:** In fractional Knapsack, the thief can fraction the items. A Fractional Knapsack problem can be solved using the greedy method.
- **Dummy:** It is an external node that is added to a tree.

7.9 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. How is a binomial coefficient computed?
2. Write a note on waterfalls and Floyds algorithms.
3. Discuss the concept of optimal binary search trees.
4. Write a short note explaining knapsack problems.

Long Answer Questions

1. Discuss in detail about Floyd-Warshall Algorithm.
2. What are knapsack problems? Also discuss about fractional Knapsack.
3. Find an optimal solution to the Knapsack instance $n = 9$, W = 12, ($v1$, $v2$, $v3$, $v4$, $v5$, $v6$, $v7$) = (16, 5, 15, 7, 16, 18, 3) and ($w1$, $w2$, $w3$, $w4$, $w5$, $w6$, $w7$) = (3, 3, 4, 7, 1, 4, 1).

7.10 FURTHER READINGS

NOTES

Levitin, Anany. *Introduction to Design and Analysis of Algorithms*. Delhi: Pearson Education.

Ellis Horowitz, S. Sahani and Rajasekaran, *Fundamentals of Computer Algorithms.* Delhi: Galgotia Publications.

Goodrich, M T and R. Tomassia. *Algorithm Design: Foundations, Analysis and Internet Examples*. Delhi: John wiley and Sons.

UNIT 8 GREEDY TECHNIQUE

NOTES

Structure

8.0 INTRODUCTION

The greed technique solves an optimization problem by iteratively building a solution. It always selects the optimal solution with each iteration. The problem is an optimization, greedy algorithms thus use a priority queue as an algorithmic paradigm. This follows the problem solving heuristic of making the locally optimal choice at each stage with the intent of finding a global optimum. In many cases, a greedy strategy does not usually produce an optimal solution, but still a greedy heuristic may yield to locally optimal solutions that approximate a globally optimal solution in a very reasonable amount of time.

8.1 OBJECTIVES

After going through this unit, you will be able to:

- Understand the general method
- Discuss the functionality of subset paradigm
- Explain activity selection problem
- Discuss Huffman codes

8.2 GENERAL METHOD

Greedy method is an algorithm that leads to an optimal solution by making a series of locally optimum choices, the choices that are best at that time. In greedy algorithm, an optimal solution is constructed in stages, each expands a partially constructed solution until the complete solution is obtained. At each stage, a choice is made such that:

NOTES

- It is feasible; should satisfy the given constraint.
- It is locally optimal; should be the best among all other feasible solutions at that time.
- It is irrecoverable; once the decision is made, it should not be changed at a later stage.

Consider an optimization problem having `n` inputs, a set of constraints and an objective function. Any subset of inputs that satisfies the constraints is called feasible solution. We are required to find the optimal solution (the feasible solution for which the objective function has either maximum or minimum value).

The greedy algorithm works as follows:

- Select an input from input domain.
- Check whether this input is an optimal solution by applying some selection procedure.
- Include this input to the partially constructed optimal solution if it results in feasible solution, otherwise reject this input.

Note that for selection procedure, some optimization measure must be formulated. The optimization measure may be the objective function. Though for a given problem, many different optimization measures can be used, most of them lead to algorithms that produce sub-optimal solutions. This version of greedy technique is called the **subset paradigm**. Given a list of `n` inputs (say, `list[1..n]`), we are to choose an optimal subset from the list. Algorithm 8.1 describes the greedy technique for the subset paradigm.

In this algorithm, we have used three functions including `select()`, `feasible()` and `union()`.The `select()` chooses an input from `list[]` and removes it. The `feasible()` determines whether a value can be included in the solution vector. The `union()` combines the feasible value determined by the `feasible()` with the solution and updates the objective function.

Algorithm 8.1: Greedy Method

```
Greedy(list,n)
//list is an array and n is the number of inputs.
1. Set solution=0
2. Set i=1
3. while (i = n)
4. {
5.    Set val=select(list)  //assign  the  selected  input
   value                          //to val
      If feasible(solution,val)
6.          Set solution=union(solution,val)
7.    Set i=i+1
8. }
9. return solution
```

8.2.1 Container Loading Problem

NOTES

Consider that a large ship has to be loaded with cargo. The cargo is containerized and all containers are of the same size. The weights of different containers may be different. Let the cargo capacity of the ship be c. Let the weight of the ith container be w_i, where $1 \le i \le n$. We wish to load the ship with the maximum number of containers.

To solve this problem using the greedy method, consider a variable x whose value is either 0 or 1.

If $x[i] = 0$, it means that ith container is not loaded in the cargo.

If $x[i] = 1$, it means that ith container is loaded in the cargo.

We wish to assign values to x_is that satisfies the following constraints:

$$\sum_{i=1}^{n} w[i] \times [i] \le c \text{ and } x[i] \in \{0,1\}$$

The optimization function is

$$\sum_{i=1}^{n} x[i]$$

There exists many feasible solutions because there exists many values of $x[i]$s which satisfy the given constraints and the feasible solution which maximizes, $\sum_{i=1}^{n} x[i]$ is an optimal solution.

Hence, we proceed according to the greedy method as: In the first stage, we select the container with the least weight, then select the container with the next smallest weight and continue in this way until the capacity of cargo is reached or we have finished with the containers. Selecting the containers in this way will keep the total weight of the containers minimum, and hence leave maximum capacity so that more and more containers will be loaded in the cargo.

CONTAINER-LOADING (A, capacity, *n*, *x*)

1. MERGESORT(A)
2. **for** $i \leftarrow 1$ to n
3. **do** $x[i] \leftarrow 0$
4. $i \leftarrow 1$
5. **while** ($i \le n$ and A[i].weight $\le$ capacity)
6. **do** x[A[i].id] $\leftarrow 1$
7. capacity $\leftarrow$ capacity $-$ A[i].weight
8. $i \leftarrow i + 1$

In this pseudocode, we are given an array **A** in which weights of all containers are arranged. The capacity of the ship is denoted by **capacity**. The number of containers

NOTES

are denoted by ***n***. ***x*** denotes whether a container is selected or not, accordingly x is 1 or 0.

A[i].weight denotes the weight of the container at location ***i*** in the array **A**.

A[i].id denotes the identifier in the range **1** to ***n***. This id denotes at which location the container is given in the original array.

Analysis: In Step1 we use merge sort technique to sort the containers according to their weight in the increasing order. We can also use heap sort here to sort these containers. So, Step1 takes O (n log n) time. Steps 2 and 3 take O(n) time and similarly Steps 5 to 8 take O(n) time.

So, $T(n) = O(n \log n) + O(n) = O(n \log n)$

Example 8.1: Suppose we have 8 containers whose weights are 100, 200, 50, 90, 150, 50, 20 and 80, and a ship whose capacity, $c = 400$. Use CONTAINER-LOADING algorithm to find an optimal solution to this container loading problem.

Solution: Apply the above **CONTAINER-LOADING** algorithm as:

Initially: A

100	200	50	90	150	50	20	80

In Step1 we use a sorting technique and thus the array becomes,

20	50	50	80	90	100	150	200

In Steps 2 and 3, we set $x[i] = 0$, which indicates that till now we have selected no container.

$x[1] = x[2] = x[3] = x[4] = x[5] = x[6] = x[7] = x[8] = 0$

n	***i***	**capacity**	**A[*i*].weight**	**A[*i*].id**	***x*[A[*i*].id]**	
8	1	400	20	7	1	
	2	380	50	3	1	
	3	330	50	6	1	
	4	280	80	8	1	
	5	200	90	4	1	
	6	110	100	1	1	
	7	10	150	5	0	Condition fails
	8	10	200	2	0	Condition fails

So, the desired solution is $x[1.....8] = [1, 0, 1, 1, 0, 1, 1, 1]$

$\sum_{i=1}^{n} x[i] = 6$ is an optimal solution.

NOTES

Example 8.2: Suppose you have 6 containers whose weights are 50, 10, 30, 20, 60 and 5, and a ship whose capacity $c = 100$. Use CONTAINER-LOADING algorithm to find an optimal solution to this container loading problem.

Solution: Apply the above CONTAINER-LOADING algorithm as:

Initially: A

50	10	30	20	60	5

After sorting the array becomes,

5	10	20	30	50	60

n	***i***	**capacity**	**A[*i*].weight**	**A[*i*].id**	***x*[A[*i*].id]**	
6	1	100	5	6	1	
	2	95	10	2	1	
	3	85	20	4	1	
	4	65	30	3	1	
	5	35	50	1	0	condition fails
	6	35	60	5	0	condition fails

So, the desired solution is $x[1 \ldots\ldots 6] = [0, 1, 1, 1, 0, 1]$

$\sum_{i=1}^{n} x[i] = 4$ is an optimal solution.

8.2.2 An Activity Selection Problem

Consider that there are certain competing activities for which resources have to be scheduled in an exclusive manner, with the goal of selecting the maximum size set of mutually compatible activities. The greedy algorithm helps us to achieve this goal. Suppose we have a set of n proposed activities given as $S = \{a_1, a_2, a_3, \ldots\ldots a_n\}$. Each activity requires a resource in a mutually exclusive manner, i.e., the resource can be used by only one activity at a time. Each activity a_i has a **start time** s_i and a **finish time** f_i, where $0 \le s_i < f_i < \infty$. If selected, activity a_i takes place during the half-open time interval $[s_i, f_i)$. Activities a_i and a_j are **compatible** if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap (i.e., a_i and a_j are compatible if $s_i \ge f_j$ or $s_j \ge f_i$). The **activity selection problem** is to select the maximum-size subset of mutually compatible activities.

GREEDY-ACTIVITY-SELECTOR (*s*, *f*)

1. $n \leftarrow$ length[s]
2. A $\leftarrow \{a_1\}$
3. $i \leftarrow 1$
4. **for** $m \leftarrow 2$ to n

5. **do if** $s_m \geq f_i$
6. then A $\leftarrow$ A U $\{a_m\}$
7. $i \leftarrow m$
8. return A

NOTES

In this problem, we have to first select the minimum duration activity, i.e., the activity which holds the resource for the least duration. Then we ignore all those activities which are not compatible with the selected activity and select the next minimum duration activity which is compatible. Assume that the activities are arranged in the order of their increasing finish times so that the process of selecting an activity becomes faster.

Example 8.3: Consider the following 11 activities along with their start and finish time.

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

Compute a schedule where the largest number of activities takes place.

Solution: First, we arrange the activities in the increasing order of their time. In this example, the activities are already given in the increasing order of their finish time. According to algorithm, in Step 2, we select the first activity in set **A**. In Steps 4 to 7 we have a **for** loop from the 2nd to the nth activity; select the activity whose start time is greater or equal to the activity already selected, i.e., select those activities which are compatible with the selected activity.

n	**A**	***i***	***m***	s_m	f_i	
11	a_1	1	2	3	4	Condition fails
			3	0	4	Condition fails
			4	5	4	Condition true
	$a_1 \cup a_4$	4				
			5	3	7	Condition fails
			6	5	7	Condition fails
			7	6	7	Condition fails
			8	8	7	Condition true
	$a_1 \cup a_4 \cup a_8$	8				
			9	8	11	Condition fails
			10	2	11	Condition fails
			11	12	11	Condition true
	$a_1 \cup a_4 \cup a_8 \cup a_{11}$					
	Return $a_1 \cup a_4 \cup a_8 \cup a_{11}$					

RECURSIVE-ACTIVITY-SELECTOR (*s*, *f*, *i*, *j*)

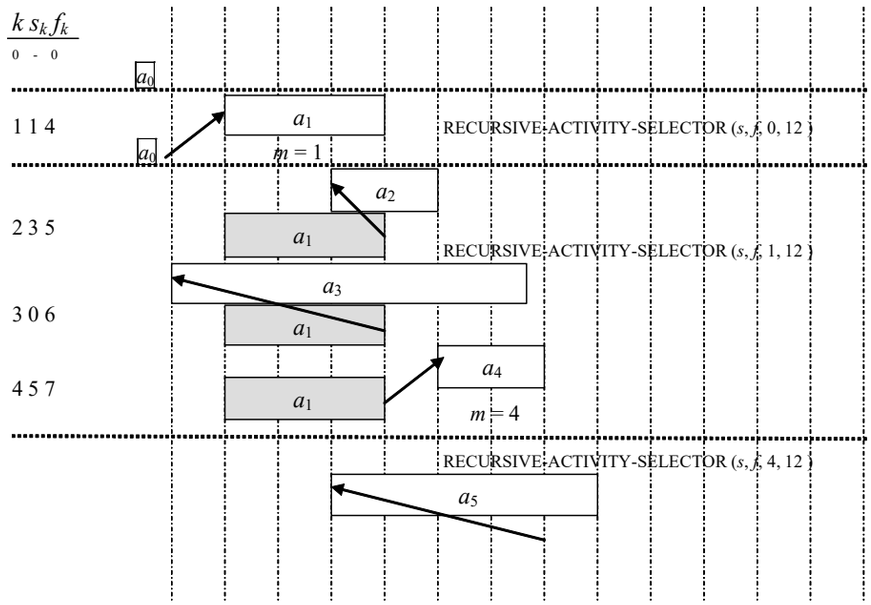
1. $m \leftarrow i + 1$
2. **while** $m < j$ and $s_m < f_i$
3. **do** $m \leftarrow m + 1$
4. **if** $m < j$
   5. **then** return $\{a_m\} \cup$ **RECURSIVE-ACTIVITY-SELECTOR (*s*, *f*, *m*, *j*)**
6. **else** return Φ

The initial call is **RECURSIVE-ACTIVITY-SELECTOR (*s*, *f*, 0, *n* + 1).**

The operation of **RECURSIVE-ACTIVITY-SELECTOR** is shown as follows:

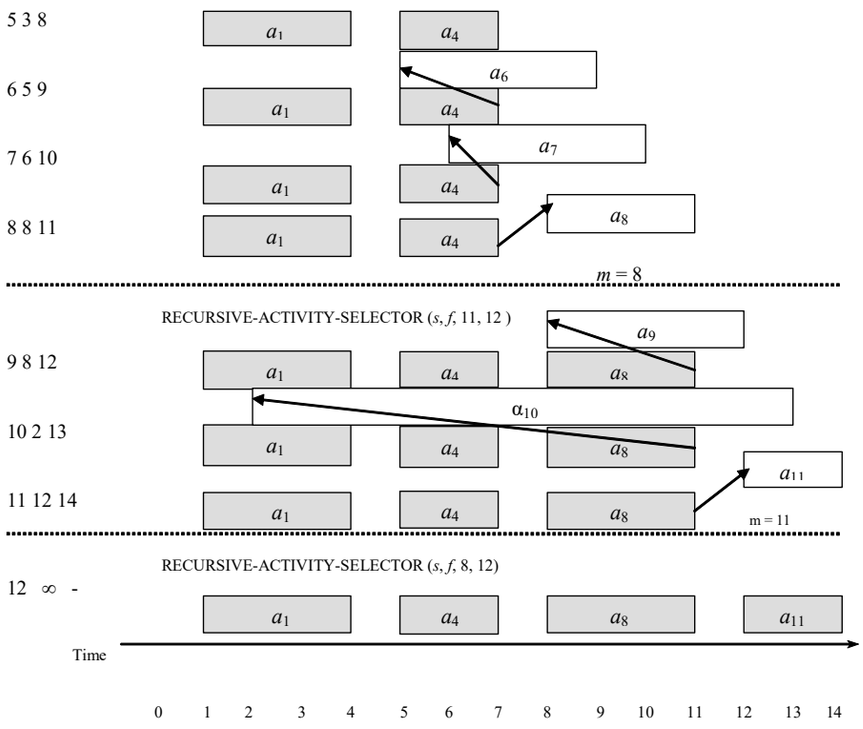
Analysis: Both the versions, i.e., iterative and recursive runs in Θ(*n*) time if the activities are arranged in the increasing order according to their finish time.

If the activities are not sorted, then first sort them either using merge sort or heap sort which takes O(*n* log *n*) time.



NOTES

NOTES



8.2.3 Huffman Codes

Huffman code is a technique through which data can be encoded and it is a very effective technique for compressing data. Using this technique, we can save up to 20 to 90 per cent data depending upon the file. Huffman greedy algorithm uses frequencies of occurrence of characters so that we can build up an optimal solution to represent each character as a binary string.

Suppose we have a file having 100,000 characters and there are only six distinct characters whose frequencies of occurrence in a file is given below. We can compact this file either using fixed length code or variable length code.

Characters	*a*	*b*	*c*	*d*	*e*	*f*	Total
Frequency(in thousands)	45	13	12	16	9	5	100

Fixed Length Code: Using fixed length code, each character in the file is represented by equal number of bits. As there are only six characters, three bits are only required to represent each character as,

a	000
b	001
c	010
d	011
e	100
f	101

So, the total bits required are 3×10^5.

Variable Length Code: Variable length coding does much better than fixed length coding, i.e., the saving percentage increases using this technique. If we represent each character with unequal number of bits, then

NOTES

a	0
b	101
c	100
d	111
e	1101
f	1100

So, the total bits required are: $(45000 \times 1) + (13000 \times 3) + (12000 \times 3) + (16000 \times 3) + (9000 \times 4) + (5000 \times 4) = 2.24 \times 10^5$ bits,

A saving of approximately 25 per cent and this is an optimal character coding for this file.

While using variable length code, we use prefix codes. Prefix codes are those codes in which no codeword is a prefix of some other codeword. Prefix codes are used because they simplify decoding.

Example 8.4: (a) Is 101, 0011, 011, 1011 is a prefix code?

(b) Is 0, 101, 1100, 1101, 100 is a prefix code?

Solution: (a) 101, 0011, 011, 1011 is not a prefix code because here the codeword 101 is a prefix of codeword 1011.

(b) 0, 101, 1100, 1101, 100 is a prefix code because here no codeword is a prefix of some other codeword.

HUFFMAN (C)

1. $n \leftarrow |C|$
2. $Q \leftarrow C$
3. **for** $i \leftarrow 1$ to $n - 1$
4. **do** Allocate a new node z
5. left[z] $\leftarrow x \leftarrow$ EXTRACT-MIN(Q)
6. right[z] $\leftarrow y \leftarrow$ EXTRACT-MIN(Q)
7. $f[z] \leftarrow f[x] + f[y]$
8. INSERT(Q, z)
9. return EXTRACT-MIN(Q)

In this algorithm, we have a set C which contains the characters. The characters are maintained in a priority queue Q according to the increasing order of their frequencies. **INSERT(Q, *z*)** inserts a node z into the priority queue.

NOTES

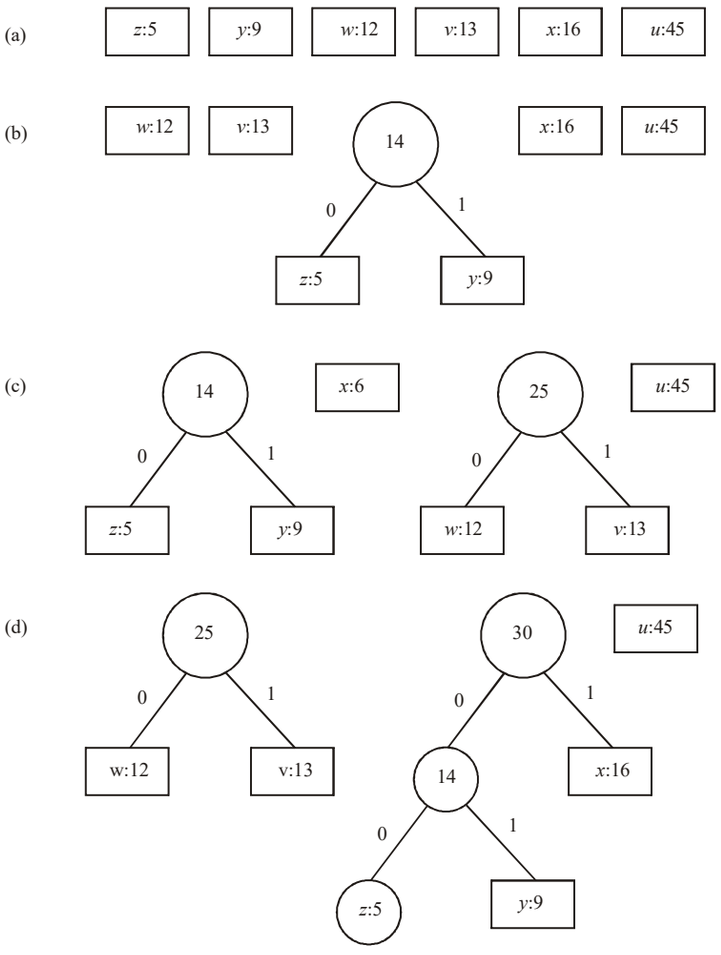
EXTRACT-MIN(Q) removes and returns the element having the minimum key from the priority queue.

Analysis: The priority queue in Step 2 can be initialized in O(n) time. This priority is created by building max heap. Steps 3 to 8 are executed exactly $n - 1$ times, and since each heap operation takes O(log n) time, Steps 3 to 8 contribute to O(n log n) time.

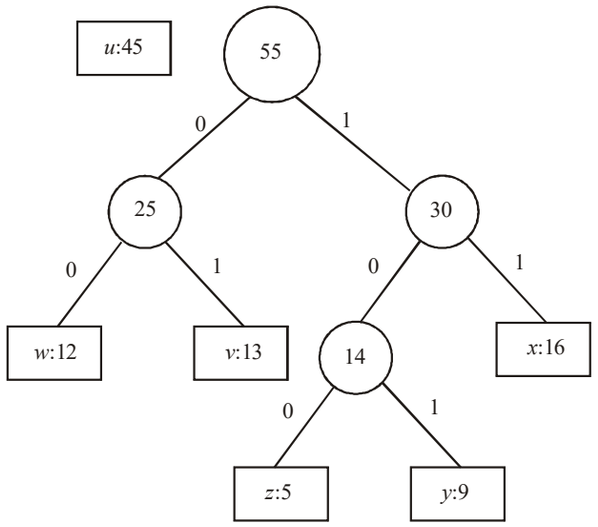
Thus, the running time of Huffman code on a set C of n characters takes O(n log$n$) time.

Example 8.5: What is an optimal Huffman code for the following set of frequencies? $u : 45, v : 13, w : 12, x : 16, y : 9, z : 5$

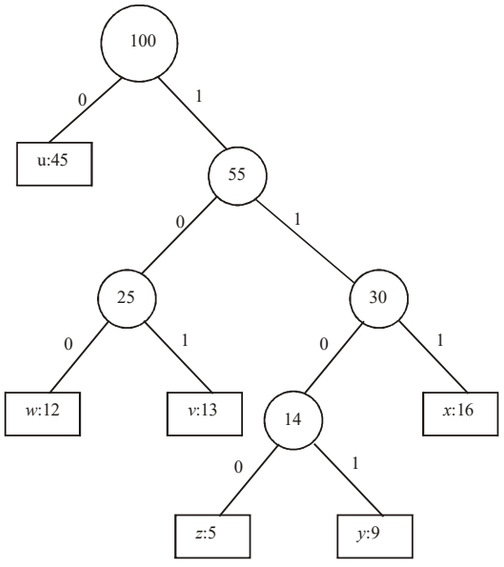
Solution: First, arrange the characters in the increasing order of their frequencies.



(e)



(f)



To find the codeword for each character, we start from the root and reach the leaf which contains the character. So, the codeword for each character is given as:

$u \Rightarrow 0$

$v \Rightarrow 101$

$w \Rightarrow 100$

$x \Rightarrow 111$

$y \Rightarrow 1101$

$z \Rightarrow 1100$

NOTES

NOTES

Check Your Progress

1. What is Greedy method?
2. What is Huffman code?

8.3 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. Greedy method is an algorithm that leads to an optimal solution by making a series of locally optimum choices, the choices that are best at that time.
2. Huffman code is a technique through which data can be encoded and it is a very effective technique for compressing data.

8.4 SUMMARY

- Greedy method is an algorithm that leads to an optimal solution by making a series of locally optimum choices, the choices that are best at that time.
- Note that for selection procedure, some optimization measure must be formulated.
- Huffman code is a technique through which data can be encoded and it is a very effective technique for compressing data.
- Using Huffman code technique, we can save up to 20 to 90 per cent data depending upon the file.
- Huffman greedy algorithm uses frequencies of occurrence of characters so that we can build up an optimal solution to represent each character as a binary string.
- Using fixed length code, each character in the file is represented by equal number of bits.
- Variable length coding does much better than fixed length coding, i.e., the saving percentage increases using this technique.

8.5 KEY WORDS

- **Fixed Length Code:** Using this code each character in the file is represented by equal number of bits.
- **Variable Length Code:** It is a code that does much better than fixed length coding, i.e., the saving percentage increases using this technique.

8.6 SELF ASSESSMENT QUESTIONS AND EXERCISES

NOTES

Short Answer Questions

1. What is the general method?
2. Discuss the functionality of subset paradigm.
3. Explain activity selection problem and its use.
4. Discuss about the Huffman codes.

Long Answer Questions

1. Suppose you have 6 containers whose weights are 20, 40, 50, 10, 40 and 15, and a ship whose capacity $c = 100$. Use CONTAINER-LOADING algorithm to find an optimal solution to this container loading problem.
2. What are prefix codes? Discuss their nature and functionality. Also explain if the following are prefix codes?
   (a) 101, 0011, 011, 1011?
   (b) 0, 101, 1100, 1101, 100?
3. What is an optimal Huffman code for the following set of frequencies? $u : 45, v : 13, w : 12, x : 16, y : 9, z : 5$

8.7 FURTHER READINGS

Levitin, Anany. *Introduction to Design and Analysis of Algorithms*. Delhi: Pearson Education.

Ellis Horowitz, S. Sahani and Rajasekaran, *Fundamentals of Computer Algorithms.* Delhi: Galgotia Publications.

Goodrich, M T and R. Tomassia. *Algorithm Design: Foundations, Analysis and Internet Examples*. Delhi: John wiley and Sons.

UNIT 9 APPLICATIONS

NOTES

Structure

9.0 INTRODUCTION

A non-recursive technique is anything that doesn't use recursion. Some algorithms that fall under this are Prim's, Kruskal's, and Dijkstra's. the Prim's algorithm is a greedy algorithm type that finds a minimum spanning tree for a weighted undirected graph. It can find a subset of the edges that forms a tree that includes every vertex, where in the total weight of all the edges in the tree is minimized. Kruskal's algorithm, on the other hand is a minimum-spanning-tree algorithm which finds an edge of the least possible weight that connects any two trees. Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph. This unit will elaborate on these algorithms.

9.1 OBJECTIVES

After going through this unit, you will be able to:

- Understand Prim's algorithm
- Discuss about Kruskal's algorithm
- Explain about Dijkstra's algorithm

9.2 MINIMAL SPANNING TREE

A **spanning tree** of a connected graph `G` is a tree that covers all the vertices and the edges required to connect those vertices in the graph. Formally, a tree `T` is called a **spanning tree** of a connected graph `G` if the following two conditions hold.

1. T contains all the vertices of G, and
2. All the edges of T are subsets of edges of G.

For a given graph G with n vertices, there can be many spanning trees and each tree will have n-1 edges. For example, consider the graph shown in Figure 9.1. Since this graph has four vertices, each spanning tree must have 4-1 = 3 edges. Some of the spanning trees for this graph are shown in Figure 9.2. Observe that in spanning trees, there exists only one path between any two vertices and insertion of any other edge in the spanning tree results in a cycle.

Note: The weight of a spanning tree is the sum of the weight of edges in that tree.

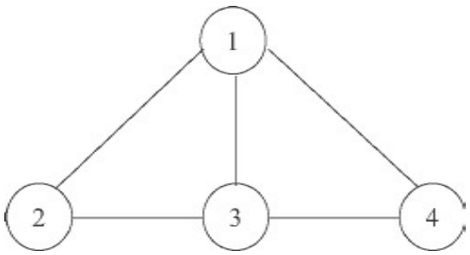


Fig. 9.1 *A Simple Graph G*

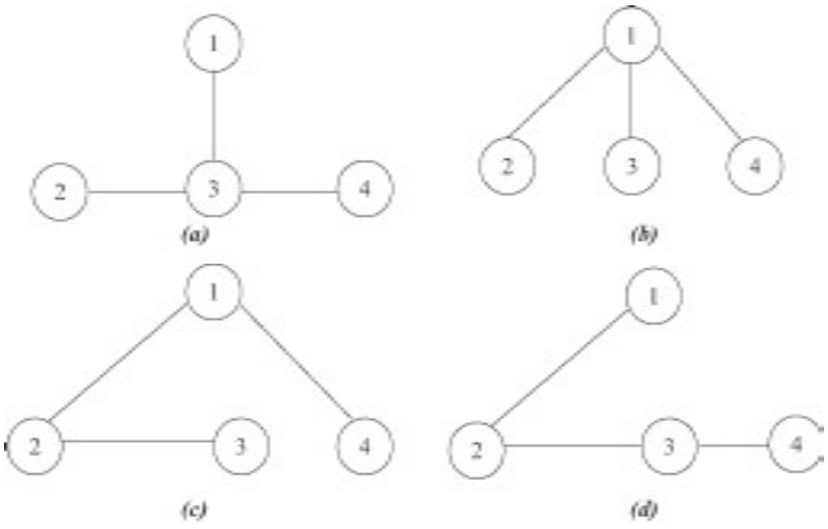


Fig. 9.2 *Spanning Trees of Graph G*

For a connected weighted graph G, it is required to construct a spanning tree T such that the sum of weights of the edges in T must be minimum. Such a tree is called a **minimal spanning tree**. There are various approaches for constructing a minimal spanning tree out of which Kruskal's algorithm and Prim's algorithm are commonly used and both applies greedy strategy to form the minimal spanning tree.

NOTES

NOTES

9.2.1 Kruskal's Algorithm

In Kruskal's approach, initially, all the vertices, `n`, of the graph are considered as distinct partial tree having one vertex and all its edges are listed in the increasing order of their weights. The minimal spanning tree is constructed by repeatedly inserting one edge at a time until exactly `n-1` edges are inserted. The edges are inserted in the increasing order of their weights. Further, an edge is inserted in the tree only if its inclusion does not form a cycle.

Consider an undirected weighted connected graph shown in Figure 9.3. In order to construct the minimal spanning tree `T` for this graph, the edges must be included in the order (1,3), (1,2), (1,6), (4,6), (3,6), (2,6), (3,4), (1,5), (3,5) and (4,5). This sequence of edges corresponds to the increasing order of weights (3, 4, 5, 6, 7, 8, 9, 12, 14 and 15). The first four edges (1,3), (1,2), (1,6) and (4,6) are included in `T`. The next edge that is to be inserted in order of cost is (3,6). Since, its inclusion in the tree forms a cycle, it is rejected. For same reason, the edges (2,6) and (3,4) are rejected. Finally, on the insertion of the edge (1,5), `T` has `n-1` edges. Thus, the algorithm terminates and the generated tree is a minimal spanning tree. The weight of this minimal spanning tree is 30. All the steps are shown in Figure 9.4.

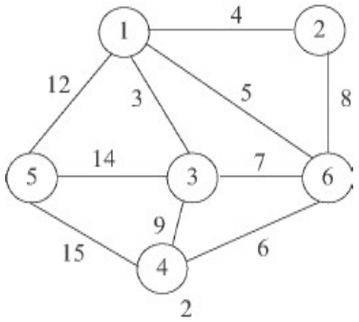


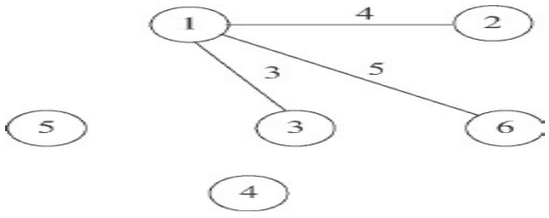
Fig. 9.3 *An Undirected Connected Graph*

(a) *Distinct Vertices*

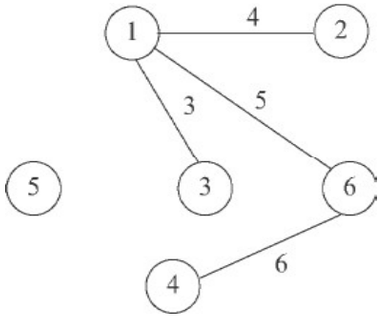
(b) *Insertion of Edge (1, 3) with Minimum Weight 3*

NOTES

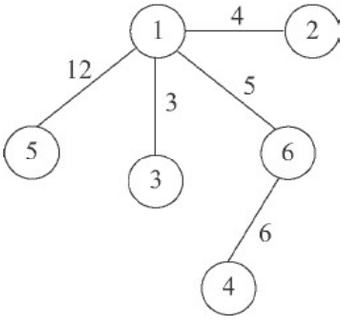
(c) *Insertion of Edge (1,2) with Weight 4*



(d) Insertion of Edge (1,6) with Weight 5



(e) *Insertion of Edge (6,4) with Weight 6*



(f) *Insertion of Edge (1,5) with Weight 12*

Fig. 9.4 *Constructing Minimal Spanning Tree Using Kruskal's Algorithm*

NOTES

Algorithm 9.1: Kruskal's Algorithm

```
Greedy_kruskal(E)
//E is the set of edges in graph G containing n nodes. MST
//contains the edges in the minimum spanning tree
 1. Set i=n-1
 2. while(i = n-1)
 3. {
 4.   Find minimum cost edge(x,y) from set of edges
 5.   Set E={x,y}
 6.   Set root_x=find(x) //find the root node of tree
                         //containing x
      Set root_y=find(y) //find the root node of tree
                         //containing y
 7.   If(root_x ? root_y)
 8.   {
 9.       Merge x and y
10.       Set MST=union(MST,E) //add minimum edge to the tree
11.   }
12.       Set i=i+1
13. }
```

9.2.2 Prim's Algorithm

Kruskal's algorithm requires listing the edges in the increasing order of their weights and at each step we need to determine whether the inclusion of a new edge results in a cycle. This leads to an extra overhead. To reduce this overhead, we can use Prim's algorithm to find the minimal spanning tree of a graph.

