Study on Selected Greedy Algorithms

Fairooz Maliha Rahman
Student Id:011132010
Supta Bhowmick
Student Id:011132020
Luthfun Naher Eva
Student Id:011132037
Anwara Khatun Anu
Student Id:011052003

A thesis in the Department of Computer Science and Engineering presented in partial fulfillment of the requirements for the Degree of Bachelor of Science in Computer Science and Engineering

United International University
Dhaka, Bangladesh
January 2018

Declaration

This is to certify that the work entitled "**Study on Selected Greedy Algorithms**" is the outcome of the research carried out by us under the supervision of Prof. Dr. Mohammad Nurul Huda, Professor & MSCSE Coordinator, Dept. of CSE, United International University, Dhaka, Bangladesh.

_____

Fairooz Maliha Rahman, Student ID:011132010, Dept. of CSE
Supta Bhowmick, Student ID:011132020, Dept. of CSE
Luthfun Naher Eva, Student ID:011132037, Dept. of CSE
Anwara Khatun Anu, Student ID: 011052003, Dept. of CSE

In my capacity as supervisor of the candidates' thesis, I certify that the above statements are true to the best of my knowledge.

_____

Prof. Dr. Mohammad Nurul Huda
Professor & MSCSE Coordinator
Dept. of CSE, United International University,
Dhaka, Bangladesh

Abstract

We studied some Greedy Algorithms, implemented to observe the in depth functionalities. We used the main idea of Knapsack problem to solve some problems which are inspired by real life applications. Afterwards we thought of implementing MST with Genetic Algorithm. Genetic Algorithm is one of the most evolutionary methods in the field of computer science for solving optimization problem. With the help of Genetic Algorithm we designed an algorithm that makes it possible to solve MST problems more efficiently.

Acknowledgement

We are indebted to Almighty Allah for the favor that we could conclude our thesis work. We are really obliged to our honorable supervisor Dr. Mohammad Nurul Huda, Professor, Department of Computer Science and Engineering. His sincere, valuable guidance and encouragement helped us to reach in this position. Our thesis work was more reinforced by his assistance and dedicated involvement.

We express our sincerest respect to all of the Department faculty members for their encouragement and cooperation. We would like to thank you very much for all of your advice and support over these years.

Table of Contents

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Algorithms cannot be designed hoping to be the magical solution to a complicated problem. Different techniques are required to solve different kind of problems. Greedy algorithms are one of the techniques. Greedy algorithms make a locally-optimal choice that will lead to a globally-optimal solution.

We have studied selected greedy algorithms which are Prim's and Kruskal's algorithms for minimum spanning tree, Dijkstra's and Bellman-Ford's algorithms for shortest path. We have also studied Knapsack problems and Huffman coding. Following the study we tried to implement and solve minimum spanning tree problem. We used a variant of Genetic algorithms to solve the problem.

In monitoring other Greedy Algorithms, it was seen that some Greedy algorithms do not achieve best optimal solution in several cases as they seem to be short sighted. They have only one chance to determine the optimal solution and thus the decision becomes irrevocable [2]. The main contribution of this paper is to determine such a Greedy Algorithm that can overcome this weakness and can give the best optimal solution. The main goal is to use some properties of genetic algorithm. We made different combinations of elimination technique to make all possible spanning tree and furthermore picked the spanning tree with the least objective function value. By doing so the possibility of getting best optimal minimum spanning tree was increased.

Chapter 2

Background

For particular problem greedy algorithms takes decision based on the information they have on that particular time. They emphasize on local analysis rather than global. As they focus only on single basis to solve the problem they are called ‘greedy’.

2.1 Why we use greedy algorithm

Generally if a problem contains an objective function and that needs to be optimized which could be either minimization or maximization then greedy algorithms are suitable candidate for it. While running, the Greedy algorithm makes greedy choices at each step to make sure to optimize the objective function considering that it never goes back and reverses decision [1].

Greedy Algorithms can solve problems having the following properties:

1. Greedy Choice Property
2. Optimal Substructure.

Also Greedy Algorithms solve combinatorial problems having the properties of matroid.

Considering the real life applications, Greedy Algorithms are used in various branches such as:

- Internal Scheduling
- Minimization (Minimum spanning tree)
- Knapsack problems
- Data compression (Huffman)
- Gaming
- Coin change problems
- Optimal Merging
- Topological Sort

2.2 When Greedy Algorithms fail

We know that greedy algorithms try to obtain local optimizations so that it can find reasonable global solutions. But this property leads greedy algorithms to fail in some cases. So sometimes they tend to find solutions that are even less than optimal solutions or sub optimal solutions [2]. Such case happened in the following scenario.

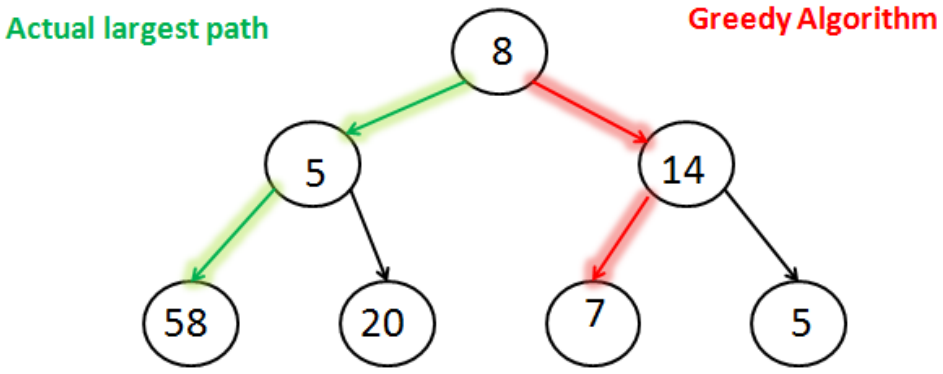


Figure 2.1: How Greedy Algorithm fails to obtain optimal solution

In this scenario we wanted to find the largest sum. At each step they consider the information that they have. So at the beginning they choose 14 as it appears to be larger than other neighbors. Further they approach the path where they find the largest value. Which is 8+14+7= 29. But this value is not the largest sum. The best solution would be 8+5+58 = 71. As Greedy Algorithms only emphasize to the local space instead of global space, they gradually fail to gain the best solution. For this property Greedy Algorithms are often called ‘short sighted’.

2.3 Types

Standard:

1. Huffman Coding
2. Job Sequencing Problem
3. Egyptian Fraction
4. Efficient Huffman Coding for sorted input
5. Activity Selection Problem

Graph Theory:

Minimum Spanning Tree:

1. Prim's Algorithm (Minimum Spanning Tree)
2. Kruskal's Algorithm (Minimum Spanning Tree)
3. Reverse delete algorithm (Minimum Spanning Tree)
4. Boruvka's Minimum Spanning Tree
5. Dial's Algorithm

Shortest Path:

1. Dijkastra's Algorithm (Shortest Path)
2. Bellman-Form Algorithm (Shortest Path)

Adjacency List Representation:

1. Dijkstra's Adjacency List Representation
2. Prim's adjacency list representation

Operating Systems:

Scheduling:

- Shortest Job First

Memory Management:

1. Best Fit algorithm
2. First Fit algorithm
3. Worst Fit algorithm

Chapter 3

Selected Greedy Algorithms

3.1 Prim's Algorithm for minimum spanning tree

3.1.1 History

In 1930 Prim's algorithm was developed by mathematician Vojtěch Jarník. But it was not renowned and famous that time. Later in 1957 computer scientist Robert C. Prim rediscovered and republished it. 2 years later in 1959, another scientist Edsger W. Dijkstra did the same. Prim's algorithm is called by many names such as the Prim–Jarník algorithm, DJP algorithm, Jarník's algorithm or the Prim–Dijkstra algorithm.

3.1.2 Algorithm

3.1.2.1 Pseudocode of prim's algorithm

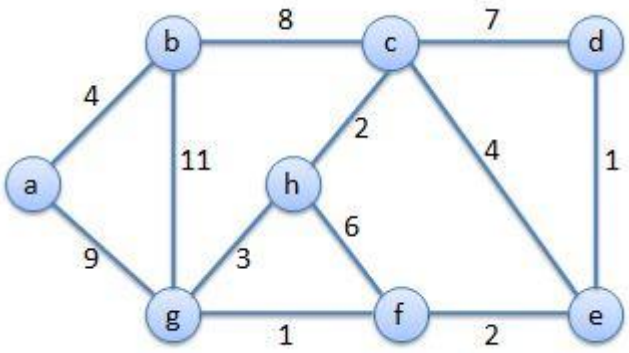
```
Minimum(vertex)
        n=INT_MAX
        for i=0 to vertex-1
                if q.i>0 and key.i<n
                        n=key.i
                        k=i
        return k

Prime_algo(e,v)
        for i=0 to vertex-1
                key.i=INT_MAX
                predecessor.i=-1
                q.i=1
        key.0=0
        for j=0 to vertex-1
                u=Minimum(vertex)
```

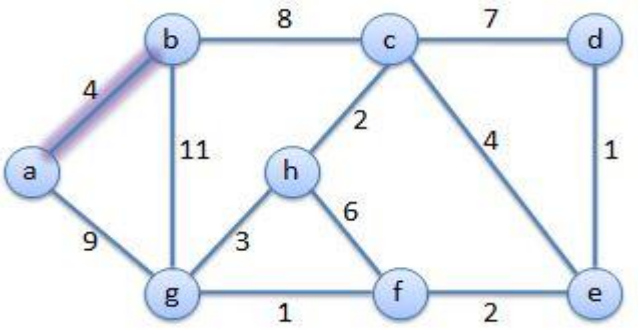
```
a.j=u
for i=0 to vertex-1
        if e(u,i)!=0 and q.i>0
                if e(u,i)<key.i
                        key.i=e(u,i)
                        predecessor.i=u
q.u=0
```

3.1.2.2 Graphical representation and description

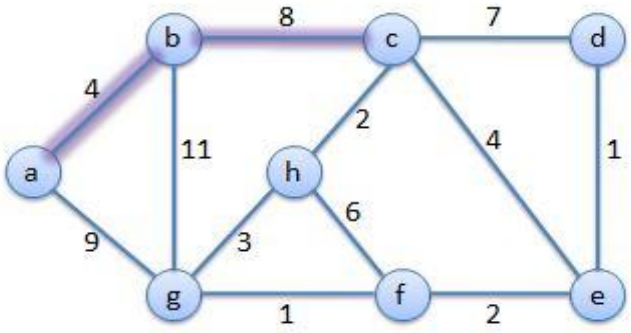
The concept working behind Prim's algorithm is to build MST with choosing one edge at a time. It then uses the tree currently built to make branches out from it. The final tree keeps the entire partial minimum spanning tree which keeps all the other components connected with one another. Here is given a graph with 8 vertices and 12 edges. With the help of this graph, Prim's algorithm working process is shown below.



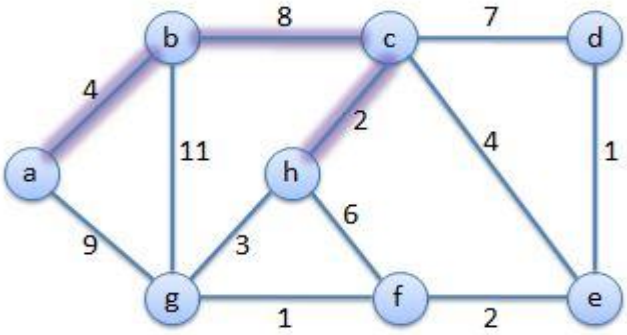
(a)The original graph



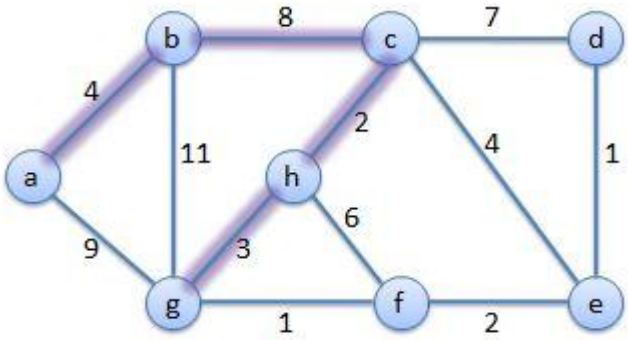
(b) Prim's algorithm first chooses an initial vertex. The smallest edge connected with this vertex is chosen to build MST.



