CSCE-4133
Algorithms

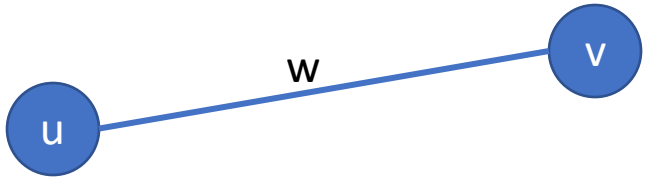
Programming Review
Graph Implementation

Fall 2024

Prof. Khoa Luu

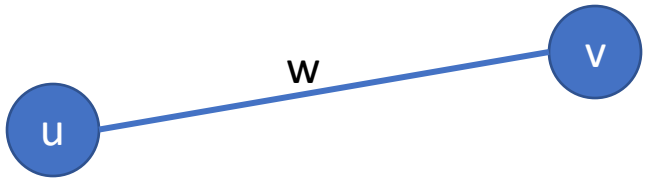
Prepared by Thanh-Dat Truong

Edge Structure



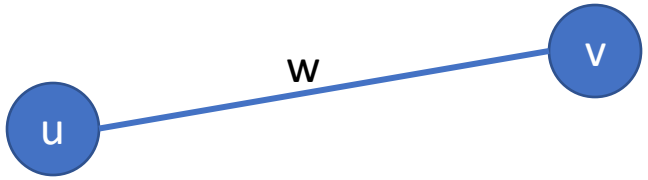
Edge Structure

```
struct Edge {
  int u; int v; int w;
  Edge();
  Edge(int u, int v, int w);
};
```



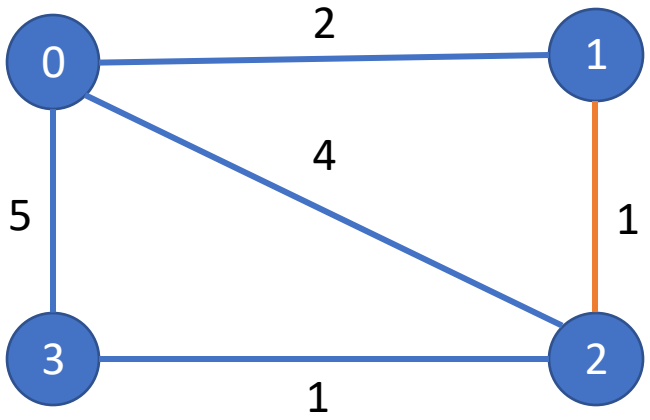
Edge Structure

```
struct Edge {
  int u; int v; int w;
  Edge();
  Edge(int u, int v, int w);
};
```



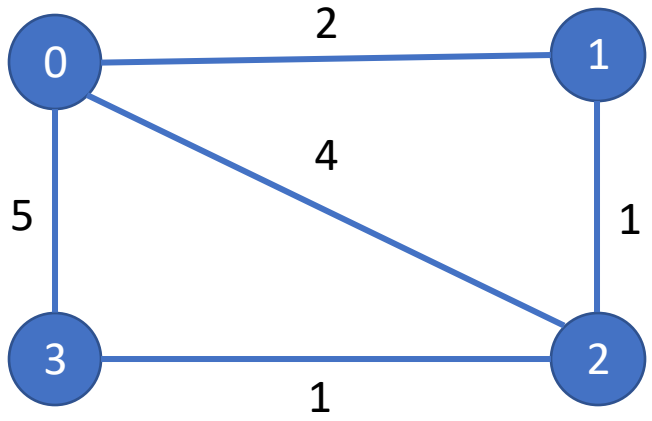

```
Edge e(1, 5, 10);
std::cout << "u = " << e.u << ". v = " << e.v << ". w = " << e.w;
```

Graph Structure



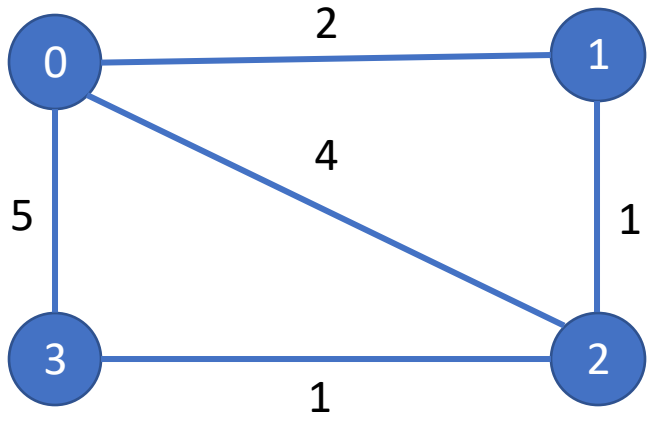
Graph Structure

```
class Graph {
  public:
    int n; // Number of vertices
    std::vector<std::vector<Edge> > e; // Adjacent list
  public:
    Graph(int n);
    ~Graph();
    void insertEdge(int u, int v, int w, bool directed = false);
}
```



