MCA12

KRISHNA KANTA HANDIQUI STATE OPEN UNIVERSITY

Housefed Complex, Dispur, Guwahati - 781 006

Master of Computer Applications

DESIGN AND ANALYSIS OF ALGORITHMS

CONTENTS

Subject Expert

Prof. Anjana Kakati Mahanta, Deptt. of Computer Science, Gauhati University
Prof. Jatindra Kr. Deka, Deptt. of Computer Science and Engineering,
Indian Institute of Technology, Guwahati
Prof. Diganta Goswami, Deptt. of Computer Science and Engineering,
Indian Institute of Technology, Guwahati

Course Coordinator

Tapashi Kashyap Das, Assistant Professor, Computer Science, KKHSOU
Arabinda Saikia, Assistant Professor, Computer Science, KKHSOU

SLM Preparation Team

Units	Contributors
1, 4, 6	**Nabajyoti Sarma** Guest Faculty, Deptt. of Computer Science, Gauhati University
2, 3, 5, 7	**Irani Hazarika** Guest Faculty, Deptt. of Computer Science, Gauhati University

July 2012

Printed and published by Registrar on behalf of the Krishna Kanta Handiqui State Open University.

The university acknowledges with thanks the financial support provided by the **Distance Education Council, New Delhi**, for the preparation of this study material.

Housefed Complex, Dispur, Guwahati- 781006; Web: www.kkhsou.in

COURSE INTRODUCTION

This course is on ***Design and Analysis of Algorithms***. An algorihm is a systematic method containing a sequence of instructions to solve a computational problem. It takes some inputs, performs a well defined sequence of steps and produces some output. Algorithm design and analysis form a central theme in computer science. With this course we illustrate various concepts associated with algorithm design and analysis. The course consists of the following seven units:

Unit - 1 is an introductory unit on algorithms.With this unit learners will be acquainted with analysis of algorithm, complexity, various notations etc.
Unit - 2 concentrates on divide and conquer.
Unit - 3 is on Greedy method.
Unit - 4 concentrates on dynamic programming. .
Unit - 5 deals with backtracking.
Unit - 6 is on branch and bound.
Unit - 7 is on NP-Hard and NP-complete problems.

Each unit of this course includes some along-side boxes to help you know some of the difficult, unseen terms. Some “EXERCISES” have been included to help you apply your own thoughts. You may find some boxes marked with: “LET US KNOW”. These boxes will provide you with some additional interesting and relevant information. Again, you will get “CHECK YOUR PROGRESS” questions. These have been designed to make you self-check your progress of study. It will be helpful for you if you solve the problems put in these boxes immediately after you go through the sections of the units and then match your answers with “ ANSWERS TO CHECK YOUR PROGRESS” given at the end of each unit.

MASTER OF COMPUTER APPLICATIONS

Design and Analysis of Algorithm

DETAILED SYLLABUS

Unit 1 : Introduction to Algorithms (Marks:)
Algorithm, analysis, time complexity and space complexity, O-notation, Omega notation and Theta notation, Heaps and Heap sort, Sets and disjoint set, union and find algorithms. Sorting in linear time.

Unit 2 : Divide and Conquer (Marks:)
Divide and Conquer: General Strategy, Exponentiation. Binary Search, Quick Sort and Merge Sort

Unit 3 : Greedy Method (Marks:)
General Strategy, Knapsack problem, Job sequencing with Deadlines, Optimal merge patterns, Minimal Spanning Trees and Dijkstra's algorithm.

Unit 4 : Dynamic Programming (Marks:)
General Strategy, Multistage graphs, OBST, 0/1 Knapsack, Traveling Salesperson Problem, Flow Shop Scheduling

Unit 5 : Backtracking (Marks:)
Backtracking: General Strategy, 8 Queen's problem, Graph Coloring, Hamiltonian Cycles, 0/1 Knapsack

Unit 6 : Branch and Bound (Marks:)
General Strategy, 0/1 Knapsack, Traveling Salesperson Problem

UNit 7 : P, NP-HARD AND NP-COMPLETE PROBLEMS (Marks:)
Basic concepts, non-deterministics algorithms, NP-HARD and NP-COMPLETE classes, COOKS theorem

UNIT – 1 INTRODUCTION TO ALGORITHM

UNIT STRUCTURE

1.1 LEARNING OBJECTIVES

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

- understand the concept of algorithm
- know the process of algorithm analysis
- know the notations for defining the complexity of algorithm
- learnthe method to calculate time complexity of algorithm
- know the different operations on disjoint set
- learn methods for sorting data in linear time

1.2 INTRODUCTION

The concept of an algorithm is the basic need for any programming development in computer science. Algorithm exists for many common problems, but designing an efficient algorithm is a

challenge and it plays a crucial role in large scale computer system. In this unit we will discuss about the algorithm analysis. Also we will discuss few algorithms for sorting data in linear time. We will also discuss algorithm for disjoint set operations.

1.3DEFINITION OF ALGORITHM

Definition: An algorithm is a well-defined computational method, which takes some value(s) as input and produces some value(s) as output.In other words, an algorithm is a sequence of computational steps that transforms input(s) into output(s).

Each algorithm must have

- **Specification:** Description of the computational procedure.
- **Pre-conditions:** The condition(s) on input.
- **Body of the Algorithm:** A sequence of clear and unambiguous instructions.
- **Post-conditions**: The condition(s) on output.

Consider a simple algorithm for finding the factorial of ***n***.

Algorithm Factorial (n)

```
Step 1: FACT = 1
Step 2: for i = 1 to n do
Step 3: FACT = FACT * i
Step 4: print FACT
```

In the above algorithm we have:

Specification: Computes *n*!.
Pre-condition: *n*>= 0
Post-condition: FACT = *n*!

1.4 ALGORITHM ANALYSIS

Programming is a very complex task, and there are a number of aspects of programming that make it so complex. The first is that most programming projects are very large, requiring the coordinated efforts of many people. (This is the case of software engineering.) The next is that manyprogramming projects involve storing and accessing large quantities of data efficiently. (This is the case of data structures and databases.) The last is that many programming projects involve solving complexcomputational problems, for which simplistic or naive solutions may not be efficient enough. The complexproblems may involve numerical data which need to computed accurately up to high precision (in

caseof numerical analysis). This is where the topic of algorithm design and analysis is important.

Although the algorithms discussed in this course will often represent only a tiny fraction of the code that isgenerated in a large software system, this small fraction may be very important for the success of the overallproject.

If unfortunately someonedesign an inefficient algorithm anddata structure to solve the problem, and then take the poor design and attempt to fine-tune its performance, then often no amount of fine-tuning is going to make a substantialdifference. So at the design phase of the algorithm itself care should be taken to design an efficient algorithm.

The focus of this course is on how to design good algorithms, and how to analyze their efficiency. This is one of the most basic aspects of good programming.

1.5 COMPLEXITY

Once we develop an algorithm, it is always better to check whether the algorithm is efficient or not. The efficiency of an algorithm depends on the following factors:

- Accuracy of the output
- Robustness of the algorithm
- User friendliness of the algorithm
- Time required to run the algorithm
- Space required to run the algorithm
- Reliability of the algorithm
- Extensibility of the algorithm

To be a good program, all the above mentioned factors are very important. When we design some algorithm it should be user friendly and produce correct output(s) for all the possible set of input(s). A well designed algorithm should not take very long amount of time and also it should not uselarge amount of main memory. A well design algorithm is always reliable and it can be extended as per requirement.

In case of complexity analysis, we mainly concentrate on the time and space required by a program to execute.So complexity analysis is broadly categorized into two classes

- Space complexity
- Time complexity

1.5.1 SPACE COMPLEXITY

Now a day's,memory is becoming more and more cheaper, even though it is very much important to analyze the amount of memory used by a program. Because, if the algorithm takes memory beyond the capacity of the machine, thenthe algorithm will not beable to execute. So, it is very much important to analyze the space complexity before execute it on the computer.

Definition [SpaceComplexity]: The Space complexity of an algorithm is the amount of main memory needed to run the program till completion.

To measure the space complexity in absolute memory unit has the following problems

The space required for an algorithm depends on space required by the machine during execution, they are

i) Programspace
ii) Data space.

i) The program space is fixed and it is used to store the temporary data, object code,etc.
ii) The data space is used to store the different variables, data structures defined in the program.

In case of analysis we consider only the data space, since program space is fixed and dependon the machine where it is executed.

Consider the following algorithms for exchange two numbers:

Algo1_exchange (a, b)

```
Step 1: tmp = a;
Step 2: a = b;
Step 3: b = tmp;
```

Algo2_exchange (a, b)

```
Step 1: a = a + b;
Step 2: b = a - b;
Step 3: a = a - b;
```

The first algorithm uses three variables a, b and tmpand the second one take only two variables, so if we look from the space complexity perspective the second algorithm is better than the first one.

1.5.2TIME COMPLEXITY

Definition [Time Complexity]: The Time complexity of an algorithm is the amount of computer time it needs to run the program till completion.

To measure the time complexity in absolute time unit has the following problems

1. The time required for an algorithm depends on number of instructions executed by the algorithm.
2. The execution time of an instruction depends on computer's power. Since, different computers take different amount of time for the same instruction.
3. Different types of instructions take different amount of time on same computer.

For time complexity analysis we design a machine by removing all the machine dependent factors called Random Access Machine (RAM). The random access machine model of computation was devised by John von Neumann to study algorithms. The design of RAM is as follows

1. Each “simple” operation (+, -, =, if, call) takes exactly 1 unit cost.
2. Loops and subroutine calls are not simple operations, they depend upon the size of the data and the contents of a subroutine.
3. Each memory access takes exactly 1 unit cost.

Consider the following algorithm for add two number

```
Algo_add (a,b)

Step 1. C = a + b;
Step 2.return C;
```

Here this algorithm has only two simple statements so the complexity of this algorithm is 2

Consider another algorithm for add n even number

```
Algo_addeven (n)

Step 1. i = 2;
Step 2.sum = 0;
Step 3.while i <= 2*n
Step 4.sum = sum + i
Step 5. i = i + 2;
Step 6.end while;
Step 7.return sum;
```

Here,
Step 1, Step 2 and Step 7 are simple statements and they will execute only once.
Step 3 is a loop statement and it will execute as many times as the loop condition is true and one more time for check the condition is false.
Step 5 and Step 6 are inside the loop so it will run as much as the loop condition is true
Step 6 just indicate the end of while and no cost associated with it.

Statement	Cost	Frequency	Total cost
Step 1. i = 2;	1	1	1
Step 2.sum = 0;	1	1	1
Step 3.while i <= 2*n	1	n+1	n+1
Step 4.sum = sum + i	1	n	n
Step 5. i = i + 2;	1	n	n
Step 6.end while;	0	1	0
Step 7.return sum;	1	1	1
Total cost			3n+4

CHECK YOUR PROGRESS

1. State True or False
   a) Time complexity is the time taken to design an algorithm.
   b) Space complexity is the amount of space required by a program during execution
   c) An algorithm may not produce any output.
   d) Algorithm are computer programs which can be directly run into the computer.
   e) If an algorithm is designed for a problem then it will work for all the valid inputs for the problem.

1.6ASYMPTOTIC NOTATION

When we calculate the complexity of an algorithm we often get a complex polynomial. To simplify this complex polynomial we use some notation to represent the complexity of an algorithm called: Asymptotic Notation.

Θ (Theta) Notation

For a given function g(n), Θ(g(n)) is defined as

$$\Theta(g(n)) = \begin{cases} f(n): \text{there exist constants } c_1 > 0,\ c_2 > 0 \text{ and } n_0 \in N \\ \qquad \text{such that } 0 \le c_1\, g(n) \le f(n) \le c_2\, g(n) \text{ for all } n \ge n_0 \end{cases}$$

In other words a function f(n) is said to belongs to Θ(g(n)), if there exists positive constants c_1 and c_2 such that $0 \le c_1\, g(n) \le f(n) \le c_2 g(n)$ for sufficiently large value of n. Fig 1.1 gives a intuitive picture of functions f(n) and g(n), where f(n) = Θ (g(n)). For all the values of n at and to right of n_0, the values of f(n) lies at or above $c_1 g(n)$and at or below $c_2 g(n)$.In other words, for all $n \ge n_0$, the function f(n) is equal to g(n) to within a constant factor. So,g(n) is said an **asymptotically tight bound** for f(n).

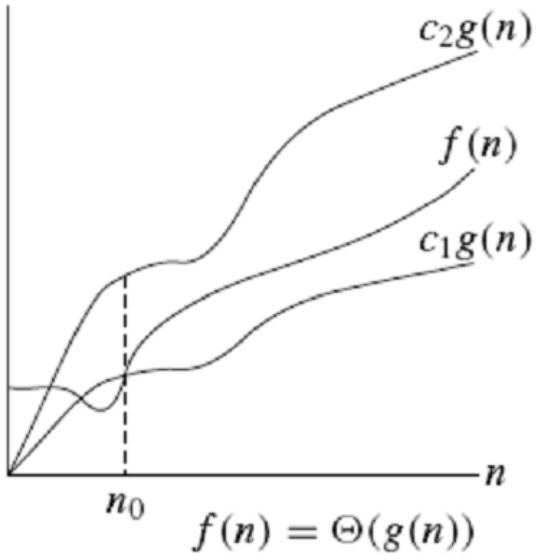


Fig 1.1 : Graphic Example of Θ notation.

For example

$f(n) = ½\ n^2 - 3n$

let, $g(n) = n^2$

to proof f(n) = Θ (g(n)) we must determine the positive constants c_1, c_2 and n_0 such that

$c_1 n^2 \le ½\ n^2 - 3n \le c_2 n^2$ for all $n \ge n_0$

dividing the whole equation by n^2, we get

$$c_1 \leq ½ - 3/n \leq c_2$$

We can make the right hand inequality hold for any value of $n \geq 1$ by choosing $c_2 \geq ½$. Similarly we can make the left hand inequality hold for any value of $n \geq 7$ by choosing $c_1 \leq {}^1/_{14}$. Thus, by choosing $c_1 = {}^1/_{14}$, $c_2 = ½$. And $n_0 = 7$ we can have $f(n) = \Theta(g(n))$. That is $½n^2 - 3n = \Theta(n^2)$.

O (Big O) Notation

For a given function g(n), O(g(n)) is defined as

$$O(g(n)) = \begin{cases} f(n): \text{there exist constants } c > 0, \text{ and } n_0 \in N \\ \quad \text{such that } 0 \leq f(n) \leq c\, g(n) \text{ for all } n \geq n_0 \end{cases}$$

In other words a function f(n) is said to belongs to O(g(n)), if there exists positive constant c such that $0 \leq f(n) \leq c\, g(n)$ for sufficiently large value of n.Fig 1.2 gives a intuitive picture of functions f(n) and g(n), where f(n) = O (g(n)). For all the values of n at and to the right of n_0, the values of f(n) lies at or below cg(n). Sog(n) is said an **asymptotically upper bound** for f(n).

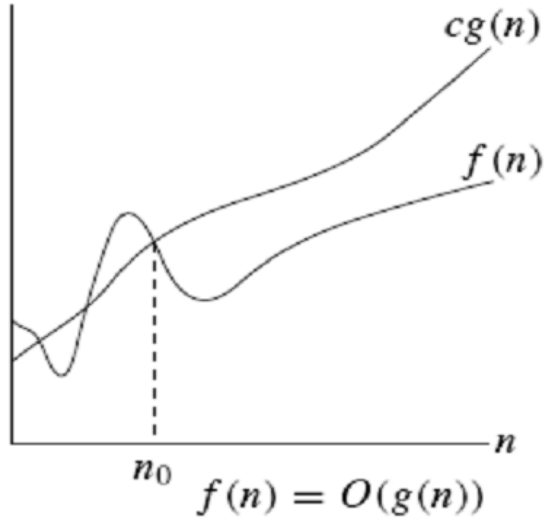


Fig 1.2 : Graphic Example of O notation.

Ω (Big Omega) Notation

For a given function g(n), Ω (g(n)) is defined as

$$\Omega(g(n)) = \begin{cases} f(n): \text{there exist constants } c > 0, \text{ and } n_0 \in N \\ \quad \text{such that } 0 \leq c\, g(n) \leq f(n) \text{ for all } n \geq n_0 \end{cases}$$

In other words, a function f(n) is said to belongs to Ω (g(n)), if there exists positive constant c such that 0 ≤ c g(n) ≤ f(n) for sufficiently large value of n. Fig 1.3 gives a intuitive picture of functions f(n) and g(n), where f(n) = Ω (g(n)). For all the values of n at and to the right of n_0, the values of f(n) lies at or above cg(n). Sog(n) is said an **asymptotically lower bound** for f(n).

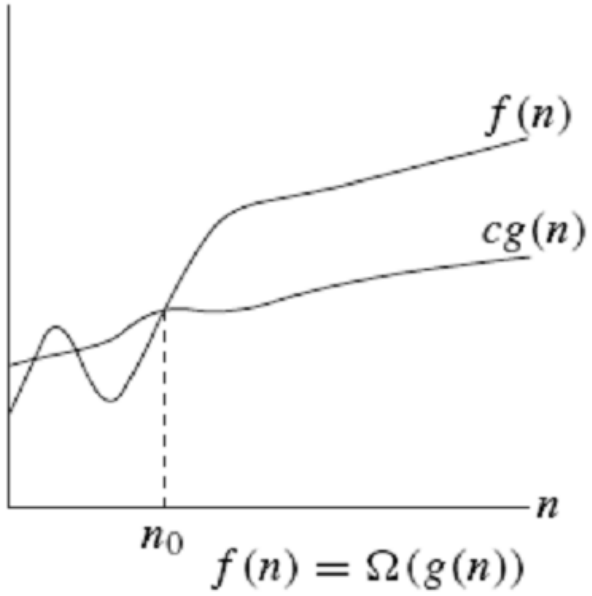


Fig 1.3 : Graphic Example of notation

The growth patterns of order notations have been listed below:

O(1) < O(log(n)) < O(n) < O(n log(n)) < $O(n^2)$ < $O(n^3)$… <$O(2^n)$.

The common name of few order notations is listed below:

Notation	Name
O(1)	Constant
O(log(n))	Logarithmic
O(n)	Linear
O(n log(n))	Linearithmic
$O(n^2)$	Quadratic
$O(c^n)$	Exponential
O(n!)	Factorial

A Comparison of typical running time of different order notations for different input size listed below:

$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	4294967296

Now let us take few examples of above asymptotic notations

1. **Prove that $3n^3 + 2n^2 + 4n + 3 = O(n^3)$**

Here,

$f(n) = 3n^3 + 2n^2 + 4n + 3$

$g(n) = O(n^3)$

to proof $f(n) = O(g(n))$ we must determine the positive constants c and n_0 such that

$3n^3 + 2n^2 + 4n + 3 \leq c\, n^3$ for all $n \geq n_0$

dividing the whole equation by n^3, we get

$3 + 2/n + 4/n^2 + 3/n^3 \leq c$

We can make the inequality hold for any value of $n \geq 1$ by choosing $c \geq 12$. Thus, by choosing $c \geq 12$and $n_0 = 1$ we can have $f(n) = O(g(n))$.

Thus,$3n^3 + 2n^2 + 4n + 3 = O(n^3)$.

2. **Prove that $3n^3 + 2n^2 + 4n + 3 = \Omega(n^3)$**

Here,

$f(n) = 3n^3 + 2n^2 + 4n + 3$

$g(n) = O(n^3)$

to proof $f(n) = \Omega(g(n))$ we must determine the positive constants c and n_0 such that

$c\, n^3 \leq 3n^3 + 2n^2 + 4n + 3$ for all $n \geq n_0$

dividing the whole equation by n^3, we get

$c \leq 3 + 2/n + 4/n^2 + 3/n^3$

We can make the inequality hold for any value of $n \geq 1$ by choosing $c \leq 3$. Thus, by choosing $c = 3$ and $n_0 = 1$ we can have $f(n) = \Omega(g(n))$.

Thus, $3n^3 + 2n^2 + 4n + 3 = \Omega(n^3)$.

3. **Prove that $7n^3 + 7 = \Theta(n^3)$**

Here,

$f(n) = 7n^3 + 7$

$g(n) = O(n^3)$

to proof $f(n) = \Theta(g(n))$ we must determine the positive constants c_1, c_2 and n_0 such that

$c_1\, n^3 \leq 7n^3 + 7 \leq c_2\, n^3$ for all $n \geq n_0$

dividing the whole equation by n^3, we get

$$c_1 \leq 7 + 7/n^3 \leq c_2$$

We can make the right hand inequality hold for any value of $n \geq 1$ by choosing $c_2 \geq 14$. Similarly we can make the left hand inequality hold for any value of $n \geq 1$ by choosing $c_1 \leq 7$. Thus, by choosing $c_1 = 7$, $c_2 = 14$. And $n_0 = 1$ we have $f(n) = \Theta(g(n))$.

Thus, $7n^3 + 7 = \Theta(n^3)$.

Now let us take few examples of Algorithms and represent their complexity in asymptotic notations

Example 1. Consider the following algorithm to find out the sum of all the elements in an array

Statement	**Cost**	**Frequency**	**Total cost**
Sum_Array(arr[], n)			
Step 1. i = 0;	1	1	1
Step 2. s = 0;	1	1	1
Step 3.while i < n	1	n+1	n+1
Step 4. s = s + arr [i]	1	n	n
Step 5. i = i + 1;	1	n	n
Step 6.end while;	0	1	0
Step 7.return s;	1	1	1
Total Cost			3n + 4

So,
Here f(n) = 3n + 4
Let, g(n) = n
If we want to represent it in O notation then we have to show that for some positive constant c and n_0

$$0 \leq f(n) \leq c\, g(n)$$
$$\Rightarrow 0 \leq 3n + 4 \leq c\, n$$

Now if we take n = 1 and c = 7

$$\Rightarrow 0 \leq 3 \times 1 + 4 \leq 7 \times 1$$

Which is true, so we can say that for $n_0 = 1$ and c = 7

f(n) = O (g(n)) that is
3n+4 = O(n)

Example 2. Consider the following algorithm to addtwo square matrix.

Statement	**Cost**	**Frequency**	**Total cost**
Mat_add(a[],n,b[])			
Step 1. i = 0	1	1	1
Step 2. j = 0;	1	1	1
Step 3.while i < n	1	n+1	n+1
Step 4.while j < n	1	n(n+1)	n(n+1)
Step 5.c[i][j] = a[i][j] + b[i][j]	1	n*n	n*n
Step 6. j = j + 1	1	n*n	n*n
Step 7.end inner while;	0	n	0
Step 8. i = I + 1	1	n	n
Step 9.end outer while	0	1	0
Step 10.return c;	1	1	1
Total Cost			$3n^2 + 3n + 4$

Here $f(n) = 3n^2 + 3n + 4$

Let, $g(n) = n^2$

If we want to represent it in Ω notation then we have to show that for some positive constant c and n_0

$$0 \leq c\, g(n) \leq f(n)$$

$$\Rightarrow 0 \leq c\, n^2 \leq 3n^2 + 3n + 4$$

Now if we take n = 1 and c = 3

$$\Rightarrow 0 \leq 3 \times 1 \leq 3 \times 1^2 + 3 \times 1 + 4$$

Which is true, so we can say that for $n_0 = 1$ and c = 3,

$$f(n) = \Omega(g(n))$$

i.e. $3n^2 + 3n + 4 = O(n^2)$

In analysis of algorithms three different cases may be considered depending on the input to the algorithms.These are,

Worst Case: This is the upper bound for execution time with anyinput(s). It guarantees that irrespective of the type of input(s), the algorithm will not take any longer than the worst case time.

Best Case: This is the lower bound for execution time with any input(s). It guarantees that under any circumstances,the algorithm will beexecuted at leastfor best case time. That is the minimum time required by the algorithm to execute for any input.

Average case: This is the execution time taken by thealgorithm forany random input(s)to the algorithm. In this case, for the inputs, the algorithm takes a time which is in between the upper and lower bound.

Example 2. Consider the following Insertion sort algorithm

Algorithm Insertion_Sort (a[n])

```
Step 1: i = 2
Step 2: while i < n
Step 3: num = a[i]
Step 4: j = i
Step 5: while (( j>1) && (a[j-1] >num))
Step 6: a[j] = a[j-1]
Step 7: j = j-1
Step 8: end while (inner)
Step 9: a[j] = num
Step 10: i = i + 1
Step 11: end while (outer)
```

Worst case Analysis of Insertion Sort

In worst case, inputs to the algorithm will be reversely sorted.Sothe loop statementswill run for maximum time. In worst case, every time we will find a[j-1]>num in statement 5 as true, so statement 5 will run for 2 + 3 + 4 + … + n times total n(n+1) - 1 times. Statement 6 will run for 1 + 2 + 3 + … + n-1 times total n(n-1) times. Same time as statement 6 will be taken by statement 7.

Statement	Cost	Frequency	Total cost
Step 1	1	1	1
Step 2	1	n	n
Step 3	1	n-1	n-1
Step 4	1	n-1	n-1
Step 5	1	n(n+1)-1	n(n+1)-1
Step 6	1	n(n-1)	n(n-1)
Step 7	1	n(n-1)	n(n-1)
Step 8	0	n-1	0
Step 9	1	n-1	n-1
Step 10	1	n-1	n-1
Step 11	0	1	0
Total Cost			$3n^2 + 4n - 4$

Here $f(n) = 3n^2 + 4n - 4$
Let, $g(n) = n^2$

If we want to represent it in O notation then we have to show that for some positive constant c and n_0, the following must be true,

$$0 \leq f(n) \leq c\, g(n)$$

$\Rightarrow 0 \leq 3n^2 + 4n - 4 \leq c\, n^2$

Now if we take n = 1 and c = 7

$\Rightarrow 0 \leq 3x1^2 + 4x1 - 4 \leq 7 \times 1^2$

Which is true.So we can say that for $n_0 = 1$ and c = 7

f(n) = O (g(n))

i.e$3n^2 + 4n - 4 = O(n^2)$

The worst case time complexity of insertion sort is $O(n^2)$.

Average case Analysis of Insertion Sort

In Average case, inputs to the algorithm will be random. Here, half of the time we will find a[j-1]>num is true and false in other half. So statement 5 will run for (2 + 3 + 4 + … + n)/2 times total (n(n+1)–1)/2 times. Statement 6 will run for (1 + 2 + 3 + … + n-1)/2 times total (n(n-1))/2 times. Same for statement 7 as statement 6.

Statement	Cost	Frequency	Total cost
Step 1	1	1	1
Step 2	1	n	n
Step 3	1	n-1	n-1
Step 4	1	n-1	n-1
Step 5	1	(n(n+1)-1)/2	(n(n+1)-1)/2
Step 6	1	(n(n-1))/2	(n(n-1))/2
Step 7	1	(n(n-1))/2	(n(n-1))/2
Step 8	0	n-1	0
Step 9	1	n-1	n-1
Step 10	1	n-1	n-1
Step 11	0	1	0
Total Cost			$^3/_2n^2 + {}^7/_2n - 4$

Now, $^3/_2n^2 + {}^7/_2n - 4 = O(n^2)$

The average case time complexity of insertion sort is $O(n^2)$.

Best case Analysis of Insertion Sort

In best case, inputs will be already sorted. So a[j-1]>num will be false always. Statement 5 will run for n times (only to check the condition is false). Statement 6 will run for 0 times since while loop will be false always. Statement 7 will also run for same times as statement 6.

Statement	Cost	Frequency	Total cost
Step 1	1	1	1
Step 2	1	n	n
Step 3	1	n-1	n-1
Step 4	1	n-1	n-1
Step 5	1	n	n
Step 6	1	0	0
Step 7	1	0	0
Step 8	0	n-1	0
Step 9	1	n-1	n-1
Step 10	1	n-1	n-1
Step 11	0	1	0
Total Cost			5n- 3

Now 5n-3 = O(n)

The best case time complexity of insertion sort is O(n)

CHECK YOUR PROGRESS

2. State True or False.

a) $7\ n^3 + 4n + 27 = O\ (n^3)$

b) $2n^2 + 34 = \Omega\ (n^3)$

c) $2n^2 + 34 = O\ (n^3)$

d) $2n^2 + 34 = \Theta\ (n^3)$

e) $2n^2 + 34 = \Omega\ (n)$

f) $2n^2 + 34 = \Theta\ (n^2)$

g) $2n^7 + 4n^3 + 2n = \Omega\ (n^3)$

h) $2n^4 + 3n^3 + 17n^2 = O\ (n^3)$

1.7 HEAPS AND HEAP SORT

A heap is a complete binary tree, which follows either the max-heap or min-heap properties. If the children(s) of every node have value less than the value of its parent node, then the heap is called max-heap.On the other hand if the value of children(s) of every node are greater than the value of its parent node, then the heap is called min-heap.

Forthese two cases, the root will always have either the highest or lowest value of the heap. For further discussion we will consider only the max-heap.

Heap Representation:

- A Heap can be efficiently represented in an array.
- The root is stored at the first place, i.e. a[1].
- The children of the node *i* are located at 2**i* and 2**i* +1.
- Theparent of a node stored in i^{th} location is at floor (i/2).

The array representation of a heap is given in the figure below.

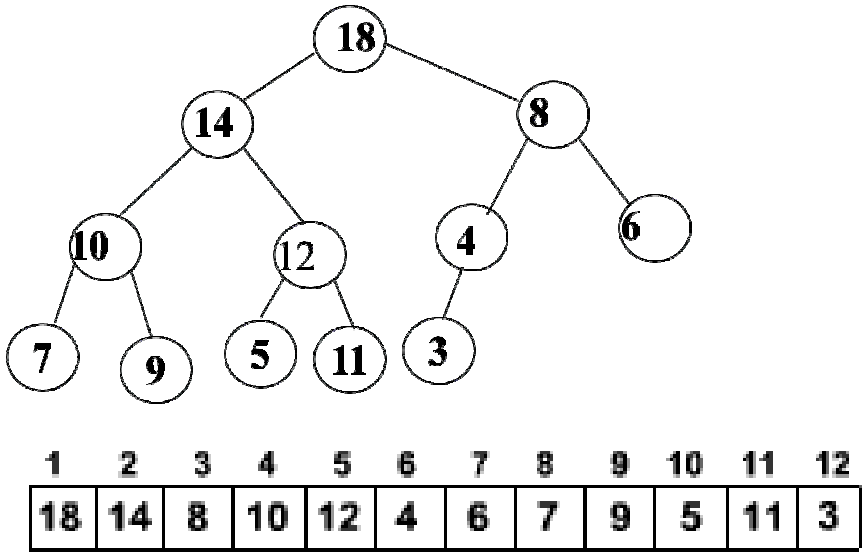


Fig 1.4 A Max Heap

Insertion:

To insert an element *x* into the heap, we first place the data into the next available location of the heap, so that it is still a complete binary tree but not necessarily a heap. If *x* can be placed in that position without violating heap property, then we are done with the insertion. Otherwise we have to exchange the element with its parent.We need to continue this process until *x* can be placed in the right position. Figure 1.5 to 1.7 shows different steps involving in insertion of 16 in the heap shown in fig 1.4.

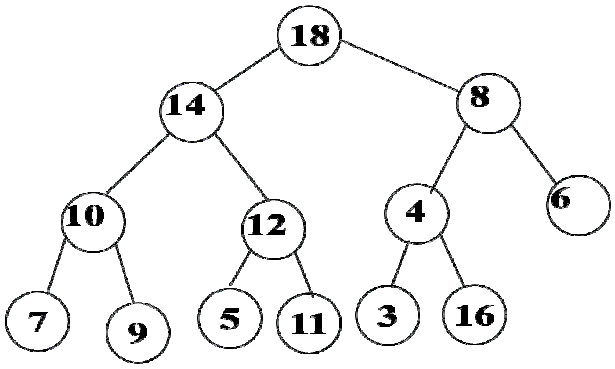


Fig 1.5 Adding 16 to the previous heap.

After adding the node 16 in to the next available location of the heap shown in fig 1.4, the new tree will be as shown in Fig 1.5.Since it follows all the properties of complete binary tree, but it does notsatisfy the properties of max heap since the parent of 16 is 4. To make it a heap we need to exchange the newly inserted element 16 with its parent. Fig 1.6 shows the new tree after exchange of 16 and 4

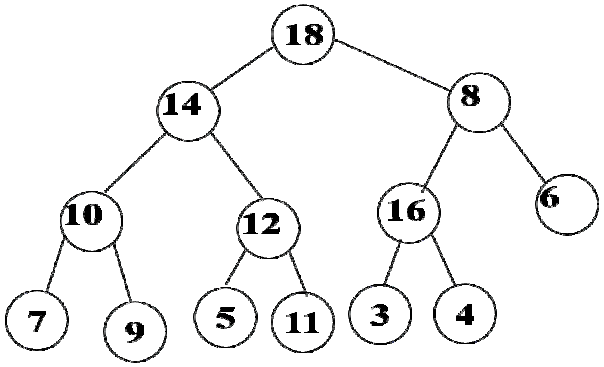


Fig 1.6 Steps of Insertion

Now after exchange of 16 and 4, it is still not a heap since the parent of 16 is 8, which violate the max heap property. To make it a heap we need to exchange 16 with its parent 8. Fig 1.7 shows the new tree after exchange of 16 and 8.

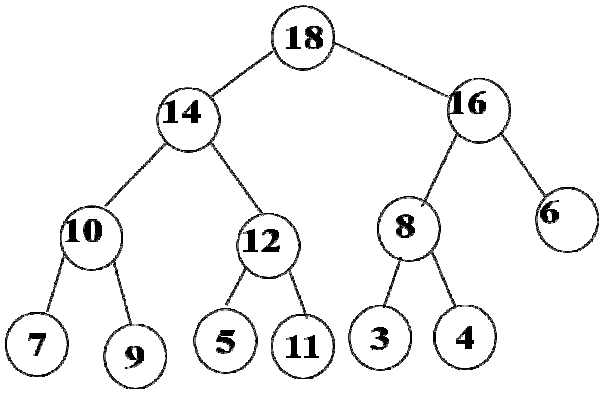


Fig 1.7 Heap after inserting 16

Now after exchange of 16 and 8 it follows the property of max heap. So Fig 1.7 is the heap after inserting 16 into the heap at fig 1.4.

Algorithm Heap_Insertion (arr[], item,N)

```
Step 1:  set N=N+1 and ptr=N;
Step 2:  while ptr ≥ 1
Step 3:     par = floor (prt/2)
Step 4:     if item ≤ arr[par]
Step 5:           arr[ptr]=item and return;
Step 6:     end if
Step 7:     arr[ptr]=arr[par]
Step 8:     ptr=par
Step 9:   end while
Step 10: arr[1]=item
Step 11: return
```

Deletion:

In the delete process of a heap, we only delete the root element from the heap. In this process, we first delete the root element and replace it by the last element of the heap, so that it is still a complete binary tree but not necessarily a heap. After replacing the root by the last element, if it maintains the heap properties, then we are done with the deletion. Otherwise we exchange the element with its children.Incase of Max-heap we exchange the element with the child having maximum value and in case of Min-heap we exchange the element with the child having minimum value, we have to continue this process until the element can be placed in the right position of the heap. Figure 1.8 to 1.10 shows different steps involving in deletionin the heap in fig 1.7.

