326.041 (2015S) – Practical Software Technology

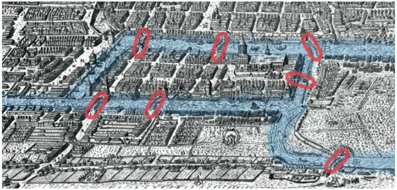
(Praktische Softwaretechnologie)

Graphs, Weighted Graphs, Shortest Path

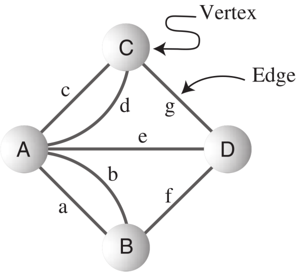
Alexander Baumgartner
Alexander.Baumgartner@risc.jku.at

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria

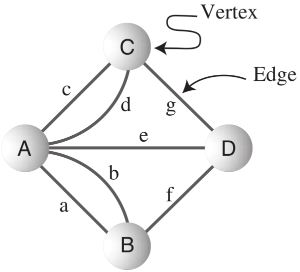
The 7 bridges of Königsberg I



Is there a way to walk across all bridges without recrossing any of them?
Leonhard Euler solved the problem in 1735 by transforming it into a graph.



The 7 bridges of Königsberg II

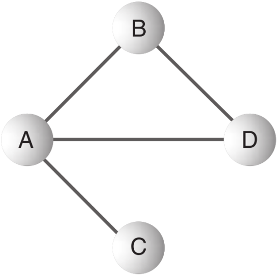


Euler observed that:

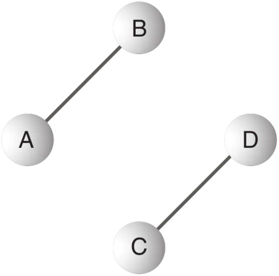
- During any walk in the graph, the number of times one enters a non-terminal vertex equals the number of times one leaves it.
- It follows that, for each land mass (except for the ones chosen for the start and finish), the number of bridges touching that land mass must be even.

Graph – Basic Notions I

- A **graph** $G = (V, E)$ is a set V of **vertices** and a collection E of pairs of vertices from V, called **edges**.
- A way of representing connections or relationships between pairs of objects from some set V.
- A **path** is a sequence of edges.
- A graph is said to be **connected** if there is at least one path from every vertex to every other vertex.



a) Connected Graph



b) Non-connected Graph

Graphs – Basic Notions II

- Edges in a graph $G = (V, E)$ are either **directed or undirected.**
  - A directed edge from u to v is an ordered pair (u, v) with $u, v \in V$.
  - An undirected edge between u and v is a set $\{u, v\}$ with $u, v \in V$.
- In **weighted** graphs, edges are given a weight number. E.g.:
  - The physical distance between two vertices.
  - The time it takes to get from one vertex to another.
  - How much it costs to travel from vertex to vertex.

  Formally it is modeled by a weight function $w : E \to \mathbb{R}$.

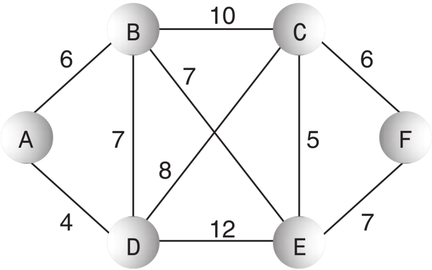
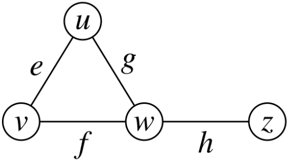
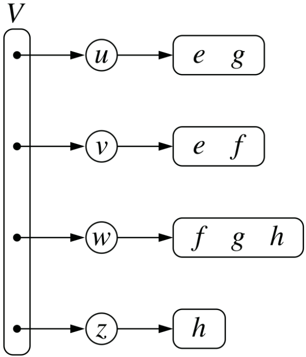


Figure: An undirected weighted graph.

