Batch: A3 **Roll No.: 16010121045**

Experiment No: 4

Grade: AA / AB / BB / BC / CC / CD /DD

Signature of the Staff In-charge with date

Title: Implementation of Single source shortest path by Greedy strategy

Objective: To learn the Greedy strategy of solving the problems for different types of problems

CO to be achieved:

CO 2 Describe various algorithm design strategies to solve different problems and analyse Complexity.

Books/ Journals/ Websites referred:

1. **1. Ellis horowitz, Sarataj Sahni, S.Rajsekaran," Fundamentals of computer algorithm", University Press**
2. **T.H.Cormen ,C.E.Leiserson,R.L.Rivest and C.Stein," Introduction to algortihtms",2nd Edition ,MIT press/McGraw Hill,2001**
3. **https://www.mpi-inf.mpg.de/~mehlhorn/ftp/ShortestPathSeparator.pdf**
4. **en.wikipedia.org/wiki/Shortest_path_problem**
5. **www.cs.princeton.edu/~rs/AlgsDS07/15ShortestPaths.pdf**

Pre Lab/ Prior Concepts:
Data structures, Concepts of algorithm analysis

Historical Profile:
Sometimes the problems have more than one solution. With the size of the problem, every time it's not feasible to solve all the alternative solutions and choose a better one. The greedy algorithms aim at choosing a greedy strategy as solutioning method and proves how the greedy solution is better one.
Though greedy algorithms do not guarantee optimal solution, they generally give a better and feasible solution.

The path finding algorithms work on graphs as input and represent various problems in the real world.

New Concepts to be learned: Application of algorithmic design strategy to any problem, Greedy method of problem solving Vs other methods of problem solving, optimality of the solution

Topic: GREEDY METHOD

Theory: The greedy method suggests that one can devise an algorithm that work in stages, considering one input at a time. At each stage, a decision is made regarding whether a particular input is in an optimal solution. This is done by considering the inputs in an order determined by some selection procedure. If the inclusion of the next input into the partially constructed optimal solution will result in an infeasible solution, then this input is not added to the partial solution. Otherwise, it is added. The selection procedure itself is based on some optimization measures may be plausible for a given problem. Most of these, however, will result in algorithms that generate suboptimal solutions. This version of the greedy technique is called the **subset paradigm**.

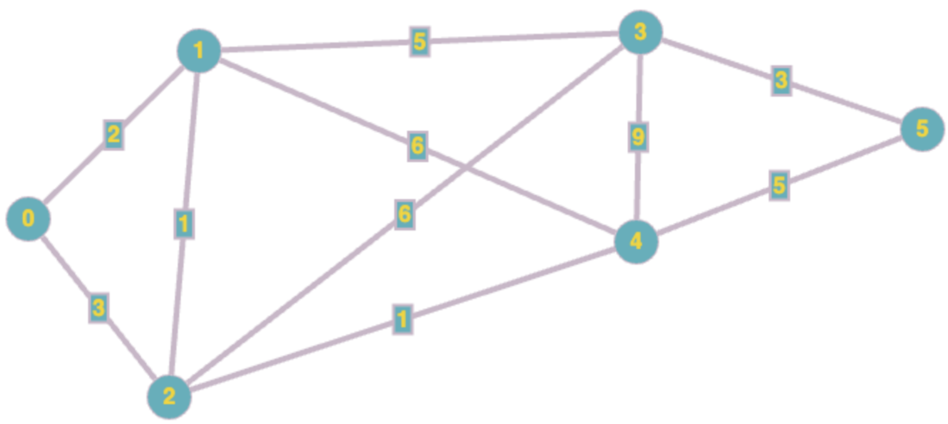
Control Abstraction:

```
SolType Greedy (Type s [], int n)

// a[1:n] contains the n inputs.

{SolType solution = EMPTY;
        // Initialize the solution.
        For (int i=1; I<=n; i++) {
                Type x = Select (a);

        If Feasible (solution, x)

        Solution = Union (solution, x) ;

}

return solution;

}
```