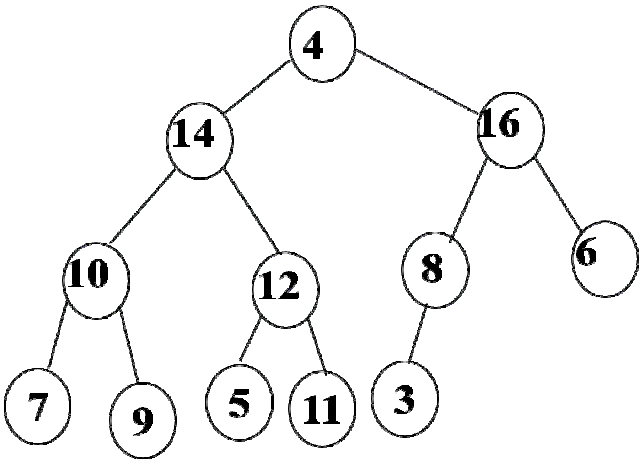


Fig 1.8After replacing root by the last element

After replacing root by the last element in the heap at fig 1.7,we have a complete binary treeas shown in fig 1.8.But it is not a heap, since the children of 4 violates the max heap property. To make it a heap we need to exchange the 4 with the child having maximum value, which is 16. Fig 1.9 shows the tree after exchange of 4 and 16

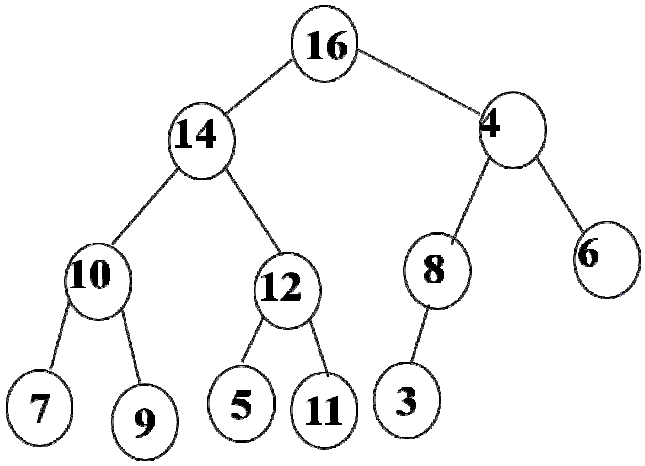


Fig 1.9Intermediate steps of deletion

Now after exchange of 16 and 4 it is still not a heap since children of 4 violate the max heap property. To make it a heap we need to exchange 4 with its child having maximum value, which is 8. Fig 1.10 shows tree after exchange 4 and 8.

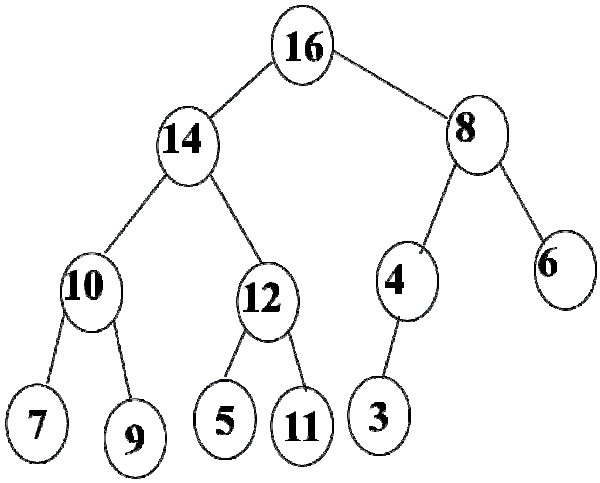


Fig 1.10 Heap after deletion

Now after exchange of 4 and 8 it follows the property of max heap. Fig 1.10 is the heap after deletion.

Algorithm Heap_Deletion (arr[], item, N)

```
Step 1:  item= arr[1]
Step 2:  last = arr[N]
Step 3:  ptr=1, left = 2 and right =3
Step 4:  while right ≤ N
Step 5:       if last ≥ arr[left]and last ≥ arr[right]
Step 6:            arr[ptr] = last and return
Step 7:       end if
Step 8:       if arr[right] ≤ arr[left]
Step 9:            arr[ptr]= arr[left] and ptr = left
Step 10:     else
Step 11:           arr[ptr] = arr[right] and ptr = right
Step 12:     end if
Step 13:     left=2*ptr and right= left+1
Step 14:  end while
Step 15:  if left==N and if last <arr[left]
Step 16:     ptr=left
Step 17:  end if
Step 18:  arr[ptr]= last
Step 19:  return
```

Heap Sort

Suppose an array A with N element is given. The heap sort algorithm consist of two phases to sort A

Phase 1: Build heap from the elements of A

Phase 2: Repeatedly delete the root from the heap

Algorithm Heap_Sort (arr[], N)

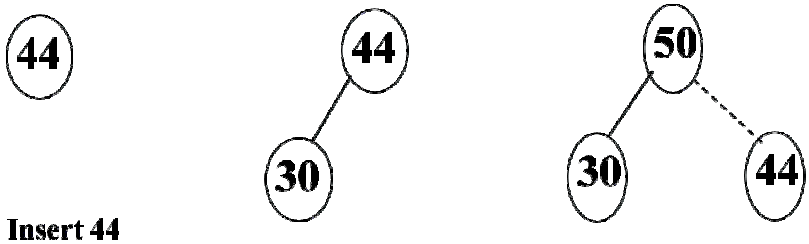
```
Step 1:  for i=1 to N-1
Step 2:      call Heap_insert(arr, arr[i+1], i)
Step 3:  end for
Step 4:  while N>1
Step 5:       call Heap_delete(arr, item , N)
Step 6:       arr[N+1]=item
Step 7:   end while
Step 8:   EXIT
```

Example

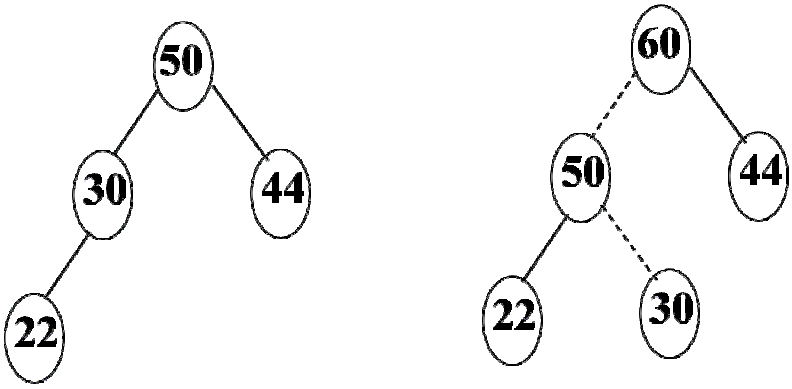
Suppose we want to sort the following elements by using heap sort

44, 30, 50, 22, 60, 55, 77

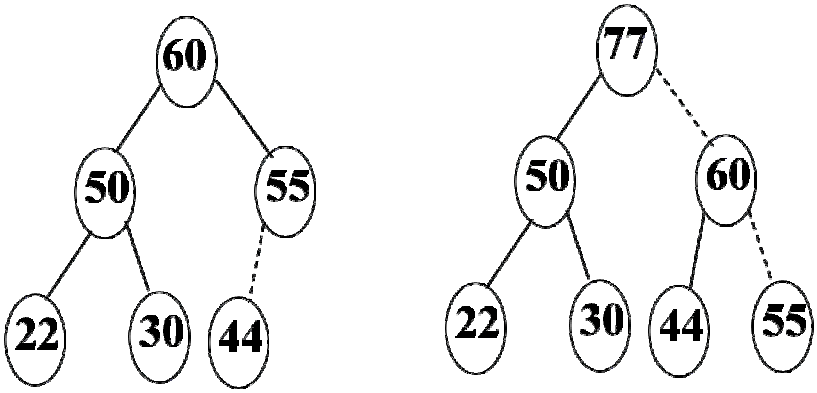
This can be accomplished by, first inserting the elements in to a heap one after another and then delete the root repeatedly until the heap is not empty.



a) Insert 44 **b) Insert 30** **c) Insert 50**

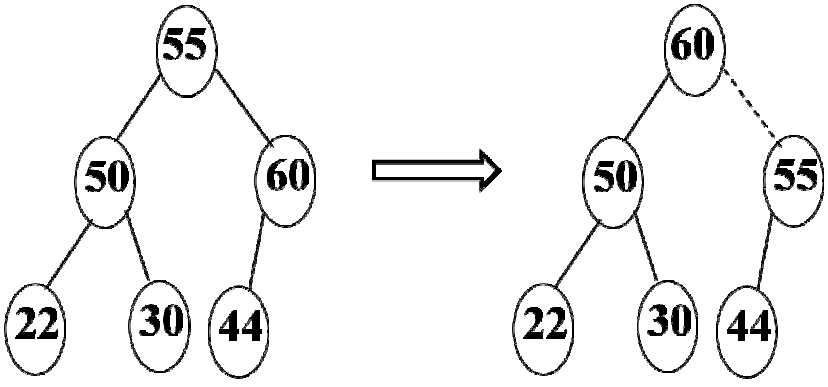


d) Insert 22 **e) Insert 60**

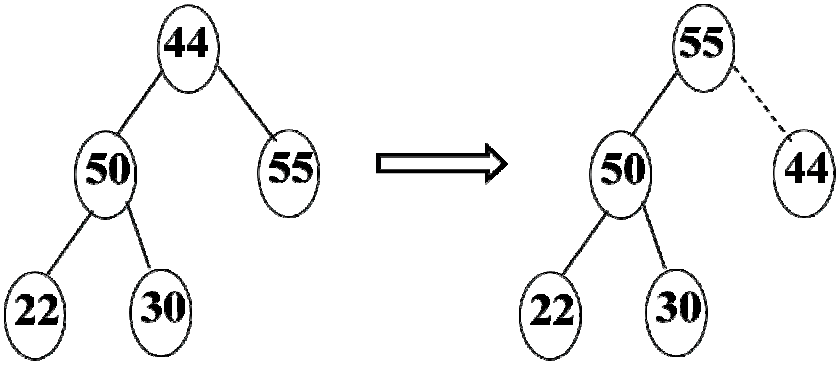


f) Insert 55 **g) Insert 77**

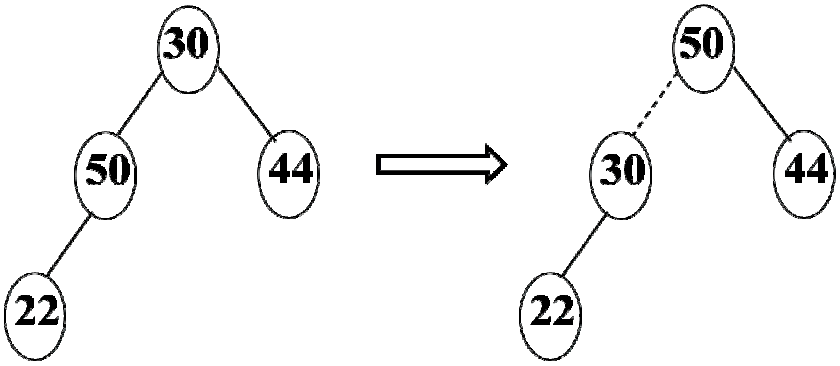
Heap insertion is done, now we will delete the root repeatedly until the heap become empty.



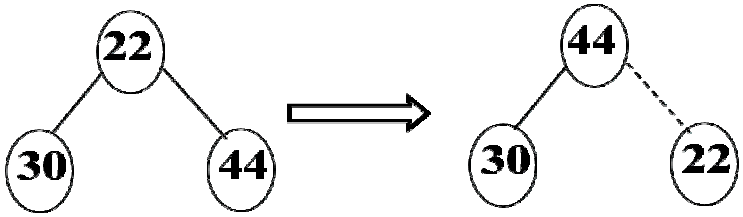
Output : 77 **Output : 77**



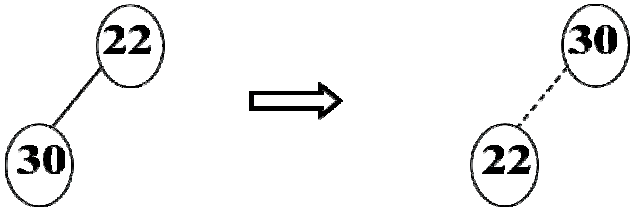
Output : 77, 60



Output : 77, 60, 55



Output : 77, 60, 55, 50



Output : 77, 60, 55, 50, 44

Output : 77, 60, 55, 50, 44, 30

NULL

Output : 77, 60, 55, 50, 44, 30, 22

Fig 1.11 Heap Sort

1.8 SET AND DISJOINT SET

In this section we will study the representation ofsets as forests. Here we assume that the elements of sets are the numbers 1,2,3, ... n, and we also assume that sets represented here are pair wise disjoint. For example if n=7, then the elements can be partitioned into two disjoint sets S_1 = {1,4,7} and S_2 = {2,3,5,6}. Fig 1.12 shows one possible representation of these sets. Here the usual method for representing child- parent relationship is not used, instead the links are maintain from child to parent.

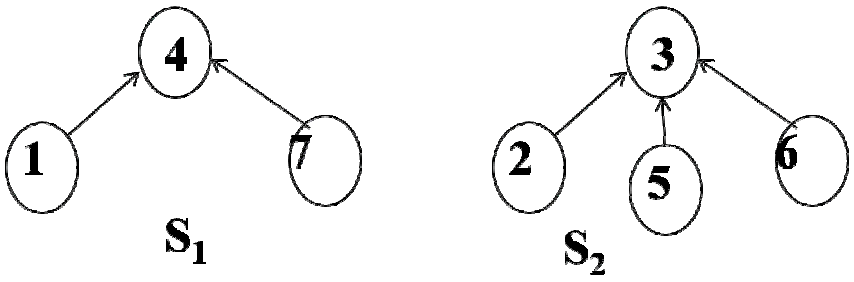


Fig 1.12 Possible tree representation of sets

1.9 UNION FIND ALGORITHM

Disjoint set Union: If $S_i$and S_j are two disjoint sets then their union S_iU S_j= all elements x such that x is in S_i or S_j.Thus S_1U S_2 = {1,2,3,4,5,6,7}. After union of any two sets $S_i$and S_j, the sets $S_i$and S_j do not exist independently anymore.They are replaced by S_iU S_j as a collection of sets.

Find(i): Given the element i, find the set containing i,eg. 4 isin S_1.

To obtain the union of two sets , all we need to do is, set the pointer of one of the roots to point the other root.For example,in fig 1.13 two possible representation of S_1U S_2 are shown.

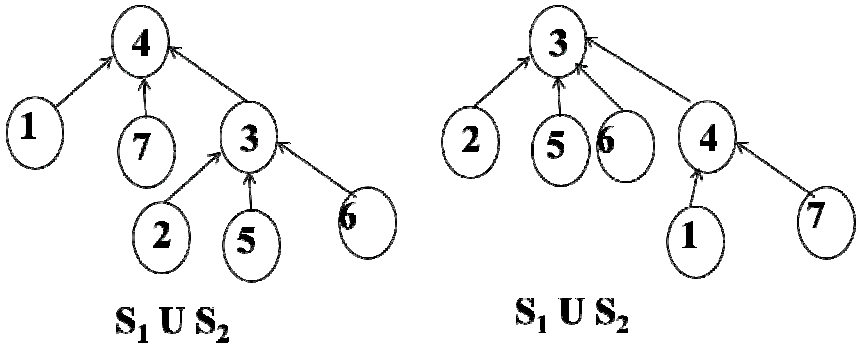


Fig: 1.13 Possible Representation of S_1U S_2

In computer representation of sets each set has a name with the elements of the set. In link representation of sets a pointer is maintained to point the root of the tree representing the set, and in addition each root also maintain a pointer to the set name.To

determine which set an element is currently in, we follow the parent links to the root of its tree and use the pointer to find the set name. In fig 1.14 linkrepresentation of $S_1$and S_2 is shown.

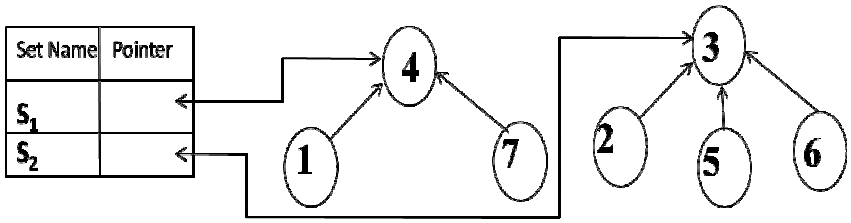


Fig 1.14 Data representation for S_1 and S_2

Since the set elements are numbered 1 through n, we can represent, the tree nodes using an array p [1 : n], where n is the maximum number of elements. The i^{th} element of this array represents the tree node that contains element i. The array elements give the parent pointer of the corresponding tree node. Fig 1.15 shows representation of sets S_1 and S_2, where the root node have parent -1.

I	[1]	[2]	[3]	[4]	[5]	[6]	[7]
P	4	3	-1	-1	3	3	4

Fig: 1.15 Array representation of S_1 and S_2

Now we can implement *Find(i),* by following the indices, starting at i until we reach a node with parent value -1. For example Find(6) starts at 6 and then moves to 6's parent 3. Since p[3] is –ve , we have reached the root. The operation Union(i, j) is also equally simple, we pass the two trees root i and j, by adopting the convention first tree become the sub-tree of the second, the statement p[i]=j; accomplished the union.

1.10 SORTING IN LINEAR TIME

We have already discussed several sorting algorithms which can sort data in O(n log n) time,merge sort and heapsort achieve this upper bound in the worst case; quicksort achieves it on average. Moreover, for each of these algorithms, we can produce a sequence of n inputs that causes the algorithm to run in Ω(n log n) time.

These algorithms share an interesting property, that is, the sorted order defined is based on comparisons between the input elements. We call such sorting algorithms comparison sorts. All the sorting algorithms introduced this far are comparison sorts.

Now we will discuss two sorting algorithms-counting sort and radix sort, that run in linear time. Needless to say, these algorithms use operations other than comparisons to determine the sorted order

Counting Sort

Counting sort assumes that each of the n input elements is an integer in the range 1 to k, for some integer k. When k = O(n), the sort runs in O(n) time.

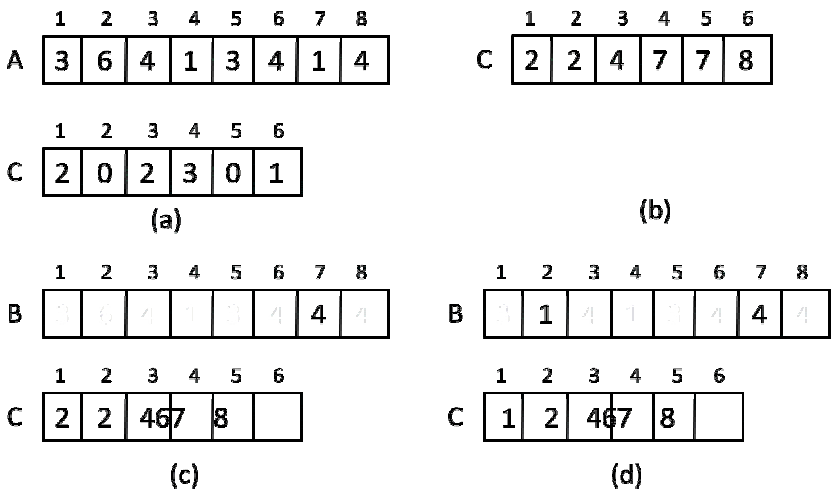
The basic idea of counting sort is to determine the number of elements less than each of the input element. This information can be used to place element x directly into its position in the output array. For example, if there are 17 elements less than x, then x belongs to output position 18. This scheme need to modified slightly to handle the situation when several elements have the same value, since we don't want to put them all in the same position.

In the algorithm of counting sort, we assume that the input is an array A[1 . . n], and length[A] = n. We require two other arrays: the array B[1 . . n] holds the sorted output, and the array C[1 . . k] provides temporary working space.

Algorithm Counting-Sort (A, B, k)

```
1.  for i = 1 to k
2.      do C[i] = 0
3.  for j = 1 to length[A]
4.      do C[A[j]] = C[A[j]] +1
5.  // C[i] now contains no of element equal to i
6.  for j = 2 to k
7.      do C[i] = C[i] +C[i-1]
8.
9.  for j = length[A] down to 1
10.     do B[C[A[j]]] = A[j]
11.         [A[j]] = C[A[j]] -1
```

An example of counting sort is shown below:



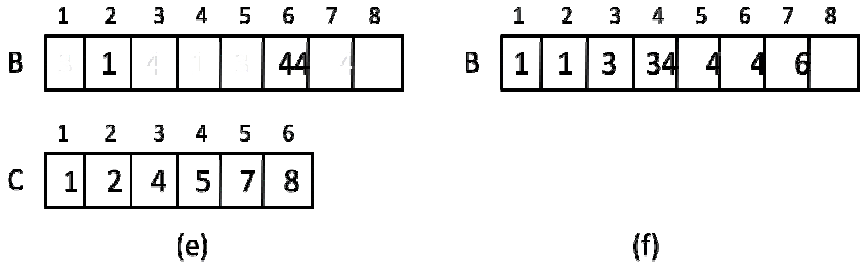


Fig: 1.16 counting sort

In Fig 1.16,the operation of COUNTING-SORT of an input array A[1 . . 8] is shown, where each element of A is a positive integer no larger than k = 6. In fig 1.16 (a) the array A and the auxiliary array C after execution of step 4 is shown. In fig 1.16 (b) the array C after execution of step 7 is shown. In fig 1.16(c)-(e) the output array B and the auxiliary array C after one, two, and three iterations of the loop in steps 9-11, respectively. In fig 1.16 (f) the final sorted output array B is shown.

How much time does counting sort require? The for loop of steps 1-2 takes time O(k), the for loop of steps 3-4 takes time O(n), the for loop of steps 6-7 takes time O(k), and the for loop of steps 9-11 takes time O(n). Thus, the overall time is O(k + n). In practice, we usually use counting sort when we have k = O(n), in which case the running time is O(n). So counting sort beats the lower bound of Ω(n 1g n).

Radix Sort

Radix sort is a small method that many people intuitively use when alphabetizing a large list of names. (In case of names Radix is 26, since there are 26 letters in the alphabet). Specifically, the list of names is first sorted according to the first letter of each names, that is, the names are arranged in 26 classes. Intuitively, one might want to sort numbers on their most significant digit. But Radix sort do counter-intuitively by sorting on the least significant digits first. On the first pass entire numbers are sorted on the basis of least significant digit and iscombined in an array. Then on the second pass, the entire numbers are sorted again based on the second least-significant digits and is combined in an array and so on.

RADIX-SORT(A, d)

1. for i ← 1 to d
2. do use a stable sort to sort array A on digit i

The code for radix sort is straightforward. The following procedure assumes that each element in the n-element array A has d digits, where digit 1 is the lowest-order digit and digit d is the highest-order digit.

The analysis of the running time depends on the stable sort used as the intermediate sorting algorithm. When each digit is in the range 1 to k, and k is not too large, counting sort is the obvious choice. Each pass over n d-digit numbers takes time θ(n + k). There are d passes, so the total time for radix sort is θ(dn + kd). When d is constant and k = O(n), radix sort runs in linear time.

Following example shows how Radix sort operates on seven 3-digits number :

Input: 329, 457, 657, 839, 436, 720, 355

0	720		
1			
2		720, 329	
3		436, 839	329, 355
4			436, 457
5	355	355, 457, 657	
6	436		657
7	457, 657		720
8			839
9	329, 839		

Output: 329, 355, 436, 457, 657, 750, 839

Fig 1.17 Radix sort example.

Figure 1.17shows the operation of radix sort on a list of seven 3-digit numbers.

CHECK YOUR PROGRESS

3. State True or False.

a) The find operation can be performed even if the sets are not disjoint.

b) Counting sort is efficient when the range of numbers to be sort is small.

c) In Max-Heap root always contain the minimum value of the heap.

1.11 LET US SUM UP

- An algorithm is a sequence of computational steps that start with a set of input(s) and finish with valid output(s)
- An algorithm is correct if for every input(s), it halts with correct output(s).
- Computational complexity of algorithms are generally referred to by space complexity and time complexity of the program
- The Space complexity of an algorithm is the amount of main memory is needed to run the program till completion.
- The Time complexity of an algorithm is the amount of computer time it needs to run the program till completion.
- $O(1) < O(\log(n)) < O(n) < O(n \log(n)) < O(n^2) < O(n^3) \ldots < O(2^n)$.
- Heap is a complete binary tree with the properties of either Max heap or Min heap
- Union operation on set combine two set by making one of the root as the child of the other root.
- Find operation on set returns the set-name of the set where the node belongs.
- Radix sort and counting sort are linear time sorting algorithm

1.12 FURTHER READINGS

1. T. H. Cormen, C. E. Leiserson, R.L.Rivest, and C. Stein, "Introduction to Algorithms", Second Edition, Prentice Hall of India Pvt. Ltd, 2006.
2. Ellis Horowitz, SartajSahni and SanguthevarRajasekaran, Fundamental of data structure in C, Second Edition, Universities Press, 2009.
3. Alfred V. Aho,John E. Hopcroft and Jeffrey D. Ullman, "The Design and Analysis of Computer Algorithms", Pearson Education, 1999.
4. Ellis Horowitz, SartajSahni and SanguthevarRajasekaran, Computer Algorithms/ C++, Second Edition, Universities Press, 2007.

.

1.13 ANSWERS TO CHECK YOUR PROGRESS

1.

a) False b)True c)False d)False e)True

2.

a) True b)False c)True d)False e) True
f) True g) True h) False

3.

a) False b)True c)False

1.14 MODEL QUESTIONS

1. Given an array of n integers, write an algorithm to find the smallest element. Find number of instruction executed by your algorithm. What are the time and space complexities?
2. Write a algorithm to find the median of n numbers. Find number of instruction executed by your algorithm. What are the time and space complexities?
3. Write an algorithm to sort elements by bubble sort algorithm. What are the time and space complexities?
4. Explain the need of Analysis of Algorithm.
5. Prove the following

   ii) $3n^5 - 7n + 4 = \Theta(n^5)$
   iii) $^1/_3n^4 - 7n^2 + 3n = \Theta(n^4)$
   iv) $2n^2 + n + 4 = \Theta(n^2)$
   v) $3n^5 - 7n + 4 = O(n^5)$
   vi) $3n^5 - 7n + 4 = \Omega(n^5)$

6. Sort the following element by using Heap sort algorithm

   17, 19,13, 16,12, 9, 14, 18, 6 15, 22, 27, 8

7. Sort the following elements by using counting sort algorithm

   7, 9, 9, 6, 4, 2, 8, 6, 4, 3, 7, 2, 1

8. Sort the following element by using Radix sort algorithm

   177, 129,153, 196,122, 339, 514, 188, 666 245, 292, 207

*****

UNIT- 2 : DIVIDE AND CONQUER

UNIT STRUCTURE

2.1 LEARNING OBJECTIVE

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

- know the concept divide and conquer
- describe the application of divide and conquer method in binary search, quick sort and merge sort techniques
- elaborate the application of divide and conquer technique in exponentiation

2.2 INTRODUCTION

In the previous unit, you are acquainted with the basic idea about the algorithms, time complexity of algorithms and some other related issues. In this unit, we will introduce you the ***'divide and***

conquer' approach that is used in the design of algorithms. This technique is the basis of designing efficient algorithms for all kinds of problems, such as sorting techniques like quick sort, merge sort and in searching techniques like binary search etc.

2.3 DIVIDE AND CONQUER GENERALSTRATEGY

Divide and conquer algorithm is an important algorithm designed paradigm based on multi-branched recursion. A divide and conquer algorithm works by recursively breaking down a problem into two or more independent sub-problems of the same type, until they become simple enough to solved directly. Generally, the sub-problems solved by a divide and conquer is non-overlapping i.e solution to a problem is depend upon only on sub-problems, but is not depend upon sub-sub-problems.

The general methodology applied in the divide and conquer technique is as follows :

Step 1: Divide the problem into two or more independent sub-Problems (not necessarily same type).

Step 2: Solve (conquer) the each sub-problem recursively to the Smallest possible size.

Step 3: Combine these solution of the sub-problems into a solution to the whole problem.

2.4 DIVIDE AND CONQUER ALGORITHM APPLIED IN BINARY SEARCH

Binary search is a well known instance of divide and conquer method. For binary search divide and conquer strategy is applied recursively for a given sorted array is as follows:

Divide: Divide the selected array at the middle. It creates two

sub-array, one left sub-array and other right sub-array.

Conquer: Find out the appropriate sub-array.

Combine: Check for the solution to key element.

For a given sorted array of N element and for a given key element (value to be searched in the sorted array), the basic idea of binary search is as fallows –

1. First find the middle element of the array

2. Compare the middle element with the key element.

3. There are three cases

- If it is the key element then search is successful.
- If it is less than key element then search only the lower half of the array.
- If it is greater than key element then search only the upper half of the array.

4. Repeat 1, 2 and 3 until the key element found or sub-array sizes become one.

Algorithm for binary search

```
1.    Set Lower = 0, Upper = N -1
2.    Mid = ( Lower + Upper ) / 2
3.    while ( Lower ≤Upper) and A [ Mid ] != Item repeat
      Steps 4 and 5
4.    if ( Item < A [ Mid ] )  then
5.    Upper = Mid -1
6.    else   Lower = Mid +1
7.    Mid = ( Lower + Upper ) / 2
8.    if ( A [ Mid ] = = Item ) then
9.    Print  “Search successful”
10.   else   Print  “Item is not found”
11.   end
```

Here, **Lower, Upper and Mid** denotes the beginning, ending and middle index of an array A [ ] respectively. Item means the key element to be searched in the given array A [ ]. If size of A [ ] is N then beginning and the ending indices are 0 and N -1 respectively.

In ***step 1*** the algorithm initially sets the value of Lower = 0 and Upper = N -1.

In ***step 2*** it calculate Mid, the index of the middle element, for the array A [ ].

In ***step 3*** while the beginning index (Lower) is less then end index (Upper) and middle element (A [Mid]) is not equal key element (Item) then repeat step 4 and 5.

In ***step 4*** if A [Mid] is less than the Item, then the algorithm searches in the left sub-array. So, the beginning index remain same and the end index of the left sub-array becomes Mid -1. Hence, 'Upper' is set as Mid -1. Else if A [Mid] is greater than Item, then the algorithm searches in the right sub-array. So, the beginning index of the right sub-array becomes Mid+1 and end index remain same. Hence the 'Lower' is set as Mid+1.

In ***step 5*** again middle element (Mid) is calculated for the selected sub-array in step 4.

In ***step 6*** algorithm is terminated either if middle element (A[Mid]) is equal to Item or beginning index (Lower) is greater than end index (Upper)(i.e when subarray sizes become one) . In first case , terminate when search element is found and in later case terminate when search is not successful.

Example 2.5.1: Suppose A is an array of 6 elements. Search an element 10 in the array using binary search.

0	1	2	3	4	5
3	4	7	10	12	13

Solution:

Step 1: Here Lower = 0, Upper = 5, Item = 10

Step 2: First we have to calculate middle element for the array A

Mid = (Lower + Upper) / 2

= (0 + 5) / 2

=2

Mid *divides* the array into two subarray as follows

Lower= 0	1	Mid = 2	3	4	Upper=5
3	4	7	10	12	13

Step 3: Here *Lower < Upper* and *A [Mid] ! =10*. So *continue Step4 and Step 5*

Step 4: Select *(Conquer)* an appropriate subarray.

Here A [Mid] =7

7<10 , select *the right subarray* to search for the element.

Lower = Mid+1

=2 +1

=3

Now the subarray is-

Lower = 3	4	Upper = 5
10	12	13

Step 5: Mid = (Lower + Upper) / 2

= (3 + 5) / 2

= 4

Lower = 3	Mid = 4	Upper = 5
10	12	13

divide the array again in Mid = 4.

Here, *Lower < Upper* and *A [Mid] != 10* So**,** *repeat step 4 and Step 5* again

A [4] =12

12 >10, So search in the *left subarray.*

Upper = Mid -1

= 4 -1

=3

New subarray is-

Lower=3 Upper=3

10

Step 6: Mid = (Lower + Upper) / 2

= (3 + 3) / 2

= 3

Here, *Lower = Upper* and *A [Mid] = =10*

So, *PRINT "Search successful"*

Step 7: End

CHECK YOUR PROGRESS

1. Fill in the blanks

a) Binary search is applied in already----------array.

b) In binary search each time the algorithm finds out the --------

element.

c) The algorithm divide the array into two halves in ---------.

d) Divide and conquer algorithm is applied in a problem when sub-problems are------.

e) Divide and conquer algorithm divide the problem into -------- sub-problems.

2.5 DIVIDE AND CONQUER ALGORITHM APPLIED IN MERGE SORT

Merge sort is also one of the 'divide and conquer' class of algorithms. This is a sorting algorithm to sort an unordered list of element. Merge sort is a recursive algorithm that splits the array into two sub-arrays, sorts each sub-array, and then merges the two sorted arrays into a single sorted array. The base case of the recursion is when a subarray of size 1 (or 0). Merge sort algorithm also closely follow divide and conquer strategy. It is an external sorting algorithm.

Divide : Divide N element array to be sorted into two subarray of N / 2 element each.

Conquer : Sort the subarrays recursively using merge sort.

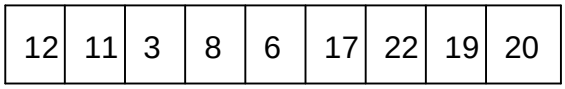
Combine : Merge the two sorted sub-array to produce final sorted array.

Suppose we have to sort a array of N element, A [p…..r]. Initially p = 1 and r=N

To sort A [p .. r]

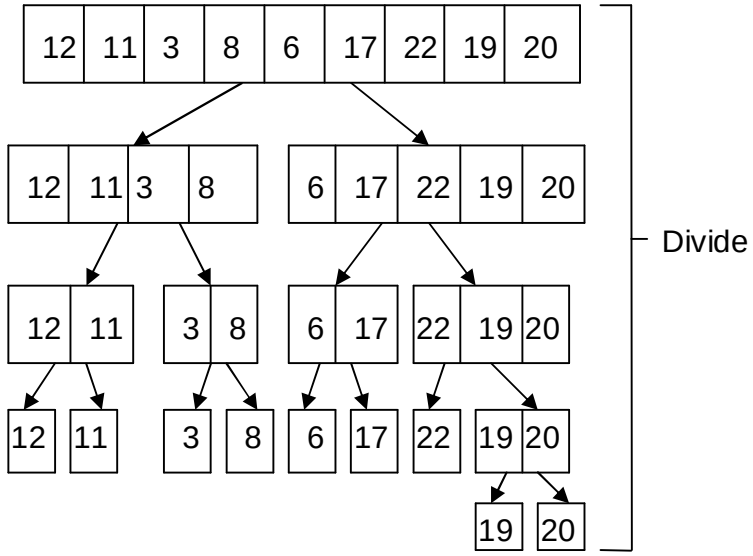
1. ***Divide Step:*** If a given array *A* has zero or one element, simply return; it is already sorted. Otherwise, split *A* [p .. r] into two subarrays *A* [p .. q] and *A* [q + 1 .. r], each containing about half of the elements of *A* [p .. r]. That is, *q* is the halfway point of *A* [p .. r].
2. ***Conquer Step*:** Conquer by recursively sorting the two subarrays *A* [p .. q] and A [q + 1 .. r].
3. ***Combine Step****:* Combine the elements back in A [p .. r] by merging the two sorted subarrays A [p .. q] and *A* [*q* + 1 .. r] into a unique sorted sequence.

