

Unit 12

Greedy Technique

Structure:

- 12.1 Introduction
 - Objectives
- 12.2 Introduction to Greedy Technique
 - Types of greedy algorithm
 - Greedy choice property
 - Applications of greedy technique
- 12.3 Prim's Algorithm
 - Description
 - Correctness
 - Time complexity
- 12.4 Kruskal's Algorithm
 - Description
 - Correctness
 - Time complexity
- 12.5 Dijkstra's Algorithm
 - Description
 - Correctness
 - Time complexity
- 12.6 Huffman Trees
 - Huffman code
 - Constructing Huffman tree
 - Constructing Huffman code
- 12.7 Summary
- 12.8 Glossary
- 12.9 Terminal Questions
- 12.10 Answers

12.1 Introduction

In the previous unit we learnt about dynamic programming which is an optimization technique. In this unit you will learn the concepts of Greedy technique algorithms that are used for optimization problems such as Kruskal's algorithm and Prim's algorithm for finding minimum spanning trees. You will also learn about Dijkstra's algorithm for finding single-source shortest paths, and the algorithm for finding optimum Huffman trees.

Objectives:

After studying this unit you should be able to:

- describe the Greedy technique
- construct Prim's, Kruskal's and Dijkstra's algorithm
- check for correctness of Prim's, Kruskal's and Dijkstra's algorithm
- construct the algorithm for finding optimum Huffman trees

12.2 Introduction to Greedy Technique

The Greedy technique constructs a solution to an optimization problem through a sequence of choices, each expanding a partially constructed solution until a complete solution to the problem is reached.

The sequences of choices made should be:

- Feasible, i.e. satisfying the constraints
- Locally optimal with respect to a neighborhood definition
- Greedy in terms of some measures and irrevocable

The greedy algorithm, always takes the best immediate solution while finding an answer. Greedy algorithms find the globally optimal solution for some optimization problems.

Now, let us see the different types of greedy techniques.

12.2.1 Types of greedy algorithm

Greedy algorithms can be classified as 'short sighted', and as 'non-recoverable'. They are ideal for problems with optimal substructure. Greedy algorithms are best suited for simple problems like giving change. The greedy algorithm can also be used as a selection algorithm to prioritize options within a search, or branch and bound algorithm. The following are a few variations to the greedy algorithm:

Pure greedy algorithms

Orthogonal greedy algorithms

Relaxed greedy algorithms

Pure greedy algorithm

Pure greedy algorithm makes local choices in all iterations in order to find an optimal solution. The Pure greedy algorithm chooses functions to use in approximating.

Relaxed greedy algorithm

The Relaxed greedy algorithm provides the approximation order, and gives a constructive proof of the estimate. There are several variants of the relaxed greedy algorithm and their application for different dictionaries.

Orthogonal greedy algorithm

In Orthogonal greedy algorithm the best approximation from the functions generated at each iteration is taken.

12.2.2 Greedy-choice property

In greedy algorithms a globally optimal solution is arrived by making a locally optimal (greedy) choice. That is to say when considering which choice to make, the choice that looks best in the current problem, without considering results from sub problems is selected. The choices made by a greedy algorithm can depend on choices of the past, but it cannot depend on any future choices or on the solutions to sub problems. This is where greedy algorithm differs from dynamic programming.

In dynamic programming, choices are made at each step, but the choice usually depends on the solutions to sub problems. Dynamic-programming solves problems in bottom-up manner that is solving from smaller sub problems to larger sub problems.

Therefore a greedy strategy usually progresses in a top-down fashion, making one greedy choice after another, reducing each given problem instance to a smaller one.

There are various applications of greedy technique. Let us see those next.

12.2.3 Applications of greedy technique

If a greedy algorithm is proven to yield the global optimum solution for a given problem class, it becomes the method of choice because it is faster than other optimization methods hence such algorithms can be applied in the following areas:

Optimal solutions:

- Change making for “normal” coin denominations
- Minimum spanning tree (MST)
- Single-source shortest paths
- Simple scheduling problems
- Huffman codes

Approximations/heuristics:

- Traveling salesman problem (TSP)
- Knapsack problem
- Other combinatorial optimization problems

Self Assessment Questions

1. The choices made in a greedy algorithm cannot depend on _____ choices.
2. The _____ is greedy in the sense that at each iteration it approximates the residual possible by a single function.
3. A greedy strategy usually progresses in a _____ fashion.

