Graphs

Graphs
Outline and Reading

Graphs (§12.1)
- Definition
- Applications
- Terminology
- Properties
- ADT

Data structures for graphs (§12.2)
- Edge list structure
- Adjacency list structure
- Adjacency matrix structure
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
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
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
Terminology (cont.)

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

Property 1

\[ \sum_v \deg(v) = 2m \]

Proof: each edge is counted twice

Property 2

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

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

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

Example

- \( n = 4 \)
- \( m = 6 \)
- \( \deg(v) = 3 \)

What is the bound for a directed graph?

Notation

| \( n \) | number of vertices |
| \( m \) | number of edges |
| \( \deg(v) \) | degree of vertex \( v \) |
Main Methods of the Graph ADT

Vertices and edges
- are positions
- store elements

Accessor methods
- aVertex()
- incidentEdges(v)
- endVertices(e)
- isDirected(e)
- origin(e)
- destination(e)
- opposite(v, e)
- areAdjacent(v, w)

Update methods
- insertVertex(o)
- insertEdge(v, w, o)
- insertDirectedEdge(v, w, o)
- removeVertex(v)
- removeEdge(e)

Generic methods
- numVertices()
- numEdges()
- vertices()
- edges()
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
**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

![Graph with adjacency matrix](image-url)

- **Vertices:** u, v, w
- **Edges:** u-v, v-w
- **Adjacency matrix:**
  - u: [0, 1, 0]
  - v: [0, 0, 1]
  - w: [0, 0, 0]
## Asymptotic Performance

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Edge List</th>
<th>Adjacency List</th>
<th>Adjacency Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space</td>
<td>$n + m$</td>
<td>$n + m$</td>
<td>$n^2$</td>
</tr>
<tr>
<td>incidentEdges($v$)</td>
<td>$m$</td>
<td>$\text{deg}(v)$</td>
<td>$n$</td>
</tr>
<tr>
<td>areAdjacent ($v$, $w$)</td>
<td>$m$</td>
<td>$\min(\text{deg}(v), \text{deg}(w))$</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>$\text{deg}(v)$</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
Outline and Reading

Definitions (§12.1)
- Subgraph
- Connectivity
- Spanning trees and forests

Depth-first search (§12.3.1)
- Algorithm
- Example
- Properties
- Analysis

Applications of DFS
- Path finding
- Cycle finding
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 $G$. 

Spanning subgraph
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 $G$.

Connected graph

Non connected graph with two connected components
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.
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()

setLabel($u$, UNEXPLORED)

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

setLabel($e$, UNEXPLORED)

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

if $.getLabel(v)$ = UNEXPLORED

$DFS(G, v)$

else

setLabel($e$, BACK)

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

setLabel($v$, VISITED)

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

if $getLabel(e) = UNEXPLORED$

$w \leftarrow opposite(v,e)$

if $getLabel(w) = UNEXPLORED$

setLabel($e$, DISCOVERY)

$DFS(G, w)$

else

setLabel($e$, BACK)
Example

unexplored vertex
visited vertex
unexplored edge
discovery edge
back edge
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

\( \text{DFS}(G, v) \) visits all the vertices and edges in the connected component of \( v \)

Property 2

The discovery edges labeled by \( \text{DFS}(G, v) \) 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 $\sum_v \deg(v) = 2m$
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 \( pathDFS(G, v, z) \)

- setLabel\((v, VISITED)\)
- \( S.push(v) \)
- if \( v = z \)
  - return \( S.elements() \)
- for all \( e \in G.incidentEdges(v) \)
  - if getLabel\((e) = UNEXPLORED\)
    - \( w \leftarrow opposite(v,e) \)
    - if getLabel\((w) = UNEXPLORED\)
      - setLabel\((e, DISCOVERY)\)
      - \( S.push(e) \)
      - \( pathDFS(G, w, z) \)
      - \( S.pop(e) \)
    - else
      - setLabel\((e, BACK)\)
- \( S.pop(v) \)
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 $cycleDFS(G, v, z)$

- $setLabel(v, VISITED)$
- $S.push(v)$
- 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 $T.elements()$
- $S.pop(v)$
Breadth-First Search
Outline and Reading

Breadth-first search (§12.3.2)
- Algorithm
- Example
- Properties
- Analysis
- Applications

