Data Structures and Algorithms

Graphs
Outline

• Graphs
• Depth-First Search
• Breadth-First Search
• Directed Graphs
• Shortest Path
• Minimum Spanning Tree
Graphs

• A graph is a pair \((V, E)\), where
  – \(V\) is a set of nodes, called vertices
  – \(E\) is a collection of pairs of vertices, called edges
  – Vertices and edges are positions and store elements
• Example:
  – A vertex represents an airport and stores the three-letter airport code
  – An edge represents a flight route between two airports and stores the mileage of the route
Edge Types

- **Directed edge**
  - ordered pair of vertices \((u,v)\)
  - first vertex \(u\) is the origin
  - second vertex \(v\) is the destination
  - e.g., a flight

- **Undirected edge**
  - unordered pair of vertices \((u,v)\)
  - e.g., a flight route

- **Directed graph**
  - all the edges are directed
  - e.g., route network

- **Undirected graph**
  - all the edges are undirected
  - e.g., flight network

\[\text{ORD} \quad \text{flight} \quad AA 1206 \quad \text{PVD}\]

\[\text{ORD} \quad 849 \quad \text{miles} \quad \text{PVD}\]
Applications

• Electronic circuits
  – Printed circuit board
  – Integrated circuit
• Transportation networks
  – Highway network
  – Flight network
• Computer networks
  – Local area network
  – Internet
  – Web
• Databases
  – Entity-relationship diagram

Phạm Bảo Sơn - DSA
Terminology

- End vertices (or endpoints) of an edge
  - U and V are the endpoints of a
- Edges incident on a vertex
  - a, d, and b are incident on V
- Adjacent vertices
  - U and V are adjacent
- Degree of a vertex
  - X has degree 5
- Parallel edges
  - h and i are parallel edges
- Self-loop
  - j is a self-loop
Terminology (cont.)

- **Path**
  - sequence of alternating vertices and edges
  - begins with a vertex
  - ends with a vertex
  - each edge is preceded and followed by its endpoints

- **Simple path**
  - path such that all its vertices and edges are distinct

- **Examples**
  - $P_1=(V,b,X,h,Z)$ is a simple path
  - $P_2=(U,c,W,e,X,g,Y,f,W,d,V)$ is a path that is not simple
• **Cycle**
  – circular sequence of alternating vertices and edges
  – each edge is preceded and followed by its endpoints

• **Simple cycle**
  – cycle such that all its vertices and edges are distinct

• **Examples**
  – $C_1=(V,b,X,g,Y,f,W,c,U,a,\epsilon)$ is a simple cycle
  – $C_2=(U,c,W,e,X,g,Y,f,W,d,V,a,\epsilon)$ is a cycle that is not simple
Properties

Property 1

\[ \sum_v \text{deg}(v) = 2m \]

Proof: each edge is counted twice

Property 2

In an undirected graph with no self-loops and no multiple edges

\[ m \leq n (n - 1)/2 \]

Proof: each vertex has degree at most \((n - 1)\)

What is the bound for a directed graph?

**Notation**

- \( n \): number of vertices
- \( m \): number of edges
- \( \text{deg}(v) \): degree of vertex \( v \)

**Example**

- \( n = 4 \)
- \( m = 6 \)
- \( \text{deg}(v) = 3 \)
Main Methods of the Graph ADT

- Vertices and edges
  - are positions
  - store elements
- Accessor methods
  - endVertices(e): an array of the two endvertices of e
  - opposite(v, e): the vertex opposite of v on e
  - areAdjacent(v, w): true iff v and w are adjacent
  - replace(v, x): replace element at vertex v with x
  - replace(e, x): replace element at edge e with x
- Update methods
  - insertVertex(o): insert a vertex storing element o
  - insertEdge(v, w, o): insert an edge (v, w) storing element o
  - removeVertex(v): remove vertex v (and its incident edges)
  - removeEdge(e): remove edge e
- Iterator methods
  - incidentEdges(v): edges incident to v
  - vertices(): all vertices in the graph
  - edges(): all edges in the graph
Data Structures for Graphs

- Edge List
- Adjacency List
- Adjacency Matrix
Edge List Structure

- **Vertex object**
  - element
  - reference to position in vertex sequence
- **Edge object**
  - element
  - origin vertex object
  - destination vertex object
  - reference to position in edge sequence
- **Vertex sequence**
  - sequence of vertex objects
- **Edge sequence**
  - sequence of edge objects
Adjacency List Structure

- Edge list structure
- Incidence sequence for each vertex
  - sequence of references to edge objects of incident edges
- Augmented edge objects
  - references to associated positions in incidence sequences of end vertices

Phạm Bảo Sơn - DSA
Adjacency Matrix Structure

- Edge list structure
- Augmented vertex objects
  - Integer key (index) associated with vertex
- 2D-array adjacency array
  - Reference to edge object for adjacent vertices
  - Null for non-adjacent vertices
- The "old fashioned" version just has 0 for no edge and 1 for edge

Phạm Bảo Sơn - DSA
Asymptotic Performance

- $n$ vertices, $m$ edges
- no parallel edges
- no self-loops
- Bounds are “big-Oh”