12.3 Prim's Algorithm

The previous section gave an introduction to the Greedy technique. Let us now discuss Prim's algorithm which is based on this technique. Prim's algorithm constructs a minimum spanning tree through a sequence of expanding sub-trees. The minimum spanning tree is obtained by selecting the adjacent vertices of already selected vertices. The tree starts from an arbitrary root vertex and grows until the tree spans all the vertices in the graph.

12.3.1 Description

This strategy is greedy since the tree is added at each step with an edge that contributes the minimum amount possible to the tree's weight. After every step, the current tree expands in the greedy manner by attaching it to the nearest vertex not in the tree. The algorithm stops after all the vertices have been included. Since the algorithm expands a tree by exactly one vertex on each of its iterations, the total number of such iterations is $n-1$, where n is the number of vertices.

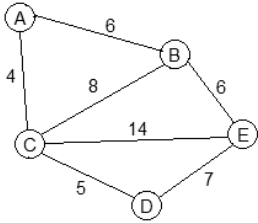
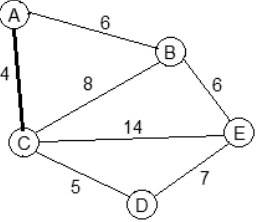
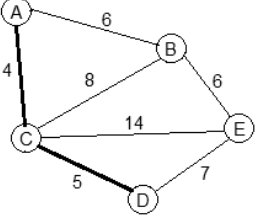
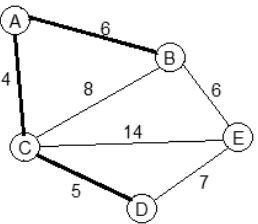
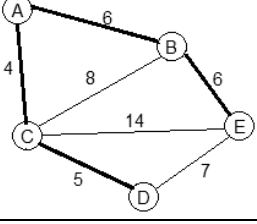
Let us apply Prim's algorithm to the graph considered in table 12.1 and analyze the working of the algorithm

Matrix 1 is the matrix representation of the graph considered in table 12.1

$$\begin{matrix}
 A \\
 B \\
 C \\
 D \\
 E
 \end{matrix}
 \begin{bmatrix}
 \begin{matrix} A & B & C & D & E \end{matrix} \\
 \begin{matrix} 0 & 6 & 4 & \infty & \infty \end{matrix} \\
 \begin{matrix} 6 & 0 & 8 & \infty & 6 \end{matrix} \\
 \begin{matrix} 4 & 8 & 0 & 5 & 14 \end{matrix} \\
 \begin{matrix} \infty & \infty & 5 & 0 & 7 \end{matrix} \\
 \begin{matrix} \infty & 6 & 14 & 7 & 0 \end{matrix}
 \end{bmatrix}$$

Matrix 1

Table 12.1: Application of Prim's Algorithm

Graph	Description
	Let us consider this weighted graph. The numbers near the edges indicate their weight.
	Choose Vertex C arbitrarily as the starting point. Vertices A, B, E and D are connected to C through a single edge. AC being the shortest arc with length 4 is chosen as the second vertex along with the edge AC.
	The vertex nearest to either C or A is the next vertex chosen. Arc AB is of length 6, arc CB is of length 8, arc CE is of length 14 and arc CD is of length 5. Therefore D is the smallest distance away, so the vertex D and the arc CD are chosen next.
	Next we have to choose a vertex from the remaining vertices which are nearest to either A, C or D. Vertex B, connected through arc AB is of length 6 is chosen.
	E is the remaining vertex and the shortest arc BE is chosen. The minimum spanning tree selected now has a weight of 21.

Pseudocode of Prim's Algorithm

1. (Initializations).

$O = \{1\}$ ($V(1)$ root of the T tree).

$P = \{2, \dots, n\}$

2. For every j belonging to P , $e(j) := c[e(j, 1)]$, $p(j) = 1$

(all peaks connected to the root. By definition of the cost function: $e(j) = \text{infinite}$ when $V(j)$ does not connect to $V(1)$).

3. Choose a k for which $e(k) \leq e(j)$ for every j belonging to P . In case of tight choose the smaller one. Exchange the O set with the set produced by the union of the O set and $\{k\}$. Exchange the P set with the set produced by the difference of the P set and $\{k\}$ ($P \leftarrow P - \{k\}$) If $P = \emptyset$ then stop.

4. For every j belonging to P compare $e(j)$ with $c[e(k, j)]$. If $e(j) > c[e(k, j)]$ exchange $e(j) \leftarrow c[e(k, j)]$. Go back to Step 1.