Graph Structure

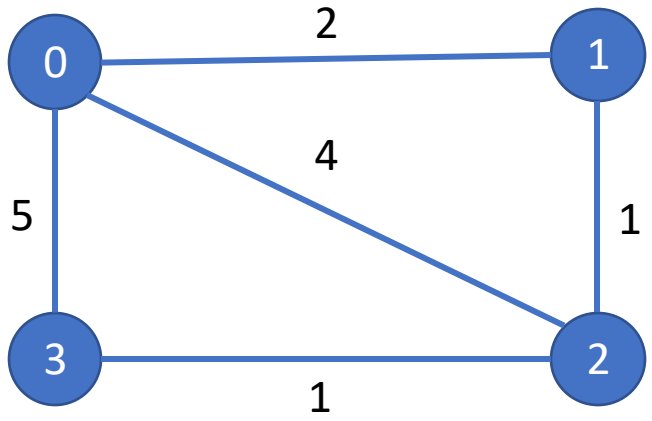
```
class Graph {
  public:
    int n; // Number of vertices
    std::vector<std::vector<Edge> > e; // Adjacent list
  public:
    Graph(int n);
    ~Graph();
    void insertEdge(int u, int v, int w, bool directed = false);
}
```




```
Graph G(4)
G.insertEdge(0, 1, 2); G.insertEdge(0, 2, 4); G.insertEdge(0, 3, 5);
G.insertEdge(1, 2, 1);
G.insertEdge(2, 3, 1);
```

Graph Structure

```
class Graph {
  public:
    int n; // Number of vertices
    std::vector<std::vector<Edge> > e; // Adjacent list
  public:
    Graph(int n);
    ~Graph();
    void insertEdge(int u, int v, int w, bool directed = false);
}
```




```
int u = 0;
for (int i = 0; i < G.e[u].size(); ++i) {
  int v = G.e[u][i].v;
  int w = G.e[u][i].w;
  std::cout << "u = " << u << ". v = " << v << ". w = " << w;
}
```

Prim's Algorithm

```
std::vector<Edge> constructMSTPrim(Graph G) {

}
```

Prim's Algorithm

```
std::vector<Edge> constructMSTPrim(Graph G) {
  std::vector<Edge> MST; std::vector<bool> T(G.n, false);
  std::vector<int> distance(G.n, INT_MAX), parent(G.n, -1);
```

```
}
```

Prim's Algorithm

```
std::vector<Edge> constructMSTPrim(Graph G) {
 std::vector<Edge> MST; std::vector<bool> T(G.n, false);
 std::vector<int> distance(G.n, INT_MAX), parent(G.n, -1);
 T[0] = true;




}
```

Prim's Algorithm

```
std::vector<Edge> constructMSTPrim(Graph G) {
 std::vector<Edge> MST; std::vector<bool> T(G.n, false);
 std::vector<int> distance(G.n, INT_MAX), parent(G.n, -1);
 T[0] = true;
 for (auto e: G.e[0]) {
  int v = e.v;  int w = e.w;
  distance[v] = w ; parent[v] = 0;
 }









}
```

Prim's Algorithm

```
std::vector<Edge> constructMSTPrim(Graph G) {
 std::vector<Edge> MST; std::vector<bool> T(G.n, false);
 std::vector<int> distance(G.n, INT_MAX), parent(G.n, -1);
 T[0] = true;
 for (auto e: G.e[0]) {
  int v = e.v;  int w = e.w;
  distance[v] = w ; parent[v] = 0;
 }
 for (int i = 1; i <= G.n - 1; ++i) {









  }

}
```

