CSEN 703 - Analysis and Design of Algorithms

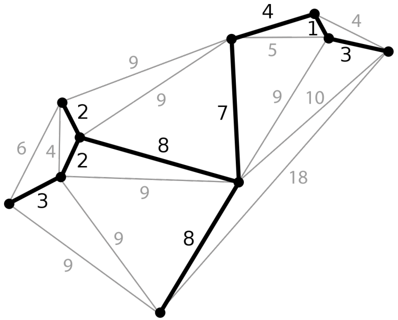
Lecture 9 - Graph Algorithms II

Dr. Nourhan Ehab
nourhan.ehab@guc.edu.eg

Department of Computer Science and Engineering
Faculty of Media Engineering and Technology

Outline

Minimum Spanning Trees



Definition

A Spanning Tree is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles. A Minimum Spanning Tree (MST) is a spanning tree with the minimum total edge weight.

Motivating Example

Example

A town has a set of houses and a set of roads. A road connects only two houses u and v and has a repair cost $w(u, v)$. We want to repair enough and (no more) roads such that everyone stays connected (can reach every house from all other houses) such that the total repair cost is minimum.

Motivating Example

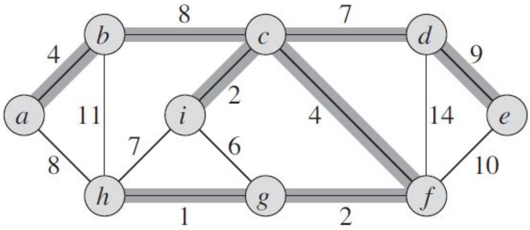
GUC German University in Cairo

Example

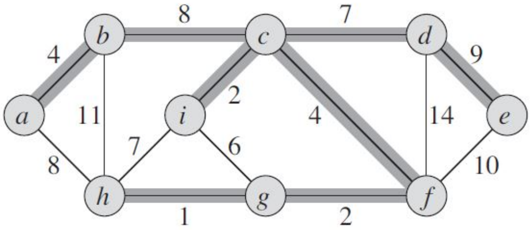
A town has a set of houses and a set of roads. A road connects only two houses u and v and has a repair cost $w(u, v)$. We want to repair enough and (no more) roads such that everyone stays connected (can reach every house from all other houses) such that the total repair cost is minimum.

Other applications in communication networks and circuit design.

Properties of MST

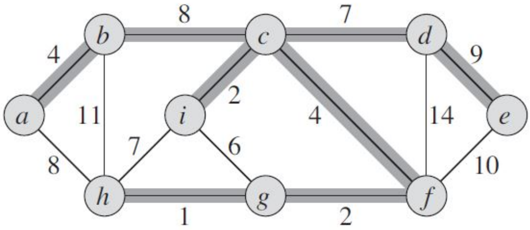


Properties of MST



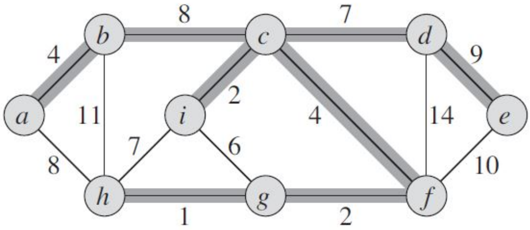
- An MST always has $|V| - 1$ edges.

Properties of MST



- An MST always has $|V| - 1$ edges.
- An MST is acyclic.

Properties of MST



- An MST always has $|V| - 1$ edges.
- An MST is acyclic.
- An MST may not be unique. We get another MST by replacing (b, c) with (a, h) with a total cost of 37.

Generic MST Algorithm

We build a set A of edges:

Generic MST Algorithm

We build a set A of edges:

- Initially A is empty edges.

Generic MST Algorithm

We build a set A of edges:

- Initially A is empty edges.
- As we add edges to A, we maintain that A remains a subset of some MST.

Generic MST Algorithm

We build a set A of edges:

- Initially A is empty edges.
- As we add edges to A, we maintain that A remains a subset of some MST.
- We call an edge (u, v) a safe edge if and only if A is a MST and $A \cup \{(u, v)\}$ is also a MST.