<table>
<thead>
<tr>
<th></th>
<th>Edge List</th>
<th>Adjacency List</th>
<th>Adjacency Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space $n + m$</td>
<td>$n + m$</td>
<td>$n + m$</td>
<td>$n^2$</td>
</tr>
<tr>
<td>incidentEdges$(v)$</td>
<td>$m$</td>
<td>deg$(v)$</td>
<td>$n$</td>
</tr>
<tr>
<td>areAdjacent $(v, w)$</td>
<td>$m$</td>
<td>min(deg$(v)$, deg$(w)$)</td>
<td>1</td>
</tr>
<tr>
<td>insertVertex$(o)$</td>
<td>1</td>
<td>1</td>
<td>$n^2$</td>
</tr>
<tr>
<td>insertEdge$(v, w, o)$</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>removeVertex$(v)$</td>
<td>$m$</td>
<td>deg$(v)$</td>
<td>$n^2$</td>
</tr>
<tr>
<td>removeEdge$(e)$</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Depth-First Search
Subgraphs

• A subgraph S of a graph G is a graph such that
  – The vertices of S are a subset of the vertices of G
  – The edges of S are a subset of the edges of G

• A spanning subgraph of G is a subgraph that contains all the vertices of G
Connectivity

• A graph is connected if there is a path between every pair of vertices
• A connected component of a graph $G$ is a maximal connected subgraph of $G$
Trees and Forests

• A (free) tree is an undirected graph $T$ such that
  – $T$ is connected
  – $T$ has no cycles
This definition of tree is different from the one of a rooted tree
• A forest is an undirected graph without cycles
• The connected components of a forest are trees
Spanning Trees and Forests

- A spanning tree of a connected graph is a spanning subgraph that is a tree.
- A spanning tree is not unique unless the graph is a tree.
- Spanning trees have applications to the design of communication networks.
- A spanning forest of a graph is a spanning subgraph that is a forest.
Depth-First Search

- Depth-first search (DFS) is a general technique for traversing a graph
- A DFS traversal of a graph G
  - Visits all the vertices and edges of G
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning forest of G
- DFS on a graph with \( n \) vertices and \( m \) edges takes \( O(n + m) \) time
- DFS can be further extended to solve other graph problems
  - Find and report a path between two given vertices
  - Find a cycle in the graph
- Depth-first search is to graphs what Euler tour is to binary trees
DFS Algorithm

- The algorithm uses a mechanism for setting and getting “labels” of vertices and edges

**Algorithm DFS(G)**

**Input** graph \( G \)

**Output** labeling of the edges of \( G \) as discovery edges and back edges

for all \( u \in G\) vertices()

\[ \text{setLabel}(u, \text{UNEXPLORED}) \]

for all \( e \in G\) edges()

\[ \text{setLabel}(e, \text{UNEXPLORED}) \]

for all \( v \in G\) vertices()

if \( \text{getLabel}(v) = \text{UNEXPLORED} \)

\[ \text{DFS}(G, v) \]

Algorithm **DFS(G, v)**

**Input** graph \( G \) and a start vertex \( v \) of \( G \)

**Output** labeling of the edges of \( G \) in the connected component of \( v \) as discovery edges and back edges

\[ \text{setLabel}(v, \text{VISITED}) \]

for all \( e \in G\) incidentEdges(v)

if \( \text{getLabel}(e) = \text{UNEXPLORED} \)

\[ w \leftarrow \text{opposite}(v,e) \]

if \( \text{getLabel}(w) = \text{UNEXPLORED} \)

\[ \text{setLabel}(e, \text{DISCOVERY}) \]

\[ \text{DFS}(G, w) \]

else

\[ \text{setLabel}(e, \text{BACK}) \]
Example

- A: unexplored vertex
- B: visited vertex
- C: unexplored edge
- D: discovery edge
- E: back edge

Phạm Bảo Sơn - DSA
Example (cont.)
DFS and Maze Traversal

- The DFS algorithm is similar to a classic strategy for exploring a maze
  - We mark each intersection, corner and dead end (vertex) visited
  - We mark each corridor (edge) traversed
  - We keep track of the path back to the entrance (start vertex) by means of a rope (recursion stack)
Properties of DFS

Property 1

$DFS(G, v)$ visits all the vertices and edges in the connected component of $v$

Property 2

The discovery edges labeled by $DFS(G, v)$ form a spanning tree of the connected component of $v$
Analysis of DFS

• Setting/getting a vertex/edge label takes $O(1)$ time
• Each vertex is labeled twice
  – once as UNEXPLORED
  – once as VISITED
• Each edge is labeled twice
  – once as UNEXPLORED
  – once as DISCOVERY or BACK
• Method incidentEdges is called once for each vertex
• DFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure
  – Recall that $\Sigma_v \deg(v) = 2m$
Path Finding

- We can specialize the DFS algorithm to find a path between two given vertices \( u \) and \( z \).
- We call \( DFS(G, u) \) with \( u \) as the start vertex.
- We use a stack \( S \) to keep track of the path between the start vertex and the current vertex.
- As soon as destination vertex \( z \) is encountered, we return the path as the contents of the stack.