DFS vs. BFS
- Comparison of applications
- Comparison of edge labels
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 \textit{BFS}(G)

\begin{itemize}
    \item \textbf{Input} graph \textit{G}
    \item \textbf{Output} labeling of the edges and partition of the vertices of \textit{G}
\end{itemize}

\begin{algorithmic}
    \FORALL{\textit{u} \in \textit{G.vertices}}()
        \STATE \textit{setLabel}(\textit{u}, \text{UNEXPLORED})
    \ENDFOR
    \FORALL{\textit{e} \in \textit{G.edges}}()
        \STATE \textit{setLabel}(\textit{e}, \text{UNEXPLORED})
    \ENDFOR
    \FORALL{\textit{v} \in \textit{G.vertices}}()
        \IF{\textit{getLabel}(\textit{v}) = \text{UNEXPLORED}}
            \STATE \textbf{BFS}(\textit{G, v})
        \ENDIF
    \ENDFOR
\end{algorithmic}

Algorithm \textit{BFS}(G, s)

\begin{itemize}
    \item \textit{L}_0 \leftarrow \text{new empty sequence}
    \item \textit{L}_0\textit{.insertLast}(s)
    \item \textit{setLabel}(s, VISITED)
    \item \textit{i} \leftarrow 0
    \WHILE{\neg \textit{L}_i\textit{.isEmpty}}()
        \STATE \textit{L}_{i+1} \leftarrow \text{new empty sequence}
        \FORALL{\textit{v} \in \textit{L}_i\textit{.elements}()}
            \FORALL{\textit{e} \in \textit{G.incidentEdges}(\textit{v})}
                \IF{\textit{getLabel}(\textit{e}) = \text{UNEXPLORED}}
                    \STATE \textit{w} \leftarrow \text{opposite}(\textit{v,e})
                    \IF{\textit{getLabel}(\textit{w}) = \text{UNEXPLORED}}
                        \STATE \textit{setLabel}(\textit{e}, DISCOVERY)
                        \STATE \textit{setLabel}(\textit{w}, VISITED)
                        \STATE \textit{L}_{i+1}\textit{.insertLast}(\textit{w})
                    \ELSE
                        \STATE \textit{setLabel}(\textit{e}, CROSS)
                    \ENDIF
                \ENDIF
                \STATE \textit{i} \leftarrow \textit{i} +1
            \ENDFOR
        \ENDFOR
    \ENDWHILE
\end{itemize}
Example

- A visited vertex
- unexplored vertex
- unexplored edge
- discovery edge
- cross edge
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 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
Directed Graphs
Outline and Reading (§12.4)

- Reachability (§12.4.1)
  - Directed DFS
  - Strong connectivity

- Transitive closure (§12.4.2)
  - The Floyd-Warshall Algorithm

- Directed Acyclic Graphs (DAG’s) (§12.4.3)
  - Topological Sorting
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.
Directed DFS

We can specialize the traversal algorithms (DFS and BFS) to digraphs by traversing edges only along their direction.

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

Graphs
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: $O(n+m)$. 
Graphs

**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

Diagram:

- Vertices: a, b, c, d, e, f, g
- Edges:
  - a to c
  - c to g
  - g to b
  - b to e
  - e to d
  - d to f
- Strongly connected components:
  - {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-1

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

$i \leftarrow 1$

for all $v \in G$.vertices()
    denote $v$ as $v_i$
    $i \leftarrow i + 1$
    $G_0 \leftarrow G$

for $k \leftarrow 1$ to $n$
    $G_k \leftarrow G_{k-1}$
    for $i \leftarrow 1$ to $n$ ($i \neq k$)
        for $j \leftarrow 1$ to $n$ ($j \neq i, k$)
            if $G_{k-1}$.areAdjacent($v_i, v_k$) \land $G_{k-1}$.areAdjacent($v_k, v_j$)
                if $\neg G_k$.areAdjacent($v_i, v_j$)
                    $G_k$.insertDirectedEdge($v_i, v_j, k$)

return $G_n$
Floyd-Warshall Example
Floyd-Warshall, Iteration 1

Graphs 57
Floyd-Warshall, Iteration 2
Floyd-Warshall, Iteration 3

Graphs 59

SFO → ORD → JFK → BOS
SFO → DFW → JFK → BOS
SFO → LAX → JFK → BOS
SFO → MIA → JFK → BOS
SFO → ORD → ORD → JFK → BOS
SFO → DFW → DFW → JFK → BOS
SFO → LAX → LAX → JFK → BOS
SFO → MIA → MIA → JFK → BOS
SFO → ORD → ORD → ORD → JFK → BOS
SFO → DFW → DFW → DFW → JFK → BOS
SFO → LAX → LAX → LAX → JFK → BOS
SFO → MIA → MIA → MIA → JFK → BOS
SFO → ORD → ORD → ORD → ORD → JFK → BOS
SFO → DFW → DFW → DFW → DFW → JFK → BOS
SFO → LAX → LAX → LAX → LAX → JFK → BOS
SFO → MIA → MIA → MIA → MIA → JFK → BOS

