CSCE-4133
Algorithms

Homework 4
Minimum Spanning Tree
and
Shortest Path

Fall 2024

Prof. Khoa Luu

Prepared by Thanh-Dat Truong

Compilation

- In this homework, ***you use only a single Makefile***.
  - It will automatically detect your Operator System (e.g., Linux, MacOS, or Windows) and configurate the compilation rules.
  - For Windows users, you have to make sure that you set the correct paths for MINGW_BIN and OPENCV_DIR:
  - **MINGW_BIN=<path to mingw64>/bin**
  - **OPENCV_DIR=<path to opencv>/build/install**

Compilation

- **OpenCV is OPTIONAL, you can work on homework WITHOUT installing and using OpenCV**
  - Open file ***Makefile*** and edit line ***OPENCV=1*** to ***OPENCV=0.***

Compiling

- Compile source code with/without OpenCV
  - Open Makefile, change the line OPENCV=1 or OPENCV=0
  - On terminal, go to the homework folder (a folder contains Makefile)
  - Type:
    - On Linux/MacOS: **make prim** or **make kruskal** or **make dijkstra**
    - On Windows: **mingw32-make prim** or **mingw32-make kruskal** or **mingw32-make dijkstra**

Important Notes

- The pseudo codes provided in the lecture sides are NOT the actual solution.
- The pseudo codes are the ideas to complete the homework.

Prim’s Algorithm

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









}
```

Prim's Algorithm

```
std::vector<Edge> constructMSTPrim(Graph G) {
  std::vector<Edge> edges = G.exportEdges();
  MST ← { } and T ← { 0 }
















}
```

Prim's Algorithm

```
std::vector<Edge> constructMSTPrim(Graph G) {
  std::vector<Edge> edges = G.exportEdges();
  MST ← { } and T ← { 0 }
  for each vertex v in V do # V is a set of vertices of G
    if G has an edge (0, v) then
      distance[v] ← w(0, v) and parent[v] ← 0
    else
      distance[v] ← ∞ and parent[v] ← -1
   end if
  end do











}
```

Prim’s Algorithm

```
std::vector<Edge> constructMSTPrim(Graph G) {
  std::vector<Edge> edges = G.exportEdges();
  MST ← { } and T ← { 0 }
  for each vertex v in V do # V is a set of vertices of G
    if G has an edge (0, v) then
      distance[v] ← w(0, v) and parent[v] ← 0
    else
      distance[v] ← ∞ and parent[v] ← -1
   end if
  end do
  for i from 1 to |V| - 1 do







  end do

}
```

Prim's Algorithm

```
std::vector<Edge> constructMSTPrim(Graph G) {
  std::vector<Edge> edges = G.exportEdges();
  MST ← { } and T ← { 0 }
  for each vertex v in V do # V is a set of vertices of G
    if G has an edge (0, v) then
      distance[v] ← w(0, v) and parent[v] ← 0
    else
      distance[v] ← ∞ and parent[v] ← -1
   end if
  end do
  for i from 1 to |V| - 1 do
    u ← argmin_{u ∈ V and u ∉ T} distance[u]
    MST ← MST ∪ {(parent[u], u, w(u, parent[u])}
    T ← T ∪ {u}




  end do

}
```

Prim’s Algorithm

```
std::vector<Edge> constructMSTPrim(Graph G) {
  std::vector<Edge> edges = G.exportEdges();
  MST ← { } and T ← { 0 }
  for each vertex v in V do # V is a set of vertices of G
    if G has an edge (0, v) then
      distance[v] ← w(0, v) and parent[v] ← 0
    else
      distance[v] ← ∞ and parent[v] ← -1
   end if
  end do
  for i from 1 to |V| - 1 do
    u ← argmin_{u ∈ V and u ∉ T} distance[u]
    MST ← MST ∪ {(parent[u], u, w(u, parent[u])}
    T ← T ∪ {u}
    for each vertex v in V do
      if v ∉ T and G has an edge (u, v) and w(u, v) < distance[v] then
        distance[v] ← w(u, v) and parent[v] ← u
      end if
    end do
  end do
  return MST
}
```

Kruskal’s Algorithm

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











}
```

Kruskal’s Algorithm

```
std::vector<Edge> constructMSTKruskal(Graph G) {
  std::vector<Edge> edges = G.exportEdges(); // Graph's edges
  Sort list of edges in the increasing order of edge’s weight









}
```

Kruskal’s Algorithm

```
std::vector<Edge> constructMSTKruskal(Graph G) {
  std::vector<Edge> edges = G.exportEdges(); // Graph's edges
  Sort list of edges in the increasing order of edge’s weight
  MST ← { }
  T ← { }







}
```

Kruskal’s Algorithm

```
std::vector<Edge> constructMSTKruskal(Graph G) {
  std::vector<Edge> edges = G.exportEdges(); // Graph's edges
  Sort list of edges in the increasing order of edge’s weight
  MST ← { }
  T ← { }
  for each edge e in sorted list of edges do




  end do

}
```