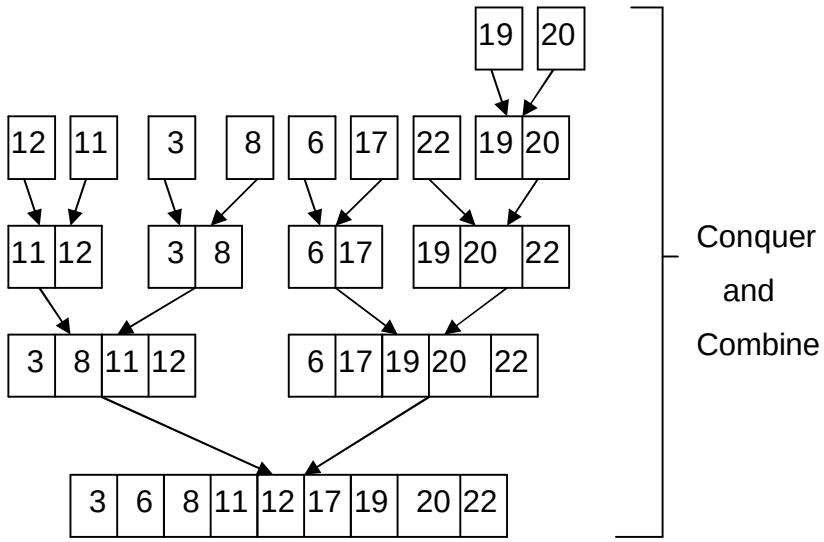
Example 2.6.1: Sort the following data using merge sort



Solution:

The merge sort strategy is applied as fallows-





This is the final sorted array.

2.7 PSEUDO CODE FOR MERGE SORT ALGORITHM

```
MERGE_SORT ( A , p , r)
  1. if ( p < r )
  2. then q = ( p + r ) / 2
```

```
  3.        MERGE_SORT ( A , p , q )
  4.        MERGE_SORT ( A , q + 1 , r )
  5.        MERGE ( A , p , q , r )
```

```
MERGE ( A , p , q , r )
  1.  n1 = q – p + 1
  2.  n2 = r - q
  3.  create arrays L [ 1…..n1 + 1 ]  and  R [ 1……n2 + 1 ]
  4.  for i=1 to n1
  5.        do   L [ i ] = A [ p + i – 1 ]
  6.  for  j=1 to n2
  7.        do   R [ j ] = A [ q + j ]
  8.  L [ n1 +1 ] = ∞
  9.R [ n2 + 1 ] = ∞
  10.  i =1
  11. j =1
  12. for k = p to r
   13.     do if L [ i  ] ≤ R [ i ]
   14.        then  A [ k ] = L [ i ]
   15.        i = i +1
   16.     else   A [ k ] = R [ j ]
   17.        j = j + 1
```

Here the procedure MERGE_SORT (A , p , r) sorts the element in the subarray A [p….r] . i.e p is the first element index and r is the last element index of the subarray. If p ≥ r , the subarray is atmost one element and is already sorted. Otherwise, the divide

step (step 2 in MERGE_SORT (A, p, r) procedure) simply computes an index q that partition A [p….r] into two subarray A [p …..q] and A [q + 1…..r] containing n/2 elements in each subarray

To sort a sequence A of N element , the initial call is MERGE_SORT(A,1, N).

Next, we have to merge the sorted subarrays obtain from the MERGE_SORT (A , p , r) procedure using MERGE(A , p , q , r), where A is an sorted array, p , q and r indices of the element such that p ≤ q < r. This procedure merge two sorted sub-array A[p..q] and A [q + 1….r] and form a single sorted subarray and replaces the current subarray A [p..r].

In details the MERGE procedure is work as follows-

Line 1 compute the length n_1 of the subarray A [p…q] .

Line 2 compute the length n_2 of the subarray A [q + 1…r].

Line 3 create array 'L' (left) and 'R' (right) of length n_1+1 and n_2+1 respectively.

Line 4-5 the for loop copies the subarray A [p…q] into L [1..n_1].

Line 6-7 it copies the subarray A [q + 1…r] into R [1..n_2].

Line 8-9 put '∞' at the end of the array L (i.e in n_1+1)and R (i.e in n_2 + 1).

Line12-17 find the smallest element between L[i] and R [j], where i =1….n_1 and j =1….n_2.

- If L [i] ≤ R [j] then it copies L [i] to A and increase i to i +1,
- Otherwise, copies R [j] to A and increase j to j +1.

Example:

Let us see, how merging is done between two sorted subarrays using the MERGE procedure. After merging is done the subarrays are combined to one sorted array.

In the following sequence the procedure calls MERGE(A,13,15,17) works as below .

Here p = 13
q = 15
r = 17
L= left subarray
R= right subarray
i = index of element's of left subarray L
j = index of element's of right subarray R

A k=13 14 15 16 17

....	10	11	14	9	13	…

L i=1 2 3 4

10	11	14	∞

R j=1 2 3

9	13	∞

Here, k = 13, i =1 and j = 1
R [1] < L [1] i.e 9 < 10.
So, A [1] = R [1]
= 9
j = j + 1 =1 + 1 = 2

A

	k=13	14	15	16	17	
…	9	11	14	9	13	…

Next, k = 14, i = 1, j = 2

L [1] < R [2] i.e 10 < 13

So, A [14] = L [1]

=10

i = i + 1 = 1 + 1 = 2

A

	13	k=14	15	16	17	
…	9	10	14	9	13	…

Next, k = 15, i = 2, j = 2

L [2] < R [2], i.e 11 < 13

So, A [15] = L [2]

=11

i = i + 1 = 2 + 1 = 3

A

	13	14	k=15	16	17	
…	9	10	11	9	13	…

Next, k = 16 , i = 3, j = 2

R [2] < L [3] , i.e 13 < 14

So, A [16] = R [2]

=13

j = j + 1 = 2 + 1 = 3

A	13	14	15	k = 16	17	
…	9	10	11	13	13	…

Next, k = 17, i = 3, j = 3

L [3] < R [3], i.e 14 < ∞

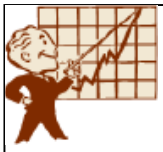
So, A [17] = L [3]

=14

i = i + 1 = 3 + 1 = 4

A	13	14	15	16	k = 17	
…	9	10	11	13	14	…

Next, k=18 which is greater then r(i.e 17). So the MERGE procedure is terminated here.



CHECK YOUR PROGRESS

2. Fill in the blanks

a. In divide step merge sort algorithm divides the array elements to be sorted up to array size becomes ---- or -----.

b. Merge sort algorithm merges two ----- subarrays.

c. For merging two subarrays merge sort algorithm uses another ----- subarray.

d. Merge sort algorithm is an-------sorting algorithm.

2.7 DIVIDE AND CONQUER ALGORITHM APPLIED IN QUICK SORT

It is one of the widely used internal sorting algorithm. In its basic form it was developed by C.A.R Hoare in 1960. The basis of quick

sort is divide and conquer strategy, i.e divide the problem (list to be sorted) into sub-problems (sub-lists) , until solved sub-problems (sorted sub-list) are found. The divide and conquer approach can be used in quick sort differently from merge sort. In merge sort , the list to be sorted is divided at its midpoint into subarrays which are independently sorted and later merged. In quick sort, the division to the sorted subarrays is made, so that the sorted subarrays do not need to merge later.

Divide and conquer strategy for quick sort to sort an array A [p…r] is as follows:

Divide: partition the array A [p…r] into two sub-arrays A[p…q -1] and A [q + 1…..r] such that each element of A [p…q -1] is less than or equal to A [q], which in turn, less than or equal to each element of A [q + 1…r]. Compute the index q as part of this partitioning procedure.

Conquer: Sort the two subarrays A[p…q-1] and A[q+1…r] by recursive calls to quick sort .

Combine: Since the subarrays are sorted in place, no work is needed to combine them. The entire array A [p…r] is now sorted.

2.7.1 ALGORITHM FOR QUICK SORT

Quick_Sort **:**

Step 1 : If *First* < *Last* then begin /* here *First* and *Last* are the index of the first and last

elements in the array*/

Step 2 : Partition the elements in the subarray *First*….*Last*

Step 3 : Apply ***Quick_Sort*** to the first subarray.

Step 4 : Apply ***Quick_Sort*** to the second subarray.

end

For this algorithm two stopping cases are-

- If *First* = *Last* . i.e. only one element in the subarray to be sorted.
- if *First* > *Last*. i.e. no element in the subarray to be sorted.

Partition algorithm**:**

To partition an array A[ ] partition algorithm is

Step 1: Define the *Pivot* value as the contents of the Array, A [*First*].

Step 2: Initialize *Up* to the *First* and *Down* to the *Last*

Step 3: Repeat step 4,5,6 until *Up* ≥ *Down*

Step 4: Increment Up until Up selects the first element greater than the *Pivot* value.

Step 5: Decrement *Down* until it selects the first element less than or equal to the *Pivot* value.

Step 6: If *Up* < *Down* exchange their values.

Step 7: Exchange A [*First*] and A [*Down*].

Step 8: Define *PivotIndex* as *Down*

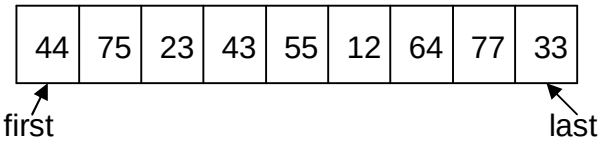
Example :

Sort the following data of array A using quick sort.

A = 44 75 23 43 55 12 64 77 33

Solution:

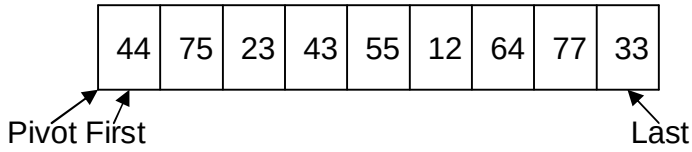
1. Assign *First* to first element and *Last* to last element.



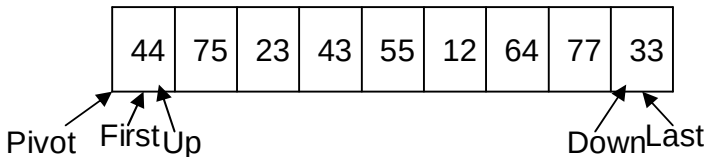
2. First < Last,

   So, assign Pivot to First .

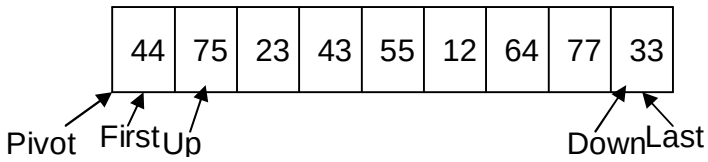
   Pivot = First



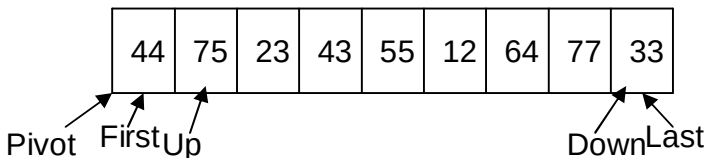
3. Assign Up to first element and Down to last element.



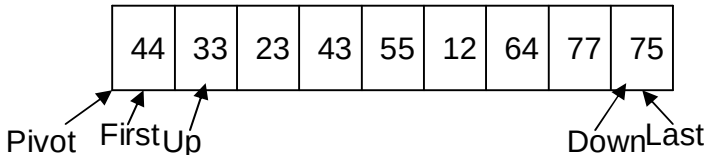
a) if (A [Pivot] ≥ A [Up]) then Up++



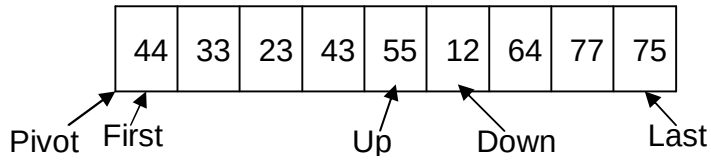
b) if (A [Pivot] < A [Down]) Down --

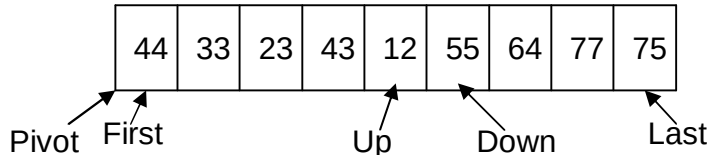


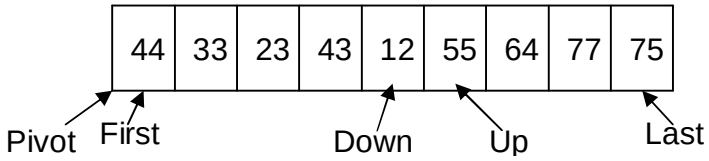
c) if (Up < Down) exchange A [Up] and A [Down]



d) Repeat a), b) and c) if Up < Down

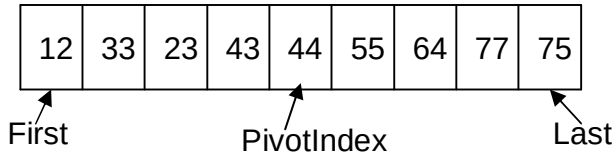




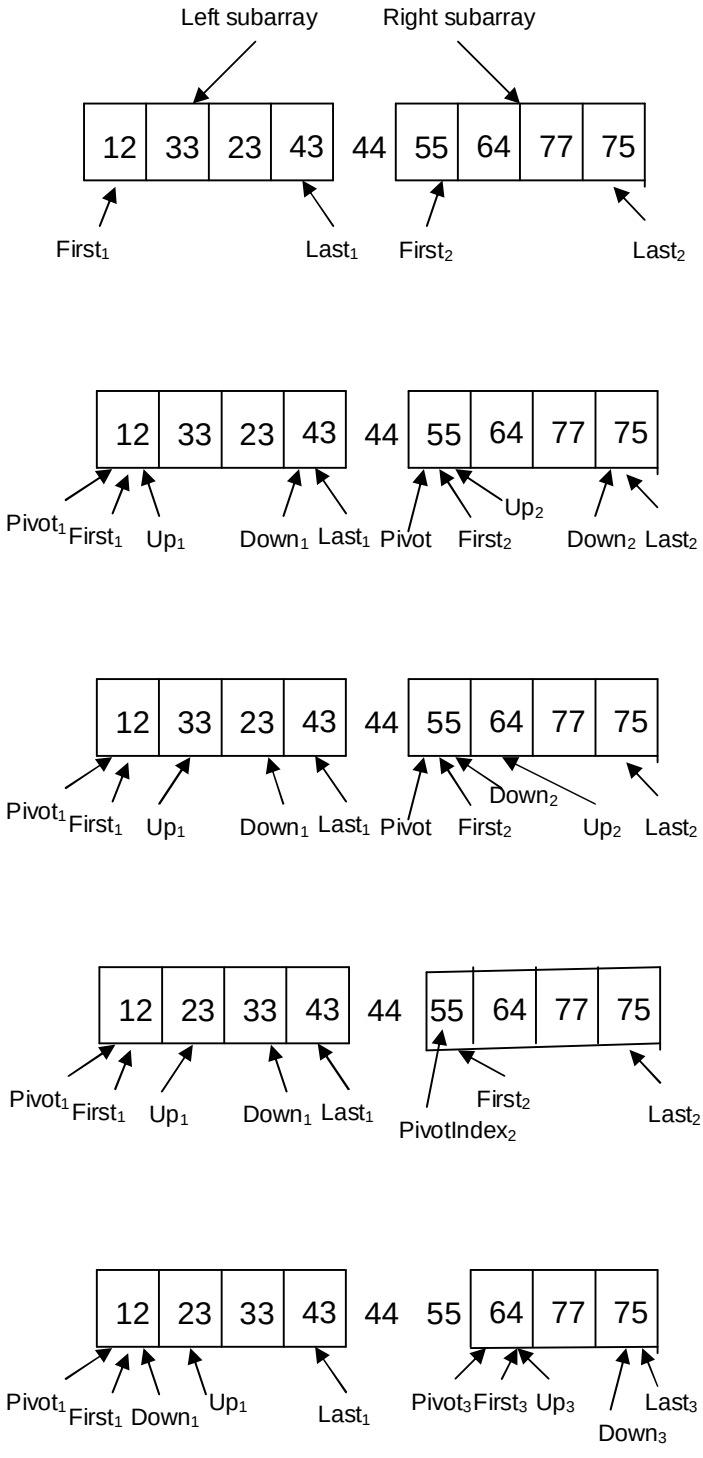


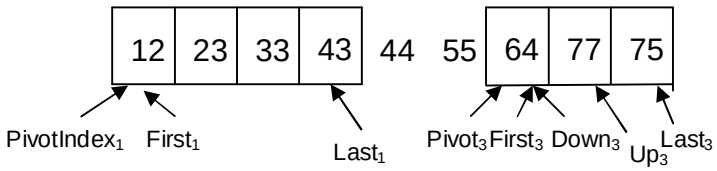
e) Here, Up > Down

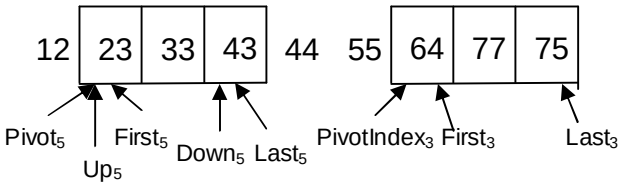
Exchange A [Pivot] and A [Down] and assign Down as PivotIndex

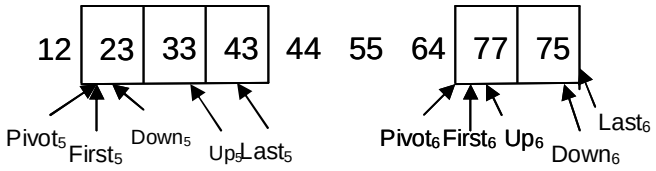


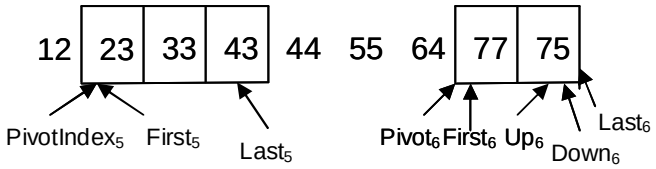
4. This gives two subarrays , Left subarray and Right subarray. Again we have to apply Quick Sort procedure in these sub arrays to sort it.

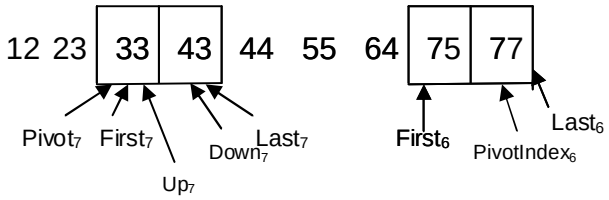


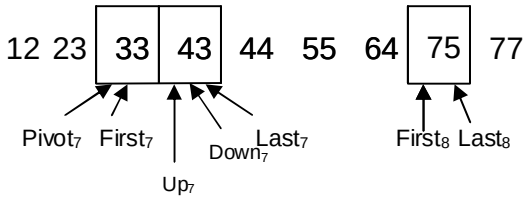

12 23 33 43 44 55 64 77 75
PivotIndex1 First1 Last1
Pivot3 First3 Down3 Up3 Last3

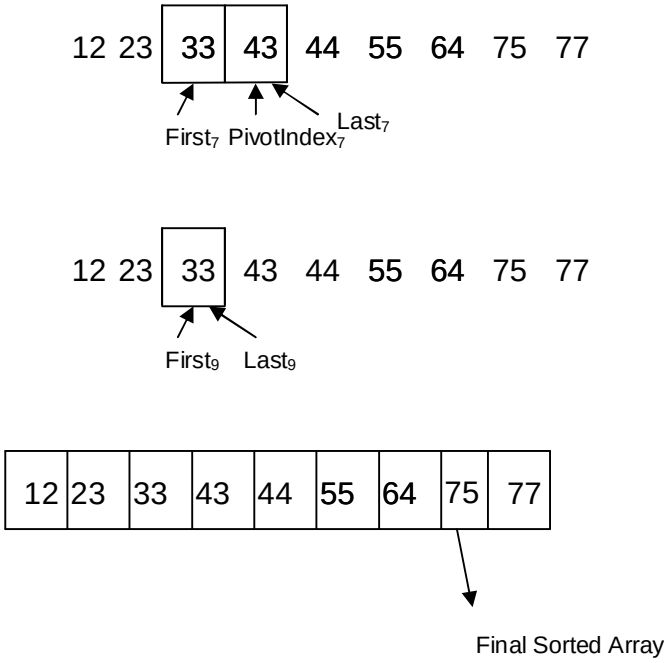

12 23 33 43 44 55 64 77 75
Pivot5 First5 Up5 Down5 Last5
PivotIndex3 First3 Last3


12 23 33 43 44 55 64 77 75
Pivot5 First5 Down5 Up5 Last5
Pivot6 First6 Up6 Down6 Last6


12 23 33 43 44 55 64 77 75
PivotIndex5 First5 Last5
Pivot6 First6 Up6 Down6 Last6


12 23 33 43 44 55 64 75 77
Pivot7 First7 Up7 Down7 Last7
First6 PivotIndex6 Last6


12 23 33 43 44 55 64 75 77
Pivot7 First7 Up7 Down7 Last7
First8 Last8



2.7.2 PSEUDO CODE FOR QUICK SORT

The pseudo code for the quick sort algorithm is given below :

```
        QUICKSORT ( A , p , r)
1.      if p < r
2.       then q = PARTITION ( A , p , r )
3.            QUICKSORT  ( A , p , q – 1 )
4.              QUICKSORT ( A ,q + 1 , r )
```

To sort an entire array A, the initial call is
QUICKSORT (A ,1, length [A])

Next, the PARTITIONING procedure which rearranges the subarray A [p….r] in place.

```
PARTITION (A , p , r)
```

```
1.      x = A [ r ]
2.      i = p -1
3.      for j = p to r -1
4.        do if A [ j ] ≤ x
5.             then i = i +1
6.                  exchange A [ i ] = A [ j ]
7.      exchange A [ i + 1 ] = A [ r ]
8.      return i + 1
```

CHECK YOUR PROGRESS

3. What is the type of quick sort algorithm? External or internal?
4. Why does not quick sort algorithm need to combine the sorted subarrays later?

2.8 EXPONENTIATION

Let a and n are two integers. Suppose that we need to compute a^n for some reasonably large n. For simplicity we can assume that n > 0.

The simplest algorithm perform n-1 multiplication by computing a x a x….x a.

Using divide and conquer strategy the problem can be solved by another way. We can consider as

n = n / 2 + n / 2

If n is even then $a^n = (a^{n/2})^2$

If n is odd then $a^n = a \times (a^{n/2})^2$

The function is as follows-

```
function Power ( a , n )
{
    if ( n==0)
       return 1;
    X = Power ( a , n/2 );
    if n is even then
      return ( X^2 );
    else
      return ( a x X^2 );
}
```

The above algorithm illustrate divide and conquer principle by divide the problem as evenly as possible.

In ***divide step***, the problem is divided into two sub-problems exponent (X , n/2) and exponent (X, (n+1)/2).

In ***conquer step***, the sub-problems are solved recursively.

In ***combine step***, solution of the sub-problems are combine by multiplying them.

Example:

To compute a^{29} the above algorithm will work as follows-

Here n is odd. So first calculate a x $(a^{n/2})^2$

a^{29}=a x a^{28} here, n=28

=a x $(a^{14})^2$

= a x $((a^7)^2)^2$

=a x $((a \times (a^3)^2)^2)^2$

=a x $((a \times (a \times (a^2))^2)^2)^2$

Example:

Compute 2^8 using divide and conquer method.

Here a=2, n=8

N is even . So it calculate $(a^{n/2})^2$

$2^8 = (2^4)^2$

$= ((2^2)^2)^2$

$= (4^2)^2$

$=16^2$

$=256$

Example:

Compute 3^7 using divide and conquer method.

Here a=3, n=5

N is odd. So, calculate a x $(a^{n/2})^2$

$3^5 = 3 \times (3^3)^2$

$=3 \times (3 \times (3^2))^2$

$=3 \times (3 \times 9)^2$

$=3 \times (27)^2$

$=3 \times 729$

$=2187$

2.9 LET US SUM UP

- Divide and conquer algorithm has three steps.
- Divide the problem into smaller independent sub-problems.
- Conquer by solving these sub-problems.
- Combine these sub-problems to together.
- The sub-problems solved by a divide and conquer is non overlapping.
- For binary search divide and conquer strategy is applied recursively for a given sorted array.
- Merge sort is a recursive algorithm that splits the array into two subarrays , sorts each subarray , and then merges the two sorted arrays into a single sorted array. The array is divided until its size becomes 0 or 1.
- Merge sort is an external sorting algorithm.

- In merge sort in divide step sub-problems are divided into two halves.
- In conquer step sub-problems are sorted individually.
- In combine Step sub-problems are combine to find the resultant sorted array.
- Quick sort is an internal sorting algorithm. In its basic form it was developed by C.A.R Hoare in 1960.
- In merge sort , the list to be sorted is divided at its midpoint into subarrays which are independently sorted and later merged. In quick sort, the division to the sorted subarrays is made, so that the sorted subarrays do not need to merged later.
- The quick sort algorithm stop when there is only one element in the subarray to be sorted or if there is no element in the subarray to be sorted.

2.10 ANSWERS TO CHECK YOUR PROGRESS

1.

a) sorted, b) middle, c) middle, d) non-overlapping, e) independent

CHECK YOUR PROGRESS – 2

2.

a) 1,0, b) sorted, c) temporary d) external

3. Internal
4. In quick sort sorting is done in place. So, there is no need to combine the sub array later.

2.11 FURTHER READINGS

- T. H. Cormen, C. E. Leiserson, R.L.Rivest, and C. Stein, "Introduction to Algorithms", Second Edition, Prentice Hall of India Pvt. Ltd, 2006.
- Ellis Horowitz, Sartaj Sahni and Sanguthevar Rajasekaran, Computer Algorithms/ C++, Second Edition, Universities Press, 2007.

2.12 MODEL QUESTIONS

1. What is divide and conquer strategy?
2. What is external and internal sorting? Give examples.
3. How does the binary search algorithm follow the divide and conquer method? Explain with an example
4. Write a recursive and non recursive function for binary search algorithm.
5. How does merge sort follow the divide and conquer strategy? Give one example.
6. What are the difference between quick sort and merge sort algorithm?
7. Write a recursive function to sort elements using merge sort.
8. Write quick sort algorithm and explain with an example.
9. Sort the following element using quick sort.

   34 12 45 67 55 23 11 17 19 38 28 44 40
10. Compute 2^{10} using divide and conquer method.
11. What are the sub-problems for compute 3^9 using divide and conquer method? Also find out how the sub-problems are combined.
12. In the following element search the key element 12 using binary search, which uses divide and conquer method.

    3 4 6 7 9 10 12 13 14 18
13. Sort the following elements using merge sort

    23 45 19 14 16 12 30 34 15 18 10 9

UNIT – 3 : GREEDY METHOD

UNIT STRUCTURE

3.1 LEARNING OBJECTIVE

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

- know about the greedy algorithm
- describe the greedy method applied in knapsack problem
- elaborate the job sequencing with deadline
- define minimum spanning tree problem
- describe application of minimum spanning tree problem in Prim's and Kruskal algorithm
- elaborate the shortest path problem using Dijkstra algorithm

3.2 INTRODUCTION

Greedy algorithm is typically used in optimization problem. Algorithm for optimization problems go through a sequence of steps. All of these problems have n inputs and require us to obtain a subset that satisfies some constraints. Any subset that satisfies some constraints is called feasible solution. The solution finds a given objective function which value is either maximizes or minimizes. A feasible solution that does this is called an optimal solution. In this unit, we will discuss about the concept of greedy methods and its application in various problems like Knapsack problems and minimum spanning tree etc.

3.3 GENERAL STRATEGY OF GREEDY ALGORITHM

A greedy algorithm always makes the choice that looks best at the moment. That is it makes a locally optimal choice that may be lead to a globally optimal solution. This algorithm is simple and more efficient compared to other optimization algorithm. This heuristic strategy does not always produce an optimal solution, but sometimes it does.

There are two key ingredients in greedy algorithm that will solve a particular optimization problem.

1. Greedy choice property
2. Optimal substructure

1. *Greedy choice property*:

A globally optimal solution can be arrived at by making a locally optimal (greedy) choice. In other words, when a choice is to be made, then it looks for best choice in the current problem, without considering results from the sub-problems. In

this algorithm choice is made that seems best at the moment and solve the sub-problems after the choice is made. The choices made by a greedy algorithm may depend on choices so far, but it can not depend on any future choice or solution to the sub-problems. The algorithm progress in a top down manner, making one greedy choice one after another, reducing each given problem instances into smaller one.

2. *Optimal substructure:*

A problem is said to have optimal substructure if an optimal solution can be constructed efficiently from optimal solution to its sub-problem. The optimal substructure varies across problem domain in two ways-

- **i)** How many sub-problems are used in an optimal solution to the original problem.
- **ii)** How many choices we have in determining which sub-problem to use in an optimal solution.

In Greedy algorithm a sub-problem is created by having made the greedy choice in the original problem. Here, an optimal solution to the sub-problem, combined with the greedy choice already made, yield an optimal solution to the original problem.

3.4 KNAPSACK PROBLEM

There are n items, i^{th} item is worth v_i dollars and weight w_i pounds, where v_i and w_i are integers. Select item to put in knapsack with total weight is less than W, So that the total value is maximized. This problem is called knapsack problem.
This problem finds, which items should choice from n item to obtain maximum profit and total weight is less than W.

The problem can be explained as follows-

A thief robbing a store finds n items, the i^{th} item is worth v_i dollar and weight w pounds, where v_i and w_i are integers. He wants to take as valuable load as possible, but he can carry atmost W pounds in his knapsack, where W is an integer. Which item should he take?

There are two types of knapsack problem.

1. ***0-1 knapsack problem****:*
   In **0**-1 knapsack problem each item either be taken or left behind.

2. ***Fractional knapsack problem***:
   In fractional knapsack problem fractions of items are allowed to choose.

3.5 GREEDY STRATEGY APPLIED IN 0-1 KNAPSACK PROBLEM

The greedy algorithm in 0-1 knapsack problem can be applied as follows-

1. ***Greedy choice***:
   Take an item with maximum value per pound.
2. ***Optimal substructure***:
   Consider the most valuable load that weights atmost W pounds. These W pounds can be choose from n item. If j^{th} item is choose first then remaining weight $W-w_i$ can be choose from n-1 remaining item excluding j.

3.6 GREEDY STRATEGY APPLIED IN FRACTIONAL KNAPSACK PROBLEM

1. ***Greedy choice:***
   Take an item or fraction of item with maximum value per pound.
2. ***Optimal substructure***:
   If we choose a fraction of weight w of the item j, then the remaining weight atmost W-w can be choose from the n-1 item plus w_i-w pounds of item j.

Although, both the problems are similar, the fractional knapsack problem is solvable by greedy strategy, but 0-1 knapsack problem are not solvable by greedy algorithm.

Consider the following problem-

There are 3 items. The knapsack can hold 50 pounds. Item1 weight 10 pounds and its worth is 60 dollar, item2 weight 20 pounds and its worth 100 dollars, item3 weight 30 pounds and its weight 120 dollars. Find out the items with maximum profit which the knapsack can carry.

Solution:

Here,

W = 50 pounds

Item	***Weight (w pound)***	***Worth (v dollar)***
Item1	10	60
Item2	20	100
Item3	30	120

Let, an item I has weight w_i pounds and worth v_i dollar.

Value per pound of I = v_i / w_i .

Thus, value per pound for-

Item1 = w_1 / v_1

= 60 dollars / 10 pounds

= 6 dollars/pounds

Item2 = w_2 / v_2

= 100 dollars / 20 pounds

= 5 dollars/pounds

Item3 = w_3 / v_3

= 120 dollars / 30 pounds

= 4 dollars/pounds

We can select maximum of 50 pounds.

So, using greedy strategy in 0-1 knapsack problem

1st choice is Item1.

2nd choice is Item2.

Total weight = 10 + 20 pounds

= 30 pounds

Total worth = 60 +100 dollars

=160 dollars

But this is not the optimal choice.

The optimal choice will choose item 2 and 3. Then,

Total weight = 20 + 30 pounds

= 50 pounds

Total worth = 100 + 120 dollars

= 220 dollars.

Hence , 0-1 knapsack problem is not solved by greedy strategy.

Now, using greedy strategy in fractional knapsack problem –

1st choice is item1.

2nd choice is item2

Total weight = 30 pounds

But the size of the knapsack is 50 pounds.

So, it will take remaining 20 pounds from item3 (fraction of item3) and its worth is 4 x 20=80 dollars.

Hence,

Total weights = 50 pounds.

Total worth = 60+100+ 80 dollars

=240 dollars.

Hence , an optimal solution can be obtain from fractional knapsack problem using greedy strategy.

CHECK YOUR PROGRESS

1. Write True or False

   a) Greedy choice always looks for the best choice in the current problem.

   b) 0-1 knapsack problem is solvable by greedy algorithm.

2. What is optimal substructure?

3. What is greedy strategy for knapsack problem?

3.7 JOB SEQUENCING WITH DEADLINE

Now, we will discuss about the job sequencing problem. The problem is stated as below-

1. There are n jobs to be processed on a machine
2. Each job i has a deadline $d_i \geq 0$ and profit $p_i \geq 0$
3. p_i is earned iff the job is completed by its deadline
4. To complete the job, it is processed in one machine for a unit of time.
5. Only one machine is available for processing job
6. Only one job is processed at a time on the machine.
7. A feasible solution is a subset of job J such that each job is completed by its deadline.
8. An optimal solution is a feasible solution with a maximum profit.

This problem can be solved by greedy algorithm. For the optimal solution, after choosing a job , it will add the next job to the subset such that $\sum_{i \in J} p_i$, increases and resulting subset become feasible. p_i is the total profit of i^{th} subset of jobs. In other words we have to check all possible feasible subset J with their total profit value, for a given set of jobs.

Feasible solution for a set of job J is such that, if the jobs of set J can be processed in the order without violating any deadline then J is a feasible solution.

Example :

Let ,

no. of job, n = 4 and

jobs are 1, 2, 3, 4

profit $(p_1,p_2,p_3,p_4) = (100,10,15,27)$

deadline $(d_1,d_2,d_3,d_4) = (2,1,2,1)$.

Find the optimal solution set.

Solution:

SL No.	***Feasible Solution***	***Processing Sequence***	***Profit***
1	(2,1)	(1,2)	110
2	(1,3)	(1,3) or(3,1)	115
3	(1,4)	(4,1)	127
4	(2,3)	(2,3)	25
5	(3,4)	(4,3)	42
6	(1)	(1)	100
7	(2)	(2)	10
8	(3)	(3)	15
9	(4)	(4)	27

Here solution 3 is optimal. The optimal solution is got by processing the job 1 and 4 in the order job 4 followed by job 1. The

maximum profit is 127. Thus, the job 4 begins at time zero and job 1 end at time 2.