According to Prim's algorithm, the minimal spanning tree is constructed in a sequential manner using greedy approach. Initially, any vertex (say, V_i) is randomly selected from the graph having n vertices and then its associated vertex (say, V_j) is found such that the edge (V_i, V_j) is the minimal (that is, having smallest weight) and this edge is added to the tree. Now, V_i and V_j is considered as a sub tree and another vertex V_k that is the closest neighbor of this sub tree is found and the corresponding minimal edge is added to the tree. This greedy strategy is continued until we get the tree with n vertices connected by $n-1$ edges.

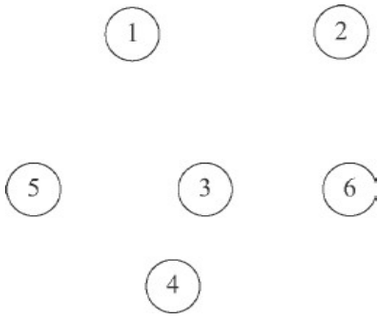
The Prim's algorithm can be implemented easily with the help of adjacency weight matrix representation of the graph, which contains weights of edges as its entries. To understand the working of Prim's algorithm, an undirected weighted graph $G(V,E)$ having n vertices labeled as $(V_1, V_2, V_3, \ldots, V_n)$. Starting from the vertex V_1, the corresponding row is scanned in the adjacency weight matrix to find the smallest entry (say corresponding to vertex V_k) and the edge (V_1, V_k) is inserted in the tree. Now, smallest value in both the rows corresponding to V_1 and V_k in the adjacency weight matrix is searched. Suppose such an entry (say, corresponding to vertex V_m) is found in the row of V_k. The edge (V_k, V_m) in the sub tree is inserted thereby connecting the vertices V_1, V_k, and V_m. Similarly, the

other smallest entry in the rows of V_1, V_k, and V_m is found out and corresponding edge is inserted in the sub tree. This process is continued until all the n vertices get connected by n-1 edges. Figure 9.5 shows the adjacency weight matrix of the graph shown in Figure 9.3 and the Figure 9.6 illustrates the steps for constructing the minimal spanning tree for this graph using greedy strategy.

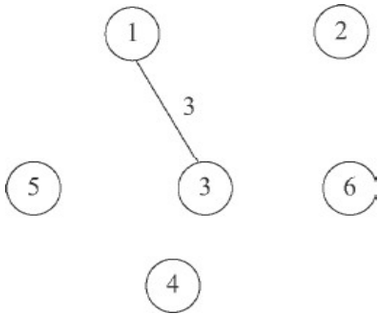
NOTES

	1	2	3	4	5	6
1	0	4	3	0	12	5
2	4	0	0	0	0	8
3	3	0	0	9	14	7
4	0	0	9	0	15	6
5	12	0	14	15	0	0
6	5	8	7	6	0	0

Fig. 9.5 *Adjacency Weight Matrix*

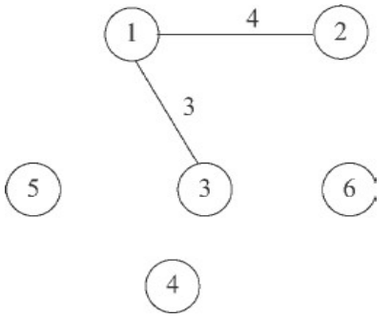


(a) *Initial Tree*

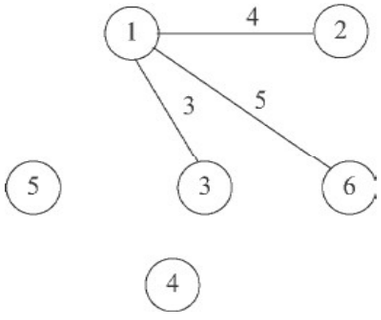


(b) *Insertion of Edge (1, 3) with Weight 3*

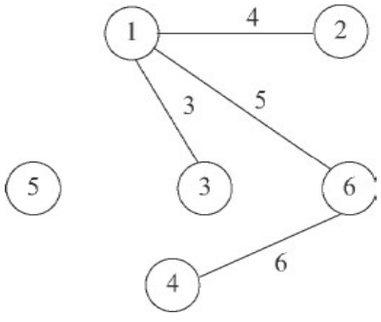
NOTES



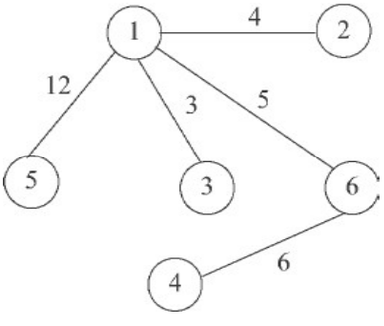
(c) *Insertion of Edge (1, 2) with Weight 4*



(d) *Insertion of Edge (1, 6) with Weight 5*



(e) *Insertion of Edge (6, 4) with Weight 6*



(f) *Insertion of Edge (1, 5) with Weight 12*

Fig. 9.6 *Constructing the Minimum-Cost Tree Using Prim's Algorithm*

Algorithm 9.2: Prim's Algorithm

```
Greedy_Prims(Edge,r,cost,MST)
//Edge is the set of edges in Graph G having r vertices and
//cost C. MST[1:r-1,1:2] is the array to hold set minimum
//cost edges in the spanning tree
 1. Set Minimum_cost=cost[m,n] //[m,n] is the edge having
                               //minimum cost in Edge
 2. Set MST[1,1]=m
 3. Set MST[1,2]=n
 4. Set j=1
 5. while(j = r)
 6. {
 7.   If(cost[j,n]<cost[j,m]) //determine the adjacent
                              //vertex
 8.         Set near_vertex[j]=n
 9.      Else
10.         Set near_vertex[j]=m
11.      Set j=j+1
12. }
13. Set near_vertex[m]=0
14. Set near_vertex[n]=0
15. Set j=2
16. while(j = r-1)          //build the remaining spanning
tree
17. {
     //Let i is any index such that near_vertex[i]?0 and
     //cost[i,near_vertex[i]] is minimum
18.   Set MST[j,1]=i
19.   Set MST[j,2]=near_vertex[i]  //determine the next edge
                                   //to be included in the
                                         //spanning tree
20.   Set Minimum_cost= Minimum_cost+cost[i,near_vertex[i]]
21.   Set near_vertex[i]=0
22.   Set m=1
23.   while(m = r)                 //update next_vertex
24.   {
25.      If((near_vertex[m]?0) AND (cost[m,near_
                                      vertex[m])>cost[m,i]))
26.         Set near_vertex[m]=i
27.      Set m=m+1
28.   }
29.   Set j=j+1
30. }
31. return  Minimum_cost
```

NOTES

NOTES

9.3 DIJKSTRA'S ALGORITHM

Dijkstra's algorithm is first proposed by E.W. Dijkstra, a Dutch computer scientist to solve a special kind of shortest path problem known as **single shortest path problem**. In this problem, given a directed graph G(V,E) with a weight assigned to each edge in G and a source vertex V_o, we have to find the shortest path from V_o to all the other vertices of G. For example, consider a graph shown in Figure 9.7.

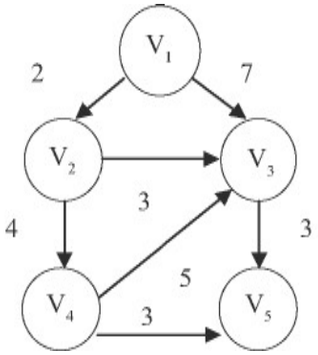


***Fig. 9.7** A Directed Graph*

The different shortest paths, assuming that V_1 is the source vertex are given below:

***Table 9.1** Shortest Path*

Vertex	Shortest Path	Length of the Shortest Path
From V_1 to V_2	$V_1 \rightarrow V_2$	2
From V_1 to V_3	$V_1 \rightarrow V_2 \rightarrow V_3$	5
From V_1 to V_4	$V_1 \rightarrow V_2 \rightarrow V_4$	6
From V_1 to V_5	$V_1 \rightarrow V_2 \rightarrow V_4 \rightarrow V_5$	9

Algorithm 9.3: Dijkstra's Algorithm

```
Greedy_Dijkstra(v,cost,d,n)
//d is the weight assigned to the vertex in graph G with n
//vertices. G represented by the adjacency matrix cost. v
//is the source vertex
 1. Set i=1
 2. while(i = n)
 3. {        //initializing path
 4.       Set S[i]=false
 5.       Set d[i]=cost[v,i]
 6. }
 7. Set S[v]=true
```

```
 8. Set d[v]=0                  // initialize source vertex
 9. Set j=2
10. while(j = n)
11. {
12.       Select u such that d[u] is minimum from v

13.       Set S[u]=true
14.       for each x adjacent to u with path[x]=false
15.       {
16.          If(d[x]>d[u]+cost[u,x]) //update the distances
17.             Set d[x]=d[u]+cost[u,x]
18.       }
19.       Set j=j+1
20. }
```

Example 9.1: Find the shortest path for the given graph using Dijkstra's algorithm assuming that the source vertex is 1.

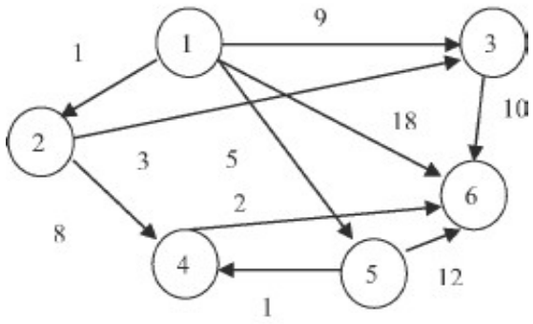


Fig. 9.8 *A Directed Graph G*

Solution: The adjacency matrix for the above graph, *G* is shown below:

	1	2	3	4	5	6
1	∞	1	9	∞	5	18
2	∞	∞	3	8	∞	∞
3	∞	∞	∞	∞	∞	10
4	∞	∞	∞	∞	∞	2
5	∞	∞	∞	4	∞	12
6	∞	∞	∞	∞	∞	∞

Fig. 9.9 *Adjacency Matrix for Graph, G*

NOTES

To compute the shortest path for all the vertices, follow the steps given below:

Step 1:

Consider all the outward edges from the source vertex 1 (see Figure 9.10(a)).Initially, the source vertex has weight 0 and other vertices connected directly to it will have the weight specified on their edges (see Figure 9.10(b)). This weight is the distance d, of the vertex from the source vertex. The vertex that is not directly connected to the source vertex has weight equal to ". At this step, S = {1}, where S contains the list of vertices that have already been visited.

Step 2:

Next, select the vertex having least weight among all the vertices i.e. vertex 2 and consider the outward edges from vertex 2 (see Figure 9.10(c)). Then, set the distance of vertex 4 as 8+1=9 as specified on the edge from 2 to 4 and adding the distance of vertex 2 to it as well (see Figure 9.10(d)). Also, change the distance, d, of vertex 3 to 3 as it is the shorter distance (vertex 2 to vertex 3) from the one assigned previously (vertex 1 to vertex 3). Thus, S = {1, 2}.

Step 3:

Now, the next vertex with least weight after considering vertex 2 is 3 (see Figure 9.10(e)). Connect all the outgoing edges from 3 and adjust the distance of the vertices accordingly. If any vertex can have the shorter distance by using the edge from vertex 3, then its assigned distance, d, will be replaced with new distance as done for vertex 6 (see Figure 9.10(f)). Thus, S = {1, 2, 3}.

Step 4:

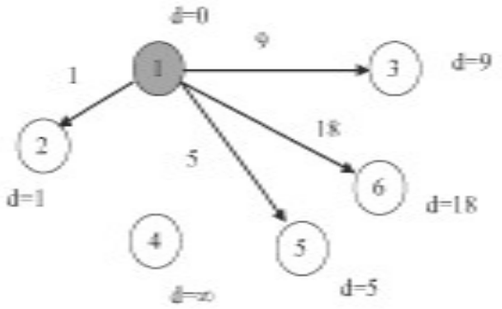
Thereafter, the next vertex with least weight after considering vertex 2 is 5 (see Figure 9.10(g)). Adjust the distance of the vertices according to vertex 5 (see Figure 9.10(h)). If any vertex can have the shorter distance by using the edge from vertex 5, then its assigned distance, d, will be replaced with the new distance as done for vertex 4. Thus, S = {1, 2, 3, 5}.

Step 5:

Consider vertex 4 now as it has the least weight after vertex 5 (see Figure 9.10(i)) and change the value for distance, d, of other vertices only if the new value is less than the previously assigned value as done for vertex 6 (see Figure 9.10(j)). Thus, S = {1, 2, 3, 5, 4}.

Step 6:

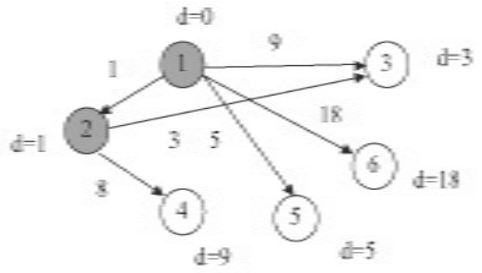
Finally, select vertex 6. Figure 9.10(l) shows the shortest distance from vertex 1 (source vertex) to every other vertex V. Thus, S = {1, 2, 3, 5, 4, 6}.



(a)

	1	2	3	4	5	6
1	0	1	9	∞	5	18

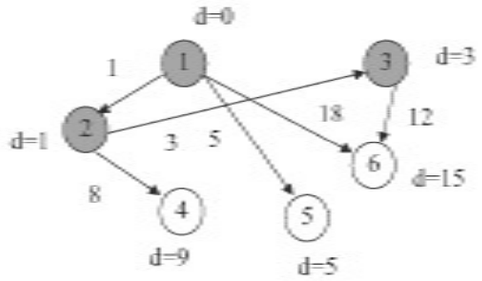
(b)



(c)

	1	2	3	4	5	6
1	0	1	3	8	5	18

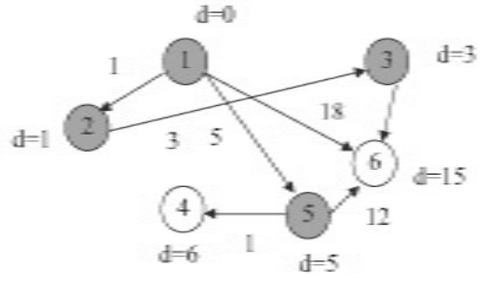
(d)



(e)

	1	2	3	4	5	6
1	0	1	3	9	5	15

(f)



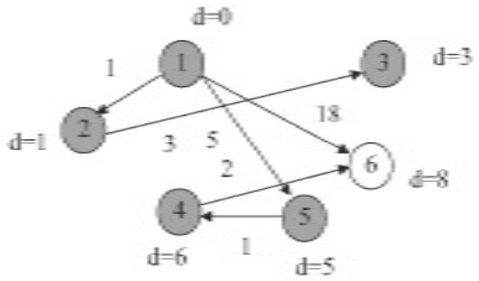
(g)

	1	2	3	4	5	6
1	0	1	3	6	5	15

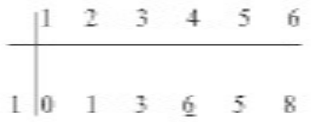
(h)

NOTES

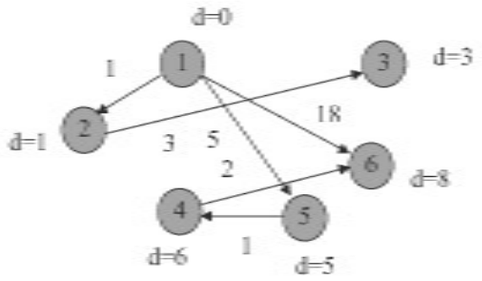
NOTES



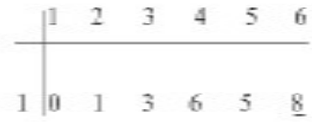
(i)



(j)



(k)



(l)

Fig. 9.10 *Stages of Dijkstra Algorithm with Their Shortest Distance*

Check Your Progress

1. What does Kruskal's algorithm require?
2. What is a spanning tree of connected graph?

9.4 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. Kruskal's algorithm requires listing the edges in the increasing order of their weights.
2. A spanning tree of a connected graph G is a tree that covers all the vertices and the edges required to connect those vertices in the graph.

9.5 SUMMARY

- A spanning tree of a connected graph G is a tree that covers all the vertices and the edges required to connect those vertices in the graph.
- In Kruskal's approach, initially, all the vertices, n, of the graph are considered as distinct partial tree having one vertex and all its edges are listed in the increasing order of their weights.

- Kruskal's algorithm requires listing the edges in the increasing order of their weights and at each step we need to determine whether the inclusion of a new edge results in a cycle.
- Dijkstra's algorithm is first proposed by E.W. Dijkstra, a Dutch computer scientist to solve a special kind of shortest path problem known as single shortest path problem.
- According to Prim's algorithm, the minimal spanning tree is constructed in a sequential manner using greedy approach.

9.6 KEY WORDS

- **Kruskal's Algorithm**: It is a minimum-spanning-tree algorithm which finds an edge of the least possible weight that connects any two trees in the forest.
- **Dijkstra's Algorithm**: It is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks.

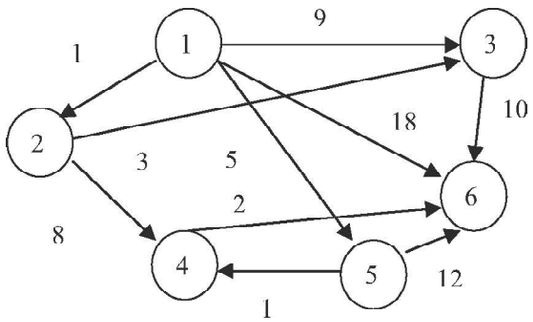
9.7 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. Write a short note on prim's algorithm.
2. Discuss about Kruskal's algorithm.
3. Explain about Dijkstra's algorithm.

Long Answer Questions

1. Find the shortest path for the given graph using Dijkstra's algorithm assuming that the source vertex is 1.



2. "There are various approaches for constructing a minimal spanning tree." Explain in detail.

3. "The minimal spanning tree is constructed by repeatedly inserting one edge at a time until exactly n-1 edges are inserted." Discuss.

NOTES

9.8 FURTHER READINGS

Levitin, Anany. *Introduction to Design and Analysis of Algorithms*. Delhi: Pearson Education.

Ellis Horowitz, S. Sahani and Rajasekaran, *Fundamentals of Computer Algorithms.* Delhi: Galgotia Publications.

Goodrich, M T and R. Tomassia. *Algorithm Design: Foundations, Analysis and Internet Examples*. Delhi: John wiley and Sons.

NOTES

BLOCK - IV
SORTING AND OPTIMIZATION PROBLEM

UNIT 10 SORTING AND SEARCHING ALGORITHMS

Structure

10.0 INTRODUCTION

Algorithm analysis should begin with a clear statement of the task to be performed. This allows us both to check that the algorithm is correct and to ensure that the algorithms we are comparing perform the same task. A sorting algorithm in computer science is an algorithm that puts elements of a list in a certain order. A search algorithm on the other hand is a step-by-step procedure used to locate specific data among a certain collection of data. This is also considered a fundamental procedure in computing. In computer science the difference between a fast application and a slower one often lies in the use of the proper search algorithm. This unit will explain sort and searching algorithms in detail.

10.1 OBJECTIVES

After going through this unit, you will be able to:

- Understand the concept of decrease and conquer
- Explain the functionality of insertion sort
- Discuss topological sorting
- Differentiate between DFS and BFS

NOTES

10.2 DECREASE AND CONQUER

This is a problem solving approach especially used to perform searching and sorting operations. By adopting approaches like divide and conquer, Decrease and conquer the main problem is reduced to smaller sub- problem at each step of its execution. However, decrease and conquer is not same as that of divide and conquer .

The main and principle approach of decrease and conquer strategy involves three major activities:

1. **Decrease:** Reduce the main problem domain into smaller sub-problem instances and extend the solution space.
2. **Conquer:** Resolve these smaller sub-problems to obtain desired result.
3. **Extend:** Extend the result obtained in step 2 to arrive at final problem result.

The complexity of the problem can be reduced by three different variations of decrease and conquer approach of problem solving:

(a) Decrease by a Constant amount

(b) Decrease by a Constant factor or

(c) Decrease by a Variable factor

Decrease by Constant Amount:

In decrease and conquer approach, the problem domain is reduced by same constant amount at every individual step till problem arrives to desired result. In other words at every interaction of execution the main problem is reduced by some constant amount or factor. In most of the cases this constant amount has been assigned with integer value 1. Examples of the algorithms where decrease by constant factor is used to solve the problem are:

- Insertion sort
- Depth first search
- Breadth first Search
- Topological sorting
- Problems to generate permutations, subsets, etc

Decrease by a Constant Factor

In decrease and conquer approach, the problem domain is reduced by same constant factor on every occurrence of an iteration or step in program logic till a resulted is arrived. In most of the cases this constant amount has been assigned with integer value 2 and the reduction by constant factor other than two is very rare situation in an algorithm. Examples of the algorithms where decrease by constant factor is used to solve the problem are:

NOTES

- Binary Search
- Fake Coin Problem
- Russian Peasant Multiplication
- Josephus problem, etc.

Decrease by variable Size

In decrease and conquer approach, the main problem instance is reduced by variable size reduction factor at each individual step or iteration of an algorithm. In other words the reduction factor varies from one iteration to another. Examples of the algorithms where decrease by constant factor is used to solve the problem are:

- Euclid's algorithm for GCD
- Partition-based algorithm for selection problem
- Interpolation search
- Search and insertion in binary search trees, etc.

10.3 INSERTION SORT

Sorting is an algorithmic approach used to arrange the elements of a list or an array in some specific order say ascending or descending order. The order used to sort the array elements can be either numerical or lexicographical order. Selection and implementation of a proper sorting technique always adds efficiency in sorting process. There are various sorting techniques used to perform different sorting procedures. Few popular sorting techniques used frequently by programmers are bubble sort, selection sort, insertion sort, merge sort, quick sort, heap sort, radix sort and bucket sort. After implementing any of these sorting approaches, the random elements of an array can put in some sorting order however, all of these sorting approaches have varying efficiency towards their execution.

Insertion sort is one of the most used sorting approach where sorting of an array is performed by sorting one array element at a time only. Insertion sort is based on the principle that one element of the array is considered in each iteration of sorting process to locate its proper position in an array. The same procedure continues till all the elements of an array are visited, compared and sorted so that each element can hold the correct place in a sorted list. During sorting process the main array is divided into two sub-arrays that is left and right sub-array. Left sub-array is treated as the sorted array and the right sub-array as unsorted array. Each element from right sub-array is considered in every single iteration to get sorted and is inserted at proper position in sorted left sub-array. The sorting and inserting is performed sequentially. This sorting approach is not preffered for larger size arrays and its worst case complexity is $O(n^2)$.

NOTES

Working of Insertion Sort

In order to under the practical aspect of insertion sort lets consider the following example.

Let 'Array' is an unsorted array with following elements

Array[5]=[14,33,27,10,35,19]

The insertion sort in the above array begins by comparing the first two elements of Array[5], that is 14 and 33. While comparing 14 and 33 it is reported that the elements are already in sorted form (ascending order) therefore, no swapping is done and 14 becomes the first element of left sub-array(sorted array).

The next step is to compare 33 with 27. After comparing it is reported that 27 is smaller than 33. Therefore, the element 27 needs to be inserted at position that is before 33 by performing swapping. Before the element 27 is placed in sorted sub-array it checks and compares with all the elements within the left sorted sub-array so that the swapped element gets exact place in sorted array sub-array. The array 'Array[5]' will look like:

Array[5]=[14,27,33,10,35,19]

Now, the comparison will again begin from element 14 proceeds across 27 then 33 so on. As 14 is greater than 10 the elements need to be swapped at it will be placed at first location in sorted sub-array. The whole comparing, sorting and inserting an element by swapping will continue till a perfect sorted list is not obtained. After iterating across all elements the final sorted array 'Array[5]' will look like as;

Array[5]=[10,14,19, 27,33,35]

Algorithm Insertion sort

```
Step 1 - If it is the first element, it is already sorted. return 1;
Step 2 - Pick next element
Step 3 - Compare with all elements in the sorted sub-list
Step 4 - Shift all the elements in the sorted sub-list that is greater than the
         value to be sorted
Step 5 - Insert the value
Step 6 - Repeat until list is sorted
```

Psedocode of insertion sor

ALGORITHM *InsertionSort*$(A[0..n-1])$

//Sorts a given array by insertion sort
//Input: An array $A[0..n-1]$ of n orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
for $i \leftarrow 1$ **to** $n-1$ **do**
    $v \leftarrow A[i]$
    $j \leftarrow i-1$
    while $j \geq 0$ **and** $A[j] > v$ **do**
        $A[j+1] \leftarrow A[j]$
        $j \leftarrow j-1$
    $A[j+1] \leftarrow v$

```
Step 1 - If it is the first element, it is already sorted. return 1;
Step 2 - Pick next element
Step 3 - Compare with all elements in the sorted sub-list
Step 4 - Shift all the elements in the sorted sub-list that is greater than the
         value to be sorted
Step 5 - Insert the value
Step 6 - Repeat until list is sorted
```

Check Your Progress

1. What is insertion sort?
2. What is decrease and conquer?

10.4 DFS AND BFS

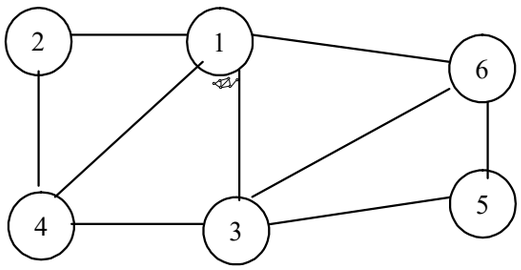
Traversing a Graph

One of the most common operations that can be performed on graphs is traversing, i.e., visiting all vertices that are reachable from a given vertex. The most commonly used methods for traversing a graph are depth-first search and breadth-first search.

10.4.1 Depth-First Search

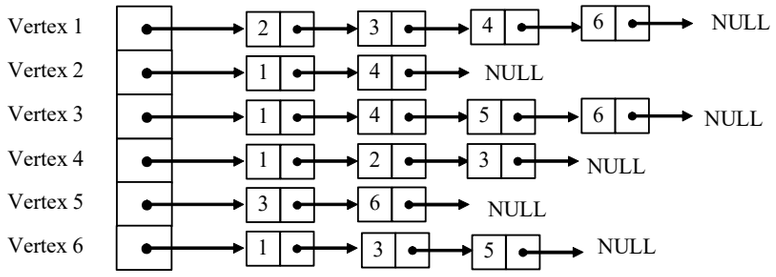
In depth-first search, starting from any vertex, a single path P of the graph is traversed until a vertex is found whose all-adjacent vertices have already been visited. The search then backtracks on path P until a vertex with unvisited adjacent vertices is found and then begins traversing a new path P' starting from that vertex, and so on. This process continues until all the vertices of graph are visited. It must be noted that there is always a possibility of traversing a vertex more than one time. Thus, it is required to keep track whether the vertex is already visited.

For example, the depth-first search for a graph, shown in Figure 10.1(a), results in a sequence of vertices 1, 2, 4, 3, 5, 6, which is obtained as follows:



(a) Graph

NOTES



(b) Adjacency list of graph

Fig. 10.1 *Graph and Its Adjacency List*

1. A vertex is visited, for example, 1.
2. Its adjacent vertices are 2, 3, 4, and 6. Any unvisited vertex can be selected, for example, 2.
3. Its adjacent vertices are 1 and 4. Since vertex 1 is already visited, an unvisited vertex 4 is selected.
4. Its adjacent vertices are 1, 2, and 3. Since vertices 1 and 2 are already visited, an unvisited vertex 3 is selected.
5. Its adjacent vertices are 1, 4, 5, and 6. Since vertices 1 and 4 are already visited, an unvisited vertex, for example 5, is selected.
6. Its adjacent vertices are 3 and 6. An unvisited vertex 6 is selected. Since all the adjacent vertices of vertex 6 are already visited, visited adjacent vertices are backtracked to find if there is any unvisited vertex. As all the vertices are visited, the algorithm is terminated.

To implement depth-first search, an array of pointers `arr_ptr` is maintained, which stores the address of the first vertex in each adjacency list and a boolean-valued array visited to keep track of the visited vertices. Initially, all the values of visited array are initialized to False to, indicating that no vertex has yet been visited. As soon as a vertex is visited, its value changes to True in visited array. The recursive algorithm for depth-first search has been illustrated here.

Algorithm 10.1: Depth-First Search (Recursive)

```
void depth_first_ search(v, arr_ptr)   //v is the vertex of
the graph
1. Set visited[v] = True    //mark first vertex as visited
2. Print v
3. Set ptr = *(arr_ptr+v)
             //assign address of adjacency list of vertex v to
   ptr
4. While ptr != NULL
   If visited[ptr->info] = False  //check if vertex is not
visited
      Call depth_first_search(ptr->info, arr_ptr)
                  //call depth_first_search
   Else
      ptr = ptr->next;
   End If
   End While
5. End
```

NOTES

Example 10.1: A program to illustrate the depth-first search algorithm is as follows:

```
#include <stdio.h>
#include <conio.h>
#define True 1
#define False 0
#define MAX 10
typedef struct node
{
    int info;
    struct node *next;
}Node;
int visited[MAX];     /* global variable; all the values
are                         initialized to 0 */
void create_graph(Node *[], int);
void input(Node *[], int);
void depth_first_search(int, Node *[]);
void display(Node *[], int);
void main()
{
    Node *arr_ptr[MAX];     /* array of pointers to node
                               type structures */
    int nvertex;
    clrscr();
    printf("\nEnter the number of vertices in Graph: ");
    scanf("%d", &nvertex);
    create_graph(arr_ptr, nvertex);
    input(arr_ptr, nvertex);
    printf("\nValues are inputted in the graph");
    display(arr_ptr, nvertex);
    printf("\n\nDepth First Search is:\t");
    depth_first_search(1, arr_ptr);
    getch();
  }
void create_graph(Node *arr_ptr[],
int num)        /* to create an empty graph, the entire
                */
```

NOTES

```
 {              /* adjacency list is initilialized
                        with NULL */
   int i;
   for(i=1; i<=num; i++)
   arr_ptr[i]=NULL;
 }
 void input(Node *arr_ptr[], int num)
 {
    Node *nptr,*save;
    int i,j,num_vertex,item;
    for(i=1; i<=num; i++)
    {
       printf("Enter the no. of vertices in adjacency
list a[%d]             : ",i);
       scanf("%d", &num_vertex);
       for(j=1; j<=num_vertex; j++)
       {
         printf("Enter the value of vertex : ");
         scanf("%d", &item);
         nptr=(Node*)malloc(sizeof(Node));
         nptr->info=item;
         nptr->next=NULL;
         if(arr_ptr[i]==NULL)
            arr_ptr[i]=last=nptr;
         else
         {
            save->next=nptr;
            save=nptr;
         }
       }
    }
 }
void display(Node *arr_ptr[], int num)
{
   int i;
```

NOTES

```
    Node *ptr;
    printf("\n\nGraph is:\n");
    for(i=1;i<=num;i++)
    {
       ptr=arr_ptr[i];
       printf("\na[%d]   ",i);
       while(ptr != NULL)
       {
        printf(" -> %d", ptr->info);
        ptr=ptr->next;
       }
    }
 }
void depth_first_search(int v, Node *arr_ptr[])
{
    Node *ptr;
    visited[v]=True;  /* mark first vertex as visited */
printf("\%d\t", v);
ptr=*(arr_ptr+v);      / * assign address of adjacency
list to                                    ptr */
 while(ptr!=NULL)
    {
       if(visited[ptr->info]==False)
          depth_first_search(ptr->info, arr_ptr);
       else
          ptr=ptr->next;
    }
}
```

The output of the program is as follows:

```
Enter the number of vertices in Graph: 6
Enter the no. of vertices in adjacency list a[1] : 4
Enter the value of vertex : 2
Enter the value of vertex : 3
Enter the value of vertex : 4
```

NOTES

```
Enter the value of vertex : 6
Enter the no. of vertices in adjacency list a[2] : 2
Enter the value of vertex : 1
Enter the value of vertex : 4
Enter the no. of vertices in adjacency list a[3] : 4
Enter the value of vertex : 1
Enter the value of vertex : 4
Enter the value of vertex : 5
Enter the value of vertex : 6
Enter the no. of vertices in adjacency list a[4] : 3
Enter the value of vertex : 1
Enter the value of vertex : 2
Enter the value of vertex : 3
Enter the no. of vertices in adjacency list a[5] : 2
Enter the value of vertex : 3
Enter the value of vertex : 6
Enter the no. of vertices in adjacency list a[6] : 3
Enter the value of vertex : 1
Enter the value of vertex : 3
Enter the value of vertex : 5
Values are inputted in the graph
Graph is:
a[1]    -> 2 -> 3 -> 4 -> 6
a[2]    -> 1 -> 4
a[3]    -> 1 -> 4 -> 5 -> 6
a[4]    -> 1 -> 2 -> 3
a[5]    -> 3 -> 6
a[6]    -> 1 -> 3 -> 5

Depth First Search is:  1      2      4      3      5
6
```

It must be noted that a depth-first search can also be implemented non-recursively by using a stack explicitly. In this implementation, all unvisited vertices—adjacent to the one being visited—are placed onto a stack and then the TOP element of the stack is popped to find the next vertex to visit. This process is repeated until the stack is empty.

