IQRA National University Peshawar

Sessional Assignment

Subject: ADVANCED ALGORITHM ANALYSIS

Submitted to: Dr. Atif Ishtiaq (HOD CS)

Submitted by: Noor rahman

Reg Id 14232

MSCS 2017-18

Department of Computer Science

Q. 1) Explain Greedy Algorithms with the help of example.

Answer:

A greedy algorithm proceeds in the same way as scrooge did. That is, it grabs data items in sequence, each time taking the one that is deemed "best" according to some criterion, without regard for the choices it has made before or will make in the future. One should not get the impression that there is something wrong with greedy algorithms because of the negative connotations of scrooge and the word "greedy." They often lead to very efficient and simple solutions. Like dynamic programming, greedy algorithms are often used to solve optimization problems. However, the greedy approach is more straightforward. In dynamic programming, a recursive property is used to divide an instance into smaller instances. In the greedy approach, there is no division into smaller instances. A greedy algorithm arrives at a solution by making a sequence of choices, each of which simply looks the best at the moment. That is, each choice is locally optimal. The hope is that a globally optimal solution will be obtained, but this is not always the case. For a given algorithm, t determine whether the solution is always optimal

Example

A simple example illustrates the greedy approach. Joe, the sales clerk, often encounters the problem of giving change for a purchase. Customers usually don't want to eceive a lot of coins. For example, most customers would be aggravated if he gave them 87 pennies when the change was $0.87. Therefore, his goal is not only to give the correct change, but to do so with as few coins as possible. A solution to an instance of Joe's change problem is a set of coins that adds up to the amount he owes the customer, and an optimal solution is such a set of minimum size. A greedy approach to the problem could proceed as follows. Initially there are no coins in the change. Joe starts by looking or the largest coin (in value) he can find. That is, his criterion for deciding which coin is best (locally optimal) is the value of the coin. This is called the selection procedure in a greedy algorithm.

Q.2) Explain Huffman Coding with example.

Answer:

Even though the capacity of secondary storage devices keeps getting larger and their cost keeps getting smaller, the devices continue to fill up due to increased storage demands. Given a data file, it would therefore be desirable to find a way to store the file as efficiently as possible. The problem of **data compression** is to find an efficient method or encoding a data file. Next, we discuss the encoding method, called **Huffman code**, and a greedy algorithm for finding a Huffman encoding for a given file.
A common way to represent a file is to use a binary code. In such a code, each character is represented by a unique binary string, called the code word. A fixed-length binary code represents each character using the same number of bits. For example, suppose our character set is {a, b, c}. Then we could use 2 bits to code each character, since this gives us four possible code words and we need only three. We could code as follow.

a:00 b:01 c:11

Given this code, if our file is

ababbbbc,

Our encoding is

0001000111010111

Figure Binary tree corresponding to Code.

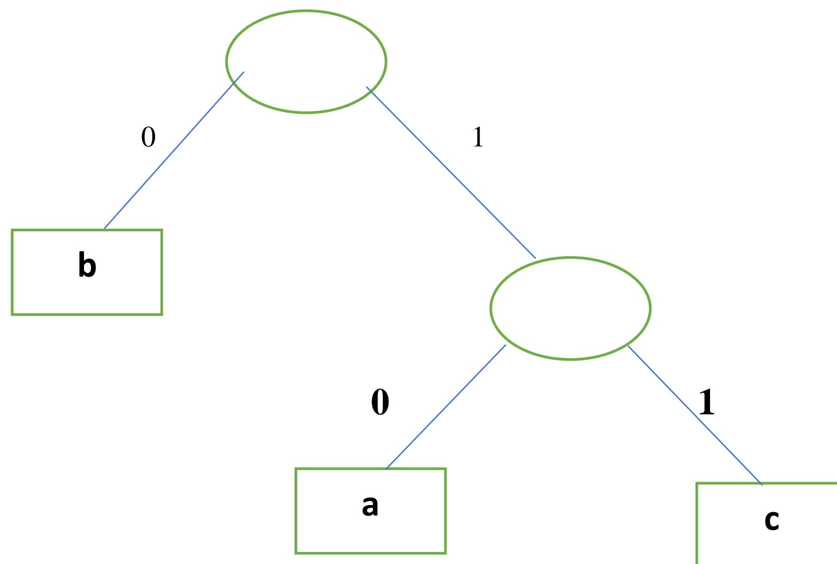


Figure . The binary character code for Code C2 in Example 4.7 appears in (a), while the one for Code C3 (Huappears in (b).

```
struct nodetype
{
  char symbol;        // The value of a character.
  int frequency;      // The number of times the character
                      // is in the file.
  nodetype* left;
  nodetype* right;
};
```