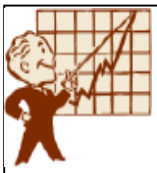
Consider solution 3 i.e maximum profit job subset J = (1,4)
Here , at first J= Ø and $\sum_{i \in J} p_i$=0.

Job 1 is added to J as it has the largest profit and is a feasible solution.

Next add job 4 .Then also J = (1,4) is feasible because if the job processes in the sequence (4,1) then job 4 will start in zero time and job 1 will finish in 2 time within its deadline.

Next if job 3 is added then j=(1,3,4) is not feasible because all the job 1,3,4 can not be completed within its deadline. So job 3 is not added to the set. Similarly after adding job 2 J=(1,2,4) is not feasible.

Hence J = (1,4) is a feasible solution set with maximum profit 127. This is an optimal solution.



CHECK YOUR PROGRESS

4. True/False
   i. In job sequencing, a feasible solution is a subset of job such that each job is completed by its deadline.
   ii. In job sequencing an optimal solution is a feasible solution with a minimum profit.

3.8 OPTIMAL MERGE PATTERNS

Now let us discuss about the optimal merge patterns. It can be stated as follows :

- Two sorted file containing n and m records respectively could be merged together to obtain one sorted file in time O(n + m). When more than two sorted files are merged together then merge can be done by repeatedly merging the sorted files in pairs.

  For example-

Problem 1: There are 5 sorted files F1,F2,F3,F4,F5 and each file has 20,30,10,5,30 records respectively.

If merge these files pair wise then-

M1 = F1&F2
= 20 + 30
= 50 (i.e merging F1 and F2 requires 50 moves)

M2 = M1&F3
= 50 +10
=60

M3 = M2&F4
= 60 + 5
= 65

M4 = M3&F5
= 65 + 30
= 95

Hence Total time required to moves records is –

50+60+65+95 = 270

- Different pairing requires different amount of computing time. The problem can be stated as-

What is the optimal way to pair wise merge *n* sorted files?
Or What is the minimum time needed to pair wise merge *n* sorted files?

We can solved this problem using greedy algorithm. The greedy algorithm attempt to find an optimal merge pattern.

Greedy method for optimal merge pattern:
Sorts the list of file and at each step merge the two smallest size files together.

Example: The above given problem 1 can be solved as follows-

Sort the files according to their number of records.
(5,10,20,30,30) = (F4,F3,F1,F2,F5)

Merge the first two files-
(5,10,20,30,30) => (15,20,30,30)
Merge the next two files-
(15,20,30,30) => (30,30,35)
Merge the next two files-
(30,30,35) => (35,60)
Merge the last two files-
(35,60) => (95)
Hence, total time require is 15+35+60+95= 205

This is the optimal merge pattern for the given problem instance. This merging is also called two way merge pattern because each merge step involve merging of two files.

The two way merge pattern can be represented by binary merge trees. For the above *problem1* the binary merge tree representing the optimal merge pattern is as follows-

.

Here the leaf nodes are given as square and represent the five given files. These nodes are called external nodes. The remaining nodes are drawn as circle and is called internal nodes. Each internal node has exactly two children and it represent file obtained by merging the files represented by its two children. The number in the each node is the length (i.e the number of records) of the file represented by that record.

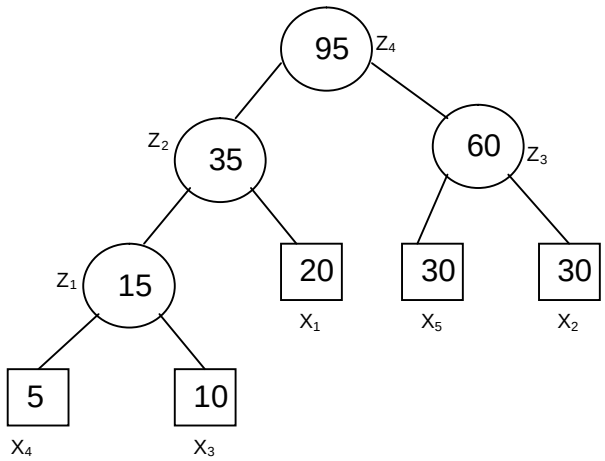


Fig 3.1 Binary merge tree representing a merge pattern.

Here a node at level *i* is at a distance of *i -1* from the root (In the above tree x_4 is at a distance 3 from root z_4).

If d_i is the distance from the root to external node for a file x_i and q_i is the length of the file x_i , then the total number of records move for the binary merge tree is-

$$\sum_{i=1..n} d_i g_i$$

This sum is called the weighted external path length of the tree. An optimal two way merge pattern is minimum weighted external path length of a binary merge tree.

CHECK YOUR PROGRESS

5. How does the greedy choice property applied in optimal merge pattern problem?

6. True/False

i. Optimal merge pattern is also called two way merge pattern.

ii. In optimal merge pattern, in each step two largest files are merged.

3.9 MINIMUM SPANNING TREE

Before going to the definition of the minimum spanning tree let us define what a spanning tree is :

Spanning tree:

A spanning tree is a connected graph, say G = (V, E) with V as set of vertices and E as set of edges, is its connected acyclic sub-graph that contain all the vertices of the graph.

Now the minimum spanning tree can be defined as:

Minimum spanning tree:

A minimum spanning tree T of a positive weighted graph G is a minimum weighted spanning tree in which total weight of all edges are minimum

w(T) = $\sum_{(u,v) \in T}$ **w (u, v)** is minimized.

Where w(u, v) is the cost of the edge (u, v).

For example;

Let us consider connected graph G given in fig

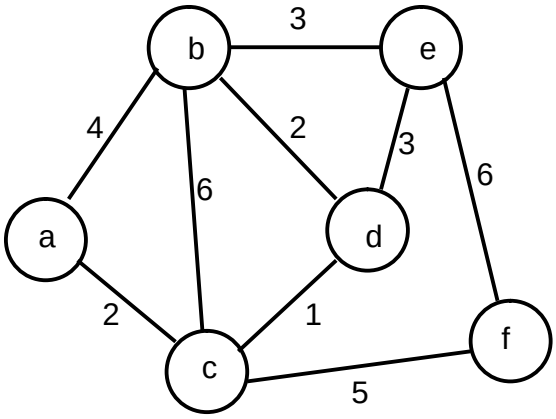


Fig. 3.2 A Connected graph G

Now, the minimum spanning trees are for the graph G is-

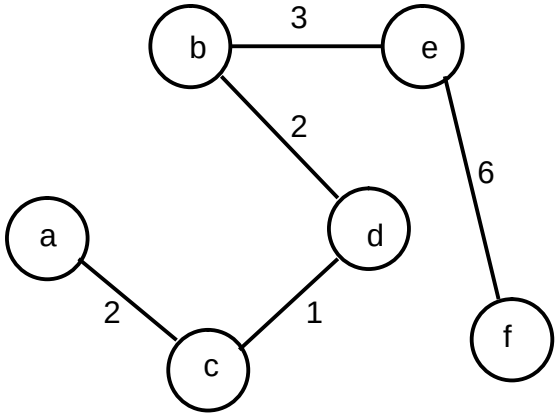


Fig. 3.3 A spanning tree for the graph G

Application of Minimum spanning tree:

i. In design of electric circuit network .
ii. It is used in traveling salesman problem.

The ***minimum spanning tree problem*** is the problem of finding a minimum spanning tree for a given weighted connected graph

There are two algorithms to solve minimum spanning tree problem

1. Kruskal algorithm
2. Prim algorithm

The general approaches of these algorithms are-

- The tree is built edge by edge.
- Let T be the set of edges selected so far.
- Each time a decision is made. Include an edge e to T s.t. Cost (T) + w (e) is minimized, and T U {e} does not create a cycle.

Both these algorithms are greedy algorithm. Because at each step of an algorithm, one of the best possible choices must be made. The greedy strategy advocates making the choice that is best at the moment. Such a strategy is not generally guaranteed to globally optimal solution to a problem.

3.10 PRIM'S ALGORITHM

The prim's algorithm uses greedy method to build the sub-tree edge by edge to obtain a minimum cost spanning tree. The edge to include is chosen according to some optimization criterion. Initially the tree is just a single vertex which is selected arbitrarily from the set V of vertices of a given graph G. Next edge is added to the tree by selecting the minimum weighted edge from the remaining edges and which does not form a cycle with the earlier selected edges. The tree is represented by a pair (V', E') where V' and E' represent set of vertices and set of edges of the sub-tree of minimum spanning tree.

The algorithm is as follows-
The algorithm continuously increases the size of a tree, one edge at a time, starting with a tree consisting of a single vertex, until it finds all vertices.

- *Input:* A non-empty connected weighted graph with vertices *V* and edges *E* (the weights are positive).

- *Initialize:* V' = {*x*}, where *x* is an arbitrary node (starting point) from V,

  E' = { }

- Repeat until V' = V;
- Choose an edge (*u*, *v*) with minimal weight such that *u* is in V' and *v* is not in V' (if there are multiple edges with the same weight, any of them may be picked)
- Add *v* to V' and (*u*, *v*) to E' if edge (u, v) will not make a cycle with the edges already in E'.
- Output: V' and E' describe a minimal spanning tree

Example:

Let us consider the following graph G.

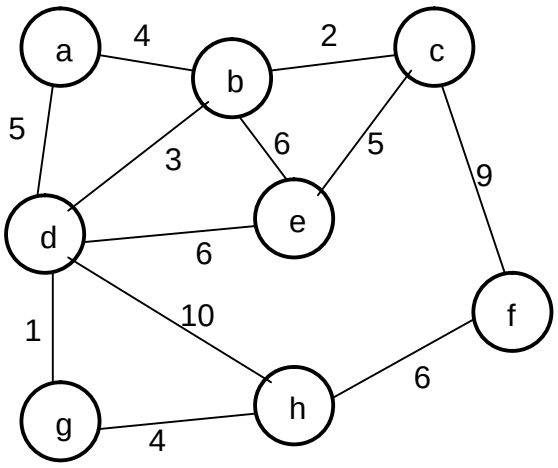


Fig. 3.4 Prim's algorithm applied on the Graph G

Initially vertex *a* is selected. So, V' will contain *a*.

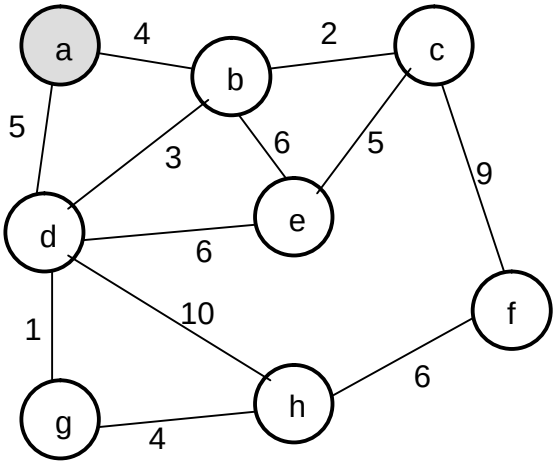


Fig. 3.5 Vertex a is selected

V' = {a}

E' = Ø

After first iteration, the minimum weight edge connected *a* and other vertices of V is selected. In this case from vertex *a* there are two edges *ab* and *ad* to vertex *b* and *d*.

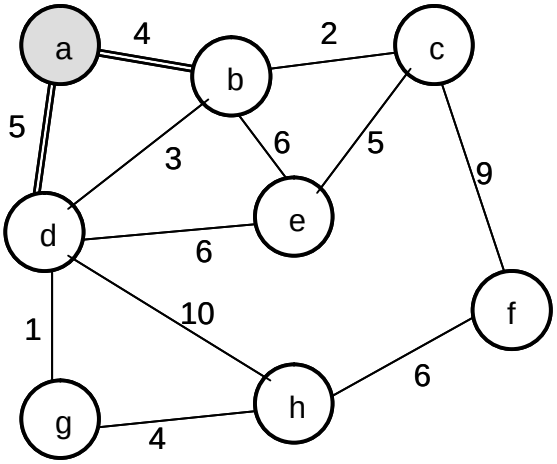


Fig. 3.6 Finds the minimum weighted edge

Between *ab* and *ad* weight of *ab* is minimum. Hence, after first iteration vertex *b* is include to V' and edge *ab* is included to E'.

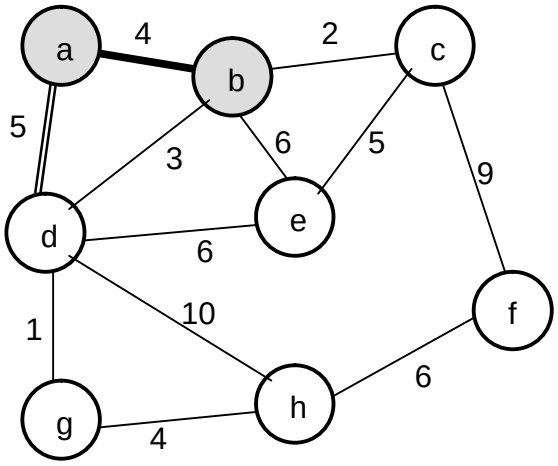


Fig. 3.7 Minimum weighted edge selected

V' = {a, b}

E' ={ ab }

In the next iteration we select the minimum weight edge, which does not make a cycle with previously selected edges in E', from the edges not included in E' and edges connected one vertex from V' and another vertex not in V'. Here edges from a and b to any other vertex. Here, edges are ad, bd, be, bc from which we can select the minimum weight edge.

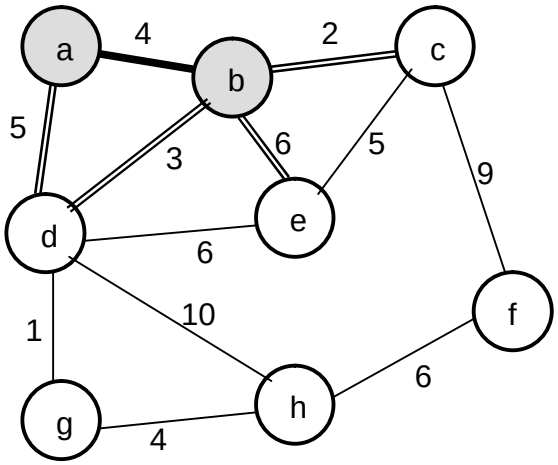


Fig. 3.8 Finds the minimum weighted edge

Here, weight of bc is minimum and it does not make a cycle with ab. Thus bc is selected in this iteration.

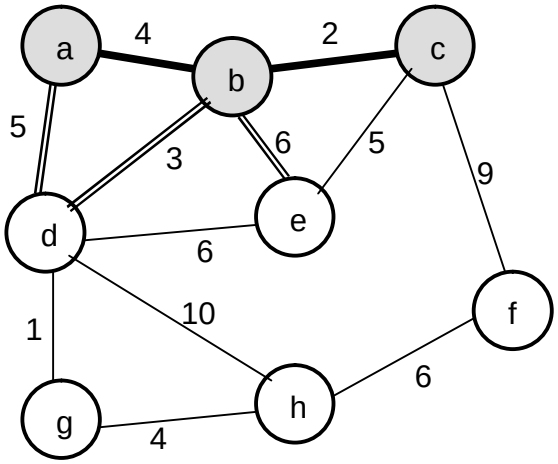


Fig. 3.9 Minimum weighted edge selected

V' = { a, b, c }

E' = { ab, bc }

In the next iteration we can consider the edges that have a,b or c as one of the vertex . Here the edges are ad, bd, be, ce, cf. we can not consider ab and bd because they are already selected.

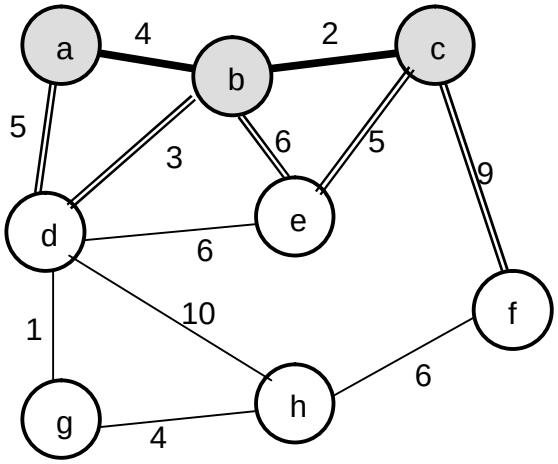


Fig. 3.10 Finds minimum weighted edge

From these edges weight of bd is minimum and it does not make a cycle with the edge in E'. Thus bd is selected.

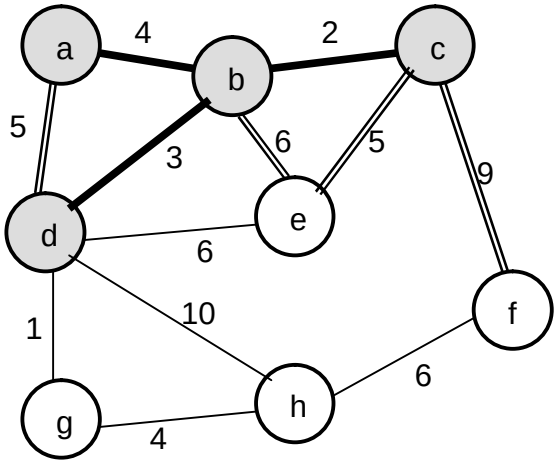


Fig. 3.11 Minimum weighted edge selected

V' = { a, b, c, d }

E' = { ab, bc, bd }

In the next iteration we consider the edges (excluding already selected edges) that have a, b, c, d as one vertex. Here edges are ad, be, ce, cf, de, dh, dg.

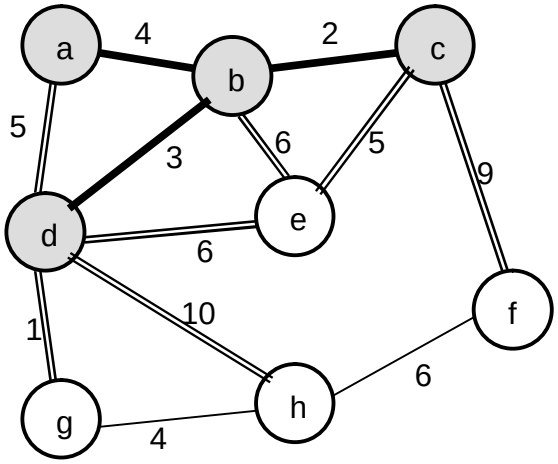


Fig. 3.12 Finds Minimum weighted edge

The weight of dg is minimum and it does not make a cycle with the edges in E'. Thus dg is selected.

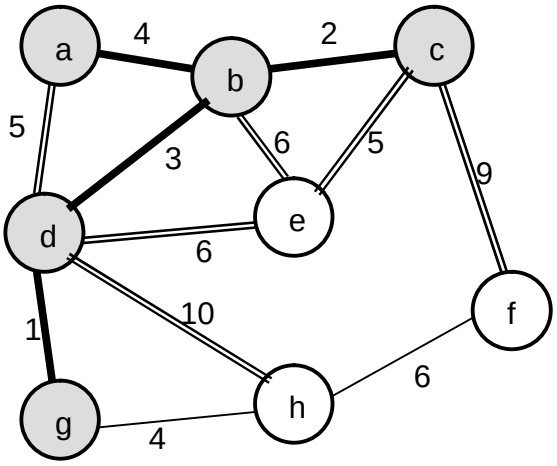


Fig. 3.13 Minimum weighted edge selected

V' = { a, b, c, d, g }

E' = { ab, bc, bd, dg }

In the next iteration considered edges are ad, be, de, gh, dh, ce, cf .

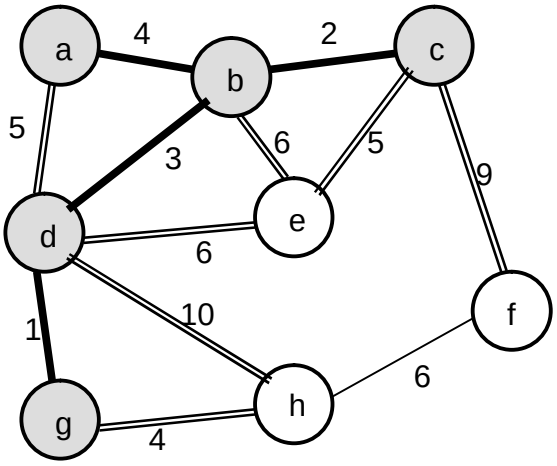


Fig. 3.14 Finds Minimum weighted edge

Among these edges weight of gh is minimum and it does not make any cycle with already selected edges in E'. Thus, gh is selected.

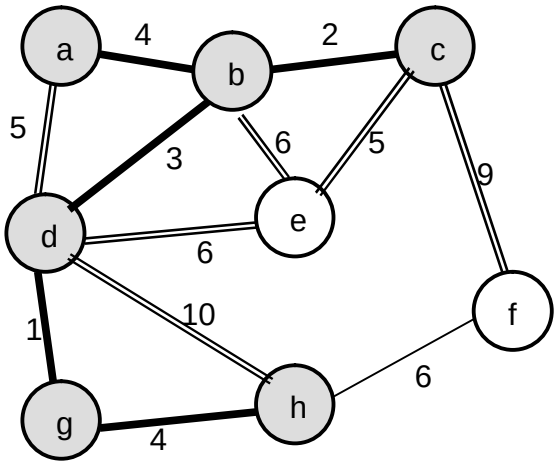


Fig. 3.15 Minimum weighted edge selected

V' = { a, b, c, d, g, h }

E' = { ab, bc, bd, dg, gh }

In the next iteration consider the edges that has one vertex from V' and connect another vertex excluding already selected edges. Here edges are ad, be, ce, cf, de, dh, hf.

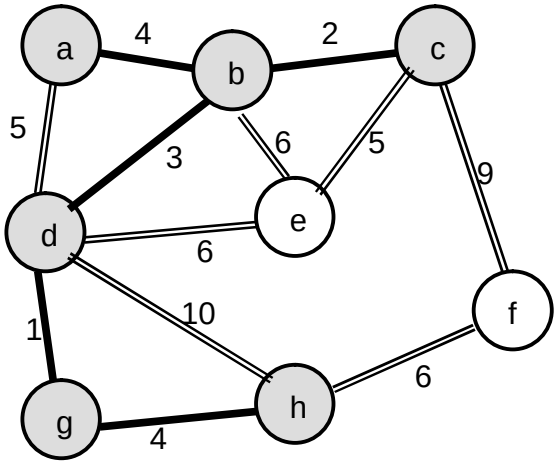


Fig. 3.16 Finds Minimum weighted edge

Among these weight ad and ce are minimum. If select ad then it make a cycle with the already selected edge ab and bd of E'. So,

ad can not be selected. If we select ce it will not make a cycle with the edges of E'.

Thus ce is selected.

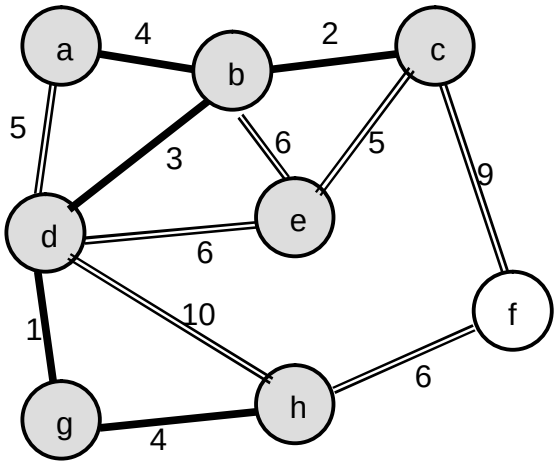


Fig. 3.17 Minimum weighted edge selected

V' = { a, b, c, d, e, g, h }

E' = { ab, bc, bd, dg, gh, ce }

In the next iteration considered edges are ad, be, de, dh, cf, hf.

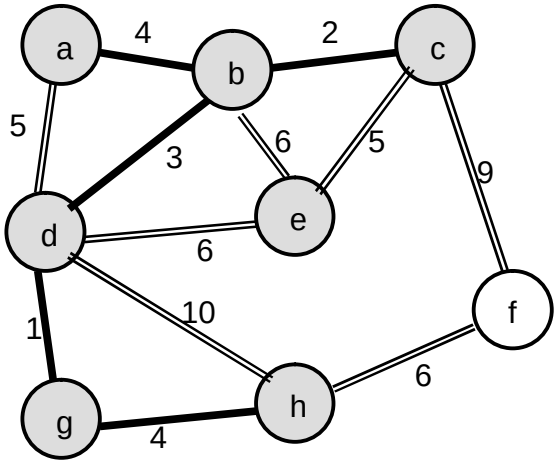


Fig. 3.18 Finds Minimum weighted edge

Among these weight of ad is minimum. But it makes a cycle with already selected edges ad and ab. So, ad is rejected. Next minimum weight is of edge de, be and hf. But this two edges will also make cycle. So de and be are also rejected. hf will not make a cycle. Thus hf is considered.

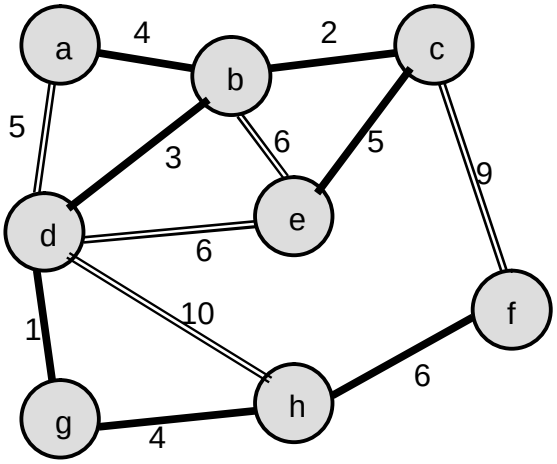


Fig. 3.19 Minimum weighted edge selected

V' = { a, b, c, d, e, f, g, h }

E' = { ab, bc, bd, dg, gh, ce, hf }

Next, the edges be, de, cf ,ad, dh can not included to form the tree because they make a cycle with already selected edges. Hence the final spanning tree is-

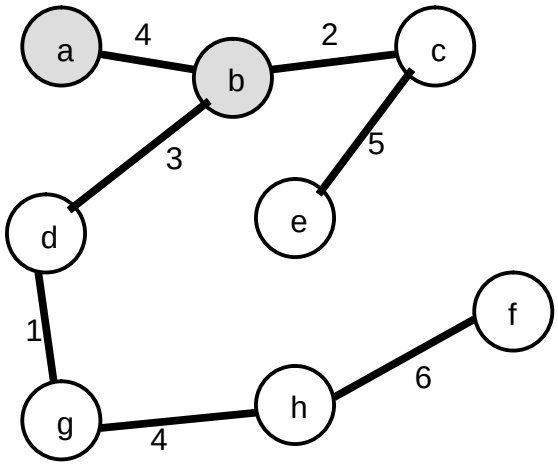


Fig. 3.20 Final spanning tree of graph G

3.11 KRUSKAL ALGORITHM

Another method of finding minimum spanning tree is Kruskal algorithm. In this algorithm the edges of the graph are considered in non decreasing order. The result is a forest of trees that grows until all the trees in a forest (all the components) merge in a single tree.

The algorithm is as follows-

- create a forest *F* (a set of trees), where each vertex in the graph is a separate tree
- create a set *S* containing all the edges in the graph
- while *S* is nonempty and F is not yet spanning
- remove an edge with minimum weight from *S*
- if that edge connects two different trees, then add it to the forest, combining two trees into a single tree
- otherwise discard that edge.

Example:

Let us consider the following graph-

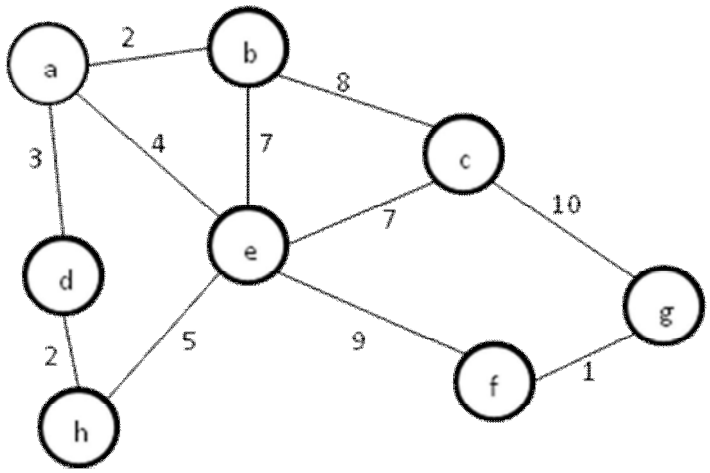


Fig. 3.21 Krushkal algorithm applied on graph G

Initially there are 8 vertices. So initially there are 8 trees in the forest F and S contains all the edges in the graph.

F = a b c d e f g h

S = ab, bc, ad, ae, be, ce, dh, cg, ef, fg, he

In the first iteration we consider the smallest weight edge from the set S. If the both vertex of the edge connect two different trees in the forest F then that edge is selected and combined the two trees into a single tree. Here, in first iteration weight of fg edge is 1, which is minimum. It connects two vertices f and g. In the forest F, f and g belongs to two separate tree. Thus, fg edge is selected and fg is removed from the set S and f and g trees are combined into a single tree fg in F.

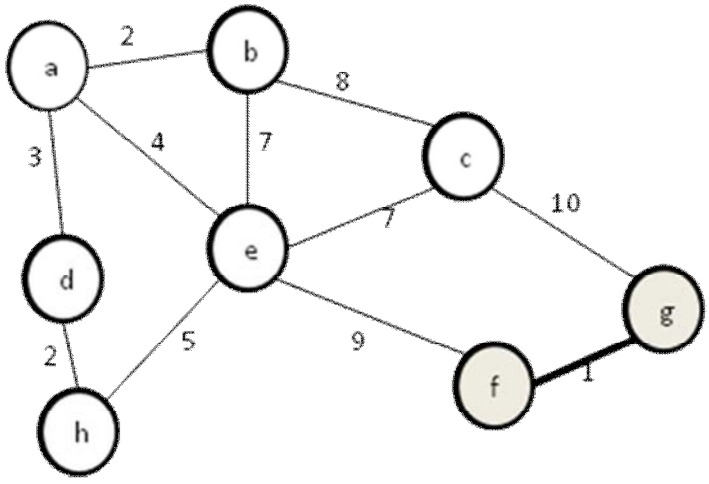


Fig. 3.22

F = a b c d gf e h

S = ab, bc, ad, ae, be, ce, dh, cg, ef, he

In the second iteration, after removing fg edge form S we consider the minimum weight edge from the new set S i.e from remaining edges we consider the minimum weight edge. Next the minimum weight edges are ab and dh. Each of which has weight 2. We can consider any one of the edge. Because for each edges their

connected vertices belongs to two different tree. Let us select the edge ab. Thus,

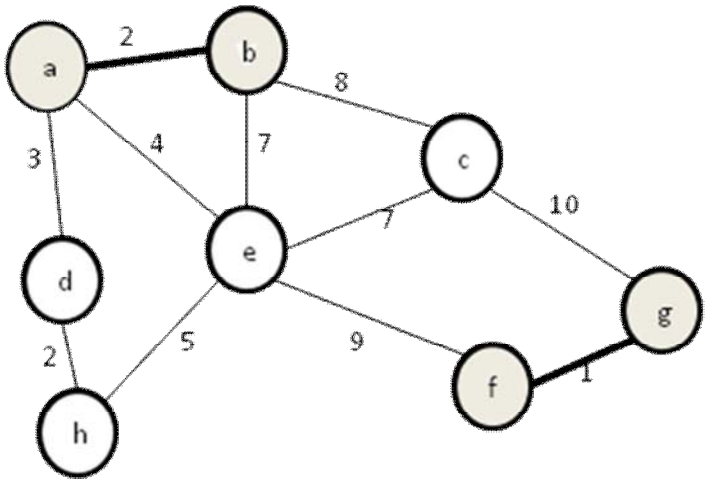


Fig. 3.23

F = ab c d gf e h

S = bc, ad, ae, be, ce, dh, cg, ef, he

In third iteration minimum weight edge in S is dh of weight 2. Now vertex d and h of edge dh belongs to two different tree. Hence, we can select edge dh. Thus, F and S becomes

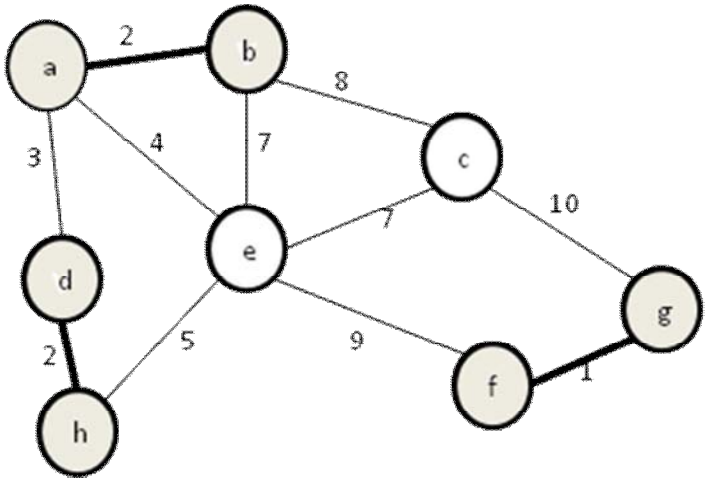


Fig. 3.24

F = ab c dh gf e

S = bc, ad, ae, be, ce, cg, ef, he

In fourth iteration from the remaining edges set S minimum edge is ad of weight 3.Now the vertices a and d belongs to two different tree ab and dh respectively in F. So. Edge ad is selected. Now

,

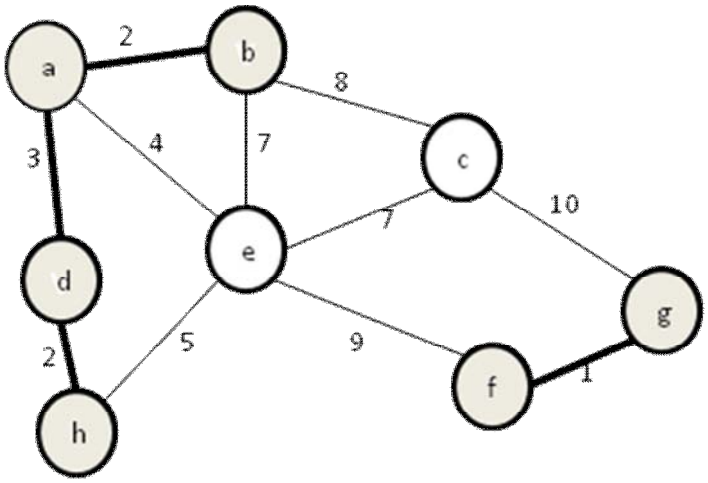


Fig. 3.25

F = abdh c gf e

S = bc, ae, be, ce, cg, ef, he

Now, in next iteration the minimum weight edge in S is ae. Here a and e belongs to two different tree abdh and e respectively in F. Hence, ae is selected and removed from S and two tree combined to a single tree abdhe. Thus,

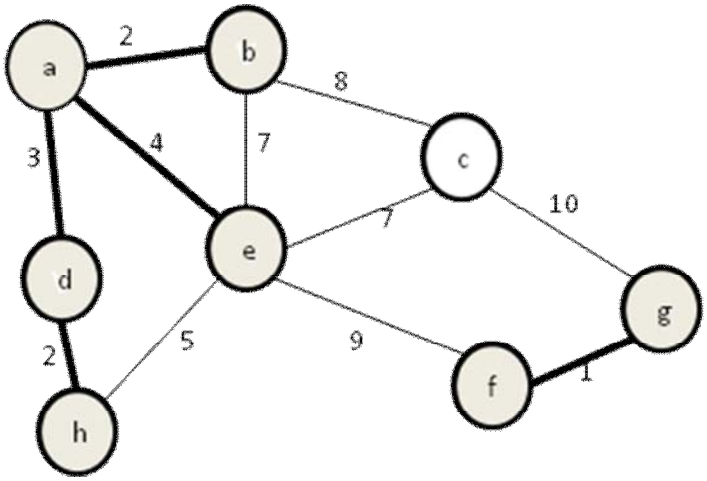


Fig. 3.26

F = abdhe c gf

S = bc, be, ce, cg, ef, he

In next iteration minimum weight edge in S is he of weight 5. Now, h and e belongs to same tree abdhe. So, edge he is not selected. Simply remove he from S. So, S becomes

S = bc, be, ce, cg, ef

Now, from S the minimum weight edges are be and ce of weight 7 in each. For edge be , b and e is from same tree abdhe in F. So, be can't selected. So, this be remove from S.