Generic MST Algorithm

We build a set A of edges:

- Initially A is empty edges.
- As we add edges to A, we maintain that A remains a subset of some MST.
- We call an edge (u, v) a safe edge if and only if A is a MST and $A \cup \{(u, v)\}$ is also a MST.

```
GENERIC-MST(G, w)
A = ∅
while A does not form a spanning tree
    find an edge (u, v) that is safe for A
    A = A ∪ {(u, v)}
return A
```

Generic MST Algorithm

We build a set A of edges:

- Initially A is empty edges.
- As we add edges to A, we maintain that A remains a subset of some MST.
- We call an edge (u, v) a safe edge if and only if A is a MST and $A \cup \{(u, v)\}$ is also a MST.

```
GENERIC-MST(G, w)
A = ∅
while A does not form a spanning tree
    find an edge (u, v) that is safe for A
    A = A ∪ {(u, v)}
return A
```

Safe Edges

- But how do we find a safe edge?

Safe Edges

- But how do we find a safe edge?
- Let $S \subset V$ be any set of vertices that includes u but not v (so that v is in $V{-}S$).

Safe Edges

- But how do we find a safe edge?
- Let $S \subset V$ be any set of vertices that includes u but not v (so that v is in $V - S$).
- In any MST there has to be one edge that connects S with $V - S$.

Safe Edges

- But how do we find a safe edge?
- Let $S \subset V$ be any set of vertices that includes u but not v (so that v is in $V-S$).
- In any MST there has to be one edge that connects S with $V-S$.
- We can choose the one with minimum weight ((u, v) in this case). We call a safe edge with the minimum weight a light edge.

Observations about Generic MST Algorithm

- A is a forest of connected components. Initially each component is a single vertex.

Observations about Generic MST Algorithm

- A is a forest of connected components. Initially each component is a single vertex.
- A safe edge merges two components into one.

Observations about Generic MST Algorithm

- A is a forest of connected components. Initially each component is a single vertex.
- A safe edge merges two components into one.
- After $|V| - 1$ iterations of the loop, we are down to only one component, the MST.

Observations about Generic MST Algorithm

- A is a forest of connected components. Initially each component is a single vertex.
- A safe edge merges two components into one.
- After $|V| - 1$ iterations of the loop, we are down to only one component, the MST.
- This means that Generic-MST is actually a greedy algorithm!

Observations about Generic MST Algorithm

- A is a forest of connected components. Initially each component is a single vertex.
- A safe edge merges two components into one.
- After $|V| - 1$ iterations of the loop, we are down to only one component, the MST.
- This means that Generic-MST is actually a greedy algorithm!
- We will look next into two implementations of the Generic-MST algorithm: Kruskal's algorithm and Prim's algorithm.

Kruskal's Algorithm

- Idea: Add edges to A in increasing order of weight (light edges first).

Kruskal's Algorithm

- Idea: Add edges to A in increasing order of weight (light edges first).
- If the next edge does not induce a cycle among the current set of edges, then it is added to A.

Kruskal's Algorithm

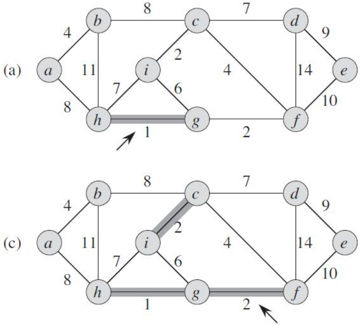
- Idea: Add edges to A in increasing order of weight (light edges first).
- If the next edge does not induce a cycle among the current set of edges, then it is added to A.
- If it does, then this edge is passed over, and we consider the next edge in order.

Kruskal's Algorithm

GUC German University in Cairo

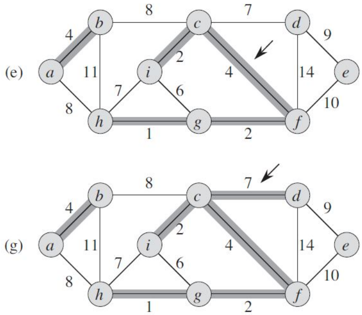
- Idea: Add edges to A in increasing order of weight (light edges first).
- If the next edge does not induce a cycle among the current set of edges, then it is added to A.
- If it does, then this edge is passed over, and we consider the next edge in order.
- As this algorithm runs, the edges of A will induce a forest on the vertices and the trees of this forest are merged together until we have a single tree containing all vertices.

Kruskal's Algorithm - Example



Kruskal's Algorithm - Example

GUC German University in Cairo



Visualization: https://visualgo.net/en/mst

Cycles Detection

Cycles Detection

- We can do this by DFS, but this is expensive!

Cycles Detection

- We can do this by DFS, but this is expensive!
- Solution? Use Disjoint Set data structure. It supports three operations:
  - Create-Set(u): creates a set containing u.
  - Find-Set(u): find the set that contains u.
  - Union(u,v): merge the sets containing u and v.