Data Structures for Graphs I

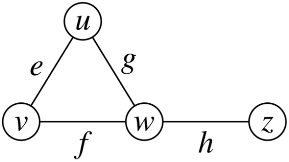
- An **edge list** is an unordered list of all edges.
- In an **adjacency list,** we additionally maintain, for each vertex, a separate list containing those edges that are incident to the vertex.

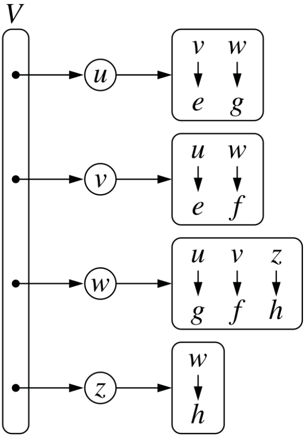




Data Structures for Graphs II

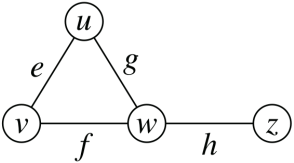
- An **adjacency map** is similar to an adjacency list, with the adjacent vertices serving as the keys.





Data Structures for Graphs III

- An **adjacency matrix** maintains an $n \times n$ matrix, for a graph with n vertices.



		0	1	2	3
u →	0		e	g	
v →	1	e		f	
w →	2	g	f		h
z →	3			h	

Generic Adjacency Map Graph I

```
public class AdjacencyMapGraph<V, E> {
    private boolean directed;
    private List<Vertex> vertices = new LinkedList<>();
    private List<Edge> edges = new LinkedList<>();
    ...
    public class Edge {
        private E elem; // Weight, name, ...
        private Vertex u, v;
    ...
    public class Vertex {
        private V elem;
        private Map<Vertex, Edge> outgoing, incoming;
        public Vertex(V elem) {
            outgoing = new HashMap<>();
            if (AdjacencyMapGraph.this.directed)
                incoming = new HashMap<>();
            else
                incoming = outgoing;
            this.elem = elem; }
    ...
```


Alexander.Baumgartner@risc.jku.at

Generic Adjacency Map Graph II

```
public class AdjacencyMapGraph<V, E> {
    ...
    public Edge getEdge(Vertex u, Vertex v) {
        return u.getOutgoing().get(v); }
    public Vertex insertVertex(V elem) {
        Vertex v = new Vertex(elem);
        vertices.add(v);
        return v; }
    public Edge insertEdge(Vertex u, Vertex v, E elem) {
        assert getEdge(u, v) == null;  // Already exists
        Edge e = new Edge(u, v, elem);
        edges.add(e);
        u.getOutgoing().put(v, e);
        v.getIncoming().put(u, e);
        return e;
    }
    ...
```

Graph Traversal – Motivation

- Visits all the vertices and edges in time proportional to their number (linear time).
- E.g., given an undirected graph G, traversal is needed to compute:
  - A path from one vertex u to another vertex v.
  - The minimal paths from a given a vertex v to all the other vertices.
  - Whether G is connected.
  - A spanning tree of G, if G is connected.
  - The connected components of G.
  - Identifying cycles in G.
- E.g., given a directed graph G, traversal is needed to compute:
  - A directed path from one vertex u to another vertex v.
  - All the vertices of G that are reachable from a given vertex v.
  - Whether G is acyclic.
  - Whether G is strongly connected.
  - The strongly connected components of G.