S = bc, ce, cg, ef

We can select ce because c and e belongs to two different tree in F. Hence

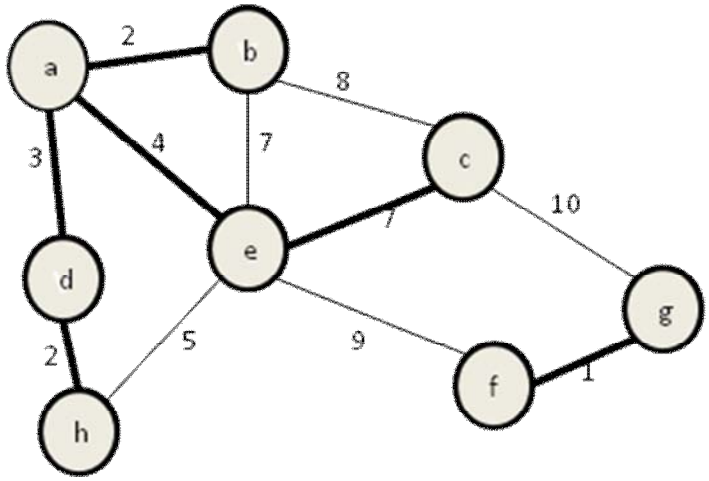


Fig. 3.27

F = abcdeh gf

S = bc, cg, ef

In next iteration minimum weight edge is bc of weight 8.We can't consider bc because b and c is in same tree in F. Thus

S = cg, ef

Next minimum weight edge in S is ef. Edge ef can selected because vertices e and f belongs to two different trees in F. Thus

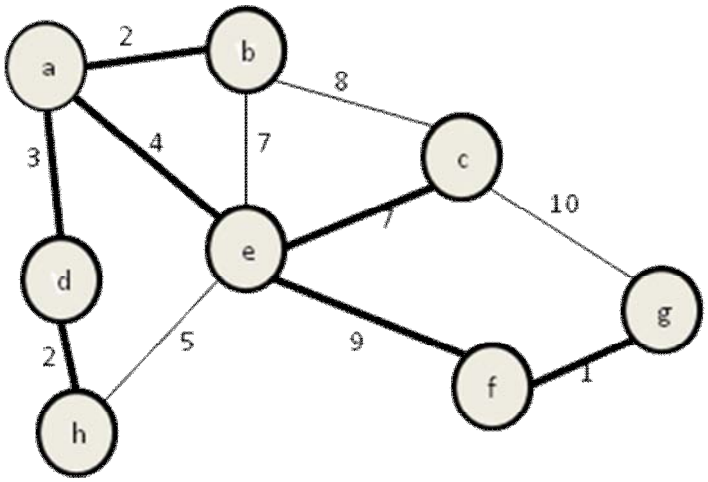


Fig. 3.28

F = abcdefgh

S = cg

In next iteration edge is cg. Edge cg is not selected because vertices c and g is in one tree in F. Hence the final tree is-

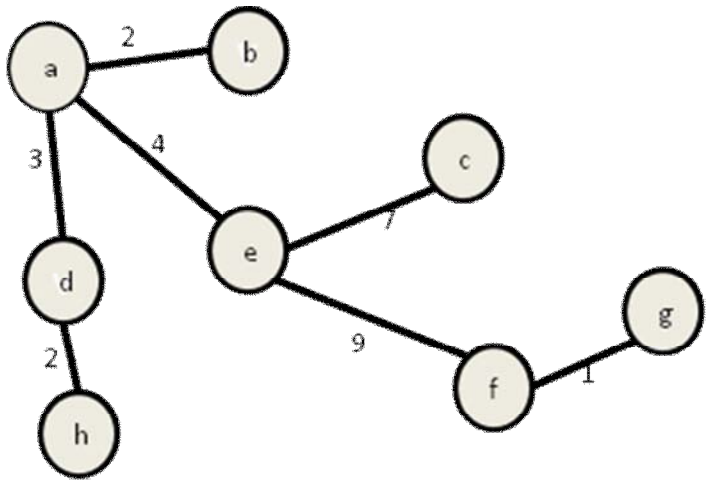


Fig. 3.29 Final tree of graph G

F = abcdefgh

S = nil

CHECK YOUR PROGRESS

7. What is minimum spanning tree?
8. What are the algorithms to solve minimum spanning tree problem ?

3.12 DIJKSTRA'S ALGORITHM

Shortest path problem:

For a given weighted and directed graph G= (V, E), the shortest path problem is the problem of finding a shortest path between any two vertex v ϵ V in graph G. The property of the shortest path is such that a shortest path between two vertices contains other

shortest path within it i.e any other sub-path of a shortest path is also a shortest path.

Single source shortest path problem:

In a single source shortest path problem , there is only one source vertex S in the vertex set V of graph G=(V, E). Now this single source shortest path problem finds out the shortest path from the source vertex S to any other vertex in v € V.

Optimal substructure of a shortest path:

Optimal substructure of a shortest path can be stated that any other sub-path of a shortest path is also a shortest path. Here is the lemma-

Lemma:

Given a weighted directed graph G=(V, E) with weight function w: E -> R, let p = (v_1,v_2 ----, v_k) be a shortest path from vertex v_1 to vertex v_k, and for any i and j such that $1 \le i \le j \le k$, *let P_{ij} = (v_1,v_{i+1}, ------,v_j) be the sub-path of P from vertex v_i to vertex v_j. Then P_{ij} is a shortest path from v_i to v_j.*

Dijsktra algorithm solves the single source shortest path problem. But the algorithm works only on a directed and positive weighted graph. Positive weighted graph means where weights of all edges are non negative i.e G=(V, E) is a positive weighted graph then w(u, v) ≥ 0. Dijsktra algorithm is a greedy algorithm.

Dijkstra algorithm is as follows-

For a given graph G=(V, E) and a source vertex s, it maintains a set F of vertices .Initially no vertex is in F. For a vertex u ϵ V - F, (i.e for a vertex which is in V, but is not in F) if it has minimum shortest distance from source s to u then u is added to F. This process is continue till V - F is not equal to null.

```
DIJKSTRA( G, w, s)
1.      INITIALIZE_SINGLE_SOURCE(G,s)
        1.1 for each vertex v € V[G]
        1.2 do d [v] = ∞
        1.3  Π [v] = NIL
        1.4  d [s] = 0
2.      F = Ø
3.      Q = V [G]
4.      while Q != Ø
5.          do u = EXTRACT_MIN(Q)
6.            F = F U {u}
7.            for each vertex v € Adj [u]
8.             do RELAX (u, v, w)
                  8.1 if d [v] > (d [u] + w(u, v))
                  8.2  then d [v] = d [u] + w(u, v)
                  8.3 Π [v] = u;
```

In line 1 (from line 1.1 to 1.4) initialize the value of d and π . Here d [v] means distance from source to vertex v and π [v] means parent of vertex v. Initially source to source distance is 0. So, d [s] = 0 . Also for all vertices v € V , d[v] is set as ∞ and π [v] as NIL.

In line 2 it initializes set F to empty set as initially no vertex is added to it.

In line 3 Q is a min-priority queue and initially it contains all vertices set V[G] of graph G.

In line 4 the while loop of line 4-8 will continue until the min-priority queue Q become empty.

In line 5 it extracts the minimum distance vertex u from source s i.e u € V - F for which d [u] is minimum.

In line 6 u is added to F.

In line 7-8 (from line 8.1 to 8.3) for all vertex v which is adjacent to u , calculate the distance to v through vertex u. If this

value is less than d[v] then update d [v] to this new value.

Make parent of v, π [v] = u.

3.12.1 Example:

Apply dijkstra algorithm for the following graph G.

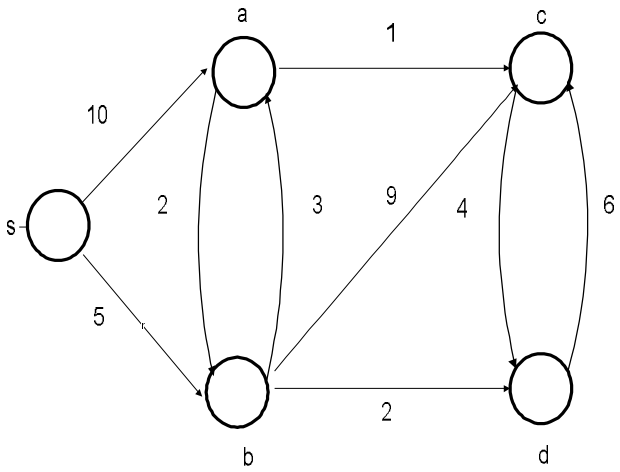


Fig. 3.30 Dijkstra algorithm applied on G

Initially

d[s]=0;

[s]=NIL;

And distance of all other vertices set as ∞.

d[a]= ∞, π [a]=NIL;

d[b]= ∞, π [b]=NIL;

d[c]= ∞, π [c]=NIL;

d[d]= ∞, π [d]=NIL;

s is added to F.

F={s}

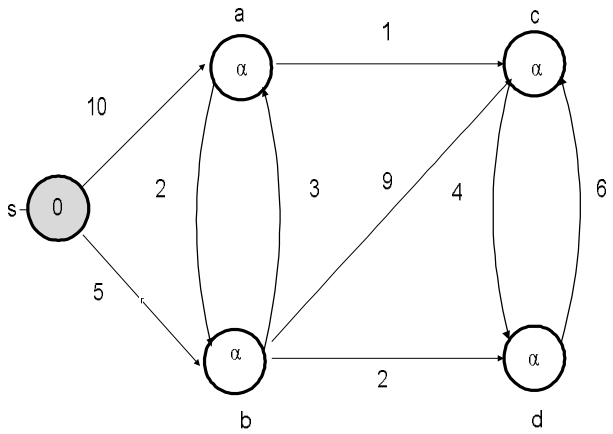


Fig. 3.31

In 2nd iteration

Adj[s]={a, b} and they are not in F.

Now d[a]=d[s]+w(s,a)

=0+10

=10

This new d[a] value is less than previous d[a] value i.e 10<∞

Hence d[a] is updated to 10

d[a]=10;

π [a]=s;

Similarly d[b]=d[s]+w(s,b)

=0+5

=5<previous d1[b]

=5<∞

d[b]=5;

π [b]=s;

and d[c]=∞, π [c]=NIL;

d[d]=∞, π [d]=NIL;

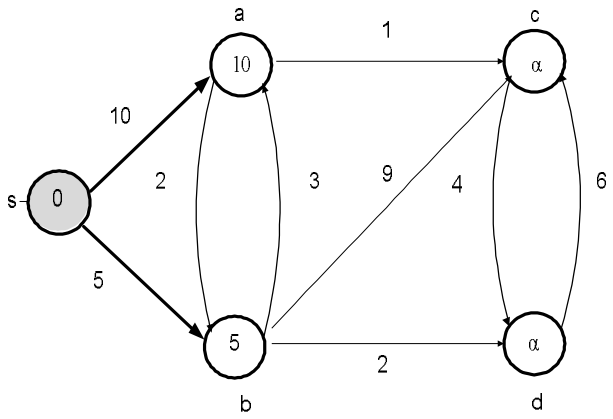


Fig. 3.32

Now, among these d[a],d[b],d[c],d[d] minimum value is d[b]i.e distance of vertex b is minimum from source.

Hence b is added to F

F={s,b}

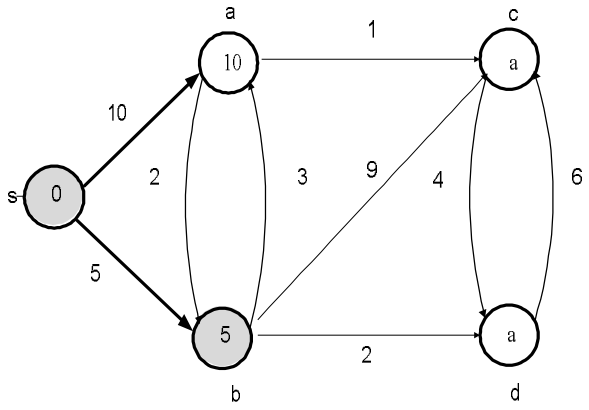


Fig. 3.33

In 3rd iteration,

Adj[b]={a,c,d} and they are not in F

For vertex a,

d[a]=d[b]+w(b,a)

=5+3

=8< previous d[a]

=8<10

Hence,

d[a]=8;

π [a]=b;

For vertex c

d[c]=d[b]+w(b,c)

=5+9

=14< previous d[c]

=14<∞

d[c]=14;

π [c]=b;

For vertex d

d[d]=d[b]+w(b,d)

=5+2

=7< previous d[d]

=7<∞

d[d]=7;

π [d]=b;

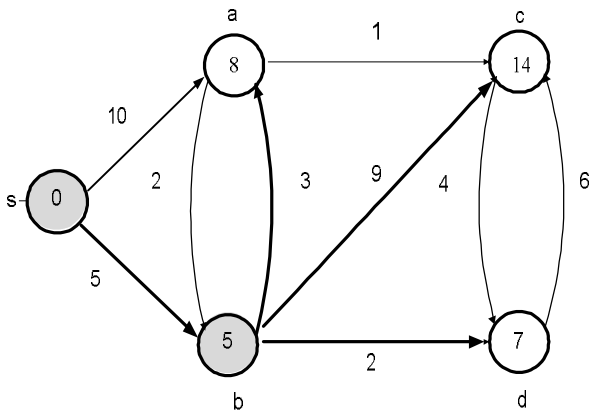


Fig. 3.34

Now among d[a],d[c],d[d] the minimum d value is d[d]

So, vertex d is added to F.

F={s,b,d}

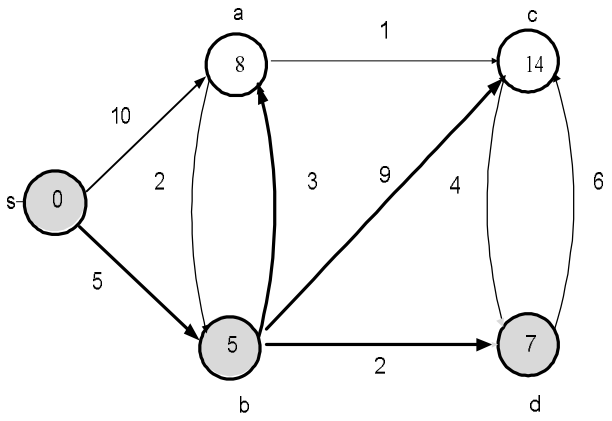


Fig. 3.35

In 4th iteration

Adj[d]={c} and c is not in F

Now d[c]=d[d]+w(d,c)

= 7+6

=13< previous d[c]

=13<14

d[c]=13;

π [c]=d;

And d[a]=8;

π [a]=s;

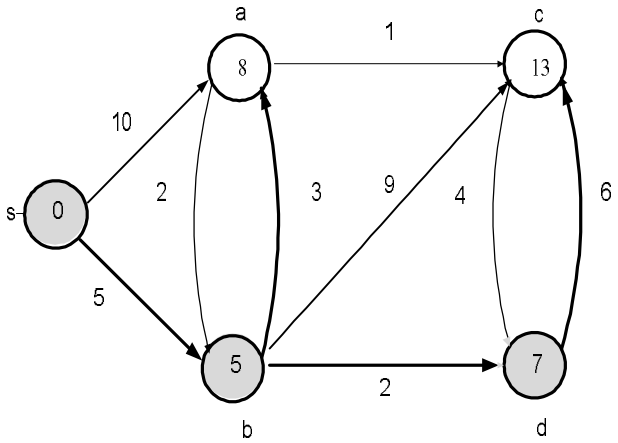


Fig. 3.36

Now the minimum of d[a] and d[c] is d[a].

So, vertex a is added to F

F={s,b,d,a}

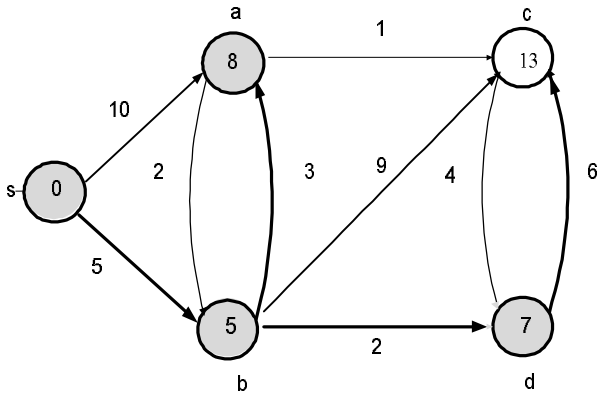


Fig. 3.37

In 5th iteration last added vertex is a.

Adj[a]={b,c},

Here c is not in F. But b is in F i.e b is already selected.

So, we will consider vertex c only.

d[c]=d[a]+w(a,c)

=8+1

=9< previous d[c]

So, d[c]=9

π [c]=a

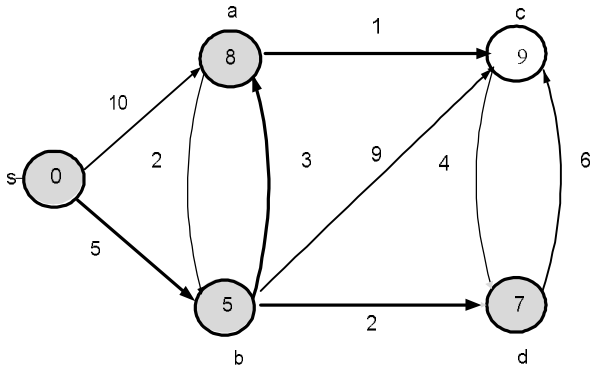


Fig. 3.38

Hence c is added to F

F={s,b,d,a,c}

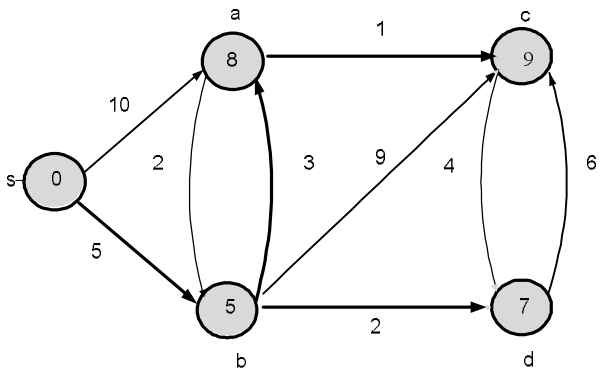
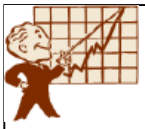


Fig. 3.39

There is no vertex to added in F.So, the algorithm terminate here.



CHECK YOUR PROGRESS

9. What is single source shortest path problem?
10. True/False
    i. In Dijkstra algorithm there are two source vertices.
    ii. Dijkstra algorithm can solve shortest path problem.

3.13 LET US SUM UP

- Greedy algorithm is typically used in optimization problem.
- Optimal solution finds a given objective function which value is either maximizes or minimizes.
- A greedy algorithm always makes the choice that looks best at the moment. That is it makes a locally optimal choice that may be lead to a globally optimal solution.

- In Greedy algorithm choice is made that seems best at the moment and solve the sub-problems after the choice is made.
- Greedy algorithm progress in a top down manner,
- A problem is said to have optimal substructure if an optimal solution can be constructed efficiently from optimal solution to its sub-problem.
- Knapsack problem: There are n items, i^{th} item is worth v_i dollars and weight w_i pounds, where v_i and w_i are integers. Select item to put in knapsack with total weight is less than W, So that the total value is maximized
- There are two types of knapsack problem.
    i. 0-1 knapsack problem
    ii. fractional knapsack problem:
- In 0-1 knapsack problem each item either be taken or left behind.
- In fractional knapsack problem fractions of items are allowed to choose.
- the fractional knapsack problem is solvable by greedy strategy, but 0-1 knapsack problem are not solvable by greedy algorithm.
- In the job sequencing with deadline problem, a feasible solution is a subset of job J such that each job is completed by its deadline and optimal solution is a feasible solution with a maximum profit.
- the optimal way to pair wise merge n sorted files
- A spanning tree is a connected graph , say G = (V, E) with V as set of vertices and E as set of edges, is its connected acyclic sub-graph that contain all the vertices of the graph.
- A minimum spanning tree T of a positive weighted graph G is a minimum weighted spanning tree in which total weight of all edges are minimum

- Two algorithm to solve minimum spanning tree problem are- Kruskal algorithm and Prim algorithm
- For a given weighted and directed graph G= (V, E), the shortest path problem is the problem of finding a shortest path between any two vertex v € V in graph G.
- In a single source shortest path problem , there is only one source vertex S in the vertex set V of graph G=(V, E).

3.14 ANSWERS TO CHECK YOUR PROGRESS

CHECK YOUR PROGRESS – 1

1. a) True, b) True

2. A problem is said to have optimal substructure if an optimal solution can be constructed efficiently from optimal solution to its sub problem.

3. i)Greedy choice ii) Optimal substructure

4. i. True ii. False

5. According to greedy choice property at each step it looks for its best solution in the current set. In optimal merge pattern sorts the list of file and at each step merge the two smallest size files(best choice at that moment) together from the current file sets.

6. i. True ii. False

7. A minimum spanning tree T of a positive weighted graph G is a minimum weighted spanning tree in which total weight of all edges are minimum

$$w(T) = \sum_{(u,v) \in T} w(u, v)$$ is minimized.

Where w(u, v) is the cost of the edge (u, v).

8. Prim's algorithm and Kruskal algorithm

9. Single source shortest path problem of graph G=(V, E) is to finds out the shortest path from the only source vertex S to any other vertex in v € V. Here,only one source vertex S in the vertex set V

10. i. False ii. True

3.15 FURTHER READINGS

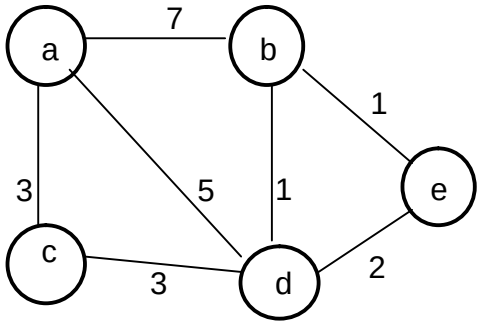
- T. H. Cormen, C. E. Leiserson, R.L.Rivest, and C. Stein, "Introduction to Algorithms", Second Edition, Prentice Hall of India Pvt. Ltd, 2006.
- Ellis Horowitz, Sartaj Sahni and Sanguthevar Rajasekaran, Computer Algorithms/ C++, Second Edition, Universities Press, 2007.

3.16 MODEL QUESTIONS

1. What is optimal substructure?
2. What is greedy strategy?
3. Write briefly about knapsack problem. Explain with an example that greedy algorithm does not work for 0-1 knapsack problem.
4. What is optimal substructure for 0-1 knapsack and fractional knapsack problem?
5. Consider the following job sequencing problem. Find the feasible solution set.

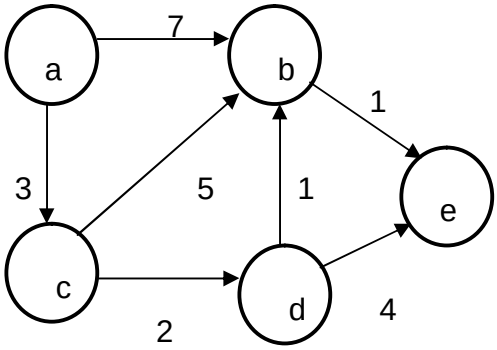
Job	1	2	3	4
Profit	10	20	15	5
Deadline	2	3	3	2

6. What is minimum spanning tree? Find the minimum spanning tree for the following graph using Prim's and Kruskal algorithm.



7. Find out the shortest path using Dijkstra algorithm for the following graph

source



8. "A globally optimal solution can be arrived at by making a locally optimal choice ". Explain briefly.

UNIT - 4DYNAMIC PROGRAMMING

UNIT STRUCTURE

4.1 LEARNINGOBJECTIVES

After goingthrough this unit, you will be able to:

- understand the concept of Dynamic Programming
- solve problems using dynamic programming approach
- get familiarize with optimality conditions

4.2 INTRODUCTION

Intheprecedingunits,wehaveseensomeelegantdesignprinciplessuch asdivide-and-conquer, greedy algorithm-thatyielddefinitivealgorithmsfor avariety ofimportant computationaltasks.Thedrawbackofthesetechniquesisthattheycano nlybeusedonveryspecific typesofproblems.In this unit, we will introduce you the dynamic programming technique. We will concentrate on elaborating 0/1 Knapsack problem and travelling salesman problem in this unit.

4.3GENERAL STRATEGY

The first step in solving an optimization problem by dynamic programming is to characterize the structure of an optimal solution. A problem issaid to possess an optimal substructure if an optimal solution to the problem contains within the optimal solutions of its sub-problems. Whena problem exhibits optimal substructure, is a good clue that dynamic programming might apply. In dynamic programming, we build an optimal solution to the problem from optimal solutions of its sub-problems. Consequently, we must take care that, the range of sub-problems we consider includes those sub-problemswhich are used in the optimal solution.Some important concept of dynamic programming are :

Stage of a Problem

The dynamic programming problem can be divided in to a sequence of smaller sub-problems called stages of the original problem.

State of a Problem

The condition of decision process at a stage is called its state. The decision variable which specify the condition of decision process at a particular stage is called state variable.

Principle of Optimality

A problem is said to satisfy the Principle of Optimality if the sub-solutions of an optimal solution of the problem are themselves optimal solutions for their sub-problems. For examples: The shortest path problem satisfies the Principle of Optimality. This is because if a,x1,x2,...,xn,b is a shortest path from node a to node b in a graph, then the portion of xi to xj on that path is a shortest path from xi to xj.

Characteristics of Dynamic Programming

i) The Problem can be divided into stages, with a policy decision at each stage
ii) Each stage consist of a number of states associated with it
iii) Decision at each stage convert the current stage in to a state associated with next stage.
iv) The state of the system at a stage is described by state variable.
v) When the current state is known, an optimal policy for the remaining stages is independent of the policy of the previous ones.
vi) The solution procedure begins by finding the optimal solution of each state from the optimal solutions of its previous stage.

Steps of Dynamic Programming

Dynamic programming design involves 4 major steps:

1. Develop a mathematical notation that can express any solution and sub-solution for the problem at hand.
2. Prove that the Principle of Optimality holds.
3. Develop a recurrence relation that relates a solution to its sub-solutions, using the mathematical notation of step 1. Indicates the initial values for that recurrence relation, and terms that signifies the final solution.
4. Write an algorithm to compute the recurrence relation.

4.4MULTISTAGE GRAPHS

A multistage graph G = (V, E)is a directed graph in which the vertices are partitioned into k ≥ 2 disjoint sets V_i, 1 ≤ i ≤ k. In addition, if <u,v> is an edge in E, then u∈ V_i and v∈ V_{i+1} for some i, 1≤i ≤ k. The set V_1 and V_k are such that $|V_1| = |V_k| = 1$. Let s and t respectively, be the vertices in V_1 and V_k. The vertex s is the source and t the sink. Let c(i , j) be the cost of edge <i,j>. The cost of a path from s to t is the sum of the costs of the edges on the path. The multistage graph problem is to find a minimum-costpath from s to t. Each set V_i defines a stage in the graph. Because of the constraints on E, every path from s to t starts in stage 1, goes to stage 2, then to stage 3 and so on until it terminates at stage k. Fig 4.1 shows a five-stage graph. A minimum-cost path from s to t is indicated by the broken edges in the figure.

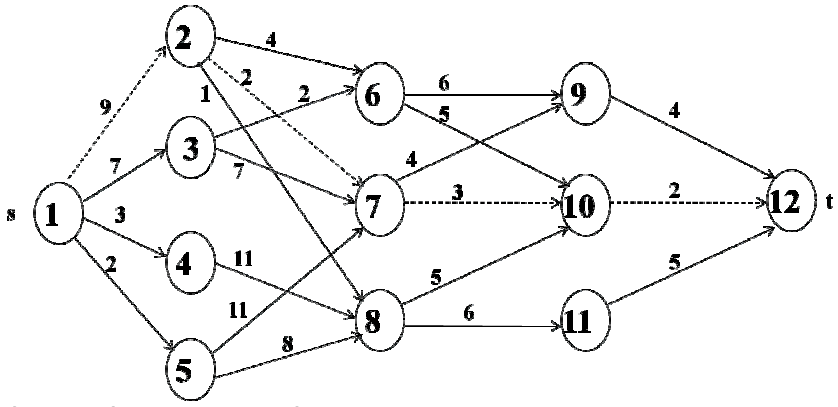


Fig 4.1 Five stage graph

A dynamic programming formulation for a k-stage graph problem is obtained by noticing the fact that, every path from s to tconsistof a sequence of k-2 decisions. The i^{th} decision involves determining which vertex in V_{i+1}, 1 ≤ i ≤ k-2, is to be on the path. It is easy to see that the principle of optimality holds for this problem. Let p(i, j) be a minimum cost path from vertex j in V_i to vertex t. Let cost(i, j)be the cost of this path. Then using the forward formulation approach, we obtain:

$$cost(i, j) = \underset{\substack{l \in Vi+1 \\ <j,l> \in E}}{MIN} \{ c(j, l) + cost (i+1, l)\} \qquad (Eq\ 4.1)$$

Since cost(k-1, j) = c(j, t), if <j,t>∈ E and cost(k-1, j) = α, if <j,t>∉E, the above equation may be solved for cost (1,s) by first computing cost(k-2, j) for all j∈V_{k-2}, then cost(k-3, j) for all j∈V_{k-3}, etc. and finally cost (1,s)

Algorithm Graph_sortest_path (Graph G, k, n, p[ ])

```
Step 1:  cost[n] = 0.0
Step 2:  for j= n-1 to 1
           //let r be a vertex such that <j,r> is an edge of G
           //and c[j][r] + cost [r] is minimum;
Step 3:    cost[j] = c[j][r] + cost [r]
Step 4:    d[j]=r;
Step 5:  end for
Step 6:  p[1]=1, p[k] =n
Step 7:  for j=2 to k-1
Step 8:    p[j]= d[p[j-1]]
Step 9:  end for
```

For the above algorithm we need to index the vertices of V from 1 to n. Indices are assigned according to stages. First index 1 is assigned to s, then the vertices in V_2 are indexed, then vertices in V_3, and so on, vertex t has index n.

The multistage graph problem can also be solved using the backwardapproach. Let bp(i,j) be a minimum-cost path from vertex s to a vertex jin V_i. Let bcost(i,j) be the cost of bp(i,j). From the backward approach weobtain

$$bcost(i, j) = \underset{\substack{l \in Vi-1 \\ <j,l> \in E}}{MIN} \{ c(j, l) + bcost (i-1, l)\} \qquad (Eq\ 4.2)$$

Algorithm BGraph_sortest_path (Graph G, k, n, p[ ])

```
Step 1:  bcost[1] = 0.0
Step 2:  for j= 2 to n
           //let r be a vertex such that <r, j> is an edge of G
           //and c[r][j] + bcost [r] is minimum;
Step 3:    bcost[j] = c[r][j] + bcost [r]
Step 4:    d[j]=r;
Step 5:  end for
Step 6:  p[1]=1, p[k] =n
Step 7:  for j=k-1 to 2
Step 8:    p[j]= d[p[j +1]]
Step 9:  end for
```

Since bcost(2,j) = c(1, j) if (1,j)∈E and bcoat(2,j) = ∞ if (1,j)∉ E,bcost(i,j) can be computed using (4.2) by first computing bcost for i = 3,then for i = 4, and so on.

All-pairs shortest paths

Ifwewantto find the shortest pathnotjust betweensandt butbetweenall pairs of vertices then,oneapproachwouldbeto execute our generalshortest-path algorithmfrom |V| times, oncefor eachstartingnode. The totalrunningtime wouldthen beO(|V|^2|E|). We'llnowseeabetteralternative, theO(|V|^3)dynamicprogramming-basedFloyd-Warshall algorithm.

Finding a better algorithm by using dynamic programming approach, the first question came to our mind is that, whether a better sub-problem exists for computingdistancesbetweenallpairsofverticesin agraph? Simplysolvingtheproblemformoreandmorepairsorstartingpointsis unhelpful,becauseit leadsrightbacktotheO(|V|^2|E|)algorithm.

Oneidea comesto mind isthat, the shortest pathu→ w_1→ …. →w_l→ vbetweenuandvusessomenumberof intermediate nodespossiblynone.
Supposewedisallowintermediatenodesaltogether.
Thenwecansolveall-pairsshortestpathsatonce,theshortestpathfromuto vis simplythe directedge(u, v), if it exists. Now let usgraduallyexpandthesetofpermissibleintermediate nodes. Wecandothisonenodeatatime,updatingtheshortestpathlengths ateachstage. Eventuallythissetgrowsto allofV, atwhichpointallthe verticesareallowedtobeonallpaths, andwehavefound thetrueshortestpathsbetweenverticesof thegraph.

Moreconcretely, numbertheverticesin Vas {1, 2,3 ….,n}, andlet dist(i;j;k)denotethelengthoftheshortestpathfrom itojinwhichonlynodes {1,2,…,k} canbeusedasintermediates. Initially,dist(i;j;0)isthelengthofthedirectedgebetween i andj, ifit exist, andis α otherwise.

Ifweexpandtheintermediatesettoinclude anextranodek,wemustreexamineallpairs i,jandcheckwhetherusingkasanintermediatepointgivesusashorterpathfrom i toj. Butthisis easy:ashortestpathfrom i tojthatuseskalongwithpossiblyother lower numberedintermediatenodesgoesthroughkis justonce. Andwehavealreadycalculatedthelength oftheshortestpathfromi tokandfromktojusingonly lowernumbered vertices.

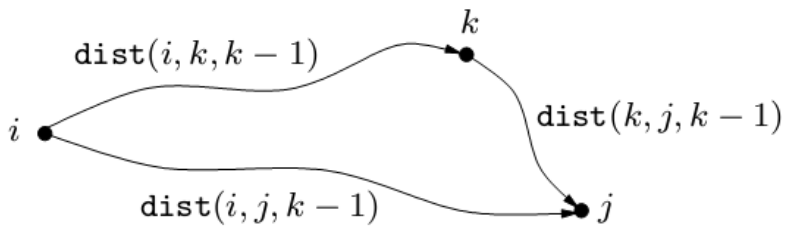


Fig 4.2 Computing Path

Thus, usingkgivesusashorterpathfromi tojifandonlyif
dist(i, k, k-1)+dist(k, j, k-1) <dist(i;j;k 1);
inwhichcasedist(i, j, k)shouldbeupdatedaccordingly.
HereistheFloyd-Warshallalgorithm – andit takesO($|V|^3$)time.