Floyd-Warshall, Iteration 4
Floyd-Warshall, Iteration 5
Floyd-Warshall, Iteration 6
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 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. Go out w/ friends
10. Sleep
11. Dream about graphs
Algorithm for Topological Sorting

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

Running time: \( O(n + m) \). Why?
Topological Sorting
Algorithm using DFS

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 $e \in G.incidentEdges(v)$
if `.getLabel(e) = UNEXPLORED`
    $w \leftarrow \text{opposite}(v,e)$
    if `.getLabel(w) = UNEXPLORED`
        `setLabel(e, DISCOVERY)`
        `topologicalDFS(G, w)`
    else
        \{ $e$ is a forward or cross edge \}
Label $v$ with topological number $n$
$n \leftarrow n - 1$

O($n+m$) time
Topological Sorting Example
Topological Sorting Example
Topological Sorting Example
Topological Sorting Example
Topological Sorting Example

Graphs
Topological Sorting Example
Topological Sorting Example
Topological Sorting Example
Topological Sorting Example
Topological Sorting Example
Shortest Paths
Outline and Reading

- Weighted graphs (§12.1)
  - Shortest path problem
  - Shortest path properties
- Dijkstra’s algorithm (§12.6.1)
  - Algorithm
  - Edge relaxation
- The Bellman-Ford algorithm
- Shortest paths in DAGs
- All-pairs shortest paths
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 Path Problem

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).
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:

$$d(z) \leftarrow \min\{d(z), d(u) + \text{weight}(e)\}$$
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 $\text{DijkstraDistances}(G, s)$

```plaintext
Q ← new heap-based priority queue
for all $v \in G.\text{vertices}()$
  if $v = s$
    $\text{setDistance}(v, 0)$
  else
    $\text{setDistance}(v, \infty)$
  $l ← Q.\text{insert}(\text{getDistance}(v), v)$
  $\text{setLocator}(v, l)$
while $\neg Q.\text{isEmpty}()$
  $u ← Q.\text{removeMin}()$
  for all $e ∈ G.\text{incidentEdges}(u)$
    \{ relax edge $e$ \}
    $z ← G.\text{opposite}(u, e)$
    $r ← \text{getDistance}(u) + \text{weight}(e)$
    if $r < \text{getDistance}(z)$
      $\text{setDistance}(z, r)$
      $Q.\text{replaceKey}(\text{getLocator}(z), r)$
```
Analysis

- **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
Extension

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.vertices()
  ...
  setParent(v, \emptyset)
  ...

  for all \ e \in G.incidentEdges(u)
    { relax edge e }
    \ z \leftarrow G\text{.opposite}(u,e)
    \ r \leftarrow \text{getDistance}(u) + \text{weight}(e)
    \text{if} \ r < \text{getDistance}(\ z)
      \text{setDistance}(\ z,r)
      \text{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 $d(F) \geq d(D)$, 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 $\text{BellmanFord}(G, s)$

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

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

- Works even with negative-weight edges
- Uses topological order
- Uses simple data structures
- Is much faster than Dijkstra’s algorithm
- Running time: $O(n+m)$.

Algorithm \texttt{DagDistances}(G, s)

\begin{algorithmic}
\State \textbf{for all} $v \in G$.vertices()
\If{$v = s$}
\State \texttt{setDistance}(v, 0)
\Else
\State \texttt{setDistance}(v, $\infty$)
\EndIf
\EndFor
\State \textbf{Perform a topological sort of the vertices}
\For{$u \leftarrow 1$ \textbf{to} $n$ \textbf{do} \{in topological order\}}
\For\textbf{each} $e \in G$.outEdges($u$)
\{relax edge $e$\}
\State $z \leftarrow G$.opposite($u$, $e$)
\State $r \leftarrow \texttt{getDistance}(u) + \text{weight}(e)$
\If{$r < \texttt{getDistance}(z)$}
\State \texttt{setDistance}($z$, $r$)
\EndIf
\EndFor
\EndFor
\end{algorithmic}
DAG Example

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

Graphs (two steps)
All-Pairs Shortest Paths

- Find the distance between every pair of vertices in a weighted directed graph G.
- We can make n calls to Dijkstra’s algorithm (if no negative edges), which takes \( O(nm\log n) \) time.
- Likewise, n calls to Bellman-Ford would take \( O(n^2m) \) time.
- We can achieve \( O(n^3) \) time using dynamic programming (similar to the Floyd-Warshall algorithm).

