Experiment-3.2

Aim of the Experiment :

Write a program to find the shortest path in a graph using Dijkstra’s Algorithm

1. Problem Description :

We have to find the shortest path in a given graph. We will understand the Dijkstra’s Algorithm for shortest path in a graph.

2. Algorithm :

```
function dijkstra(G, S)
    for each vertex V in G
        distance[V] <- infinite
        previous[V] <- NULL
        If V != S, add V to Priority Queue Q
    distance[S] <- 0
    while Q IS NOT EMPTY
        U <- Extract MIN from Q
        for each unvisited neighbour V of U
            tempDistance <- distance[U] + edge_weight(U, V)
            if tempDistance < distance[V]
                distance[V] <- tempDistance
                previous[V] <- U
    return distance[], previous[]
```

3. Complexity Analysis:

The time complexity of Dijkstra's algorithm depends on the data structure used for the priority queue. Here is a breakdown of the time complexity based on different implementations:

Using an unsorted list as the priority queue: O(V2), where V is the number of vertices in the graph. In each iteration, the algorithm searches for the vertex with the smallest distance among all unvisited vertices, which takes O(V) time. This operation is performed V times, resulting in a time complexity of O(V^2).

Using a sorted list or a binary heap as the priority queue: O(E + V log V), where E is the number of edges in the graph. In each iteration, the algorithm extracts the vertex with the smallest distance from the priority queue, which takes O(log V) time. The distance updates for the neighboring vertices take O(E) time in total. This operation is performed V times, resulting in a time complexity of O(V log V + E log V). Since E can be at most V^2, the time complexity is O(E + V log V).

4. Pseudo Code :

```
Dijkstra(G, s)
  for each vertex v in G
      dist[v] = infinity
      previous[v] = undefined
  dist[s] = 0
  Q = the set of all vertices in G
  while Q is not empty
      u = vertex in Q with smallest dist[]
      remove u from Q
```

```
  for each neighbor v of u

    alt = dist[u] + length(u, v)

    if alt < dist[v]

      dist[v] = alt

      previous[v] = u

 return previous[]
```