Algorithm All Path(cost, n)

```
Step 1:  for i := 1 to n
Step 2:      for j :=1 to n
Step 2:          dist(I,j,0) := 1;
Step 3:      end for
Step 5:  end for
Step 4:  for all (i, j) ∈ E
Step 5:      dist (i, j, 0) = ℓ (i, j)
Step 6:  end for
Step 7:  for k:=1 to n
Step 8:      for i:=1 to n
Step 7:          for j :=1 to n
Step 8: dist(i, j,k) = min {dist(i,k,k-1) + dist(k, j, k-1), dist(i,j,k-1)}
Step 9:          end for
Step 11:     end for
Step 12: end for
```

Single Source Shortest Path

Problem: Given a directed graph *G(V,E)* with *weighted edgesw(u,v)*, define the *path weight* of a path *p* as

$$w(p)=\sum_{i=1}^{k} w(v_{i-1}, v_i)$$

For a given source vertex *s*, find the *minimum weight paths* to every vertex reachable from *s* denoted

$$\delta(s, v)=\begin{cases} \min\{w(p) \quad s \rightarrow v\} \\ \infty \text{ otherwise} \end{cases}$$

The final solution will satisfy certain caveats:

- The graph cannot contain any *negative weight cycles* (otherwise there would be no minimum path since we could simply continue to follow the negative weight cycle producing a path weight of -∞).
- The solution cannot have any *positive weight cycles*
- The solution can be assumed to have no zero weight cycles (since they would not affect the minimum value).

Therefore given these caveats, we know that the shortest paths must be *acyclic* (with ≤ |*V*| distinct vertices) ⇒ ≤ |*V*| - 1 edges in each path.

We can use this observation on the maximum number of edges on a cycle-free shortest path to obtain an algorithm to determine a shortest path from a source vertex to all remaining vertices in the graph.

Let $dist^{l}[u]$ be the length of a shortest path from the source vertex v to vertex u under the constraint that the shortest path contains at most *l* edges. Then, $dist^{1}[u] = cost[v,u]$, $1 \le u \le n$. As noted earlier, when there are no cycles of negative length, we can limit our search for shortest paths to paths with at most n - 1 edges. Hence, $dist^{n-1}[u]$ is the length of an unrestricted shortest path from v to u.

Our goal then is to compute $dist^{n-1}[u]$ for all u. This can be done using the dynamic programming methodology. First, we make the following observations:

1. If the shortest path from v to u with at most k, k > 1, edges has not more than k - 1 edges, then $dist^{k}[u] - dist^{k-1}[u]$.

2. If the shortest path from v to u with at most k, k > 1, edges has exactly k edges, then it is made up of a shortest path from v to some vertex j followed by the edge (j,u). The path from v to j has k - 1 edges, and its length is $dist^{k-1}[j]$. All vertices j such that the edge (j, u) is in the graph are candidates for j. Since we are interested in a shortest path, the i that minimizes $dist^{k-1}[i] + cost[i, u]$ is the correct value for j.

These observations result in the following recurrence for dist:

$$dist^{k}[u] = \min \{dist^{k-1}[u], \min \{dist^{k-1}[i] + cost[i,u]\}\}$$

This recurrence can be used to compute $dist^{k}$ from $dist^{k-1}$, for k= 2, 3,... ,n - 1.

Algorithm BellmanFord(v, cost, dist, n)

```
Step 1:  for i := 1 to n do
Step 2:      dist[i] := cost[v, i];
Step 3:  end for
Step 4:  for k := 2 to n - 1 do
Step 5:      for each u such that u≠v and u has at least
         one incoming edge
Step 6:          for each <i, u> in the graph
Step 7:              if dist[u] >dist[i] + cost[i,u]
Step 8:                  dist[u] := dist[i] + cost[i,u];
Step 9:              end if
Step 10:         end for
Step 11:     end for
Step 12: end for
```

4.5OPTIMAL BINARY SEARCH TREE

A binary search tree is a tree where the key values are stored in the internal nodes, the external nodes (leaves) are null nodes, and the keys are ordered lexicographically, i.e. For each internal node all the keys in the left sub-tree are less than the keys in the node, and all the keys in the right sub-tree are greater.

When we know the frequency of searching each one of the keys, it is quite easy to compute the expected cost of accessing each node in the tree. An optimal binary search tree is a binary search tree which has minimal expected cost of locating each node. In our problem, we are not concerned with the frequency of searching for a missing node. For example:

Node ID	0	1	2	3	4	5
Key	A	B	C	D	E	F
Frequency	4	1	1	2	8	16

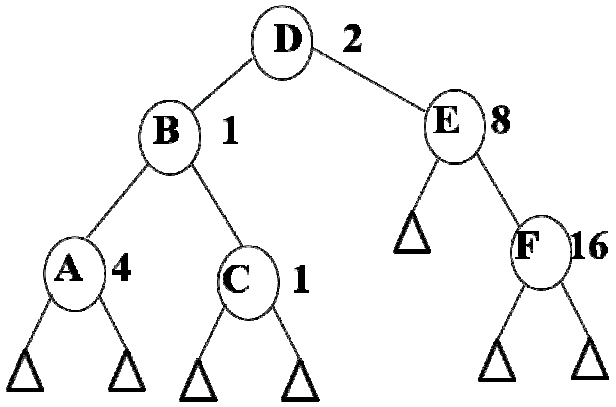


Fig 4.3: Optimal Binary search tree ex1.

[2*1 + (1+8)*2 + (4+1+16)*3] = 83

The expected cost of successful search is 83, is computed by multiplying each frequency by its level (starting w i^{th} the root at 1). A different tree will have a different expected cost:

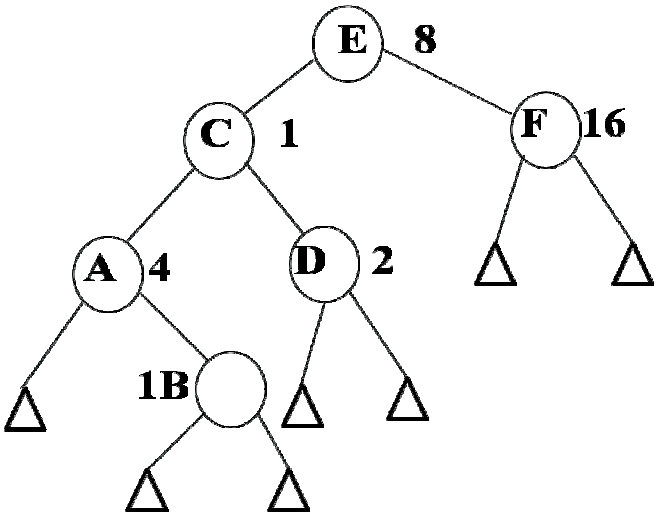


Fig 4.4: Optimal Binary search tree ex2

[8*1 + (1+16)*2 + (4+2)*3 + 1 *4] = 64

It's clear that the tree in fig 4.3 is not optimal. - It is easy to see that the nodes having higher frequencies are closer to the root, then the tree will have a lower expected cost.

In obtaining a cost function for binary search trees, it is useful to add a external node in place of every empty sub-tree in the search tree. If a binary search tree represents n identifiers, then there will be exactly n internal nodes and n+1 external nodes.

If a successful search terminates at an internal node at level l, then the expected cost contribution from the internal node $a_i$is p(i) * level(a_i).

Unsuccessful searches terminates the external nodes, let the unsuccessful searches terminates at node E_i, if the failure node is at level l, then only l-1 comparisons will be made, so the cost contribution of this node is q(l) * (level(E_i) -1)

The preceding decision leads to the following formula for the expected cost of a binary search tree.

$$\sum_{1 \le i \le n} p(i) * level(a_i) + \sum_{1 \le i \le n} q(i) * (level(E_i) - 1) \qquad \text{(Eq 4.3)}$$

We define a optimal binary search tree for the identifier set { a_1, a_2, …., a_n} to be a binary search tree for which Eq 4.3 is minimum.

To solve this problem by dynamic programming we need to view the construction of such a tree as the result of a sequence of decisions and then observe that the principle of optimality holds when applied to the problem state resulting from a decision. A possible approach to this would be to make a decision as to which of the a_i's should be assigned to the root of the tree. If we choose a_k, then it is clear that the internal nodes for a_1, a_2, …., a_{k-1} as well as external nodes for the classes E_1, E_2, …., E_{k-1} will be in the left

subtree, l, of the root. The remaining nodes will be in the right subtree, r. Define

$$\text{cost(l)} = \sum_{1 \le i < k} p(i) * \text{level}(a_i) + \sum_{1 \le i < k} q(i) * (\text{level}(E_i) - 1)$$

$$\text{cost(r)} = \sum_{k < i \le n} p(i) * \text{level}(a_i) + \sum_{k < i \le n} q(i) * (\text{level}(E_i) - 1)$$

In both cases the level is measured by regarding the root of the respective subtree to be at level 1.

Using w(l,j) to represent the sum $q(i) + \sum_{l=i+1}^{j} (q(l)+p(l))$, we obtain the following as the expected cost of the search tree

$$p(k) + \text{cost(l)} + \text{cost(r)} + w(0, k-1) + w(k, n) \qquad \text{Eq4.4}$$

If the tree is optimal then Eq4.4 must be minimum. Hence, cost(l) must be minimum over all the binary search trees containing $a_1, a_2, \ldots, a_{k-1}$ and $E_1, E_2, \ldots, E_{k-1}$. Similarly cost(r) must be minimum. If we use c(l,j) to represent the cost of an optimal binary search tree, t_{ij}, containing $a_{i+1}, a_{i+2}, \ldots, a_j$ and $E_{i+1}, E_{i+2}, \ldots, E_j$, then for the tree to be optimal, we must have cost(l) = c(0, k-1) and cost(r) = c(k,n). In addition k must be choose such that

$$p(k) + c(0, k-1) + c(k,n) + w(0, k-1) + w(k, n)$$

is minimum. Hence c(0,n) we obtain

$$c(0,n) = \min_{i \le k \le n} \{p(k) + c(0, k-1) + c(k,n) + w(0, k-1) + w(k, n)\} \quad \text{Eq 4.5}$$

wegan generalize Eq 4.5 to obtain any c(i,j)

$$c(i,j) = \min_{i \le k \le j} \{p(k) + c(i, k-1) + c(k,j) + w(i, k-1) + w(k, j)\}$$

$$c(i,j) = \min_{i \le k \le j} \{c(i, k-1) + c(k,j)\} + w(i,j) \qquad \text{Eq 4.6}$$

Equation 4.6 can be solved for c(0,n) by first computing all c(i,j) such that j-i=1. Next we can compute all c(i,j) such that j-i =2, then all c(i,j) with j-i=3, etc. if during this computation we record the root r(i,j) of each tree t_{ij}, then an optimal binary search tree can be constructed from these r(i,j).

CHECK YOUR PROGRESS

1. State True or False

a) All the problems can be solved by using dynamic programming technique.
b) To solve a problem by using dynamic programming, the problem must have to possess principle of optimality.
c) A multistage graph can have a cycle.
d) A optimal binary search tree is a binary search tree which has minimal expected cost of locating each node

4.6 0/1 KNAPSACK PROBLEM USING DYNAMIC PROGRAMMING

In the previous unit, we have discussed about the Knapsack problem, and found that fractional knapsack problem can be solved by using greedy strategy. The 0-1 knapsack problem can only be solved by using dynamic programming. Below we will discuss methods for solving 0-1 knapsack problem.

The naive way to solve this problem is to cycle through all 2^n subsets of the n items and pick the subset with a legal weight that maximizes the value of the knapsack. But, we can find a dynamic programming algorithm that will usually do better than this brute force technique.

Our first attempt might be to characterize a sub-problem as follows:

Let S_k be the optimal subset of elements from $\{I_0, I_1,...I_k\}$. But what we find is that the optimal subset from the elements $\{I_0, I_1,... I_{k+1}\}$ may not correspond to the optimal subset of elements from $\{I_0, I_1,...I_k\}$ in any regular pattern. Basically, the solution to the optimization problem for S_{k+1} might NOT contain the optimal solution from problem S_k.

To illustrate this, consider the following example:

Item	Weight	Value
I_0	3	10
I_1	8	4
I_2	9	9
I_3	8	11

The maximum weight the knapsack can hold is 20.

The best set of items from $\{I_0, I_1, I_2\}$ is $\{I_0, I_1, I_2\}$ but the best set of items from $\{I_0, I_1, I_2, I_3\}$ is $\{I_0, I_2, I_3\}$. In this example, note that this optimal solution, $\{I_0, I_2, I_3\}$, does NOT build upon the previous optimal solution, $\{I_0, I_1, I_2\}$. Instead it builds upon the solution, $\{I_0, I_2\}$, which is really the optimal subset of $\{I_0, I_1, I_2\}$ with weight 12.

So, now, let us rework on our examplewith the following idea:Let B[k, w] represents the maximum total value of a subset S_k with weight w. Our goal is to find B[n, W], where n is the total number of items and W is the maximal weight, the knapsack can carry.

Using this definition, we have $B[0, w] = w_0$, if $w \geq w_0$.
$= 0$, otherwise

Now, we can derive the following relationship that B[k, w] obeys:

$B[k, w] = B[k - 1,w]$, if $w_k > w$
$= \max \{ B[k - 1,w], B[k - 1,w - w_k] + v_k\}$

In general:

1) The maximum value of a knapsack with a subset of items from $\{I_0, I_1, ...I_k\}$ with weight w is the same as the maximum value of a knapsack with a subset of items from $\{I_0, I_1, ... I_{k-1}\}$ with weight w, if weights of item k is greater than W.
   Basically, we can NOT increase the value of our knapsack with weight w if the new item we are considering weighs more than W – because it WON'T fit!!!

2) The maximum value of a knapsack with a subset of items from $\{I_0, I_1, ... I_k\}$ with weight w could be the same as the maximum value of a knapsack with a subset of items from $\{I_1, I_2, ... I_{k-1}\}$ with weight w, if item k should not be added into the knapsack.

3) The maximum value of a knapsack with a subset of items from $\{I_0, I_1, ... I_k\}$ with weight w could be the same as the maximum value of a knapsack with a subset of items from $\{I_0, I_1, ... I_{k-1}\}$ with weight $w-w_k$, plus item k.

You need to compare the values of knapsacks in both case 2 and 3 and take the maximal one.

Recursively, we will still have an $O(2^n)$ algorithm. But, using dynamic programming, we simply perform in just two loops - one loop running n times and the other loop running W times.

Here is a dynamic programming algorithm to solve the 0/1 Knapsack problem:

Input: S, a set of n items as described earlier, W the total weight of the knapsack. (Assume that the weights and values are stored in separate arrays named w and v, respectively.)

Output: The maximal value of items in a valid knapsack.

```
int i, k;
for (i=0; i<= W; i++)
    B[i] = 0

for (k=0; k<n; k++)
{
for (i = W; i>= w[k]; i--)
        {
        if (B[i – w[k]] + v[k]> B[i])
            B[i] = B[i – w[k]] + v[k]
        }
}
```

Clearly the run time of this algorithm is O(nW), based on the nested loop structure and the simple operation inside of both loops. When comparing this with the previous $O(2^n)$, we find that depending on W, either the dynamic programming algorithm is more efficient or the brute force algorithm could be more efficient.

Let's run through an example:

I	Item	w_i	v_i
0	I_0	4	6
1	I_1	2	4
2	I_2	3	5
3	I_3	1	3
4	I_4	6	9
5	I_5	4	7

W = 10

Item	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	6	6	6	6	6	6	6
1	0	0	4	4	6	6	10	10	10	10	10
2	0	0	4	5	6	9	10	11	11	15	15
3	0	3	4	7	8	9	12	13	14	15	18
4	0	3	4	7	8	9	12	13	14	16	18
5	0	3	4	7	8	10	12	14	15	16	19

4.7 TRAVELLING SALESMAN PROBLEM

We have seen how to apply dynamic programming to a subset selectionproblem (0/1 knapsack). Now we turn our attention to a permutation problem.Note that permutation problems usually are much harder to solve thansubset problems as there are n! different permutations of n objects whereasthere are only 2^ndifferent subsets of n objects (n!> 2n). Let G = (V,E)be a directed graph with edge costs c_{ij}. The variable $c_{ij}$is defined such thatc_{ij}> 0 for all i and j and c_{ij} = α if (i,j) ∉ E. Let |V| = n and assumen > 1. A tour of G is a directed simple cycle that includes every vertex inV. The cost of a tour is the sum of the cost of the edges on the tour. Thetraveling salesperson problem is to find a tour of minimum cost.

Different problems can be viewed as the traveling salesman problem.For example, suppose we have to define the route a postal van to pick up mail from mail boxes located at n different sites. If we represent the situation by graphs then the vertices of the graph will be different cities and the edges of the graph are the paths between two cities and the weight of a edge can be the distance between the cities. Our task is to find the route taken by the postal van is a tour with minimum cost or length.

In the following discussion, without losing the main concept, we takethe tour as a simple path that starts and ends at thestarting vertex. Every tour consists of an edge (1,k) for some k ∈ V - {1} and a path from vertex k to vertex 1. The path from vertex k to vertex 1 goes through each vertex in V - {1, k} exactly once. It is easy to see that if the tour is optimal, then the path from k to 1 must be a shortest k to 1 path going through all vertices in V - {1,k}. Hence, the principle of optimality holds. Let g(i,S) be the length of a shortest path starting at vertex i, going through all vertices in S, and terminating at vertex 1. The function g(1, V - {1}) is the length of an optimal salesman's tour. From the principal of optimality it follows that

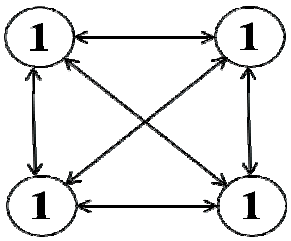
$$g(1, V-\{1\}) = \min_{2 \le k \le n} \{c_{1k} + g(k, V - \{1, k\})\}$$

In general

$$g(i, S) = \min_{j \in S} \{c_{ij} + g(j, S - \{j\})\}$$

The above equation can be solved for g(1, V - {1}) if we know g(k, V - {1, k}) for all choices of k. The g values can be obtained by using this equation. Clearly, g(i, Ø) = c_{j1}, 1≤ i ≤ n. Hence, we can use this equation to obtain g(i, S) for all S of size 1. Then we can obtain g(i,S) for S with |S| = 2, and so on. When |S|< n - 1, the values of i and S for which g(i, S) is needed are such that i ≠ 1, 1∉ S, and i ∉S.

Consider the directed graph of Fig 4.5(a). The edge lengths are given by matrix c of Fig 4.5(b)



(a)

$$\begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$

(b)

Fig 4.5: Directed graph and Edge matrix c

Thus,

$g(2,Ø) = c_{21} = 5$
$g(3,Ø) = c_{31} = 6$
$g(4,Ø) = c_{41} = 8.$

Using the above equation we obtain

$g(2,\{3\}) = c_{23} + g(3,Ø) = 15$ $g(2,\{4\}) = 18$
$g(3,\{2\}) = 18$ $g(3,\{4\}) = 20$
$g(4,\{2\}) = 13$ $g(4,\{3\}) = 15$

Next, we compute g(i, S) with |S| =2, $i \neq 1$, $1 \notin S$ and $i \notin S$.

$g(2,\{3,4\}) = \min \{c_{23}+g(3,\{4\}), c_{24}+g(4,\{3\})\} = 25$
$g(3,\{2,4\}) = \min \{c_{32}+g(2,\{4\}), c_{34}+g(4,\{2\})\} = 25$
$g(4,\{2,3\}) = \min \{c_{42} + g(2,\{3\}), c_{43}+ g(3,\{2\})\} = 23$

Finally, we obtain

$g(1, \{2,3,4\}) = \min \{c_{12}+g(2, \{3,4\}), c_{13}+g(3, \{2,4\}), c_{14}+g(4, \{2, 3\})\}$
$= \min \{35,40,43\}$
$= 35$

An optimal tour of the graph of Figure has length 35. A tour of this length can be constructed if we retain with each g(i, S) the value of j that minimizes the right-hand side of the graph. Let J(i,S) be this value. Then, J(1,{2,3,4}) = 2. Thus the tour starts from 1 and goes to 2. The remaining tour can be obtained from g(2, {3, 4}). SoJ(2, {3, 4}) = 4. Thus the next edge is (2,4). The remaining tour is for g(4, {3}). So J(4, {3}) = 3. The optimal tour is 1, 2, 4, 3, 1.

4.8 FLOW SHOP SCHEDULING

Often the processing of a job requires the performance of several distincttasks. Computer programs run in a multiprogramming environment areinput and then executed. Following the execution, the job is queued for outputand the output is eventually printed. In a general flow shop we may have n jobs each requiring m tasks $T_{1i}, T_{2i}, \ldots, T_{mi}$, $1 \leq i \leq n$, to be performed.Task T_{ji} is to be performed on processorP_j $1 < j < m$. The time requiredto complete task T_{ji} is t_{ji}. A schedule for the n jobs is an assignment of tasksto time intervals on the processors. Task T_{ji} must be assigned to

processorP_j. No processor may have more than one task assigned to it in any timeinterval. Additionally, for any job i the processing of task T_{ji}, j > 1, cannotbe started until task $T_{j-1,i}$ has been completed. For exampleTwo jobs have to be scheduled on three processors, the task times are given by the matrix J

$$J = \begin{bmatrix} 2 & 0 \\ 3 & 3 \\ 5 & 2 \end{bmatrix}$$

Two possible schedules for the jobs are shown in Figure 4.6

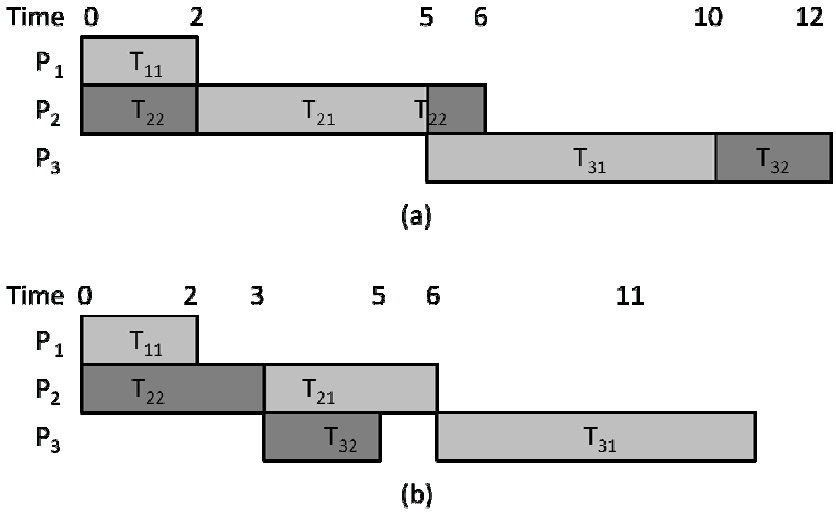


Fig 4.6: Two possible schedules for above example

A non-preemptive schedule is a schedule in which the processing of a task on any processor is not terminated until the task is complete. A schedule for which this need not be true is called preemptive. The schedule of Fig 4.6 (a) is a preemptive schedule. Fig 4.6(b) shows a non-preemptive schedule. The finish time $f_i(S)$ of job i is the time at which all tasks of job i have been completed in schedule S. In Figure 4.6(a), $f_1(S) = 10$ and $f_2(S) = 12$. In Figure 4.6(b), $f_1(s) = 11$ and $f_2(S) = 5$, The finish timeF(S) of a schedule S is given by

$$F(S)= \max_{1 \le i \le n} \{f_i(S)\}$$

The mean flow time MFT(S) is defined to be

$$MFT(S) = \frac{1}{n} \sum_{1 \le i \le n} f_i(S)$$

An optimal finish time (OFT) schedule for a given set of jobs is a non-preemptive schedule S for which F(S) is minimum over all non-preemptiveschedules S. A preemptive optimal finish time (POFT) schedule, optimalmean finish time schedule (OMFT),

and preemptive optimal mean finish(POMFT) schedule are defined in the obvious way.

Although the general problem of obtaining OFT and POFT schedules form > 2 and of obtaining OMFT schedules is computationally difficult, dynamic programming leads to an efficient algorithm to obtainOFT schedules for the case m = 2. In this section we consider this specialcase.

For convenience, we shall use a_i to represent t_{1i}, and b_i to representt$_{2i}$. For the two-processor case, one can readily verify that nothing is tobe gained by using different processing orders on the two processors (this isnot true for m > 2), Hence, a schedule is completely specified by providinga permutation of the jobs. Jobs will be executed on each processor in thisorder. Each task will be started at the earliest possible time. The schedule of Figure 4.7 is completely specified by the permutation (5, 1, 3, 2, 4).We make the simplifying assumption that $a_i \neq 0$, $1 \leq i \leq n$. Note that ifjobs with $a_i = 0$ are allowed, then an optimal schedule can be constructedby first finding an optimal permutation for all jobs with $a_i \neq 0$ and thenadding all jobs with $a_i = 0$ (in any order) in front of this permutation (seethe exercises).

Fig 4.7: A Schedule

It is easy to see that an optimal permutation (schedule) has the propertythat given the first job in the permutation, the remaining permutation isoptimal with respect to the state of the two processors. Let $\sigma_1, \sigma_2,...,\sigma_k$ be a permutation prefix defining a schedule for jobs $T_1, T_2,..., T_k$. For this schedule let f_1 and f_2 be the timesat which the processing of jobs $T_1,T_2,... ,T_k$ is completed on processors $P_1$and P2 respectively. Let $t = f_2 - f_1$. The state of the processors following the sequence of decisions $T_1, T_2,...,T_k$ is completely characterized by t. Letg(S, t) be the length of an optimal schedule for the subset of jobs S underthe assumption that processor 2 is not available until time t. The length ofan optimal schedule for the job set {1, 2,..., n} is g({1, 2,... ,n},0).
Since the principle of optimality holds, we obtain

$$g(\{1, 2, \ldots, n\}, 0) = \min_{1 \leq i \leq n} \{a, + g(\{l, 2, \ldots, n\} - \{i\}, b_i)\} \qquad (4.7)$$

Equation 4.7 generalizes to 4.8 for arbitrary S and t. Thisgeneralization requires that $g(\Phi, t) = \max\{t, 0\}$ and that $a_i \neq 0$, $1 \leq i \leq n$.

$$g(S, t) = \min_{i \in S} \{a_i + g(S - \{i\}, bi + \max\{t - a_i, 0\})\} \qquad (4.8)$$

The term max $\{t - a_i,0\}$ comes into (4.8) as task T_{2i} cannot start until$\max\{a_i,t\}$ (P2 is not available until time t). Hence $f2 - f1 = b_i + \max\{a_i,t\} - a_i = b_i + \max\{t - a_i,0\}$. We can solve for g(S,t) using an approach similarto that used to solve travelling salesman problem.

However, it turns out that (4.8) can be solvedalgebraically and a very simple rule to generate an optimal schedule obtained.

Consider any schedule R for a subset of jobsS, Assume that P_2 is notavailable until time t. Let i and j be the first two jobs in this schedule.Then, from (4.8) we obtain

$g(S,t) = a_i + g(S - \{i\}, b_i + \max \{t-a_i,0\})$
$g(S,t) = a_i + a_j + g(S - \{i,j\},b_j + \max \{b_i + \max \{t - a_i,0\} - a_j,0\})$ (4.9)

Equation 4.9 can be simplified using the following result:

$t_{ij} = b_j + \max \{b_i + \max \{t - a_i, 0\} - a_j, 0\}$
$= b_j + b_i - a_j + \max \{\max \{t - a_i, 0\}, a_j - b_i\}$
$= b_j + b_i - a_j + \max \{t - a_i, a_j - b_i, 0\}$
$t_{ij} = b_j + b_i - a_j - a_i + \max \{t, a_i + a_j - b_i, a_i\}$

If jobs i and j are interchanged in R, then the finish time g'(S,t) is

$g'(S,t) = ai + aj + g(S - \{i,j\},t_{ji})$

where,$t_{ji} = b_j + b_i - a_j - a_i + \max \{t, a_i + a_j - b_j, a_j\}$

Comparing g(S,t) and g'(S,t), we see that if (4.10) below holds, then
$g(S,t) \leq g'(S,t)$.

$\max \{ t, a_i + a_j - b_i, a_i\} \leq \max \{t, a_i + a_j - b_j, a_j\}$ (4.10)

In order for (4.10) to hold for all values of t, we need

$\max \{a_i + a_j - b_i, a_j\} \leq \max \{a_i + a_j - b_j, a_j\}$

$ora_i + a_j + \max \{-b_i, -a_j\} \leq a_i + a_j + \max \{-b_j, -a_j\}$

or $\min \{b_i, a_j\} \geq \min \{b_j, a_i\}$ (4.11)

From (4.11) we can conclude that there exists an optimal schedule inwhich for every pair (i,j) of adjacent jobs, $\min\{b_i, a_j\} \geq \min\{b_j, a_i\}$.Hence, it suffices to generate any schedule for which (4.11) holds for everypair of adjacent jobs. We can obtain a schedule with this property by makingthe following observations from (4.11). If $\min\{a_1, a_2,..., a_n, b_1, b_2,... ,b_n\}$ is a_i, then job i should be the first job in an optimal schedule. If $\min \{a_1, a_2,..., a_n, b_1, b_2,... ,b_n\}$ is b_j, then job j should be the last job in an optimalschedule. This enables us to make a decision as to the positioning of oneof the n jobs. Equation 4.11 can now be used on the remaining n - 1 jobs
to correctly position another job, and so on. The scheduling rule resultingfrom (4.11) is therefore:

1. Sort all the a_i and b_j into nondecreasing order.
2. Consider this sequence in this order. If the next number in the sequence is a_j and job j hasn't yet been scheduled, schedule job j at the leftmostavailable spot. If the next number is b_j and job j

hasn't yet beenscheduled, schedule job j at the rightmost available spot. If j hasalready been scheduled, go to the next number in the sequence.

Note that the above rule also correctly positions jobs with $a_i = 0$. Hence,these jobs need not be considered separately.

CHECK YOUR PROGRESS

2. Sate True or False.

a) 0/1 knapsack problem can also be solved by using greedy strategy.

b) Travelling Salesman problem is to find out the shortest cycle in the graph covering all the vertices

c) 0/1 knapsack does not possess a optimal substructure.

d) Flow shop scheduling problem is to find out the optimal sequence to run n jobs in m processors.

4.9 LET US SUM UP

- A problem can be solved by dynamic programming only when it possesses optimal substructure.
- A problem is said to satisfy the principle of optimality, if the sub solutions of an optimal solution of the problem are themselves optimal solution for their sub problems.
- In dynamic programming wefirst solve the sub-problems and then use these solutions to get the optimal solution in recursive manner.

4.10 FURTHER READINGS

1. T. H. Cormen, C. E. Leiserson, R.L.Rivest, and C. Stein, "Introduction to Algorithms", Second Edition, Prentice Hall of India Pvt. Ltd, 2006.

2. Ellis Horowitz, SartajSahni and SanguthevarRajasekaran, Fundamental of data structure in C, Second Edition, Universities Press, 2009.
3. Alfred V. Aho,John E. Hopcroft and Jeffrey D. Ullman, "The Design and Analysis of Computer Algorithms", Pearson Education, 1999.
4. Ellis Horowitz, SartajSahni and SanguthevarRajasekaran, Computer Algorithms/ C++, Second Edition, Universities Press, 2007.

.

4.11 ANSWERS TO CHECK YOUR PROGRESS

1. a) Falseb)Truec) Falsed) True

1. a) Falseb) Truec) Falsed) True

4.12 PROBABLE QUESTIONS

1. Explain the characteristics of dynamic programming.
2. Describe the steps of dynamic programming algorithm.
3. Solve 0/1 knapsack problem using dynamic programming.
4. Flow shop scheduling algorithm possess the optimal sub-structure, explain it.
5. With an example explain how 0/1 knapsack problem can be solved by using dynamic programming.
6. Describe the method of solving travelling salesman problem using dynamic programming.
7. Explain, what optimal binary search tree is.

UNIT 5 : BACKTRACKING

5.1 LEARNING OBJECTIVE

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

- describe about the concept of backtracking
- know the 8-Queen problem and its solution using backtracking
- elaborate the graph coloring problem
- know the Hamiltonian cycle and its solution using backtracking

5.2 INTRODUCTION

Backtracking is a method for searching a set of solutions or find an optimal solution for satisfies some given constraint to a problem. The name backtrack was first introduce by D. H. Lehmer in 1950's. In this unit we will introduce you the backtracking method. Moreover, we will discuss here about a set of new problems like 8-Queens problem, Graph coloring problem and Hamiltonian Cycle.

5.3 GENERAL STRATEGY FOR BACKTRACKING

In backtracking method the solution set can be represented by an **n** tuple $(x_1,x_2,\ldots\ldots,x_n)$, where x_i are chosen from some finite set S_i. This method can be used for optimization problem to find one or more solution vector that maximize or minimize or satisfy a given criterion function $C(x_1, x_2,\ldots,x_n)$. For example sorting n data of an array **A [1:n]** is a problem to find the solution set, that has an **n**- tuple $(x_1, x_2,\ldots x_i, x_{i+1},\ldots\ldots,x_n)$. For this problem x_i is the index of i^{th} smallest element in the array A .The criterion function is $C(a [x_i] \leq a [x_{i+1}])$ and S_i is the finite set of integers in the range ***[1,n].***

Let the set S_i has size m_i. There are $m = m_1 m_2 \ldots m_n$, **n**- tuples for the solution set that satisfy the criterion function **C**. If brute force approach is applied to find the solution then it will form all n-tuple and solve each on to determine optimal solutions. But if backtracking methods applied here instead of brute force it will take less than m trial to determine the solution. Back tracking method at each step forms a partial solution set $(x_1, x_2,\ldots\ldots,x_i)$ and check it if this has any chances to find a solution depending upon the criterion function C. If this partial solution set no way lead to an optimal solution, then ignore the test vectors from m_{i+1} to m_n entirely.

All the solutions in backtracking require a set of constraints to satisfy. It is divided into two categories:

1. ***Explicit constraint :***

   Explicit constraints are rules that restrict each x_i to take on values only from a given set. This constraint depends on the particular instance I of problem being solved – All tuples that satisfy the explicit constraints define a possible solution space for I.

   Examples of explicit constraints

   i) $x_i \geq 0$, or all nonnegative real numbers

   ii) $x_i = \{0, 1\}$

2. ***Implicit constraints:***

   Implicit constraints are rules that determine which of the tuples in the solution space of I satisfy the criterion function. Implicit constraints describe the way in which the x_i must relate to each other.

There are two types of solution space tuple formulation:

1. ***Variable size tuple:***

   In this method for the solution vector ($x_1, x_2, \ldots, x_k$), x_i will represent indices of i^{th} choices for $1 \leq i \leq k$. Here size of the solution vector can varies for a problem.

2. ***Fixed sized tuple:***

   In this method for solution vector($x_1, x_2, x_3, \ldots x_n$), x_i € { 0,1 } and $1 \leq i \leq n$, such that x_i is 0 if i^{th} element not chosen and 1 otherwise. Here solution vector sizes are same for a problem.