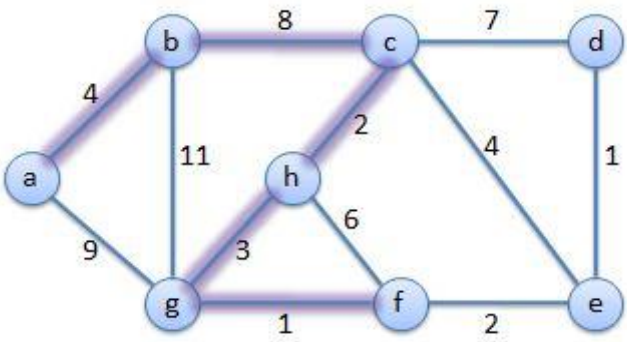
(c) Prim then finds the smallest edge among the edges connected with the vertices a and b. So edge bc is chosen.



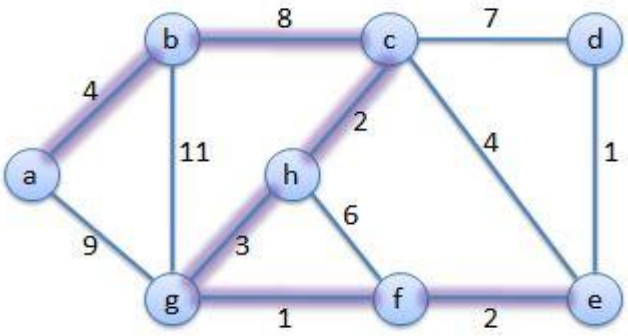
(d) Prim then finds the next smallest edge among the edges connected with the vertices a, b and c. So edge ch is chosen.



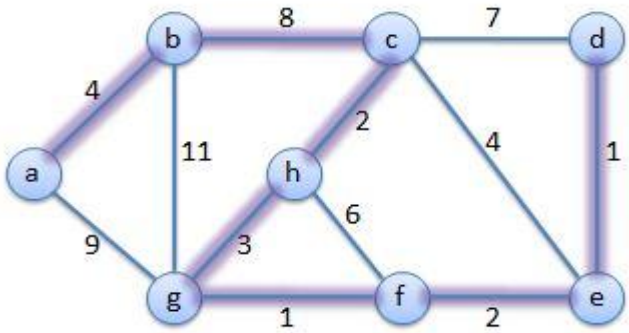
(e) Prim then finds the another smallest edge among the edges connected with the vertices a, b, c and h. So edge hg is chosen.



(f) Prim then finds the next smallest edge among the edges connected with the vertices a, b, c, h and g. So edge gf is chosen.



(g) Prim then finds the other smallest edge among the edges connected with the vertices a, b, c, h, g and f. So edge fe is chosen.



(h) Prim then finds the next smallest edge among the edges connected with the vertices a, b, c, h, g, f and e. So edge de is chosen. And thus the optimal MST is created.

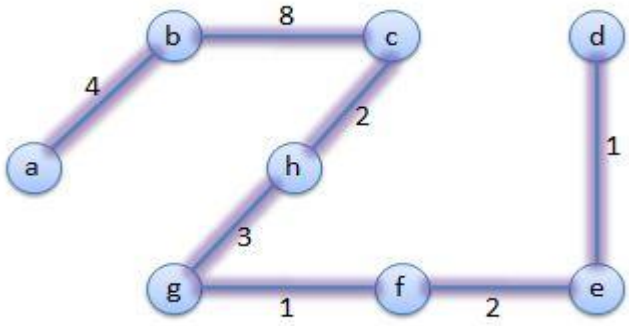


Figure 3.1: Final Minimum Spanning Tree generated by using Prim's Algorithm.

3.1.2.3 Time complexity

We used adjacency list and binary heap to solve Prim's algorithm. We came to find time complexity as the order O(VlogV + ElogV) = O(ElogV**)** where V is the number of vertices and E is the number of edges of the given graph.

The time could be reduced by using Fibonacci heap instead of binary heap and adjacency list. Then the time complexity becomes O(VlogV+E+V)=O(E+VlogV).

For sparse graphs, Prim's algorithm works faster. As the number of vertex and edge increases and graph becomes more dense, Prim's algorithm runs in linear time.

3.2 Kruskal's Algorithm for minimum spanning tree

3.2.1 History

In February 1956 **.** Jewish American mathematician, statistician, computer scientist Joseph Bernard Kruskal, Jr. first invented that MST problem can be solved in other ways. This algorithm was first published in the book named 'Proceedings of the American Mathematical Society' in the chapter called "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem". From then this algorithm was named after Kruskal and became popular as Kruskal's Algorithm.

3.2.2 Algorithm

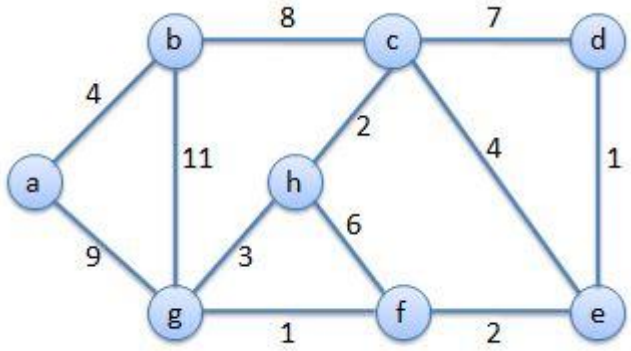
3.2.2.1 Pseudocode of Kruskal algorithm

```
kruskal_algo(e,v)
        make a disjoint set A which is initially empty
        sort matrix e in non-decreasing order by weight
        for each (u,v) of the sorted list
                if findset(u)!=findset(v)
                        A←A ∪ (u,v)
                        union (u,v)
        return A
```

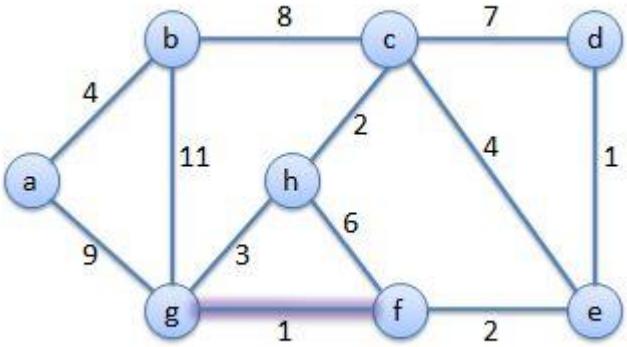
3.2.2.2 Graphical representation and description

The concept working behind Kruskal's algorithm is to build MST with choosing smallest edge among all the edges present in the graph in each step. This algorithm checks if any cycle is formed in currently built structure that would going to be part of the MST. If no cycle is formed then it picks the edge and adds to the structure. If the algorithm finds two edges of same value then it chooses one arbitrarily and again checks for any cycle. It works in the same way until the resultant structure has (V-1) edges where V is the number of vertices. Thus it gets the optimal Minimum Spanning Tree.

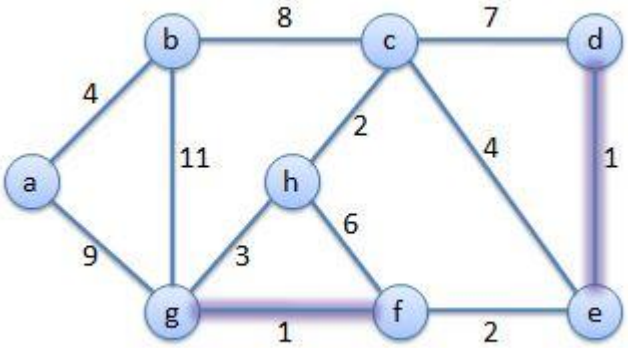
Here is given a graph with 8 vertices and 12 edges. With the help of this graph, Kruskal's algorithm working process is shown below.



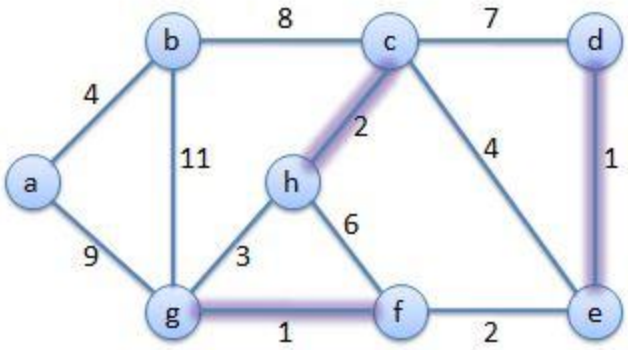
(a)The original graph



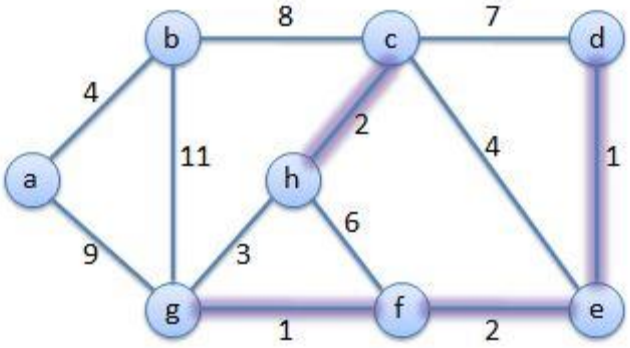
(b) gf and ed are the smallest weighted edges in the graph. So kruskal's algorithm picks one edge arbitrarily. So here gf is chosen.



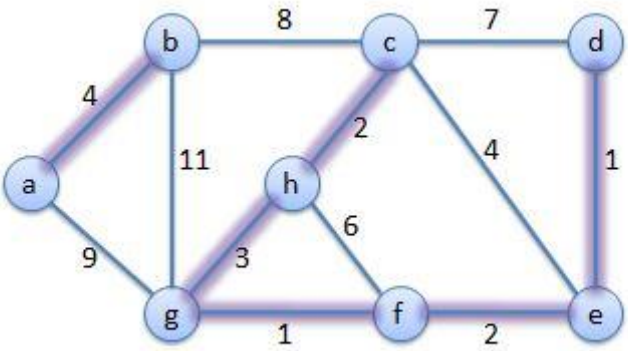
(c) Now the shortest edge that does not form a cycle is edge ed. So the algorithm chooses ed.



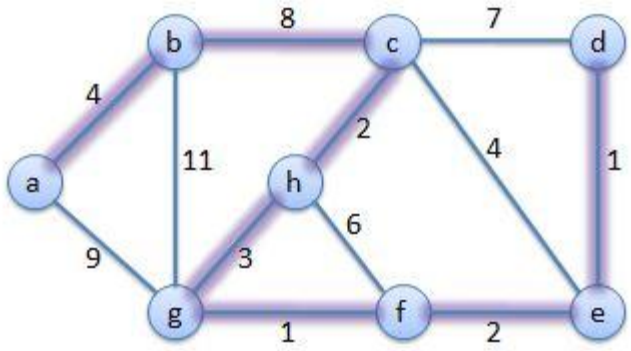
(d) In the same way next smallest edge ch is picked.



(e) Next edge fe is picked as it is the next smallest edge



(f) As the next smallest edge becomes gh so it is picked. In the next step both ab and ce have the same weight. The algorithm discards ce as it creates a cycle so ab is chosen.



(g) edge hf having the weight 6 is the next smallest weight but as it creates a cycle, the edge is discarded. Same thing happened with cd having the weight 7. So the algorithm chooses edge bc with the weight 8. The algorithm stops working in this stage as it finds (V-1) here 8-1=7 edges for the graph.