Cycles Detection

- We can do this by DFS, but this is expensive!
- Solution? Use Disjoint Set data structure. It supports three operations:
  - Create-Set(u): creates a set containing u.
  - Find-Set(u): find the set that contains u.
  - Union(u,v): merge the sets containing u and v.
- The vertices of the graph will be elements to be stored in the sets; the sets will be vertices in each tree of A (stored as a simple list of edges).

Kruskal's Algorithm - Pseudo Code

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ G.V
    MAKE-SET(v)
sort the edges of G.E into nondecreasing order by weight w
for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
return A
```

Kruskal's Algorithm - Analysis

- $O(V)$ to create V sets.
- $O(E\ log(E))$ to sort the edges in line 4.
- $O(E)$ iterations for the loop from lines 5-8.
- There is an implementation of disjoint sets called disjoint forests that performs the find and union operations in $O(log(V))$ time.
- Total time then is $O(E\ log(E) + E\ log(V))$. Assuming that the graph is connected, then $E \geq V - 1$. Therefore, the runtime is $O(E\ log(E))$.

Prim's Algorithm

- It works by adding leaves one at a time to the current tree.

Prim's Algorithm

- It works by adding leaves one at a time to the current tree.
- Start with the root vertex r (it can be any vertex). At any time, the subset of edges A forms a single tree.

Prim's Algorithm

- It works by adding leaves one at a time to the current tree.
- Start with the root vertex r (it can be any vertex). At any time, the subset of edges A forms a single tree.
- Let S be the vertices of A. At each step, a light edge connecting a vertex in S to a vertex in $V - S$ is added to the tree.

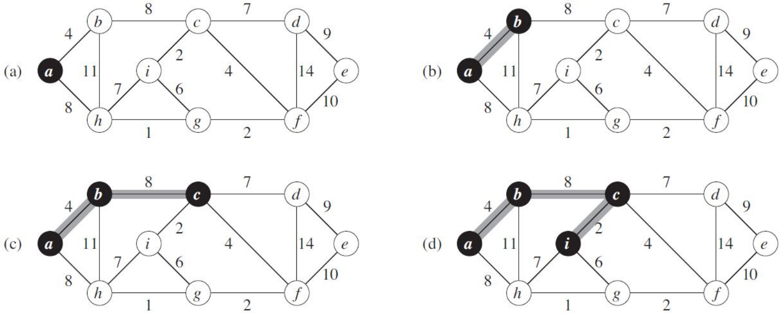
Prim's Algorithm

- It works by adding leaves one at a time to the current tree.
- Start with the root vertex r (it can be any vertex). At any time, the subset of edges A forms a single tree.
- Let S be the vertices of A. At each step, a light edge connecting a vertex in S to a vertex in $V - S$ is added to the tree.
- The tree grows until it spans all the vertices in V.

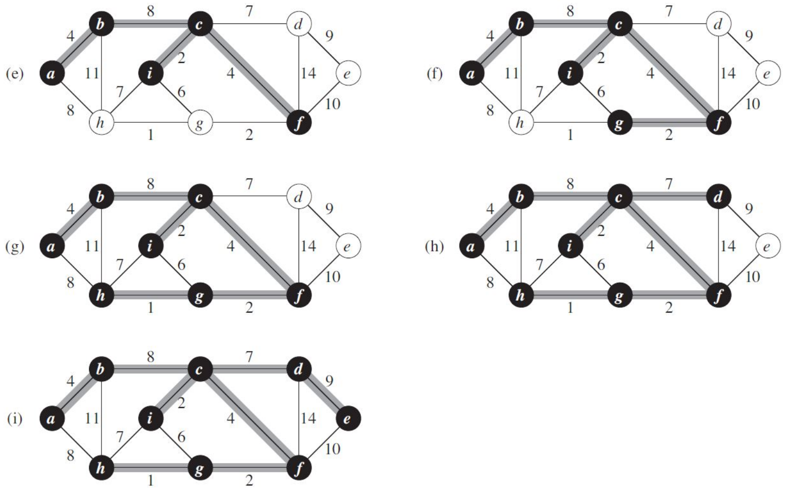
Prim's Algorithm

- It works by adding leaves one at a time to the current tree.
- Start with the root vertex r (it can be any vertex). At any time, the subset of edges A forms a single tree.
- Let S be the vertices of A. At each step, a light edge connecting a vertex in S to a vertex in $V - S$ is added to the tree.
- The tree grows until it spans all the vertices in V.
- A priority queue is used supporting the following operations:
    - Insert (Q, u, key): Insert u with the key value key in Q.
    - Extract_Min(Q): Extract the item with minimum key value in Q.

Prim's Algorithm - Example



Prim's Algorithm - Example



Prim's Algorithm - Pseudo Code