Furthermore, we need to use a priority queue. In a **priority queue**, the element with the highest priority is always removed next. In this case, the element with the highest priority is the character with the lowest frequency in the file. A priority queue can be implemented as a linked list, but more efficiently as a heap (See Section 7.6 for a discussion of heaps.) **Huffman's algorithm** now follows.

n = number of characters in the file;

Arrange n pointers to nodetype records in a priority queue PQ as follows: For each pointer p in PQ

```
p-> symbol = a distinct character in the file;
p-> frequency = the frequency of that character in the file;
p-> left = p-> right = NULL;
```

The priority is according to the value of frequency, with lower frequencies having higher priority.

```
for (i=1; i < = n-1; i++) {   // There is no solution check; rather,
    remove(PQ, p);            // solution is obtained when i = n - 1.
    remove(PQ, q);            // Selection procedure.
    r = new nodetype;         // There is no feasibility check.
    r->left = p;
    r->right = q;
    r->frequency = p->frequency + q->frequency;
    insert(PQ, r);
}
remove(PQ, r);
return r;
```

If a priority queue is implemented as a heap, it can be initialized in $\theta(n)$ time. Furthermore, each heap operation requires $\theta(\lg n)$ time. Since there are $n-1$ passes through the **for**-i loop, the algorithm runs in $\theta(n \lg n)$ time.

Q.3) Explain Dijkstra's Algorithm.

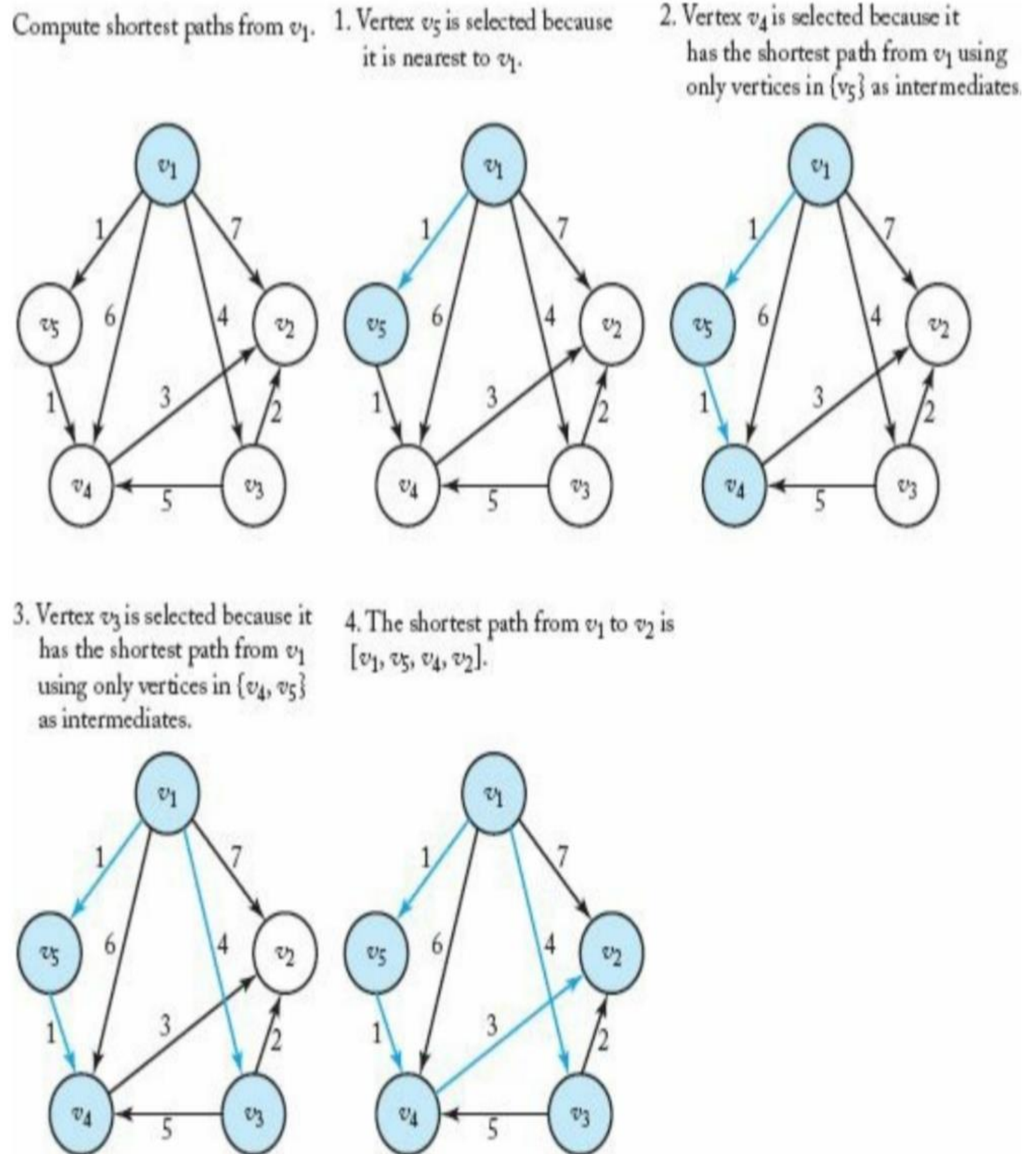
This algorithm is similar to Prim's algorithm for the Minimum Spanning Tree problem. We initialize a set Y to contain only the vertex whose shortest paths are to be determined. For focus, we say that the vertex is v1. We initialize a set F of edges to being empty. First we choose a vertex v that is nearest to v1, add it to Y , and add the edge < v1, v > to F. (By < v1, v > we mean the directed edge from v1 to v.) That edge is clearly a shortest path from v1 to v. Next we check the paths from v1 to the vertices in V - Y that allow only vertices in Y as intermediate vertices. A shortest of these paths is a shortest path (this needs to be proven). The vertex at the end of such a path is added to Y , and the edge (on the path) that touches that vertex is added to F. This procedure is continued until Y equals V , the set of all vertices. At this point, F contains the edges in shortest paths. The following is a high-level algorithm for this approach.

```
Y = {v1};
F = ∅;

while (the instance is not solved){

  select a vertex v from V − Y, that has a    // selection
  shortest path from v1, using only vertices  // procedure and
  in Y as intermediates;                      // feasibility check

  add the new vertex v to Y;
  add the edge (on the shortest path) that touches v to F;

  if (Y == V)
    the instance is solved;                  // solution check
}
```