12.3.2 Correctness

Proof of correctness is proved by induction that each sub-tree generated by Prim's algorithm is a part of some other minimum spanning tree. The basis of the induction is that $T(0)$ consisting of a single vertex must be a part of any minimum spanning tree. Let us assume that $T(i-1)$ is part of some minimum spanning tree T , where $i = 0, \dots, n-1$.

This can be proved by contradicting and assuming that any minimum spanning tree of the graph can contain $T(i)$ let $e = (u, v)$ be the minimum weighted edge from a vertex in $T(i-1)$ to a vertex not in $T(i-1)$ used by the algorithm to expand $T(i-1)$ to $T(i)$, by our assumption, e cannot belong to T , but if we add e to T , a cycle must be formed. This cycle must contain another edge (u_1, v_1) which is connecting a vertex v_1 belonging to $T(i-1)$ to a vertex u_1 not in $T(i-1)$.

By deleting (u_1, v_1) we obtain another spanning tree of the entire graph whose weight is less than or equal to the weight of T . Because the weight of (u, v) is less than or equal to the weight of (u_1, v_1) this is a minimum spanning tree which contradicts the assumption that no minimum spanning tree contains $T(i)$. This proves the correctness of Prim's algorithm.

12.3.3 Time complexity

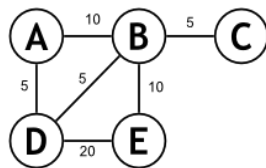
An implementation done by an adjacency matrix graph representation and searching an array of weights to find the minimum weight edge requires $O(V)$ running time. Using a binary heap data structure and an adjacency list representation, Prim's algorithm can be proved to run in time $O(E \log V)$ where E is number of edges and V is number of vertices. Using a sophisticated Fibonacci heap, this can be got to $O(E + V \log V)$, which is asymptotically faster when the graph is dense enough and function of E is $\Omega(V)$.

Self Assessment Questions

4. The _____ is obtained by selecting the adjacent vertices of already selected vertices.
5. Each _____ generated by prim's algorithm is a part of some other minimum spanning tree.
6. The greedy strategy in prim's algorithm is greedy since the tree is added with an edge that contributes the _____ amount possible to the tree's weight.

Activity 1

Apply Prim's algorithm to the graph and find the minimum spanning tree for the graph.



12.4 Kruskal's Algorithm

In the previous section we discussed Prim's algorithm. In this section we will explain Kruskal's algorithm which is also a greedy algorithm for minimum spanning tree problem that yields an optimal solution.

12.4.1 Description

Kruskal's algorithm finds a particular subset of the edges that are able to form a tree that contain all the nodes (vertices) without forming a cycle within the graph, but the total weight of the tree is minimized. If the graph is not connected, then the algorithm yields a minimum spanning forest.

The theory of Kruskal's algorithm: Create a forest in which each vertex is an individual tree. Then create a set S that contains all of the graph's edges. Search for the edge that has the minimum weight. If the edge connects two different trees, then include it to the forest and combine the two trees into one; otherwise, discard the edge.

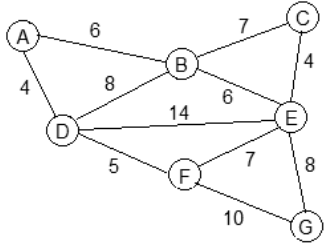
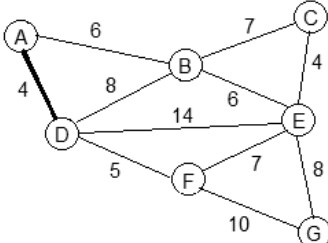
Let us apply Kruskal's algorithm to the graph considered in table 12.2 and analyze the working of the algorithm

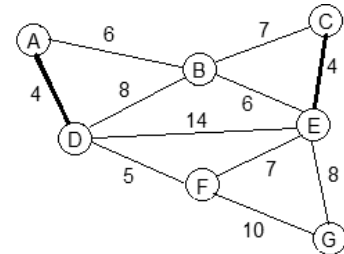
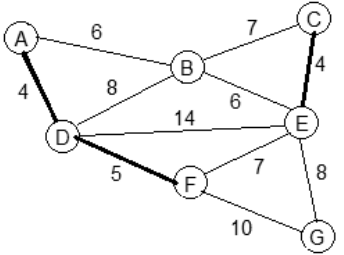
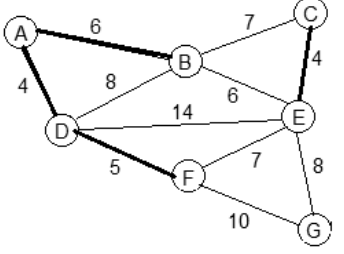
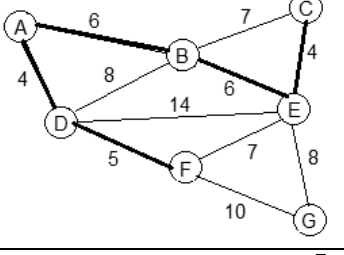
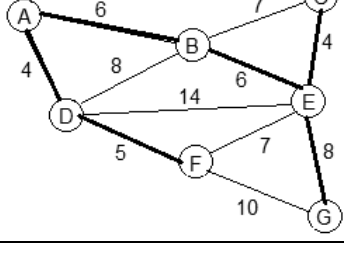
Matrix 2 is the matrix representation of the graph considered in table 12.2