5. Source Code for Experiment :

```
#include <iostream>
#include <vector>
#define INT_MAX 10000000
using namespace std;
void DijkstrasTest();

int main() {
   cout<<"Name: Ashish Kumar\n";
   cout<<"UID: 23MAI10008\n\n";

   DijkstrasTest();
   cout<<endl;
   return 0;
}
class Node;
class Edge;
void Dijkstras();
vector<Node*>* AdjacentRemainingNodes(Node* node);
Node* ExtractSmallest(vector<Node*>& nodes);
int Distance(Node* node1, Node* node2);
bool Contains(vector<Node*>& nodes, Node* node);
void PrintShortestRouteTo(Node* destination);
```

```
vector<Node*> nodes;
vector<Edge*> edges;
class Node {
  public:
 Node(char id)
   : id(id), previous(NULL), distanceFromStart(INT_MAX) {
   nodes.push_back(this);
 }

public:
 char id;
 Node* previous;
 int distanceFromStart;
};

class Edge {
  public:
 Edge(Node* node1, Node* node2, int distance)
   : node1(node1), node2(node2), distance(distance) {
   edges.push_back(this);
 }

 bool Connects(Node* node1, Node* node2) {
   return (
     (node1 == this->node1 &&
      node2 == this->node2) ||
     (node1 == this->node2 &&
      node2 == this->node1));
 }
  public:
 Node* node1;
 Node* node2;
 int distance;
};
```

```
void DijkstrasTest() {
  Node* a = new Node('a');
  Node* b = new Node('b');
  Node* c = new Node('c');
  Node* d = new Node('d');
  Node* e = new Node('e');
  Node* f = new Node('f');
  Node* g = new Node('g');

  Edge* e1 = new Edge(a, c, 1);
  Edge* e2 = new Edge(a, d, 2);
  Edge* e3 = new Edge(b, c, 2);
  Edge* e4 = new Edge(c, d, 1);
  Edge* e5 = new Edge(b, f, 3);
  Edge* e6 = new Edge(c, e, 3);
  Edge* e7 = new Edge(e, f, 2);
  Edge* e8 = new Edge(d, g, 1);
  Edge* e9 = new Edge(g, f, 1);
  a->distanceFromStart = 0;  // set start node
  Dijkstras();
  PrintShortestRouteTo(f);
}
void Dijkstras() {

while (nodes.size() > 0) {
    Node* smallest = ExtractSmallest(nodes);
    vector<Node*>* adjacentNodes =
      AdjacentRemainingNodes(smallest);
    const int size = adjacentNodes->size();
    for (int i = 0; i < size; ++i) {
      Node* adjacent = adjacentNodes->at(i);
      int distance = Distance(smallest, adjacent) +
            smallest->distanceFromStart;
      if (distance < adjacent->distanceFromStart) {
        adjacent->distanceFromStart = distance;
```

```
        adjacent->previous = smallest;
      }
    }
    delete adjacentNodes;
  }
}

// Find the node with the smallest distance,
// remove it, and return it.
Node* ExtractSmallest(vector<Node*>& nodes) {
  int size = nodes.size();
  if (size == 0) return NULL;
  int smallestPosition = 0;
  Node* smallest = nodes.at(0);
  for (int i = 1; i < size; ++i) {
    Node* current = nodes.at(i);
    if (current->distanceFromStart <
      smallest->distanceFromStart) {
      smallest = current;
      smallestPosition = i;
    }
  }
  nodes.erase(nodes.begin() + smallestPosition);
  return smallest;
}
// Return all nodes adjacent to 'node' which are still
// in the 'nodes' collection.
vector<Node*>* AdjacentRemainingNodes(Node* node) {
  vector<Node*>* adjacentNodes = new vector<Node*>();
  const int size = edges.size();
  for (int i = 0; i < size; ++i) {
    Edge* edge = edges.at(i);
    Node* adjacent = NULL;
    if (edge->node1 == node) {
      adjacent = edge->node2;
    } else if (edge->node2 == node) {
```