Prim's Algorithm

```
std::vector<Edge> constructMSTPrim(Graph G) {
 std::vector<Edge> MST; std::vector<bool> T(G.n, false);
 std::vector<int> distance(G.n, INT_MAX), parent(G.n, -1);
 T[0] = true;
 for (auto e: G.e[0]) {
  int v = e.v;  int w = e.w;
  distance[v] = w ; parent[v] = 0;
 }
 for (int i = 1; i <= G.n - 1; ++i) {
  int u, minDistance = INT_MAX;









  }

}
```

Prim's Algorithm

```
std::vector<Edge> constructMSTPrim(Graph G) {
 std::vector<Edge> MST; std::vector<bool> T(G.n, false);
 std::vector<int> distance(G.n, INT_MAX), parent(G.n, -1);
 T[0] = true;
 for (auto e: G.e[0]) {
  int v = e.v;  int w = e.w;
  distance[v] = w ; parent[v] = 0;
 }
 for (int i = 1; i <= G.n - 1; ++i) {
  int u, minDistance = INT_MAX;
  for (int i = 0; i < G.n; ++i)
   if (distance[i] < minDistance && T[i] == false)
    minDistance = distance[i], u = i;









 }

}
```

Prim's Algorithm

```
std::vector<Edge> constructMSTPrim(Graph G) {
 std::vector<Edge> MST; std::vector<bool> T(G.n, false);
 std::vector<int> distance(G.n, INT_MAX), parent(G.n, -1);
 T[0] = true;
 for (auto e: G.e[0]) {
  int v = e.v;  int w = e.w;
  distance[v] = w ; parent[v] = 0;
 }
 for (int i = 1; i <= G.n - 1; ++i) {
  int u, minDistance = INT_MAX;
  for (int i = 0; i < G.n; ++i)
   if (distance[i] < minDistance && T[i] == false)
    minDistance = distance[i], u = i;
  T[u] = true;









 }

}
```

Prim's Algorithm

```
std::vector<Edge> constructMSTPrim(Graph G) {
 std::vector<Edge> MST; std::vector<bool> T(G.n, false);
 std::vector<int> distance(G.n, INT_MAX), parent(G.n, -1);
 T[0] = true;
 for (auto e: G.e[0]) {
  int v = e.v;  int w = e.w;
  distance[v] = w ; parent[v] = 0;
 }
 for (int i = 1; i <= G.n - 1; ++i) {
  int u, minDistance = INT_MAX;
  for (int i = 0; i < G.n; ++i)
   if (distance[i] < minDistance && T[i] == false)
    minDistance = distance[i], u = i;
  T[u] = true;
  MST.push_back(Edge(u, parent[u], distance[u]));









 }

}
```

Prim's Algorithm

```
std::vector<Edge> constructMSTPrim(Graph G) {
 std::vector<Edge> MST; std::vector<bool> T(G.n, false);
 std::vector<int> distance(G.n, INT_MAX), parent(G.n, -1);
 T[0] = true;
 for (auto e: G.e[0]) {
  int v = e.v;  int w = e.w;
  distance[v] = w ; parent[v] = 0;
 }
 for (int i = 1; i <= G.n - 1; ++i) {
  int u, minDistance = INT_MAX;
  for (int i = 0; i < G.n; ++i)
   if (distance[i] < minDistance && T[i] == false)
    minDistance = distance[i], u = i;
  T[u] = true;
  MST.push_back(Edge(u, parent[u], distance[u]));
  for (auto e: G.e[u]) {



  }
 }

}
```

Prim's Algorithm

```
std::vector<Edge> constructMSTPrim(Graph G) {
 std::vector<Edge> MST; std::vector<bool> T(G.n, false);
 std::vector<int> distance(G.n, INT_MAX), parent(G.n, -1);
 T[0] = true;
 for (auto e: G.e[0]) {
  int v = e.v;  int w = e.w;
  distance[v] = w ; parent[v] = 0;
 }
 for (int i = 1; i <= G.n - 1; ++i) {
  int u, minDistance = INT_MAX;
  for (int i = 0; i < G.n; ++i)
   if (distance[i] < minDistance && T[i] == false)
    minDistance = distance[i], u = i;
  T[u] = true;
  MST.push_back(Edge(u, parent[u], distance[u]));
  for (auto e: G.e[u]) {
   int v = e.v;  int w = e.w;
   if (T[v] == false && w < distance[v])
    distance[v] = w, parent[v] = u;
  }
 }

}
```