A	$\begin{bmatrix} A & B & C & D & E & F & G \\ 0 & 6 & \infty & 4 & \infty & \infty & \infty \\ B & 6 & 0 & 7 & 8 & 6 & \infty & \infty \\ C & \infty & 7 & 0 & \infty & 4 & \infty & \infty \\ D & 4 & 8 & \infty & 0 & 14 & 5 & \infty \\ E & \infty & 6 & 4 & 14 & 0 & 7 & \infty \\ F & \infty & \infty & \infty & 5 & 7 & 0 & 10 \\ G & \infty & \infty & \infty & \infty & 8 & 10 & 0 \end{bmatrix}$
-----	---

Matrix 2

Table 12.2: Application of Kruskal's Algorithm

Graph	Description
	Let us consider this graph for Kruskal's algorithm. The numbers near the arcs indicate their weight.
	AD and CE are the shortest arcs, with length 4, let us choose AD arbitrarily.

Graph	Description
	CE is now the arc with shortest length that does not form a cycle, with length 4, so it is chosen as the second arc.
	The next arc, DF with length 5, is chosen using the same method.
	The next shortest arcs are AB and BE, both with length 6. AB is chosen arbitrarily. The arc BD is not included because there already exists a path between B and D; if included it would form a cycle ABD.
	The process chooses the next-smallest arc, BE with length 6. Arcs BC, DE and FE are not chosen as they would form the loop BCE, DEBA, and FEBAD respectively.
	Finally, the process finishes with the arc EG of length 8. Now all the edges are included in the tree.

Pseudocode of Kruskal's Algorithm

```

1 function Kruskal ( $G = \langle N, A \rangle$ : graph; length:  $A \rightarrow \mathbb{R}^+$ ): set of edges
2   Define an elementary cluster  $C(v) \leftarrow \{v\}$ .
3   Initialize a priority queue Q to contain all edges in G, using the weights
   as keys.
4   Define a forest  $T \leftarrow \emptyset$  //T will ultimately contain the edges of the
   MST
5   // n is total number of vertices
6   while T has fewer than n-1 edges do
7     // edge u, v is the minimum weighted route from u to v
8      $(u, v) \leftarrow Q.\text{removeMin}()$ 
9     // prevent cycles in T. add u, v only if T does not already contain a
     path between u and v.
10    // the vertices has been added to the tree.
11    Let  $C(v)$  be the cluster containing v, and let  $C(u)$  be the cluster
     containing u.
12    if  $C(v) \neq C(u)$  then
13      Add edge  $(v, u)$  to T.
14      Merge  $C(v)$  and  $C(u)$  into one cluster, that is, union  $C(v)$  and  $C(u)$ .
15    return tree T

```

12.4.2 Correctness

Kruskal's algorithm can be proved by two parts. Firstly we prove that the algorithm produces a spanning tree without forming a cycle and secondly we prove that the constructed spanning tree is of minimal weight.

Spanning tree

Let P be a connected, weighted graph and Y the sub-graph of P produced by the algorithm. Y will not have a cycle, since the last edge added to that cycle would have been within one sub-tree and not between two different trees. Y cannot be discarded, since the first encountered edge that joins two components of Y would have been added by the algorithm. Thus, Y is a spanning tree of P.

A real life application of spanning tree is in the phone network design. Consider that you have to connect all your office branches with a telephone network. The costs of these connections are based on the distance between the cities where the branches are situated. We want a set of lines which

connects all the branch offices with a minimum cost. Here, you can use a spanning tree to build a network.