Graph Traversal – Depth-First Search

Depth-first search traversal starting at a vertex v:

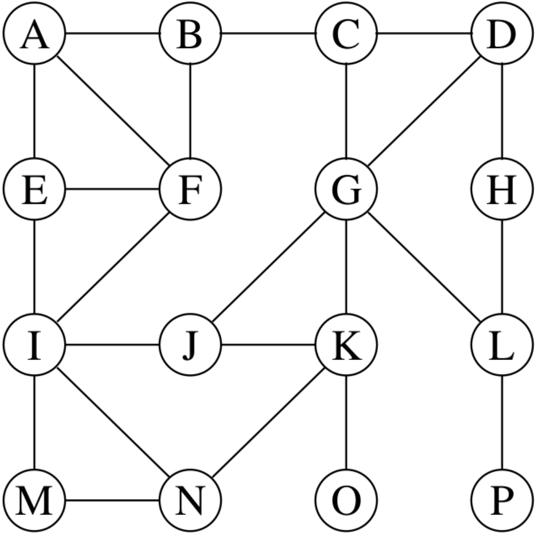
Input: A graph G and a vertex v of G.
Output: All vertices reachable from v, with their discovery edges.

Algorithm $DFS(G, v)$:

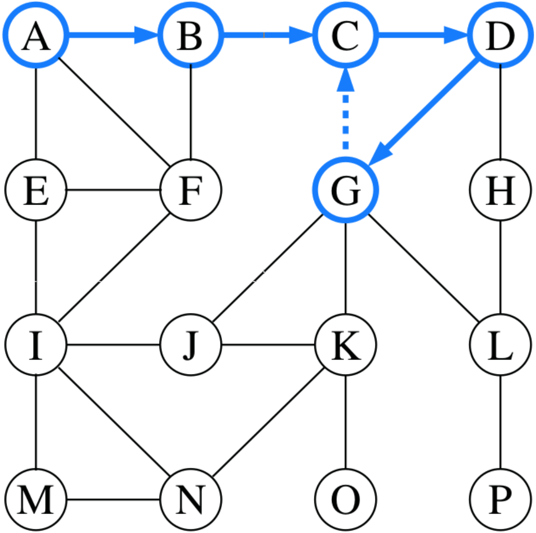
- Mark vertex v as visited.
- for each of v's outgoing edges, $e = (v, u)$ do
-     if vertex u has not been visited then
-         Record edge e as the discovery edge for vertex u.
-         Recursively call $DFS(G, u)$.

Traversal yields a depth-first search tree rooted at a starting vertex v.