Prim's Algorithm

```
std::vector<Edge> constructMSTPrim(Graph G) {
 std::vector<Edge> MST; std::vector<bool> T(G.n, false);
 std::vector<int> distance(G.n, INT_MAX), parent(G.n, -1);
 T[0] = true;
 for (auto e: G.e[0]) {
  int v = e.v;  int w = e.w;
  distance[v] = w ; parent[v] = 0;
 }
 for (int i = 1; i <= G.n - 1; ++i) {
  int u, minDistance = INT_MAX;
  for (int i = 0; i < G.n; ++i)
   if (distance[i] < minDistance && T[i] == false)
    minDistance = distance[i], u = i;
  T[u] = true;
  MST.push_back(Edge(u, parent[u], distance[u]));
  for (auto e: G.e[u]) {
   int v = e.v;  int w = e.w;
   if (T[v] == false && w < distance[v])
    distance[v] = w, parent[v] = u;
  }
 }
 return MST;
}
```

Kruskal's Algorithm

```
std::vector<Edge> constructMSTKruskal(Graph G) {

}
```

Kruskal's Algorithm

```
std::vector<Edge> constructMSTKruskal(Graph G) {
  std::vector<Edge> edges = G.exportEdges();




}
```

Kruskal's Algorithm

```
std::vector<Edge> constructMSTKruskal(Graph G) {
  std::vector<Edge> edges = G.exportEdges();
  std::vector<Edge> MST;




}
```

Kruskal's Algorithm

```
std::vector<Edge> constructMSTKruskal(Graph G) {
  std::vector<Edge> edges = G.exportEdges();
  std::vector<Edge> MST;
  DisjointSet T(G.n);




}
```

Kruskal's Algorithm

```
std::vector<Edge> constructMSTKruskal(Graph G) {
  std::vector<Edge> edges = G.exportEdges();
  std::vector<Edge> MST;
  DisjointSet T(G.n);

  int size_of_edges = edge.size()
  sort(edges, 0, size_of_edges - 1);
```

```
}
```

Kruskal's Algorithm

```
std::vector<Edge> constructMSTKruskal(Graph G) {
  std::vector<Edge> edges = G.exportEdges();
  std::vector<Edge> MST;
  DisjointSet T(G.n);

  int size_of_edges = edge.size()
  sort(edges, 0, size_of_edges - 1);

  for (auto e: edges) {









  }

}
```

Kruskal's Algorithm

```
std::vector<Edge> constructMSTKruskal(Graph G) {
  std::vector<Edge> edges = G.exportEdges();
  std::vector<Edge> MST;
  DisjointSet T(G.n);

  int size_of_edges = edge.size()
  sort(edges, 0, size_of_edges - 1);

  for (auto e: edges) {
    int u = e.u;
    int v = e.v;
    int w = e.w;




  }

}
```

Kruskal's Algorithm

```
std::vector<Edge> constructMSTKruskal(Graph G) {
  std::vector<Edge> edges = G.exportEdges();
  std::vector<Edge> MST;
  DisjointSet T(G.n);

  int size_of_edges = edge.size()
  sort(edges, 0, size_of_edges - 1);

  for (auto e: edges) {
    int u = e.u;
    int v = e.v;
    int w = e.w;
    if (T.isOnSameSet(u, v) == false) {


    }
  }

}
```

Kruskal's Algorithm

```
std::vector<Edge> constructMSTKruskal(Graph G) {
  std::vector<Edge> edges = G.exportEdges();
  std::vector<Edge> MST;
  DisjointSet T(G.n);

  int size_of_edges = edge.size()
  sort(edges, 0, size_of_edges - 1);

  for (auto e: edges) {
    int u = e.u;
    int v = e.v;
    int w = e.w;
    if (T.isOnSameSet(u, v) == false) {
      T.join(u, v);

    }
  }

}
```

Kruskal's Algorithm

```
std::vector<Edge> constructMSTKruskal(Graph G) {
  std::vector<Edge> edges = G.exportEdges();
  std::vector<Edge> MST;
  DisjointSet T(G.n);

  int size_of_edges = edge.size()
  sort(edges, 0, size_of_edges - 1);

  for (auto e: edges) {
    int u = e.u;
    int v = e.v;
    int w = e.w;
    if (T.isOnSameSet(u, v) == false) {
      T.join(u, v);
      MST.push_back(Edge(u, v, w));
    }
  }

}
```