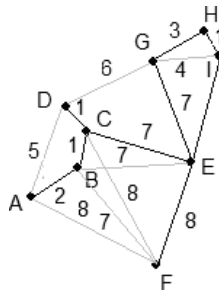


Figure 12.1: Spanning Tree

The figure 12.1 shows a spanning tree with the office branches given as A, B, C, D, E, F, G, H, I. Here the minimum distance between each branch is chosen for the connection.

12.4.3 Time complexity

The time complexity of Kruskal's algorithm depends completely on the implementation of Union and Find-Set. The Union-Find data structure implementation shows its efficiency only through amortized analysis.

Union-Find data structure

The Union-Find data structure is helpful for managing equivalence classes, and is vital for Kruskal's algorithm. This data structure helps us to store and manipulate equivalence classes. Equivalence class has to satisfy reflexive, symmetric and transitive properties. Each class has a representative element. We can unite two equivalence classes together and hence create a new equivalence class. The data structure therefore supports the operations - Make set, Union and Find.

Makeset(x) initializes a set with element x. Union(x, y) will union two sets together. Find(x) returns the set containing x. A simple implementation of this data structure is a tree defined by a parent array. A set is stored as a tree where the root represents the set, and all the elements in the set are descendents of the root. Find(x) works by keeping track of the parent pointers back until we reach the root parent. Makeset and Union are $O(1)$ operations but Find is an $O(n)$ operation, because the tree can get long and thin, depending on the order of the parameters in the calls to the Union. Particularly it is bad to point the taller tree to the root of the shorter tree.

We can fix this by changing Union. Union(x, y) will not just set the parent of x to y . Instead it will calculate which tree, x or y , has more number of nodes. Then it points the parent of the tree with the fewer nodes to the root of the tree with more nodes. This idea guarantees that the height of a tree is at most $\log n$. This means that the Find operation has become $O(\log n)$.

We can analyze Kruskal's algorithm by implementing the Union-Find data structure in this way. The sorting of the edges can be done in $O(e \log e)$ which is $O(e \log n)$ for any graph. For each edge (u, v) we check whether u and v are in the same tree. This is done with two calls to Find which is $O(\log n)$, and we unite the two if necessary which is $O(1)$. Therefore the loop is $O(e \log n)$. Hence the total time complexity is $O(e \log n)$.

An amortized analysis path compression is another way to make the trees even shorter and improve performance. The p operations of Union and Find using weighted union and path compression takes time $O(p \log^* n)$. Therefore each operation on the average is taking $O(\log^* n)$ i.e. Kruskal's algorithm runs in time $O(e \log^* n)$.

Self Assessment Questions

7. In Kruskal's algorithm if the graph is not connected, then the algorithm yields a _____.
8. The Union-Find data structure is helpful for managing _____ which is vital for Kruskal's algorithm.
9. Correctness of Kruskal's algorithm can be proved by saying that the constructed spanning tree is of _____.

12.5 Dijkstra's Algorithm

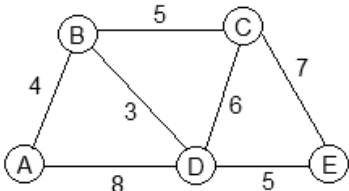
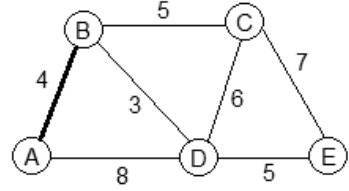
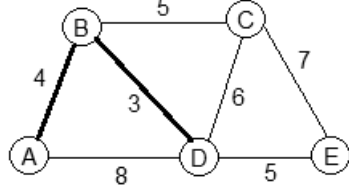
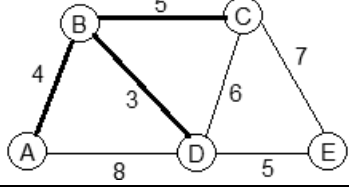
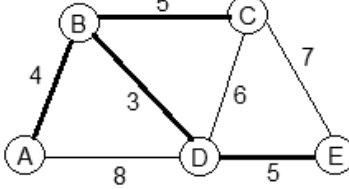
Let us now analyze Dijkstra's algorithm which is also a greedy algorithm and is similar to Prim's algorithm. It is applicable to both directed and undirected graphs.

12.5.1 Description

Dijkstra's algorithm finds solution for the single-source shortest path problem for a tree or graph with nonnegative edge path costs. For a given vertex in the graph, the algorithm finds the path with lowest cost between the originating vertex and every other vertex. It is also used for finding costs of shortest paths from a vertex to a destination vertex by ending the algorithm once the shortest path to the destination vertex has been determined.