Algorithm:

```
Algorithm ShortestPaths(v, cost, dist, n)
// dist[j], 1 ≤ j ≤ n, is set to the length of the shortest
// path from vertex v to vertex j in a digraph G with n
// vertices. dist[v] is set to zero. G is represented by its
// cost adjacency matrix cost[1 : n, 1 : n].
{
    for i := 1 to n do
    { // Initialize S.
        S[i] := false; dist[i] := cost[v, i];
    }
    S[v] := true; dist[v] := 0.0; // Put v in S.
    for num := 2 to n − 1 do
    {
        // Determine n − 1 paths from v.
        Choose u from among those vertices not
        in S such that dist[u] is minimum;
        S[u] := true; // Put u in S.
        for (each w adjacent to u with S[w] = false) do
            // Update distances.
            if (dist[w] > dist[u] + cost[u, w])) then
                    dist[w] := dist[u] + cost[u, w];
    }
}
```

Example Graph:



Solution:

Vertex	Shortest Distance
0	0
1	2
2	3
3	7
4	4
5	9

Code:

```
#include <bits/stdc++.h>
using namespace std;
#define INF 0x3f3f3f3f

typedef pair<int, int> p;

void addEdge(vector<p> adj[], int v, int u, int wt)
{
    adj[v].push_back(make_pair(u, wt));
    adj[u].push_back(make_pair(v, wt));
}

void dijkstras(vector<p> adj[], int src, int n)
{
    priority_queue<p, vector<p>, greater<p>> pq;
    vector<int> dist(n, INF);
    vector<int> parent(n, -1); // Keep track of parent nodes
    pq.push(make_pair(0, src));
    dist[src] = 0;
    while (!pq.empty())
    {
        int u = pq.top().second;
        pq.pop();
        for (auto x : adj[u])
        {
            int v = x.first;
            int weight = x.second;
            if (dist[v] > dist[u] + weight)
            {
                dist[v] = dist[u] + weight;
                pq.push(make_pair(dist[v], v));
                parent[v] = u; // Update parent of v
            }
        }
    }
    cout << "Vertex Distance from Source\n";
    for (int i = 0; i < n; i++)
        cout << i << "\t\t" << dist[i] << "\n";
```

```
    // Print the shortest path from source to each vertex
    cout << "\nShortest Paths:\n";
    for (int i = 0; i < n; i++)
    {
        cout << "Path to vertex " << i << ": ";
        if (dist[i] == INF)
        {
            cout << "No path";
        }
        else
        {
            vector<int> path;
            int curr = i;
            while (curr != src)
            {
                path.push_back(curr);
                curr = parent[curr];
            }
            path.push_back(src);
            reverse(path.begin(), path.end());

            for (int j = 0; j < path.size(); j++)
            {
                cout << path[j];
                if (j != path.size() - 1)
                    cout << " -> ";
            }
        }
        cout << "\n";
    }
}

int main()
{
    int vertices, edges;
    cout << "Enter Vertices: ";
    cin >> vertices;
    cout << "Enter Edges: ";
```

```
    cin >> edges;
    vector<p> adj[vertices];
    for (int i = 0; i < edges; i++)
    {
        int u, v, wt;
        cin >> u >> v >> wt;
        addEdge(adj, u, v, wt);
    }
    dijkstras(adj, 0, vertices);
    return 0;
}
```

Output:

```
❯ cd "/Users/pargatsinghdhanjal/Deskto
Enter Vertices: 6
Enter Edges: 10
0 1 2
0 2 3
1 2 1
1 4 6
1 3 5
2 3 6
2 4 1
3 4 9
3 5 3
4 5 5
Vertex Distance from Source
0               0
1               2
2               3
3               7
4               4
5               9

Shortest Paths:
Path to vertex 0: 0
Path to vertex 1: 0 -> 1
Path to vertex 2: 0 -> 2
Path to vertex 3: 0 -> 1 -> 3
Path to vertex 4: 0 -> 2 -> 4
Path to vertex 5: 0 -> 2 -> 4 -> 5
```

Time Complexity for single source shortest path

Min-priority queue : **O(V + E log V)**

Conclusion:

Successfully completed the given experiment on Dijkstra's algorithm to findout the shortest path in a weighted graph.