Kruskal's Algorithm

```
std::vector<Edge> constructMSTKruskal(Graph G) {
  std::vector<Edge> edges = G.exportEdges();
  std::vector<Edge> MST;
  DisjointSet T(G.n);

  int size_of_edges = edge.size()
  sort(edges, 0, size_of_edges - 1);

  for (auto e: edges) {
    int u = e.u;
    int v = e.v;
    int w = e.w;
    if (T.isOnSameSet(u, v) == false) {
      T.join(u, v);
      MST.push_back(Edge(u, v, w));
    }
  }
  return MST;
}
```

Dijkstra's Algorithm

```
std::vector<int> searchShortestPath(Graph &G, int start, int destination) {

}
```

Dijkstra's Algorithm

```
std::vector<int> searchShortestPath(Graph &G, int start, int destination) {
  std::vector<bool> visited(G.n, false);
  std::vector<int> distance(G.n, INT_MAX), parent(G.n, -1);









}
```

Dijkstra's Algorithm

```
std::vector<int> searchShortestPath(Graph &G, int start, int destination) {
  std::vector<bool> visited(G.n, false);
  std::vector<int> distance(G.n, INT_MAX), parent(G.n, -1);
  distance[start] = 0;   parent[start] = -1;









}
```

Dijkstra's Algorithm

```
std::vector<int> searchShortestPath(Graph &G, int start, int destination) {
  std::vector<bool> visited(G.n, false);
  std::vector<int> distance(G.n, INT_MAX), parent(G.n, -1);
  distance[start] = 0;   parent[start] = -1;
  for (int i = 1; i <= G.n - 1; ++i) {









  }




}
```

Dijkstra's Algorithm

```
std::vector<int> searchShortestPath(Graph &G, int start, int destination) {
 std::vector<bool> visited(G.n, false);
 std::vector<int> distance(G.n, INT_MAX), parent(G.n, -1);
 distance[start] = 0;   parent[start] = -1;
 for (int i = 1; i <= G.n - 1; ++i) {
  int u, minDistance = INT_MAX;
  for (int i = 0; i < G.n; ++i)
    if (distance[i] < minDistance && visited[i] == false)
      minDistance = distance[i], u = i;









 }

}
```

Dijkstra's Algorithm

```
std::vector<int> searchShortestPath(Graph &G, int start, int destination) {
 std::vector<bool> visited(G.n, false);
 std::vector<int> distance(G.n, INT_MAX), parent(G.n, -1);
 distance[start] = 0;   parent[start] = -1;
 for (int i = 1; i <= G.n - 1; ++i) {
  int u, minDistance = INT_MAX;
  for (int i = 0; i < G.n; ++i)
    if (distance[i] < minDistance && visited[i] == false)
      minDistance = distance[i], u = i;
  visited[u] = true;









 }









}
```

Dijkstra's Algorithm

```
std::vector<int> searchShortestPath(Graph &G, int start, int destination) {
 std::vector<bool> visited(G.n, false);
 std::vector<int> distance(G.n, INT_MAX), parent(G.n, -1);
 distance[start] = 0;   parent[start] = -1;
 for (int i = 1; i <= G.n - 1; ++i) {
  int u, minDistance = INT_MAX;
  for (int i = 0; i < G.n; ++i)
    if (distance[i] < minDistance && visited[i] == false)
      minDistance = distance[i], u = i;
  visited[u] = true;
  for (auto e: G.e[u]) {




  }
 }


}
```

Dijkstra's Algorithm

```
std::vector<int> searchShortestPath(Graph &G, int start, int destination) {
 std::vector<bool> visited(G.n, false);
 std::vector<int> distance(G.n, INT_MAX), parent(G.n, -1);
 distance[start] = 0;   parent[start] = -1;
 for (int i = 1; i <= G.n - 1; ++i) {
  int u, minDistance = INT_MAX;
  for (int i = 0; i < G.n; ++i)
    if (distance[i] < minDistance && visited[i] == false)
      minDistance = distance[i], u = i;
  visited[u] = true;
  for (auto e: G.e[u]) {
   int v = e.v; int w = e.w;
   if (visited[v] == false && distance[u] + w < distance[v])
     distance[v] = distance[u] + w , parent[v] = u;
  }
 }
}
```