The mechanism of Dijkstra's algorithm is similar to prim's algorithm. But they solve different problems and the priorities are computed in different ways. Dijkstra's algorithm compares path lengths and adds the edge weights, while prim's algorithm compares the edge weights. The operation of the Dijkstra's algorithm is explained in Table 12.3.

Table 12.3: Illustrating the Operation of Dijkstra's Algorithm

Graph	Description
	Let us consider this graph for Dijkstra's algorithm. The numbers near the arcs indicate their weight
	Let us choose A as the source vertex, consider the two vertices connected to A, comparing the path lengths. Vertex B is chosen with path length AB.
	To include vertices C and D, consider the path length through the vertex B which is already chosen. The shortest path length is included. Therefore vertex D with path length 7 is included to the tree
	The remaining vertices are C and E, with path lengths 9 and 12 respectively. Hence vertex C is included to the tree.
	The only remaining vertex is E, whose path length through C is 16 and through D is 12. Hence vertex E is included through D which is of shorter length.

Let us see the pseudocode for Dijkstra's algorithm.

Pseudocode for Dijkstra's Algorithm

```
1 function Dijkstra(Graph, source):
2   for each vertex v in Graph:      // Initializations
3     dist[v] := infinity
      // Unknown distance function from source to v
4     previous[v] := undefined
      // Previous node in optimal path from source
5     dist[source] := 0
      // Distance from source to source
6   Q := the set of all nodes in Graph
      // All nodes in the graph are unoptimized
7   while Q is not empty:
      // The main loop
8     u := vertex in Q with smallest dist[]
9     if dist[u] = infinity:
10      break
      // all remaining vertices are inaccessible from source
11    remove u from Q
12    for each neighbor v of u:
      // where v has not yet been removed from Q.
13      alt := dist[u] + dist_between(u, v)
14      if alt < dist[v]:
      // Relax (u,v,a)
15        dist[v] := alt
16        previous[v] := u
17  return dist[]
```

12.5.2 Correctness

Let us show that the algorithm is correct with all edge weights being nonnegative. Let $\delta(v,n)$ be the cost of the shortest path from the source vertex v to node n .

Claim: If the partially constructed tree T_i , constructed after i edges has been added, is a sub-tree of some shortest path tree, then the tree T^{i+1} , constructed after $i+1$ edges are added is also a sub-tree.

Proof of claim: Let T_{short} be a shortest path tree that contains T as a sub-tree. Let (m, n) be the $(i+1)_{\text{st}}$ edge added by the algorithm. If T_{short} contains (m, n) , the proof is complete. Otherwise, let P be the path of T_{short} from the source v to n .

Let T_{short}^0 be the tree obtained from T_{short} by removing the edge n of tree T_{short} . The sub-tree of T_{short} rooted at n hangs beneath m in T_{short}^0 .

We can claim that the path from v to n in T_{short}^0 is not costly than the path from v to n in T_{short} and so T_{short}^0 is indeed a shortest path tree of the graph. To see this, let c be the first node on path P that is not in tree T . Let b be the predecessor of c on path P . Because we have assumed that all the weights are non-negative, and a shortest path from v to n goes through c , then

$$\delta(v, c) \leq \delta(v, n): \quad \text{Eq. 12.1}$$

When x is added to the tree, the algorithm ensures that

$$\text{val}(c) \leq \text{val}(b) + \text{wt}(b, c) = \delta(v, c): \quad \text{Eq. 12.2}$$

Also, at the point when v is added to the tree, then

$$\text{val}(n) \leq \text{val}(c) \quad \text{Eq. 12.3}$$

The algorithm always adds a node with minimum cost and n is selected for addition before c . therefore by putting equations Eq 12.1, Eq 12.2, and Eq 12.3, we have

$$\text{val}(n) \leq \text{val}(c) \leq \delta(v, c) \leq \delta(v, n):$$

Now because $\text{parent}(n) = m$, we can say there is a shortest path from the source to n that passes through m and so T_{short}^0 is certainly a shortest path tree of the graph. Therefore this completes the proof.

12.5.3 Time complexity

The time complexity for the Dijkstra algorithm on a graph with n nodes and m edges is $O(n^2)$ because it permits directed cycles. Finding the shortest path from a source node to all other nodes in the graph has time complexity of $O(n^2)$ for node selection and $O(m)$ for distance updates. While $O(n^2)$ is the complexity for dense graphs, the complexity can be improved notably for sparse graphs.