Algorithm \( \text{pathDFS}(G, v, z) \)

\begin{align*}
\text{setLabel}(v, \text{VISITED}) \\
\text{S.push}(v) \\
\text{if } v = z \\
\quad \text{return } \text{S.elements()} \\
\text{for all } e \in G.\text{incidentEdges}(v) \\
\quad \text{if } \text{getLabel}(e) = \text{UNEXPLORED} \\
\quad \quad w \leftarrow \text{opposite}(v, e) \\
\quad \quad \text{if } \text{getLabel}(w) = \text{UNEXPLORED} \\
\quad \quad \quad \text{setLabel}(e, \text{DISCOVERY}) \\
\quad \quad \quad \text{S.push}(e) \\
\quad \quad \text{pathDFS}(G, w, z) \\
\quad \quad \text{S.pop}(e) \\
\quad \text{else} \\
\quad \quad \quad \text{setLabel}(e, \text{BACK}) \\
\quad \quad \text{S.pop}(v)
\end{align*}
Cycle Finding

- We can specialize the DFS algorithm to find a simple cycle.
- We use a stack $S$ to keep track of the path between the start vertex and the current vertex.
- As soon as a back edge $(v, w)$ is encountered, we return the cycle as the portion of the stack from the top to vertex $w$.

Algorithm cycleDFS($G$, $v$, $z$)

1. $setLabel(v, VISITED)$
2. $S.push(v)$
3. for all $e \in G.incidentEdges(v)$
   - if $getLabel(e) = UNEXPLORED$
     - $w \leftarrow opposite(v, e)$
     - $S.push(e)$
     - if $getLabel(w) = UNEXPLORED$
       - $setLabel(e, DISCOVERY)$
       - $pathDFS(G, w, z)$
       - $S.pop(e)$
     - else
       - $T \leftarrow$ new empty stack
       - repeat
         - $o \leftarrow S.pop()$
         - $T.push(o)$
       - until $o = w$
     - return $T.elements()$
   - else
     - $S.pop(v)$

Breadth-First Search
Breadth-First Search

• Breadth-first search (BFS) is a general technique for traversing a graph
• A BFS traversal of a graph G
  – Visits all the vertices and edges of G
  – Determines whether G is connected
  – Computes the connected components of G
  – Computes a spanning forest of G
• BFS on a graph with $n$ vertices and $m$ edges takes $O(n + m)$ time
• BFS can be further extended to solve other graph problems
  – Find and report a path with the minimum number of edges between two given vertices
  – Find a simple cycle, if there is one
BFS Algorithm

- The algorithm uses a mechanism for setting and getting “labels” of vertices and edges

Algorithm **BFS(G)**
- **Input** graph G
- **Output** labeling of the edges and partition of the vertices of G

for all u ∈ G.vertices()
    setLabel(u, UNEXPLORED)

for all e ∈ G.edges()
    setLabel(e, UNEXPLORED)

for all v ∈ G.vertices()
    if getLabel(v) = UNEXPLORED
        BFS(G, v)

Algorithm **BFS(G, s)**
- $L_0 \leftarrow$ new empty sequence
- $L_0$.insertLast(s)
- setLabel(s, VISITED)
- $i \leftarrow 0$

while ¬$L_i$.isEmpty()
    $L_{i+1} \leftarrow$ new empty sequence
    for all $v \in L_i$.elements()
        for all $e \in G$.incidentEdges($v$)
            if getLabel($e$) = UNEXPLORED
                $w \leftarrow$ opposite($v$, $e$)
                if getLabel($w$) = UNEXPLORED
                    setLabel($e$, DISCOVERY)
                    setLabel($w$, VISITED)
                    $L_{i+1}$.insertLast($w$)
                else
                    setLabel($e$, CROSS)

        $i \leftarrow i + 1
Example

- **unexplored vertex**
- **visited vertex**
- **unexplored edge**
- **discovery edge**
- **cross edge**

Phạm Bảo Sơn - DSA
Example (cont.)
Example (cont.)
Properties

Notation

\( G_s \): connected component of \( s \)

Property 1

\( BFS(G, s) \) visits all the vertices and edges of \( G_s \)

Property 2

The discovery edges labeled by \( BFS(G, s) \) form a spanning tree \( T_s \) of \( G_s \)

Property 3

For each vertex \( v \) in \( L_i \)

- The path of \( T_s \) from \( s \) to \( v \) has \( i \) edges
- Every path from \( s \) to \( v \) in \( G_s \) has at least \( i \) edges
Analysis

• Setting/getting a vertex/edge label takes $O(1)$ time
• Each vertex is labeled twice
  – once as UNEXPLORED
  – once as VISITED
• Each edge is labeled twice
  – once as UNEXPLORED
  – once as DISCOVERY or CROSS
• Each vertex is inserted once into a sequence $L_i$
• Method incidentEdges is called once for each vertex
• BFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure
  – Recall that $\sum_v \deg(v) = 2m$
Applications

• Using the template method pattern, we can specialize the BFS traversal of a graph $G$ to solve the following problems in $O(n + m)$ time
  – Compute the connected components of $G$
  – Compute a spanning forest of $G$
  – Find a simple cycle in $G$, or report that $G$ is a forest
  – Given two vertices of $G$, find a path in $G$ between them with the minimum number of edges, or report that no such path exists
# DFS vs. BFS

<table>
<thead>
<tr>
<th>Applications</th>
<th>DFS</th>
<th>BFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spanning forest, connected components, paths, cycles</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Shortest paths</td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>Biconnected components</td>
<td>√</td>
<td></td>
</tr>
</tbody>
</table>

**DFS**

- A
- B
- C
- D
- E
- F

**BFS**

- A
- B
- C
- D
- E
- F
DFS vs. BFS (cont.)