10.4.2 Breadth-First Search

In a breadth-first search, starting from any vertex, all the adjacent vertices are traversed. Then any one of the adjacent vertices is selected and all the other adjacent vertices, that have not been visited yet, are traversed. This process

continues until all the vertices have been visited. For example, the breadth-first search for the graph shown in Figure 10.7(a) results in a sequence 1, 2, 3, 4, 6, 5 which is obtained as follows:

1. Visit a vertex, for example, 1.
2. Its adjacent vertices are 2, 3, 4 and 6. All these vertices are visited one by one. Any vertex, for example, 2 is selected and its adjacent vertices are found.
3. Since all the adjacent vertices for 2, *i.e.,* 1 and 4 are already visited. Vertex 3 is selected.
4. Its adjacent vertices are 1, 5 and 6. Vertex 5 is visited as 1 and 6 were already visited.
5. Since all the vertices have been visited, the process is terminated.

The implementation of breadth-first search is quite similar to the implementation of depth-first search. The difference is that the former uses a queue instead of a stack (either implicitly via recursion or explicitly) to store the vertices of each level as they are visited. These vertices are then taken one by one and their adjacent vertices are visited and so on until all the vertices have been visited. The algorithm terminates when the queue becomes empty.

Algorithm 10.2: Breadth-First Search

```
void breadth_first_search(arr_ptr)

1. Set v = 1
2. Set visited[v] = True   //mark first vertex as visited
3. Print v
4. call qinsert(v)         //insert this vertex in queue
5. While isqempty() = False   // check if there is element in queue
       Call qdelete()    // returning an integer value v
       Set ptr=*(arr_ptr+v)
              //assign address of adjacency list to ptr, ptr is a
       pointer of type node
       While(ptr != NULL)
              If visited[ptr->info] = False
                     Call qinsert(ptr->info)
                     Set visited[ptr->info] = True
                     Print ptr->info
              End If
       End While
       Set ptr = ptr->next
     End While
6. End
```

Example 10.2: A program to illustrate the breadth first search algorithm is as follows:

```
#include <stdio.h>
#include <conio.h>
#define MAX 10
#define True 1
```

NOTES

```
#define False 0
typedef struct node
{
    int info;
    struct node *next;
}Node;

int visited[MAX];
int queue[MAX];
int Front, Rear;
void create_graph(Node *[], int num);
void input(Node *[], int num);
void breadth_first_search(Node *[]);
void qinsert(int);
int qdelete();
int isqempty();
void display(Node *[], int num);
void main()
{
    Node *arr_ptr[MAX];
    int nvertex;
    clrscr();
    printf("\nEnter the number of vertices in Graph: ");
    scanf("%d", &nvertex);
    create_graph(arr_ptr, nvertex);
    input(arr_ptr, nvertex);
    printf("\nValues are inputted in the graph");
    display(arr_ptr, nvertex);
    Front=Rear=-1;
    breadth_first_search(arr_ptr);
    getch();
  }
  void create_graph(Node *arr_ptr[], int num)
  {
       int i;
       for(i=1; i<=num; i++)
          arr_ptr[i]=NULL;
  }

  void input(Node *arr_ptr[], int num)
```

NOTES

```
 {
    Node *nptr,*save;
    int i,j,num_vertex,item;
    for(i=1; i<=num; i++)
    {
       printf("Enter the no. of vertices in adjacency
list          a[%d] : ",i);
       scanf("%d", &num_vertex);
       for(j=1; j<=num_vertex; j++)
       {
         printf("Enter the value of vertex : ");
         scanf("%d", &item);
         nptr=(Node*)malloc(sizeof(Node));
         nptr->info=item;
         nptr->next=NULL;
         if(arr_ptr[i]==NULL)
            arr_ptr[i]=save=nptr;
         else
         {
          save->next= nptr;
          save=nptr;
         }
       }
    }
 }
void display(Node *arr_ptr[], int num)
{
   int i;
   Node *ptr;
   printf("\n\nGraph is:\n");
   for(i=1;i<=num;i++)
   {
      ptr=arr_ptr[i];
      printf("\na[%d]   ",i);
      while(ptr != NULL)
      {
          printf(" -> %d", ptr->info);
         ptr=ptr->next;
      }
   }
}
```

NOTES

```
void breadth_first_search(Node *arr_ptr[])
{
    Node *ptr;
    int v=1;
    visited[v]=True;          //mark first vertex as visited
    printf("\nBreadth First Search: %d\t", v);
    qinsert(v);               //insert this vertex in queue
    while(isqempty()==False)
    {
       v=qdelete();
       ptr=*(arr_ptr+v);      //assign address of adjacency
                              list to ptr
       while(ptr != NULL)
       {
          if(visited[ptr->info]==False)
          {
             qinsert(ptr->info);
             visited[ptr->info]=True;
             printf("%d\t", ptr->info);
          }
       ptr=ptr->next;
       }
}
}
void qinsert(int vertex)
{
    if (Rear==6)
    {
       printf("Overflow! Queue is Full");
       exit();
    }
     queue[++Rear]=vertex;
     if (Front==-1)
       Front=0;
 }
int qdelete()
{
    int item;
    if (Front==-1)
    {
```

```
        printf("Underflow! Queue is empty");
        exit();
    }
    item=queue[Front];
    if (Front==Rear)
        Front=Rear=-1;
    else
        Front++;
    return item;
}

int isqempty()
{
    if (Front==-1)
        return True;
    return False;
}
```

The output of the program is as follows:

```
Enter the number of vertices in Graph: 6
Enter the no. of vertices in adjacency list a[1] : 4
Enter the value of vertex : 2
Enter the value of vertex : 3
Enter the value of vertex : 4
Enter the value of vertex : 6
Enter the no. of vertices in adjacency list a[2] : 2
Enter the value of vertex : 1
Enter the value of vertex : 4
Enter the no. of vertices in adjacency list a[3] : 4
Enter the value of vertex : 1
Enter the value of vertex : 4
Enter the value of vertex : 5
Enter the value of vertex : 6
Enter the no. of vertices in adjacency list a[4] : 3
Enter the value of vertex : 1
Enter the value of vertex : 2
Enter the value of vertex : 3
Enter the no. of vertices in adjacency list a[5] : 2
Enter the value of vertex : 3
Enter the value of vertex : 6
```

NOTES

NOTES

```
Enter the no. of vertices in adjacency list a[6] : 3
Enter the value of vertex : 1
Enter the value of vertex : 3
Enter the value of vertex : 5
Values are inputted in the graph
Graph is:
a[1]    -> 2 -> 3 -> 4 -> 6
a[2]    -> 1 -> 4
a[3]    -> 1 -> 4 -> 5 -> 6
a[4]    -> 1 -> 2 -> 3
a[5]    -> 3 -> 6
a[6]    -> 1 -> 3 -> 5
Breadth First Search: 1   2   3   4   6   5
```

10.5 TOPOLOGICAL SORTING

Applications of Graphs

Graphs have various applications in diverse areas. Various real-life situations like traffic flow, analysis of electrical circuits, finding shortest routes, applications related with computation, etc., can easily be managed by using graphs. Some of the applications of graphs like topological sorting and minimum spanning trees have been discussed in the following section.

10.5.1 Topological Sorting

The topological sort of a directed acyclic graph is a linear ordering of the vertices such that if there exists a path from vertex x to y, then x appears before y in the topological sort. Formally, for a directed acyclic graph G = (V, E), where V = $\{V_1, V_2, V_3, \ldots, V_n\}$, if there exists a path from any V_i to V_j then V_i appears before V_j in the topological sort. An acyclic directed graph can have more than one topological sorts. For example, two different topological sorts for the graph illustrated in Figure 10.2 are (1, 4, 2, 3) and (1, 2, 4, 3).

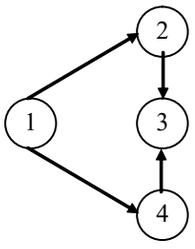


Fig. 10.2 *Acyclic Directed Graph*

Clearly, if a directed graph contains a cycle, the topological ordering of vertices is not possible. It is because for any two vertices V_i and V_j in the cycle, V_i precedes V_j as well as V_j precedes V_i. To this, exemplify let us study the simple cyclic directed graph shown in Figure 10.3. The topological sort for this graph is (1, 2, 3, 4) assuming the vertex 1 as starting vertex. Since, there exists a path from vertex 4 to 1 then according to the definition of a topological sort, vertex 4 must appear before vertex 1, which contradicts the topological sort generated for this graph. Hence, topological sort can exist only for an acyclic graph.

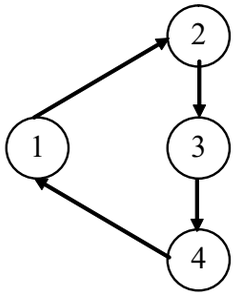


Fig. 10.3 *Cyclic Directed Graph*

In an algorithm to find the topological sort of an acyclic directed graph, the indegree of the vertices is considered. Following are the steps that are repeated until the graph is empty:

1. Any vertex V_i with 0 indegree is selected.
2. Vertex V_i is added to the topological sort (initially, the topological sort was empty).
3. Vertex V_i is removed along with its edges from the graph and the indegree of each adjacent vertex of V_i is reduced by one.

To illustrate this algorithm, an acyclic directed graph is shown in Figure 10.4 as follows:

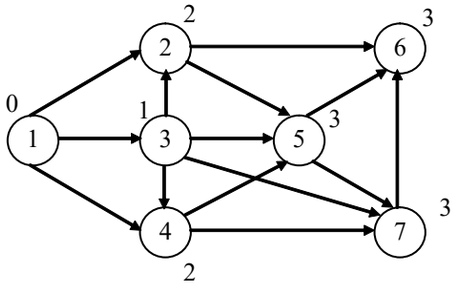
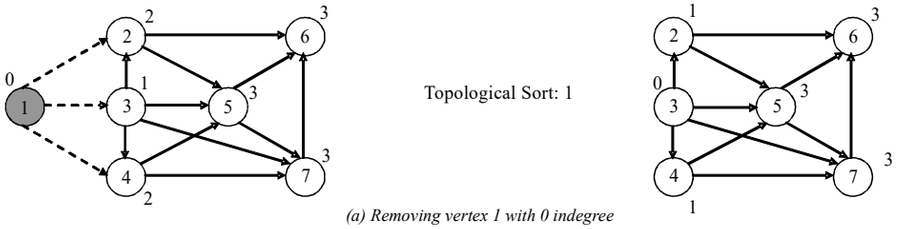


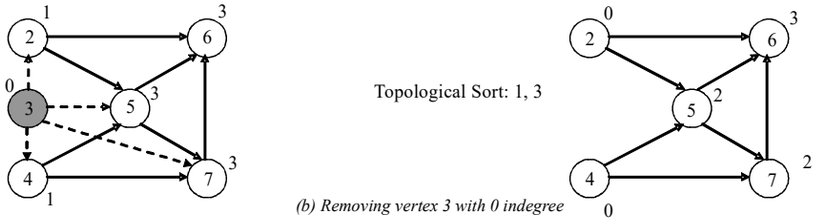
Fig. 10.4 *Acyclic Directed Graph*

NOTES

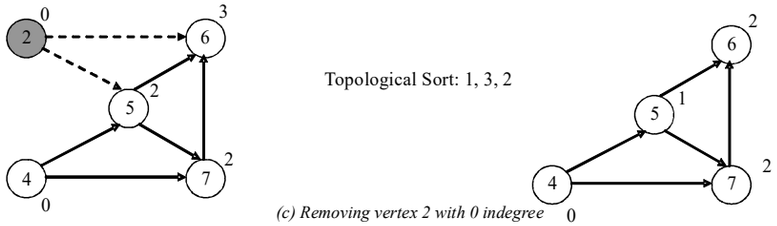
The steps for finding topological sort for this graph are shown in Figure 10.5 as follows:



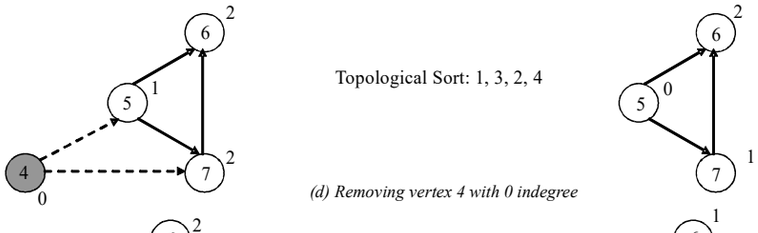
(a) Removing vertex 1 with 0 indegree



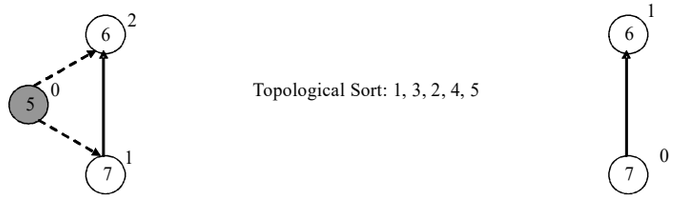
(b) Removing vertex 3 with 0 indegree



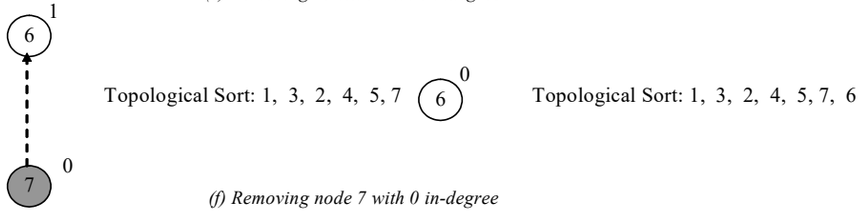
(c) Removing vertex 2 with 0 indegree



(d) Removing vertex 4 with 0 indegree



(e) Removing vertex 5 with 0 indegree



(f) Removing node 7 with 0 in-degree

Fig. 10.5 *Steps for Finding Topological Sort*

NOTES

Another possible topological sort for this graph is (1, 3, 4, 2, 5, 7, 6). Hence, it can be concluded that the topological sort for an acyclic graph is not unique. Topological ordering can be represented graphically. In this representation, edges are also included to justify the ordering of vertices as shown in Figure 10.6.

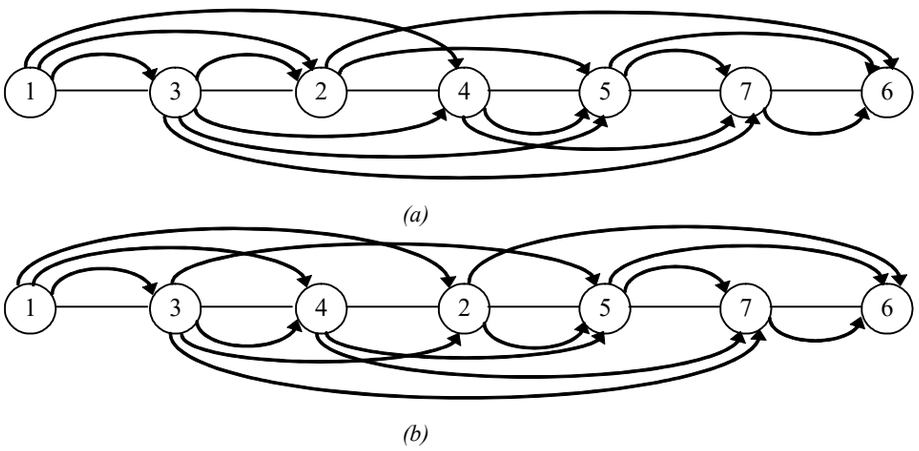


Fig. 10.6 *Graphical Representation of Topological Sort*

Topological sort is useful for proper scheduling of various sub tasks to be executed for completing a particular task. In computer field, it is used for scheduling instructions. For example, consider a task in which smaller number is to be subtracted from a larger one. The set of instructions for this task is as follows:

```
1.  If A>B then goto Step 2, else goto Step 3
2.  C = A-B, goto Step 4
3.  C = B-A, goto Step 4
4.  Print C
5.  End
```

The two possible scheduling orders to accomplish this task are (1, 2, 4, 5) and (1, 3, 4, 5). From this, it can be concluded that instruction 2 cannot be executed unless instruction 1 is executed before it. Moreover, these instructions are non repetitive hence they are acyclic in nature.

Check Your Progress

3. What is done to implement depth-first search?
4. List a few applications of graphs.

10.6 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

NOTES

1. Insertion sort is one of the most used sorting approach where sorting of an array is performed by sorting one array element at a time only.
2. Decrease and Conquer is a problem solving approach especially used to perform searching and sorting operations.
3. To implement depth-first search, an array of pointers arr_ptr is maintained.
4. A few applications of Graphs include traffic flow, analysis of electrical circuits, finding shortest routes and applications related to computation.

10.7 SUMMARY

- One of the most common operations that can be performed on graphs is traversing, i.e., visiting all vertices that are reachable from a given vertex.
- In depth-first search, starting from any vertex, a single path P of the graph is traversed until a vertex is found whose all-adjacent vertices have already been visited.
- To implement depth-first search, an array of pointers arr_ptr is maintained, which stores the address of the first vertex in each adjacency list and a boolean- valued array visited to keep track of the visited vertices.
- In a breadth-first search, starting from any vertex, all the adjacent vertices are traversed.
- The implementation of breadth-first search is quite similar to the implementation of depth-first search.
- In an algorithm to find the topological sort of an acyclic directed graph, the indegree of the vertices is considered.
- Topological sort is useful for proper scheduling of various sub tasks to be executed for completing a particular task.
- A spanning tree of a connected graph G is a tree that covers all the vertices and the edges required to connect those vertices in the graph.
- For a connected weighted graph G, it is required to construct a spanning tree T such that the sum of weights of the edges in T is minimum.
- In Kruskal's approach, initially, all the vertices n of a graph are considered as a distinct partial tree having one vertex and all its edges are listed in increasing order of their weights.
- Kruskal's algorithm requires listing of edges in the increasing order of their weights and at each step one needs to determine whether the inclusion of a new edge will result in a cycle or not.

- According to Prim's algorithm, the minimum spanning tree is constructed in a sequential manner.

10.8 KEY WORDS

NOTES

- **Traressing:** It is one of the most common operations that can be performed on graphs.
- **Minimum Spanning Tree:** It is a spanning tree whose sum of weights of the edges in T is minimum.

10.9 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. What do you mean by decrease and conquer?
2. Explain the functionality of insertion sort.
3. Discuss topological sorting and its use.

Long Answer Questions

1. "In depth-first search, starting from any vertex, a single path P of the graph is traversed until a vertex is found whose all-adjacent vertices have already been visited." Elaborate.
2. Give points of differentiation between DFS and BFS.
3. Explain topological sorting on a graph.

10.10 FURTHER READINGS

Levitin, Anany. *Introduction to Design and Analysis of Algorithms*. Delhi: Pearson Education.

Ellis Horowitz, S. Sahani and Rajasekaran, *Fundamentals of Computer Algorithms.* Delhi: Galgotia Publications.

Goodrich, M T and R. Tomassia. *Algorithm Design: Foundations, Analysis and Internet Examples*. Delhi: John wiley and Sons.

NOTES

UNIT 11 GENERATING COMBINATORIAL OBJECTS

Structure

11.0 INTRODUCTION

Important combinatorial objects are: permutations, combinations, and subsets of a set. We can assume that the objects to permute are consecutive integers. We assume this because the integers can represent the indices into an array. The number of permutations is n!. If the set contains the elements a0, a1, ..., an-1, then we can represent a subset as a binary number of length n, each bit being 0 if the corresponding element is not in the set, and 1 if it is. This unit will expain about combinatorial objects in detail.

11.1 OBJECTIVES

After going through this unit, you will be able to:

- Discuss about general combinational objects
- Understand the mechanism behind transform and conquer
- Differentiate between heap and heap sort

11.2 GENERATING COMBINATIONAL OBJECTS

Combinational objects are characterized as an object that can be put into one-to-one correspondence with finite set of integers.

The combinational analysis is a part of mathematics which instructs one to find out and show all the possible patterns by which a given combination of number

of things might be related and combined with the goal that one might be sure that he has not missed any collection or arrangement of the conceivable things that has not been counted. Combinatorial Objects results into the generation of : all subsets of a given set, all possible permutations of numbers in set and partition of an integer n into k parts.

In order to under this concept let's consider the following example to generate subsets from a given collection of numbers that constitute a set.

- In the event that the set contains the elements a0,a1,….,an-1,an. And a subset is expressed as binary string of length equal to the size of set say n. Let's assume a set S={1,2,3} where n=3. Each bit is assigned as 0 if that corresponding bit from the set is dropped in subset and if selected then it is assigned by bit 1.
- Therefore, all the combinations of bits starting from null set that is s={0,0,0} to {1,2,3} that is numbers from 0 to 2n-1 will be generated.
- The procedure continues up to the generation of all possible subsets.

Example

Rank	Binary	Subset
0	000	{}
1	001	{3}
2	010	{2}
3	011	{2,3}
4	100	{1}
5	101	{1,3}
6	110	{1,2}
7	111	{1,2,3}

Another example can be simply by generating permutations of a number.

11.3 TRANSFORM AND CONQUER

Transform and conquer is an approach wherein a given problem domain is transformed to other domain to obtain the solution of the original problem domain. The main aim to transform from one domain to another is avoid the complexity involved in the problem if solved in its original form by implementing the simplicity and familiarity of the transformed domain to arrive at solution. Once the solution in the transformed domain is obtained it is latter on changed / converted to its original domain.

A simple example to understand is to add to two roman numbers say III and IX. In order to solve this roman numbers are transformed to Arabic numerical and thereafter addition is performed and obtained result is changed back to roman form.

III = 3, IX = 9

III + IX $\rightarrow$ 3 + 9 = 12

12 $\rightarrow$ XII

NOTES

There are three different variants of transform and conquer these are:

(a) Instance simplification

(b) Representational Change

(c) Problem Reduction

(a) ***Instance Simplification***: In this type of variant the original problem domain is transformed to simpler and familiar domain to increase the understandability and convenience in problem solving. Presorting is an example of instance specification.

(b) ***Representational Change***: In this type of variant the presentational aspect of the domain instance is changed to another presentation to make the problem solving more efficient and easy.

(c) ***Problem Reduction***: In this approach of transform and conquer wherein a problem domain is transformed to another problem domain for which an algorithm to obtain the solution is already existing. Example for this is to compute LCM via GCD or reduction of graph problem.

11.3.1 Presorting

Presorting is defined as sorting an array before actually processing it. It is an old concept was data is sorted to increase easiness in obtaining the solution. Let's assume a list of items when it is unsorted, to perform presorting on this unsorted or random list one can:

- Perform efficient searching an item in list.
- Can calculate median easily
- Check the frequency of numbers to find any repetition or uniqueness.
- It is also used to solve many geometric problems also.

Searching with Presorting

Searching using presorting is usually carried out in two steps:

Step 1: Sort the array by available sorting approach.

Step 2: Apply binary Search.

Finding Uniqueness of the element using presorting

Again to reveal the uniqueness or frequency of an element in array or list follows a two-step process:

Step 1: Sort the array or list by available sorting approach.

Step 2: Scanning each individual element of list to check their uniqueness with adjacent elements in an array or list.

Algorithm for finding uniqueness of element

```
Begin
 Algorithm Sort_Array( )
For i=0 to size-1
If Array[i]=Array[i+1] then
 Return not unique element
Else
Return Unique element
End
```

11.3.2 Heap

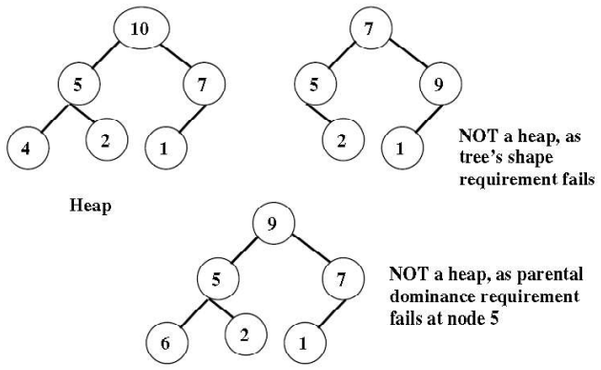
Definition of Heap: Heap data (binary) structure is an array of objects that can be viewed as a nearly complete binary tree or a heap is a binary tree represented as an array of objects with the conditions that it is essentially complete and the key at each node is ≥ keys at its children nodes.

A heap is a specialized tree-based data structure which is essentially an almost complete tree that satisfies the heap property such that in a max heap, for any given node C, if P is a parent node of C, then the key (the value) of P is greater than or equal to the key of C. In a min heap, the key of P is less than or equal to the key of C. The node at the 'Top' of the heap (with no parents) is called the 'Root' node.

Heap is, therefore, a binary tree with nodes of the tree which are assigned with some keys and must satisfy the following criteria:

Tree's Structure or Shape: Binary tree is essentially complete that means the tree is completely filled on all levels with a possible exception where the lowest level is filled from left to right, and some rightmost leaves may be missing.

Heap Order or Parental Dominance Requirements: For every node 'i' in a binary tree, except the root node the value stored in 'i' is greater than or equal to the values of its children node.



NOTES

In other words heap can be defined as a complete binary tree with the property that the value at each node is at least as large(as small as) the values at its children(if exists). This property is also called as heap property.

Heap construction

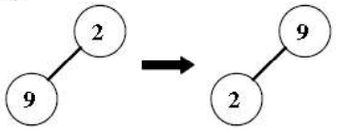
Construct heap for the list: 2, 9, 7, 6, 5, 8

There are two ways that can be used for heap construction:

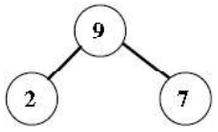
1. Top-down construction: Constructs a heap by successive insertions of a new key into a previously constructed heap.

a. Insert 2 into empty tree.

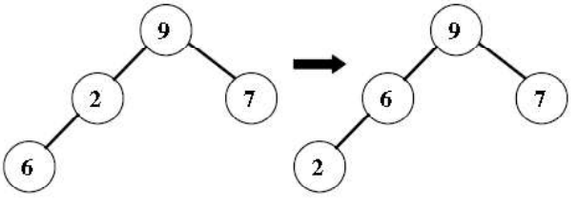
b. Insert 9



c. Insert 7



d. Insert 6



2. Bottom-up Heap Construction

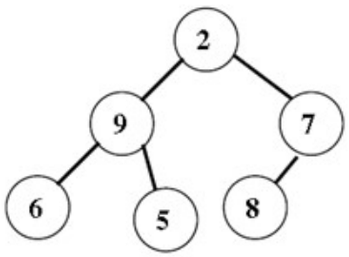
/constructs a heap from the elements of a given array by bottom-up algorithm

//i/p: An array H[1…n] of orderable items

//o/p: A heap H[1…n]

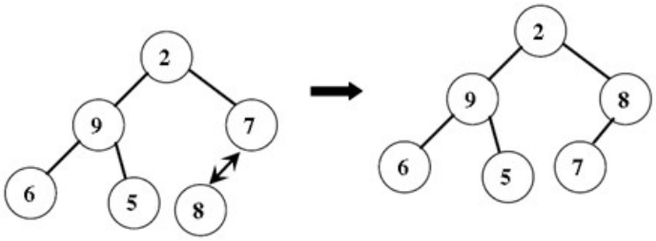
```
for i ← ⌊n/2⌋ down to 1 do
        k ← i
        v ← H[k]
        heap ← false
        while NOT heap AND 2*k ≤ n do
                j ← 2 * k
                if j < n
                        if H[j] < H[j+1]
                                j ← j + 1
                if v ≥ H[j]
                        heap ← true
                else
                        H[k] ← H[j]
                        K ← j
        H[k] ← v
```

Initialize the essentially complete binary tree with n nodes by placing keys in the order given and then heapify the tree.

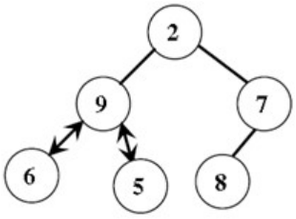


Heapify

Compare 7 with its children

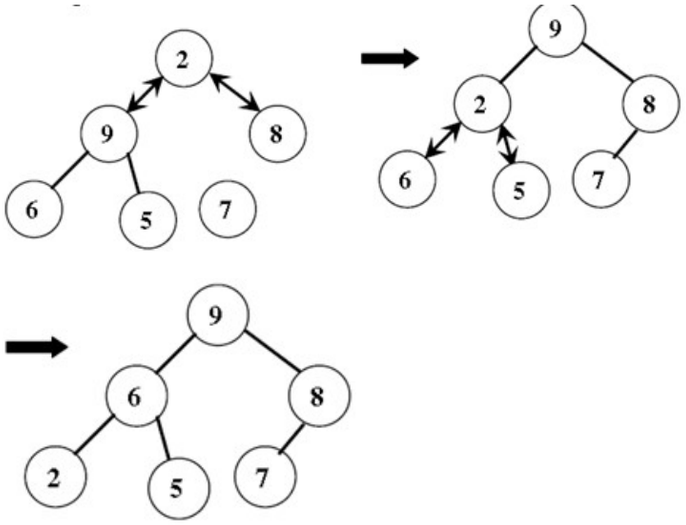


b. Compare 9 with its children



NOTES

C. Compare 2 with its children



Properties of Heaps

1. The height of a heap is floor(lg n).
2. The root node of heap has highest priority item.
3. A node and all the descendants is a heap.
4. Array can be used to implement heap and all operations applicable on arrays can be performed on heap.
5. If root node is indexed as 1 then left child has 2i and right child has 2i+1 as their corresponding indexes.
6. At any level i of heap, heap possesses 2i elements.
7. In heap the value of the parent node is always higher than its children nodes.

Heapsort

Stage 1: Construct a heap for a given list of n keys

Stage 2: Repeat operation of root removal n-1 times:

Exchange keys in the root and in the last (rightmost) leaf

Decrease heap size by 1

If necessary, swap new root with larger child until the heap condition holds

Analysis of Heapsort

HEAPSORT(A)

1 BUILD-HEAP(A)

2 for i -> length[A] downto 2

3 do exchange A[1] <-> A[i]

4 heap-size[A] <- heap-size[A] -1

5 HEAPIFY(A, 1)

NOTES

Check Your Progress

1. What are combinational objects characterized as?
2. What is presorting defined as?

11.4 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. Combinational objects are characterized as an object that can be put into one-to-one correspondence with finite set of integers.
2. Presorting is defined as sorting an array before actually processing it.