Self Assessment Questions

10. Dijkstra's algorithm solves the single-source _____ problem for a tree.
11. The algorithm finds the path with lowest cost between the _____ vertex and every other vertex.
12. The time complexity of Dijkstra's algorithm can be improved for _____ graphs.

12.6 Huffman Trees

In the previous sections we learnt about algorithms which apply greedy technique. But in this section we are going to learn about a tree which gives optimal solution using greedy technique. A Huffman tree is a binary tree which minimizes the weighted path length from the root to the leaves which are predefined.

12.6.1 Huffman code

Huffman codes are digital data compression codes which are the outcome of the brilliant piece of work by Prof. David A. Huffman (1925-1999). Huffman codes give good compression ratios. Even today, after 50 years, Huffman codes have not only survived but are unbeatable in many cases. Huffman compression is a compression technique where there is no loss of information when the data is compressed i.e. after we decompress the data, the original information can be got. Hence it is named as lossless compression. Lossless compression is desired in compressing text documents, bank records etc.

Data encoding schemes fall into two categories,

- 1) **Fixed length encoding** – In fixed length encoding all symbols are encoded using the same number of bits. An example of fixed length encoding is ASCII code which uses 7 bits to encode a total of 128 different symbols. The difficulty with fixed length codes is that the probability of occurrence of the symbols to be encoded is not considered. A symbol that occurs 1000 times is encoded with the same number of bits as a symbol which comes only 10 times. This disadvantage makes fixed length encoding inefficient for data compression.

- 2) **Variable length encoding** – Variable length encoding removes this difficulty by assigning less number of bits to symbols which occur more often and more number of bits to symbols whose frequency of occurrence is less. The Huffman Encoding scheme falls in the category of variable length encoding i.e. code for the symbol depends on the frequency of occurrence of that symbol.

Huffman coding is again classified into two different groups –

- 3) **Static Huffman coding** – Static Huffman coding is done with the help of statistical symbol frequency tables in which symbol frequencies are known before the actual coding takes place
- 4) **Adaptive Huffman coding** – In adaptive Huffman compression the symbol frequencies need not be known in advance. Symbols are encoded as they are encountered.

12.6.2 Constructing Huffman tree

The following sequence of steps is to be followed to construct a Huffman tree:

1. Input all symbols along with their respective frequencies
2. Create leaf nodes representing the symbols scanned
3. Let S be a set containing all the nodes created in step 2
4. To create the Huffman Tree:
 - i. Sort the nodes (symbols) in S with respect to their frequencies.
 - ii. Create a new node to combine the two nodes with least frequencies.
 - iii. Frequency of this new combined node will be equal to sum of frequencies of nodes which were combined. This newly created combined node will be the parent of two nodes which were combined.
 - iv. Replace, in S, the two nodes which were combined with the new combined node.

After the 4th step you will be left with only one node, which is the root of the Huffman tree, having frequency equal to sum of all frequencies of all symbols. Thus a tree is generated with leaf nodes containing the basic symbols whose code is to be found.

Table 12.4: Symbol Frequency Table

Symbol	Frequency of occurrence
A	24
B	12
C	10
D	8
E	8

With the help of an example we will learn how to construct a Huffman tree

Using the symbols and frequencies from the table 12.4, we create the leaf nodes and then sort them.

Symbols 'D' and 'E' have least frequency, 8; these 2 nodes are combined to make a node 'DE' having frequency $8+8=16$. This new node 'DE' is the parent node of the nodes 'D' and 'E', and 'DE' replaces 'D' and 'E'.

Again we sort the nodes; now 'DE' and 'C' having least frequencies i.e. 16 and 10 each. This time we combine 'DE' and 'C' to create a new node 'DEC' having frequency 26.

Nodes 'DE' and 'C' are replaced by their parent 'DEC'.

Combine 'B' and 'DEC' to create 'BDEC' having frequency 12 of 'B' and 26 of 'DEC'. Hence 'BDEC' becomes the parent of 'B' and 'DEC' with frequency 38.

At last only two nodes are left namely 'BDEC' and 'A'. We again sort them and combine both of them to form 'ABDEC' which has frequency count of 62.

After making 'ABDEC' parent of 'A' and 'BDEC' and replacing them with 'ABDEC'; we have created the Huffman tree for the symbols in Table 12.4. Node 'ABDEC' is the root of the tree.