Example:

Dijkstra's Algorithm
Problem: Determine the shortest paths from v1 to all other vertices in a weighted directed graph.
Inputs: integer $n \geq 2$, and a connected, weighted, directed graph containing n vertices
The graph is represented by a two-dimensional array W, which has both its rows and columns indexed from 1 to n, where W [i] [j] is the weight on the edge from the ith vertex to the jth vertex.

Figure . A weighted, directed graph (in upper-left corner) and the steps in Dijkstra's algorithm for that graph. The
vertices in Y and the edges in F are shaded in color at each step.



```
void dijkstra (int n, const number W[][], set_of_edges& F)
{
  index i, vnear;
  edge e;
  index touch[2..n];
  number length[2..n];

  F = ∅;
  for (i = 2; i <= n; i++){              // For all vertices, initialize v1
      touch[i] = 1;                      // to be the last vertex on the
      length[i] = W[1][i];               // current shortest path from
  }                                      // v1, and initialize length of
                                         // that path to be the weight
                                         // on the edge from v1.
  repeat (n − 1 times){                  // Add all n − 1 vertices to Y.
     min = ∞;
     for (i = 2; i <= n; i++)            // Check each vertex for
        if (0 ≤ length[i] < min){        // having shortest path.
           min = length[i];
           vnear = i;
        }
     e = edge from vertex indexed by touch[vnear]
         to vertex indexed by vnear;
     add e to F;
     for (i = 2; i <=n; i++)
        if (length[vnear] + W[vnear][i] < length[i]){
           length[i] = length[vnear] + W[vnear][i];
           touch[i] = vnear;             // For each vertex not in Y,
        }                                // update its shortest path.
     length[vnear] = −1;                 // Add vertex indexed by vnear
  }                                      // to Y.
}
```

Because we are assuming that there is a path from v1 to every other vertex, the variable vnear has a new value in each iteration of the repeat loop. If this were not the case, the algorithm, as written, would end up adding the last edge over and over until n - 1 iterations of the repeat loop were completed. Algorithm 4.3 determines only the edges in the shortest paths. It does not produce the lengths of those paths. These lengths could be obtained from the edges. Alternatively, a simple modification of the algorithm would enable it to compute the lengths and store them in an array as well

…………………………………………………………………………………………….