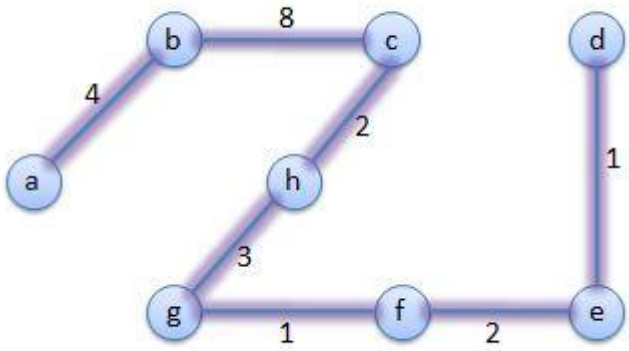


Figure 3.2: The final Minimum Spanning Tree generated by using Kruskal's Algorithm.

3.2.2.3 Time Complexity

The time complexity of Kruskal's Algorithm is $O(E \log E)$ or $O(E \log V)$ where V is the number of vertex and E is the number of edge. The two main properties of Kruskal's algorithm are sorting the edges and making union of the edges. It takes $O(E \log E)$ time for sorting the edges and $O(\log V)$ time for applying union to the edges. As the value of E is almost $O(V^2)$, so $O(\text{LogV}) = O(\text{LogE})$.

3.3 Dijkstra's algorithm for Single Source Shortest Path

3.3.1 History

In 1959 Edsger Wybe Dijkstra (1930-2002) invented a systemic way to find the shortest path in a graph between any two given nodes. He published in a 3-page article named 'A note on two problems in connexion with graphs', where the algorithm was described. Since then this algorithm was known as Dijkstra's Algorithm and it has become the underlying theory for all other single source shortest path algorithm for general directed and undirected graphs.

Many authors proposed different modification of Dijkstra's algorithm by reducing the run time using heuristics.

3.3.2 Algorithm

3.3.2.1 Pseudocode of Dijkstra algorithm

```
function Dijkstra(e, s)
    create vertex set qi
    for each vertex v in e
        distance[v] ← ∞
        predecessor[v] ← -1
        add v to qi
    distance[s] ← 0
    while qi is not empty
        u ← vertex in qi with min distance[u]
        remove u from qi
        for each neighbor v of u
```

b ← distance[u] + len(u, v)

if b <distance[v]

distance[v] ← b

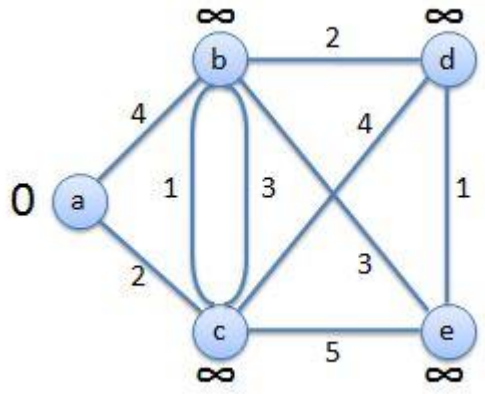
predecessor [v] ← u

return distance[],predecessor []

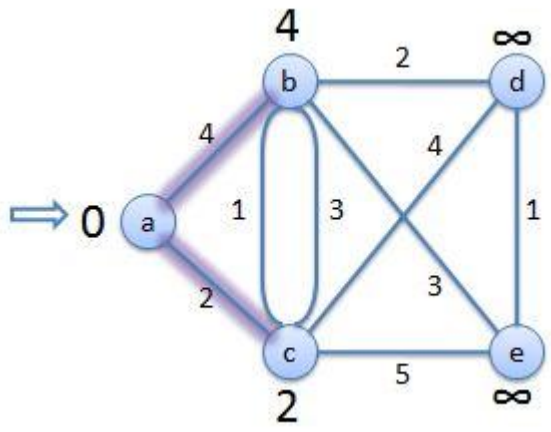
3.3.2.2 Graphical representation and description

Dijkstra's Algorithm can work in both directed and undirected connected graph having non negative weights. Initially this algorithm marks the source vertex with 0 and other vertices with *infinity* (∞).Then it assigns all the vertices connected with the source with their respective weight which becomes their tentative distance and chooses a vertex having least tentative distance. Now in the same way the distances of all other vertices connected with this particular vertex will be assigned or updated. If the calculated distance of a vertex becomes larger than the previous tentative distance, the value will not be updated whereas if the calculated value is lesser than the previous tentative value, the value will get updated. For a problem where the destination of the graph is given, the algorithm stops when it finds the final updated value of the destination vertex. For problems of finding the smallest path of all the vertices from the source vertex, the algorithm stops when the final distance all the vertices are found.

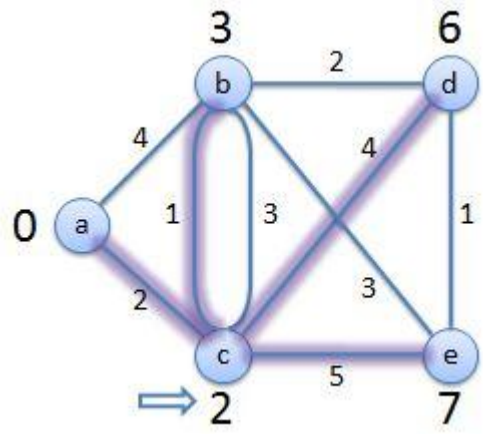
We will use a weighted graph of 5 vertices and 8 edges to show how the algorithm works.



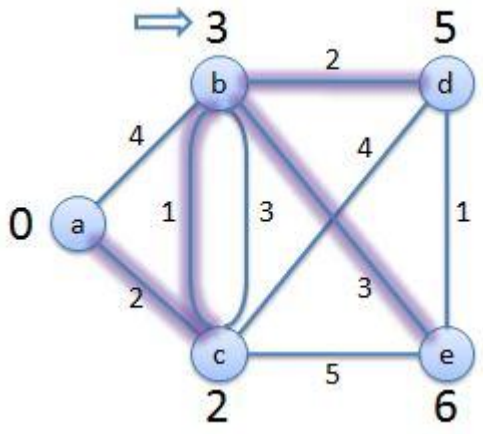
(a) The original graph where cost for source is assigned as 0 and ∞ is assigned as the costs of other vertices.



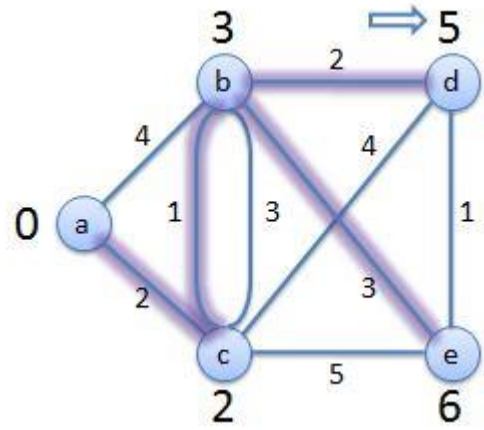
(b) From source a, vertices b and c are reachable. So respectively the costs of vertex b and vertex c are updated as 4 and 2.



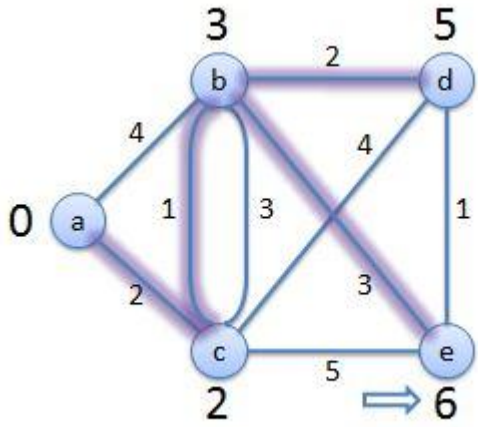
(c) Here vertex c having the smallest weight gets chosen. From c vertices d and e are reachable with cost of 6 and 7 respectively which are smaller than ∞. For vertex b, it is accessible from c with a cost of 3, which is smaller than the previous value 4.



(d) Here vertex b is picked. Vertex d and vertex e are accessible from vertex b with costs which are lower than their previous costs. So the costs for both vertex d and vertex e are updated.



(e) In this step vertex d has the lowest cost among the other vertices which are unvisited or whose costs are not finalized. Vertices b, c and e are accessible but their costs are not updated. If we want to update the cost of c from d the cost will be 5+4 = 9 which is larger than the current cost of vertex c. Same goes for vertex b and vertex e.



(f) Vertex e is chosen. The costs of any connected vertices are not updated.

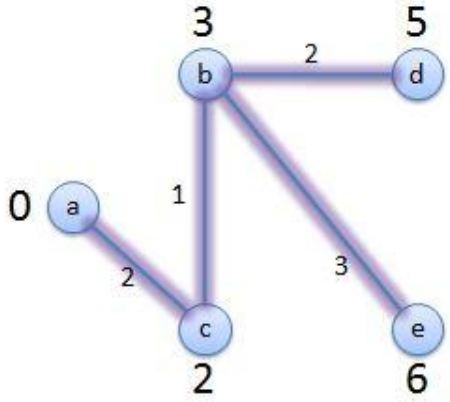


Figure 3.3: Final Shortest paths found using Dijkstra's Algorithm

3.3.2.3 Time Complexity

If binary heap is used for Priority Queue implementation the time complexity will be O(E+V)*O(LogV) which is O((E+V)*LogV) = O(ELogV) as decrease-key operation takes O(Log n) time. Using Fibonacci Heap the complexity can be reduced to O(E + VLogV) as for decrease-key operation Fibonacci Heap takes O(1) time [5].

3.3.3 Real Life Application

- Geographical maps, Google maps, IP routing, telephone network etc use this algorithm.

3.4 Bellman-Ford algorithm for Single Source Algorithm

3.4.1 History

Bellman-Ford algorithm was first published in 1958 by Richard Bellman though it was proposed by Alfonso Shimbel in 1955. In 1956 Lester Ford Jr published the same algorithm. This is why the algorithm is called Bellman-Ford algorithm.

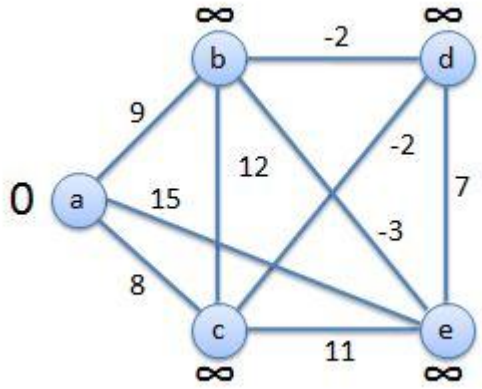
3.4.2 Algorithm

3.4.2.1 Pseudocode of Bellamn-Ford algorithm

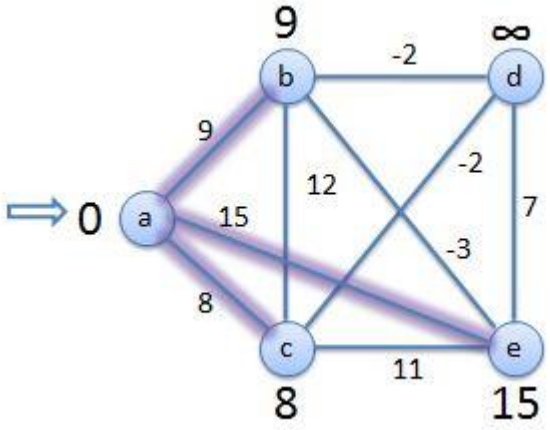
```
Relex(u,v,e)
        if d.v>d.u+e(u,v)
                d.v=d.u+e(u,v)
                r.v=u
Single_source(vertex)
        for i=1 to vertex
                d.i=10000
                r.i=-1
        d.0=0
Bellmanford_algo(e,vertex)
        Single_source(vertex)
        for i=0 to vertex-2
                for u=0 to vertex-1
```

```
                    for v=0 to vertex-1
                            if e(u,v)!=0
                                    Relex(u,v,e)
            for u=0 to vertex-1
                    for v=0 to vertex-1
                            if d.v>d.u+e(u,v)
                                    return False    // Negative cycle is detected
            return True
```

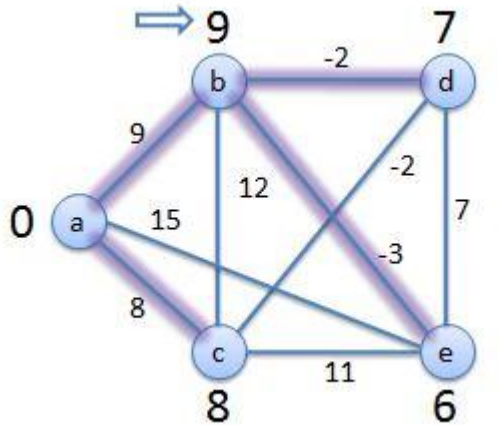
3.4.2.2 Graphical representation and description



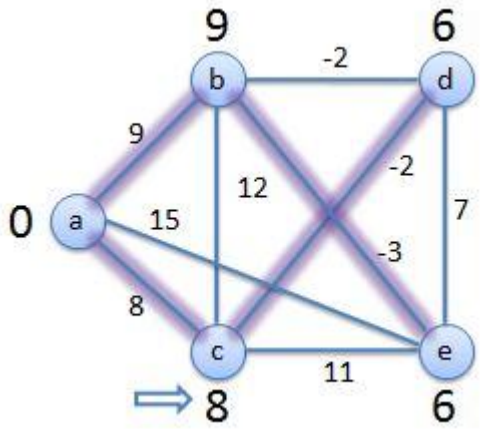
(a) The original graph where cost for source is assigned as 0 and ∞ is assigned as the costs of other vertices.