Back edge \((v,w)\)
- \(w\) is an ancestor of \(v\) in the tree of discovery edges

Cross edge \((v,w)\)
- \(w\) is in the same level as \(v\) or in the next level in the tree of discovery edges

DFS

BFS

Phạm Bảo Sơn - DSA
Digraphs

• A digraph is a graph whose edges are all directed
  – Short for “directed graph”

• Applications
  – one-way streets
  – flights
  – task scheduling
Digraph Properties

• A graph $G=(V,E)$ such that
  – Each edge goes in one direction:
    • Edge $(a,b)$ goes from $a$ to $b$, but not $b$ to $a$.
• If $G$ is simple, $m \leq n^*(n-1)$.
• If we keep in-edges and out-edges in separate adjacency lists, we can perform listing of in-edges and out-edges in time proportional to their size.
Digraph Application

- **Scheduling**: edge \((a, b)\) means task \(a\) must be completed before \(b\) can be started.

Diagram:

- ics21 → ics22 → ics23
- i cs51 → ics53
- i cs31 → i cs141 → i cs121 → i cs171
- The good life

Phạm Bảo Sơn - DSA 45
Directed DFS

- We can specialize the traversal algorithms (DFS and BFS) to digraphs by traversing edges only along their direction.
- In the directed DFS algorithm, we have three types of edges:
  - back edges
  - forward edges
  - cross edges
- A directed DFS starting at a vertex $s$ determines the vertices reachable from $s$. 
Reachability

• DFS tree rooted at v: vertices reachable from v via directed paths
Strong Connectivity

• Each vertex can reach all other vertices
**Strong Connectivity Algorithm**

- Pick a vertex \( v \) in \( G \).
- Perform a DFS from \( v \) in \( G \).
  - If there’s a \( w \) not visited, print “no”.
- Let \( G’ \) be \( G \) with edges reversed.
- Perform a DFS from \( v \) in \( G’ \).
  - If there’s a \( w \) not visited, print “no”.
  - Else, print “yes”.

- Running time: \( O(n+m) \).
Strongly Connected Components

- Maximal subgraphs such that each vertex can reach all other vertices in the subgraph
- Can also be done in $O(n+m)$ time using DFS, but is more complicated (similar to biconnectivity).

\[
\{ a, c, g \} \\
\{ f, d, e, b \}
\]
Transitive Closure

• Given a digraph $G$, the transitive closure of $G$ is the digraph $G^*$ such that
  – $G^*$ has the same vertices as $G$
  – if $G$ has a directed path from $u$ to $v$ ($u \neq v$), $G^*$ has a directed edge from $u$ to $v$
• The transitive closure provides reachability information about a digraph
Computing the Transitive Closure

- We can perform DFS starting at each vertex
  - $O(n(n+m))$

If there's a way to get from $A$ to $B$ and from $B$ to $C$, then there's a way to get from $A$ to $C$.

Alternatively ... Use dynamic programming: The Floyd-Warshall Algorithm
Floyd-Warshall Transitive Closure

- Idea #1: Number the vertices 1, 2, ..., n.
- Idea #2: Consider paths that use only vertices numbered 1, 2, ..., k, as intermediate vertices:

Uses only vertices numbered 1, ..., k
(add this edge if it’s not already in)

Uses only vertices numbered 1, ..., k-1

Uses only vertices numbered 1, ..., k-1
Floyd-Warshall’s Algorithm

- Floyd-Warshall’s algorithm numbers the vertices of $G$ as $v_1, \ldots, v_n$ and computes a series of digraphs $G_0, \ldots, G_n$
  - $G_0 = G$
  - $G_k$ has a directed edge $(v_i, v_j)$ if $G$ has a directed path from $v_i$ to $v_j$ with intermediate vertices in the set $\{v_1, \ldots, v_k\}$
- We have that $G_n = G^*$
- In phase $k$, digraph $G_k$ is computed from $G_{k-1}$
- Running time: $O(n^3)$, assuming areAdjacent is $O(1)$ (e.g., adjacency matrix)

Algorithm $FloydWarshall(G)$

Input: digraph $G$
Output: transitive closure $G^*$ of $G$

1. $i \leftarrow 1$
2. for all $v \in G.\text{vertices()}$
   - denote $v$ as $v_i$
3. $i \leftarrow i + 1$
4. $G_0 \leftarrow G$
5. for $k \leftarrow 1$ to $n$
6.   - $G_k \leftarrow G_{k-1}$
7.   - for $i \leftarrow 1$ to $n$ ($i \neq k$)
8.     - for $j \leftarrow 1$ to $n$ ($j \neq i, k$)
9.       - if $G_{k-1}.\text{areAdjacent}(v_i, v_k) \land$
10.          - $G_{k-1}.\text{areAdjacent}(v_k, v_j)$
11.             - if $\neg G_k.\text{areAdjacent}(v_i, v_j)$
12.               - $G_k.\text{insertDirectedEdge}(v_i, v_j, k)$
13. return $G_n$
Floyd-Warshall Example
Floyd-Warshall, Iteration 1

The image cannot be viewed.
Floyd-Warshall, Iteration 2
Floyd-Warshall, Iteration 3
Floyd-Warshall, Iteration 4
Floyd-Warshall, Iteration 5
Floyd-Warshall, Iteration 6

Image cannot be viewed.
Floyd-Warshall, Conclusion
A directed acyclic graph (DAG) is a digraph that has no directed cycles.

A topological ordering of a digraph is a numbering $v_1, \ldots, v_n$ of the vertices such that for every edge $(v_i, v_j)$, we have $i < j$.