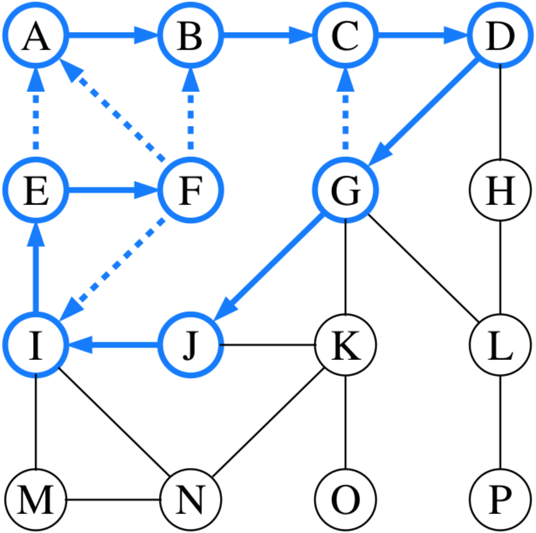
Graph Traversal – Depth-First Example



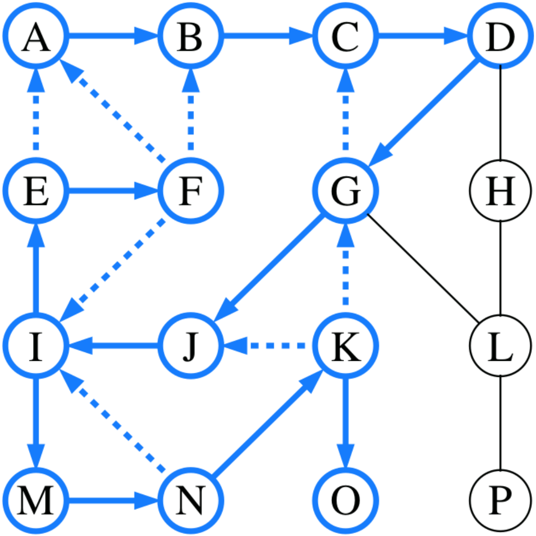
Graph Traversal – Depth-First Example



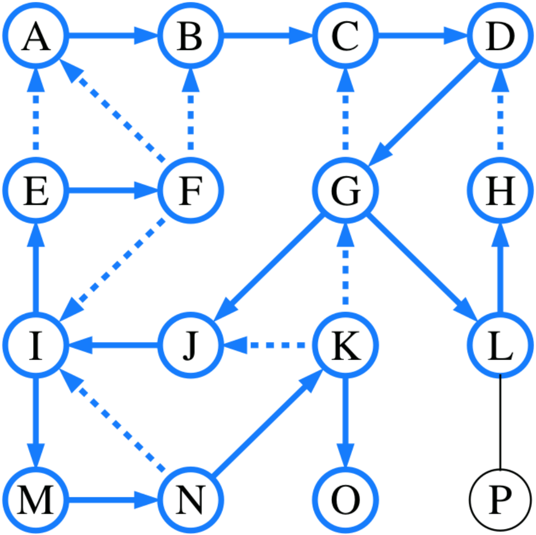
Graph Traversal – Depth-First Example



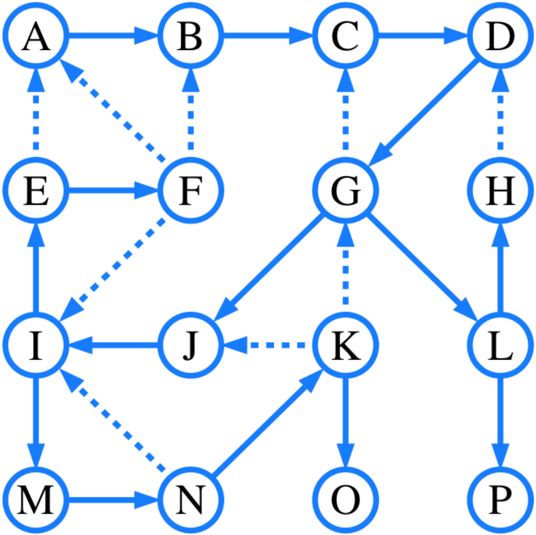
Graph Traversal – Depth-First Example



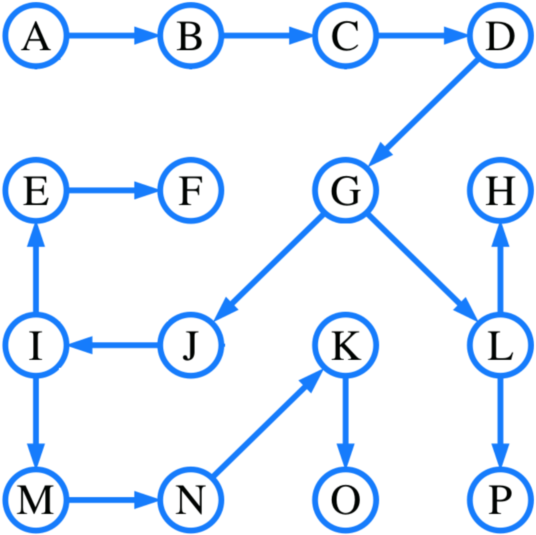
Graph Traversal – Depth-First Example



Graph Traversal – Depth-First Example



Graph Traversal – Depth-First Example



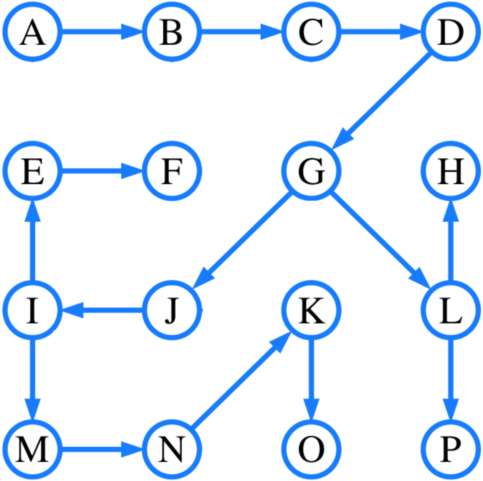
Depth-First Graph Traversal in Java

- Given a start vertex v of a graph G.
- Store all reachable vertices in $Set{<}Vertex{>}$.
- For each vertex, store discovery edge in $Map{<}Vertex,\ Edge{>}$.

```
public class AdjacencyMapGraph<V, E> {
    ...
    public void depthFirst(Vertex v, Set<Vertex> known,
                           Map<Vertex, Edge> forest) {
        known.add(v);
        for (Edge e : v.getOutgoing().values()) {
            Vertex u = e.opposite(v);
            if (!known.contains(u)) {
                forest.put(u, e);
                depthFirst(u, known, forest);
            }
        }
    }
    ...
```

Construct Path from Discovery Edge Map

- Given two vertices u, v and the map of discovery edges.
- Return the path from u to v as list of edges.



Construct Path from Discovery Edge Map

- Given two vertices u, v and the map of discovery edges.
- Return the path from u to v as list of edges.

```
public class AdjacencyMapGraph<V, E> {
