4.2 Directed Graphs

- introduction
- digraph API
- digraph search
- topological sort
- strong components
Directed graphs

**Digraph.** Set of vertices connected pairwise by *directed* edges.

![Directed graph example](image)
Road network

Vertex = intersection; edge = one-way street.
## Digraph applications

<table>
<thead>
<tr>
<th>digraph</th>
<th>vertex</th>
<th>directed edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>transportation</td>
<td>street intersection</td>
<td>one-way street</td>
</tr>
<tr>
<td>web</td>
<td>web page</td>
<td>hyperlink</td>
</tr>
<tr>
<td>food web</td>
<td>species</td>
<td>predator-prey relationship</td>
</tr>
<tr>
<td>WordNet</td>
<td>synset</td>
<td>hypernym</td>
</tr>
<tr>
<td>scheduling</td>
<td>task</td>
<td>precedence constraint</td>
</tr>
<tr>
<td>financial</td>
<td>bank</td>
<td>transaction</td>
</tr>
<tr>
<td>cell phone</td>
<td>person</td>
<td>placed call</td>
</tr>
<tr>
<td>infectious disease</td>
<td>person</td>
<td>infection</td>
</tr>
<tr>
<td>game</td>
<td>board position</td>
<td>legal move</td>
</tr>
<tr>
<td>citation</td>
<td>journal article</td>
<td>citation</td>
</tr>
<tr>
<td>object graph</td>
<td>object</td>
<td>pointer</td>
</tr>
<tr>
<td>inheritance hierarchy</td>
<td>class</td>
<td>inherits from</td>
</tr>
<tr>
<td>control flow</td>
<td>code block</td>
<td>jump</td>
</tr>
</tbody>
</table>
## Digraph API

```java
public class Digraph {
    Digraph(int V)                      // create an empty digraph with V vertices
    Digraph(In in)                     // create a digraph from input stream
    void addEdge(int v, int w)          // add a directed edge v→w
    Iterable<Integer> adj(int v)       // vertices pointing from v
    int V()                            // number of vertices
    int E()                            // number of edges
    Digraph reverse()                  // reverse of this digraph
    String toString()                  // string representation
}
```

```java
In in = new In(args[0]);
Digraph G = new Digraph(in);
for (int v = 0; v < G.V(); v++)
    for (int w : G.adj(v))
        StdOut.println(v + "->" + w);
```

- read digraph from input stream
- print out each edge (once)
Adjacency-lists digraph representation

Maintain vertex-indexed array of lists.
Adjacency-lists graph representation (review): Java implementation

```java
public class Graph {
    private final int V;
    private final Bag<Integer>[] adj;

    public Graph(int V) {
        this.V = V;
        adj = (Bag<Integer>[]) new Bag[V];
        for (int v = 0; v < V; v++)
            adj[v] = new Bag<Integer>();
    }

    public void addEdge(int v, int w) {
        adj[v].add(w);
        adj[w].add(v);
    }

    public Iterable<Integer> adj(int v) {
        return adj[v];
    }
}
```
Adjacency-lists digraph representation: Java implementation

```java
class Digraph {
    private final int V;
    private final Bag<Integer>[] adj;

    public Digraph(int V) {
        this.V = V;
        adj = new Bag[V];
        for (int v = 0; v < V; v++)
            adj[v] = new Bag<>();
    }

    public void addEdge(int v, int w) {
        adj[v].add(w);
    }

    public Iterable<Integer> adj(int v) {
        return adj[v];
    }
}
```
**Digraph representations**

**In practice.** Use adjacency-lists representation.
- Algorithms based on iterating over vertices pointing from \( v \).
- Real-world digraphs tend to be sparse.

![Comparison of digraph representations](image)

<table>
<thead>
<tr>
<th>representation</th>
<th>space</th>
<th>insert edge from ( v ) to ( w )</th>
<th>edge from ( v ) to ( w )?</th>
<th>iterate over vertices pointing from ( v )?</th>
</tr>
</thead>
<tbody>
<tr>
<td>list of edges</td>
<td>( E )</td>
<td>1</td>
<td>( E )</td>
<td>( E )</td>
</tr>
<tr>
<td>adjacency matrix</td>
<td>( V^2 )</td>
<td>( 1^\dagger )</td>
<td>1</td>
<td>( V )</td>
</tr>
<tr>
<td>adjacency lists</td>
<td>( E + V )</td>
<td>1</td>
<td>outdegree(( v ))</td>
<td>outdegree(( v ))</td>
</tr>
</tbody>
</table>

\( ^\dagger \) disallows parallel edges

---

huge number of vertices, small average vertex degree
Depth-first search in digraphs

Same method as for undirected graphs.