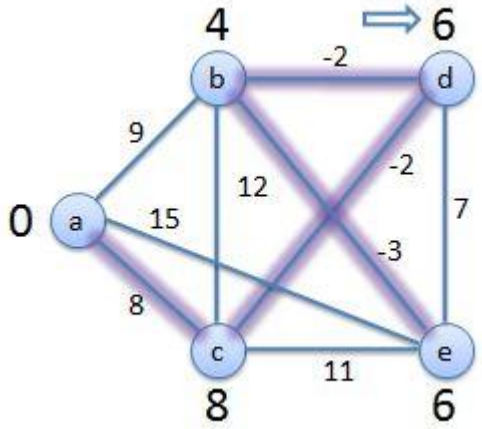
(b) From source a, vertices b and c are reachable. So respectively the costs of vertex b and vertex c are updated as 9 and 8.



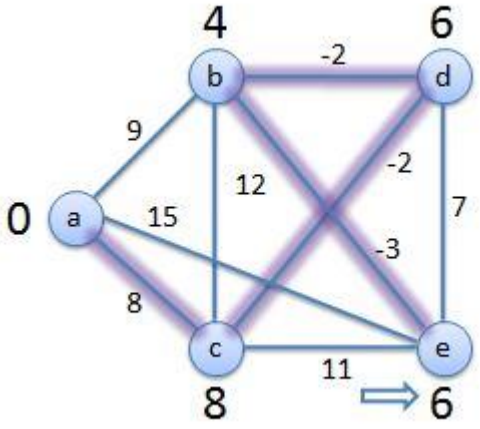
(c) Pick any of the vertices between b and c. Here b is chosen. From b vertices d and e are reachable with cost of 7 and 6 respectively which are smaller than ∞. So the distances are updated.



(d) Next c is chosen and the distance of the vertices from c are calculated. Simultaneously it checks if there is any negative cycle.



(e) As the next vertex, d is chosen. In the same way the connected distances are calculated and updated only when the distances are smaller than the previous one.



(f) In the same way e is chosen and worked like the previous steps. After this iteration all the distances are updated as the smallest distances among them. This iteration will be done V-1 times.

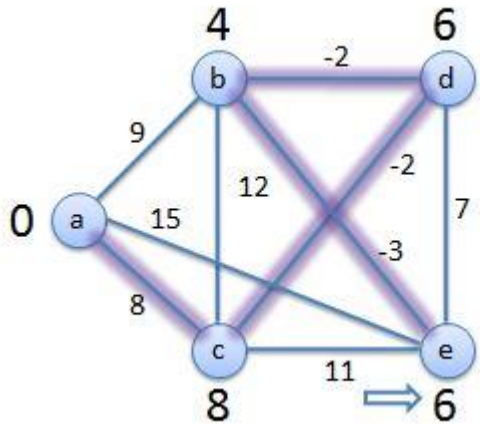


Figure 3.4: Final Shortest paths for 1st iteration using Bellman-Ford Algorithm

3.4.2.3 Time Complexity

Time complexity of Bellman-Ford is higher than that of Dijkstra's Algorithm. This is because it relaxes all the edges rather than the connected edges and also it checks for negative cycles. So time complexity becomes O(VE).

3.4.3 Real Life Application

- This algorithm is used in distance-vector routing protocol and within a system it is used as interior gateway routing [6].

3.5 Knapsack

Knapsack is an optimization problem. In the knapsack problem a set of items are given with their respective weights and values followed by the storage capacity that needs to be filled by the given items which is called knapsack. Maintaining the capacity the knapsack should be filled with the items in such a way that the sum of the values are maximized.

3.5.1 History

By looking deep to the history of knapsack it is found that mathematician Tobias Dantzig's (1884–1956) worked problems were similar to the knapsack properties. His works were known to as commonplace problem. As the problem is related to fill the knapsack with the most valuable goods that is why the problem is named 'Knapsack problem'.

3.5.2 Types

3.5.2.1 Fractional Knapsack

The idea behind fractional knapsack is that it allows to take fraction of item if the whole item exceeds the capacity. In fractional knapsack the cost per weight ratio is used to sort the given goods in non decreasing order. One by one the goods are picked to fill the capacity of the bag. By following this strategy it gives the best optimal result [1].

3.5.2.1.1 Pseudocode of fractional knapsack

```
element[],unit[],w[],cost[] //These are global array used in this algo
Insertion_sort(item)
    for j=1 to item-1
        key=unit.j
        key1=cost.j
        key2=w.j
        key3=element.j
        i=j-1
        while i>=0 and unit.i<key
            unit.i+1=unit.i
```

```
                cost.i+1=cost.i
                w.i+1=w.i
                element.i+1=element.i
                i=i-1
            unit.i+1=key
            cost.i+1=key1
            w.i+1=key2
            element.i+1=key3
Knapsack_algo(item,sack)
    for i=0 to item-1
            unit.i=cost.i/w.i
    Insertion_sort(item)
    for i=0 to item-1
            if bag>=sack
                break
            elseif bag+w.i<=sack
                bag+=w.i
                total+=cost.i
            else
                temp=sack-bag
```

bag+=temp

total+=temp*unit.i

3.5.2.1.2 How it works

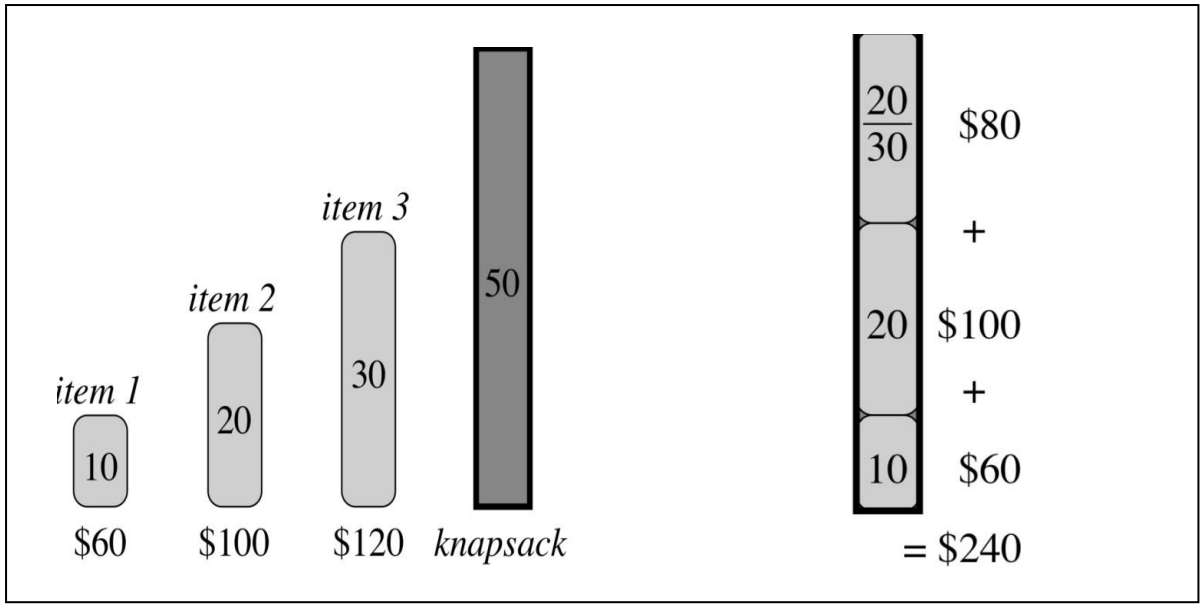


Figure 3.5: Fractional knapsack's working process

Firstly items are sorted in ascending order according to the cost per weight ratio. The ratios of the items are respectively 6, 5, 4 and the knapsack capacity being 50 kg. First item 1 is chosen. Item 2 has the second highest ratio so it is placed in the bag. The bag's remaining capacity becomes 50-30=20 kg but item 3 is of 30kg. As item 3 exceeds the capacity of the bag, here fractional knapsack takes the fraction of item 3. So it will take remaining capacity/weight of item 3 which is 20/30 kg and the cost becomes (20/30)*$120 = $80. Thus the bag is filled and the cost becomes $60+$100+$80=$240 which the highest and most optimal cost.

3.5.2.2 0-1 Knapsack

The objective of 0-1 knapsack is that it will take given items in such a way that maximizes the profit and does not exceed the capacity of the sack. The main concept is that the algorithm takes item as a whole or not at all concentrating on the sack's capacity. So weight of the items has to be less or exactly equal to the boundary of the sack. It strictly prevents to take the fraction of an item. So 0-1 knapsack works in binary way such as keep it or leave it. This is the reason of the algorithm being called as 0-1

knapsack. Unlike fractional knapsack, 0-1 knapsack does not follow greedy algorithm. As the algorithm does not follow greedy approach and so optimal solution is never guaranteed [1].