    ...
    public List<Edge> constructPath(Vertex u, Vertex v,
                        Map<Vertex, Edge> forest) {
        LinkedList<Edge> path = new LinkedList<>();
        if (forest.get(v) != null) {
            while (v != u) {
                Edge edge = forest.get(v);
                path.addFirst(edge);
                v = edge.opposite(v);
            }
        }
        return path;
    }
    ...
```

Connected Components

- Test whether a **graph** is (weakly) **connected:**
  Call $depthFirst(v, known, forrest)$ with an arbitrary vertex v and then test whether $known.size() == vertices.size()$.
- Compute all (weakly) connected components:

```java
public class AdjacencyMapGraph<V, E> {
    ...
    public Map<Vertex, Edge> depthFirstComplete() {
        Set<Vertex> known = new HashSet<>();
        Map<Vertex, Edge> forest = new HashMap<>();
        for (Vertex u : vertices())
            if (!known.contains(u))
                depthFirst(u, known, forest);
        return forest;
    }
    ...
```

- Which vertices are in which component?
- Use unique ID to mark vertices: $depthFirst(u, known, forest, uid)$.

Graph Traversal – Breadth-First Search

"Sending out explorers, in all directions, who collectively traverse a graph."
Breadth-first search starts at a vertex v, which is at level 0.

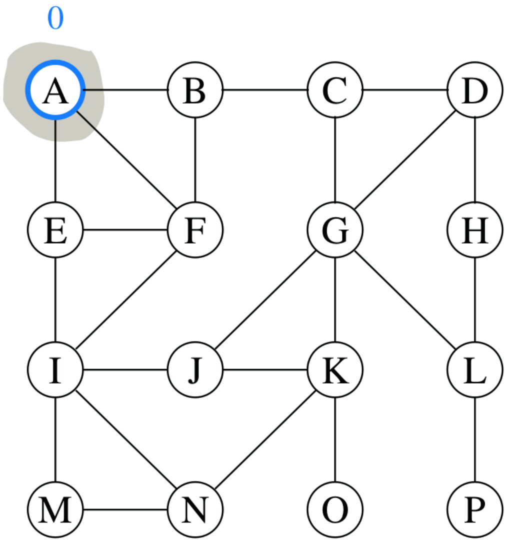
Input: A graph G and a vertex v of G.
Output: All vertices reachable from v, with their discovery edges.