11.5 SUMMARY

- Combinational objects are characterized as an object that can be put into one-to-one correspondence with finite set of integers.
- The combinational analysis is a part of mathematics which instructs one to find out and show all the possible patterns by which a given combination of number of things might be related and combined
- In the event that the set contains the elements a0,a1,….,an-1,an. And a subset is expressed as binary string of length equal to the size of set say n.
- The main aim to transform from one domain to another is avoid the complexity involved in the problem.
- Once the solution in the transformed domain is obtained it is latter on changed/ converted to its original domain.
- There are three different variants of transform and conquer these are:
  (i) Instance simplification
  (ii) Representational Change
  (iii) Problem Reduction
- Presorting is defined as sorting an array before actually processing it.
- Binary tree is essentially complete that means the tree is completely filled on all levels with a possible exception where the lowest level is filled from left to right, and some rightmost leaves may be missing.

NOTES

11.6 KEY WORDS

- **Instance Simplification:** In this type of variant the original problem domain is transformed to simpler and familiar domain to increase the understandability and convenience in problem solving.
- **Representational Change:** In this type of variant the presentational aspect of the domain instance is changed to another presentation to make the problem solving more efficient and easy.
- **Problem Reduction:** In this approach of transform and conquer wherein a problem domain is transformed to another problem domain for which an algorithm to obtain the solution is already existing.

11.7 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. Write a short note on Instance Simplification.
2. What do you mean by representational change?
3. Write a short note on transform and conquer.
4. What is presorting?

Long Answer Questions

1. "Transform and conquer is an approach wherein a given problem domain is transformed to other domain to obtain the solution of the original problem domain." Discuss in detail.
2. "Heap is a binary tree where nodes of the tree are assigned with some keys and they must satisfy a few criteria." Elaborate and list these.
3. What do you mean by heap construction? Write a detailed note.

11.8 FURTHER READINGS

Levitin, Anany. *Introduction to Design and Analysis of Algorithms*. Delhi: Pearson Education.

Ellis Horowitz, S. Sahani and Rajasekaran, *Fundamentals of Computer Algorithms.* Delhi: Galgotia Publications.

Goodrich, M T and R. Tomassia. *Algorithm Design: Foundations, Analysis and Internet Examples*. Delhi: John wiley and Sons.

UNIT 12 OPTIMIZATION PROBLEMS

NOTES

Structure

12.0 INTRODUCTION

Optimization problem is a mathematical problem or any computational domain where the main purpose of the problem solution is to reveal the best possible solution of the problem among the all possible outcomes of problem. In other words optimization problems involve to find the feasible solution space which has either the minimum or maximum value of the object function. For example, the travelling salesman problem is an optimization problem in which a salesman tries to find the best and feasible path(minimum cost path) among all possible ways to perform efficient sales by effectively traverse across nodes with minimum travelling cost. This unit will explain about optimization problems.

12.1 OBJECTIVES

After going through this unit, you will be able to:

- Understand reductions and its application
- Discuss about travelling salesperson problem
- Analyze bounding and its implication

12.2 REDUCTIONS

This is a problem solving approach in which a main problem is transformed into a simple problem by implementing various transforming approaches like transform-and-conquer, divide-and-conquer or decrease-and-conquer. The main purpose

NOTES

of reducing the main problem instance into smaller sub-problem is to increase the simplicity, reduce complexity so that the deduction of the solution becomes easy. Suppose there is a problem 'P' for which one has to find the possible solution. Using reduction approach breaks the problem 'P' into simpler sub-problems say S_1, S_2, .., S_n. The problem solving proceeds by obtaining a solution for sub-problems and thereafter, consequently the solution of the main problem.

Example 12.1: Let's suppose you have an algorithm A1 to solve problem P1. Meantime, another problem P2 has arrived, the nature of problem P2 is similar to that of previous problem P1. In order to obtain the solution for Problem P2 following approaches can be used:

- The solution of problem P2 can be started from scratch.
- You try to equate or extend the use algorithm A1 to solve.
- You try to reduce or transform the problem P2 to P1 thereafter it can be simply solved using Algorithm A1.
- The reduction of problem P2 to P1 further involve following:
  - o Transforming all inputs to problem P_2 into inputs to problem P_1.
  - o It also solves the problem P_2 using algorithm A_1 which are actually meant for problem P_1.
  - o Treats all outputs obtained using A_1 as output to problem P_2.

In order words a problem P1 can be transformed or reduced to Problem P2 if there is a special function say *f* that which accepts any input '*i*' to P_1 and transforms that input to this function *f(i)* of Problem P_2, such that the solution obtained for problem P_2 using *f(i)* can be treated as solution for problem P_1 on '*i*'. This is also expressed in Figure 12.1 below:

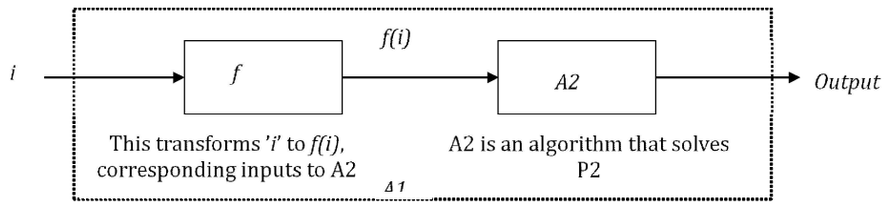


Fig. 12.1 *A1 is an algorithm to solve P1*

Example 12.2:

Case (a): performing multiplication of two matrices M1 and M2 and return the result of multiplication.

Case (b): Performing a squaring operation of matrix M the result is squared matrix M.

The algorithm that is used to perform multiplication of two matrices can also be used to perform the squaring of a matrix by reducing the matrix for which squaring is to be done into two matrices and then implement the algorithm as was used for matrix multiplication.

12.3 REDUCTION TO GRAPH PROBLEMS

There are number of problems that can be efficiently solved by extending to reduction approach as discussed above. The reduction approach is not only limited to liner or two or three dimensional problems only but can be used to solve graphical problems also. To solve the problem of arranging eight queens problem or to find minimum cost path in travelling salesperson problem or Knapsack problems can be easily resolved using reduction approach on their graphical problem presentation. In order to understand reduction to graph problems let's consider travelling salesperson problem in detail.

12.4 TRAVELLING SALESPERSON PROBLEM

Travelling salesperson problem describes a graphical solution for the sales person to perform effective sales by travelling all the connected cites without repeating any city and finally returning to its original city from where he started his sales travel. The path the salesperson selects or chooses must be a minimum cost route/ path. Let's reduce this whole situation by constructing a directed graph G(V,E) which defines the particular instance of the travelling salesperson main problem domain. Let c_{ij} is the cost required by salesperson to travel from node 'i' to node 'j', in other words cij is cost of edge(i,j). Nodes are represented by vertices of graph where V is the set of all vertices in graph G(V,E) and $V=\{v_1,v_2,\ldots,v_n\}$.Each edge in graph G(V,E) is assigned a particular weight that represents a cost of travel associated with that corresponding edge(i,j). Let's assume that the initial vertex from where salesperson starts his travel is labeled as $v_{1,}$ therefore, the solution space of the problem S is expressed as S={1,X,1}. Where 'X' is all possible permutations of intermediate nodes {2,3,..,n}.

In order to solve traveling salesman problem to find the minimum cost path starting from some node v_1 and returning back to the same node without repeating any intermediate node. Reducing the travelling salesman's problem using branch and bound algorithm the cost matrix describing the cost factor that is associated with various paths that can be opted during his travel is also reduced. The cost matrix is reduced if and only if a row of the matrix has at least one zero and remaining entries are non-negative numbers. By other definition a matrix is said to be reduced if and only if its every row and column is reduced (that is contains at least one zero).

NOTES

12.4.1 Branching

The branching part of the algorithm works by dividing the solution space into two sub-graphs or groups. In more accurate senses each node divides the solution (remaining solution) into two halves wherein, one some nodes are included into final solution and others are excluded to be part of the solution. Each node is associated with lower bound and the same is represented in Figure 12.2 below.

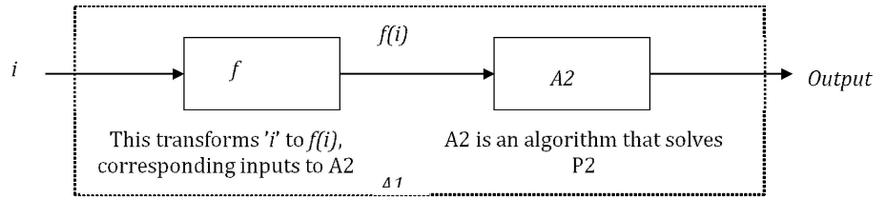


Fig. 12.2 *Branching*

12.4.2 Bounding

Bounding deals how to compute the cost associated with each node. The cost at each node is obtained by performing the following operations on cost matrix.

- Subtract a constant from any row and column. Subtracting this constant does not impact the optimal solution of the desired path.
- The cost of the path changes but not the path itself.
- Let's consider M as the cost matrix of a Graph G=(V,E).
- The cost associated with each individual node is calculated as:
  - Let R is any node in the graph G and A(R) is its associated reduced matrix
  - The cost of the child S associated with node R say C :
    - Set row '*i*' and column '*j*' to infinity.
    - Set A(i,j) to infinity
    - Reduced C and let RCL represents reduced cost.
    - Cost(S)=Cost(R)+RCL+A(i,j)
    - The reduced matrix M_R of M is produced and let L be the valued subtracted from M.
    - L is the lower bound of the conceived path and the cost of the path is reduced by value L.

Let's consider the following example to understand the operational aspect of this reduction approach of problem solution.

Let M is the cost matrix and is equal to

NOTES

$$\begin{bmatrix} inf & 20 & 30 & 10 & 11 \\ 15 & inf & 16 & 4 & 2 \\ 3 & 5 & inf & 2 & 4 \\ 19 & 6 & 18 & inf & 3 \\ 16 & 4 & 7 & 16 & inf \end{bmatrix}$$

The space tree that is the outcome of reduction matrix is represented in the Figure 12.3 below.

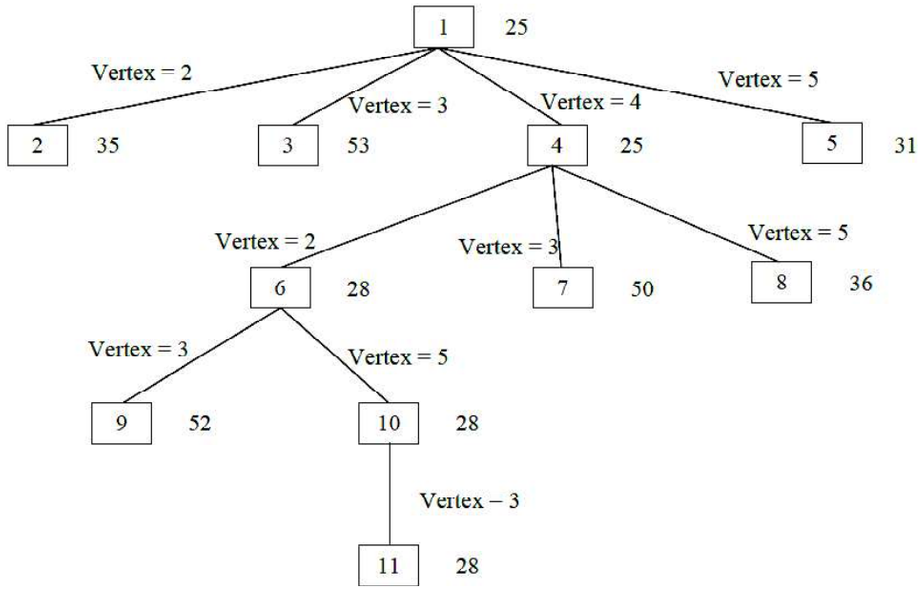


Fig. 12.3 *Space Tree*

Let's assume that travel salesman starts from Node 1: **Node 1**

Reduced matrix:

- Reducing row one by 10 the cost matrix M will become

$$\begin{bmatrix} inf & 10 & 20 & 0 & 1 \\ 15 & inf & 16 & 4 & 2 \\ 3 & 5 & inf & 2 & 4 \\ 19 & 6 & 18 & inf & 3 \\ 16 & 4 & 7 & 16 & inf \end{bmatrix}$$

- Reducing row 2 by 2 the cost matrix M will become

$$\begin{bmatrix} inf & 10 & 20 & 0 & 1 \\ 13 & inf & 14 & 2 & 0 \\ 3 & 5 & inf & 2 & 4 \\ 19 & 6 & 18 & inf & 3 \\ 16 & 4 & 7 & 16 & inf \end{bmatrix}$$

NOTES

- Reducing row 3 by 2the cost matrix M will become

$$\begin{bmatrix} inf & 10 & 20 & 0 & 1 \\ 13 & inf & 14 & 2 & 0 \\ 1 & 3 & inf & 0 & 2 \\ 19 & 6 & 18 & inf & 3 \\ 16 & 4 & 7 & 16 & inf \end{bmatrix}$$

- Reducing row 4 by 3 the cost matrix M will become

$$\begin{bmatrix} inf & 10 & 20 & 0 & 1 \\ 13 & inf & 14 & 2 & 0 \\ 1 & 3 & inf & 0 & 2 \\ 16 & 3 & 15 & inf & 0 \\ 16 & 4 & 7 & 16 & inf \end{bmatrix}$$

- Reducing row 5 by 4the cost matrix M will become

$$\begin{bmatrix} inf & 10 & 20 & 0 & 1 \\ 13 & inf & 14 & 2 & 0 \\ 1 & 3 & inf & 0 & 2 \\ 16 & 3 & 15 & inf & 0 \\ 12 & 0 & 3 & 12 & inf \end{bmatrix}$$

- Reducing column 1 by 1 the cost matrix M will become

$$\begin{bmatrix} inf & 10 & 20 & 0 & 1 \\ 12 & inf & 14 & 2 & 0 \\ 0 & 3 & inf & 0 & 2 \\ 15 & 3 & 15 & inf & 0 \\ 11 & 0 & 3 & 12 & inf \end{bmatrix}$$

- Column 2 is already reduced as it has zero as one element
- Reducing Column 3 by 3

$$\begin{bmatrix} inf & 10 & 17 & 0 & 1 \\ 12 & inf & 11 & 2 & 0 \\ 0 & 3 & inf & 0 & 2 \\ 15 & 3 & 12 & inf & 0 \\ 11 & 0 & 0 & 12 & inf \end{bmatrix}$$

- Column 4 and Column 5 is already reduced as it has zero as one element

The reduced cost represented by RCL= **25**

Therefore, the cost of node 1 is:

Cost(1)=25(10+2+2+3+4+1+3)

Similarly, if travelling salesman starts from vertex 2: Node 2:

- set cost of edge(1,2) is: A(1,2)=10

- Set row 1 and column 2 as infinity (as edge(1,2) is selected).
- Set A(2,1) = infinity. thereafter the cost matrix will be

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ inf & inf & 11 & 2 & 0 \\ 0 & inf & inf & 0 & 2 \\ 15 & inf & 12 & inf & 0 \\ 11 & inf & 0 & 12 & inf \end{bmatrix}$$

NOTES

The matrix is reduced and RCL=0 and the cost of node 2 (from vertex 2 to 1) id

Cost(2)=Cost(1)+A(1,2)= 25+10=35

Similarly, if salesman goes to vertex 3: Node 3 the cost matrix will be:

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ 12 & inf & inf & 2 & 0 \\ inf & 3 & inf & 0 & 2 \\ 15 & 3 & inf & inf & 0 \\ 11 & 0 & inf & 12 & inf \end{bmatrix}$$

and its reduced matrix will be obtained by

- Reduce column 1 by 11

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ 1 & inf & inf & 2 & 0 \\ inf & 3 & inf & 0 & 2 \\ 4 & 3 & inf & inf & 0 \\ 0 & 0 & inf & 12 & inf \end{bmatrix}$$

The RCL=11 and the cost going through node 3 is calculated as:

Cost(3)= cost(1)+RCL+A(1,2)=25+11+17**=53**

- **If Salesman goes to vertex 4: Node 4 the cost matrix will be:**

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ 12 & inf & 11 & inf & 0 \\ 0 & 3 & inf & inf & 2 \\ inf & 3 & 12 & inf & 0 \\ 11 & 0 & 0 & inf & inf \end{bmatrix}$$

The rows and columns are already reduced therefore, RCL=0 and the cost will be calculated as:

Cost(4)=Cost(1)+RCL+A(1,4)=25+0+0=**25.**

- In case if Salesman goes to vertex 5: Node 5 the cost matrix will be:

The cost matrix needs to be reduced at row 2 by 2 and row 4 by 3, column are already reduced. The cost matrix will be:

NOTES

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ 12 & inf & 11 & 2 & inf \\ 0 & 3 & inf & 0 & inf \\ 15 & 3 & 12 & inf & inf \\ inf & 0 & 0 & 12 & inf \end{bmatrix}$$

- Reduce row 2 by 2:

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ 10 & inf & 9 & 0 & inf \\ 0 & 3 & inf & 0 & inf \\ 15 & 3 & 12 & inf & inf \\ inf & 0 & 0 & 12 & inf \end{bmatrix}$$

- Reduce row 4 by 3:

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ 10 & inf & 9 & 0 & inf \\ 0 & 3 & inf & 0 & inf \\ 12 & 0 & 9 & inf & inf \\ inf & 0 & 0 & 12 & inf \end{bmatrix}$$

Therefore RCL=5 and the cost will be calculated as:

Cost(5)=Cost(1)+RCL+A(1,5)=25+5+1= **31**

Let's summarize the cost spent by salesman from node 1 to all its adjacent nodes:2,3,4 and 4:

- From node 1to 2 cost(1,2) is 35.
- From node 1 to 3 cost(1,3) is 53.
- From node 1 to 4 cost(1,4) is **25** and
- From node 1 to 5 cost(1,5) is 31

The procedure that was followed from node or vertex 1 to its adjacent nodes can be repaired to obtain the cost associated with each path form each possible node that is from, 2->x, 3->x, 4—> x, etc., that means up to whole graph is explored.

After exploring the whole graph the minimum or optimal path for salesman is observed to be the following:

Minimum cost path=(node 1—> 4 —>6 —>8—>11—>1)

You can read more on Travelling Salesmen Problem in the next unit.

Check Your Progress

1. What is reduction?
2. What is optimization problem?

NOTES

12.5 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. It is a problem solving approach in which a main problem is transformed into a simple problem by implementing various transforming approaches.
2. Optimization problem is a mathematical problem or any computational domain where the main purpose of the problem solution is to reveal the best possible solution of the problem among the all possible outcomes of problem.

12.6 SUMMARY

- Reduction is a problem solving approach in which a main problem is transformed into a simple problem by implementing various transforming approaches like transform-and-conquer, divide-and-conquer or decrease-and-conquer.
- Optimization problem is a mathematical problem or any computational domain where the main purpose of the problem solution is to reveal the best possible solution of the problem among the all possible outcomes of problem.
- The main purpose of reducing the main problem instance into smaller sub-problem is to increase the simplicity, reduce complexity so that the deduction of the solution becomes easy.
- The algorithm that is used to perform multiplication of two matrices can also be used to perform the squaring of a matrix by reducing the matrix for which squaring is to be done into two matrices and then implement the algorithm as was used for matrix multiplication.
- The reduction approach is not only limited to liner or two or three dimensional problems only but can be used to solve graphical problems also.
- To solve the problem of arranging eight queens problem or to find minimum cost path in travelling salesperson problem or Knapsack problems can be easily resolved using reduction approach on their graphical problem presentation.

NOTES

- Travelling salesperson problem describes a graphical solution for the sales person to perform effective sales by travelling all the connected cites without repeating any city and finally returning to its original city from where he started his sales travel.
- Reducing the travelling salesman's problem using branch and bound algorithm the cost matrix describing the cost factor that is associated with various paths that can be opted during his travel is also reduced.

12.7 KEY WORDS

- **Cost Matrix:** It describe the cost factor that is associated with various paths that can be opted during traversal.
- **Optimization problem:** It is a mathematical problem or any computational domain where the main purpose of the problem solution is to reveal the best possible solution of the problem among the all possible outcomes of problem.

12.8 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. Write a note on reduction problem solving approach.
2. What is cost matrix?
3. How will you find the cost associated with a path?

Long Answer Questions

1. "The main purpose of reducing the main problem instance into smaller sub-problem is to increase the simplicity, reduce complexity so that the deduction of the solution becomes easy." Explain in detail.
2. "Travelling salesperson problem describes a graphical solution for the sales person to perform effective sales by travelling all the connected cites without repeating any city and finally returning to its original city from where he started his sales travel." Elaborate.
3. "Bounding deals how to compute the cost associated with each node. The cost at each node is obtained by performing the following operations on cost matrix." Discuss.

12.9 FURTHER READINGS

Levitin, Anany. *Introduction to Design and Analysis of Algorithms*. Delhi: Pearson Education.

Ellis Horowitz, S. Sahani and Rajasekaran, *Fundamentals of Computer Algorithms.* Delhi: Galgotia Publications.

Goodrich, M T and R. Tomassia. *Algorithm Design: Foundations, Analysis and Internet Examples*. Delhi: John wiley and Sons.

NOTES

BLOCK - V

BACKTRACKING AND GRAPH TRAVERSALS

NOTES

UNIT 13 GENERAL METHOD

Structure

13.0 INTRODUCTION

Backtracking is a generalized term for starting at the end of a goal, and incrementally moving backwards, gradually building a solution. Depth first is an algorithm for traversing or searching a tree. Graph traversals on the other hand are also known as graph search and refers to the process of visiting each vertex in a graph. Such traversals are classified by the order in which the vertices are visited. This unit will explain about these in detail.

13.1 OBJECTIVES

After going through this unit, you will be able to:

- Understand assignment problem and its significance
- Discuss 8-Queen's problem
- Explain what graph coloring is
- Analyze the Traveling Salesman Problem

13.2 8-QUEEN'S PROBLEM

NOTES

The name 'backtrack' was first given by D.H. Lehmer in 1950s. Backtracking is a very useful technique used for solving the problems that require finding a set of solutions or an optimal solution satisfying some constraints. In this technique, if several choices are there for a given problem, then any choice is selected and we proceed towards finding the solution. However, if at any stage, it is found that this choice does not provide the required solution then from that point, we backtrack to previous step and select another choice. The process is continued until the desired solution to the given problem is obtained.

In most of the applications of backtrack method, an optimal solution is expressed as an n-tuple $(a_1, a_2, a_3, ..., a_n)$, where a_i belongs to some finite set S_i. Further, the solution is based on finding one or more vectors that maximizes or minimizes or satisfies a criterion function $P(a_1, a_2, a_3, ..., a_n)$. Criterion function is also called as **bounding function**. Note that sometimes, it is required to find all vectors that satisfy P.

Consider an example that depicts the idea behind this technique. Let f[1:n] be an unsorted array containing n elements and we have to sort this array using the backtracking technique. The sorted sequence can be represented as an n-tuple $(a_1, a_2, a_3, ..., a_n)$, where a_i is the index of the i^{th} smallest element in array f. The criterion function is given by f[a_i] d" f[a_{i+1}] for 1d"i<n. The set S_i is a finite set of integers in the range [1,n].

If k_i is the size of the set S_i, then there are k (where, $k=k_1k_2k_3...k_n$) n-tuples that may be the possible solutions satisfying the criterion function P. The brute force approach can be used to define all the k n-tuples, evaluate each one of them to determine the optimal solution. But this method is tedious and time-consuming. On the other hand, backtracking requires less number of trials to determine the solution. The idea behind backtracking technique is to generate the solution vector by adding one component at a time and to use modified criterion functions $P_i (a_1, a_2, a_3, ..., a_i)$. These functions are used to check whether solution vector $(a_1, a_2, a_3, ..., a_i)$ leads to an optimal solution or not. If at any stage, it is found that the partial vector $(a_1, a_2, a_3, ..., a_i)$ cannot result in optimal solution then $k_{i+1}...k_n$ possible test vectors need not to considered and hence, can be ignored completely from the set of feasible solutions.

Note that most of the problems that can be solved by using backtracking technique require all the solutions to satisfy a complex set of constraints. These constraints are divided into two categories: *explicit* and *implicit*.

- **Explicit Constraints:** These are the rules that allow each a_i to take values from the given set only. They depend upon the particular instance I of the problem being solved. Some examples of explicit constraints are given below:

a_i=0 or 1 or S_i={0,1}

a_ie”0 or S_i={all nonnegative real numbers}

l_id”a_id”u_i or S_i={b: l_id”bd”u_i}

NOTES

Notice that the solution space for I is defined by all the tuples that satisfy the explicit constraints.

- **Implicit Constraints:** These are the rules that identify all the tuples in the solution space of I which satisfy the bounding function.

Some Important Terminologies

Backtracking algorithm finds the problem solutions by searching the solution space (represented as a set of n tuples) for the given problem instance in a systematic manner. To help in searching, a tree organization is used for the solution space, which is referred to as **state space tree**. The set of all the paths from the root node of the tree to other nodes is referred to as the **state space** of the problem. Each node in the state space tree defines a **problem state**. A problem state for which the path from the root node to the node defining that problem state defines a tuple in the solution space (that is, the tuple satisfies the explicit constraints), is referred to as the **solution state**. A solution state (say, s) for which the path from the root node to s defines a tuple that satisfies the implicit constraints is referred to as an **answer state**.

After defining these terms, we can understand how a problem can be solved using backtracking. Solving any problem using backtracking involves the following four steps.

1. Construct the state space tree for the given problem.
2. Generate the problem states from the state space tree in a systematic manner.
3. Determine which problem states are the solution states.
4. Determine which solution states are the answer states.

The most important of these steps is the generation of problem states from the state space tree. This can be performed using two methods. Both methods are similar in the sense as they both start from the root node and proceed to generate other nodes. While generating the nodes, nodes are referred to as live nodes, E-nodes and dead nodes. A node which has been generated but its children have not yet been generated is known as **live node**. A live node is referred to as an **E-node** (expanded node) if its children are currently being generated. After all the children of an E-node have been generated or if a generated node is not to be further expanded, it is referred to as a **dead node**. Notice that in both methods, we have a list of live nodes. Moreover, a live node can be killed at any stage without further generating all its children with the help of a bounding function. The bounding functions for any problem are chosen such that whatever method is adopted, it always generates at least one answer node.

NOTES

Both methods differ in the path they follow while generating the problem states. In one method, the state space tree is traversed in depth-first manner for generating the problem states. When a new child (say, C) of the current E-node (say, R) is generated, C becomes the new E-node. The node R becomes the E-node again when the subtree C has been fully explored. In contrast, in the second method, an E-node remains an E-node until all its children have been generated, that is, it becomes a dead node. The former state generation method is referred to as backtracking, while the latter method of state generation is referred to as branch and bound method.

Note: Many tree organizations are possible for a single solution space.

8-Queen's Problem

Eight queen's (8-queen's) problem is the challenge to place eight queens on the chessboard so that no two queens attack each other. By attacking, we mean that no two queens can be on the same row, column, or diagonal. For this problem, all the possible set of solutions can be represented as 8-tuples ($x_1, x_2, ..., x_8$) where x_i is the column on which queen i is placed. Further, the explicit constraint is S_i={1, 2, 3, 4, 5, 6, 7, 8} for 1d"id"8, as according to the first constraint, only one queen can be placed in one row. Thus, the solution space consists of 8^8 8-tuples. However, the other two constraints that queens must be on different columns and no two queens can be on the same diagonal, are the implicit constraints. If we consider only first one of the implicit constraints, then the solution space size is reduced from 8^8 tuples to 8! tuples.

Before discussing the solution to 8-queen's problem, let us consider the **n-queen's problem**. It is the generalization of the 8-queen's problem. Here, the problem is to place n queens on an n*n chessboard such that no two queens attack each other. We can observe that such an arrangement is possible for only $n \geq 4$ because for n=1, the problem has a trivial solution and for n=2, and n=3, no solution exists.

Assuming n=4, we have to place four queens on a 4*4 chessboard such that no two queens attack each other. That is, queens must be on different rows, columns and diagonal. The explicit constraint for this problem is s_i={1,2,3,4}, where 1d"id"4 as no two queens can be placed in the same row. In addition, two implicit constraints are that no two queens can be placed in the same column and also not on the same diagonal. Here, if only rows and columns are to be different, then solution space consists of 4! ways in which queens can be placed on the chessboard but if third constraint, that is, no two queens can be on the same diagonal is considered, then the size of the solution space is reduced very much. Consider Figure 13.1 for understanding how backtracking works for 4-queen's problem.

NOTES

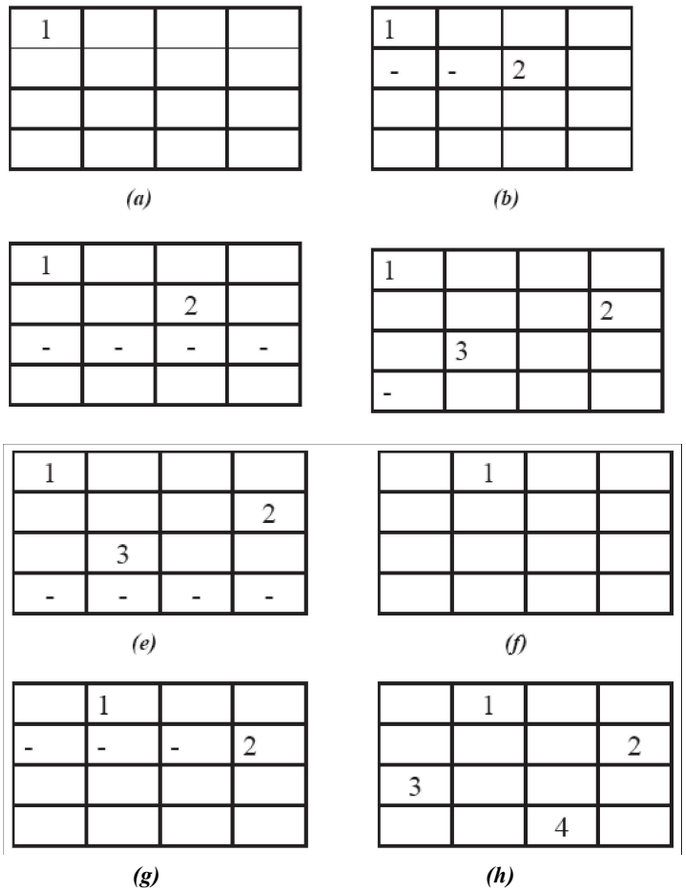


Fig. 13.1 *Solution to 4-Queen's Problem*

Initially, the chessboard is empty so first queen can be placed anywhere on the chessboard. However, we consider that queens are placed row wise. Let first queen is placed in the first row, first column (1, 1) as shown in Figure 13.1(a). Now, second queen can be placed either in the second row, third column (2, 3) or second row, fourth column (2, 4) because if queen is placed at position (2, 1) or (2, 2), then the queens will attack each other, violating the constraints. Let the second queen is placed at position (2, 3) as shown in Figure 13.1(b). Observe that this position does not lead to an optimal solution because if queen is placed at this position then third queen cannot be placed anywhere. So, we backtrack by one step and place the second queen at position (2, 4), the next possible solution as shown in Figure 13.1(d). Then, the third queen can be placed at (3, 2) as shown in Figure 13.1(e). But again this position leads to a dead end as there is no space left for the fourth queen to be placed. So, from here we backtrack and change the position of the first queen to (1, 2) as shown in Figure 13.1(f). Doing so, now the second queen is placed at position (2, 4) as shown in Figure 13.1(g), followed by the third queen placed at position (3, 1) and the fourth queen at

position (4, 3) as shown in Figure 13.1(g). Finally, all queens have been placed on the 4*4 chessboard and we get the optimal solution as 4-tuple vector {2, 4, 1, 3}. For other possible solutions, the whole process is repeated again with choosing alternative options.

NOTES

Figure 13.2 shows the state space tree for 4-queen's problem using backtracking technique. This technique generates the necessary nodes and stops if the next node does not satisfy the constraint, that is, if two queens are attacking. It can be seen that all the solutions in the solution space for 4-queen's problem can be represented as 4-tuples (x_1, x_2, x_3, x_4) where x_i represents the column on which queen `i` is placed.