5.4 TREE ORGANIZATION FOR SOLUTION SPACE IN BACKTRACKING

- Backtracking method determine solution of a problem by searching for the solution set in the solution space. This searching can be organized in a tree called ***state space tree.***
- Each node in the tree can be defined as ***problem state.***
- A path from root to any other node defines a ***partial solution vector***, can be called as ***state space.***
- A ***solution state*** is a node ***s*** for which each node from ***root node*** to node ***s*** together can represent a tuple in solution set.
- ***Answer states*** are solution state which satisfies an implicit constraint.
- The ***tree organization*** of ***solution space*** is referred to as ***state space tree.***
- In state space tree problem state are generated from root node and then generated other nodes.
- A ***live node*** is a generated node, for which all of its children node have not yet generated.
- A ***E-node (Expanded node)*** is a live node, whose children are currently being generated.
- A ***dead node*** is that , which is not expanded further and all of its children is generated.
- ***Bounding functions*** are used to bound the searching in the tree. It ***kills a live node without generating children if it does not lead to a feasible solution.***
- Depth first node generated with bounding function is called ***backtracking.***
- In ***state space tree***
    i. ***root*** of the tree represent ***0 choices***.
    ii. ***1^{st} level node*** represents ***1^{st} choices***
    iii. ***2^{nd} level node*** represent ***2^{nd} choices***.
    iv. ***n^{th} level node*** represent ***n^{th} choices***.

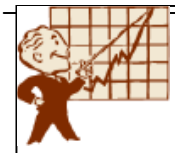
- A node n is called ***non-promising*** if it can not lead to a feasible solution and for this node n bounding function ***B(n) = 0.*** Otherwise, it is called ***promising node*** and bounding function ***B(n)=1.***
- If node is non-promising then it is bounded or kill using bounding function. Then for this node its sub-trees are not generated.
- A state space tree is called ***pruned state space tree*** if it consist of only expanded node .

5.5 MAIN IDEA FOR BACKTRACKING

Backtracking method do depth first search of a state space tree. If a node is promising i.e B(n)=1 then search is continue to its child node , otherwise if a node is non promising i.e B(n)=0 ,backtrack to its parent node.

Recursive algorithm for general backtracking is-

```
Backtrack (node n)
{
    if C(n) = 1
            Report feasible solution n
    else
            Stop
    if (B(n) = 0) return;
    for every child n' of n Backtrack(n')
}
The procedure is invoked by Backtrack(root)
```



CHECK YOUR PROGRESS

1. State True/False:

i. Backtracking method can reduced search space in the state space tree.

ii. Nodes whose children are being generated is called live nodes.

iii. In variable sized tuple method size of solution vector can varies.

iv. Pruned state space tree consist only expanded node.

5.6 8-QUEEN'S PROBLEM

Given a chess board of field 8 x 8. The 8-Queen problem is to place 8-Queen on the chess board, so that no two Queen can "attack" each other. A Queen can attack vertically, horizontally and diagonally.

N-Queen Problem:

It is a generalized problem of 8-Queen problem. N Queens are placed on a chess board of size n x n, without having attack each other.

In chess, queens can move all the way horizontally, vertically or diagonally (if there is no other queen in the way).But, no two Queen can attack each other. So, due to this restriction, each queen must be on a different row and column.

Backtracking strategy for 8-Queen problem is as follows-

1. Let us, in the chess board rows and columns are numbered from 1 to 8 and also queens are numbered from 1 to 8.

2. Without loss of generality, assume that i^{th} queen can be placed in i^{th} row, because no two queen can place in the same row.
3. All solution can represented as 8-tuple (x_1, x_2,…. x_8), where x_i is the column number of the i^{th} row of i^{th} queen placed.
4. Here explicit constraints are S_i = {1, 2, 3, 4, 5, 6, 7, 8}, $1 \le i \le 8$ and the solution space will consist of 8^8 8-tuple.
5. According to the implicit constraints no two queen can on the same row.
6. So, all solution are permutation of 8-tuple(1, 2, 3, 4, 5, 6, 7, 8)
7. Thus the searches is reduce to 8^8 8-tuple to 8! tuple.

We know from the above that, in 8 -Queen problem all the solution can represented as 8-tuple (x_1, x_2,….,x_8), where x_i is the column number of i^{th} row where.i^{th} queen placed. These all x_i's are distinct because of the implicit constraint that no two queen can placed in same column. We assume already in no. 2 that i^{th} queen can be placed in i^{th} row only. So, no two queen can placed in same row. Now, we have only to decide whether two queens are on the same diagonal or not.

If chess board square fields are numbered as two dimensional array a [1:8] [1:8] then we find that for all diagonal element "row-column" value is same. For example if 1^{st} queen is placed in a [1,3] square then the 2^{nd} queen will placed diagonally if it placed in a [2,4] square. Here "row-column" values are 2 and it is same for both queen.

Let two queen are placed in position (m, n) and (x, y)
Then two queens can placed diagonally if

m - n = x - y----------(1)

m + n = x + y----------(2)

(1) => n – y = m - x

(2) => n – y = x – m

Therefore two queens can be on the same diagonal if and only if

| n – y | = | m – x |

This is an another implicit constraint .

Example:

Here is an example of 4-queen problem-

The state space tree generated by 4-queen problem is as follows-

Here node at level i represent i^{th} queen placed at i^{th} row. i.e at level 1 it represents as 1^{st} queen places in 1^{st} row. X_i in i^{th} level represents column number of i^{th} queen placed in i^{th} row. i.e $x_2 = 3$ at level 2 means 2^{nd} queen is in 3^{rd} column of the 2^{nd} row.

Nodes are generated in depth first search manner.

A the path from root to leaf will represent a tuple in solution space. All tuples are distinct and some tuples may not lead to a feasible solution.

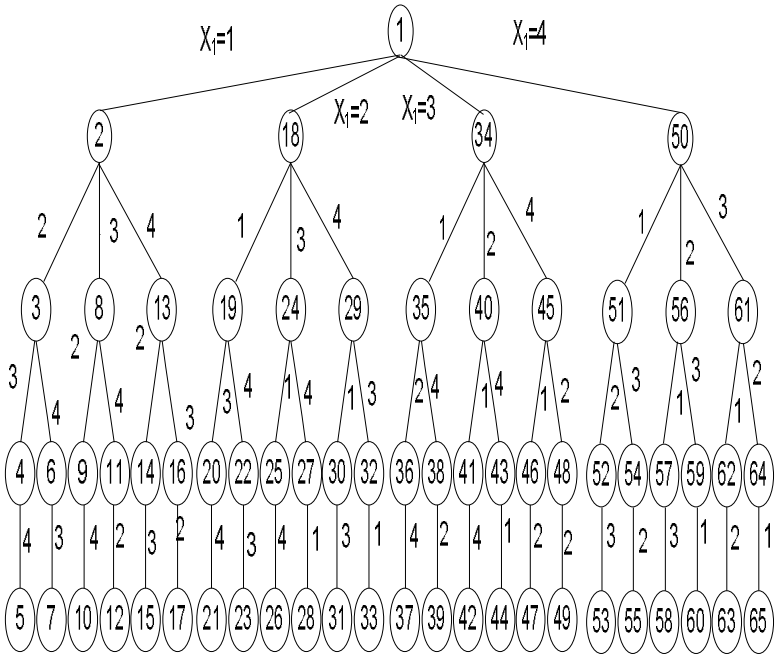


Fig. 5.1 State space tree

Now by using backtracking method we can bound the search of the state space tree using some constraint so that searching require less time.

For this problems to bound a node n constraints or bounding conditions B(n) are-

1. No two queens can place in same row i.e x_i always represents i^{th} queen is in i^{th} row.
2. No two queen in same column i.e values of x_i's are always distinct.
3. For two queen placed in (m, n) and (x, y) position in the chess , value of | n – y | cannot same as | m – x |

When a node is bounded using bounding condition it will not generate any nodes in its sub-tree because nodes in its sub-tree will not give a feasible solution any more.

The portion of pruned state space tree after applying bounding condition is as follows:

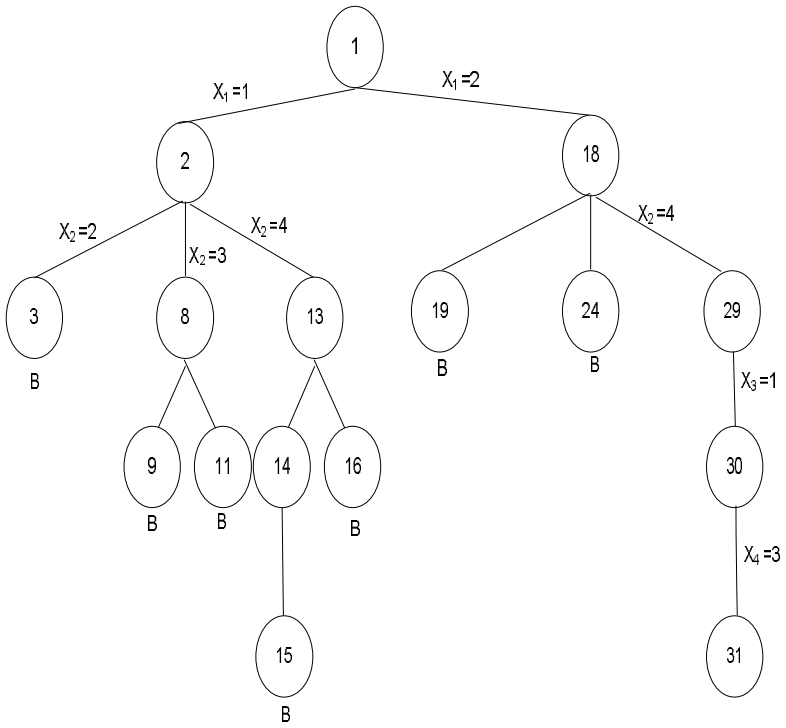


Fig. 5.2 Pruned state space tree

Here node 3 is bounded because-

At level 1, x_1=1 means first time 1st queen is placed in 1st row, 1st column i.e position is (1,1)

At level 2 , x_2=2 means second time 2nd queen is placed in 2nd row, 2nd column i.e position(2,2)

Thus they will place in diagonally. It will violet the implicit constraint or bounding condition. So this combination can not give a feasible solution any more. So, the children of node 3 will not generated further. Hence node 3 will bound.

Here is a path from root 1 to leaf 31 and this will generate one feasible solution set (2, 4, 1, 3) where x_1=2, x_2=4, x_3=1, x_4=3.

Position of the 4 queens are (1,2), (2,4), (3,1) and(4,3) respectively.

A recursive backtracking function for n-Queen problem:

/* placed search for a new queen*/

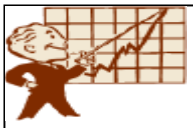
```
bool QPlace ( int k, int i )
  {
    for ( int m = 1; m < k; m++ )
      {
      if (( x [m] == i ) || (abs ( x[m] – i ) == abs ( m – k )))
      return (false);
      return (true);
      }
  }
```

/* Solution to n queen*/

```
void nQueen ( int k,  int i )
{
  for ( int i =1; i ≤ n; i++)
    if ( QPlace( k, i ))
     {
        x [k] = i;
       if ( k == n )
         {
           for ( int m = 1; m ≤ n; m++ )
           {cout << x [m] <<' '<<;
              cout<<endl;
           }
     }
        else
           nQueen ( k + 1, n );
      }
    }
```

Here QPlace (k, i) will return a boolean value true or false. The function return true if k^{th} queen can placed in i^{th} column and assigned it to x [k]. This value x [k] = i is distinct from x [1]…… x [k-1]. It also ensures that no two queen is placed in same diagonal.

Next nQueen (k, n) will solve the n-Queen problem recursively using backtracking method.



. **CHECK YOUR PROGRESS**

2. State True/False:
   i. In n- Queen Problem two Queens can attack each other.
   ii. If two Queens placed diagonally it will violet the implicit constraint.

5.7 GRAPH COLORING PROBLEM

Let G be a graph and m be a given integer. Is there any way to color the vertices of graph G using m color in such a way that no two adjacent vertices have same color? This is called as m-color ability decision problem. According to the graph coloring theory if d is the degree of a given graph, then it can be colored with d +1 color. The minimum number of color required in graph coloring problem to color vertices is called chromatic number. The m-color ability optimization problem is to determine the chromatic number of the graph G. This is called graph coloring problem.

For example graph in the following fig can be colored with minimum 3 colors 1, 2, 3. So, chromatic number is 3.

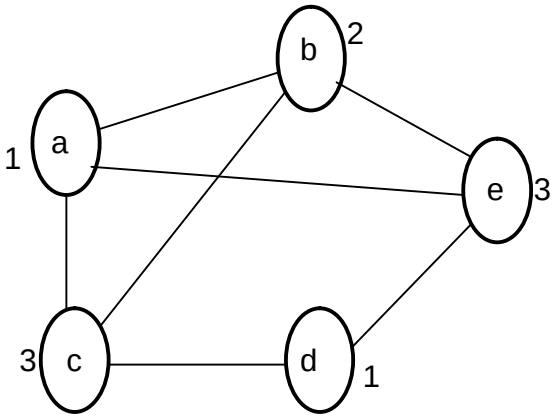


Fig. 5.3 A graph G

Solution using backtracking:

Suppose we represent graph G of n vertices by adjacency matrix G [1:n] [1:n], where G [i] [j] =1,if there is an edge between vertex i and j in G and G [i] [j] =0, otherwise.

The colors of the graph can be numbered from 1 to m. The solution set are represented by (x_1, x_2,…..,x_n), where x_i is the color of vertex i. The state space tree for this problem has degree m and height n + 1. Each node at level i has m children correspond to m color and represents i^{th} vertex of graph G. The left most node has assigned color 1 and rightmost vertex has assigned color m. Node at level n + 1 is leaf.

Example:

G is a graph which has 3 vertices (n = 3) and we have to solve the graph coloring problem for this graph G using 3 color (m = 3)

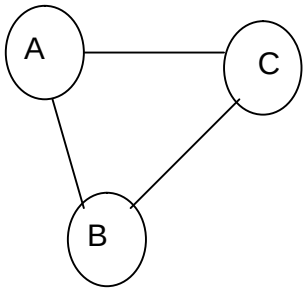


Fig. 5.4 A graph with 3-vertices

The general state space tree for this problem is as follows-

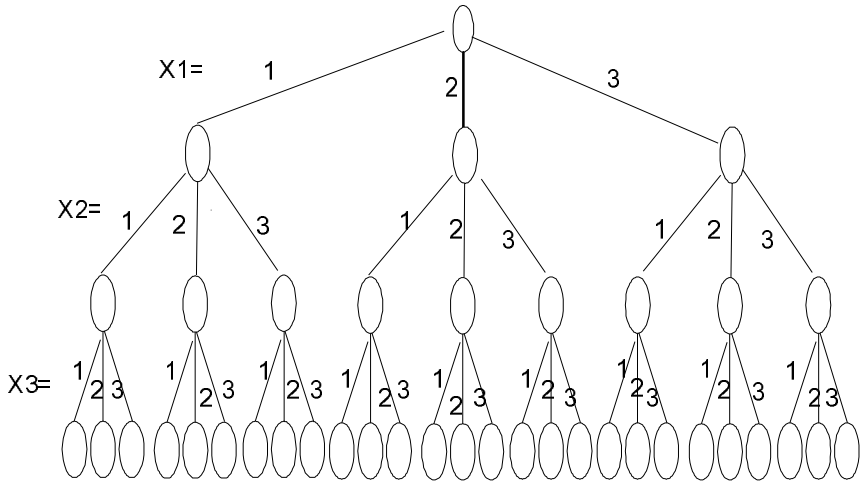


Fig. 5.5 A state space tree

There are possible three color to color vertex A. Hence x_1 has three values, x_1=1, x_1=2, x_1=3.

If we color the vertex A using color 1 then for 2nd vertex B there are three possible colors1, 2,3. Hence, x_2=1,x_2=2,x_2=3 and similarly for vertex C .

Now if we apply backtracking method to solve the problem then it will use a bound in function to kill a node in the state space tree. The bounding function for this problem is that no two adjacent vertexes have same color.

The pruned state space tree after applying bounding function is as follows-

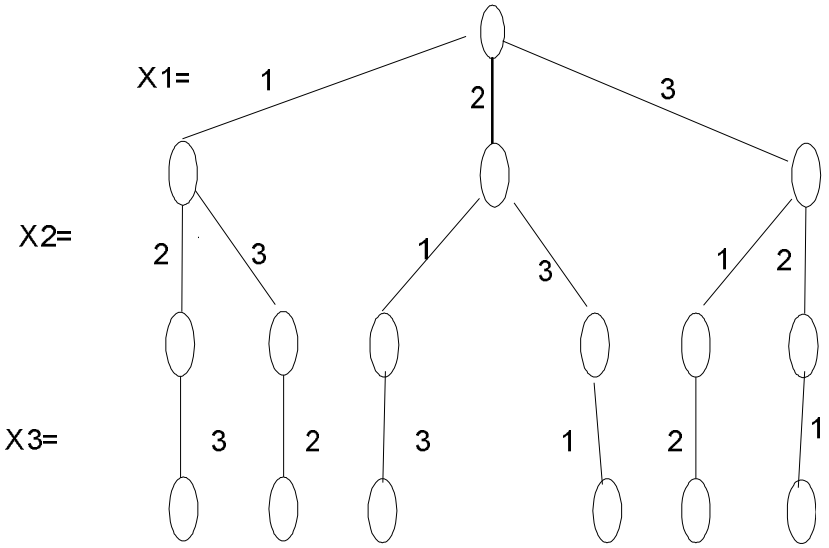


Fig. 5.6 A pruned state space tree

For the graph in fig vertex 1 is the adjacency of vertex 2.So, if we color vertex 1 using color 1 i.e x_1=1 then we cannot color vertex 2 with color 2 i.e x_2=1 because it will violet the bounding condition. Hence node 3 can not lead to a feasible solution and it will bounded. Other nodes are bounded similarly.

Some feasible solution sets are

i) (1,2,3), where x_1=1,x_2=2,x_3=3,

ii) (2,1,3), where x_1=2,x_2=1,x_3=3

iii) (3,2,1), where x_1=3,x_2=2,x_3=1

Here x_i represents i^{th} vertex color value.

***Recursive algorithm for graph coloring problem**:*

/* finding color value of all vertex*/ \

```
void mcoloring ( int k )
{     do
      {
         NextValue ( k );
            if ( ! x [ k ] )  break;
                if ( k == n )
                     {  for ( int i =1; i ≤ n; i++ )
                          cout  <<  x[ i ] ;
                          cout << endl;
                      }
                  else
                     mcoloring ( k + 1 );
 }
  while (1);
  }
```

/* Generating next color */

```
void NextValue ( int k )
{     do
      {
        x [ k ] = x [ k + 1] % ( m + 1 );
          if ( ! x [ k ] ) return;
            for ( int j = 1; j ≤ n; j++ )
                  {
                     if (G [ k ] [ j ] && (x [ k ] == x [ j ] ))
                       break;
                   }
              if ( j == n + 1 ) return;
          }
           while(1);
   }
```

Here k is the next vertex to color

n is the number of vertices

x [i] is the color of i^{th} vertex.

The function mcoloring first create the adjacency matrix G [i] [j] of the graph G and then initialize x [k] = 0, for all $1 \leq k \leq n$. First invoke the procedure by mcoloring (1). Function NextValue produces possible color for vertex k and assign it to x_k. Function mcoloring repeatedly picks a color value and assign it to x_k and then calls mcoloring recursively.

CHECK YOUR PROGRESS

3. What is the decision problem of graph coloring?
4. What is the chromatic number of a graph?

5.8 HAMILTONIAN CYCLE

A Hamiltonian cycle of a connected undirected graph with n vertices is a cyclic path along n edges, such that each vertex visits once in graph G and return to the starting vertex. It is named after William Hamilton.

Example :

Following is graph G with 8 vertices.

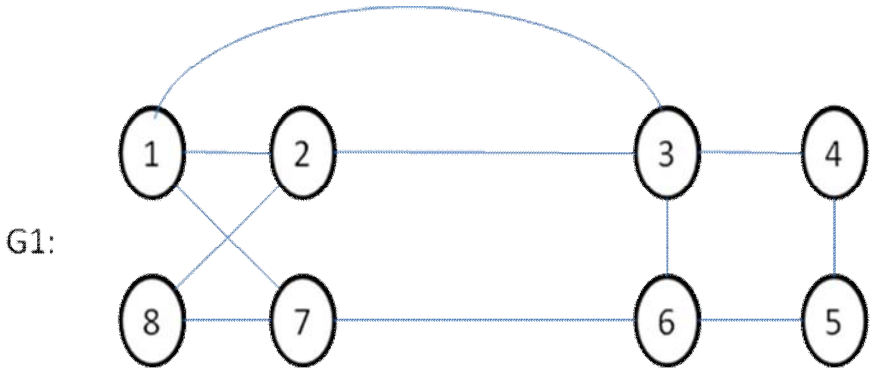


Fig. 5.7 A graph with 8 vertices

The Hamiltonian cycle of this graph is- 1, 2, 8, 7, 6, 5, 4, 3, 1

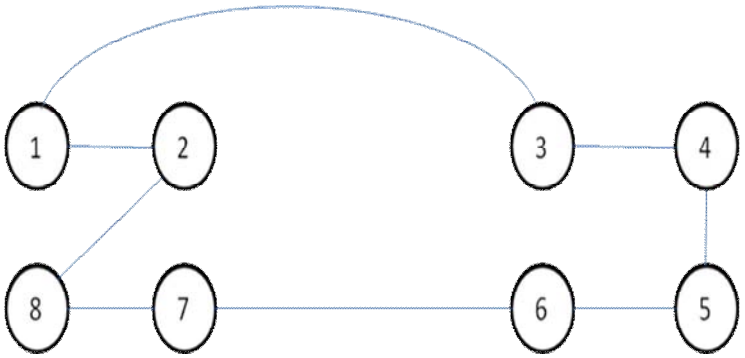


Fig. 5.8 Hamiltonian cycle

Backtracking method for Hamiltonian cycle:

Now, using backtracking method we can find out the Hamiltonian cycles in a graph which has n vertices. The solution set can represented as (x_1, x_2,……,x_n), where $1 \leq i \leq n$ and x_i represents the i^{th} visited vertex of the current considered cycle.

We have to determine value of x_i i.e possible vertex to select. For i =1, x_1 can be any vertex chooses from n vertex. To determine value of x_i we have already determine x_1, x_2,……..,x_{i-1}. Hence, the x_i can be choose as

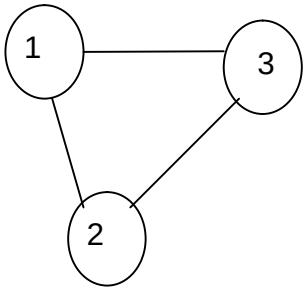
i) any vertex v which is not assigned to x_1, x_2…..and x_{i-1} from the n vertices.

ii) v is connected by an edge to x_{i-1}

The last vertex x_n must be connected to both x_{n-1} and x_1.

Example:

Consider the following graph and find out all the Hamiltonian cycle.



The state space tree for the graph is-

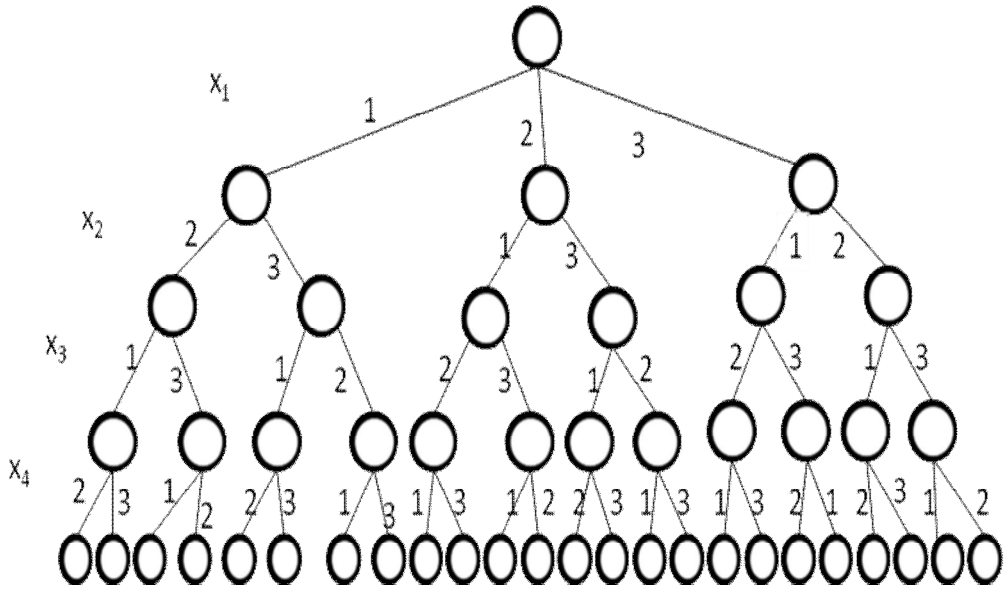


Fig. 5.9 State space tree

For using backtracking the bounding functions are-

i) The solution vector (x_1, x_2,……x_n) is defined such that value of x_i's are distinct, for all $1 \leq i \leq n$, because one vertex visit only once.

ii) There is an edge between (x_{i-1}, x_i)and (x_i, x_{i+1})

iii) There is an edge between x_n and x_1, (for Hamiltonian cycle).

Now, the pruned state space tree is as follows-

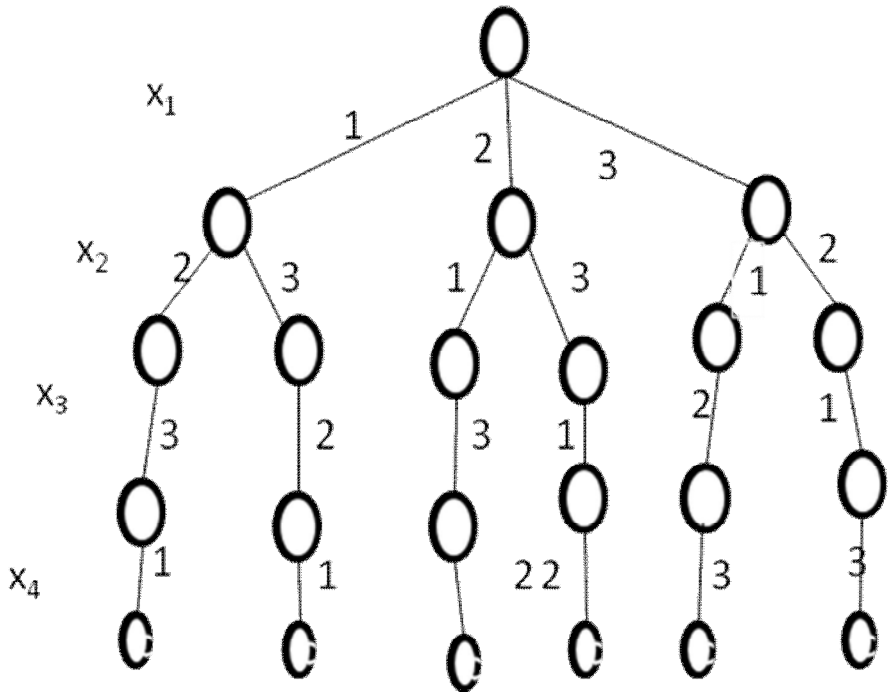


Fig. 5.10 Pruned state space tree

Recursive function for Hamiltonian cycle:

/* for finding Hamiltonian cycle*/

```
void Hamiltonian ( int k )
{
    do
     {
          NextValue ( k );
             if ( ! x [ k ] )  return;
```

```
            if ( k == n )
               {
                 for ( int i =1; i ≤ n; i++)
                     cout << x [ i ] <<" "<<"\n";
                }
             else
                 Hamiltonian ( k + 1 )
        }
    while(1);
}
```

/* generating next vertex*/

```
void NextValue ( int k )
    {
       do
          {
             x [ k ] = ( x [ k ] + 1 ) % ( n + 1 );          //Next vertex
              if ( ! x [ k ] ) return;
              if ( G [ x [ k – 1 ] [ x [ k ] ])
                {
                   for ( int j = 1; j ≤ k - 1; j++)
                      if ( x [ j ] == x [ k ] )  break ;
                if ( j == k )
                if ( ( k < n ) || (( k == n ) && G [ x [ n ] ] [ x [ 1] ] ))
                return;
             }
         }
   while(1);
 }
```

This program first initializes the adjacency matrix G [1:n] [1:n] and x [1] = 1 and x [2:n] = 0.

Hamiltonian function is invoked by Hamiltonian (2).

5.9 BACKTRACKING METHOD FOR 0-1 KNAPSACK PROBLEM

0-1knapsack problem :

Given n items, for which weight set is W = (w_1, w_2,….,w_n), where w_i is positive weight of i^{th} item. There are n profits (p_1, p_2,….,p_n), where p_i represents profit of i^{th} item and capacity of knapsack is m, m > 0.

The 0-1 knapsack problem chooses subset of weight set W such that

$$\sum_{1 \le i \le n} w_i x_i \le m \quad \text{and} \quad \sum_{1 \le i \le n} p_i x_i \text{ is maximized}$$

Where x_i € { 0, 1 }

i.e i^{th} item is selected then x_i = 1,

= 0, Otherwise

There are two tree organizations possible for this problem. Variable tuple sized formulation and fixed tuple sized formulation. In solution vector x_i can be assigned with 0 or 1 in 2^n distinct ways.

Here bounding function is that total profit of the chooses item is maximized and total weight of chooses item is atmost m .

This bounding function for a live node can be obtained by using an upper bound on the value of the best feasible solution. If a upper

bound for a live node is not higher than the value of the best solution then the node can bounded or kill.

Here we consider fixed tuple sized formulation. For a node S at level k+1, the value of x_i, $1 \le i \le k$ has already determined. The upper bound for a node S can be obtained by making $x_i = 0$ or 1 for $k + 1 \le i \le n$.

A recursive function for 0-1 knapsack using backtracking:

/* bounding function for 0-1 knapsack*/

```
float Bound ( float cp, float cw, int k )
{
    float b = cp, c = cw;
     for ( int i = k + 1; i ≤ n; i++ )
     c = c + w [ i ];
      if ( c < m )
       b = b + p [ i ];
       else

 return ( b + ( 1 - ( c – m ) / w [ i ] * p [ i ] );
    }
 return(b);
}
```

/* Backtracking method for 0-1 knapsack*/

```
void Knap( int k, float cp, float cw)
{
   if ( cw + w [ k ] ≤ m )
      y [ k ] = 1;
        if ( k < n )
          Knap ( k + 1, cp + p [ k ], cw + w [ k ] );
```

```
        if (( cp + p [ k ] > fp ) && ( k == n ))
          {
            fp = cp + p [ k ];
            fw = cw + w [ k ];
              for ( int j = 1; j ≤ k ; j++)
                x [ j ] = y [ j ];
            }
    }
if ( Bound (cp, cw, k ) ≥ fp )
  {
     y [ k ] = 0;
      if ( k < n )
        Knap ( k + 1, cp, cw );
          if (( cp > fp ) &&( k == n ))
            {
               fp = cp;
               fw = cw ;
                 for ( int j = 1; j ≤ k; j++)

          x [ j ] = y [ j ];
              }
            }
        }
```

Here

cp = current total profit of the chosen items,

cw = current total weight of all chosen items.

k = index of last considered item .

m = capacity of knapsack

w[i] = weight of ith item.

p[i] = profit of ith item.

P[i]/w[i] ≥ p[i+1]/w[i+1], for all 1≤i<n

n= total item numbers

fw= final total weights in knapsack
fp= final maximum profit
x[k]==0, if w[k] is not in knapsack,
== 1 ,Otherwise.

The above method to determine an upper bound for a node at level k+1 of state space tree, function Bound(cp, cw, k) is used.

Initially fp is set to -1. This method s invoked by Knap(1,0,0). When fp != -1 , x[k],1≤k<n , is such that $\sum_{i=1..n} p[i]\, x[i] = fp$.

The path y[j],1 ≤ j ≤ k is the path to the current node.
The current weight $cw = \sum_{i=1..k-1} w[i]\, y[i]$

The current profit $cp = \sum_{i=1..k-1} p[i]\, y[i]$

CHECK YOUR PROGRESS

5. What is Hamiltonian cycle?

5.10 LET US SUM UP

- Backtracking is a method for searching a set of solutions or find an optimal solution for satisfy some given constraint to a problem

- In backtracking method the solution set can be represented by an n tuple (x_1,x_2,…….,x_n), where x_i are chosen from some finite set S_i.
- Backtracking method can be used for optimization problem to find one or more solution vector that maximize or minimize or satisfy a given criterion function
- In backtracking constraint to be satisfied can be divided into two categories- Implicit constraint and Explicit constraint
- Two types of tuple formulation- Variable size tuple and Fixed sized tuple
- In backtracking method searching can be organized in a tree called state space tree.
- A solution state is a node s for which each node from root node to nodes together can represent a tuple in solution set.
- A live node is a generated node, for which all of its children node have not yet generated.
- A E-node (Expanded node) is a live node, whose children are currently being generated.
- A dead node is that , which is not expanded further and all of its children is generated
- Bounding functions are used to bound the searching in the tree.
- A node n is called non-promising if it can not lead to a feasible solution. Otherwise, it is called promising node
- A state space tree is called pruned state space tree if it consist of only expanded node
- Backtracking method do depth first search of a state space tree.
- It is a generalized problem of 8-Queen problem. N Queens are placed on a chess board of size n x n, without having attack each other.
- According to the graph coloring theory if d is the degree of a given graph, then it can be colored with d +1
- The minimum number of color required in graph coloring problem to color vertices is called chromatic number

- A Hamiltonian cycle of a connected undirected graph with n vertices is a cyclic path along n edges, such that each vertex visits once in graph G and return to the starting vertex

5.11 ANSWERS TO CHECK YOUR PROGRESS

1. i. True, ii. False, iii. True, iv. True
2. i. False, ii. True
3. Let G be a graph and m be a given integer. Is there any way to color the vertices of graph G using m color in such a way that no two adjacent vertices have same color. This is called as m-color ability decision problem.
4. The minimum number of color required in graph coloring problem to color vertices is called chromatic number.
5. A Hamiltonian cycle of a connected undirected graph with n vertices is a cyclic path along n edges, such that each vertex visits once in graph G and return to the starting vertex. It is named after William Hamilton.

5.12 FURTHER READINGS

Ellis Horowitz, Sartaj Sahni and Sanguthevar Rajasekaran, Computer Algorithms/ C++, Second Edition, Universities Press, 2007.

5.13 MODEL QUESTIONS

1. What is backtracking method?
2. Write about state space tree organization of backtracking method.

3. What is 8-queen problem? How can it solved using backtracking?
4. What is the bounding condition for n-queen problem?
5. What s graph coloring problem? What is the bounding condition for graph coloring problem?
6. How does backtracking method find Hamiltonian cycle in a graph?

UNIT - 6:BRANCH AND BOUND

UNIT STRUCTURE

6.1 LEARNINGOBJECTIVES

After goingthrough this unit, you will be able to:

- understand the concept of Branch and Bound
- solve 0/1 knapsack problem by this method
- solve travelling salesman problem by this method

6.2 INTRODUCTION

We have already covered the most important techniques such as backtracking, greedy strategy, divide and conquer, dynamic programming etc. In this chapter, we will discuss about the branch and bound technique.