Algorithm $BFS(G, v)$:

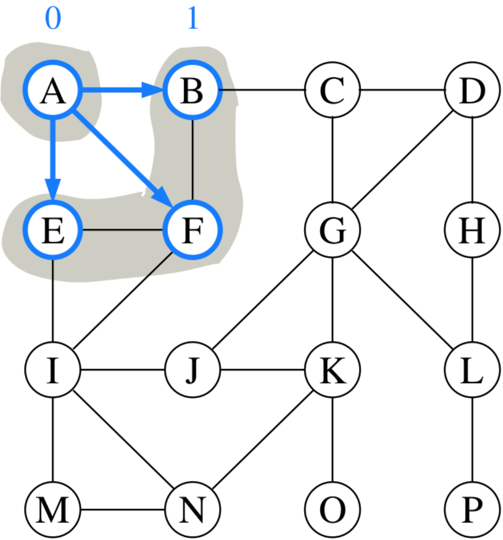
- Mark vertex v as visited.
- Put v into an empty FIFO queue Q.
- while Q is not empty
-     Poll the next vertex u from Q.
-     for each of u's outgoing edges, $e = (u, w)$ do
-         if vertex w has not been visited then
-             Record edge e as the discovery edge for vertex w.
-             Mark vertex w as visited.
-             Put w into the queue Q.

Traversal yields a breath-first search tree rooted at a starting vertex v.