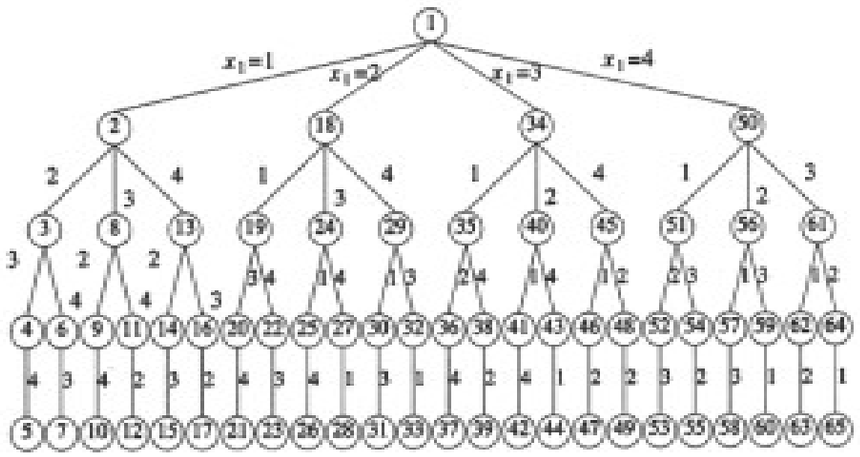


Fig. 13.2 *4-Queen's Solution Space with Nodes Numbered in DFS*

Similarly, for `n`-queen's problem, the solution space consists of as many as `n`! permutations of the `n`-tuple (x_1, x_2, ... , x_n) where x_i represents the column on which queen `i` is placed. Let us consider `n*n` chessboard to be represented as two dimensional array `a[1:n,1:n]`, where every element which is on the same diagonal from upper left to lower right has the same row-column value. Let us take two queens to be placed at positions `(u,v)` and `(w,z)`, then the two queens are on the same diagonal if,

```
   u - v = w-z or u + v = w + z
⇒ v - z = u-w or v - z = w - u
```

Thus, we can say that two queens will be on the same diagonal if and only if `|v-z|=|u-w|`. Before giving the algorithm for placing `n` queens on an `n*n` chessboard, an algorithm `QueenPlace(w,u)` is discussed which determines whether the w^{th} queen can be placed in column `u`. The algorithm returns a Boolean value accordingly.

NOTES

Algorithm 13.1: QueenPlace Function

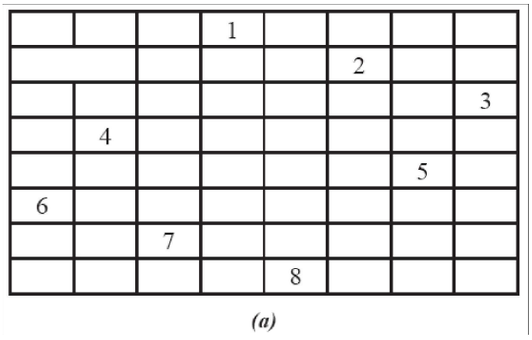
```
QueenPlace(w,u)
//function returns true if queen can be placed at wth row
//and ith column
1. Set i=1
2. while(i = w-1)
3. {
4. If((a[i]=u) OR (Abs(a[i]-u)=Abs(i-w))  //checks whether
      //two queens are in the same column or same diagonal
5.    return false
6.    Set i=i+1
7. }
8.return true
```

Algorithm 13.2 describes the solutions to n-queen's problem using backtracking.

Algorithm 13.2: n-Queen's Problem Using Backtracking

```
NQueensback(w,n)
//prints all possible placements of n queens on an n*n
//chessboard so that they are non-attacking
 1. Set j=1
 2. while(j = n)
 3. {
 4.   If QueenPlace(w,j)
 5.   {
 6.       Set a[w]=j
 7.       If (w=n) // obtained feasible sequence of length
n
 8.          print(a[1:n]) //print the sequence
 9.       Else
10.          NQueensback(w+1,n) //backtrack as sequence is
                                //less than length
11.   }
12.   Set j=j+1
13. }
```

With the help of above algorithm, now we are able to find the solutions to 8-queen's problem using backtracking. Note that there are total 92 solutions to the 8-queen's problem. If rotations and reflections of the board are considered, then there are 12 unique solutions. Some possible solutions for 8-queen's problem are shown in Figure 13.3.



(a)

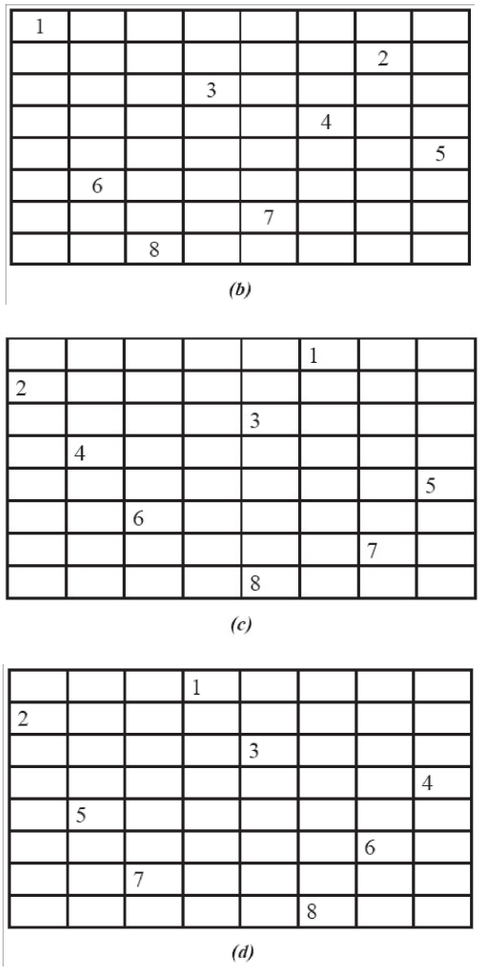


Fig. 13.3 *Different Possible Solutions for 8-Queen's Problem*

13.3 SUM OF SUBSETS

Given a set of n discrete positive numbers (usually referred to as **weights**) represented as w_i, where $1 \leq i \leq n$ and a number s, we are to find all subsets of w_i which add up to s. This is referred to as **sum of subsets** problem. For example, if $(w_1, w_2, w_3, w_4, w_5, w_6) = (6, 18, 23, 7, 2, 1)$ and $s = 25$, then the subsets whose sums is equal to 25 are: (6, 18, 1), (18, 7) and (23, 2). Instead of representing the solution vector by w_i, we can use the indices of w_i which sum to s in the solution vector. Thus, in our case the three solution vectors will be represented as (1, 2, 6), (2, 4) and (1, 5).

The sum of subsets problem can be formulated in many ways so that the solutions in each formulation are the tuples which satisfy certain explicit and implicit constraints. Two possible formulations of the solution space for this problem are by using variable- and fixed-sized tuples. In the **variable-sized tuple strategy**,

NOTES

all the solutions are represented as m-tuples ($x_1, x_2, \ldots, x_m$) where $1 \le m \le n$ and the size of tuples in different solutions may vary (as in our example). The explicit constraints in this strategy require that $x_i \in \{p \mid p \text{ is an integer and } 1 \le p \le n\}$. The implicit constraints require that:

- No two subsets should be same and that the sum of corresponding weights be equal to s.
- The solution space should not contain multiple instances of the same subset. That is, $x_i < x_{i+1}$ where $1 \le i \le m$.

On the other hand in **fixed-sized tuple strategy**, each solution is represented as a fixed size n-tuple ($x_1, x_2, \ldots, x_n$) such that x_i ($1 \le i \le n$) is either 1 or 0 depending on whether the w_i forms the part of solution vector or not, respectively. Following this strategy three possible solution vectors for our example will be represented as (1, 1, 0, 0, 0, 1), (0, 1, 0, 1, 0, 0) and (1, 0, 0, 0, 1, 0). Whatever strategy is followed, there are 2^n discrete tuples in the solution space.

Now, we present the backtracking solution for sum of subsets problem using fixed-sized tuple strategy. Figure 13.4 shows a possible tree organization for the fixed-sized tuple formulation for $n = 4$. In this figure, nodes are numbered as in depth first search (D-search) and edges are labeled such that an edge from level i to an edge at level i+1 represents the value of x_i which is either 0 or 1. Each path from the root node to a leaf node defines a solution vector of solution space. With $n = 4$, there are 2^4 (=16) leaf nodes in the tree which represent 16 possible tuples of the solution space.

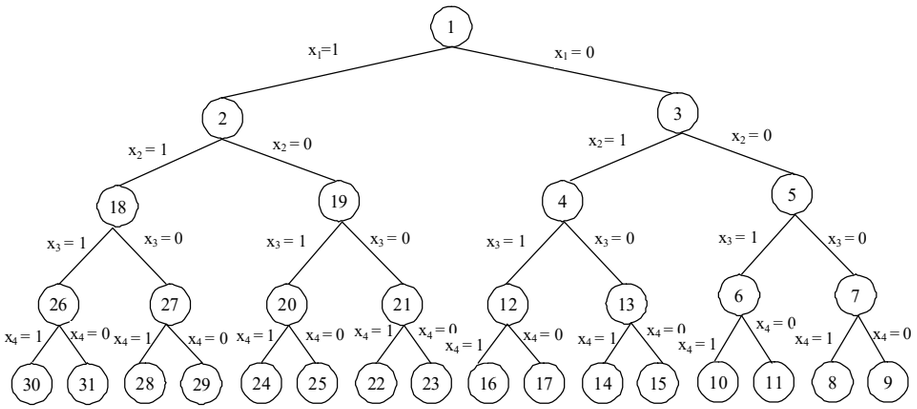


Fig. 13.4 *A Possible Solution Space Organization*

The left subtree of each node at a specific level (say, i) defines all the subsets which include the weight w_i, while the right subtree defines all subsets which do not include w_i. In other words, for each node at a specific level (say, i), the left child represents $x_i = 1$, while the right child represents $x_i = 0$. The bounding function $B_j(x_1, \ldots, x_j)$ is true if and only if:

$$\sum_{i=1}^{j} w_i x_i + \sum_{i=j+1} w_i \ge s$$

Observe that $(x_1, \ldots, x_j)$ leads to an answer node only if the condition of bounding function is satisfied. If w_i's are initially in increasing order, then $(x_1, \ldots, x_j)$ can lead to an answer node if:

$$\sum_{i=1}^{j} w_i x_i + w_{j+1} \leq s$$

Thus, the modified bounding function $B_j(x_1, \ldots, x_j)$ is true if and only if:

$$\sum_{i=1}^{j} w_i x_i + \sum_{i=j+1} w_i \geq s$$

and

$$\sum_{i=1}^{j} w_i x_i + w_{j+1} \leq s$$

The modified bounding function can be simplified if $x_j = 1$. Then:

$$\sum_{i=1}^{j} w_i x_i + \sum_{i=j+1}^{n} w_i > s$$

Algorithm 13.3 describes the solution to sum of subsets problem using recursive backtracking.

Algorithm 13.3: Sum of Subsets Problem using Backtracking

```
Sum_of_Sub_Back(m,j,p)
//prints all subsets of w[1..n] that add up to s.
//Variable m holds the value Σ_{k=1}^{j-1} w[k] * x[k] and p holds the
value
// Σ_{k=j}^{n} w[k]. The w[k]'s are in increasing order. w[1] = s and
Σ_{i=1}^{n} w[i] ≥ s

1.  //Generate left child
2.  Set x[j]=1      //include x[j] in the subset
3.  If (m+w[j]=s)
4.    print(x[1:j]) //print the subset
5.  Else
6.  {
7.      If (m+w[j]+w[j+1] = s)
8.           Sum_of_Sub_Back(m+w[j],j+1,p-w[j])  //recursive
    call                                        //to
    Sum_of_Sub_Back()
9.  }
10. //Generate right child
11. If((m+p-w[j] = s) AND (m+w[j+1] = s)
12. {
13.     Set x[j]=0  //exclude x[j] from the subset
14.     Sum_of_Sub_Back(m,j+1,p-w[j])
15. }
```

NOTES

Example 13.1: Let w={3, 4, 5, 6} and s=13.Trace Algorithm 13.3 to find all possible subsets of w that sum to s.

Solution: Given that n = 4, w = {3, 4, 5, 6} and s = 13. To find the desired subsets, we start with j = 1.Thus, m = $\Sigma_{k=1}^{0}$ w[k] * x[k] = 0 and p = $\Sigma_{k=1}^{4}$w[k] w[1]+w[2]+w[3]+w[4] = 3+4+5+6 = 18. Now follow these steps to find the subsets.

1. Set x[1] = 1 (means include w[1] in the subset) and check the condition at step 3 of algorithm. As m + w[1] = 0 + 3 = 3 ≠ 13, we move to step 7 of algorithm and check if m + w[1] + w[2] ≤ 13 or not. Since 0 + 3 + 4 = 7 ≤ 13, we again call the algorithm with arguments m = m + w[1] = 0 + 3 = 3, j = 2 and p = p – w[1] = 18 – 3 = 15 and thus, move to step 2 of algorithm.
2. Set x[2] = 1 (means include w[2] in the subset) and check the condition at step 3 of algorithm. As m + w[2] = 3 + 4 = 7 ≠ 13, we move to step 7 of algorithm and check if m + w[2] + w[3] ≤ 13 or not. Since 3 + 4 + 5 = 12 ≤ 13, we again call the algorithm with arguments m = m + w[2] = 3 + 4 = 7, j = 3 and p = p – w[2] = 15 – 4 = 11 and thus, move to step 2 of algorithm.
3. Set x[3] = 1 (means include w[3] in the subset) and check the condition at step 3 of algorithm. As m + w[3] = 7 + 5 = 12 ≠ 13, we move to step 7 of algorithm and check if m + w[3] + w[4] ≤ 13 or not. Since 7 + 5 + 6 = 18 > 13, we need to backtrack to previous step. Now, to generate the right child, we move to step 11 of algorithm and check if [(m + p – w[3] ≥ 13) AND (m + w[4] ≤ 13)] or not. Since [(7 + 11 – 5 = 13 ≥ 13) AND (7 + 6 = 13 ≤ 13)] evaluates to true, set x[3] = 0 (means remove x[3] from the subset). We then again call the algorithm with arguments m = 7, j = 4 and p = p – w[3] = 11 – 5 = 6 and thus, move to step 2 of algorithm.
4. Set x[4] = 1 (means include w[4] in the subset) and check the condition at step 3 of algorithm. As m + w[4] = 7 + 6 = 13 = 13, we print the solution as x[1..4] = {1, 1, 0, 1}. Next, we move to step 11 to generate the right child. Since now the condition at step 11 evaluates to false, the process is terminated.

13.4 GRAPH COLORING

Consider a graph G having number of nodes n and m be a given positive integer. The graph coloring problem is to determine if all the nodes of G can be colored using m colors only in such a way that no two adjacent nodes share the same color. This type of problem is known as **m-colorability decision problem**. There is another type of graph coloring problem known as **m-colorability optimization problem**, in which it is required to determine the minimum number of colors (smallest integer m)

NOTES

using which all the nodes of graph G can be colored provided that no two adjacent nodes have the same color. The integer m is known as **chromatic number** of the graph. For example, the graph shown in Figure 13.5 can be colored using three colors 1, 2, and 3. Hence, the chromatic number for the graph is 3.

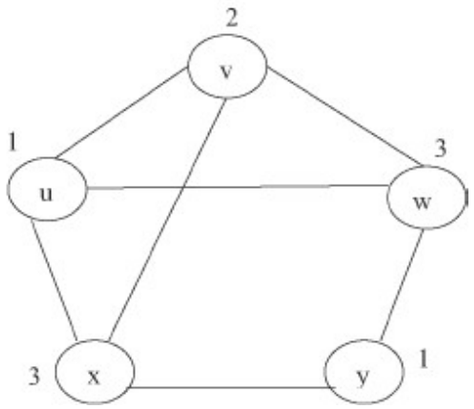


Fig. 13.5 *Graph and its Coloring*

Now, consider the 4-color problem for planar graphs which is a special case of the m-colorability decision problem. A **planar graph** is defined as a graph that can be drawn in a plane in such a way that no two edges of graph cross each other. Figure 13.6 shows a map with five regions. Now, the problem is to determine whether all the regions can be colored in such a way that no two adjacent regions have the same color by using the given four colors only.

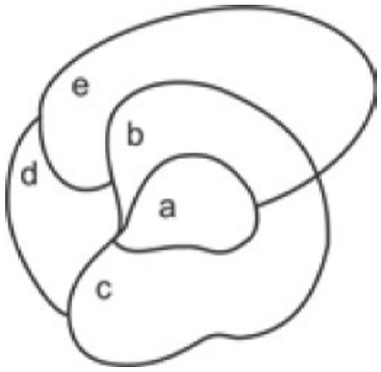


Fig. 13.6 *Map with Five Regions*

The map shown in Figure 13.6 can be transformed into a graph shown in Figure 13.7 where each region acts as a node of a graph and two adjacent regions are represented by an edge joining the corresponding nodes.

NOTES

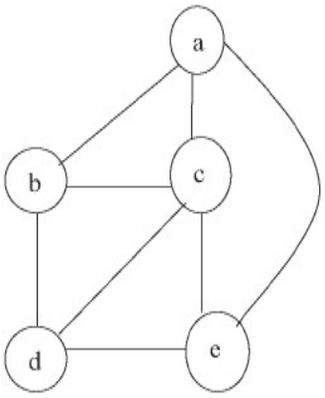


Fig. 13.7 *Planar Graph Representation*

Now, we are to determine all the different ways in which the given graph can be colored using at most `m` colors. Let us consider the graph `G` represented by its adjacency matrix `G[1:n,1:n]` where `n` is defined as the number of nodes in the graph. For each edge `(i,j)` in `G`, `G[i,j]`=1; otherwise `G[i,j]`=0. The colors are represented by the numbers 1, 2, 3, …, `m`. The solutions to the problem are represented in the form of `n`-tuple $(a_1, a_2, \ldots, a_n)$ where a_i is the color of the node `i`. The algorithm for this problem is given here.

Algorithm 13.4: m-Coloring Graph Problem

```
mColoring(h)
//h is the index of the next node to be colored. Array a[h]
//holds the color of each of the n nodes
 1. do
 2. {
 3.   do                 //assigning color to hth node
 4.   {
 5.       Set  a[h]=(a[h]+1)mod(m+1)  //Select  next  highest
color
 6.       If(a[h]=0)     //check if all colors have been used
 7.          return
 8.       Set k=1
 9.       while(k = n)//this  loop  determines  whether  the
color
                  //is distinct from the adjacent colors
10.       {
11.          If((G[h,k]?0 AND (a[h]=a[k])) //check if (h,k)
             //is an edge and if adjacent nodes have the same
          //color
12.                break
13.          Set k=k+1
14.       }
15.       If(k=n+1)
16.          break          //New color found
17.   }while(false)         //Try to find another color
18.   If(a[h]=0)            //No new color for this node
19.       return
20.   If(h=n)            //All nodes are colored with utmost
                         //m colors
21.       Print(a[1:n])  //displays the solution vector
22.   Else
23.       mColoring(h+1) //Call mColoring()for value h+1
24. }while(false)
```

NOTES

Example 13.2: Trace the Algorithm 13.4 for the sample planar graph shown in Figure. 13.8 and obtain its chromatic number.

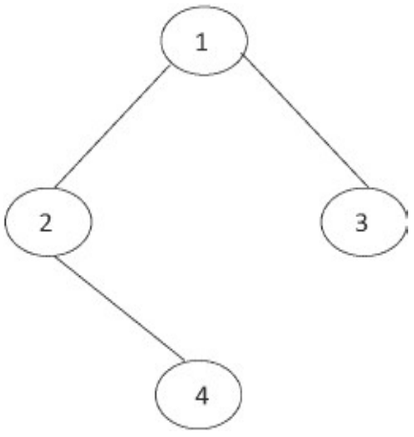


Fig. 13.8 *Sample Planar Graph*

Solution: For the given graph, we have `n`=4. Thus, the chromatic number `m` = 4 - 1 = 3. Initialize `a[]` as {0, 0, 0, 0}.

Step 1: Follow the steps of the algorithm for `h`=1, that is, node 1. From step 5, we get, `a[1]=((a[1]+1) mod(3+1))`=1. This makes the condition in step 6 to evaluate to false. Next, for `k`=1, the condition specified in step 11, that is, `(G[1,1]'"0 AND a[1]=a[1])` to false and hence, `k` is incremented. Observe that the condition also evaluates to false for `k`=2, 3 and 4. When `k`=5, the while loop is exited. Now, as the condition specified in step 15 evaluates to true, we exit from the inner do loop as well. Next, the conditions at step 18 and 20 evaluate to false, thus the statement at step 23 is executed and `mColoring ()` is called for `h`=2. The array `a[h]` becomes {1, 0, 0, 0} up to this step.

Step 2: For `h`=2, we get `a[2]=((a[2]+1) mod(3+1))`=1 from step 5. This makes the condition in step 6 to evaluate to false. Next, for `k`=1, the condition `(G[2,1]'"0 AND a[2]=a[1])` evaluates to true and hence, we exit the while loop. Now, as the condition at step 15 evaluates to false, we loop back to step 5 and get `a[2]`=2. This makes the condition in step 6 to again evaluate to false. Next, for `k`=1, the condition `(G[2,1]'"0 AND a[2]=a[1])` evaluates to false and hence, the value of `k` is incremented. Observe that the condition evaluates to false until `k`=5. Observe that the condition also evaluates to false for `k`=2, 3 and 4. When `k` =5, the while loop is exited. The condition at step 15 evaluates to true and we exit from the inner do loop. Next, as the condition at step 18 and 20 evaluate to false, the statement at step 23 is executed and `mColoring()` is called for `h`=3. The array `a[h]` becomes {1, 2, 0, 0} up to this step.

NOTES

Step 3: The above procedure executes for h=3 and we obtain the modified array a[h]={1, 2, 3, 0}.

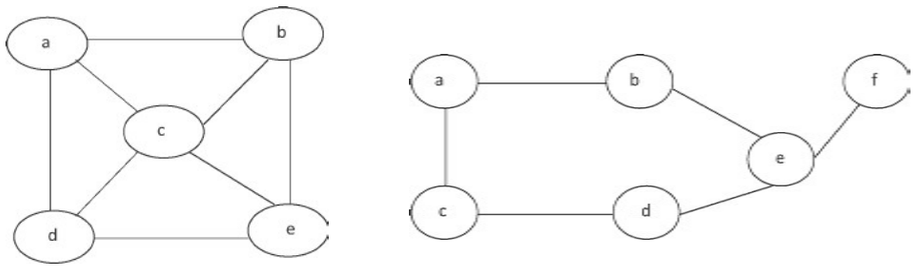
Step 4: For h=4, we get a[4]=((a[4]+1) mod(3+1))=1 from step 5. This makes the condition in step 6 to evaluate to false. Next, for k=1, the condition (G[4,1]'"0 and a[4]=a[1]) evaluates to false, so the value of k is incremented. The condition also evaluates to false for k=2, 3 and 4 and thus, the while loop is exited. For k=5, the condition specified in step 15 evaluates to true and thus, we exit from the inner do loop. Next, the condition at step 18 evaluates to false and the condition at step 20 evaluates to true. Hence, the array a[h]={1, 2, 3, 1} is printed.

Check Your Progress

1. What is Eight queen's problem?
2. What is the graph coloring problem?
3. What are two possible formulations of the solution space?

13.5 HAMILTONIAN CYCLES

Let G=(V,E) be a finite graph (directed or undirected) with n vertices and n edges. A **Hamiltonian cycle** *c* of *G* is a cycle that goes through every vertex exactly once and returns to its initial position. That is, if a cycle starts from some vertex v_1 that belongs to G and the vertices are visited in a sequence as v_1, v_2, v_3, ..., v_{n+1} then the edges (v_i, v_{i+1}) are in E where i =1d"id"n and the v_i are distinct except for v_1 and v_{n+1}, which are the same. A Hamiltonian cycle is also called as **Hamiltonian circuit**. Consider Figure 13.9 representing two graphs. The graph shown in Figure 13.9 (a) has a Hamiltonian cycle a, b, e, d, c, a. On the other hand, the graph shown in Figure 13.9 (b) does not contain Hamiltonian cycle.



(a) Graph with Hamiltonian Cycle *(b) Graph without Hamiltonian Cycle*

Fig. 13.9 *Graphs with and without Hamiltonian cycle*

Note: The graph is said to be Hamiltonian if and only if Hamiltonian cycle exists.

NOTES

To find a Hamiltonian cycle in a graph G using backtracking, we start with any arbitrary vertex, say 1, the first element of the partial solution becomes the root of the implicit tree. Then, the next adjacent vertex is selected and added to the tree. In case, at any stage, it is found that any vertex other than vertex 1 makes a cycle, backtrack one step and proceed by selecting another vertex.

It is clear that the solution vector obtained using backtracking technique is in the form of $(a_1, a_2, ..., a_n)$, where a_i depicts the i^{th} visited vertex of the cycle. To find the solution for Hamiltonian cycle problem, it is required to find the set of the candidate vertices for a_i if $a_1, a_2, ..., a_{i-1}$ have already been chosen. If first vertex is to be chosen (that is, $i=1$), then any one of the n vertices can be assigned to a_i. But we assume that all the cycle begins from vertex 1 (that is, $a_1=1$). Considering this, the algorithm to find all the Hamiltonian cycles in a graph G using recursive backtracking algorithm is given below.

Algorithm 13.5: Finding Hamiltonian Cycles in a Graph

```
HamiltonianCycle(i)
//Graph is represented as an adjacency matrix G[1:n,1:n].
//All cycles begin at vertex 1. a[] is the solution vector.
 1. do             //Generate legal values for a[i]
 2. {
 3.  do
 4.  {
 5.      Set a[i]=(a[i]+1)mod(n+1)  //Next vertex
 6.      If(a[i]=0)
 7.         break
 8.      If(G[a[i-1],a[i]]?0) //Check the existence of edge
 9.      {
10.         Set k=1
11.            while(k = i-1)   //Checking for distinctness
12.            {
13.               If(a[k]=a[i])
14.                break
15.               Else
16.               Set k=k+1
17.            }
18.            If(k=i)  //If true, vertex is distinct
19.            {
20.               If((k = n) AND G[a[n],a[1]]?0))
21.                break
22.            }
23.      }
24.  }while(false)
25.  If(a[i]=0)
26.      return
27.  If(i=n)
28.      Print(a[1:n])     //Returns Hamiltonian cycle path
29.  Else
30.      HamiltonianCycle(i+1)
31. }while(false)
```

In this algorithm, initially the adjacency matrix G[1:n,1:n] and a[2:n] are set to 0 and a[1] is set to to 1. a[1:n-1] is a path of n-1 distinct vertices. The algorithm starts by generating possible value for a[i]. a[i] is assigned to the

NOTES

next highest numbered vertex which is not present in `a[1:i-1]` and is connected by an edge to `a[i-1]`; otherwise, `a[i]`=0. After assigning value to `a[i]`, the function `HamiltonianCycle()` for next vertex (that is, `i=i+1`) is executed. It is executed repeatedly until `i`'"`n`. If `i=n`, `a[i]` is connected to `a[1]`.

Example 13.3: Consider a graph `G = (V,E)` shown in Figure 13.10. Find a Hamiltonian cycle using backtracking method.

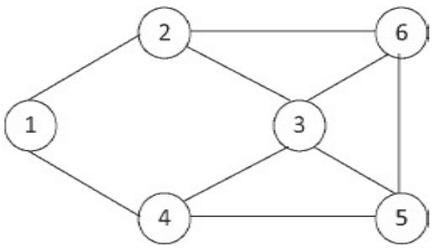
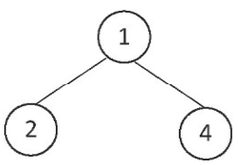


Fig. 13.10 *An Undirected Graph*

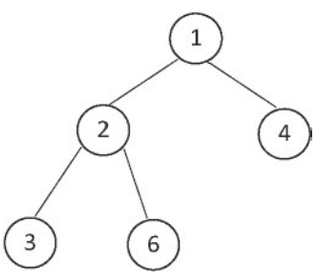
Solution: To find the Hamiltonian cycle using backtracking, we will proceed as follows:

- Start from any vertex, say 1 which becomes the root of implicit tree.

- Select any vertex from the vertices 2 and 4 which are adjacent to vertex 1, say 2. Note that we can select any vertex but generally we choose vertex in numerical order.

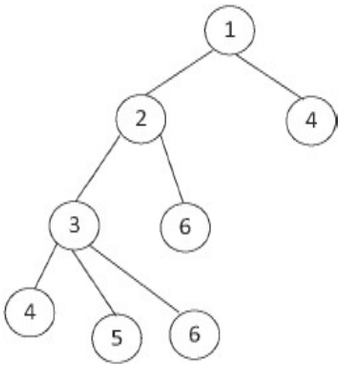


- Select vertex 3 from the vertices 1, 3 and 6 that are adjacent to 2 as the vertex 1 has already been visited and vertex 3 comes first in numerical order from the remaining vertices.

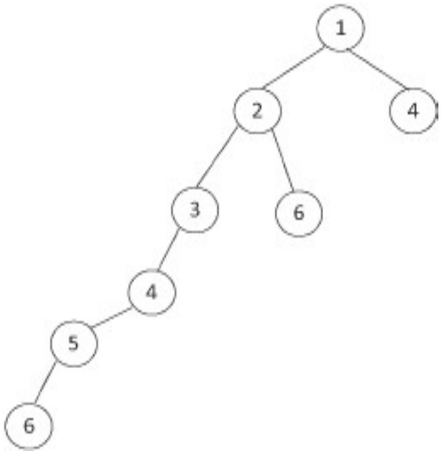


- Select the vertex 4 from the vertices 2, 4, 5 and 6 that are adjacent to 3.

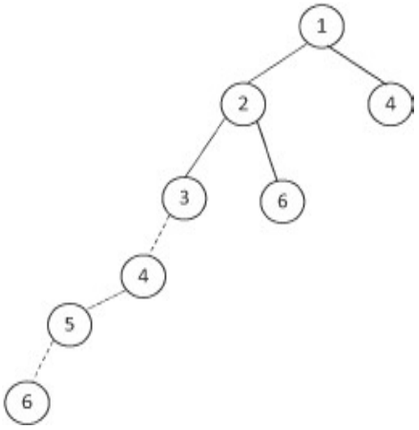
NOTES



- Select vertex 5 from vertices 1, 3 and 5 that are adjacent to 4. Similarly, select vertex 6 from the vertices 3, 4, 6 that are adjacent to vertex 5. Now, the vertices adjacent to vertex 6 are 2, 3 and 5 but they have already visited, that is we have reached at a vertex (dead end) from where we cannot find a complete solution.

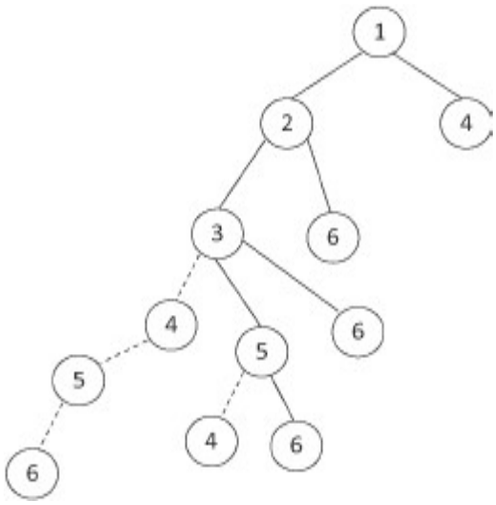


- Backtrack one step and remove vertex 6 from the partial solution. For same reason, backtrack to vertex 3 and remove vertices 5 and 4 from the partial solution.

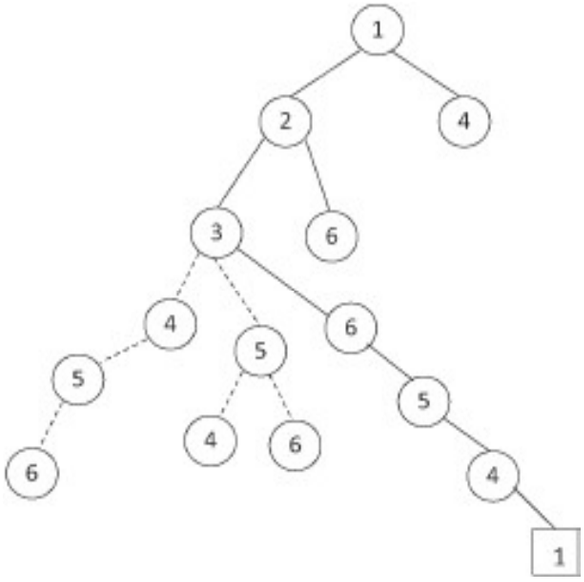


NOTES

- Select the vertex 5 from the vertices adjacent to 3. After proceeding further as done earlier, we have reached at vertex 4, one again, at dead end. So backtrack to vertex 5 and select vertex 6. From this vertex, we cannot proceed further.



- Backtrack to the vertex 3 from where we can proceed further. After proceeding from vertex 3, we have reached to vertex 1, that is, a Hamiltonian circuit is obtained. So, the complete solution is 1-2-3-6-5-4-1.



Backtracking is useful in the problems where there are many possibilities but few of them are required to test for complete solution. It is similar to brute force algorithm but much faster than brute force method, as it removes a large number of possibilities with a single test. The elimination of possibilities in one step is known as **pruning**.