3.5.2.2.1 Pseudocode of 0-1 knapsack

```
element[],w[],cost[] //This are global array used in this algo
Insertion_sort(item)
        for j=1 to item-1
                key1=cost.j
                key2=w.j
                key3=element.j
                i=j-1
                while i>=0 and cost.i<key1
                        cost.i+1=cost.i
                        w.i+1=w.i
                        element.i+1=element.i
                        i=i-1
                cost.i+1=key1
                w.i+1=key2
                element.i+1=key3
Knapsack01_algo(item,sack)
        Insertion-sort(item)
        for i=0 to item-1
                if bag>=sack
                        break
                elseif bag+w.i<=sack
                        bag+=w.i
                        total+=cost.i
                else
                        y=i+1
                        for ii=y to item-1
                                if bag+w.ii<=sack
                                        bag+=w.ii
                                        total+=cost.ii
```

3.5.2.2. 2 How it works

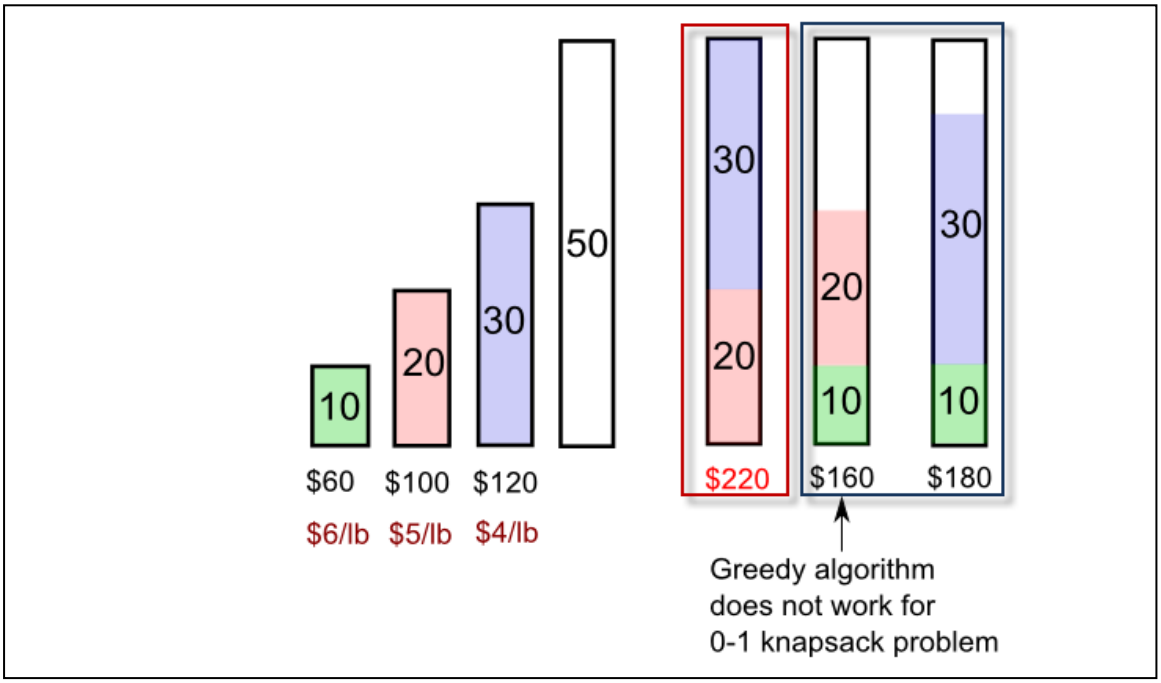


Figure 3.6: 0-1 Knapsack's working process

In a knapsack problem every items are associated with their weight and cost. The goal is to take the items within the sack's capacity and to maximize the cost. In the first run the problem can be solved using the properties of Greedy algorithm. As like fractional knapsack idea, the items are sorted in decreasing order by the ratio of their cost per weight. So the first item is the item having weight 10kg. 0-1 knapsack will take the item and then check if the capacity is full. The algorithm then picks the second item with weight 20kg and put it into the sack. After that, when it checks the sack capacity it is found that the remaining capacity becomes 50-30=20kg and that the weight of the third item exceeds that capacity. 0-1 Knapsack cannot take only a part of the third item. So necessarily it will leave the third item and terminate the program. The final result shows the cost $160 which is not optimal nor the maximum cost. Clearly 0-1 Knapsack problem cannot be solved by following Greedy Algorithm.

Here comes the second run where 0-1 Knapsack problem is solved in other way where the cost is maximized. The items are sorted in decreasing order by their cost. The item having the highest cost is picked and inserted into the sack. If the sack capacity is not filled, it then chooses the next item having second highest cost which is the item with cost $100. If the item does not exceed the sack capacity it is inserted or else the algorithm

searches for next item that can be fit to the remaining capacity. As the given sack is filled with first two items, the program will stop here. Here the final cost is $120+$100=$220. This cost is clearly greater than the one solved by using Greedy algorithm.

The algorithm is terminated when the sack capacity is full or there is no other item in the list to fill the remaining capacity.

3.6 File compression using Huffman Coding

3.6.1 History

The history behind the invention of Huffman coding is very interesting. It was invented by David Huffman in 1951. He submitted it as final term paper to his professor Robert Fano who was having trouble in solving the problem similar to this.

3.6.2 Algorithm

It reads characters from the file and calculates frequencies for each character. It prepares leaf node for each character and puts it in priority queue. Then extracts each node, combine their frequencies and again puts it in the priority queue. This process stops when there is only one node. From this tree the compressed file size is calculated and compared with the original size.

3.6.2.1 Pseudocode of Huffman coding

```
a[],b[]      //Global array
before_insert()
        freopen("huff.txt","r",stdin)
        while c=getchar() !=EOF
                j=(int)c
                a.j=a.j+1


ori_bit()
        i=cc
        j=ceil(log i/log 2)
        pp=(int)j
```

```
del()
        if start!=NULL
                t=head
                head=head->NULL
                t->next=NULL
        return t

summ(j)
        tt=tt+j

printArray(ints,countt,lan)
        j=(countt-1)*ints.len-1
        summ(j)

printPathsRecur(node,countt,path,pathLan)
        if node==NULL
                return
        path.pathLan=node->freq
        pathLan++
        countt++
        if node->left==NULL and node->right==NULL
                printArray(path,countt,pathLan)
        else
                printPathsRecur(node->left,countt,path,pathLan)
                printPathsRecur(node->right,countt,path,pathLan)

printpaths(head)
        printPathsRecur(start,0,path,0)

showchild(head)
        if temp==NULL
                return
```

```
        if temp->left!=NULL
                print(temp->name,temp->freq,temp->left->name,temp->left-
>freq,temp->right->name,temp->right->freq)
        else
                print(temp->name,temp->freq)
        showchild(temp->left)
        showchild(temp->right)

    main
        head=NULL
        before_insert()
        for i=0 to 1000
                if a.i!=0
                        b.0=(char)i
                        j=a.i
                        strncpy(z->name,b,1)
                        z->freq=j
                        z->next=NULL
                        z->left=NULL
                        z->right=NULL
                        cc++
                        insert(head,z)
        show()
        ori_bit()
        while(head->next!=NULL)
                p=del(head)
                q=del(head)
                strcpy(r->name,"-")
                r->freq=p->freq+q->freq
                r->next=NULL
                r->left=p
                r->right=q
                insert(head,r)
```

```
final=head->freq
j=pp*final
printpaths(head) // tree teavarse and count the comressed number

if tt<j //tt=Compressed number
        print(compresed)
        o=final*8
        oo=((o-tt)/o)*100 //Percentahe compressed
else
        print(not compresed)
showchild(head)
```

3.6.2.2 Graphical representation and description

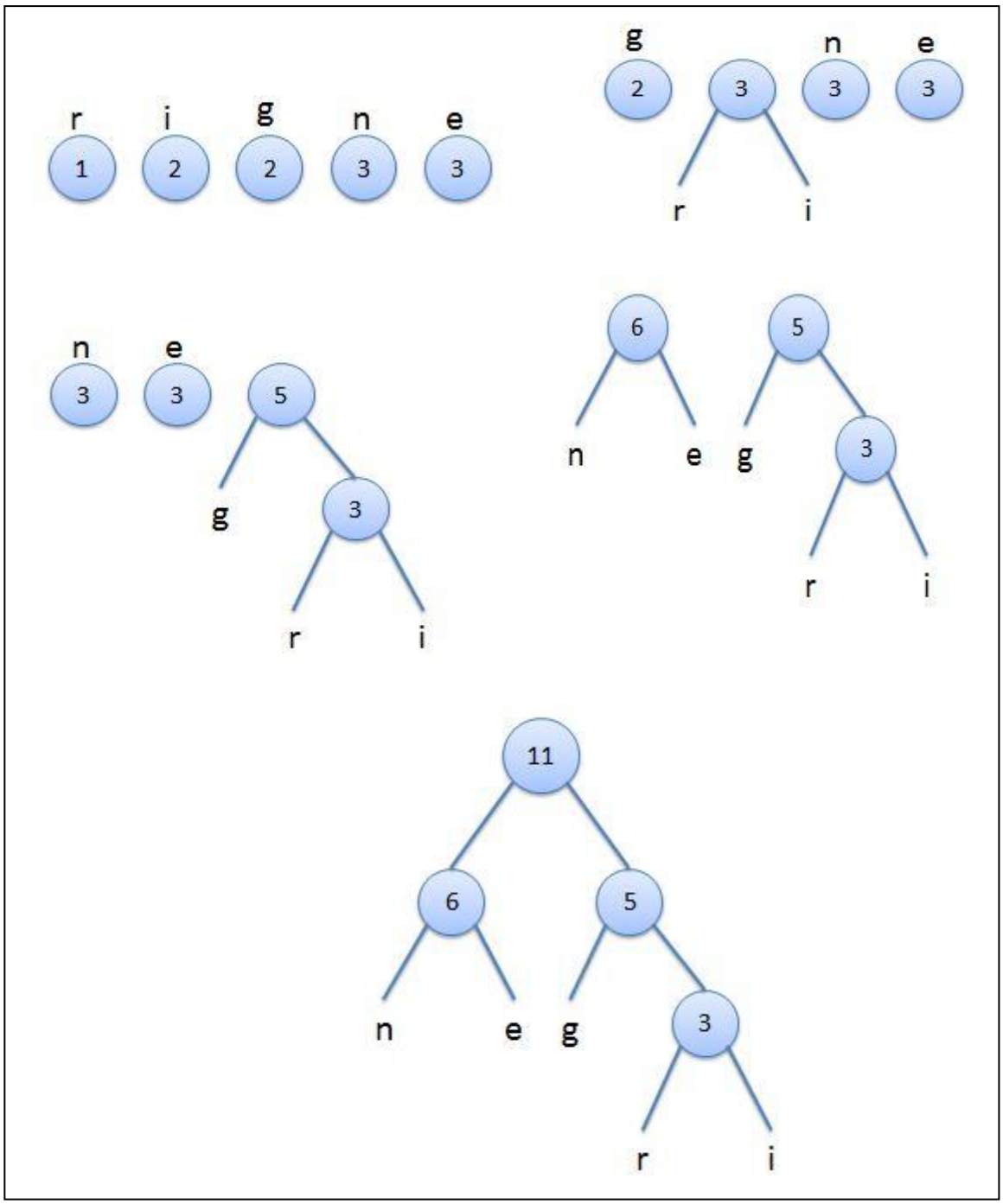


Figure 3.7: Huffman tree construction process

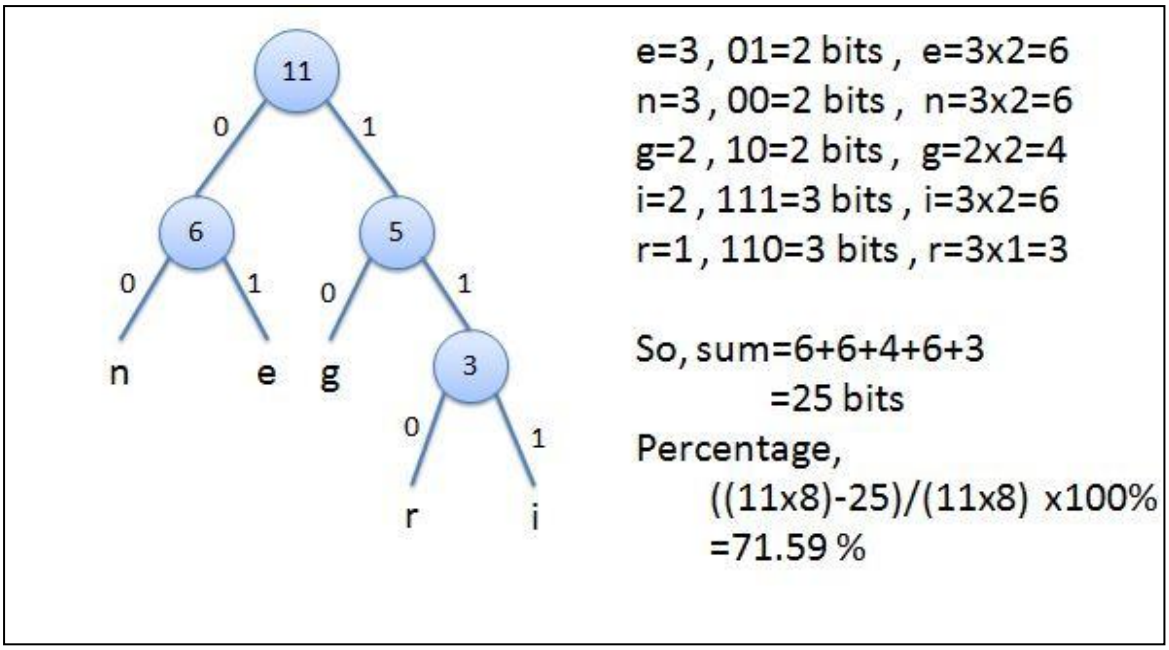


Figure 3.8: Compression calculation of Huffman coding

3.6.2.3 Work Process

(a) Inserting characters:

The algorithm first reads a file and count different character's frequencies which are stored in an array. Iteratively the characters are inserted along with their frequencies in a linked list in ascending order.