- Every undirected graph is a digraph (with edges in both directions).
- DFS is a digraph algorithm.

**DFS (to visit a vertex v)**

Mark v as visited.
Recursively visit all unmarked vertices w pointing from v.
Depth-first search demo

To visit a vertex \( v \):
- Mark vertex \( v \) as visited.
- Recursively visit all unmarked vertices pointing from \( v \).

\[
\begin{align*}
0 & \rightarrow 1 \\
1 & \rightarrow 2 \\
2 & \rightarrow 3 \\
3 & \rightarrow 4 \\
4 & \rightarrow 5 \\
5 & \rightarrow 6 \\
6 & \rightarrow 7 \\
7 & \rightarrow 8 \\
8 & \rightarrow 9 \\
9 & \rightarrow 10 \\
10 & \rightarrow 11 \\
11 & \rightarrow 12 \\
12 & \rightarrow 0
\end{align*}
\]
To visit a vertex $v$:

- Mark vertex $v$ as visited.
- Recursively visit all unmarked vertices pointing from $v$. 

---

**Depth-first search demo**

---

<table>
<thead>
<tr>
<th>$v$</th>
<th>marked[]</th>
<th>edgeTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>–</td>
</tr>
<tr>
<td>1</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>9</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>10</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>11</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>12</td>
<td>F</td>
<td>–</td>
</tr>
</tbody>
</table>
Depth-first search (in directed graphs)

Code for directed graphs identical to undirected one.
[substitute Digraph for Graph]

```java
public class DirectedDFS {
    private boolean[] marked;

    public DirectedDFS(Digraph G, int s) {
        marked = new boolean[G.V()];
        dfs(G, s);
    }

    private void dfs(Digraph G, int v) {
        marked[v] = true;
        for (int w : G.adj(v))
            if (!marked[w]) dfs(G, w);
    }

    public boolean visited(int v) {
        return marked[v];
    }
}
```

- Constructor marks vertices reachable from s
- Recursive DFS does the work
- Client can ask whether any vertex is reachable from s
- True if path from s
Reachability application: mark-sweep garbage collector

Mark-sweep algorithm. [McCarthy, 1960]

- Mark: mark all reachable objects.
- Sweep: if object is unmarked, it is garbage (so add to free list).

Memory cost. Uses 1 extra mark bit per object (plus DFS stack).
Depth-first search in digraphs summary

DFS enables direct solution of simple digraph problems.
✓ • Reachability.
   • Path finding.
   • Topological sort.
   • Directed cycle detection.

Basis for solving difficult digraph problems.
• 2-satisfiability.
• Directed Euler path.
• Strongly-connected components.

SIAM J. COMPUT.
Vol. 1, No. 2, June 1972

DEPTH-FIRST SEARCH AND LINEAR GRAPH ALGORITHMS
ROBERT TARJAN

Abstract. The value of depth-first search or “backtracking” as a technique for solving problems is illustrated by two examples. An improved version of an algorithm for finding the strongly connected components of a directed graph and an algorithm for finding the biconnected components of an undirected graph are presented. The space and time requirements of both algorithms are bounded by \( k_1 V + k_2 E + k_3 \) for some constants \( k_1, k_2, \) and \( k_3 \), where \( V \) is the number of vertices and \( E \) is the number of edges of the graph being examined.
Breadth-first search in digraphs

Same method as for undirected graphs.

- Every undirected graph is a digraph (with edges in both directions).
- BFS is a digraph algorithm.

**BFS (from source vertex s)**

Put s onto a FIFO queue, and mark s as visited.
Repeat until the queue is empty:
- remove the least recently added vertex v
- for each unmarked vertex pointing from v:
  add to queue and mark as visited.

**Proposition.** BFS computes shortest paths (fewest number of edges) from $s$ to all other vertices in a digraph in time proportional to $E + V$. 
Directed breadth-first search demo

Repeat until queue is empty:
- Remove vertex $v$ from queue.
- Add to queue all unmarked vertices pointing from $v$ and mark them.

Graph $G$
Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex $v$ from queue.
- Add to queue all unmarked vertices pointing from $v$ and mark them.

```
<table>
<thead>
<tr>
<th>v</th>
<th>edgeTo[]</th>
<th>distTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>
```

done
Topological sort

**DAG.** Directed *acyclic* graph.

**Topological sort.** Redraw DAG so all edges point upwards.

Solution. DFS. What else?
Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.

```
postorder
4 1 2 5 0 6 3

topological order
3 6 0 5 2 1 4
```

done
public class DepthFirstOrder
{
    private boolean[] marked;
    private Stack<Integer> reversePost;

    public DepthFirstOrder(Digraph G)
    {
        reversePost = new Stack<Integer>();
        marked = new boolean[G.V()];
        for (int v = 0; v < G.V(); v++)
            if (!marked[v]) dfs(G, v);
    }

    private void dfs(Digraph G, int v)
    {
        marked[v] = true;
        for (int w : G.adj(v))
            if (!marked[w]) dfs(G, w);
        reversePost.push(v);
    }

    public Iterable<Integer> reversePost()
    { return reversePost;  }
}
Minimum spanning tree