Dijkstra's Algorithm

```
std::vector<int> searchShortestPath(Graph &G, int start, int destination) {
 std::vector<bool> visited(G.n, false);
 std::vector<int> distance(G.n, INT_MAX), parent(G.n, -1);
 distance[start] = 0;   parent[start] = -1;
 for (int i = 1; i <= G.n - 1; ++i) {
  int u, minDistance = INT_MAX;
  for (int i = 0; i < G.n; ++i)
    if (distance[i] < minDistance && visited[i] == false)
      minDistance = distance[i], u = i;
  visited[u] = true;
  for (auto e: G.e[u]) {
   int v = e.v; int w = e.w;
   if (visited[v] == false && distance[u] + w < distance[v])
     distance[v] = distance[u] + w , parent[v] = u;
  }
 }
 std::vector<int> path;  int u = destination;




}
```

Dijkstra's Algorithm

```
std::vector<int> searchShortestPath(Graph &G, int start, int destination) {
 std::vector<bool> visited(G.n, false);
 std::vector<int> distance(G.n, INT_MAX), parent(G.n, -1);
 distance[start] = 0;   parent[start] = -1;
 for (int i = 1; i <= G.n - 1; ++i) {
  int u, minDistance = INT_MAX;
  for (int i = 0; i < G.n; ++i)
    if (distance[i] < minDistance && visited[i] == false)
      minDistance = distance[i], u = i;
  visited[u] = true;
  for (auto e: G.e[u]) {
   int v = e.v; int w = e.w;
   if (visited[v] == false && distance[u] + w < distance[v])
    distance[v] = distance[u] + w , parent[v] = u;
 }
}
 std::vector<int> path;  int u = destination;
 while (u != -1) {

 }


}
```

Dijkstra's Algorithm

```
std::vector<int> searchShortestPath(Graph &G, int start, int destination) {
 std::vector<bool> visited(G.n, false);
 std::vector<int> distance(G.n, INT_MAX), parent(G.n, -1);
 distance[start] = 0;   parent[start] = -1;
 for (int i = 1; i <= G.n - 1; ++i) {
  int u, minDistance = INT_MAX;
  for (int i = 0; i < G.n; ++i)
    if (distance[i] < minDistance && visited[i] == false)
      minDistance = distance[i], u = i;
  visited[u] = true;
  for (auto e: G.e[u]) {
   int v = e.v; int w = e.w;
   if (visited[v] == false && distance[u] + w < distance[v])
    distance[v] = distance[u] + w , parent[v] = u;
 }
}
 std::vector<int> path;  int u = destination;
 while (u != -1) {
  path.push_back(u); u = parent[u]
 }


}
```

Dijkstra's Algorithm

```
std::vector<int> searchShortestPath(Graph &G, int start, int destination) {
 std::vector<bool> visited(G.n, false);
 std::vector<int> distance(G.n, INT_MAX), parent(G.n, -1);
 distance[start] = 0;   parent[start] = -1;
 for (int i = 1; i <= G.n - 1; ++i) {
  int u, minDistance = INT_MAX;
  for (int i = 0; i < G.n; ++i)
    if (distance[i] < minDistance && visited[i] == false)
      minDistance = distance[i], u = i;
  visited[u] = true;
  for (auto e: G.e[u]) {
   int v = e.v; int w = e.w;
   if (visited[v] == false && distance[u] + w < distance[v])
    distance[v] = distance[u] + w , parent[v] = u;
 }
}
 std::vector<int> path;  int u = destination;
 while (u != -1) {
  path.push_back(u); u = parent[u]
 }
 std::reverse(path.begin(), path.end());

}
```

Dijkstra's Algorithm

```
std::vector<int> searchShortestPath(Graph &G, int start, int destination) {
 std::vector<bool> visited(G.n, false);
 std::vector<int> distance(G.n, INT_MAX), parent(G.n, -1);
 distance[start] = 0;   parent[start] = -1;
 for (int i = 1; i <= G.n - 1; ++i) {
  int u, minDistance = INT_MAX;
  for (int i = 0; i < G.n; ++i)
    if (distance[i] < minDistance && visited[i] == false)
      minDistance = distance[i], u = i;
  visited[u] = true;
  for (auto e: G.e[u]) {
   int v = e.v; int w = e.w;
   if (visited[v] == false && distance[u] + w < distance[v])
    distance[v] = distance[u] + w , parent[v] = u;
  }
 }
 std::vector<int> path;  int u = destination;
 while (u != -1) {
  path.push_back(u); u = parent[u]
 }
 std::reverse(path.begin(), path.end());
 return path;
}
```

Demo