(b) After insertion the numbers of all the new nodes are counted and converted into how many bits they need to be represented. This is needed to compute the compression size. For 'engineering' the algorithm has created 5 nodes. 5 is then converted into bits by log5/log2 , takes the ceil value (3) and it is stored.

(c) Creating child:

From the linked list, the first and second nodes are picked and their frequencies are summed. So node 'r' and node 'i' are picked then inserted into a new node with no name and summed frequency. Following that this new node has a left child being the node 'r' and right being the node 'i'. Then this new node is inserted into the linked list. This process is continued until there is only one node in the linked list.

(d) Evaluating compression:

The root of the final tree structure has frequency of 11. This is multiplied by 3 and it is stored. The result becomes 33 bit.

Secondly the frequency of each character is multiplied by their length in the tree structure. Here character ‘e’ with frequency 3 is multiplied by its length in the tree 2. This is done for all the different characters and all their results are added. This result being 25 bit.

Clearly the file has been compressed from 33 to 25 bit. Percentage of compression is then calculated which is 71.59% compressed [7].

Chapter 4

Experiments

4.1 Experimental Results

4.1.1 Minimum Spanning Tree

We randomly generated undirected weighted graphs with various nodes and edges then performed Prim's and Kruskal's Algorithm with these graphs. The results are following.

Table 4.1: Time complexity (in second) comparison between Prim's and Kruskal's Algorithm

Number of nodes	**Prim's Algorithm (sec)**	**Kruskal's Algorithm (sec)**
100	0.016	0.187
300	0.046	0.296
500	0.068	0.374
700	0.078	0.436
900	0.084	0.546
1000	0.094	0.592

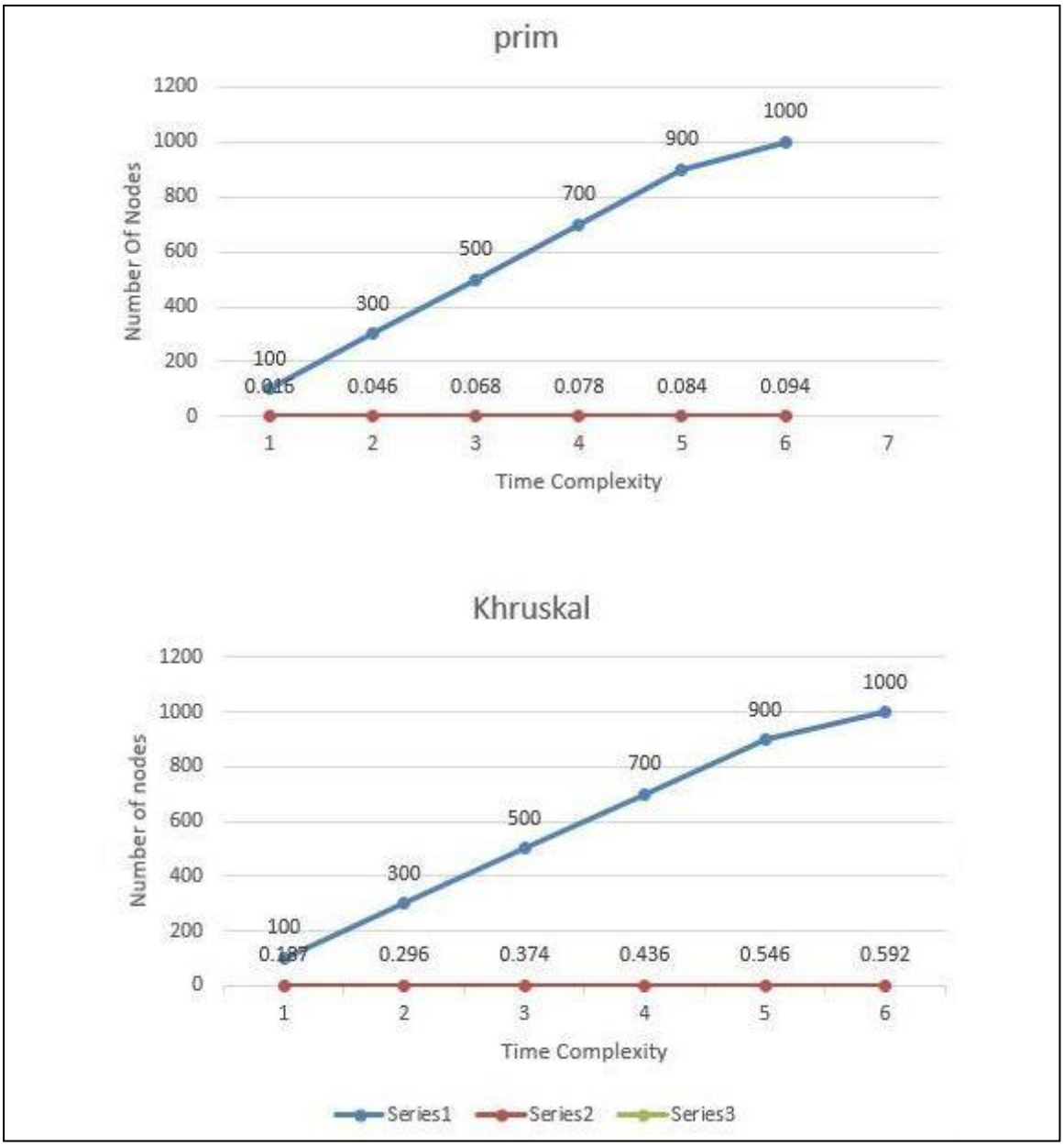


Figure 4.1: Time complexity (in second) comparison shown in graph between Prim's and Kruskal's Algorithm

Results Found:

- Time complexity of both Prim's Algorithm and Kruskal's Algorithm is O(ElogV)
- Prim's algorithm works faster in really dense graph whereas Kruskal's algorithm performs better in typical sparse graphs [4].

4.1.2 Single Source Shortest Path

Table 4.2: Time complexity (in second) comparison between Bellman-Ford and Dijkstra's Algorithm

Number of nodes	Bellman Ford (sec)	Dijkstra (sec)
100	0.031	0.140
300	0.171	0.265
500	0.261	0.437
700	1.341	0.561
900	2.683	0.749
1000	3.713	0.811

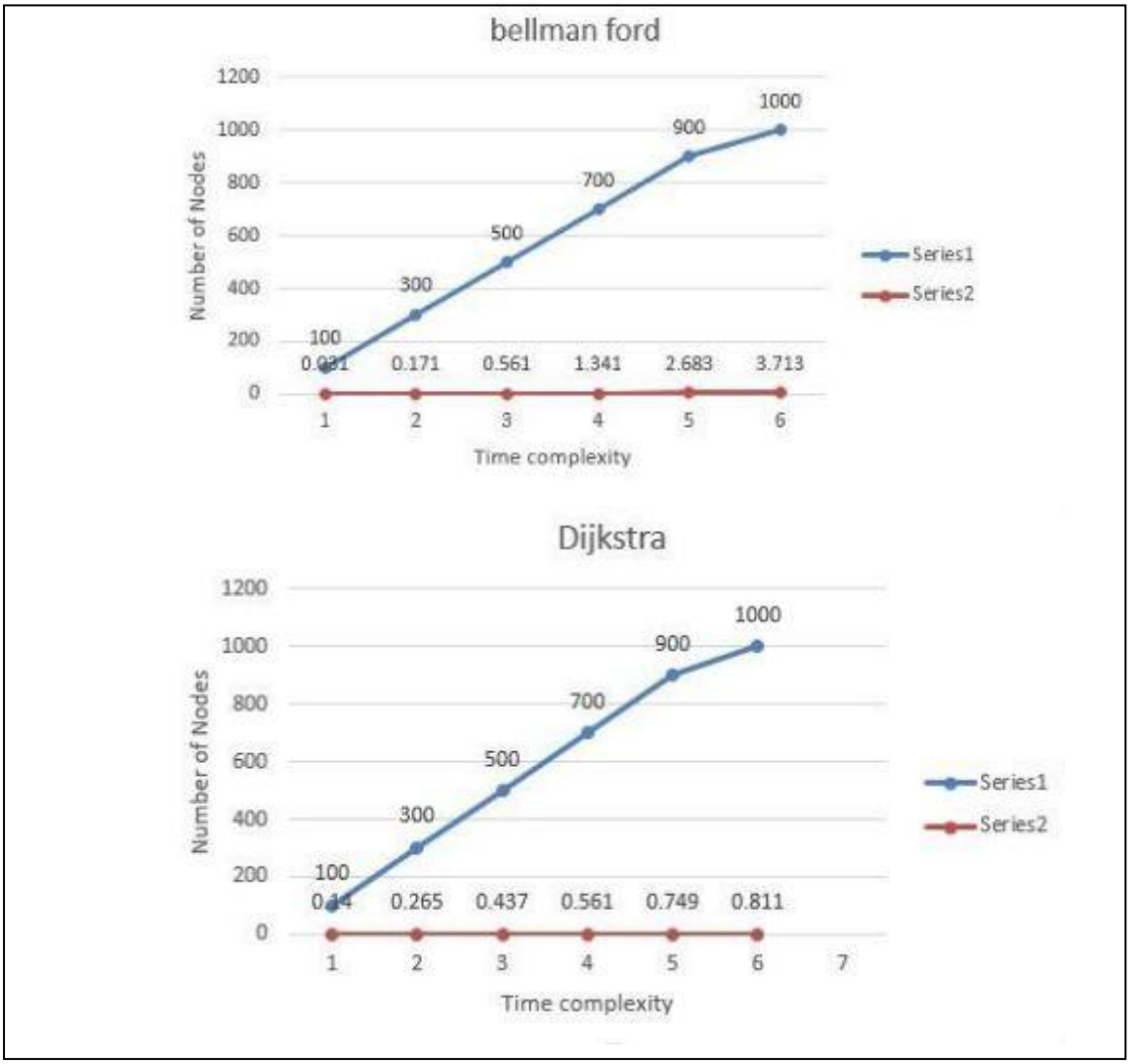


Figure 4.2: Time complexity (in second) comparison shown in graph between Bellman-Ford and Dijkstra's Algorithm

Results Found:

- In one iteration Bellman-Ford Algorithm relaxes all edges simultaneously but Dijkstra's Algorithm being a greedy algorithm relaxes only the neighboring edges from the particular vertex.
- Bellman-Ford Algorithm works for graphs having negative weights whereas Dijkstra's Algorithm cannot function in this kind of graphs.
- Bellman-Ford Algorithm can detect negative cycles, Dijkstra's Algorithm doesn't have this feature.
- Bellman-Ford's time complexity is higher which is O(VE), Dijkstra's Algorithm's time complexity being O(ELogV) [5].

4.2 Experiments with Different Instance

4.2.1 Piggy bank problem solved with 0-1 Knapsack

Problem: Suppose we have different types of coins. Those coins have specific quantity and volume for example: the number of 5 taka coin is 20 and each coin's volume is 1.06. We have a given piggy bank that we have to fill with different type of coin and the sum of the coins has to be maximum.