Example: in a task scheduling digraph, a topological ordering a task sequence that satisfies the precedence constraints.

Theorem
A digraph admits a topological ordering if and only if it is a DAG.
Topological Sorting

- Number vertices, so that \((u,v)\) in \(E\) implies \(u < v\)

A typical student day:

1. Wake up
2. Study computer sci.
3. Eat
4. Nap
5. More c.s.
6. Work out
7. Play
8. Write c.s. program
9. Make cookies for professors
10. Sleep
11. Dream about graphs
Algorithm for Topological Sorting

Method TopologicalSort(G)
    H ← G // Temporary copy of G
    n ← G.numVertices()
    while H is not empty do
        Let v be a vertex with no outgoing edges
        Label v ← n
        n ← n - 1
        Remove v from H

• Running time: O(n + m). How…?
Topological Sorting
Algorithm using DFS

• Simulate the algorithm by using depth-first search

Algorithm topologicalDFS(G)
Input dag G
Output topological ordering of G
n ← G.numVertices()
for all u ∈ G.vertices()
    setLabel(u, UNEXPLORED)
for all e ∈ G.edges()
    setLabel(e, UNEXPLORED)
for all v ∈ G.vertices()
    if getLabel(v) = UNEXPLORED
        topologicalDFS(G, v)

• O(n+m) time.

Algorithm topologicalDFS(G, v)
Input graph G and a start vertex v of G
Output labeling of the vertices of G in the connected component of v
setLabel(v, VISITED)
for all e ∈ G.incidentEdges(v)
    if getLabel(e) = UNEXPLORED
        w ← opposite(v, e)
        if getLabel(w) = UNEXPLORED
            setLabel(e, DISCOVERY)
            topologicalDFS(G, w)
        else
            {e is a forward or cross edge}
Label v with topological number n
n ← n - 1

Phạm Bảo Sơn - DSA
Topological Sorting Example
Topological Sorting Example
Topological Sorting Example
Topological Sorting Example
Topological Sorting Example
Topological Sorting Example

Diagram showing a directed graph and an example of topological sorting.
Topological Sorting Example

Diagram of a directed graph showing the topological order of nodes.
Topological Sorting Example
Topological Sorting Example
Topological Sorting Example

Diagram showing a directed graph with nodes labeled 1 to 9.
Shortest Paths
Weighted Graphs

- In a weighted graph, each edge has an associated numerical value, called the weight of the edge.
- Edge weights may represent distances, costs, etc.
- Example:
  - In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports.
Shortest Paths

• Given a weighted graph and two vertices $u$ and $v$, we want to find a path of minimum total weight between $u$ and $v$.
  – Length of a path is the sum of the weights of its edges.
• Example:
  – Shortest path between Providence and Honolulu
• Applications
  – Internet packet routing
  – Flight reservations
  – Driving directions
Shortest Path Properties

Property 1:
A subpath of a shortest path is itself a shortest path

Property 2:
There is a tree of shortest paths from a start vertex to all the other vertices

Example:
Tree of shortest paths from Providence
Dijkstra’s Algorithm

• The distance of a vertex \( v \) from a vertex \( s \) is the length of a shortest path between \( s \) and \( v \)
• Dijkstra’s algorithm computes the distances of all the vertices from a given start vertex \( s \)
• Assumptions:
  – the graph is connected
  – the edges are undirected
  – the edge weights are nonnegative

• We grow a “cloud” of vertices, beginning with \( s \) and eventually covering all the vertices
• We store with each vertex \( v \) a label \( d(v) \) representing the distance of \( v \) from \( s \) in the subgraph consisting of the cloud and its adjacent vertices
• At each step
  – We add to the cloud the vertex \( u \) outside the cloud with the smallest distance label, \( d(u) \)
  – We update the labels of the vertices adjacent to \( u \)
Edge Relaxation

• Consider an edge $e = (u, z)$ such that
  - $u$ is the vertex most recently added to the cloud
  - $z$ is not in the cloud

• The relaxation of edge $e$ updates distance $d(z)$ as follows:
  
  $$d(z) \leftarrow \min\{d(z), d(u) + \text{weight}(e)\}$$

![Diagram showing edge relaxation process with updated distances]
Example
Example (cont.)
Dijkstra’s Algorithm

- A priority queue stores the vertices outside the cloud
  - Key: distance
  - Element: vertex
- Locator-based methods
  - $\text{insert}(k,e)$ returns a locator
  - $\text{replaceKey}(l,k)$ changes the key of an item
- We store two labels with each vertex:
  - Distance ($d(v)$ label)
  - locator in priority queue

**Algorithm DijkstraDistances($G, s$)**

$Q \leftarrow \text{new heap-based priority queue}$

for all $v \in G\text{.vertices()}$

- if $v = s$
  - $\text{setDistance}(v, 0)$
- else
  - $\text{setDistance}(v, \infty)$

$l \leftarrow Q\text{.insert(getDistance}(v), v)$

$\text{setLocator}(v,l)$

while $\neg Q\text{.isEmpty()}$

$u \leftarrow Q\text{.removeMin()}$

for all $e \in G\text{.incidentEdges}(u)$

  \{ relax edge $e$ \}

$z \leftarrow G\text{.opposite}(u,e)$

$r \leftarrow \text{getDistance}(u) + \text{weight}(e)$

if $r < \text{getDistance}(z)$

  $\text{setDistance}(z,r)$

$Q\text{.replaceKey(getLocator}(z),r)$
Analysis of Dijkstra’s Algorithm

• Graph operations
  – Method incidentEdges is called once for each vertex

• Label operations
  – We set/get the distance and locator labels of vertex $z$ $O(\deg(z))$ times
  – Setting/getting a label takes $O(1)$ time