13.6 BRANCH AND BOUND

NOTES

The branch and bound procedure involves two steps: one is branching and the second is bounding. In **branching**, the search space (S) is split into two or more smaller sets say, S_1, S_2, …, S_n (also known as **nodes**) whose union covers S. This step is called so as this process is repeated recursively to each of the smaller sets to form a tree structure. The second step **bounding** computes the lower bound and the upper bound of each node of the tree. The main idea behind branch and bound approach is to reduce the number of nodes that are eligible to become an answer node by safely discarding a node whose lower bound is greater than the upper bound of some other node in the tree formed so far. This step is called **pruning**, and is usually implemented by maintaining a global variable U (shared among all nodes of the tree) that records the minimum upper bound seen among all subsets examined so far. Any node whose lower bound is greater than U can be discarded. This step is repeated till the candidate set S is reduced to a single element or when the upper bound and lower bound become same.

In branch and bound technique, there are two additional requirements that are not required in backtracking. These additional requirements differentiate both these techniques and help in finding an optimal solution for a given problem. These additional requirements are:

- **Lower or Upper Bound:** For each node, there is an upper bound in case of a maximization problem and a lower bound in case of a minimization problem. This bound is obtained by using partial solution.
- **Previous Checking:** The bound for each node is calculated by using partial solution. This calculated bound for the node is checked with the best previous partial result. If the new partial solution leads to worse case, then the bound with the best solution so far, is selected and we do not further explore that part. Otherwise, the checking continues until a complete solution to the problem is obtained using the best result obtained so far.

13.6.1 Branch And Bound Search Methods

As we have already discussed that branch and bound technique deals with finding the optimum solution by using bound values (upper bound and lower bound). These bound values are calculated using D-search, breadth-first search techniques and least cost (LC) search. In branch and bound approach, the **breadth-first search** is called as FIFO search as the list of live nodes are in a queue that follows first-in-first-out list. Similarly, **D-search** is called as LIFO search, as the list of live nodes is a last-in-first-out list. These calculations help us to select the path that we have to follow and explore the nodes. These search methods have been discussed in detail in this section.

NOTES

FIFO Branch and Bound Search

In FIFO branch and bound search, each new node is placed into a queue. Once all the children of the current E-node have been generated, the node at the front of the queue becomes the new E-node. To understand how the nodes can be explored in FIFO branch and bound search, consider the state space tree shown in Figure 13.11.

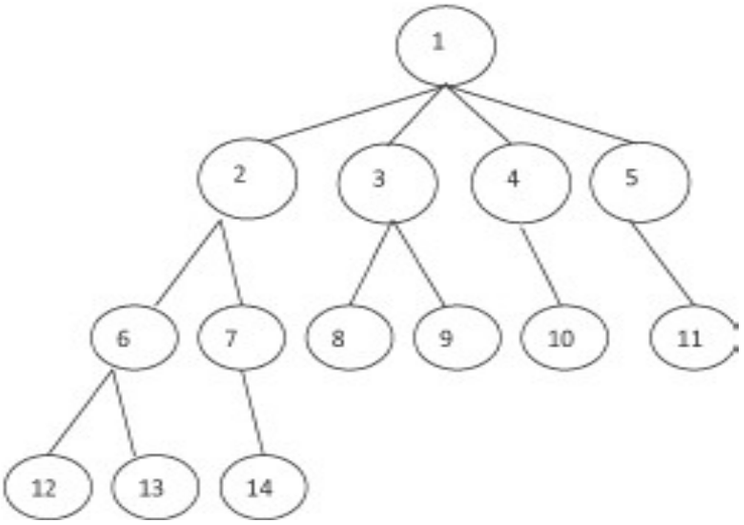


Fig. 13.11 *FIFO Branch and Bound Search*

Assume that node 14 is the answer node and all others are killed.

Initially, the queue (represented as Q) is empty and the root node 1 is the E-node. Expand and generate the children of E-node (node 1), that is, node 2, 3, 4 and 5 and place them in a queue as shown below.

2	3	4	5	...

Here, the live nodes are node 2, 3, 4 and 5. Using the FIFO approach, remove the element placed at the front of queue, that is, node 2. The next E-node now is node 2, thus generate all its children, that is, node 6 and 7 and put them in the rear of queue as shown below.

3	4	5	6	7	...

Now, node 3 becomes E-node. Note that the children of node 3, which are node 8 and 9 are not placed in the queue as they are already killed. Thus, the queue becomes as shown below.

4	5	6	7	...

Now, as node 4 becomes the E-node, remove it from the queue. The child of node 4, which is node 10 is not placed in the queue as it is already killed. Thus, the queue becomes as shown as follows.

5	6	7	...

Next, node 5 becomes the E-node, so remove it from the queue. Its child node, that is, node 11 is not added in the queue as it is already killed. Thus, the queue becomes as shown below.

NOTES

6	7	...

Now, as node 6 becomes the E-node, remove it from the queue. The children of node 6, which are node 12 and 13 are not entered in the queue as they are already killed. Thus, the queue becomes as shown below.

7	...

Now, node 7 becomes the E-node. Remove it from the queue and generate its only child node, that is, node 14. The queue now becomes as shown below.

14	...

As node 14 is the answer node, the search ends here. Thus, the optimal path obtained using FIFO branch and bound search is ($1 \rightarrow 2 \rightarrow 7 \rightarrow 14$).

LIFO Branch and Bound Search

In LIFO branch and bound search, the children of an E-node are placed in a stack instead of a queue. Thus, the elements are pushed and popped from one end. To understand the LIFO branch and bound search, consider the state space tree shown in Figure 13.12. Assume that the answer node is node 12 and all others are killed nodes.

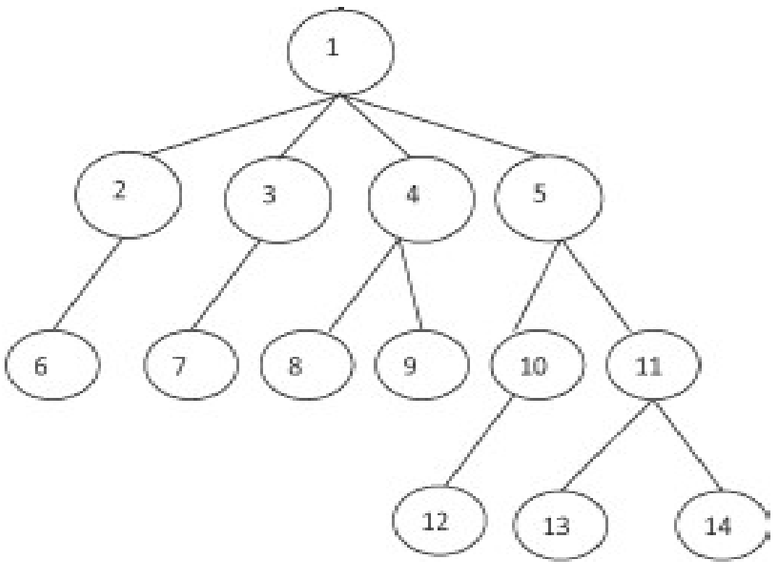
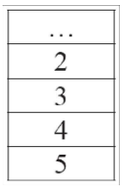


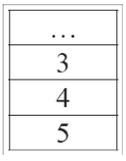
Fig. 13.12 *LIFO Branch and Bound Search*

Further, assume a stack S, which is empty initially, and the root node 1 is the E-node. Push all children of node 1 (which are node 2, 3, 4 and 5) in the stack with node 2 as the top element.

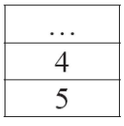
NOTES



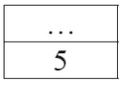
Now, node 2 becomes the E-node. Pop it from the stack, generate its children and push them onto the stack. As the only child of node 2 is node 6, which is already killed, it is not pushed onto the stack. Thus, the stack becomes as shown here.



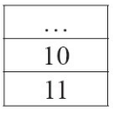
Next, the node 3 becomes E-node, so pop it from the stack. Since its child node, node 7 is killed node; it is not pushed onto the stack. Thus, the stack becomes as shown below.



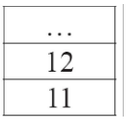
Now, node 4 becomes the E-node, so pop it from the stack. The children of node 4, which are node 8 and 9 are not pushed onto the stack, as they are killed nodes. Thus, the stack becomes as shown below.



Now, node 5 becomes the E-node. Pop it from the stack, generate all its children and push them into the stack as shown below.



Next, pop the current E-node, that is node 10 and push node 12 onto the stack, as shown here.



As node 12 is the answer node, the search ends here. Thus, the optimal path obtained with LIFO branch and bound search is ($1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 10 \rightarrow 12$).

NOTES

Least Cost Search

The important element of branch and bound technique is selection of E-node. Firstly, we select an E-node and explore further nodes connected with it. Thus, this selection should be perfect and efficient so that it leads to answer node quickly. The LIFO and FIFO branch and bound search methods do not give any preference to nodes that will lead to answer quickly instead just push them behind the current live nodes. Further, the selection of E-node using these methods is complicated. For making the search faster for an answer node, we can assign an intelligent ranking function `(x)` to each live node. The next E-node is selected on the basis of this ranking function. The ranking function requires additional computation for calculating the cost needed to reach the answer node from the live node. The next E-node is selected on the basis of the smallest value of `(x)`.

Let `"(x)` be an estimate of additional effort needed to reach an answer node from node `x`. Then a rank can be assigned to node `x` using function `(.)` such that `(x)=f(h(x))+"(x)`. Here, `h(x)` is the cost of reaching `x` from the starting (root) node and `f(.)` is any non-decreasing function. The search strategy that selects the next E-node by computing a cost function `(x)=f(h(x))+"(x)` always chooses a live node with least `(.)` as its next node. Therefore, this search technique is known as Least Cost (LC) search. Note that if `"(x)=0` and `f(h(x))` to be a level of node `x`, then we have BFS and if we take `f(h(x))=0` and `"(x)d" "(y)` where `y` is a child of `x`, then the search is D-search. This implies that BFS and D- search are the special cases of LC search. Note that an LC search with bounding function is called as **Least Cost Branch and Bound (LCBB) search**.

Estimating Cost Function Using LC Search

In LC search, the cost function `C(.)` can be estimated as:

- If `x` is an answer node then the cost function, `C(x)` is computed as the path from `x` to the root in state space tree.
- If `x` is not an answer node and the subtree of node `x` does not contain answer node, then `C(x)=` ". Else, `C(x)` is equal to the minimum cost answer node in the subtree of `x`.

Note: (.) with f(h(x)) can be an approximation of C(.).

Control Abstractions for LC Search

Consider `C()` be a cost function for the nodes in a state space tree `S`. Let `x` is a node in `S`, then `C(x)` is the minimum cost of any answer node in subtree `S` with root `x`.

NOTES

Thus, C(S) is the minimum-cost answer node in tree S. Usually, it is difficult to find an easily computable cost function C(). Therefore, a heuristic to estimate C() is used. Note that this heuristic should be easy to compute and exhibits the property that when x is either an answer node or a leaf node then C(x) is equal to (x).

Algorithm 13.6 describes a procedure that uses heuristic to find an answer node. The function LCSearch outputs the path from root node to the answer node it finds. This algorithm follows the fact that when a node x becomes live, a field parent is associated with it which has the parent of node x. Whenever an answer node V is found, the path between V and x (root node) is determined using a sequence of parent values starting from the current E-node V and ending at node T. This algorithm uses two functions, namely, Least() and Add() to delete and add a live node from or to the list of live nodes, respectively. The former function finds a live node with least () and then this node is deleted from the list of live nodes. The latter function is used to add new live node x to the list of live nodes. This process continues until the live node list is not empty.

In Algorithm 13.6, we have used a record type data structure named node which has three elements as shown here.

```
node
{
       node *next, *parent;
       float cost;
}
```

Algorithm 13.6: Least Cost Search

```
LCsearch(S)
//S is a state space tree
 1. If (*S is an answer node)
 2. {
 3.   Print *S
 4.   return
 5. }
 6. Set E=S     // E-node
 7. Initialize the list of live nodes to be empty
 8. do
 9. {
10.   for each child x of E
11.   {
12.       If (x is an answer node)
13.       {
14.          print path from x to S
15.          return
16.       }
17.       Add(x)    //x is a new live node
18.       Set x->parent=E      //Pointer for path to root
19.   }
20.   If (no live node)
21.   {
22.       print("No answer node")
23.       return
24.   }
25.       Set E=Least()
26. } while (false)
```

13.7 ASSIGNMENT PROBLEM

NOTES

Assignment problem deals with the assignment of different jobs or tasks to workers in a manner so that each worker gets exactly one particular job. This one-to-one strategy of assignment adopted so that all jobs are completed in least time or at least cost. In more technical senses this problems describes the mechanism to assign 'n' different tasks to 'n' different works so that both time and cost that incurs is optimal. In addition to assignment problems also helps a programmer to overcome the situation when the number of jobs and number of works is not same. That means either of the two can more or less. For example How a salesman of company be assigned to take control of sales of other department maximize the total sales values or how to design route map for buses to ferry across cities to reduce layover time. Assignment problem can be solved by implementing following approaches:

(a) enumeration method

(b) Transportation method

(c) Simplex method

(d) Hungarian assignment method

Solution to Assignment Problem

In order to find the optimal assignment of 'n' jobs to 'n' different workers to minimize the cost of work, the first step to instantiate is to construct a n-by-n cost matrix say M. Let P represents workers ($P=P_1,P_2,P_3,\ldots,P_n$). In a constructed cost matrix M, every single value stored at M_{ij} represents the minimum cost for job 'J_i' to assigned to person 'P_i', where, 1<=i<=n and 1<=J <=n.

$$M = \begin{array}{c} \\ P_1 \\ P_2 \\ P_3 \end{array} \begin{array}{c} \begin{array}{ccc} J_1 & J_2 & J_3 \end{array} \\ \begin{bmatrix} M_{1,1} & M_{1,2} & M_{1,3} \\ M_{2,1} & M_{2,2} & M_{2,3} \\ M_{3,1} & M_{3,2} & M_{3,3} \end{bmatrix} \end{array}$$

$$M = \begin{array}{c} \\ P_1 \\ P_2 \\ P_3 \end{array} \begin{array}{c} \begin{array}{ccc} J_1 & J_2 & J_3 \end{array} \\ \begin{bmatrix} 6 & 9 & 5 \\ 4 & 8 & 3 \\ 5 & 11 & 6 \end{bmatrix} \end{array}$$

Let's consider a given cost matrix 'M' where there persons (P_1, P_2 and P_3) are assigned three jobs (J_1, J_2 *and* J_3). From the given cost matrix different assignment cases emerge:

Case 1: If person P_1 is assigned J_1, P_2 is assigned J_2 and P_3 is assigned job J_3.

NOTES

Case 2: If person P_1 is assigned job J_2, P_2 is assigned job J_1 and P_3 is assigned job J_3

Case 3: If Person P_1 is assigned Job J_3, Person P_2 job J_1 and Person P_3 job J_2

Case 4: If Person P_1 is assigned job J_2, Person P_2 job J_3 and person P_3 job J_1.

These cases continue to emerge by obtaining all possible permutations from the given problem.

Let's consider case 1 to find the total cost to perform all the three jobs assigned to persons*(P1 to P3)*

Case 1: $P_1 \rightarrow J_1$, P_2 J_2 and $P_3 \rightarrow J_3 = 6 + 8 + 6 =$ **20**

Case 2: $P^1 \rightarrow J_2$, $P_2 \rightarrow J_1$ and $P_3 \rightarrow J_3 = 9 + 4 + 6 =$ **19**

Case 3: $P_1 \rightarrow J_3$, $P_2 \rightarrow J_1$ and $P_3 \rightarrow J_2 = 5 + 4 + 11 =$ **20**

Case 4: $P_1 \rightarrow J_2$, $P_2 \rightarrow J_3$ and $P_3 \rightarrow J_1 = 9 + 3 + 5 =$ **17**

All the case must be explored to find the optimal assignment of jobs to person by obtaining the optimal value.

If we look around the above matrix M again and check the possible minimum cost in each row that can be put to perform the desired jobs. One can find that as:

$P_1 \rightarrow J_3$, $P_2 \rightarrow J_2$ and $P_3 \rightarrow J_1 = 5 + 3 + 5 =$ **13**

However, it is noticeable that the cost of any job assignment including the optimal cost solution cannot be smaller than the sum or cost obtained above that is 13(that is the sum of minimum cost values in each row). Therefore, in matrix M the sum of values should not be less than 13.In this situation the sum obtained is called as lower bound for the problem.

13.8 0/1 KNAPSACK PROBLEM

As we know, the objective of knapsack problem is to fill the knapsack with the given items in such a way that the total weight put in knapsack should not exceed the capacity of knapsack and maximum profit should be obtained. This problem is a maximization problem as we consider the maximum value of profit and hence we will use upper bound value. As already discussed, knapsack problem is defined as:

$$\text{Maximize} \left| \sum_{i=1}^{n} p_i x_i \right|$$

$$\text{subject to} \sum_{i=1}^{n} w_i x_i \leq m$$

x_i = 0 or 1, 1 d" i d" n

Where, x_i = fraction of item i placed in knapsack

p_i = profit associated with item i

w_i = weight of item i

To use branch and bound technique, construct the implicit tree as binary tree for the given problem. In this implicit tree, the left branch indicates the inclusion of the item and right branch indicates the exclusion. Note that $x_i = 1$, if we include the item; otherwise $x_i = 0$. The upper bound UB of the node can be computed as follows.

$$UB = p + (m-w)(p_{i+1}/w_{i+1})$$

Algorithm 13.7 is used to calculate the upper bound. This algorithm returns the maximum profit that is obtained by accommodating maximum items in the given knapsack.

Algorithm 13.7: Computing Upper Bound for Knapsack Problem

```
UBound(cp,cw,k,m)
//cp is the current profit total, cw is the current weight
//total, k is the index of last removed item and m is the
//knapsack size
 1. Set p=cp
 2. Set w=cw
 3. Set i=k+1
 4. while (i = n)  //n is number of weights and profits
 5. {
 6.       If (c+w[i] = m)
 7.          {
 8.             Set c=c+w[i]
 9.             Set p=p+p[i]
10.          }
11.}
12. return p
```

Example 13.4 Consider an instance of the knapsack problem where n=3, m=4, (w_1, w_2, w_3) = (2, 3, 4) and (p_1, p_2, p_3) = (3, 4, 5). Fill this Knapsack using the branch and bound technique so as to give the maximum possible profit.

Solution: To solve this problem, using branch and bound technique, follow these steps.

Step 1: Calculate profit per weight as shown here.

#Items	w_i	p_i	p_i/w_i
I_1	2	3	1.5
I_2	3	4	1.3
I_3	4	5	1.25

NOTES

Step 2: Compute the upper bound for the root node as given here.

Since, p = 0, m = 4, w = 0, p_1/w_1 = 1.5

Thus, UB = 0 + (4 - 0)*(1.5) = 6

Step 3: Include item I_1 (as indicated by the left branch in Figure 13.13) and compute the upper bound for this node as given here.

Since, p = 3, m = 4, w = 2, p_2/w_2 = 1.3

Thus, UB = 3 + (4 - 2)*(1.3) = 5.6

Step 4: Include item I_2.

Now, p = (0+3) = 3, m = 4, w = (2+3) = 5, p_2/w_2 = 1.3

Here, w>m. Since, w cannot be greater than m, we will backtrack to previous node without exploring this node.

Step 5: Exclude item I_2, that is, include item I_3.

Now, p = (0+3) = 3, m = 4, w = (2+3) = 5, p_2/w_2 = 1.3

Again, w>m, so backtrack to root node.

Step 6: Exclude item I_1 (as indicated by right branch in Figure 6.3) node in which case there is no item in the knapsack. Compute the upper bound for this node as given here.

Since, p = 0, m = 4, w = 0, p_2/w_2 = 1.3

Thus, UB = 0 + (4 - 0)*(1.3) = 5.2

Step 7: Include item I_2 and compute the upper bound for this node as given here.

Since, p = (0+4) = 4, m = 4, w = 0+3 = 3, p_3/w_3 = 1.25

Thus, UB = 4 + (4 - 3)*(1.25) = 5.25

Step 8:

Exclude item I_2, that is, include item I_3 and compute the upper bound for this node as given here.

Since, p = 0+5 = 5, m = 4, w = 0+4 = 4, p_3/w_3 = 1.25

Thus, UB = 5 + (4 - 4)*(1.25) = 5

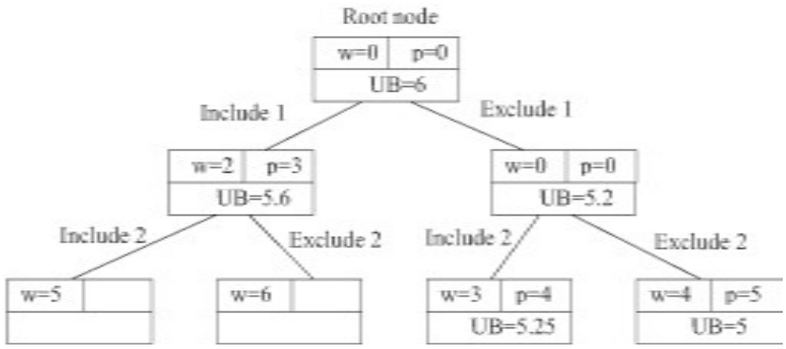


Fig. 13.13 *Implicit Graph for Knapsack Problem*

Step 9:

Finally, select the node with maximum upper bound as an optimum solution. Here, the node with item 1 having weight 2 and profit 3 has the maximum value, that is, 5.6. Thus, it gives the optimum solution to the given problem.

NOTES

Note that if the given knapsack problem is solved using backtracking technique; the solution obtained would be same as both these problem solving techniques provide the optimal solution for the knapsack problem.

13.9 TRAVELING SALESMAN PROBLEM

As you have already learned in the previous unit, in travelling salesperson problem, the salesperson is required to visit n number of cities in such an order that he visits all the cities exactly once and returns to the city from where he has started with incurring minimum cost. Let G=(V,E) be a directed graph representing travelling salesperson problem, where V is a set of vertices representing cities and E is a set of edges. Let number of vertices (cities) be n, that is |V| = n. Further, assume that c(i,j)>0 be the cost of an edge (i,j), representing cost of travelling from city i to j. Set c(i,j) =∞, when there is no edge between vertices i and j. Without loss of generality, we assume that the tour begins and ends at the vertex 1. Let S be the solution space, the tour will be of the form (1, i_1, i_2, . . . , i_{n-1}, 1) ∈ S, if and only if (i_j, i_{j+1}) ∈ E.

We will solve this problem by using LCBB method. To search the state space tree of travelling salesperson, it is required to define a cost function c(.) and the other two functions (.) and u(.) in such a way that (x)d"C(x)d"u(x) for each node x. Further the cost C(.) will be such that the solution node having least C(.) represents the shortest tour in G.

The steps to solve this problem are as follows:

1. Obtain (x) by reducing the cost matrix representing travelling salesperson problem. A matrix is said to be reduced if all its rows and columns are reduced and a row or a column is said to be reduced if and only if it contains at least one zero. The total of all the values, L, subtracted from the matrix to reduce it, is the minimum cost for any tour. This value is used as the (.) for the root of the state space tree.

   For example, consider the cost matrix representing the graph G shown in Figure 13.14.

$$\begin{bmatrix} \infty & 6 & 4 & 11 & 7 \\ 3 & \infty & 4 & 12 & 10 \\ 4 & 9 & \infty & 7 & 18 \\ 8 & 2 & 2 & \infty & 11 \\ 12 & 13 & 10 & 6 & \infty \end{bmatrix}$$

Fig. 13.14 *Cost Matrix*

Reduce this matrix by subtracting 4, 3, 4, 2, 6 and 3 from rows 1, 2, 3, 4, 5 and column 5, respectively. The reduced matrix is shown in Figure 13.15.

NOTES

(a) Reducing rows *(b) Reducing column* *(c) Reduced cost matrix*

Fig. 13.15 *Reducing Cost Matrix*

Now, L = 4+3+4+2+6+3 = 22. Hence, length of all the tours in the graph will have at least cost 22.

2. Associate the least cost to the root of the state space tree and generate all the children nodes for this node.
3. Obtain the reduced cost matrix for each child node. For this consider that A is the reduced cost matrix for node x, and y be the child node of node x. The tree edge (x, y) corresponds to the edge (i,j) included in the tour. Now, the reduced cost matrix for node y, say B can be obtained by following these steps.
   (a) Change all the entries of row i and column j to ∞.
   (b) Set entry in matrix corresponding to edge (j, i), that is A[i,j] to ∞.
   (c) Reduce all the rows and columns of the matrix so obtained except the rows and columns containing ∞.
4. Now (y) can be obtained as follows:

```
(y)= C^(x)+A[i,j] + l
```

   where, l = total value subtracted from matrix A to obtain matrix B. For the upper bound function u, ∞ can be assigned to each node x, that is u(x) = ∞. Further (.)=C(.) for leaf nodes can be determined easily since each leaf represents a unique tour.
5. Select the node with minimum (.) as next E-node and explore it further. This procedure is repeated till we get node with (.) less than (.) of all other live nodes.

Select root node 1 as E-node, as we have assumed the tour will begin from vertex 1. The (.) for the E-node is 22 (=L).

As a next step, nodes 2, 3, 4 and 5 (corresponding to vertices 2, 3, 4 and 5) are generated for E-node 1. Now, we have to calculate (.) for all these nodes corresponding to the edges (1,2), (1,3), (1,4) and (1,5).

Now, generate the reduced cost matrix for node 2, 3, 4 and 5 and compute the (.) as follows:

Node 2, path (1,2), edge: (1,2)

$$-1\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 1 & 9 & 4 \\ 0 & \infty & \infty & 3 & 11 \\ 6 & \infty & 0 & \infty & 6 \\ 6 & \infty & 4 & 0 & \infty \end{bmatrix} = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & 8 & 3 \\ 0 & \infty & \infty & 3 & 11 \\ 6 & \infty & 0 & \infty & 6 \\ 6 & \infty & 4 & 0 & \infty \end{bmatrix}_{-3} = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & 8 & 0 \\ 0 & \infty & \infty & 3 & 8 \\ 6 & \infty & 0 & \infty & 3 \\ 6 & \infty & 4 & 0 & \infty \end{bmatrix}_{-}$$

(2) = 22*((1)) +2*(cost of edge (1,2))+(1+3) = 28

Node 3, path (1,3), edge: (1,3)

$$-3\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & 9 & 4 \\ \infty & 5 & \infty & 3 & 11 \\ 6 & 0 & \infty & \infty & 6 \\ 6 & 7 & \infty & 0 & \infty \end{bmatrix} = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & 9 & 4 \\ \infty & 2 & \infty & 0 & 8 \\ 6 & 0 & \infty & \infty & 6 \\ 6 & 7 & \infty & 0 & \infty \end{bmatrix}_{-4} = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & 9 & 0 \\ \infty & 2 & \infty & 0 & 4 \\ 6 & 0 & \infty & \infty & 2 \\ 6 & 7 & \infty & 0 & \infty \end{bmatrix}_{-}$$

(3) = 22+0*(cost of edge (1, 3))+(3+4) = 29

Node 4, path (1,4), edge: (1,4)

$$-4\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & 1 & \infty & 4 \\ 0 & 5 & \infty & \infty & 11 \\ \infty & 0 & 0 & \infty & 6 \\ 6 & 7 & 4 & \infty & \infty \end{bmatrix}_{-4} = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & 1 & \infty & 0 \\ 0 & 5 & \infty & \infty & 7 \\ \infty & 0 & 0 & \infty & 2 \\ 2 & 3 & 0 & \infty & \infty \end{bmatrix}_{-}$$

(4) =22+7*(cost of edge (1,4))+(4+4) = 37

Node 5, path (1,5), edge: (1,5)

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & 1 & 9 & \infty \\ 0 & 5 & \infty & 3 & \infty \\ 6 & 0 & 0 & \infty & \infty \\ \infty & 7 & 4 & 0 & \infty \end{bmatrix}$$

(5) = 22+0*(cost of edge (1,5))+0 = 22

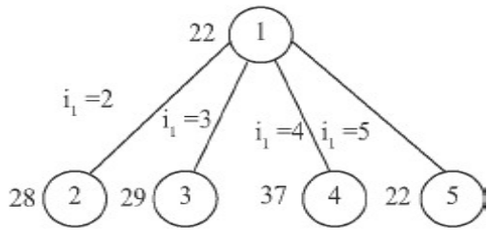


Fig. 13.16 *State Space Tree with Node 1 as E-Node*

NOTES

NOTES

Out of these, (5) is minimum. Therefore, the next vertex selected is 5 and we will select node 5 as our next E-node. Generate nodes 6, 7 and 8 (corresponding to vertices 2, 3, 4) for the E-node 5. Using the same procedure, obtain reduced cost matrix and compute (.) for nodes 6, 7 and 8.

Node 6, path (1,5,2), edges: (1,5), (5,2)

$$-1 \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 1 & 9 & \infty \\ 0 & \infty & \infty & 3 & \infty \\ 6 & \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}_{-3} = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & 5 & \infty \\ 0 & \infty & \infty & 0 & \infty \\ 6 & \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}_{-}$$

(6) =22*((5))+7*(cost of edge (5,2))+(1+3)=33

Node 7, path (1,5,3), edges: (1,5), (5,3)

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & 9 & \infty \\ \infty & 5 & \infty & 3 & \infty \\ 6 & 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}_{-3} = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & 6 & \infty \\ \infty & 5 & \infty & 0 & \infty \\ 6 & 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}_{-}$$

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & 9 & \infty \\ \infty & 5 & \infty & 3 & \infty \\ 6 & 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}_{-3} = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & 6 & \infty \\ \infty & 5 & \infty & 0 & \infty \\ 6 & 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}_{-}$$

(7) = 22+4*(cost of edge (5,3))+3=29

Node 8, path(1,5,4), edges: (1,5), (5,4)

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & 1 & \infty & \infty \\ 0 & 5 & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

(8) =22+0*(cost of edge (5,4))+0=22

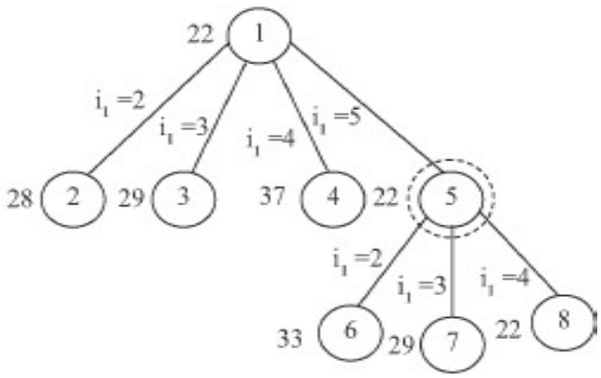


Fig. 13.17 *State Space Tree with Node 5 as E-node*

Out of these (8) is minimum. Therefore, the next vertex selected is 4 and we will select node 8 as our next E-node and generate nodes 9 and 10 (corresponding to vertices 2 and 3) for the E-node 8. Using the same procedure, obtain reduced cost matrix and compute (.) for nodes 9 and 10.

Node 9, path(1,5,4,2) , edges: (1,5), (5,4), (4,2)

$$\begin{bmatrix} \infty & \infty & -1 & \infty & \infty \\ \infty & \infty & 1 & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix} = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

(9) =22* ((8)) +0*(cost of edge (4,2))+1 = 23

Node 10, path(1,5,4,3), edges: (1,5), (5,4), (4,3)

$$-5 \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & 5 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix} = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

NOTES

NOTES

(10)=22+0*(cost of edge(4,3))+5 = 27

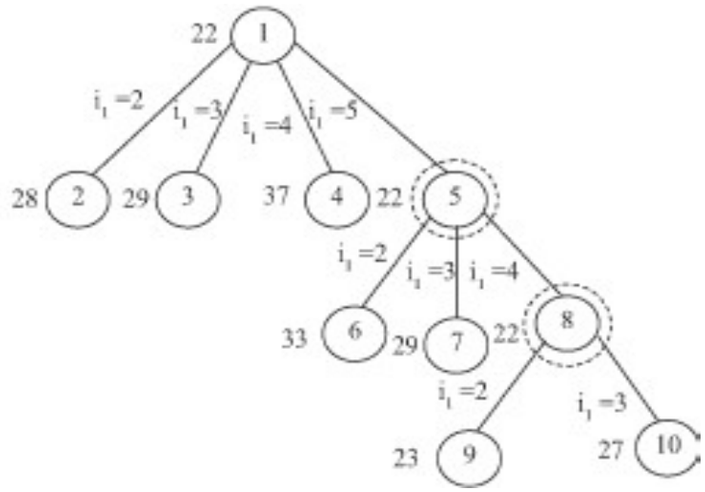


Fig. 13.18 *State Space Tree with Node 8 as E-node*

Out of these (9) is minimum. Therefore, the next vertex selected is 2 and we will select node 9 as our next E-node. Generate solution node 11 (corresponding to vertex 3) for the E-node 10. Using the same procedure, obtain reduced cost matrix and compute (.) for the node 11.