Table 4.3: Problem instance for 0-1 knapsack

Item	Quantity	Volume
5 taka coin	20	1.06
2 taka coin	10	0.72
1 taka coin	40	0.55
0.50 taka coin	30	0.53
0.25 taka coin	10	0.34
0.10 taka coin	10	0.25

4.2.1.1 Algorithm

```
Coin(item,sack)
    Insertion_sort(item)
    if i=0 to item-1
        cc=quantity.i
        if bag>=sack
            break
```

```
elseif bag+(quantity.i*volume.i)<=sack
    bag+=volume.i*quantity.i
    total+=cost.i*quantity.i
elseif bag+(quantity.i*volume.i)>sack
    for ss=0 to cc-1
        quantity.i=quantity.i-1
            if bag+(quantity.i*volume.i)<=sack
                bag+=volume.i*quantity.i
                total+=cost.i*quantity.i
                break
```

4.2.1.1 Work Process

1. All the coins are sorted in descending order according to the costs of each coin.
2. First it checks if the numbers of coins taken exceeds the capacity of the piggy bank.
3. If not, it will take all the coins available and puts in the bank.
4. If the volume of the certain number of coins exceeds the volume of the bank, the algorithm tries to take some of the coins.
5. The algorithm calculates total cost and numbers of each coin taken.

4.2.2 Meal Problem solved with Fractional knapsack

Problem: Suppose we have number of food items and limited consumption capacity. Our goal is to eat every given food item in a way that will maximize the calorie consumed.

Table 4.4: Problem instance for fractional knapsack

Item	Amount(kg)	Calorie
A	20	2.06
B	10	5.72
C	40	7.55
D	30	5.53
E	10	2.34
F	10	1.25

4.2.2.1 Algorithm

```
Meal_algo(item,sack)
    for i=0 to item-1
        unit.i=cost.i/w.i
    for i=0 to item-1
        per.i=(w.i/weightsum)*sack
    Insertion-sort(item)
    for i=0 to item-1
        if bag>=sack
            break
        else
            bag+=per.i
            total+=per.i*unit.i
            amountsum+=per.i
```

4.2.2.2 Work Process

1. The meal items are sorted in descending order according to their cost per weight ratio.
2. In relevant to the consumer's capacity, the maximum quantity of each item that the consumer can eat has been calculated. So now all the items quantity will be within the consumer's capacity.
3. It then checks if the number of items taken exceeds the consumer's capacity.
4. If not, it takes the whole item as calculated before because the items are already within the consumer's capacity.
5. The algorithm stops when the consumer's capacity is reached.

4.3 Innovation

Solving MST with Greedy Approach and Genetic Algorithm

4.3.1 Properties of Genetic Algorithm used

The core properties:

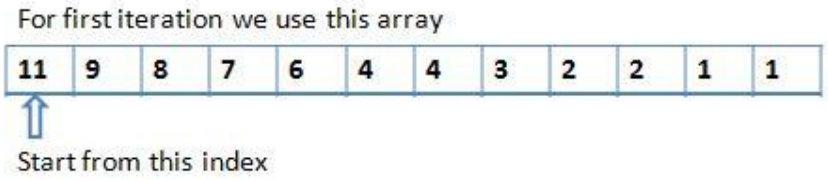
- Initial population
- Fitness function
- Selection

- Crossover
- Mutation

Among these properties we used:

- Initial population
- Fitness function
- Mutation

1. Initial candidate

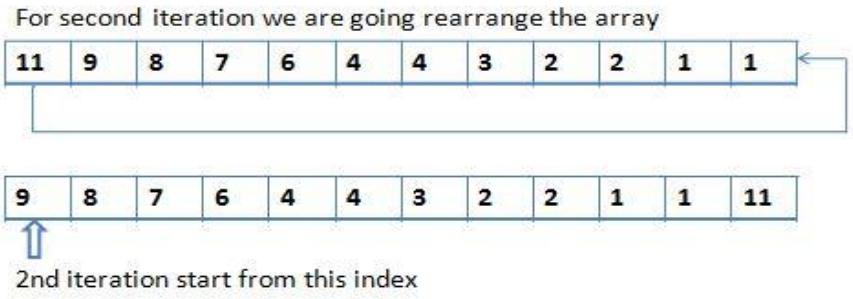


The weights of the given graph are sorted in descending order and stored in an array. Basically this array is the initial candidate. In each iteration the arrangement of the array is changed and thus new candidates are generated. All the different candidates produce population in the problem.

2. Fitness function

Each candidate generates a spanning tree. The sum of all weights of edges in that spanning tree is the fitness of that particular spanning tree. Every other candidates generate different spanning tree thus all their fitness are calculated. As the goal is to determine the minimum spanning tree, so the least fitness value is chosen with its corresponding spanning tree.

3. Mutation

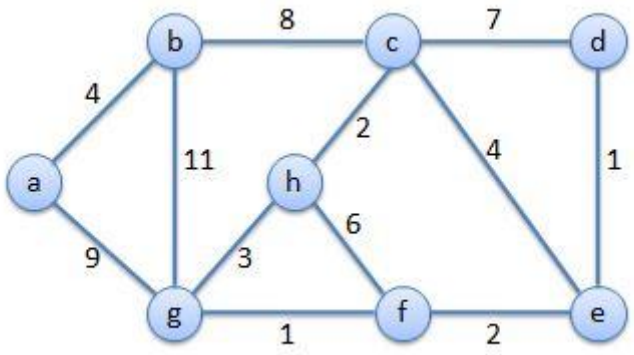


In every iteration one element of the initial candidate is changed by performing left rotation operation. By doing so it creates a variation in the candidate pool. This allows to create spanning tree of different fitness and creates large search space.

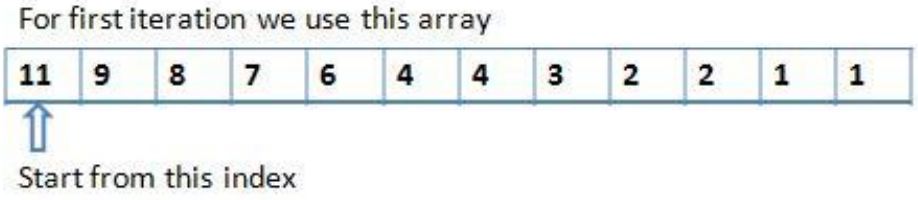
4. Termination

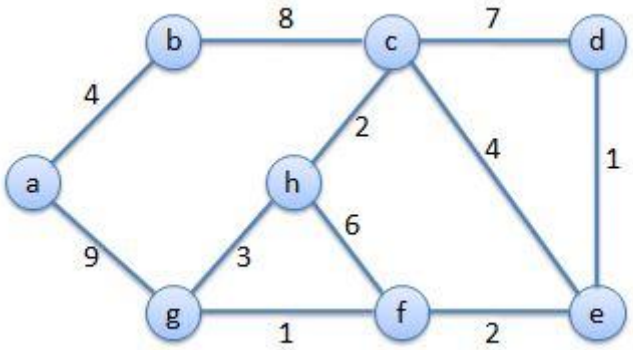
As the mutation happens by doing left rotation operation so when the algorithm has performed iteration which is equal to the number of edges in the graph, the algorithm is terminated and the solution is found [3].

4.3.2Graphical Representation and Description

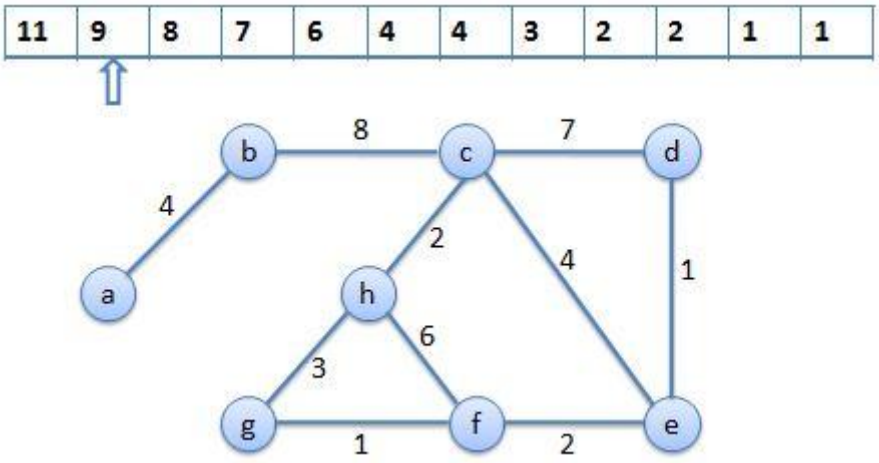


(a) The original graph

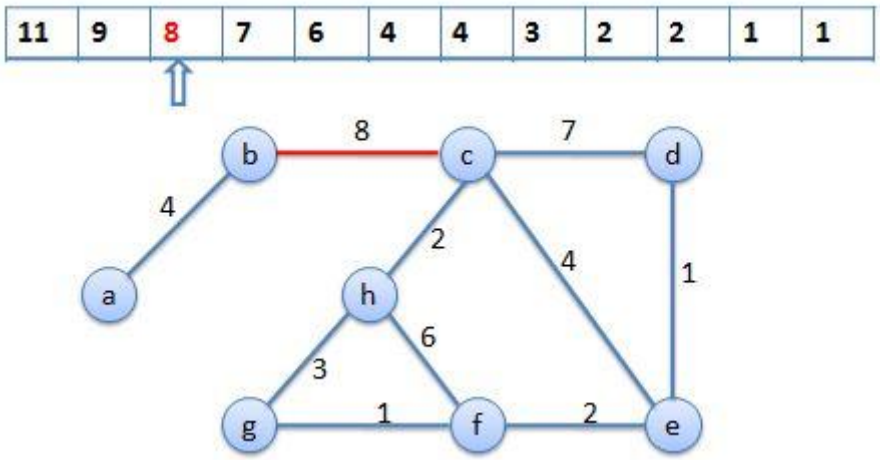




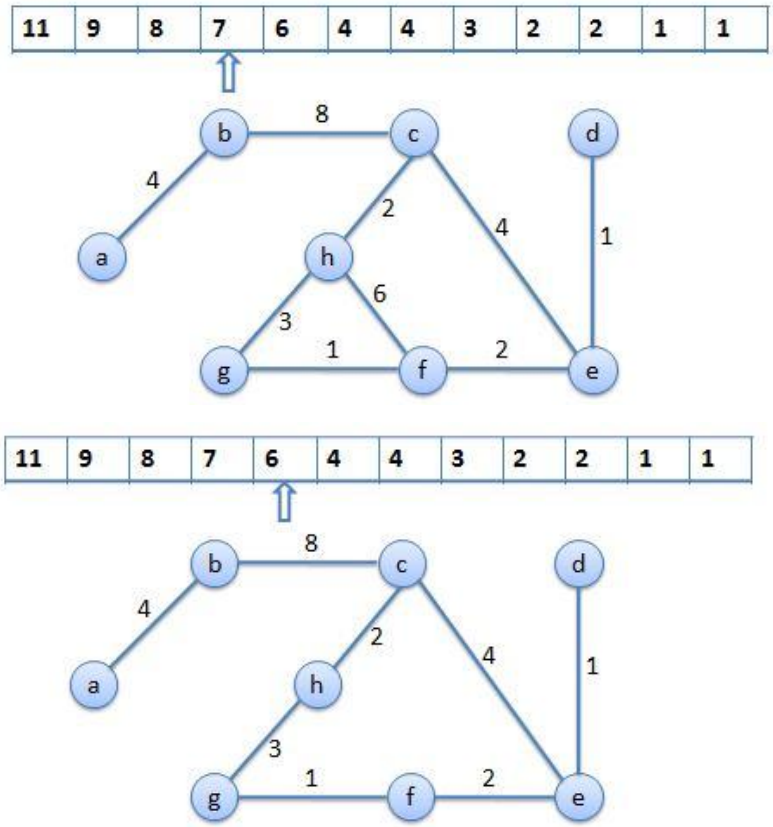
(b) The algorithm first chooses first element from the initial candidate. It eliminates the edge and checks if still the graph is connected. If the graph becomes disconnected, the algorithm does not eliminate the edge.