• Priority queue operations
  – Each vertex is inserted once into and removed once from the priority queue, where each insertion or removal takes $O(\log n)$ time
  – The key of a vertex in the priority queue is modified at most $\deg(w)$ times, where each key change takes $O(\log n)$ time

• Dijkstra’s algorithm runs in $O((n + m) \log n)$ time provided the graph is represented by the adjacency list structure
  – Recall that $\sum_v \deg(v) = 2m$

• The running time can also be expressed as $O(m \log n)$ since the graph is connected

Phạm Bảo Sơn - DSA
Shortest Paths Tree

- Using the template method pattern, we can extend Dijkstra’s algorithm to return a tree of shortest paths from the start vertex to all other vertices.
- We store with each vertex a third label:
  - parent edge in the shortest path tree
- In the edge relaxation step, we update the parent label.

Algorithm \textit{DijkstraShortestPathsTree}(G, s):

\begin{verbatim}
... 

for all \( v \in G.\text{vertices}() \)
... 
setParent(v, \emptyset)
...

for all \( e \in G.\text{incidentEdges}(u) \)
\{ relax edge \( e \) \}
\( z \leftarrow G.\text{opposite}(u,e) \)
\( r \leftarrow \text{getDistance}(u) + \text{weight}(e) \)
if \( r < \text{getDistance}(z) \)
  setDistance\( (z,r) \)
  setParent\( (z,e) \)
  \( Q.\text{replaceKey}(\text{getLocator}(z),r) \)
\end{verbatim}
Why Dijkstra’s Algorithm Works

- Dijkstra’s algorithm is based on the greedy method. It adds vertices by increasing distance.
  - Suppose it didn’t find all shortest distances. Let F be the first wrong vertex the algorithm processed.
  - When the previous node, D, on the true shortest path was considered, its distance was correct.
  - But the edge (D,F) was relaxed at that time!
  - Thus, so long as d(F) ≥ d(D), F’s distance cannot be wrong. That is, there is no wrong vertex.
Why It Doesn’t Work for Negative-Weight Edges

Dijkstra’s algorithm is based on the greedy method. It adds vertices by increasing distance.

- If a node with a negative incident edge were to be added late to the cloud, it could mess up distances for vertices already in the cloud.

C’s true distance is 1, but it is already in the cloud with \( d(C)=5 \)!
Bellman-Ford Algorithm

- Works even with negative-weight edges
- Must assume directed edges (for otherwise we would have negative-weight cycles)
- Iteration i finds all shortest paths that use i edges.
- Running time: O(nm).
- Can be extended to detect a negative-weight cycle if it exists
  - How?

**Algorithm BellmanFord(G, s)**

```plaintext
for all v ∈ G.vertices()
    if v = s
        setDistance(v, 0)
    else
        setDistance(v, ∞)
for i ← 1 to n-1 do
    for each e ∈ G.edges()
        { relax edge e }
        u ← G.origin(e)
        z ← G.opposite(u,e)
        r ← getDistance(u) + weight(e)
        if r < getDistance(z)
            setDistance(z,r)
```
Bellman-Ford Example

Nodes are labeled with their $d(v)$ values.
DAG-based Algorithm

- Works even with negative-weight edges
- Uses topological order
- Doesn’t use any fancy data structures
- Is much faster than Dijkstra’s algorithm
- Running time: $O(n+m)$.

Algorithm $\text{DagDistances}(G, s)$

for all $v \in G.\text{vertices}()$
  if $v = s$
    $\text{setDistance}(v, 0)$
  else
    $\text{setDistance}(v, \infty)$

Perform a topological sort of the vertices

for $u \leftarrow 1$ to $n$ do \{in topological order\}
  for each $e \in G.\text{outEdges}(u)$
    \{ relax edge $e$ \}
    $z \leftarrow G.\text{opposite}(u,e)$
    $r \leftarrow \text{getDistance}(u) + \text{weight}(e)$
    if $r < \text{getDistance}(z)$
      $\text{setDistance}(z, r)$
DAG Example

Nodes are labeled with their $d(v)$ values

(two steps)
Minimum Spanning Trees
Minimum Spanning Trees

Spanning subgraph
- Subgraph of a graph $G$ containing all the vertices of $G$

Spanning tree
- Spanning subgraph that is itself a (free) tree

Minimum spanning tree (MST)
- Spanning tree of a weighted graph with minimum total edge weight

• Applications
  - Communications networks
  - Transportation networks
Cycle Property:

Let $T$ be a minimum spanning tree of a weighted graph $G$

Let $e$ be an edge of $G$ that is not in $T$ and let $C$ be the cycle formed by $e$ with $T$

For every edge $f$ of $C$, $\text{weight}(f) \leq \text{weight}(e)$

Proof:

By contradiction

If $\text{weight}(f) > \text{weight}(e)$ we can get a spanning tree of smaller weight by replacing $e$ with $f$
Partition Property:
- Consider a partition of the vertices of $G$ into subsets $U$ and $V$
- Let $e$ be an edge of minimum weight across the partition
- There is a minimum spanning tree of $G$ containing edge $e$