Node 11, path (1,5,4,2,3), edges: (1,5), (5,4), (4,2), (2,3)

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

(11)=23*((9))+1*(cost of edge (2,3))+0=24

Since the (.) for all other E-nodes (nodes which are not explored (2, 3, 4)) is greater than (11), the LCBB terminates with 1,5,4,2,3,1 as the desired path with minimum cost.

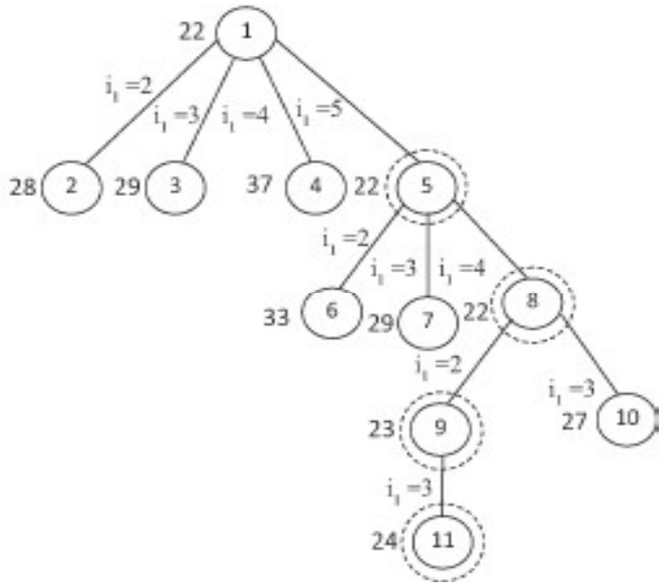


Fig. 13.19 *Final State Space Tree*

Check Your Progress

4. What is a Hamiltonian cycle *c* of *G?*
5. What does the branch and bound procedure involve?
6. What does assignment problem deal with?

13.10 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. Eight queen's (8-queen's) problem is the challenge to place eight queens on the chessboard so that no two queens attack each other.
2. The graph coloring problem is to determine if all the nodes of G can be colored using m colors only in such a way that no two adjacent nodes share the same color.
3. Two possible formulations of the solution space for are by using variable- and fixed-sized tuples.
4. A Hamiltonian cycle c of G is a cycle that goes through every vertex exactly once and returns to its initial position.
5. The branch and bound procedure involves two steps: one is branching and the second is bounding.

6. Assignment problem deals with the assignment of different jobs or tasks to workers in a manner so that each worker gets exactly one particular job.

NOTES

13.11 SUMMARY

- Backtracking is a very useful technique used for solving the problems that require finding a set of solutions or an optimal solution satisfying some constraints.
- In this technique, if several choices are there for a given problem, then any choice is selected and we proceed towards finding the solution
- Note that most of the problems that can be solved by using backtracking technique require all the solutions to satisfy a complex set of constraints. These constraints are divided into two categories: *explicit* and *implicit*.
- Backtracking algorithm finds the problem solutions by searching the solution space
- To help in searching, a tree organization is used for the solution space, which is referred to as state space tree.
- Each node in the state space tree defines a problem state.
- The most important of these steps is the generation of problem states from the state space tree.
- While generating the nodes, nodes are referred to as live nodes, E-nodes and dead nodes.
- Aplanar graph is defined as a graph that can be drawn in a plane in such a way that no two edges of graph cross each other.
- A Hamiltonian cycle *c* of *G* is a cycle that goes through every vertex exactly once and returns to its initial position.
- To find a Hamiltonian cycle in a graph G using backtracking, we start with any arbitrary vertex, say 1, the first element of the partial solution becomes the root of the implicit tree.
- The elimination of possibilities in one step is known as pruning.
- Backtracking is useful in the problems where there are many possibilities but few of them are required to test for complete solution.
- The branch and bound procedure involves two steps: one is branching and the second is bounding.

NOTES

- The second step bounding computes the lower bound and the upper bound of each node of the tree.
- In FIFO branch and bound search, each new node is placed into a queue. Once all the children of the current E-node have been generated, the node at the front of the queue becomes the new E-node.
- In LIFO branch and bound search, the children of an E-node are placed in a stack instead of a queue.

13.12 KEY WORDS

- **Explicit Constraints:** These are the rules that allow each a to take values from the given set only.
- **Implicit Constraints:** These are the rules that identify all the tuples in the solution space of I which satisfy the bounding function.

13.13 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Question

1. What is 8-Queen's problem?
2. What is graph coloring?
3. What do you mean by the Traveling Salesman Problem?
4. Write a short note on knapsack problem.

Long Answer Questions

1. What do you understand by assignment problem? Also list its significance in detail.
2. "**Eight queen's** (8-queen's) problem is the challenge to place eight queens on the chessboard so that no two queens attack each other." Discuss the queen's problem.
3. Let `w` = {3, 4, 5, 6} and `s` = 13. Trace Algorithm 13.3 to find all possible subsets of `w` that sum to `s`.
4. "A Hamiltonian cycle c of G is a cycle that goes through every vertex exactly once and returns to its initial position." Explain the Hamiltonian cycle in detail.

13.14 FURTHER READINGS

NOTES

Levitin, Anany. *Introduction to Design and Analysis of Algorithms*. Delhi: Pearson Education.

Ellis Horowitz, S. Sahani and Rajasekaran, *Fundamentals of Computer Algorithms.* Delhi: Galgotia Publications.

Goodrich, M T and R. Tomassia. *Algorithm Design: Foundations, Analysis and Internet Examples*. Delhi: John wiley and Sons.

UNIT 14 GRAPH TRAVERSALS

NOTES

Structure

14.0 INTRODUCTION

A graph is a non-linear data structure. Graph traversal (also known as graph search) refers to the process of visiting (checking and/or updating) each vertex in a graph. Such traversals are classified by the order in which the vertices are visited. This unit will explain graph traversals in detail.

14.1 OBJECTIVES

After going through this unit, you will be able to:

- Understand the significance of graph, connectedness and spanning trees
- Explain NP hard and NP complete problems
- Explain the use of Cook's theorem
- Discuss and differentiate between NP-Hard and NP-Complete Classes

14.2 GRAPHS

A graph is a non-linear data structure. A data structure in which each node has at most one successor node is called a linear data structure, for example array, linked list, stack, queue etc. A data structure in node has more than one successor node is a called non-linear data structure.

Many problems can be naturally formulated as in terms of elements and their interconnections. A graph is a mathematical representation for such situations. A graph can be defined as follows:

A graph is a set of finite verities or nodes V and a finite set of edges E. Each edge is uniquely identified by a pair of vertices [x, y]. A Graph can be represented by G(V, E).

NOTES

Graph Terminology

In a directed graph, the edges consist of an ordered pair of vertices (one-way edges). A simple path is a sequence of vertices in which no vertex is repeated and a cycle is a simple path except the first and last vertex is repeated. A complete graph is one in which every vertex of the graph is connected with every other vertex (A complete graph with N vertices will have N*[N-1] edges). A sparse graph is one with relatively fewer edges. A graph with multiple edges is called a dense graph. A weighted graph is one in which edges are associated with some weight; this weight can be distance, time or any cost function. The two vertices are called adjacent vertices if there is an edge connecting those two vertices. The degree of a vertex is determined by the number of edges incident on that vertex in an undirected graph. Every edge contributes in the degree of the exactly two vertices in an undirected graph. A loop contributes twice in the degree of a single vertex. For a directed graph, the in-degree is the number of incoming incident edges and the out-degree is the number of outgoing incident edges. The degree of a vertex is the maximum degree of a graph is maximum degree of its vertices. A graph with no cycle called a tree. A sub-graph G'(V', E') of graph G(V, E) is also a graph such that G'(V') is a subset of G(V) and G'(E') is a subset of G(E). Maximal connected sub-graph of an undirected graph is the connected component. The example of graphs is shown in Figure 14.1 and 14.2.

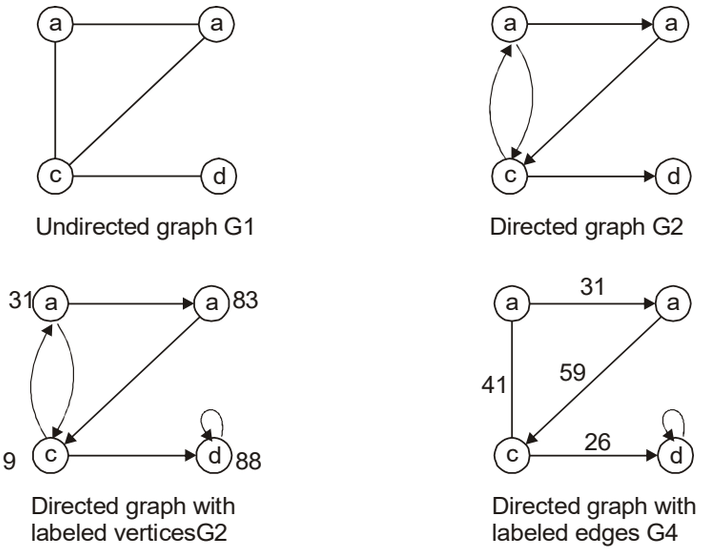
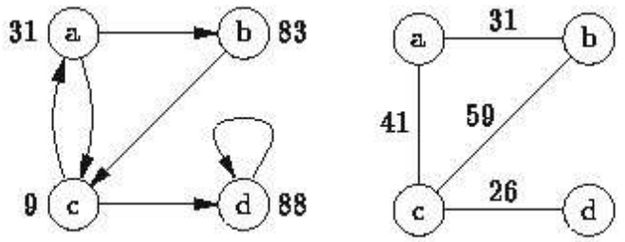


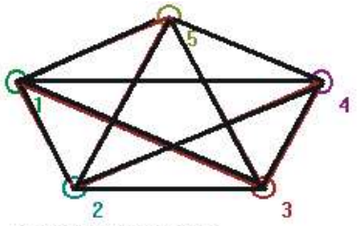
Fig. 14.1 *Examples of Graph*



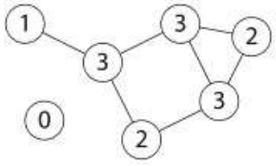
Directed graph with labeled vertices G3

Undirected graph with labeled vertices G4

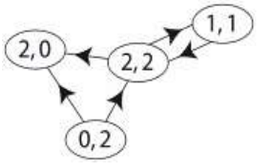
Fig. 14.2 *Examples of Graph*



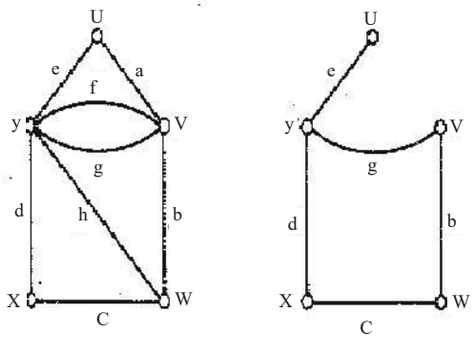
Complete Graph G5



In graph G6 vertices are labelled by their degree.



Graph G vertices are labelled by in-degree and out-degree.



Graph G7 and the subgraph G7

Fig. 14.3 *Examples of Graph Terminology*

Representing Graph in the Memory

A graph can be represented in the memory by three main ways

- Adjacency matrix
- Adjacency list
- Adjacency multi-list

- **Adjacency matrix**

An adjacency matrix is a way of representing graphs in memory. In this representation, an adjacency matrix is prepared for all the vertices that are adjacent to each other. Recall two vertices are called adjacent if there is a direct edge

NOTES

between them. An adjacency matrix for the graph is of order N X N, where N is the number of vertices. For each vertex in the graph, there is a row and a column in the adjacency matrix. The matrix entry is equal to 1 if there is an edge between row vertex and column vertex and 0 otherwise. In this representation, if the graph is an undirected graph, each edge [x, y] is represented twice (one in [x, y] and other in [y. x] cell of the matrix). But in a directed graph for each edge, there is only one entry. The example given in Figure 14.24 is a graph with three vertices a, b and c and three edges < (a, b), (b, c), (a, c)>. Therefore the adjacency matrix will be of order 3 × 3. Following the graph G is a 3 × 3 matrix structure for graph G and the adjacency matrix for graph G. One major disadvantage of adjacency representation is that it requires N × N memory space to represent a graph with N vertices. If the graph is sparse, most of the entries remain (Figure 14.4).

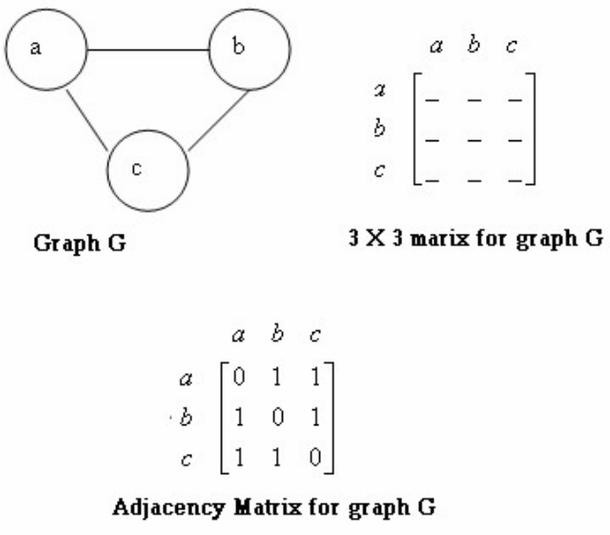


Fig. 14.4 *Adjacency Matrix Representation*

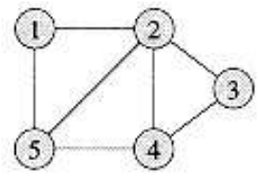
- **Adjacency list**

An adjacency list of a graph is used to keep track of all edges incident to a vertex. This representation can be done with an array of size N, every ith index specifies the list (list of incident edges) for vertex i.

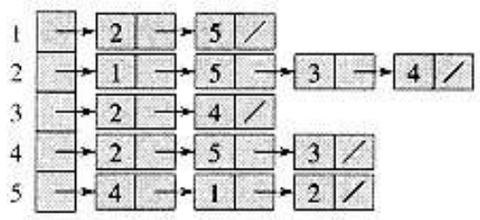
Searching vertices adjacent to each node is easy and a cheap task in this representation. In this structure addition of an edge in this structure is an easy task; whereas deletion of an existing edge is a difficult operation. Example of an adjacency list is shown in Figure 14.5. The graph contains five nodes with six edges. For each vertex there is a corresponding incident list. Vertex 1 is connected with vertex

2 and vertex 5. Therefore adjacency representation contains node 2 and node 5 in list of vertex 1.

NOTES



Graph G



Adjacency List for
Graph G

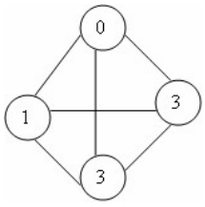
Fig. 14.5 *Adjacency List Representation*

- **Adjacency multi-list**

An adjacency multi-list is a representation in which there are two parts, directory information (an array is used to represent all the vertices of the graph and is called directory) and another part represented by the set of linked list for incident edge information. For each node of graph, there is one entry in the directory information and every directory entry node i points to an adjacency list for node i. Each edge record appears on two adjacency lists. We use the following data structure to represent the node of adjacency list.

Vi	Next 1	Vj	Next 2

Figure 14.6 shows a graph with four nodes and its adjacency multi-list representation.



NOTES

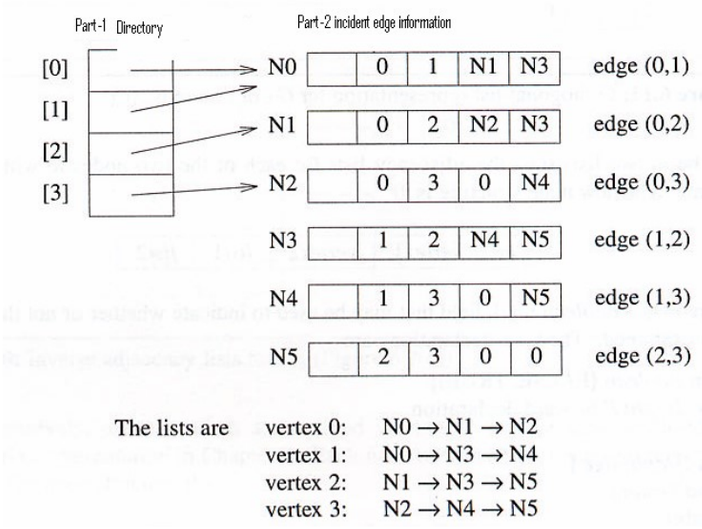


Fig. 14.6 *Adjacency Multi-List Representation*

Comparison between Adjacency Matrix and Adjacency List

An adjacency list is preferred over an adjacency matrix because of its compact structure. Also in case of sparse matrix, adjacency list requires O(E+V) space, which is much less than O(V2) space of adjacency matrix. Adjacency matrix is preferable for dense graphs.

Graph Traversal

There is always a need to traverse a graph to find out the structure of graph used in various applications (recall traversal is visiting each node of a data structure exactly once). There are systematic ways to traverse graph. Graph traversal can start from an arbitrarily chosen vertex (graph does not have a root like tree). Two main challenges in graph traversal are; first graph may contain cycles and second graph may be disconnected (an undirected graph is connected if every pair of vertices are connected with a path). There are two main graph traversal methods. Both of these methods works on directed as well as undirected graphs. These methods are

- Breadth-First Search (BFS)
- Depth-First Search (DFS)

Breadth-First Traversal

The breadth-first search (BFS) algorithm uses queues as a supporting data structure (recall queue is a first-in first-out data structure). BFS always visits level K before visiting level K+ 1 in a graph (Figure 14.7).

NOTES

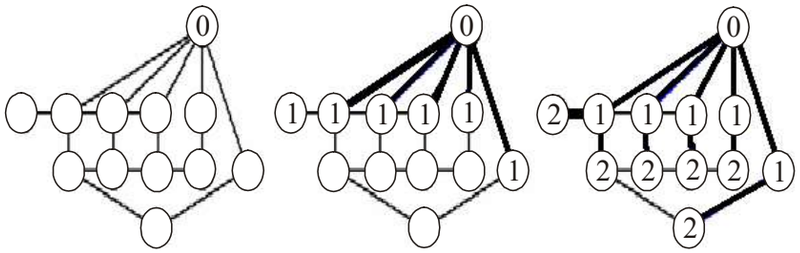


Fig. 14.7 *Example of BFS*

In general, a BFS algorithm works as follows:

- BFS algorithm visits a level at a time from left to right within a level (level is the distance from root).
- First visit the starting node of the graph. Call it node S.
- Then visit all the neighbors of starting node S.
- Then visit all the neighbors of neighbors of node S and so on.
- Keep track of neighbors of each node.
- Also ensure all the nodes of graph are visited exactly once.

To keep track of neighbours of each node a queue data structure is used.

Algorithm BFS (G, S)

// S here is the starting node for graph G. VISITED is an Boolean array of size equal to number of vertex

1. // initialize visited
   For each vertex U ? V(G) do
   VISITED[U]= FALSE
   End for
2. put the starting node S on to the queue.
3. while queue is not empty do
   a. remove the front node X from queue and set VISITED[X] = TRUE
   b. for each neighbor P of X do
   if VISITED[P] = FALSE and not already on queue then
   Add the P to the rear of queue
   end if
   end for (step 3b)
   end while
4. EXIT

NOTES

BFS Example

The following example shows the working of BFS. Given a graph G with 12 vertices and starting vertex is 1. In Figure 14.8, node added to the queue is shown by filled circle and a queue is also shown at every stage

- Traverse vertex 1.

1							

- Now visit neighbors of node 1.

2	3	4	5	6			

- Traverse neighbors of vertex 2.

3	4	5	6	7	8		

- Traverse neighbors of neighbors of node 3.

4	5	6	7	8	9		

- Visit neighbors of vertex 4.

5	6	7	8	9	10		

- Visit neighbors of vertex 5.

6	7	8	9	10	11		

- Visit neighbors of vertex 6.

7	8	9	10	11	12		

- Now visit vertex 7, but none of the neighbors is unvisited.

8	9	10	11	12			

- Now visit vertex 8, but none of the neighbors is unvisited.

9	10	11	12				

- Now visit vertex 9, but none of the neighbors is unvisited.

10	11	12					

- Now visit vertex 10, but none of the neighbors is unvisited.

11	12						

- Now visit vertex 11, but none of the neighbors is unvisited

12							

- Now visit vertex 12, but none of the neighbors is unvisited

.							

- Queue becomes empty, hence stop.

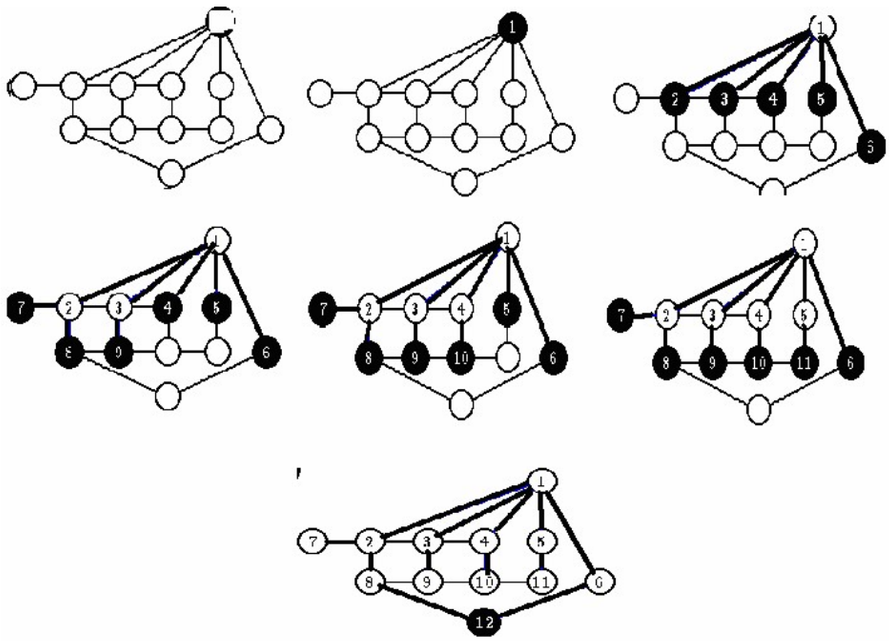


Fig. 14.8 *BFS Algorithm Steps*

Depth-First Traversal

The Depth-First-Traversal (DFS) uses stack as a supporting data structure. DFS begins from the start node and works as follows (recall stack is a first-in last-out data structure). It is a recursive algorithm that records the backtracking path from root to node presently under consideration. DFS is a way of traversal, which is very similar to preorder traversal of a tree. BFS is short and bushy whereas DFS is long and stringy. DFS works as follows:

- It begins from starting node S
- Then visits the node N along the path P which starts at node S.
- Then it visits a neighbor of a neighbor of node S and so on.

- After coming to the end of path P, similarly continue along another path P' and so on.

DFS Algorithm

NOTES

Algorithm DFS (G, S)

// given a graph G and a starting vertex S, it uses a Boolean array VISITED equivalent to the //number of nodes in graph Figure 2.

1) For each vertex U ? V(G) do
   VISITED [U] = FALSE
   End for
2) push the starting node S on to the stack
3) while stack is not empty do
   a) Pop the top node X of the stack and set VISITED[X]=TRUE
   b) For each neighbor P of node N
      If VISITED [P] =false and not already on stack then
      Push P onto the stack
      End if
      End for
      End while (step 3)
4) EXIT

DFS Example

Given a graph G (Figure 14.9) with five vertices A,B,C, D and E. DFS works as follows

Initially

STACK is empty and

VISITED:

A	B	C	D	E
FALSE	FALSE	FALSE	FALSE	FALSE

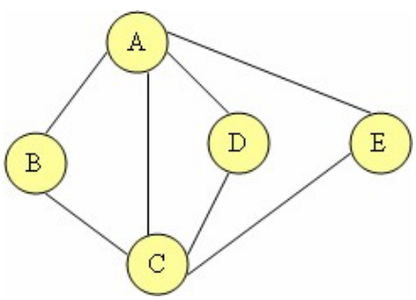


Fig. 14.9 *Graph G*

NOTES

- Firstly it begins with starting vertex S and pushes A onto the stack (Figure 14.10),

  STACK : A <-Top

  VISITED:

A	B	C	D	E
TRUE	FALSE	FALSE	FALSE	FALSE

- Now pop A from stack and push the neighbours of A onto the stack. Here neighbors of A are B, C , D and E.

  STACK : E D C B <-Top

  VISITED:

A	B	C	D	E
TRUE	FALSE	FALSE	FALSE	FALSE

- Now it pops the top of stack B and pushes neighbours of B onto the stack (which are not on the stack till now). Neighbors of B are A and C but they are already on stack so nothing is pushed.

  STACK : E D C <-Top

  VISITED:

A	B	C	D	E
TRUE	TRUE	FALSE	FALSE	FALSE

- Now it pops up C from stack and pushes neighbors of C onto the stack. Here neighbors of C are A, B, D and E. A and B are already visited and D and E are on stack so it backtracks to A.

  STACK : E D <-Top

  VISITED:

A	B	C	D	E
TRUE	TRUE	TRUE	FALSE	FALSE

- Now it pops up D from stack and push neighbors of D onto the stack. Here neighbors of D are A and C, A and C are already visited.

  STACK : E <-Top

  VISITED:

A	B	C	D	E
TRUE	TRUE	TRUE	TRUE	FALSE

NOTES

- Now it pops up E from stack and push neighbors of E onto the stack. Here neighbors of E are A and C. But both A and C are already visited.

  STACK : <-Top

  VISITED:

A	B	C	D	E
TRUE	TRUE	TRUE	TRUE	TRUE

- Now the stack is empty that means all the nodes have already been visited.

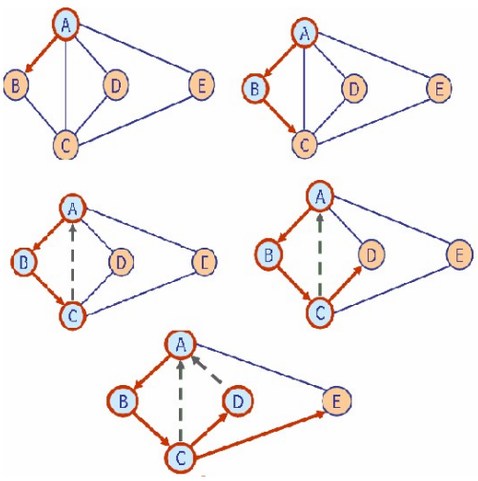


Fig. 14.10 *DFS Algorithm Steps*

14.2.1 Graph Component Algorithms

Finding connected components is required for number of graph applications. A directed graph is strongly connected if every pair of vertices in the graph is connected with each other through a path. Strongly connected components of a directed graph are its maximal strongly connected sub graphs. The strongly connected components form a partition in a given graph. We require two depth-first searches to perform this decomposition. Decomposition is required because many graph algorithms start with such decompositions; the idea generally is to divide any problem into smaller subproblems, one for each strongly connected component. To combine the solution of all subproblems connections between strongly connected components are required. Such structure is also known as the 'component' graph.

In component graph $G^{SCC} = (V^{SCC}, E^{SCC})$ of a graph G (V, E), One vertex for each strongly connected component of G is contained by V^{SCC}. If there is a directed edge from a vertex in strongly connected component of G, corresponding to vertex X, to a vertex in the strongly connected component of G, corresponding to vertex Y, then ESCC contains the edge (X, Y).

NOTES

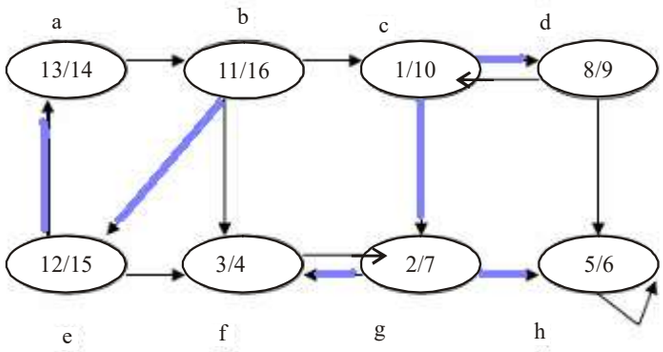
The Kosaraju's algorithm efficiently computes strongly connected components and it is the simplest to understand. There is a better algorithm than Kosaraju's algorithm called the Tarjan's algorithm which improves performance by a factor of two. Tarjan's algorithm is a simple variation of Tarjan's algorithm.

Algorithm STRONGLY_CONNECTED_COMPONENTS (G)

1. Call depth first search to compute f[X] (finishing time) for each vertex X.
2. Compute transpose of graph G as GT.
3. Call the depth first search on GT, but in DFS algorithm considers the vertices in order of decreasing finishing time (finishing time is computed in step 1) .
4. Show the vertices of each tree in DFS forest of step 3 as SCC.

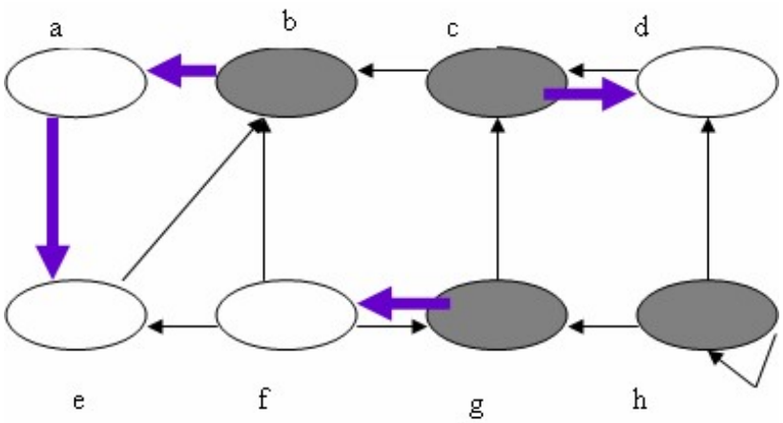
Strongly Connected Components Example

Given is a directed graph G. The SCC of G is shown by dashed lines in Figure 14. 11. This figure shows graph G. Vertices are labeled with discovery and finishing time.



(a)

Now reverse the edge direction to compute the transpose GT of graph G. The depth first tree is shown with shaded edges.



(b)

NOTES

Finally, the acyclic component graph is shown as follows.

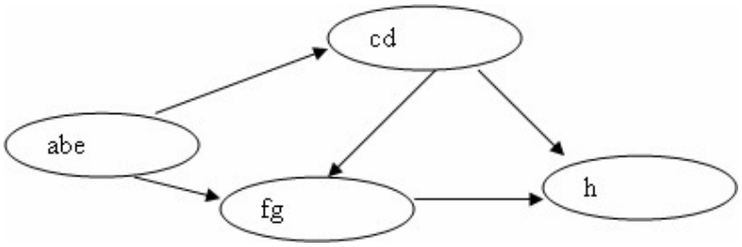


Fig. 14.11(c) *Connected Component Example*

Connected Components and Spanning Trees

In the mathematical field of graph theory, a spanning tree T of an undirected Graph G is a subgraph that is a tree which includes all of the vertices of G, with minimum possible number of edges. In general, a graph may have several spanning trees, but a graph that is not connected will not contain a spanning tree. If all of the edges of G are also edges of a spanning tree T of G, then G is a tree and is identical to T (that is, a tree has a unique spanning tree and it is itself). More precisely, a tree is a connected undirected graph with no cycles. It is a spanning tree of a Graph G if it spans G (that is, it includes every vertex of G) and is a subgraph of G (every edge in the tree belongs to G). While a spanning tree of a connected Graph G can also be defined as a maximal set of edges of G that contains no cycle, or as a minimal set of edges that connect all vertices.

Therefore, a connected Graph G can have more than one spanning tree. All possible spanning trees of Graph G, have the same number of edges and vertices. The spanning tree does not have any cycle (loops). Adding one edge to the spanning tree will create a circuit or loop, i.e., the spanning tree is maximally acyclic.

Definition: A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected.

By this definition, we can draw a conclusion that every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices. The following example illustrated the concept of connected graphs and spanning trees (Refer Figure 14.11 (d)).