Kruskal’s Algorithm

```
std::vector<Edge> constructMSTKruskal(Graph G) {
  std::vector<Edge> edges = G.exportEdges(); // Graph's edges
  Sort list of edges in the increasing order of edge’s weight
  MST ← { }
  T ← { }
  for each edge e in sorted list of edges do
    if u and v of e is not connected on T then
      MST ← MST ∪ {(u, v, w(u, v)}
      T ← T ∪ {u, v}
    end if
  end do
  return MST
}
```

Kruskal’s Algorithm

```
class DisjointSet {
  private:
    std::vector<int> parent;
    int find(int u);
  public:
    DisjointSet(int n);
    int isOnSameSet(int u, int v);
    void join(int u, int v);
};
```

Kruskal’s Algorithm

```
DisjointSet::DisjointSet(int n) {
  this->parent = std::vector<int>(n, -1);
}

int DisjointSet::find(int u) {
  if (this->parent[u] == -1)
    return u;
  else
    return this->parent[u] = this->find(parent[u]);
}
```

Kruskal’s Algorithm

```
int DisjointSet::isOnSameSet(int u, int v) {
  return (this->find(u) == this->find(v)) ? 1 : 0;
}

void DisjointSet::join(int u, int v) {
  int pu = this->find(u);
  int pv = this->find(v);
  if (pu != pv)
    this->parent[pu] = pv;
}
```

Dijkstra Algorithm’s

```
std::vector<int> searchShortestPath(Graph &G, int start, int passBy, int destination) {
  start_to_middle = searchSinglePath(G, start, passBy)
  middle_to_destination = searchSinglePath(G, passBy, destination)
  return merge start_to_middle  and middle_to_destination
}
```

Dijkstra Algorithm’s

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





}
```

Dijkstra Algorithm’s

```
std::vector<int> searchSinglePath(Graph &G, int start, int destination) {
  distance[start] ← 0
  parent[start] ← -1









}
```

Dijkstra Algorithm’s

```
std::vector<int> searchSinglePath(Graph &G, int start, int destination) {
  distance[start] ← 0
  parent[start] ← -1
  for i from 1 to |V| do



  end do



}
```

Dijkstra Algorithm's

```
std::vector<int> searchSinglePath(Graph &G, int start, int destination) {
  distance[start] ← 0
  parent[start] ← -1
  for i from 1 to |V| do
    u ← argmin_{u ∈ V and u is not visited} distance[u]
    visited[u] ← True
    Break if u is a destination




  end do



}
```

Dijkstra Algorithm's

```
std::vector<int> searchSinglePath(Graph &G, int start, int destination) {
  distance[start] ← 0
  parent[start] ← -1
  for i from 1 to |V| do
    u ← argmin_{u ∈ V and u is not visited} distance[u]
    visited[u] ← True
    Break if u is a destination
    for each vertex v in the adjacency list of u do
      if distance[u] + w(u, v) < distance[v] then
        distance[v] ← distance[u] + w(u, v) and parent[v] ← u
      end if
    end do
  end do



}
```

Dijkstra Algorithm’s

```
std::vector<int> searchSinglePath(Graph &G, int start, int destination) {
  distance[start] ← 0
  parent[start] ← -1
  for i from 1 to |V| do
    u ← argmin_{u ∈ V and u is not visited} distance[u]
    visited[u] ← True
    Break if u is a destination
    for each vertex v in the adjacency list of u do
      if distance[u] + w(u, v) < distance[v] then
        distance[v] ← distance[u] + w(u, v) and parent[v] ← u
      end if
    end do
  end do
  Path ← { }
  u ← destination
  while u ≠ -1 do


  end do

}
```

Dijkstra Algorithm's

```
std::vector<int> searchSinglePath(Graph &G, int start, int destination) {
  distance[start] ← 0
  parent[start] ← -1
  for i from 1 to |V| do
    u ← argmin_{u ∈ V and u is not visited} distance[u]
    visited[u] ← True
    Break if u is a destination
    for each vertex v in the adjacency list of u do
      if distance[u] + w(u, v) < distance[v] then
        distance[v] ← distance[u] + w(u, v) and parent[v] ← u
      end if
    end do
  end do
  Path ← { }
  u ← destination
  while u ≠ -1 do
    Path ← { u } ∪ Path
    u ← parent[u]
  end do
  return Path
}
```

Optimize By Using Heap

```
struct EdgeKeyComparison {
  constexpr bool operator()(const Edge &a, const Edge &b) const noexcept {
    return a.w > b.w;
  }
};
```

```
std::priority_queue< Edge, std::vector<Edge>, EdgeKeyComparison > heap;
```

If you want to use heap to optimize the minimum searching, you can use heap defined as above.

- Insert: `heap.push(Edge(u, -1, distance));`
- Get Minimum: `top = heap.top(); u = top.u; distance = top.w;`
- Remove top: `heap.pop();` (goes after the get minimum method)

Demo