Proof:
- Let $T$ be an MST of $G$
- If $T$ does not contain $e$, consider the cycle $C$ formed by $e$ with $T$ and let $f$ be an edge of $C$ across the partition
- By the cycle property,
  \[ \text{weight}(f) \leq \text{weight}(e) \]
- Thus, $\text{weight}(f) = \text{weight}(e)$
- We obtain another MST by replacing $f$ with $e$

Replacing $f$ with $e$ yields another MST.
Kruskal’s Algorithm

- A priority queue stores the edges outside the cloud
  - Key: weight
  - Element: edge
- At the end of the algorithm
  - We are left with one cloud that encompasses the MST
  - A tree \( T \) which is our MST

**Algorithm KrusalMST(\( G \))**

```plaintext
for each vertex \( V \) in \( G \) do
    define a Cloud(\( v \)) of \( \{ v \} \)
let \( Q \) be a priority queue.
Insert all edges into \( Q \) using their weights as the key
\( T \leftarrow \emptyset \)
while \( T \) has fewer than \( n-1 \) edges do
    edge \( e = T\).removeMin() 
    Let \( u, v \) be the endpoints of \( e \)
    if Cloud(\( v \)) \( \neq \) Cloud(\( u \)) then
        Add edge \( e \) to \( T \)
        Merge Cloud(\( v \)) and Cloud(\( u \))
return \( T \)
```
Data Structure for Kruskal Algorithm

- The algorithm maintains a forest of trees
- An edge is accepted if it connects distinct trees
- We need a data structure that maintains a partition, i.e., a collection of disjoint sets, with the operations:
  - find(u): return the set storing u
  - union(u,v): replace the sets storing u and v with their union
Representation of a Partition

- Each set is stored in a sequence
- Each element has a reference back to the set
  - operation \texttt{find}(u) takes O(1) time, and returns the set of which u is a member.
  - in operation \texttt{union}(u,v), we move the elements of the smaller set to the sequence of the larger set and update their references
  - the time for operation \texttt{union}(u,v) is \( \text{min}(n_u,n_v) \), where \( n_u \) and \( n_v \) are the sizes of the sets storing u and v
- Whenever an element is processed, it goes into a set of size at least double, hence each element is processed at most \( \log n \) times
Partition-Based Implementation

• A partition-based version of Kruskal’s Algorithm performs cloud merges as unions and tests as finds.

Algorithm **Kruskal**(*G*):

**Input**: A weighted graph *G*.

**Output**: An MST *T* for *G*.

Let *P* be a partition of the vertices of *G*, where each vertex forms a separate set.

Let *Q* be a priority queue storing the edges of *G*, sorted by their weights.

Let *T* be an initially-empty tree.

while *Q* is not empty do

(\(u,v\)) ← *Q*.removeMinElement()

if *P*.find(*u*) ≠ *P*.find(*v*) then

Add (\(u,v\)) to *T*

*T*.union(*u*, *v*)

return *T*

Running time: \(O((n + m)\log n)\)
Kruskal Example

The image cannot be displayed. Your computer may not have enough memory to open the image or the image may have been corrupted. Restart your computer, and then open the file again. If the red x still appears, you may have to delete the image and then insert it again.
Example
Example
Example

Phạm Bảo Sơn - DSA
Example
Example
Example
Example
Example

- JFK
- BOS
- MIA
- ORD
- LAX
- DFW
- SFO
- BWI
- PVD

Flights:
- JFK to BOS: 867
- BOS to PVD: 849
- PVD to JFK: 187
- JFK to ORD: 740
- ORD to JFK: 621
- JFK to LAX: 802
- LAX to JFK: 1464
- JFK to SFO: 337
- SFO to JFK: 1464
- JFK to ORD: 1846
- ORD to JFK: 1391
- JFK to DFW: 184
- DFW to JFK: 946
- JFK to SFO: 1258
- SFO to JFK: 1121
- JFK to MIA: 2342
- MIA to JFK: 2204
- MIA to DFW: 184
- DFW to MIA: 1121
- DFW to SFO: 802
- SFO to DFW: 1464
- DFW to JFK: 1846
- JFK to ORD: 184
- ORD to JFK: 1391
- JFK to LAX: 337
- LAX to JFK: 1464
- JFK to SFO: 1258
- SFO to JFK: 1121
- JFK to MIA: 2342
- MIA to JFK: 2204
- MIA to DFW: 184
- DFW to MIA: 1121
- DFW to SFO: 802
- SFO to DFW: 1464
- DFW to JFK: 1846
- JFK to ORD: 184
- ORD to JFK: 1391
- JFK to LAX: 337
- LAX to JFK: 1464
- JFK to SFO: 1258
- SFO to JFK: 1121
- JFK to MIA: 2342
- MIA to JFK: 2204
- MIA to DFW: 184
- DFW to MIA: 1121
- DFW to SFO: 802
- SFO to DFW: 1464
- DFW to JFK: 1846
- JFK to ORD: 184
- ORD to JFK: 1391
- JFK to LAX: 337
- LAX to JFK: 1464
- JFK to SFO: 1258
- SFO to JFK: 1121
- JFK to MIA: 2342
- MIA to JFK: 2204
- MIA to DFW: 184
- DFW to MIA: 1121
- DFW to SFO: 802
- SFO to DFW: 1464
- DFW to JFK: 1846
Example