In the Figure 14.11 (d), there are three spanning trees of one complete graph. A complete undirected graph can have maximum n^{n-2} number of spanning trees, where n is the number of nodes.

In the given example, since $n = 3$, hence $3^{3-2} = 3$ spanning trees are possible. Therefore, it proves that one graph can have more than one spanning tree.

NOTES

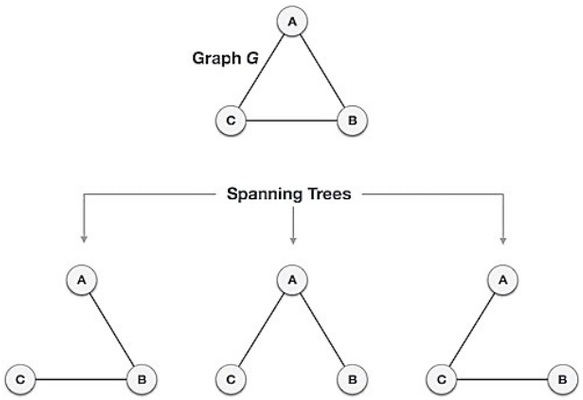


Fig. 14.11 (d) *Three Spanning Trees of One Complete Graph*

General Properties of Spanning Tree

Following are some of the significant properties of the spanning tree connected to Graph G.

- A connected Graph G can have more than one spanning tree.
- All possible spanning trees of Graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e., the spanning tree is **minimally connected**.
- Adding one edge to the spanning tree will create a circuit or loop, i.e., the spanning tree is **maximally acyclic**.

Mathematical Properties of Spanning Tree

- Spanning tree has $n-1$ edges, where n is the number of nodes (vertices).
- From a complete graph, by removing maximum $e - n + 1$ edges, we can construct a spanning tree.
- A complete graph can have maximum n^{n-2} number of spanning trees.
- Consequently, it can be concluded that spanning trees are a subset of connected Graph G and disconnected graphs do not have spanning tree.

Check Your Progress

1. What is a graph?
2. What do the edges contain in a direct graph?
3. What does the Depth-First-Traversal (DFS) uses as a supporting data structure?

NOTES

14.3 NP HARD AND NP COMPLETE PROBLEMS

Basic Concepts

So far, we have come across many problems in this book. For some problems like ordered searching, sorting, etc., there exists polynomial time algorithmic solutions with complexities ranging from O(n) to O(n^2), where n is the size of input. The problems for which there exists (or known) polynomial time solution are class P problems. There are, however, some problems like knapsack, traveling salesperson, etc., for which no polynomial time algorithm is known so far. In addition, no one has yet been able to prove that polynomial time solution cannot exist for these problems. These problems fall under another class of problem that is class NP.

Class NP problems can be further categorized into two classes of problems: *NP-hard* and *NP-complete*. An NP problem has an interesting characteristic according to which it can be solved in polynomial time if and only if every NP problem can be solved in polynomial time. Further, if an NP-hard problem can be solved in polynomial time, then all NP-complete problems can be solved in polynomial time. This implies that all NP-complete problems are NP-hard, but it is not necessary that all NP-hard problems are NP-complete.

Besides the NP-hard and NP-complete classes, there can be more problem classes having characteristic mentioned above. We will restrict our discussion to NP-hard and NP-complete classes, which are computationally related; both of these can be solved using non-deterministic computation.

14.3.1 Non-Deterministic Algorithms

Before proceeding to the concept of non-deterministic algorithms**,** let us first understand what deterministic algorithms are. The **deterministic algorithms** are algorithms in which the result obtained from each operation is uniquely defined. Till now we have been using the deterministic algorithms to solve the problems. However, to deal with NP problems, the above stated limitation on the result of each operation must be removed from the algorithm. The algorithms can be allowed to have the operations whose results are not uniquely defined but are restricted to some specified sets of possibilities. Such algorithms are known as **non-deterministic algorithms.** These algorithms are executed on special machines called **non-deterministic machines**. Such machines do not exist in practice.

For specifying non-deterministic algorithms, three functions are required to be defined which are as follows:

- **Choice(S):** It selects one of the elements of set S; selection is made at random. Consider a statement `a = Choice(1,n).` This statement will assign any one of the values in the range `[1,n]` to `a`. Note that there is not any rule to specify how these values are chosen from a set.
- **Failure():** It indicates that the algorithm terminates unsuccessfully. A non-deterministic algorithm terminates unsuccessfully if and only if there is not a

single set of choices in the specified sets of choices that can lead to the successful completion of the algorithm.

- **Success():** It indicates that algorithm terminates successfully. If there exists a set of choices that can lead to the successful completion of the algorithm, then it is certain that one set of choices will always be selected and the algorithm terminates successfully.

Note that time taken to compute the functions: `Choice()`, `Failure()` and `Success()` is `O(1)`.

Non-Deterministic Search

Consider the problem for searching an element `a` in an unordered set of integers `S[1:n]`, where $n \geq 1$. To solve this problem, we have to find the index `k` containing the element `a` that is `S[k]=a` or `k=0` if `a` does not exist in `s`. The non-deterministic algorithm to solve this problem is given in Algorithm 14.1.

Algorithm 14.1: Non-Deterministic Search

```
NDSEARCH(a,S,n)
1. Set k=Choice(1,n)
2. if (S[k]=a)
3. {
4.    Print k
5.    Success() //item found at position k
6. }
7. Print 0
8. Failure()    //item not found
```

In this algorithm, the function `Choice(1,n)` will choose a possible value from the set of allowable choices. If the subsequent requirement is not met, the function will keep trying to make new choices until it results in successful computation or it runs out of choices. This algorithm will return an index `k` of the list `S` if the element is found. Otherwise, it will terminate and report a failure.

The complexity of executing this non-deterministic algorithm is `O(1)`. Whereas, the complexity of executing every deterministic search is $\Omega(n)$ as `S` is unordered.

Non-Deterministic Sorting

Consider the problem of sorting a given set `S` of `n` unsorted elements, where $1 \leq j \leq n$. The non-deterministic algorithm to solve this problem is given in Algorithm 14.2.

Algorithm 14.2: Non-Deterministic Sorting

```
NDSORT(S,n)
 1. Set j=1
 2. while (j = n) //initializes auxillary array R[]
 3. {
 4.   Set R[j]=0
 5.   Set j=j+1
 6. }
 7.   Set j=1
 8.   while (j = n)  //assigns a position to each s[j] in R
 9.   {
```

NOTES

```
10.       Set k=Choice(1,n)     //determines the position of
                                //each integer s[j]
11.       If (R[k]?0) //ascertains that R[k]has not been
                      //already used
12.          Failure()
13.       Set R[k]=S[j] //assigns s[j] to the k position in R
14.       Set j=j+1
15. }
16. Set j=1
17. while (j = n-1)//checks whether R is in increasing order
18. {
19.       If (R[j]>R[j+1])
20.          Failure()
21.       Set j=j+1
22. }
23. Print R[1:n]
24. Success()
```

The complexity of executing this non-deterministic algorithm is `O(n)`, whereas the complexity of executing every deterministic sort is Ω`(n log n)`.

Observe that in this algorithm, we have considered that each number `S[i]` is distinct. However, this is not always the case; the numbers `S[i]` may not be distinct. In this case, there are many different permutations that can result in a sorted order. That is, the result of the above algorithm will not be uniquely defined. It is possible to specify the non-deterministic algorithms in which there is more than one set of choices that can result in the successful completion. We will restrict our discussion to non-deterministic algorithms that produce unique outcome; particularly non-deterministic decision algorithms. As many optimization problems can be easily remodeled to decision problems. In general, any problem is an **optimization problem** if every possible solution of the problem has some value associated with it and we need to find a solution with maximum or minimum value. On the other hand, a **decision problem** is a problem that always produces either 1 (yes) or 0 (no) as its result. Note that the decision problem corresponding to an optimization problem can be solved in polynomial time if and only if the optimization problem can.

For example, consider the shortest path problem. Given an undirected, unweighted graph `G(V,E)` and we need to find the shortest path between two given vertices of that graph. By shortest path, we mean a path that uses the fewest edges of graph. This is an optimization problem as there may be many paths between given vertices but we are interested in finding a path that uses the fewest edges. Now this shortest path optimization problem can be remodeled to the decision problem: Given an undirected, unweighted graph `G(V,E)` and we need to determine whether there is a path of at most `n` edges between two given vertices for some value of `n`.

Clearly, the above introduced decision problem can be solved by solving the optimization problem and then comparing its result with the input `n`. Thus, if shortest path optimization problem has a polynomial time solution, it is easy to find a polynomial time solution for the corresponding decision problem also. On the other hand, if it is known that this decision problem has no polynomial time solution, the optimization problem also cannot have. From this discussion, a general statement can be made for other cases; if a decision problem cannot be computed in polynomial time, then there is not any way by which corresponding optimization problem can be computed in polynomial time.

NOTES

Knapsack Decision Problem

Given a Knapsack of capacity c and n items, where each item i has a weight wt_i. If a fraction x_i ($0 \leq x_i \leq 1$). of items is kept into the Knapsack, then a profit of $p_i x_i$ is earned. The objective of this optimization problem is to fill the knapsack with the items in such a way that the profit earned is maximum. This optimization problem can be recast into the decision problem. The objective of knapsack decision problem is to check if the value 0 or 1 can be assigned to x_i ($1 \leq i \leq n$) such that $\Sigma p_i x_i \geq mp$ and $\Sigma wt_i x_i \leq c$ where mp is the given number. If this decision problem cannot be computed in deterministic polynomial time, then the optimization problem cannot either. The non-deterministic algorithm for knapsack decision problem is given in Algorithm 14.3.

Algorithm 14.3: Non-Deterministic Knapsack Decision Problem

```
NDKDP(p,wt,n,c,mp,x)
 1. Set W=0
 2. Set P=0
 3. Set i=1
 4. while(i = n)
 5. {
 6.   Set x[i]=Choice(0,1)  //assign 0 or 1 value
 7.   Set W=W+x[i]*wt[i]    //compute total weight
                           //corresponding to the choice of x[]
 8.   Set P=P+x[i]*p[i] //computes total profit
                         //corresponding to the choice of x[]
 9.   Set i=i+1
10. }
11. If ((W>c) OR (P<mp)) //checks if total weight is more
                          //than knapsack capacity or the
                          //resultant profit is less than
mp
12.   Failure()

13. Else
14.   Success()
```

This algorithm will terminate successfully if 0 or 1 can be assigned to each x_i in such a way that the resultant profit is mp, i.e. the result of this decision problem is 'yes'. The complexity of this non-deterministic algorithm is O(n).

Clique Decision Problem

Let us consider first the max clique problem, which is an optimization problem. In this problem, it is required to find out the size of a largest clique in a graph G(V,E), where a **clique** is a complete subgraph of maximum size in a graph. The corresponding decision problem is: Given a graph G and k as an input, the problem is to determine whether the graph contains a clique having size at least k. Further, consider that G is represented by the adjacency matrix, the numb er of vertices is given by n and the input length m is $n^2+\lfloor \log_2 k \rfloor + \lfloor \log_2 n \rfloor + 2$. The non-deterministic algorithm for clique decision problem is given in Algorithm 14.4.

Algorithm 14.4: Non-Deterministic Clique Algorithm

```
NDCDP(G,n,k)
 1. Set S=null         //initializes S to be an empty set
 2. Set i=1
 3. while(i = k)
 4. {
 5. Set t=Choice(1,n)  //chooses a set of k distinct
                       //vertices from a range of 1 to n
```

NOTES

```
 6. If (t∈S)           //determines if these vertices
                       //form a complete sub graph
 7.          Failure()
 8. Set S=Union(S,{t})  //adds t to set S
 9. Set i=i+1
10. }                   //now S has k distinct vertices
11. for all pairs (vi,vj) such that vi∈S, vj∈S and vi? vj
12. {
13.      If (vi,vj)is not an edge in G
14.         Failure()
15. }
16. Success()
```

In this algorithm, the non-deterministic time for executing first while loop is $O(n)$. The time required to execute second while loop is $O(k^2)$. The total non-deterministic time to run this algorithm is $O(n+k^2)=O(n^2)=O(m)$. Note that no polynomial time deterministic algorithm exists for this problem.

Satisfiability Problem

The objective of satisfiability problem is to determine whether a formula is true for some sequence of truth values to its Boolean variables, say x_1, x_2, ... A formula in the propositional calculus is composed of literals (a literal can be either a variable or its negation) and the operators AND ($\wedge$), OR ($\vee$), and NOT ($\bar{}$). We say a formula in n-conjunctive normal form (n-CNF) if it comprises AND of terms that are OR of n Boolean variables or their negations. For example, is in 3-CNF.

For a satisfiability problem, a polynomial time non-deterministic algorithm can be obtained easily if and only if the given formula $F(x_1, x_2,..., x_n)$ is satisfiable. The non-deterministic algorithm for satisfiability problem is given in Algorithm 14.5.

Algorithm 14.5: Non-Deterministic Satisfiability Problem

```
NDSAT(F,n)
//F is the formula and n is the number of variables x1,
x2,..,xn
 1. Set i=1
 2. while(i = n)
 3. {
 4.       Set xi=Choice(false,true)   //selects a truth
                                            value
                                     //for assignment
 5.       Set i=i+1
 6. }
 7. If F(x1,....,xn)
 8.       Success()
 9.   Else
10.       Failure()
```

The computing time for this algorithm is equal to the sum of the time taken to select a truth value ($x_1,.....,x_n$), that is, $O(n)$ and the time required to deterministically evaluate the expression F for that assignment. This time will be proportional to the length of formula F.

14.3.2 NP-Hard and NP-Complete Classes

P is defined as the set of all decision problems that can be solved by deterministic algorithms with in polynomial time, whereas, NP is defined as the set of all decision problems that can be solved by non-deterministic algorithms with in polynomial time.

NOTES

From these definitions, it is clear that P ⊆ NP, as deterministic algorithms are the special cases of non-deterministic algorithms. Now, the thing that is to be determined is whether P=NP or P '"NP. This problem is not solved yet. However, some other useful results have been obtained. One of them is stated above that is, P ⊆ NP. The relationship between P and NP on the basis of assumption P '"NP is depicted in Figure 14.12.

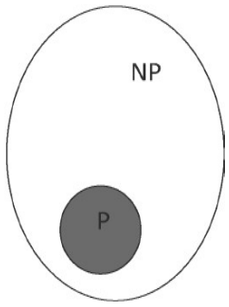


Fig. 14.12 *Relationship between P and NP Classes*

Many researchers have been working on the above stated problem that is whether P=NP or P '"NP. S.Cook tried to solve this problem by devising another question that is if there exists any single problem in NP, which can be proved to be in P. If such a problem exists, then it can be stated that P=NP. He gave the answer to that question in the theorem given as follows:

Cook Theorem: ***Satisfiability is in P if and only if P=NP.***

On the basis of this theorem, the NP complete and NP-hard classes can be formally defined. Prior to that let us first discuss the concept of reducibility. A problem, say K, is said to be *reducible* to another problem, say K_1, (denoted as $K \propto K_1$), if there exists a polynomial time algorithm to solve K_1, then K can be solved in polynomial time. Formally, a problem K reduces to another problem K_1 if there is a method to solve K by a deterministic polynomial time algorithm using a deterministic algorithm that solves K_1 in polynomial time.

Now, NP-hard and NP-complete problems can be defined as:

- **NP-Hard Problem:** A problem K is said to be NP-hard if and only if the following condition holds:
  - Satisfiability $\propto$ K, that is, if satisfiability reduces to K.
- **NP-Complete Problem**: A problem K is said to be NP-complete if and only if the following conditions hold:
  - K is NP-hard and
  - K $\in$ NP

Note: Most researchers believe (not yet proved) that P ≠ NP. Hence, if you find any problem as NP-complete, it is better to devise an algorithm for some particular instances of that problem instead of looking for a polynomial time algorithm for the problem.

Following these definitions, some conclusions can be drawn, which are as follows:

- There exist NP-hard problems that are not NP-complete.
- Only a decision problem can be NP-complete, whereas an optimization problem can be NP-hard.

NOTES

- There is a possibility that $K \propto K_1$, where K is a decision problem and K_1 is an optimization problem.
- It can also be proved that the optimization problems like knapsack, clique can be reduced to corresponding decision problems. Still optimization problems cannot be NP-complete, however decision problems can.
- There are also NP-hard decision problems that are not NP-complete.

The relationship among P, NP, NP-hard and NP-complete problems is depicted in Figure 14.13.

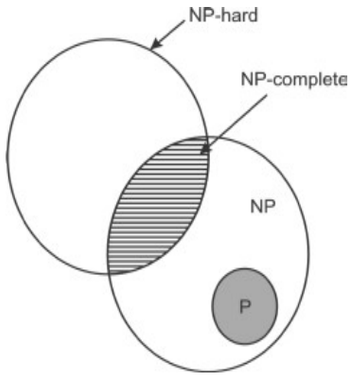


Fig. 14.13 *Relationship among P, NP, NP-Hard and NP-Complete Problems*

Note: Two problems K and K_1 are said to be polynomially equivalent if and only if $K \propto K_1$ and $K_1 \propto K$.

14.3.3 Cook's Theorem

Cook's theorem states that satisfiability is in P if and only if P=NP.

Proof: In the discussion of satisfiability problem, we have observed that satisfiability is in NP. Thus, if P=NP, then satisfiability is in NP.

Now, we have to prove that if satisfiability is in P, then P=NP. For this, we are required to show how a formula Q(A, I) can be obtained from a polynomial non-deterministic decision algorithm A and input I in such a way that Q is satisfiable if and only if A terminates successfully with input I.

Suppose the size of I is n and the time complexity for A is p(n) for some polynomial p(), then the length of Q is $O(p^3(n) \log n)$, which is equal to $O(p^4(n))$. This time is same as that of the time required to construct Q.

Now, a deterministic algorithm Z can be developed for determining the result of algorithm A with any input I. The algorithm Z will first of all compute Q and then check if Q is satisfiable using a deterministic algorithm. If the time required to check whether a formula of length m is satisfiable is O(q(m)), then the time complexity of Z is $O(p^3(n) \log n + q(p^3(n) \log n))$.

If satisfiability is in P, then q(m) is a polynomial function of m and the complexity of Z is O(r(n)) for some polynomial r(). Thus, if satisfiability is in P, then for every non-deterministic algorithm A in NP, a deterministic algorithm Z in P can be developed. Hence, the above construction proves that if satisfiability is in P, then P=NP.

NOTES

Before constructing the formula Q from A and I, several assumptions are made. Such assumptions are made on non-deterministic machines and on the form of algorithm A. These assumptions are as follows:

- Algorithm A would execute on the machine that accepts only words. Suppose, each word is w bits long, then any operation like addition, subtraction, and so on between the numbers having one word length takes one unit of time.
- The expressions in the machine contain an operator and few operands that are simple variables. For example, a + b - c, a + c, -b, etc.
- The variables in A can be either of type integer or Boolean. It does not contain constants. In case, if any constant is present in the algorithm, it will be replaced by a new variable. The prior constants linked with the new variables are taken as part of the input.
- There is no *read* or *write* statement in algorithm A. It accepts input only through its parameters. Variables other than parameters have zero value or false in case of Boolean.
- The statements that can be present in algorithm other than simple assignment statements are:
  - The functions Success() and Failure().
  - The statement goto k, where k is an instruction number.
  - The statement if c then goto a, where c is a Boolean variable and a is an instruction number.
  - The *type declaration* and *dimension statements* (to allocate array space). These are not used at the time of execution of A, so there is no need to translate them into Q.
- A does not take time units more than p(n), (where, p(n) is a polynomial) for any input having length n.

Formula Q uses many Boolean variables. The semantics for two sets of variables used in this formula are as follows:

- **S(i,j,t), where 1d"id"p(n), 1d"jd"w, 0d"td"p(n):** It indicates the status of bit j of word i after computing t steps. Each bit in a word is assigned a number (from right to left); the number starts from 1. Q is constructed so that for any truth value assignment for which Q is true, S(i,j,t) is true if and only if the corresponding bit has value 1 after computing t steps successfully of A with input I.
- **R(j,t), where 1d"jd"l, 1d"td"p(n) where l is the number of instruction in A:** It indicates the instruction to be executed at time t. Here, Q is constructed so that for any truth value assignment for which Q is true, R(j,t) is true if and only if the instruction executed by A at time t is instruction j.

There are six sub-formulas in Q: J, K, L, M, N, and O. Each sub-formula makes declaration which are as follows:

- J states that the initial status of p(n) words represents the input I; the value of all non-input variables is zero.

NOTES

- K states that the instruction to be executed first is instruction 1.
- L states that for any fixed i, exactly one of the R(j,i), 1d"jd"l can be true, that is, at the end of the ith step, there can be only one instruction to be executed next.
- M states that if R(j,i) is true, then
  - R(j,i+1) is also true if instruction j is a Success or Failure statement.
  - R(j+1,i+1) is true, if j is an assignment statement.
  - R(k,i+1) is true, if j is a goto k statement.
  - R(a,i+1) is true if j is if c then a statement and c is true. In case c is false, R(j+1,i+1) is true.
- N states that if the instruction executed at step t is not an assignment statement, then S(i,j,t)'s remain unchanged. But, if this instruction is an assignment statement, then the variable placed on the left-hand side of assignment statement can only change. This change will be determined by the right-hand side of the statement.
- O states that the instruction to be executed at time p(n) is a Success instruction. Thus, the computation is terminated successfully.

On the basis of above declarations, it can be said that Q=J ∧ K ∧ L ∧ M ∧ N ∧ O is satisfiable if and only if A terminates successfully with input I. Further, we are going to present the formulas from J to O. These formulas can be transformed into CNF. Due to this transformation, the length of Q is increased by an amount which is dependent on w and l but independent of n. It enables us to show that CNF-satisfiability is NP-complete.

1. Formula J which describes the input is given by:

$$J = \bigwedge_{\substack{1 \le i \le p(n) \\ 1 \le j \le w}} T(i, j, 0)$$

Here, if the input calls for bit S(i,j,0) to be1, then T(i,j,0) is S(i,j,0); otherwise, T(i,j,0) is equal to Hence, if there is no input, then:

$$J = \bigwedge_{\substack{1 \le i \le p(n) \\ 1 \le j \le w}} \bar{S}(i, j, 0)$$

It is clear that J is uniquely defined by I and is in CNF. Further, it is satisfiable only by the truth assignment representing the initial values of all variables in A.

2. Formula K is given by:

$$K = R(1, 1) \wedge \bar{R}(2, 1) \wedge \bar{R}(3, 1) \wedge \ldots \wedge \bar{R}(1, 1)$$

Observe that K is satisfiable if and only if the assignment R(1,1)=true and R(i,1)=false, 2d"id"l. Using the interpretation of R(i,1), it can be said that K is true if instruction 1 is executed first. Also note that K is in CNF.

3. Formula L is given by:

$$L = \wedge_{1 < t \le p(n)} L_t$$

Where, each L_t states that there is a unique instruction for step t and is defined as:

$$L_t = (R(1, t) \vee (R(2, t) \vee \ldots \vee R(l, t)) \wedge (\bigwedge_{\substack{1 \le j \le l \\ 1 \le k \le l \\ j \ne k}} (\overline{R}(j, t) \vee \overline{R}(k, t))$$

L_t is true if and only if exactly one of the $R(j,t)$'s is true where $1d''jd''l$. Also note that L is in CNF.

4. The formula M is given by:

$$M = \bigwedge_{\substack{1 \le i \le l \\ 1 \le t < p(n)}} M_{i,t}$$

Here, each $M_{i,t}$ states that either the instruction i is not the one which will be executed at time t, or if it is executed at time t, then the instruction to be executed at time t+1 will definitely be determined by the instruction i. $M_{i,t}$ is defined as:

$$M_{i,t} = \overline{R}(i, t) \vee B$$

Where, B is defined on the basis of various conditions as follows:

- $B = R(i,t+1)$, if instruction i is Success or Failure.
- $B = R(k,t+1)$, if instruction i is goto k.

if instruction i is if X then goto k, where X is a Boolean variable represented by word j. Here, it is assumed that if variable X is true, then only the value of bit 1 of X will be 1.

- $B = R(i+1,t+1)$, if instruction i is not any of the above.

In the first, second and fourth conditions, $M_{i,t}$'s are in CNF. While in third condition, it can be transformed into CNF by using a Boolean identity which is:

$$a \vee (b \wedge c) \vee (d \wedge e) \equiv (a \vee b \vee d) \wedge (a \vee c \vee d) \wedge (a \vee b \vee e) \wedge (a \vee c \vee e)$$

5. The formula N is given by:

$$N = \bigwedge_{\substack{1 \le i \le l \\ 1 \le t < p(n)}} N_{i,t}$$

Here, each $N_{i,t}$ states that at time t, the instruction i is not executed or it is and the status of the p(n) words after step t is correct with respect to the status before step t and the resultant changes from i. Formally, $N_{i,t}$ is defined as:

$$N_{i,t} = \overline{R}(i, t) \vee T$$

Where, T is defined on the basis of various conditions as follows:

- If instruction i is a goto, if-then-goto-, Success, or Failure statement, then

$$T = \bigwedge_{\substack{1 \le k \le p(n) \\ 1 \le j \le w}} ((S(k, j, t-1) \wedge S(k, j, t)) \vee (\overline{S}(k, j, t-1) \wedge \overline{S}(k, j, t))$$

where, T states that the status of p(n) words remains unchanged. Note that $N_{i,t}$'s can be transformed into CNF.

NOTES

NOTES

- If i is an instruction of type <simple variable>:=<array variable>, then T will be similar to be obtained for instruction of type <array variable>:=<simple variable>.
- If i is in the form of C:= Choice(D), then

$$T = \bigvee_{1 \le j \le k} T_j$$

Where, D is a set of the form of either $\{D_1, D_2, \ldots, D_n\}$ or r, u.

6. The formula O is given by

$$O = R(i_1, p(n)) \vee R(i_2, p(n)) \vee \ldots \vee R(i_k, p(n))$$

Where, $i_1, i_2, i_3, \ldots, i_k$ are the number of the statements corresponding to success statements in algorithm A.

On the basis of above discussion, it can be verified that $Q = J \wedge K \wedge L \wedge M \wedge N \wedge O$ is satisfiable if and only if the algorithm A with input I terminates successfully. Formula Q can be transformed into CNF. Further, observations are as follows:

- Formula J contains wp(n) literals, K contains l literals, L contains O(l2p(n)) literals, M contains O(lp(n)) literals, N contains O(lwp3(n)) literals, and O contains at most l literals. This means the total number of literals that Q contains is O(lwp3(n)), that is, nothing but O(p3(n)) since lw is constant.
- Since Q has O(wp2(n)+lp(n)) distinct literals, the bits needed to write each literal is O(log wp2(n)+lp(n)) which is equal to O(log n).Thus, the length of Q is O(p3(n)log n)= O(p4(n)) as p(n) is at least n.
- The time to construct Q from A and I is also O(p3(n)log n).

From the construction of formula Q, following conclusions can be drawn:

- Every problem in NP reduces to satisfiability as well as to CNF-satisfiability. Thus, if any one of these two problems is in P, then $NP \subseteq P$ and so P=NP.
- Since satisfiability is in NP, the construction of a CNF formula Q proves that satisfiability $\propto$ CNF-Satisfiability.
- Since satisfiability $\propto$ CNF-satisfiability and CNF-satisfiability is in NP, CNF-satisfiability is in NP-complete.
- Since satisfiability $\propto$ satisfiability and satisfiability is in NP, satisfiability is also NP-complete.

Check Your Progress

4. In what categories can class NP problems be categorized into?
5. What is the objective of satisfiability problem?

14.4 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

NOTES

1. A graph is a non-linear data structure.
2. In a directed graph, the edges consist of an ordered pair of vertices.
3. The Depth-First-Traversal (DFS) uses stack as a supporting data structure.
4. Class NP problems can be categorized into two classes of problems: *NP-hard* and *NP-complete*.
5. The objective of satisfiability problem is to determine whether a formula is true for some sequence of truth values to its Boolean variables.

14.5 SUMMARY

- A graph is a non-linear data structure. A data structure in which each node has at most one successor node is called a linear data structure, for example array, linked list, stack, queue etc.
- Many problems can be naturally formulated as in terms of elements and their interconnections.
- In a directed graph, the edges consist of an ordered pair of vertices (one-way edges).
- A weighted graph is one in which edges are associated with some weight; this weight can be distance, time or any cost function.
- Every edge contributes in the degree of the exactly two vertices in an undirected graph.
- An adjacency matrix is a way of representing graphs in memory.
- An adjacency list of a graph is used to keep track of all edges incident to a vertex.
- Searching vertices adjacent to each node is easy and a cheap task in this representation.
- An adjacency list is preferred over an adjacency matrix because of its compact structure.
- There is always a need to traverse a graph to find out the structure of graph used in various applications (recall traversal is visiting each node of a data structure exactly once).
- The Depth-First-Traversal (DFS) uses stack as a supporting data structure.
- A spanning tree of a graph G is a connected subgraph G' which is a tree and contains all the vertices (but fewer edges) of graph G.A spanning tree of graph G with N vertices contain N-1 edges.
- The Kosaraju's algorithm efficiently computes strongly connected components and it is the simplest to understand. There is a better algorithm than Kosaraju's algorithm called the Tarjan's algorithm which improves

NOTES

performance by a factor of two. Tarjan's algorithm is a simple variation of Tarjan's algorithm.

- The deterministic algorithms are algorithms in which the result obtained from each operation is uniquely defined.
- Class NP problems can be categorized into two classes of problems: *NP-hard* and *NP-complete*.

14.6 KEY WORDS

- **Choice(S)**: It selects one of the elements of set S; selection is made at random.
- **Failure():** It indicates that the algorithm terminates unsuccessfully.A non-deterministic algorithm terminates unsuccessfully if and only if there is not a single set of choices in the specified sets of choices that can lead to the successful completion of the algorithm.
- **Success():** It indicates that algorithm terminates successfully. If there exists a set of choices that can lead to the successful completion of the algorithm, then it is certain that one set of choices will always be selected and the algorithm terminates successfully.

14.7 SELF ASSESSMENT QUESTIONS AND EXERCISES

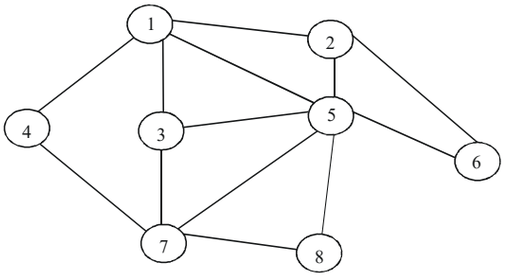
Short Answer Questions

1. What is graph?
2. Differentiate between the directed graph and weighted graph giving example.
3. Explain the terms Depth First Search (DFS) and Breadth First Search (BFS).
4. What will a graph look like if a row of its adjacency matrix consists of only zeroes?
5. What is spanning tree?
6. Explain NP hard and NP complete problems.
7. Explain the use of Cook's theorem.
8. Discuss and differentiate between NP-Hard and NP-Complete Classes.

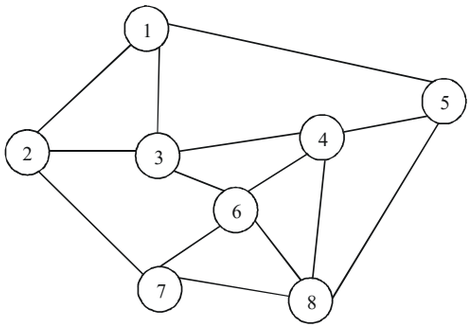
Long Answer Questions

1. Explain the terminologies associated with graphs.
2. Discuss the methods used for traversing a graph. Explain giving appropriate examples.
3. Briefly explain the graph component algorithms giving examples.

4. Explain the basic concepts and theories of NP Hard and NP Complete problems and classes.
5. Briefly explain about the non-deterministic algorithms giving appropriate examples.
6. "Finding connected components is required for number of graph applications. A directed graph is strongly connected if everypair of vertices in the graph is connected with each other through a path." Discuss.
7. "Cook's theorem states that satisfiability is in P if and only if P=NP." Explain.
8. How is Breadth-First Search (BFS) different from Depth-First Search (DFS)? Find the sequence in which vertices of the following graph will be visited during Breadth-First Search and Depth-First Search.



9. Find all the possible spanning trees for the following graph:



14.8 FURTHER READINGS

Levitin, Anany. *Introduction to Design and Analysis of Algorithms*. Delhi: Pearson Education.

Ellis Horowitz, S. Sahani and Rajasekaran, *Fundamentals of Computer Algorithms.* Delhi: Galgotia Publications.

Goodrich, M T and R. Tomassia. *Algorithm Design: Foundations, Analysis and Internet Examples*. Delhi: John wiley and Sons.