```
MST-PRIM(G, w, r)
for each u ∈ G.V
    u.key = ∞
    u.π = NIL
r.key = 0
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
```

Prim's Algorithm - Analysis

- Initialization takes $O(V)$ time.
- Loop on lines 6-11 extracts V vertices.
- Extract-Min takes $O(log(V))$.
- Total time for all Extract-Mins is thus $O(V\ log(V))$.
- The total number of iterations of the loop on lines 8-11 is $O(E)$.
- Testing membership of v in Q takes $O(log(V))$. Changing the key takes constant time.
- Total running time is then $O(V\ log(V) + E\ log(V))$. Assuming that the graph is connected, this is $O(E\ log(E))$.

Outline

1. Minimum Spanning Trees
2. Shortest Path Algorithms
3. Recap

Shortest Path Problems

- Generalization of BFS to weighted graphs. Shortest here is defined by the weight of the edges not their number.

Shortest Path Problems

- Generalization of BFS to weighted graphs. Shortest here is defined by the weight of the edges not their number.
- Problem Variants:
  1. Single-source shortest path: given a graph G, we want to find a shortest path from a given source vertex to all other vertices.
  2. All-pairs shortest path: Find a shortest path from u to v for every pair of vertices u and v.

Shortest Path Problems

- Generalization of BFS to weighted graphs. Shortest here is defined by the weight of the edges not their number.
- Problem Variants:
  1. Single-source shortest path: given a graph G, we want to find a shortest path from a given source vertex to all other vertices.
  2. All-pairs shortest path: Find a shortest path from u to v for every pair of vertices u and v.
- We shall focus on the single-source shortest path problem in this course.

Dijkstra's Algorithm

- Works on weighted graphs with positive weights.

Dijkstra's Algorithm

- Works on weighted graphs with positive weights.
- Dijkstra's algorithm is a greedy algorithm choosing to visit the cheapest connected vertices first.

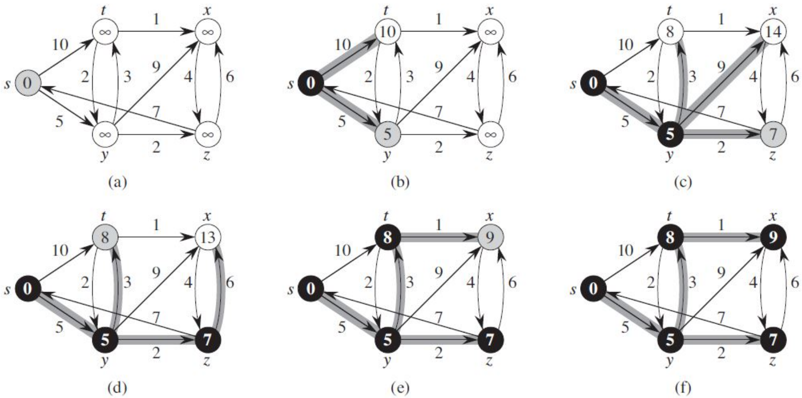
Dijkstra's Algorithm

- Works on weighted graphs with positive weights.
- Dijkstra's algorithm is a greedy algorithm choosing to visit the cheapest connected vertices first.
- The algorithm takes as input a graph G and a source vertex x.

Dijkstra's Algorithm

- Works on weighted graphs with positive weights.
- Dijkstra's algorithm is a greedy algorithm choosing to visit the cheapest connected vertices first.
- The algorithm takes as input a graph G and a source vertex x.
- Data Structures:
  - S: set of vertices whose shortest paths from x has been determined (visited vertices).
  - Q: a priority queue containing the non-visited vertices ordered by their distance from x.

Dijkstra's Algorithm - Example



Dijkstra's Algorithm - Pseudo Code

```
DIJKSTRA(G,w,s)
    INITIALIZE-SINGLE-SOURCE(G,s)      % distance is infinite, predecessor is NIL
    S ← ∅
    Q ←V[G]                             % list of vertices of the graph G
    While Q ≠ ∅
        do u ← EXTRACT-MIN(Q)          % pick a node with shortest path estimate to s
            S ← S ∪ {u}
            for each vertex v ∈ Adj[u]
                do RELAX(u,v,w)        % test whether we can improve shortest path
                                         distance to v going through u. w is the weight
                                         function between adjacent pairs of nodes w(u,v)
```