Algorithm \( \text{AllPair}(G) \) \{assumes vertices 1,...,n\}

```plaintext
for all vertex pairs \((i,j)\)
    if \( i = j \)
        \( D_0[i,i] \leftarrow 0 \)
    else if \((i,j)\) is an edge in G
        \( D_0[i,j] \leftarrow \text{weight of edge } (i,j) \)
    else
        \( D_0[i,j] \leftarrow +\infty \)

for \( k \leftarrow 1 \) to \( n \) do
    for \( i \leftarrow 1 \) to \( n \) do
        for \( j \leftarrow 1 \) to \( n \) do
            \( D_k[i,j] \leftarrow \min\{D_{k-1}[i,j], D_{k-1}[i,k]+D_{k-1}[k,j]\} \)

return \( D_n \)
```

Uses only vertices numbered 1,...,\( k \)
(compute weight of this edge)

Uses only vertices numbered 1,...,\( k-1 \)
Minimum Spanning Trees
Outline and Reading

- Minimum Spanning Trees (§12.7)
  - Definitions
  - A crucial fact
- The Prim-Jarník Algorithm (§12.7.2)
- Kruskal's Algorithm (§12.7.1)
- Baruvka's Algorithm
Minimum Spanning Tree

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 $e$
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 $d(v) = \text{the smallest weight of an edge connecting } v \text{ to a vertex in the cloud}$

At 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
- insert\((k,e)\) returns a locator
- 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 PrimJarnikMST\((G)\)

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

---

**Algorithm** PrimJarnikMST\((G)\)

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

\[
s \leftarrow \text{a vertex of } G
\]

**for all** \( v \in G\).vertices() \n  **if** \( v = s \)
    \[
    \text{setDistance}(v, 0)
    \]
  **else**
    \[
    \text{setDistance}(v, \infty)
    \]
  \[
  \text{setParent}(v, \emptyset)
  \]
  \[
  l \leftarrow Q.insert(getDistance(v), v)
  \]
  \[
  \text{setLocator}(v, l)
  \]
  **while** \( \neg Q.isEmpty() \)
  \[
  u \leftarrow Q.removeMin()
  \]
  **for all** \( e \in G.incidentEdges(u) \)
  \[
  z \leftarrow G.opposite(u, e)
  \]
  \[
  r \leftarrow \text{weight}(e)
  \]
  **if** \( r < \text{getDistance}(z) \)
    \[
    \text{setDistance}(z, r)
    \]
    \[
    \text{setParent}(z, e)
    \]
    \[
    Q.replaceKey(getLocator(z), r)
    \]
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 $\text{KruskalMST}(G)$

for each vertex $V$ in $G$ do
  define a $\text{Cloud}(v)$ of $\leftarrow \{v\}$
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 $e = T.\text{removeMin}()$
  Let $u, v$ be the endpoints of $e$
  if $\text{Cloud}(v) \neq \text{Cloud}(u)$ then
    Add edge $e$ to $T$
    Merge $\text{Cloud}(v)$ and $\text{Cloud}(u)$
return $T$
Kruskal Example

Graphs
Example
Example
Example
Example

Graphs 113
Example
Example
Example

Graphs 116

BOS
MIA
ORD
LAX
DFW
SFO
BWI
PVD
JFK

867
2704
187
849
144
1258
1235
1090
946
337
1464
1846
1391
2342
1121
2704
621
740
802

116
Example

Graphs

BOS

MIA

ORD

LAX

DFW

SFO

BWI

PVD

JFK

867

849

187

144

1391

621

740

802

184

1258

1391

1847

1235

2342

2704

1464

1235

1090

2342

802

1464

1235

337
Example

Graphs 120
Example
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:
  - \textbf{find}(u): return the set storing u
  - \textbf{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 find(u) takes $O(1)$ time, and returns the set of which $u$ is a member.
  - in operation 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 union($u,v$) is $\min(n_u,n_v)$, 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
    P.union(u,v)

return T

Running time: O((n+m)log n)
Baruvka’s Algorithm

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

Algorithm BaruvkaMST(G)

\[ T \leftarrow V \{ \text{just the vertices of } G \} \]

while \( T \) has fewer than \( n-1 \) edges do

for each connected component \( C \) in \( T \) do

Let edge \( e \) be the smallest-weight edge from \( C \) to another component in \( T \).

if \( e \) is not already in \( T \) then

Add edge \( e \) to \( T \)

end if

end for

return \( T \)

Each iteration of the while-loop halves the number of connected components in \( T \).

- The running time is \( O(m \log n) \).
Baruvka Example
Example

Graphs
Traveling Salesperson Problem

- A tour of a graph is a spanning cycle (e.g., a cycle that goes through all the vertices)
- A traveling salesperson tour of a weighted graph is a tour that is simple (i.e., no repeated vertices or edges) and has minimum weight
- No polynomial-time algorithms are known for computing traveling salesperson tours
- The traveling salesperson problem (TSP) is a major open problem in computer science
  - Find a polynomial-time algorithm computing a traveling salesperson tour or prove that none exists

Example of traveling salesperson tour (with weight 17)