**Given.** Undirected graph $G$ with positive edge weights (connected).

**Def.** A spanning tree of $G$ is a subgraph $T$ that is connected and acyclic.

**Goal.** Find a min weight spanning tree.
Minimum spanning tree

**Given.** Undirected graph $G$ with positive edge weights (connected).

**Def.** A spanning tree of $G$ is a subgraph $T$ that is connected and acyclic.

**Goal.** Find a min weight spanning tree.

---

Minimum spanning tree

Given. Undirected graph $G$ with positive edge weights (connected).

Def. A spanning tree of $G$ is a subgraph $T$ that is connected and acyclic.

Goal. Find a min weight spanning tree.
Minimum spanning tree

**Given.** Undirected graph $G$ with positive edge weights (connected).

**Def.** A spanning tree of $G$ is a subgraph $T$ that is connected and acyclic.

**Goal.** Find a min weight spanning tree.

![Graph with edge weights](image)

**spanning tree T:** cost = 50 = 4 + 6 + 8 + 5 + 11 + 9 + 7

**Brute force.** Try all spanning trees?
Weighted edge API

Edge abstraction needed for weighted edges.

```java
public class Edge implements Comparable<Edge>

    Edge(int v, int w, double weight) // create a weighted edge v-w

    int either() // either endpoint

    int other(int v) // the endpoint that's not v

    int compareTo(Edge that) // compare this edge to that edge

    double weight() // the weight

    String toString() // string representation
```

Idiom for processing an edge e: int v = e.either(), w = e.other(v);
Weighted edge: Java implementation

```java
public class Edge implements Comparable<Edge>
{
    private final int v, w;
    private final double weight;

    public Edge(int v, int w, double weight)
    {
        this.v = v;
        this.w = w;
        this.weight = weight;
    }

    public int either()
    { return v; }

    public int other(int vertex)
    {
        if (vertex == v) return w;
        else return v;
    }

    public int compareTo(Edge that)
    {
        if       (this.weight < that.weight) return -1;
        else if (this.weight > that.weight) return +1;
        else return 0;
    }
}
```

- `constructor`:
- `either endpoint`:
- `other endpoint`:
- `compare edges by weight`
# Edge-weighted graph API

```java
public class EdgeWeightedGraph {
    // constructor
    EdgeWeightedGraph(int V) { /* create an empty graph with V vertices */ }
    EdgeWeightedGraph(In in) { /* create a graph from input stream */ }

    // methods
    void addEdge(Edge e) { /* add weighted edge e to this graph */ }

    Iterable<Edge> adj(int v) { /* edges incident to v */ }
    Iterable<Edge> edges() { /* all edges in this graph */ }

    int V() { /* number of vertices */ }
    int E() { /* number of edges */ }
    String toString() { /* string representation */ }
}
```

**Conventions.** Allow self-loops and parallel edges.
Edge-weighted graph: adjacency-lists representation

Maintain vertex-indexed array of Edge lists.
Edge-weighted graph: adjacency-lists implementation

```java
public class EdgeWeightedGraph
{
    private final int V;
    private final Bag<Edge>[] adj;

    public EdgeWeightedGraph(int V)
    {
        this.V = V;
        adj = (Bag<Edge>[]) new Bag[V];
        for (int v = 0; v < V; v++)
            adj[v] = new Bag<Edge>();
    }

    public void addEdge(Edge e)
    {
        int v = e.either(), w = e.other(v);
        adj[v].add(e);
        adj[w].add(e);
    }

    public Iterable<Edge> adj(int v)
    {
        return adj[v];
    }
}
```

same as Graph, but adjacency lists of Edges instead of integers

constructor

add edge to both adjacency lists
Kruskal's algorithm demo

Consider edges in ascending order of weight.
- Add next edge to tree $T$ unless doing so would create a cycle.