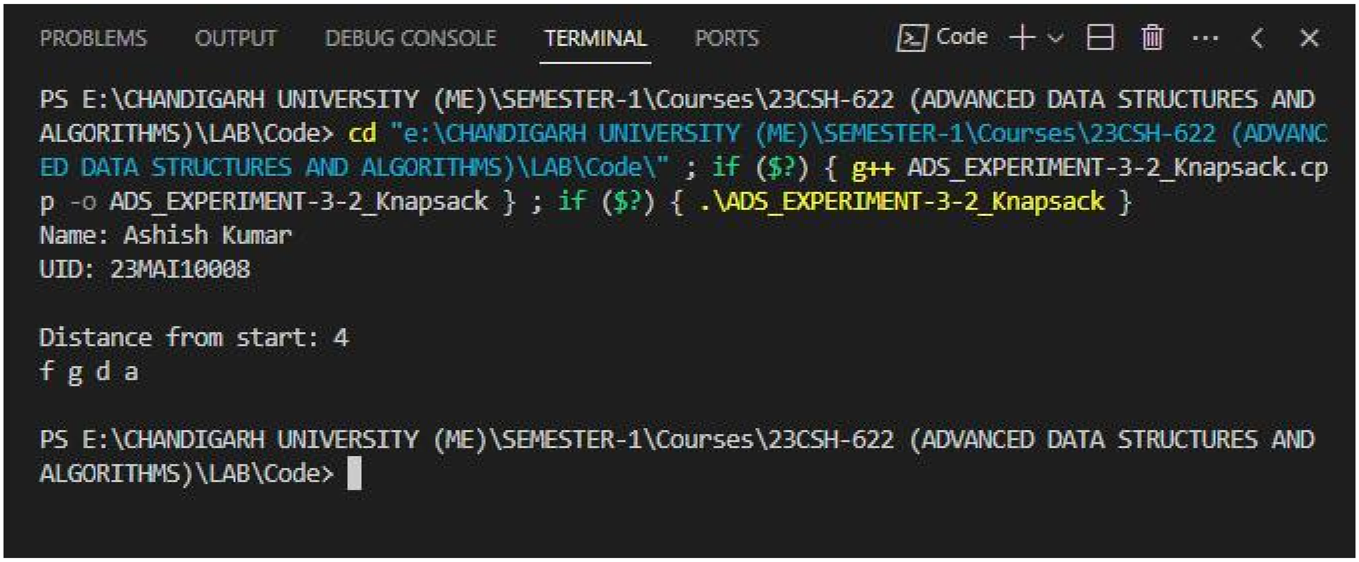
```
      adjacent = edge->node1;
    }
    if (adjacent && Contains(nodes, adjacent)) {
      adjacentNodes->push_back(adjacent);
    }
  }
  return adjacentNodes;
}
// Return distance between two connected nodes
int Distance(Node* node1, Node* node2) {
  const int size = edges.size();
  for (int i = 0; i < size; ++i) {
    Edge* edge = edges.at(i);
    if (edge->Connects(node1, node2)) {
      return edge->distance;
    }
  }
  return -1;  // should never happen
}
// Does the 'nodes' vector contain 'node'
bool Contains(vector<Node*>& nodes, Node* node) {
  const int size = nodes.size();
  for (int i = 0; i < size; ++i) {
    if (node == nodes.at(i)) {
      return true;
    }
  }
  return false;
}

void PrintShortestRouteTo(Node* destination) {
  Node* previous = destination;
  cout << "Distance from start: "
     << destination->distanceFromStart << endl;
  while (previous) {
    cout << previous->id << " ";
```

```
    previous = previous->previous;
  }
  cout << endl;
}
// these two not needed
vector<Edge*>* AdjacentEdges(vector<Edge*>& Edges, Node* node);
void RemoveEdge(vector<Edge*>& Edges, Edge* edge);
vector<Edge*>* AdjacentEdges(vector<Edge*>& edges, Node* node) {
  vector<Edge*>* adjacentEdges = new vector<Edge*>();
  const int size = edges.size();
  for (int i = 0; i < size; ++i) {
    Edge* edge = edges.at(i);
    if (edge->node1 == node) {
      cout << "adjacent: " << edge->node2->id << endl;
      adjacentEdges->push_back(edge);
    } else if (edge->node2 == node) {
      cout << "adjacent: " << edge->node1->id << endl;
      adjacentEdges->push_back(edge);
    }
  }
  return adjacentEdges;
}

void RemoveEdge(vector<Edge*>& edges, Edge* edge) {
  vector<Edge*>::iterator it;
  for (it = edges.begin(); it < edges.end(); ++it) {
    if (*it == edge) {
      edges.erase(it);
      return;
    }
  }
}
```

6. Result/Output :



```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS                    Code

PS E:\CHANDIGARH UNIVERSITY (ME)\SEMESTER-1\Courses\23CSH-622 (ADVANCED DATA STRUCTURES AND
ALGORITHMS)\LAB\Code> cd "e:\CHANDIGARH UNIVERSITY (ME)\SEMESTER-1\Courses\23CSH-622 (ADVANC
ED DATA STRUCTURES AND ALGORITHMS)\LAB\Code\" ; if ($?) { g++ ADS_EXPERIMENT-3-2_Knapsack.cp
p -o ADS_EXPERIMENT-3-2_Knapsack } ; if ($?) { .\ADS_EXPERIMENT-3-2_Knapsack }
Name: Ashish Kumar
UID: 23MAI10008

Distance from start: 4
f g d a

PS E:\CHANDIGARH UNIVERSITY (ME)\SEMESTER-1\Courses\23CSH-622 (ADVANCED DATA STRUCTURES AND
ALGORITHMS)\LAB\Code>
```

Learning outcomes (What I have learnt):

1. I learnt about how to input elements in an array.

2. I learnt about how to approach the shortest path problem.

3. I learnt about the Dijkstra's algorithm and its applications.

4. I learnt about the differences between Kruskal's and Prim's Algorithm.

5. I learnt about time complexity of Dijkstra's Algorithm.