- JFK
- BOS
- PVD
- ORD
- MIA
- DFW
- SFO
- LAX
- BWI

Distances:
- JFK to BOS: 867
- JFK to PVD: 849
- JFK to ORD: 740
- JFK to MIA: 1258
- ORD to SFO: 802
- ORD to DFW: 621
- ORD to JFK: 184
- ORD to BWI: 187
- DFW to SFO: 1464
- ORD to MIA: 1391
- ORD to BWI: 1121
- SFO to DFW: 1235
- SFO to LAX: 1464
- SFO to ORD: 802
- LAX to DFW: 1235
- LAX to SFO: 337
- DFW to JFK: 846
- DFW to MIA: 2342
- MIA to BWI: 1090
- BWI to SFO: 946
- BWI to DFW: 946
Example

Airport Codes:
- JFK
- BOS
- PVD
- ORD
- MIA
- DFW
- SFO
- LAX
- BWI

Flight Distances (miles):
- JFK to BOS: 867
- BOS to PVD: 849
- PVD to JFK: 187
- JFK to ORD: 740
- ORD to JFK: 621
- ORD to DFW: 802
- DFW to ORD: 1391
- ORD to SFO: 2342
- SFO to ORD: 1121
- ORD to MIA: 1258
- MIA to ORD: 1090
- MIA to SFO: 1846
- SFO to MIA: 946
- MIA to DFW: 1235
- DFW to MIA: 1391
- MIA to LAX: 2342
- LAX to MIA: 1464
- MIA to LAX: 1235
- LAX to SFO: 1846
- SFO to LAX: 337
- LAX to SFO: 1464
Example
Example

Phạm Bảo Sơn - DSA
The image cannot be displayed. Your computer may not have enough memory to open the image, or the image may have been corrupted. Restart your computer, and then open the file again. If the red x still appears, you may have to delete the image and then insert it again.
Prim-Jarnik’ s Algorithm

• Similar to Dijkstra’ s algorithm (for a connected graph)
• We pick an arbitrary vertex \( s \) and we grow the MST as a cloud of vertices, starting from \( s \)
• We store with each vertex \( v \) a label \( d(v) \) = the smallest weight of an edge connecting \( v \) to a vertex in the cloud

шение each step:

- We add to the cloud the vertex \( u \) outside the cloud with the smallest distance label
- We update the labels of the vertices adjacent to \( u \)
Prim-Jarnik’s Algorithm (cont.)

- A priority queue stores the vertices outside the cloud
  - Key: distance
  - Element: vertex
- Locator-based methods
  - \( \text{insert}(k,e) \) returns a locator
  - \( \text{replaceKey}(l,k) \) changes the key of an item
- We store three labels with each vertex:
  - Distance
  - Parent edge in MST
  - Locator in priority queue

Algorithm \( \text{PrimJarnikMST}(G) \)
\[
Q \leftarrow \text{new heap-based priority queue} \\
s \leftarrow \text{a vertex of } G \\
\text{for all } v \in G.\text{vertices()}
\]
- \( v = s \)
  - \( \text{setDistance}(v, 0) \)
- else
  - \( \text{setDistance}(v, \infty) \)
  - \( \text{setParent}(v, \emptyset) \)
  - \( l \leftarrow Q.\text{insert}(\text{getDistance}(v), v) \)
  - \( \text{setLocator}(v,l) \)
while \( \neg Q.\text{isEmpty()} \)
\[
u \leftarrow Q.\text{removeMin()} \\
\text{for all } e \in G.\text{incidentEdges}(u) \\
\]
- \( z \leftarrow G.\text{opposite}(u,e) \)
- \( r \leftarrow \text{weight}(e) \)
  - if \( r < \text{getDistance}(z) \)
    - \( \text{setDistance}(z,r) \)
    - \( \text{setParent}(z,e) \)
    - \( Q.\text{replaceKey}(\text{getLocator}(z),r) \)
Example
Example (contd.)
Analysis

- Graph operations
  - Method incidentEdges is called once for each vertex
- Label operations
  - We set/get the distance, parent and locator labels of vertex \( z \) \( O(\deg(z)) \) times
  - Setting/getting a label takes \( O(1) \) time
- Priority queue operations
  - Each vertex is inserted once into and removed once from the priority queue, where each insertion or removal takes \( O(\log n) \) time
  - The key of a vertex \( w \) in the priority queue is modified at most \( \deg(w) \) times, where each key change takes \( O(\log n) \) time
- Prim-Jarnik’s algorithm runs in \( O((n + m) \log n) \) time provided the graph is represented by the adjacency list structure
  - Recall that \( \sum_v \deg(v) = 2m \)
- The running time is \( O(m \log n) \) since the graph is connected
Baruvka’s Algorithm

- Like Kruskal’s Algorithm, Baruvka’s algorithm grows many “clouds” at once.

Algorithm $\text{BaruvkaMST}(G)$

\[
\begin{align*}
T & \leftarrow V \quad \{\text{just the vertices of } G\} \\
\text{while } T \text{ has fewer than } n-1 \text{ edges do} \\
\text{for each connected component } C \text{ in } T \text{ do} \\
& \quad \text{Let edge } e \text{ be the smallest-weight edge from } C \text{ to another component in } T. \\
& \quad \text{if } e \text{ is not already in } T \text{ then} \\
& \quad \quad \text{Add edge } e \text{ to } T \\
\text{return } T
\end{align*}
\]

- Each iteration of the while-loop halves the number of connected components in $T$.
  - The running time is $O(m \log n)$.
Baruvka Example