**Kruskal's algorithm demo**

an edge-weighted graph

0-7 0.16
2-3 0.17
1-7 0.19
0-2 0.26
5-7 0.28
1-3 0.29
1-5 0.32
2-7 0.34
4-5 0.35
1-2 0.36
4-7 0.37
0-4 0.38
6-2 0.40
3-6 0.52
6-0 0.58
6-4 0.93
Kruskal's algorithm demo

Consider edges in ascending order of weight.
- Add next edge to tree $T$ unless doing so would create a cycle.

\[ \begin{array}{ccc}
0-7 & 0.16 \\
2-3 & 0.17 \\
1-7 & 0.19 \\
0-2 & 0.26 \\
5-7 & 0.28 \\
1-3 & 0.29 \\
1-5 & 0.32 \\
2-7 & 0.34 \\
4-5 & 0.35 \\
1-2 & 0.36 \\
4-7 & 0.37 \\
0-4 & 0.38 \\
6-2 & 0.40 \\
3-6 & 0.52 \\
6-0 & 0.58 \\
6-4 & 0.93 \\
\end{array} \]
Prim's algorithm demo

- Start with vertex 0 and greedily grow tree $T$.
- Add to $T$ the min weight edge with exactly one endpoint in $T$.
- Repeat until $V - 1$ edges.

an edge-weighted graph
Primes's algorithm demo

- Start with vertex 0 and greedily grow tree $T$.
- Add to $T$ the min weight edge with exactly one endpoint in $T$.
- Repeat until $V - 1$ edges.

MST edges

0–7 1–7 0–2 2–3 5–7 4–5 6–2
4.4 Shortest Paths

- APIs
- shortest-paths properties
- Dijkstra's algorithm
- edge-weighted DAGs
- negative weights
Edge-weighted digraph: adjacency-lists representation

tinyEWD.txt
Dijkstra's algorithm demo

- Consider vertices in increasing order of distance from $s$ (non-tree vertex with the lowest disTo[] value).
- Add vertex to tree and relax all edges pointing from that vertex.
Dijkstra's algorithm demo

- Consider vertices in increasing order of distance from \( s \) (non-tree vertex with the lowest \text{distTo}[] \) value).
- Add vertex to tree and relax all edges pointing from that vertex.

\[
\begin{array}{c|c|c}
\text{v} & \text{distTo[]} & \text{edgeTo[]} \\
0 & 0.0 & - \\
1 & 5.0 & 0\rightarrow1 \\
2 & 14.0 & 5\rightarrow2 \\
3 & 17.0 & 2\rightarrow3 \\
4 & 9.0 & 0\rightarrow4 \\
5 & 13.0 & 4\rightarrow5 \\
6 & 25.0 & 2\rightarrow6 \\
7 & 8.0 & 0\rightarrow7 \\
\end{array}
\]

shortest-paths tree from vertex \( s \)
Dijkstra's algorithm: which priority queue?

Depends on PQ implementation: $V$ insert, $V$ delete-min, $E$ decrease-key.

<table>
<thead>
<tr>
<th>PQ implementation</th>
<th>insert</th>
<th>delete-min</th>
<th>decrease-key</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>unordered array</td>
<td>1</td>
<td>$V$</td>
<td>1</td>
<td>$V^2$</td>
</tr>
<tr>
<td>binary heap</td>
<td>$\log V$</td>
<td>$\log V$</td>
<td>$\log V$</td>
<td>$E \log V$</td>
</tr>
<tr>
<td>d-way heap (Johnson 1975)</td>
<td>$d \log_d V$</td>
<td>$d \log_d V$</td>
<td>$\log_d V$</td>
<td>$E \log_{E/V} V$</td>
</tr>
<tr>
<td>Fibonacci heap (Fredman–Tarjan 1984)</td>
<td>$1^\dagger$</td>
<td>$\log V^\dagger$</td>
<td>$1^\dagger$</td>
<td>$E + V \log V$</td>
</tr>
</tbody>
</table>

$^\dagger$ amortized

**Bottom line.**

- Array implementation optimal for dense graphs.
- Binary heap much faster for sparse graphs.
- 4-way heap worth the trouble in performance-critical situations.
- Fibonacci heap best in theory, but not worth implementing.
### Single source shortest-paths implementation: cost summary

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Restriction</th>
<th>Typical case</th>
<th>Worst case</th>
<th>Extra space</th>
</tr>
</thead>
<tbody>
<tr>
<td>topological sort</td>
<td>no directed cycles</td>
<td>E + V</td>
<td>E + V</td>
<td>V</td>
</tr>
<tr>
<td>Dijkstra (binary heap)</td>
<td>no negative weights</td>
<td>E log V</td>
<td>E log V</td>
<td>V</td>
</tr>
<tr>
<td>Bellman–Ford</td>
<td>no negative cycles</td>
<td>E V</td>
<td>E V</td>
<td>V</td>
</tr>
<tr>
<td>Bellman–Ford (queue-based)</td>
<td>no negative cycles</td>
<td>E + V</td>
<td>E V</td>
<td>V</td>
</tr>
</tbody>
</table>

**Remark 1.** Directed cycles make the problem harder.
**Remark 2.** Negative weights make the problem harder.
**Remark 3.** Negative cycles makes the problem intractable.