12.6.3 Constructing Huffman code

To assign codes start from root of the tree and assign 1 to every left branch and 0 to every right branch. We will now explain the process of tree creation. To find the code for a particular symbol, start from that symbol and traverse in direction towards the root. As soon as you encounter a branch assign the code to that branch (1/0); Suppose we need to find Huffman code for 'C'; we start from 'C' and move toward root i.e. $C \rightarrow EC \rightarrow DEC \rightarrow BDEC$

->ABDEC; we get code 1, 1, 0, 1, 0 respectively (i.e. 1 Cor C->EC, 1 Cor EC -> DEC, 0 C or DEC -> BDEC and so on). Note that this code that we have is from LSB to MSB. So the final code for 'A' will be "0". Table 12.2 shows Huffman code for all the symbols.

Table 12.5: Symbol Frequency Table with Huffman Code

Symbol	Frequency of occurrence	Huffman code
A	24	0
B	12	100
C	10	101
D	8	110
E	8	111

From table 12.5, you can notice that no codeword is also a prefix of another codeword. E.g. codeword for B is 100; now there is no other codeword which begins with 100. Codes with this property are called as Prefix codes. In prefix codes no codeword in the set is a prefix to another codeword. Huffman codes are Prefix codes. This property makes Huffman codes easy to decode. After studying much English prose, the frequency of characters has been analyzed and a Huffman code has been assigned to every character. Now suppose we want to code "ADECBA"; we can directly find Huffman code for the each of the symbols from Table 12.2 i.e. for 'A', 0 for 'D', 110 and so on. The code will look like "01101111011000".

There are 6 characters and it takes only 14 bits to transmit them. If we use normal ASCII code then it will take $7 \times 6 = 42$ bits to transmit the same string. Thus the use of Huffman codes has saved 28 bits which is around 66%. In a similar fashion Huffman codes can save from around 20% to 90% depending on the pattern of data being compressed.

Activity 2

Obtain the Huffman code for the following data and encode the text.

Character	P	Q	R	S
Frequency	55	10	10	25

Self Assessment Questions

13. Huffman codes are digital _____ codes.
14. The Huffman Encoding scheme falls in the category of _____.
15. Static Huffman coding is done with the help of _____ tables.

12.7 Summary

Optimization problems can be solved using greedy technique which involves a sequence of steps that include choices which should be feasible, optimal, and irrevocable. We discussed in this unit different algorithms that are based on the greedy technique.

Prim's algorithm is a greedy algorithm used to construct a minimum spanning tree of a weighted graph.

Kruskal's algorithm constructs a minimum spanning tree by selecting edges in the increasing order and including them in the tree such that it does not form a cycle.

Dijkstra's algorithm solves single-source shortest problems. It is similar to Prim's algorithm but considers path lengths instead of edge lengths.

Huffman trees minimize the path length from the path to the leaves.

To conclude, if a problem is solved efficiently by a greedy algorithm then it is widely accepted as the global optimal solution.

12.8 Glossary

Term	Description
Equivalence class	A set of things that are considered equivalent
Data compression	Information is encoded using fewer bits

12.9 Terminal Questions

1. Describe greedy choice property.
2. Describe the working of Prim's algorithm with an example.
3. Explain the time complexity in Kruskal's algorithm and the method of resolving it.
4. Explain the working of Dijkstra's algorithm with an example.
5. How are Huffman codes constructed?

12.10 Answers

Self Assessment Questions

1. Future
2. Pure greedy algorithm
3. Top-down
4. Minimum spanning tree
5. Sub-tree
6. Minimum
7. Minimum spanning forest
8. Equivalence classes
9. Minimal weight
10. Shortest path
11. Originating
12. Sparse
13. Data compression
14. Variable length encoding
15. Statistical symbol frequency

Terminal Questions

1. Refer to 12.2.2 – Greedy choice property
2. Refer to 12.3.1 – Description
3. Refer to 12.4.3 – Time complexity
4. Refer to 12.5 – Dijkstra's algorithm
5. Refer to 12.6.3 – Constructing Huffman codes

Reference

- Anany Levitin (2009). *Introduction to Design and Analysis of Algorithms*. Dorling Kindersley, India
- Cormen, H. Thomas (2001). *Introduction to Algorithms* MIT. Press, McGraw-Hill Book Company

E-Reference

- www.cs.cmu.edu/afs/cs/academic/class/15853-f00/.../compress1.ppt
- <http://www.cs.ubc.ca/~nando/320-2003/lectures/lecture9-2.pdf>
- <http://www.devarticles.com/c/a/Development-Cycles/Greedy-Strategy-as-an-Algorithm-Technique/2>
- <http://www.mec.ac.in/resources/notes/notes/ds/kruskul.htm>
- http://www.programmersheaven.com/2/Art_Huffman_p1