Branch and bound is an algorithm design technique that is often implemented for finding the optimal solutions in case of optimization problems; it is mainly used for combinational and discrete global optimizations of problems. In a nutshell, we opt this technique when the domain of possible candidates is way too large and all of the other algorithms fail.

6.3 GENERAL STRATEGY

Out of the techniques that we have learned both the backtracking and divide and conquer traverse the decision tree in the depth first order, though they take opposite routes. The greedy strategy picks a single route and forgets about the rest. Dynamic programming approach is most likely similar to breadth-first search.

Now if the decision tree of the problem that we are planning to solve has practically unlimited depth, then backtracking and divide and conquer algorithms cannot be used. Also we shouldn't rely on greedy because that is problem-dependent and never promises to deliver a global optimum, unless we prove it mathematically.

As our last resort we may even think about dynamic programming. The truth is that maybe the problem can indeed be solved with dynamic programming, but the implementation wouldn't be an efficient one. Additionally, it would be very hard to implement. If we have a complex problem where we need lots of parameters to describe the solutions of sub-problems, dynamic programming becomes inefficient.

In backtracking, we used depth-first search with pruning to traverse the (virtual) state space. Breadth first search with pruning can also be used for better performance in some problems. This process of using breadth first search with pruning is known as branch and bound. In breadth-first search, queue is used as an auxiliary data structure.

The drawback of breadth first search is that the complexity of it is exponential. We need an algorithm that ameliorates this issue by reducing some candidates which arenot satisfactory and, won't contribute to the optimal solution. Branch and bound is a such type of technique. Branch and bound algorithm injects some intelligence into the naive but complex breadth-first search. Instead of searching throughout the entire decision/search tree structure, it instills some sort of criteria, according to which the complexity of the breadth-first search can be reduced. For example, if we calculate the distance of each node in terms of "how far" it is located from the initial root node and "how close" it is from the solution, then the distance/cost is the sum of these two distances. However, as we can surely see, the second distance relies on heuristics. Thus, it's just a guess.Moreover, we can move through this tree based on the instilled costs, a node is a more possible candidate toward the solution if its cost is less than the other nodes. What we did here is, we add an essence of depth-search to the breath-first meaning.Here we are going to maintain a priority queue, according to which we are going to runthe breath-first search. The traditional breath first search runs from left to right, but now it will run according to the priority queue.

The beauty of this approach is that we haven't lost the not-so-possible candidates, they will be stored somewhere on the end of the priority queue, so this algorithm doesn't neglect the rest of the possible options. Summing these up, this is in fact a typical branch and bound algorithm techniqueand it relies on the guess.

Branch and bound is composed of two main actions.Firststep,*branching,* where we define the tree structure from the set of candidates in a recursive manner. The second step is*bounding,*where we calculates the upper and lower bounds of each node from the tree.Furthermore, there is a additional *pruning* step,where depending on the values of upper bound and lower bound some node can be discarded from the search.

All in all, branch and bound is very similar to backtracking. The main differences are that the branch and bound is used only in case of optimization problems, whereas backtracking cannot be, and branch and bound doesn't limit us to a particular way of traversing the tree. Backtracking always picks one single successor from the candidates, while branch and bound always has the entire list of successors in the queue.

A branch and bound algorithm is based on an advanced breadth-first search, where breath- first search is performed with the help of apriority queue instead of the traditional list. Inbranch and bound it is crucial to understand the importance of two functions: g(x) and h(x). The first function, *g(x)*, calculates the distance between the *x* node and the root node. Whereas, *h(x)*, is a heuristic function because it estimates how close the *x* node to the solution.Moreover, we can say that **f(x) = g(x) + h(x)**. The g(x) part is the path-cost function, while the h(x) part is the admissible heuristic estimate; the sum of these two is the f(x).

6.40/1 KNAPSACK PROBLEM

We are already familiar with the knapsack problem in unit 3. Let us consider a knapsack of size **K** and we want to select a set of objects from***n*** objects , where the i^{th} object has size s_i and value $v_i$such that it maximizesthe value contained in the knapsack with the contents of the knapsack less than or equal to K.

Suppose that K = 16 and n = 4, and we have the following set of objects ordered by their value density.

i	v_i	s_i	v_i/s_i
1	$45	3	$15
2	$30	5	$ 6
3	$45	9	$ 5
4	$10	5	$ 2

We will construct the state space where each node contains the total current value in the knapsack, the total current size of the contents of the knapsack, and maximum potential value that the knapsack can hold. In the algorithm, we will also keep a record of the maximum value of any node found so far. When we perform the depth-first traversal of the state-space tree, a node is "promising" if its maximum potential value is greater than this current best value.

We begin the state space tree with the root consisting of the empty knapsack. The current weight and value are obviously 0. To find

the maximum potential value we treat the problem as if it were the fractional knapsack problem and we were using the greedy algorithmic solution to that problem. We have already discuss that the greedy approach to the fractional knapsack problem yields an optimal solution. We place each of the remaining objects, into the knapsack until the next selected object is too big to fit into the knapsack. We then use the fractional amount of that object that could be placed in the knapsack to determine the maximum potential value.

totalSize = currentSize + size of remaining objects that can be fully placed

bound (maximum potential value)
= currentValue + value of remaining objects fully placed +
(K - totalSize) * (value density of item i.e partially placed)

In general, for a node at level i in the state space tree the first i items have been considered for selection and for the k^{th} object as the one that will not completely fit into the remaining space in the knapsack, these formulae can be written:

$$\text{totalSize} = \text{currentSize} + \sum_{j=j+1}^{k-1} s_j$$

$$\text{bound} = \text{currentValue} + \sum_{j=j+1}^{k-1} v_j + (K - \text{totalSize}) * (v_k/s_k)$$

For the root node currentSize = 0 and currentValue = 0

$$\text{totalSize} = 0 + s_1 + s_2 = 0 + 3 + 5 = 8$$

$$\begin{aligned}\text{bound} &= 0 + v_1 + v_2 + (K - \text{totalSize}) * (v_3/s_3)\\ &= 0 + \$45 + \$30 + (16 - 8) * (\$5)\\ &= \$75 + \$40\\ &= \$115\end{aligned}$$

The computation of the bound and the selection criteria for promising nodes is the same as before. We must replace the depth-first traversal of the state space tree with a breadth first traversal. In the depth-first traversal the auxiliary data structure used to store the nodes was the stack. In breath-first traversal, the auxiliary data structure is explicitly the queue.

6.5TRAVELLING SALESMAN PROBLEM

Instead of using a Queue to perform a breadth-first traversal of the state space, we will use a PriorityQueue and perform a "best-first" traversal. For the TSP we first compute the minimum possible tour by finding the minimum edge exiting each vertex. The sum of these edges may not form a possible tour, but

since every vertex must be visited once and only once, every vertex must be exited once. Therefore, no tour can be shorter than the sum of these minimum edges.

At each subsequent node, the lower bound for a "tour in progress" is the length of the tour to that point plus the sum of the minimum edge exiting the end vertex of the partial tour and each of the minimum edges leaving all of the remaining unvisited vertices. If this bound is less than the current minimum tour, the node is "promising" and the node is added to the queue. Initially the minTour is set to infinity. When a node whose path includes all of the vertices except one is reviewed, there is only one possible way for the tour to complete. The remaining vertex and the first are added to the path and the length of the tour is the current length plus the length of the edge to the remaining vertex and the length of the edge from there back to the starting vertex. If this tour length is better than the current minimum, it becomes the minimum tour length. Once a first complete tour is discovered, nodes whose bound is greater than or equal to this minTour are deemed "non-promising" and are pruned.

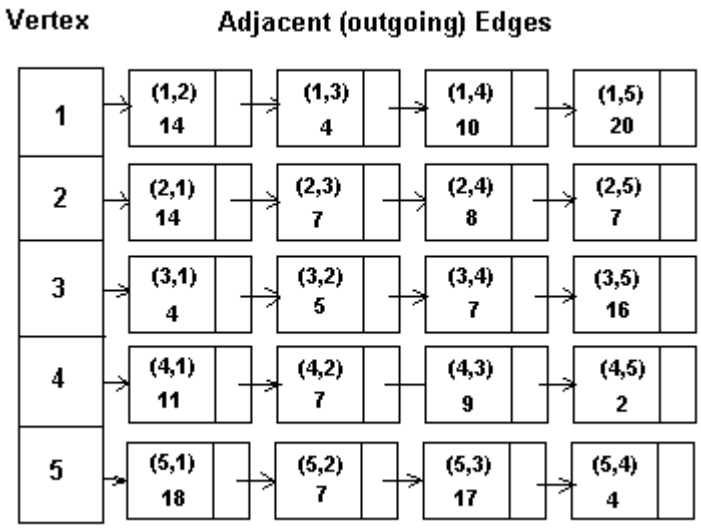
The nodes in state space must carry the following information:

- their level in the state space tree
- the length of the partial tour
- the path of the partial tour
- the bound
- (for efficiency) the last vertex in the partial tour

In a branch and bound algorithm, a node is judged to be promising before it is placed in the queue and tested again after it is removed from the queue. If a lower minTour is discovered during the time a node is in the queue, it may no longer be promising after it is removed, and it is discarded. Using a Priority Queue, the search traverses the state space tree in neither a breadth-first nor depth-first fashion, but alternates between the two approaches in a greedy, opportunistic fashion. In the example problem below, a diagram of the best-first traversal of the state space indicates by number when each of the nodes is removed from the priority queue.

Example

Let G be a fully connected directed graph containing five vertices that is represented by the following adjacency list:



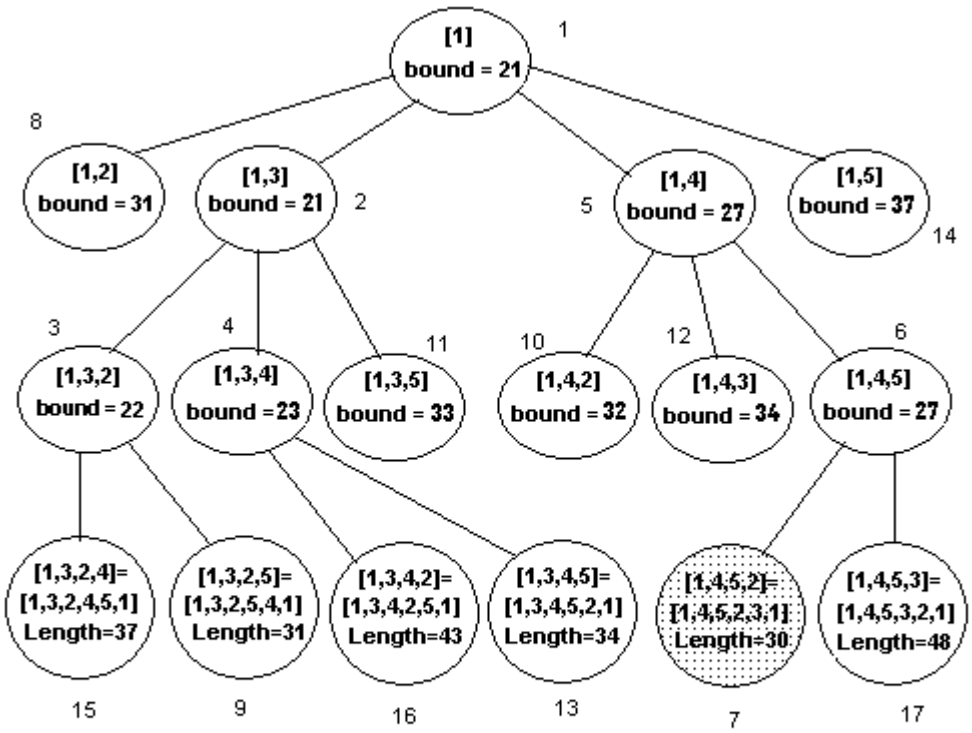
We assume in the implementation of this algorithm that vertices are labeled by an integer number and edges contain the source and sink vertices and a cost or length label. The tour will start at vertex 1, and the initial bound for the minimum tour is the sum of the minimum outgoing edges from each vertex.

Vertex 1	min (14, 4, 10, 20)	= 4
Vertex 2	min (14, 7, 8, 7)	= 7
Vertex 3	min (4, 5, 7, 16)	= 4
Vertex 4	min (11, 7, 9, 2)	= 2
Vertex 5	min (18, 7, 17, 4)	= 4
bound [1]		= 21

Since the bound for this node (21) is less than the initial minTour (□), nodes for all of the adjacent vertices are added to the state space tree at level 1. The bound for the node for the partial tour from 1 to 2 is determined to be:

bound = length from 1 to 2 + sum of min outgoing edges forvertices 2 to 5
= 14 + (7 + 4 + 2 + 4)
= 31

After each new node is added to the PriorityQueue, the node with the best bound is removed and similarly processed. The algorithm terminates when the queue is empty.



l

second node placed in the priority queue, but the 8th node to be removed. By the time it is removed and examined, a tour of length 30 which turns out to be the optimal tour, has already been discovered, and, since its bound exceeds this length, it is discarded without having to check any of the possible tours that extend it.

Here is a Branch and Bound algorithm for an adjacency list representation of a graph. If the first vertex is numbered 1 instead of 0, the array bounds for ***mark*** and ***minEdge*** would have to be length N + 1 and the loops traversing these arrays would have to be from 0 to N.

CHECK YOUR PROGRESS

1. State True or False.

a) Branch and bound technique is based on advanceddepth first traversal.

b) Branch and bound techniques uses priority queue data structure.

c) In pruning step of branch and bound technique, it removes the not so promising nodes from the search space.

d) All types of problems can be solved by using branch and bound technique.

6.6 LET US SUM UP

- Branch and Bound is a state space search method in which all the children of a node are generated before expanding any of its children.
- It is similar to backtracking technique but uses BFS-like search.
- Branch and bound techniques uses the priority queue data structure for storing the information
- Branch and bound technique mainly based on the value g(x) + h(x), where g(x) is the distance from the root to the current vertex and h(x) is a heuristic function.

6.7 FURTHER READINGS

1. T. H. Cormen, C. E. Leiserson, R.L.Rivest, and C. Stein, "Introduction to Algorithms", Second Edition, Prentice Hall of India Pvt. Ltd, 2006.
2. Ellis Horowitz, Sartaj Sahni and Sanguthevar Rajasekaran, Fundamental of data structure in C, Second Edition, Universities Press, 2009.
3. Alfred V. Aho,John E. Hopcroft and Jeffrey D. Ullman, "The Design and Analysis of Computer Algorithms",Pearson Education, 1999.
4. Ellis Horowitz, Sartaj Sahni and Sanguthevar Rajasekaran, Computer Algorithms/ C++, Second Edition, Universities Press, 2007.

.

6.8 ANSWERS TO CHECK YOUR PROGRESS

1. a) False b) True c) True d) False

6.9MODEL QUESTIONS

1. Discuss the branch and bound technique. In which situations we use this technique?
2. Explain the branch and bound technique to solve Travelling salesman problem.
3. With an help of example show how 0/1 knapsack problem can be solved by using branch and bound technique.

*****

Unit 7: P, NP-HARD and NP-COMPLETE PROBLEM

UNIT STRUCTURE

7.1 LEARNING OBJECTIVE

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

- learn about the P and NP hard problems
- know the NP complete problems
- describe the non-deterministic algorithms
- elaborate the Cook's theorem

7.2 INTRODUCTION

In this unit you will learn about non-deterministic polynomial time algorithm. There are two types of problem. One can be solve in polynomial time and other for which no polynomial time algorithm known. There are some problem that cannot solved in polynomial time till now such as Turing's "Halting problem". Here, we will discuss about these algorithm.

7.3 BASIC CONCEPTS OF P AND NP PROBLEMS

Polynomial time algorithms have running time (n^k), for some constraint k, on input of size n. Not all the problems can solve in polynomial time. For example -a sorting problem can solve in O(nlogn) polynomial time. But the problems like knapsack problem can solved in $O(2^{n/2})$ non- polynomial time .

There are two types of problem.

i) ***Decision problem***: Decision problems are problems that has 'yes' or 'no' answers i.e answers are decidable. For example: Given an edge-weighted graph G and a positive integer k, does G contains a spanning tree with total weight $\leq$ k? An algorithm for a decision problem is termed as decision algorithm.

ii) ***Optimization problem***: These types of problem finds the best solution to a problem from all feasible solution set. This type of problem finds the optimal (maximum or minimum) solution for a problem. For example: Given an edge-weighted graph G, find a spanning tree with minimum total weight. An algorithm for an optimization problem is called optimization algorithm.

NP Class:

When 'yes' instances of a decision problem can decided in polynomial time then it is called a NP class problem. NP stands for nondeterministic-polynomial time. For example in a Hamiltonian cycle problem decision problem is- given a directed graph G(V,E) of n vertices, where $v=<v_1,v_2,\ldots\ldots,v_n>$. Is there any Hamiltonian cycle for the graph G? Hamiltonian cycle is a cycle passing every vertex exactly once. It can check in polynomial time whether there is an edge between (v_i,v_{i+1}) for $1\leq i \leq n$ and (v_n,v_1).

P-Class:

Problems that are solvable in polynomial time is called class P problem.

We refer here two classes of NP problem:

i) NP-Complete
ii) NP-Hard

A problem that is NP-Complete has the property that it can be solved in polynomial time if and only if all other NP-Complete problem can solved in polynomial time.

A NP-Hard problem has the property that if it can solved in polynomial time then all NP-Complete problem can solved in polynomial time.

Hence all NP-Complete problems are NP-Hard, but all NP-Hard problems may not NP-Complete.

A decision problem is NP-complete if its corresponding language is NP-complete.
An optimization problem is NP-hard if its decision version is NP-hard.

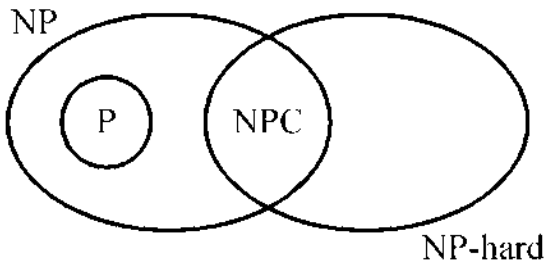


Fig:7.1 P,NP,NP-Complete, NP-Hard

CHECK YOUR PROGRESS

1. True/False
   i. Answers of a decision problem is decidable.
   ii. Optimization problem does not find optimal solution.
   iii. A decision problem is NP-complete if its corresponding language is NP-complete.

7.4 NON-DETERMINISTIC ALGORITHM

There are two types of algorithm:

i***) Deterministic algorithm:***

Algorithm where every operation uniquely defines is called deterministic algorithm. In such algorithm steps of the algorithm and way of program execution in computer is compatible.

ii) ***Non-deterministic algorithm:***

Algorithm whose outcomes are not uniquely defined but are limited to a specified possibility is called non-deterministic algorithm. When such operations are executing it is allowed to choose any

one of these outcomes depending upon a termination condition defined later.

If there exists no choices that will lead to success, then the non deterministic algorithm terminates unsuccessfully. A machine which can execute a non deterministic algorithm is called non deterministic machine. But in fact non-deterministic machine does not exist.

Following are three functions for non-deterministic algorithm-

i) ***Choice(S):*** Choose any one element from set S

ii) ***Failure( ):*** Results an unsuccessful completion

iii) ***Success( ):*** Results a successful completion

The statement a=Choice(1,n), assigns an integer in the range[1:n] to a. There is no rule to the integer that is assigned to a. The Failure() and Success() signal define the computation of the algorithm.

Example:

Consider the problem of searching an element m in a array A[n] of n element. We have to find the array index i for the element m.

Solution: A non deterministic algorithm is as below using the above three function-

```
int i= Choice(1,n);
if (A[i]=m)
     {
       cout<<i;
        Success();
      }
cout<<'0';
Failure();
```

In the above program first the Choice (1: n) pick up a integer between 1 to n and assign it to i. Then it checks if m is in A[i]. If success output will i, index of the m in A[]. If m is not in a[i] then it output number 0. Hence the non-deterministic complexity of this program is O(1). But for this same problem the deterministic algorithm complexity is Ω(n).

Example: Sort n positive element of an array a[i], 1≤i≤n using non-deterministic algorithm.

Solution: A non deterministic algorithm for this problem is –

```
void Sort(int A[ ],int n)
{
   Temp[n], i, m;
    for (i=1; i≤n; i++)        /* for loop 1*/
       Temp[i]=0;

     for (i=1; i≤n; i++)        /* for loop 2*/
       {
         m = Choice(1,n);
         if ( Temp[j])  Failure ();
           Temp[j]=A[i];
         }

    for( i=1; i<=n; i++)        /* for loop 3*/
       If( Temp[i] > Temp[i+1]) Failure();

    for(i=1; i<=n; i++)        /* for loop 4*/
       cout<< Temp[i]<<" ";
    Success();
   }
```

This non deterministic algorithm can sort elements of an array . Another array Temp[ ] is used for keep sorted element. Initially Temp[ ] initializes to 0. Within the for loop 2 ach A[i] is assigned to a location in Temp[ ]. first choose an index m of array A[ ] non-deterministically using Choice(1,n) function. Next it confirmed that Temp[m] is already not in used. Otherwise Failure(). In loop3 it will verifies that Temp is sorted . In loop 4 it will output the sorted data. Algorithm is successfully completed if the output numbers are sorted.

In theory non-deterministic algorithm works as below -

The algorithm makes several copy depending upon possible number of choices. One copy is made for each choice. If the first copy successfully complete then all other copy of choices are terminated. Otherwise, if a copy gets failure then terminates only that copy of the algorithm. A non deterministic machine does not follow all these times. Instead it can select a correct choice, every time it is made. If a correct element chooses then each time it reduces the size of the possible choices set, which can make a successful termination of the algorithm. If termination is unsuccessful then output as “unsuccessful”.

Here we discussed only the non-deterministic algorithm that has unique output. These are non-deterministic decision algorithms . When the outputs 1 a successful completion occur and if the output is 0 then there is no choices to successful completion. Thus, output of a decision algorithm is uniquely defined by input parameters and algorithm specification.

The property to makes decision problem version for an optimization problem is that if the optimization problem can solved in polynomial time then the corresponding decision problem can solved in polynomial time. In other words if the decision problem

can not solve in polynomial time then the corresponding optimization problem can not solve in polynomial time.

Example: **Clique**

A clique G'(v,e) is a maximal complete sub-graph of a graph G(V,E) and size of the clique is |v| . The optimization problem is called max clique problem which finds the size of the larges clique. The decision problem is that- Is there exist any clique G' in G that has size k, for some given k?

Formal definition is-

Clique(G') = {<G, k >: G is a graph with a clique of size k }

Let CliqueSolve(G,K) is decision algorithm to solve the problem. If G has n vertices then optimization problem can be solved by making many application of CliqueSolve for k=n,n-1,n-2,…..,1. Here CliqueSolve is used once for each k. If complexity of CliqueSolve f(n), then the optimization problem (i.e size of maximum clique) can be solve in atmost nf(n) time. Also the decision problem (i.e size of maximum clique) can be solved in g(n) time. Thus clique's optimization problem can be solve in polynomial time iff decision problem can be solve in polynomial time.

non- deterministic polynomial time algorithm for clique problem

```
void clique(int G[ ][ ], int n, int k)
{
    S=Ø;
     for( int i=1; i≤k; i++)
       {
          int t= Choice(1,n);
           if(t is in s)  Failure();
           S=s U {t};
          }
```

```
    for (all (i,j), I € s, j € s and i≠j)
       if ((i,j) ∉ G(E)) Failure();
     Success();
}
```

Non-deterministic algorithm for 0-1 knapsack problem

```
        void knapsack( int pf[ ], int tw, int msize, int r, int x[ ])
                {
                    int tw=0, pf=0;
                     for(int i=1; i≤n; i++)
                       {
                          x[i]=Choice(0,1);
                            tw= tw+ x[i] * tw[i];
                            pf = pf+ x[i] * pf[i];
                        }
                     if (( tw > msize) || (pf < r)) Failure();
                      else
                         Success();
                  }
```

Here pf= total profit

tw= total weight

msize= knapsack size

r= maximum total profit

x[i]= 1 if ith item is choosen,0 otherwise

The for loop assigns value 0 or 1 to x[i], 1<=i<=n and calculate total weight and profit for this choice of x[i]. if statement checks weather total weight is greater than knapsack capacity msize and total profit is less than maximum total profit. If the result 1 then a successful completion and failure otherwise.

Time complexity is O(n).

Satisfiability (SAT) Problem

The decision problem of circuit satisfiability problem is – "Given a Boolean combinational circuit composed of AND, OR, NOT gates, is it satisfiable ?" The formal definition is

SAT= {< S >: S is a satisfiable boolean combinational circuit }

Let x1, x2,…. are some Boolean variable whose value is either 0 or 1. A Boolean variable(x) or its negation (! x) is called a literal. A formula can be expressed using literal, AND (V) and OR(Λ) operation. Example- (x_1 V x_2) . A formula is in CNF(conjunctive normal form) if and only if it is represented as Λ_i=1..k c_i, where c_i is represented as Vn_{ij}, where n_{ij} are litrals. A truth assignment that causes Boolean formula to result in 1 is called satisfiablity assignment. It is determined by the satisfiablity problem that a formula is true for some assignment of truth values to the variables or not.

CNF- satisfiability is called the satisfiability problem to CNF.

non deterministic algorithm for satisfiability(SAT)-

/* Here F is the Boolean formula and x[1],x[2]……..,x[n] are Boolean variable.*/

```
void Sat( cnf F, int n)
{
   int x[n]
     for( int i=1; i<=n i++)
       x[i]= Choice(0,1);
     if (F(x,n) ) Success();
     else
        Failure();
}
```

This algorithm will take time O(n) to choose x[1],x[2],……,x[n]. It also takes time T to determine value of F deterministically. Hence the time required by the algorithm is O(n) + T

CHECK YOUR PROGRESS

2. What is the decision problem of circuit satisfiability?
3. What is the size of a clique?

7.5 NP-HARD AND NP-COMPLETE CLASSES

Polynomial time solvable:

An algorithm is called polynomial time solvable if there exist a polynomial x such that the algorithm can compute in O(x(n)) time. Decision problem solves by deterministic algorithm in polynomial time is called P and decision problem solves by non deterministic algorithm in polynomial time is called NP. The famous unsolved problem in computer science is P=NP or P≠NP. Theoretical computer science believe that P≠NP. From the property of NPC, if any NP-Complete problem solves in polynomial time then all problem in NP can solved in polynomial time i.e P=NP. But there is no such polynomial time algorithm discovered for NPC.

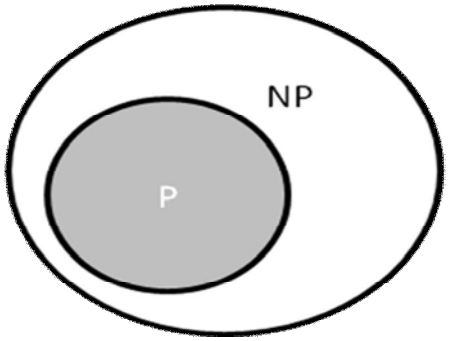


Fig. 7.2 P and NP, P≠NP

Reducibility:

A problem L_1 can be reduced to L_2 if and only if any instance P_1 of L_1 can be easily solved as an instance P_2 of L_2 by using a deterministic polynomial time algorithm. L_1 is ***polynomial time reducible*** to L_2 i.e $\boldsymbol{L_1 \leq_p L_2}$. If a problem L_1 reduces to another problem L_2 , then L_1 is "no harder to solve than L_2 ".

Now the properties of NP-Complete and NP-Hard for a language-

A language $L_1 \subseteq \{0,1\}$ is NP-Complete iff -

1. $L_1 \in NP$
2. $L_2 \leq_p L_1$ for every $L_1 \in NP$

A language $L_1 \in \{0,1\}$ is NP-Hard iff-

1. $L_2 \leq_p L_1$ for every $L_2 \in NP$, but it is not necessary to $L_1 \in NP$.

Lemma: If $L_1, L_2 \subseteq \{0,1\}^*$ are languages such that $L_1 \leq_p L_2$, then $L_2 \in P$ implies $L_1 \in P$

Proof:

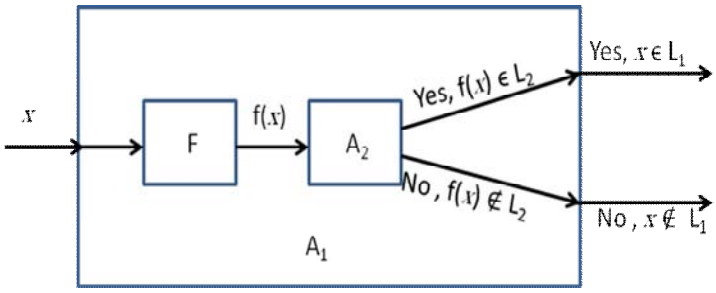
Given that, $L_2 \in P$, i.e L_2 is decided in polynomial time .

Let , for L_2 the polynomial time algorithm is A2.

Given that, $L_1 \leq_p L_2$

Let polynomial time reduction algorithm is F computes reduction function f.

Now, we have to construct an polynomial time algorithm that decide L_1.



If we give an input x € {0,1}* to the algorithm A1 then using the function F it will transform x into f(x). Next this transform output f(x) gives as input to algorithm A2. It will check whether f(x) € L_2. The output of A2 will be same as A1.

7.6 COOK'S THEOREM

Cook's theorem states that satisfiability (SAT) is NP-Complete.

Proof:

We know that from the property of NP-Complete , L is in NP-Complete iff L€ NP and for any L'€ NP, L'≤p L.

Here, we already know that (in example 5), Satisfiability has a non-deterministic polynomial time algorithm. Thus, satisfiability is in NP i.e SAT€ NP

Now, we have to show that a reduction exist in SAT for any problem L which is in NP. i.e if L € NP then L ≤p SAT

L€ NP, Hence L has a polynomial time verifier. Suppose it is T.

Then,

1. If x € L, witness y, T (x, y) = 1
2. If x ∉L, ∀ witness y, T(x, y) = 0

Since verifier T is a polynomial time verifier a circuit can be build with polynomial size for the verifier T ,The circuit contains AND, OR and NOT gates. The circuit has |x| + |y| sources, where |x| must be equal to the values of the bits in x and others |y| are variables**.**

We know the input value of x. So, we need to find input |y| variables which causes the circuit to output as 1. This means L has been reduced to check whether the circuit output is 1 or not. i.e L can be reduced to an instance SAT. It can be done as follows-

A 3-CNF , which means each clause has exactly three terms, can be used to represent each gate in the circuit.
For example:

1. Functionality of an OR gate with input p, q and output r_i is represented as follows:

$$(p \vee q \vee !r_i) \wedge (r_i \vee !a) \wedge (r_i \vee !b)$$

2. The functionality of a NOT gate with input p and output r_i is represented as follows:

$$(p \vee r_i) \wedge (!p \vee !r_i)$$

The clauses that have fewer than 3 terms, can be easily stuffing them with free literals(literals that don't affect in the result) to form clauses in 3CNF.

Let T has R gates $r_1, r_2 , r_3,\ldots..., r_R$ where r_R represent the final output of the verifier T and r_i ($1 \leq i < R$) represents the intermediate output of T .

They either takes some of the sources from (|x|+|y|) or some output r_i as input. So, the circuit can be represented as a formula in CNF:

$A = a_1 \Lambda a_2 \Lambda \ldots \Lambda a_R \Lambda r_R$

Where,

$a_i = (s_1 \Lambda s_2 \Lambda s_3)$ $s_1, s_2, s_3 \in (p, q, r_1, r_2, r_3, \ldots\ldots, r_R, !r_1, !r_2, !r_3, \ldots\ldots, !r_R)$

If a_i is not in 3CNF then it can made an equivalent formula in 3CNF by adding free variables. Thus, the circuit can be reduced to a_i, a formula in 3CNF which is satisfiable iff the original circuit gives output as 1. Hence, $L \leq_p$ SAT.

Since L is in NP. Hence, SAT is N P -Complete.

We can also say that 3-SAT is NP Complete because we considered the formulas are in 3-CNF. If we considered that formulas are in K-CNF then also K-SAT is NP-Complete, for K≥3.

CHECK YOUR PROGRESS

4. True/False

i. If a problem L_1 reduces to another problem L_2 , then L_2 is "no harder to solve than L_1 ".

ii. SAT is N P -Complete.

7.7 LET US SUM UP

- Polynomial time algorithms have running time (n^k), for some constraint k, on input of size n.

- Decision problems are problems that has 'yes' or 'no' answers i.e answers are decidable
- Optimization problem finds the optimal(maximum or minimum) solution for a problem.
- When 'yes' instances of a decision problem can decided in polynomial time then it is called a NP class problem.
- A problem that is NP-Complete has the property that it can be solved in polynomial time if and only if all other NP-Complete problem can solved in polynomial time.
- A NP-Hard problem has the property that if it can solved in polynomial time then all NP-Complete problem can solved in polynomial time.
- NP-Complete problems are NP-Hard, but all NP-Hard problems may not NP-Complete.
- Algorithm where every operation uniquely defines is called deterministic algorithm.
- Algorithm whose outcomes are not uniquely defined but are limited to a specified possibility is called non-deterministic algorithm.
- A clique G'(v, e) is a maximal complete sub-graph of a graph G(V,E)
- The decision problem of circuit satisfiability problem is – "Given a Boolean combinational circuit composed of AND, OR, NOT gates, is it satisfiable?"
- Decision problem solves by deterministic algorithm in polynomial time is called P and decision problem solves by non deterministic algorithm in polynomial time is called NP.
- If a problem L_1 reduces to another problem L_2 , then L_1 is "no harder to solve than L_2 ".
- Cook's theorem states that satisfiability (SAT) is NP-Complete.

7.8 ANSWER TO CHECK YOUR PROGRESS

1. i. True, ii. False, iii. True
2. The decision problem of circuit satisfiability problem is – "Given a Boolean combinational circuit composed of AND, OR, NOT gates, is it satisfiable ?" The formal definition is
   SAT= {< S >: S is a satisfiable boolean combinational circuit }

3. A clique G'(v,e) is a maximal complete sub-graph of a graph G(V,E) and size of the clique is |v| .

4. i. False, ii. True

7.9 FURTHER READINGS

- T. H. Cormen, C. E. Leiserson, R.L.Rivest, and C. Stein, "Introduction to Algorithms", Second Edition, Prentice Hall of India Pvt. Ltd, 2006.
- Ellis Horowitz, Sartaj Sahni and Sanguthevar Rajasekaran, Computer Algorithms/ C++, Second Edition, Universities Press, 2007.

7.10 MODEL QUESTIONS

1. What is decision and optimization problem?
2. What are the two classes of NP- problem?
3. What is P, NP class problem?
4. What is circuit satisfiability problem?
5. Write a non-deterministic algorithm for circuit satisfiabilty.
6. Write a non-deterministic algorithm for 0-1 knapsack problem.
7. What is reducibility?
8. What is non-deterministic algorithm? How does it work ?

9. Write the decision version of clique problem.
10. Write a non-deterministic polynomial time algorithm for clique problem?
11. What are the properties of NP-Complete and NP-Hard problem?
12. State and proof Cook's theorem.
13. Show that if L_1, L_2 □ {0,1}* are languages such that $L_1 \leq_p L_2$, then L_2 € P implies L_1 € P

*****