```
RELAX(u,v,w)
    If d[v] > d[u] + w(u,v)
        then d[v] ← d[u] + w(u,v)
            π[v] ← u
```

Dijkstra's Algorithm - Analysis

GUC German University in Cairo

- Initialization takes $O(V)$.
- The loop on lines 4-8 loops $O(V)$ times.
- Extract-Min takes $O(log(V))$.
- The total number of iterations of the loop on lines 7-8 are $O(E)$.
- Relax takes constant time.
- Total running time is $O(V\ log(V) + E)$.

Bellman-Ford Algorithm

- The Bellman-Ford algorithm solves the single-source shortest-paths problem in the general case in which edge weights may be negative.

Bellman-Ford Algorithm

- The Bellman-Ford algorithm solves the single-source shortest-paths problem in the general case in which edge weights may be negative.
- If there is a negative-weight cycle that is reachable from the source, the algorithm indicates that no solution exists. If there is no such cycle, the algorithm produces the shortest paths and their weights.

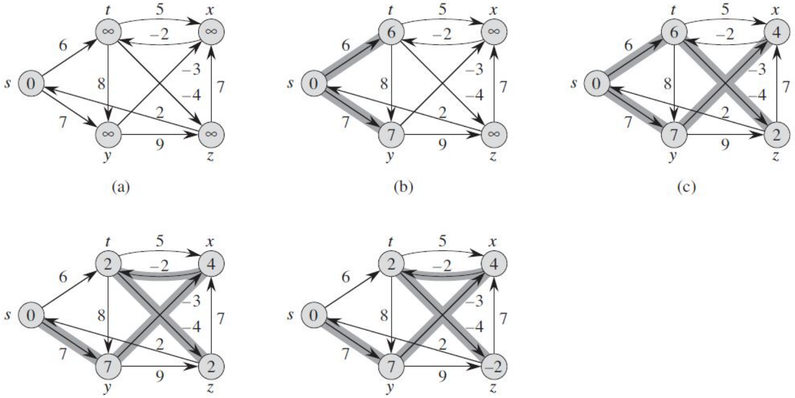
Bellman-Ford Algorithm

- The Bellman-Ford algorithm solves the single-source shortest-paths problem in the general case in which edge weights may be negative.
- If there is a negative-weight cycle that is reachable from the source, the algorithm indicates that no solution exists. If there is no such cycle, the algorithm produces the shortest paths and their weights.
- The algorithm relaxes edges, progressively decreasing an estimate $v.d$ on the weight of a shortest path from the source s to each vertex until it achieves the actual shortest-path weight.

Bellman-Ford Algorithm - Pseudo Code


```
BELLMAN-FORD(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
for i = 1 to |G.V| - 1
    for each edge (u, v) ∈ G.E
        RELAX(u, v, w)
for each edge (u, v) ∈ G.E
    if v.d > u.d + w(u, v)
        return FALSE
return TRUE
```

Bellman-Ford Algorithm - Example



(a) (b) (c) (d) (e)

Bellman-Ford Algorithm - Analysis

- Initialization takes $O(V)$.
- Loop on line 2 loops $O(V)$ times and the inner loops does $O(E)$ iterations.
- The loop on line 5 does $O(E)$ iterations.
- Total time is $O(VE)$.

Outline

1. Minimum Spanning Trees
2. Shortest Path Algorithms
3. Recap

Points to Take Home

1. Minimum Spanning Trees.
2. Kruskal's and Prim's Algorithm.
3. Dijkstra's Algorithm.
4. Bellman-Ford Algorithm.
5. Reading Material:
   - Introduction to Algorithms. Chapter 23, Sections 23.1 and 23.2.
   - Introduction to Algorithms. Chapter 24, Sections 24.1 and 24.3.

Next Lecture: Graph Algorithms III!

Due Credits

GUC German University in Cairo

The presented material is based on:

1. Previous editions of the course at the GUC due to Dr. Wael Aboulsaadat, Dr. Haythem Ismail, Dr. Amr Desouky, and Dr. Carmen Gervet.
2. Stony Brook University's Analysis of Algorithms Course.
3. MIT's Introduction to Algorithms Course.
4. Stanford's Design and Analysis of Algorithms Course.