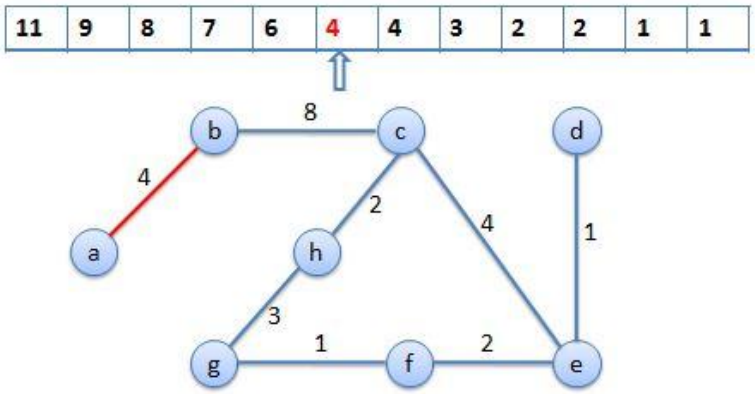


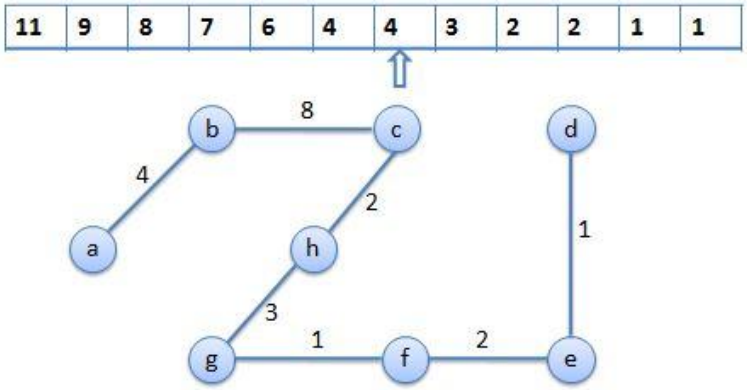
(c) Then it chooses the next element from the candidate and checks the same.

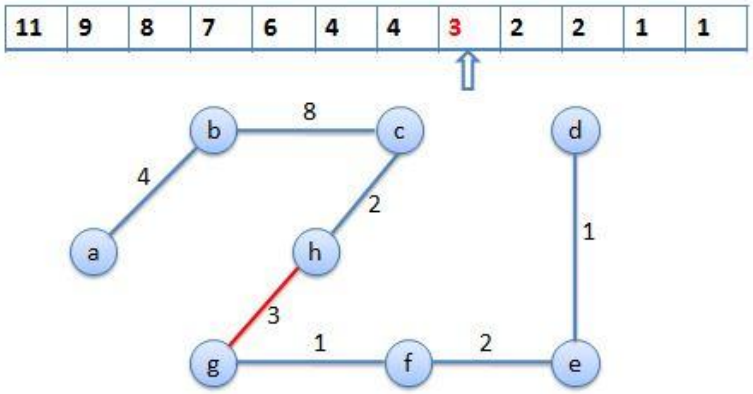


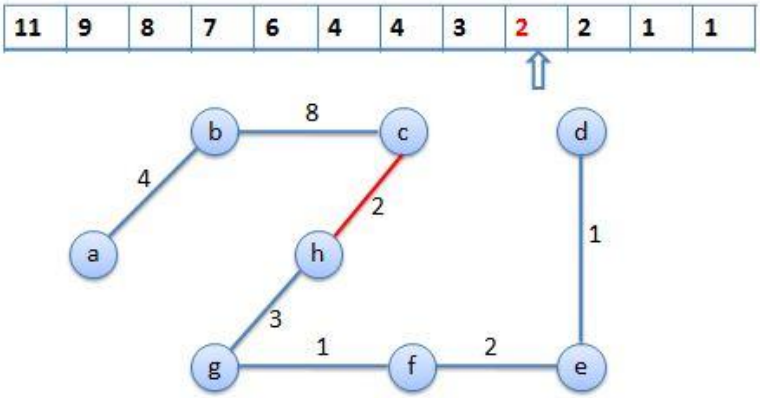
(d) Here 8 is chosen but eliminating this edge will cause an island in the graph so the algorithm doesn't eliminate it. The process is continued until the last element of the candidate is checked.




11 9 8 7 6 4 4 3 2 2 1 1
a
b
c
d
e
f
g
h
8
4
2
4
1
3
1
2


11 9 8 7 6 4 4 3 2 2 1 1
a
b
c
d
e
f
g
h
8
4
2
1
3
1
2


11 9 8 7 6 4 4 3 2 2 1 1
a
b
c
d
e
f
g
h
8
4
2
1
3
1
2


11 9 8 7 6 4 4 3 2 2 1 1
a
b
c
d
e
f
g
h
8
4
2
1
3
1
2

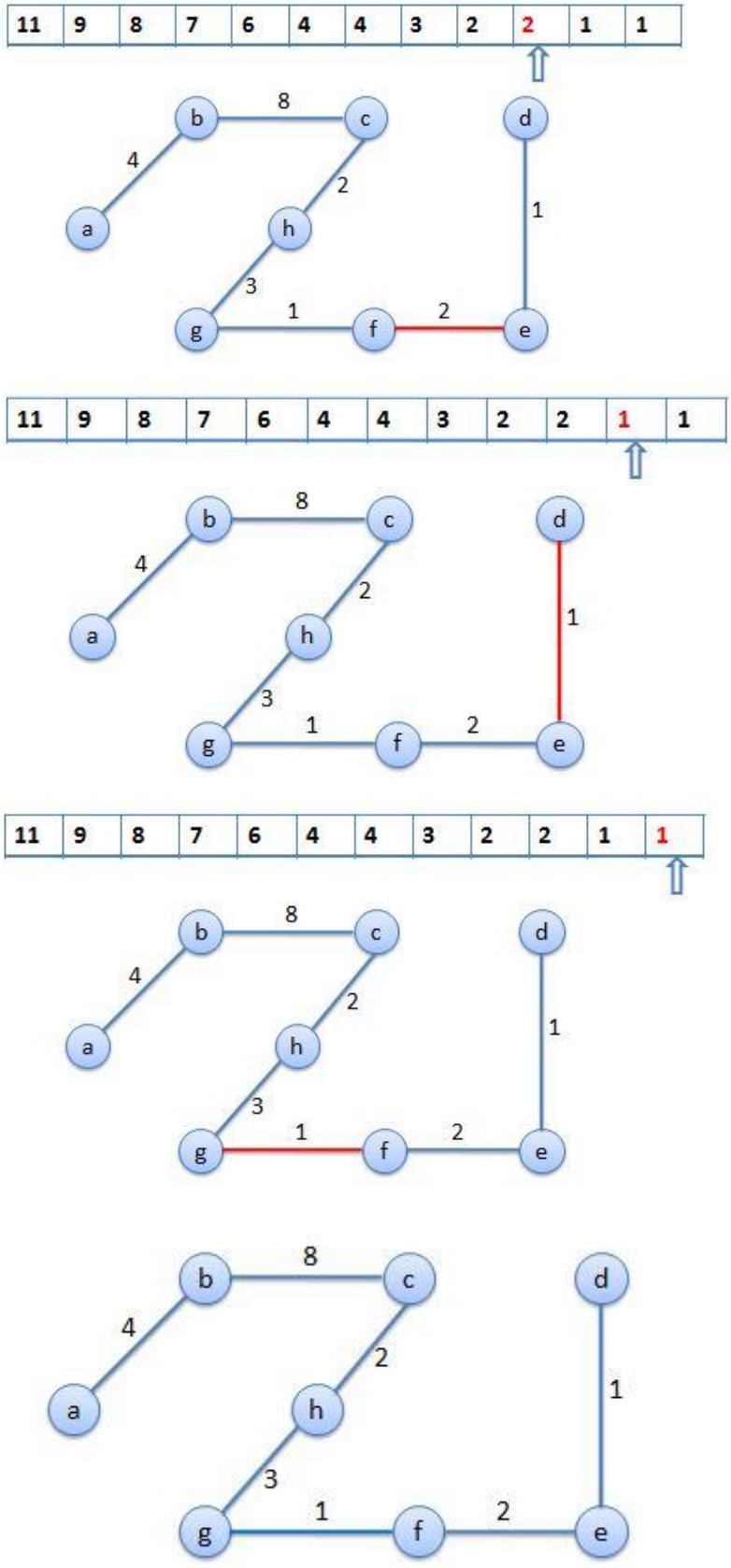


Figure 4.3: The spanning tree after first iteration

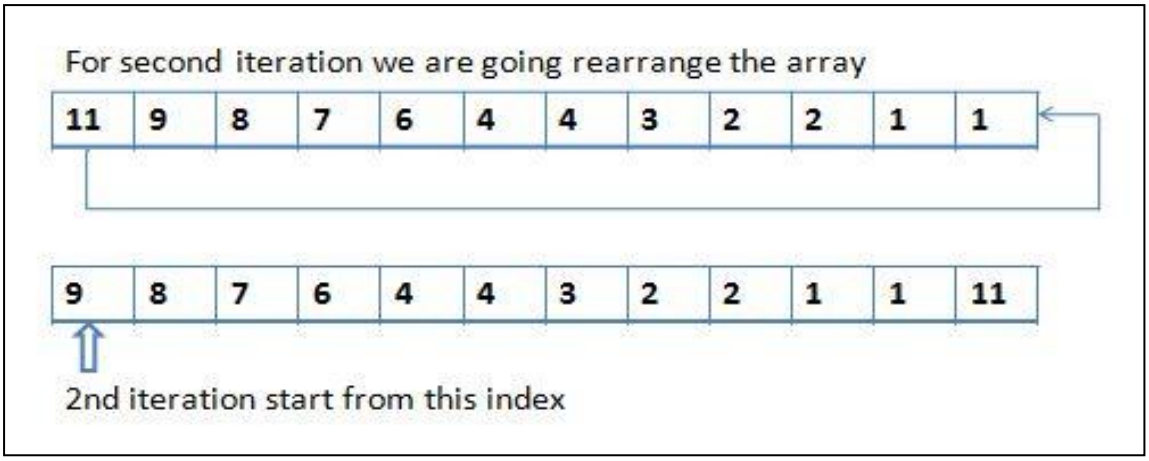


Figure 4.4: Left rotation to change the combination of initial candidate after first iteration.

Chapter 5

Conclusion and Future Works

Conclusion and current status

We studied some Greedy Algorithms such as Knapsack problem, Huffman coding. We explored some area of graph theory such as MST, single source shortest path. We tried to combine Greedy and Genetic Algorithm to implement an algorithm that solved MST problem. Our current implementation of GA takes longer time compared to any other algorithms. Also, it is not matured enough to work with huge data set.

Future works

In future we want to implement GA with all of its core properties to get better result. We will make it work for large data set and solve problem within reasonable time. We hope it will be efficient enough to work with directed graph in the cases where Kruskal's and Prim's Algorithm might have some limitations.

References

1. Charles E. Leiserson, Clifford Stein, Ronald Rivest, and Thomas H. Cormen, “Introduction to Algorithms”, MIT Press, 3rd Edition, 2009
2. Greedy Algorithm: https://en.wikipedia.org/wiki/Greedy_algorithm
3. Genetic Algorithm: https://en.wikipedia.org/wiki/Genetic_algorithm
4. Difference between Prim’s and Kruskal’s Algorithm: https://stackoverflow.com/questions/*14144279*/difference-between-prims-and-kruskals-algorithms
5. Dijkstra’s Algorithm’s Time Complexity: https://stackoverflow.com/questions/*26547816*/understanding-time-complexity-calculation-for-dijkstra-algorithm
6. Real life application of Bellman-Ford Algorithm: https://www.quora.com/Is-there-an-application-for-the-Bellman-Ford-algorithm
7. Huffman Coding: https://www.geeksforgeeks.org/greedy-algorithms-set-*3*-huffman-coding/