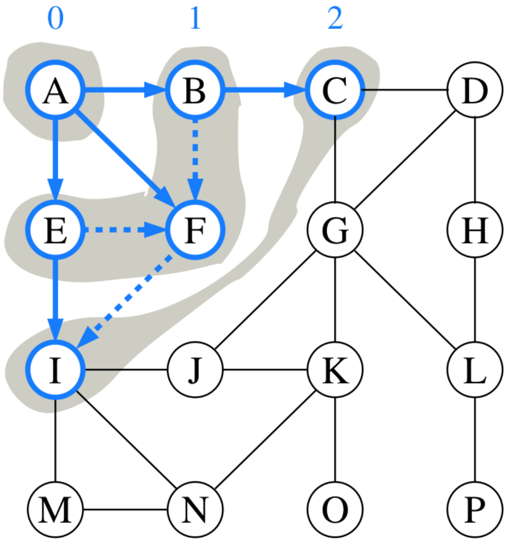
Graph Traversal – Breadth-First Example



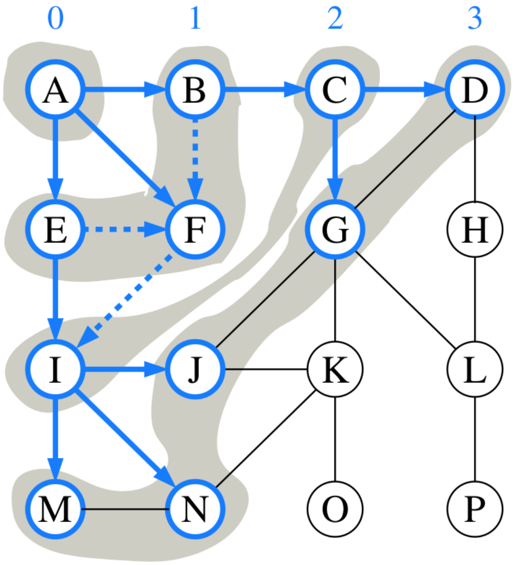
Graph Traversal – Breadth-First Example



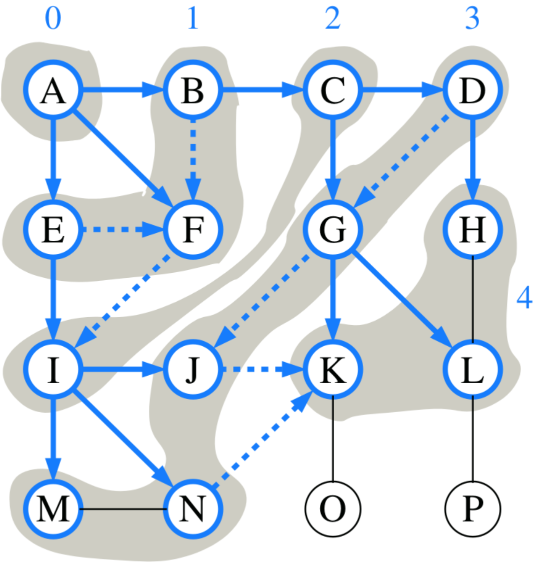
Graph Traversal – Breadth-First Example



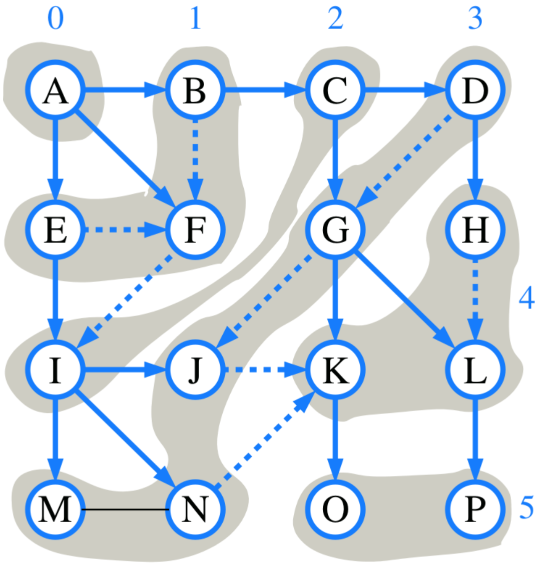
Graph Traversal – Breadth-First Example



Graph Traversal – Breadth-First Example



Graph Traversal – Breadth-First Example



Breadth-First Graph Traversal in Java

- Given a start vertex v of a graph G.
- Store all reachable vertices in $Set<Vertex>$.
- For each vertex, store discovery edge in $Map<Vertex, Edge>$.

```java
public void breathFirst(Vertex v, Set<Vertex> known,
                        Map<Vertex, Edge> forest) {
    Queue<Vertex> q = new LinkedList<>();
    known.add(v); q.add(v);
    while (!q.isEmpty()) {
        v = q.poll();
        for (Edge e : v.getOutgoing().values()) {
            Vertex u = e.opposite(v);
            if (!known.contains(u)) {
                forest.put(u, e);
                known.add(u); q.add(u);
            }
        }
    }
}
```

Properties of Breadth-First Search

- Searching for connected components is the same as for depth first search.
- We can use the method $constructPath(...)$ from before.
- The method $constructPath(...)$ returns a **shortest path** for breath first search.

Weighted Graphs

In **weighted** graphs, edges are given a weight number. E.g.:

- The physical distance between two vertices.
- The time it takes to get from one vertex to another.
- How much it costs to travel from vertex to vertex.

Formally it is modeled by a weight function $w : E \to \mathbb{R}$.

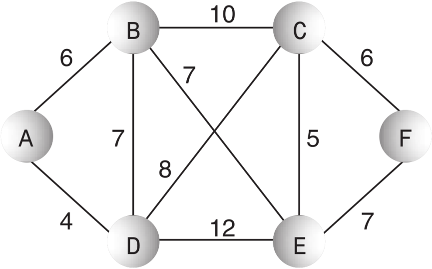
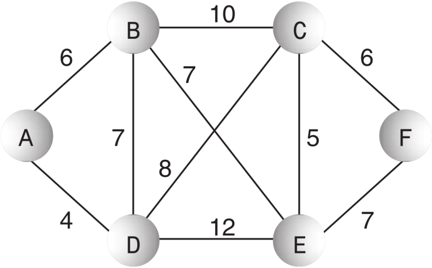


Figure: An undirected weighted graph.

Using our Generic Implementation


```
AdjacencyMapGraph<Character, Integer> g
                        = new AdjacencyMapGraph<>(false);
Vertex a = g.insertVertex('A');
Vertex b = g.insertVertex('B');
Vertex c = g.insertVertex('C');
Vertex d = g.insertVertex('D');
g.insertEdge(a, b, 6);
g.insertEdge(a, d, 4);
g.insertEdge(b, c, 10);
...
```



Edsger W. Dijkstra

- **Shortest path algorithm**, known as Dijkstra's algorithm (1959).
- Shunting yard algorithm for parsing mathematical expressions.

Figure: Edsger Wybe Dijkstra in 2002

Find "Best" Path

- **Given** a weighted graph $G = (V, E)$ and a start vertex $s \in V$.
- **Find** the cheapest way to travel from s to all the other Vertices.

Shortest Path Algorithm

- **Given** a start vertex $s \in V$ of weighted graph $G = (V, E)$ with nonnegative edge weights $w(u, v)$ for $u, v \in V$.
- **Find** the **length** of a shortest path from s to v for each vertex $v \in V$.

Algorithm $ShortestPath(G, s)$:

- Initialize a distance map D such that $s \mapsto 0$ and $v \mapsto \infty$ for $s \neq v \in V$. By $D[v]$ we denote the value associated to v.
- Let a priority queue Q contain all the mappings $v \mapsto k \in D$ where k denotes the priority. Smaller number has higher priority.
- while Q is not empty do
  - $u = Q.remove().key()$
  - for each outgoing edge (u, v) such that v is in Q do
  - if $D[u] + w(u, v) < D[v]$ then
  - $D[v] = D[u] + w(u, v)$
  - Change the key of vertex v in Q to $D[v]$.
- return D.

Shortest Path Algorithm – Result

- **Given** a start vertex $s \in V$ of weighted graph $G = (V, E)$ with nonnegative edge weights $w(u, v)$ for $u, v \in V$ and a distance map D.
- For any vertex v which is not reachable from s we get $D[v] = \infty$.
  - Obviously this solves the reachability problem.
- For any vertex v which is reachable from s we have $D[v]$ containing the length of the shortest path.
  - The shortest path to a vertex v can be red off:
  - Take all the incoming edges (u, v) and follow the edge where $D(u)$ is minimal.

Shortest Path in Maze

- We assume that 'S' stands for start and 'E' for end.
- Use an undirected graph to solve maze problem.
- No weights are needed.
- We can use breath first search.

```
public class MazeShortest {
    private int width, height;
    private AdjacencyMapGraph<Character, Integer> graph
                        = new AdjacencyMapGraph<>(false);
    private Vertex start, end;
```


Alexander.Baumgartner@risc.jku.at

Creating the Maze Graph


```
for (; in.hasNextLine(); height++) {
    char[] line = in.nextLine().toCharArray();
    for (int i = 0; i < line.length; i++, u = v) {
        v = graph.insertVertex(line[i]);
        if (v.getElem() == '#')
            continue;

        if (i > 0 && u.getElem() != '#')
            graph.insertEdge(u, v);
        int above =graph.vertices().size()-line.length -1;
        if (above > 0 &&
                graph.vertices().get(above).getElem()!='#')
            graph.insertEdge(graph.vertices().get(above),v);

        if (v.getElem() == 'S')       start = v;
        else if (v.getElem() == 'E') end = v;
    }
}
width = graph.vertices().size() / height;
```

Solving the Maze Graph

- Use the breath first search to obtain map of discovery edges.
- Construct the path from the map of discovery edges.

```
public boolean solve() {
    Set<...Vertex> known = new HashSet<>();
    Map<...Vertex, ...Edge> forest = new HashMap<>();
    graph.breathFirst(start, known, forest);
    List<...Edge> path
            = graph.constructPath(start, end, forest);
    for (int i = 1, n = path.size() - 1; i < n; i++) {
        path.get(i).getVertU().setElem('.');
        path.get(i).getVertV().setElem('.');
    }
    return !path.isEmpty();
}
```

Exercise

- A graph $G = (V, E)$ is bipartite if V can be partitioned into two sets $X \subseteq V$ and $Y = V \setminus X$ such that every edge in G has one end vertex in X and the other in Y.
- Design an algorithm for determining if an undirected graph G is bipartite.

See the guidance for this exercise on the Moodle page.