4. Graphs and Digraphs I

- introduction
- graph representation
- depth-first search
- path finding
- undirected graphs
4. **Graphs and Digraphs I**

- introduction
- graph representation
- depth-first search
- path finding
- undirected graphs
Graphs

**Graph.** Set of *vertices* connected pairwise by *edges*.

**Why study graphs and graph algorithms?**

- Broadly useful abstraction.
- Hundreds of graph algorithms.
- Thousands of real-world applications.
- Fascinating branch of computer science and discrete math.
Transportation networks

Vertex = subway stop; edge = direct route.
Social networks

Vertex = person; edge = social relationship.

"Visualizing Friendships" by Paul Butler
Vertex = Twitter account; edge = Twitter follower.
Protein-protein interaction network

Vertex = protein; edge = interaction.

Reference: Jeong et al, Nature Review | Genetics
## Graph applications

<table>
<thead>
<tr>
<th><strong>graph</strong></th>
<th><strong>vertex</strong></th>
<th><strong>edge</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>cell phone</td>
<td>phone</td>
<td>placed call</td>
</tr>
<tr>
<td>infectious disease</td>
<td>person</td>
<td>infection</td>
</tr>
<tr>
<td>financial</td>
<td>stock, currency</td>
<td>transactions</td>
</tr>
<tr>
<td>transportation</td>
<td>intersection</td>
<td>street</td>
</tr>
<tr>
<td>internet</td>
<td>router</td>
<td>fiber cable</td>
</tr>
<tr>
<td>web</td>
<td>web page</td>
<td>URL link</td>
</tr>
<tr>
<td>social relationship</td>
<td>person</td>
<td>friendship</td>
</tr>
<tr>
<td>object graph</td>
<td>object</td>
<td>pointer</td>
</tr>
<tr>
<td>protein network</td>
<td>protein</td>
<td>protein–protein interaction</td>
</tr>
<tr>
<td>circuit</td>
<td>gate, register, processor</td>
<td>wire</td>
</tr>
<tr>
<td>neural network</td>
<td>neuron</td>
<td>synapse</td>
</tr>
</tbody>
</table>
Undirected graph terminology

**Graph.** Set of vertices connected pairwise by edges.

**Path.** Sequence of vertices connected by edges, with no repeated edges.

**Def.** Two vertices are connected if there is a path between them.

**Cycle.** Path (with ≥ 1 edge) whose first and last vertices are the same.
**Directed graph terminology**

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

**Directed path.** Sequence of vertices connected by directed edges, with no repeated edges.

**Def.** Vertex $w$ is reachable from vertex $v$ if there is a directed path from $v$ to $w$.

**Directed cycle.** Directed path (with $\geq 1$ edge) whose first and last vertices are the same.
Which of these graphs is best modeled as a directed graph?

A. Facebook: vertex = person; edge = friendship.
B. Web: vertex = webpage; edge = URL link.
C. Internet: vertex = router; edge = fiber optic cable.
D. Molecule: vertex = atom; edge = chemical bond.
Some graph-processing problems

<table>
<thead>
<tr>
<th>graph problem</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>s–t path</em></td>
<td><em>Find a path between s and t.</em></td>
</tr>
<tr>
<td>shortest <em>s–t path</em></td>
<td><em>Find a path with the fewest edges between s to t.</em></td>
</tr>
<tr>
<td><em>cycle</em></td>
<td><em>Find a cycle.</em></td>
</tr>
<tr>
<td><em>Euler cycle</em></td>
<td><em>Find a cycle that uses each edge exactly once.</em></td>
</tr>
<tr>
<td><em>Hamilton cycle</em></td>
<td><em>Find a cycle that uses each vertex exactly once.</em></td>
</tr>
<tr>
<td><em>connectivity</em></td>
<td><em>Is there a path between every pair of vertices?</em></td>
</tr>
<tr>
<td><em>graph isomorphism</em></td>
<td><em>Are two graphs isomorphic?</em></td>
</tr>
<tr>
<td><em>planarity</em></td>
<td><em>Draw the graph in the plane with no crossing edges.</em></td>
</tr>
</tbody>
</table>

**Challenge.** Which problems are easy? Difficult? Intractable?
4. **Graphs and Digraphs I**

- introduction
- graph representation
- depth-first search
- path finding
- undirected graphs
Digraph representation

Vertex representation.

- This lecture: integers between 0 and $V - 1$.
- Applications: use symbol table to convert between names and integers.

Def. A digraph is simple if it has no self-loops or parallel edges.
Digraph API

```java
public class Digraph {
    // create an empty digraph with V vertices
    Digraph(int V) {
    }
    // add a directed edge v→w
    void addEdge(int v, int w) {
    }
    // vertices adjacent from v
    Iterable<Integer> adj(int v) {
    }
    // number of vertices
    int V() {
    }
    ...

    // outdegree of vertex v in digraph G
    public static int outdegree(Digraph G, int v) {
        int count = 0;
        for (int w : G.adj(v)) {
            count++;
        }
        return count;
    }
}
```

- `Digraph(int V)` creates an empty digraph with V vertices.
- `void addEdge(int v, int w)` adds a directed edge v→w.
- `Iterable<Integer> adj(int v)` returns the vertices adjacent from v.
- `int V()` returns the number of vertices.

Note: this method is in full Digraph API, so no need to re-implement.

This API allows self loops and parallel edges.
Adjacency-matrix representation

Maintain a $V$-by-$V$ boolean array; for each edge $v \rightarrow w$ in the digraph: $\text{adj}[v][w] = \text{true}$. 

Note: parallel edges disallowed
What is the running time of the following code fragment?

Assume *adjacency-matrix* representation, $V = \#$ vertices, $E = \#$ edges.

```java
def spanning_tree(G):
    # Your implementation here
```

A. $\Theta(V)$
B. $\Theta(E + V)$
C. $\Theta(V^2)$
D. $\Theta(E V)$
Adjacency-lists representation

Maintain vertex-indexed array of lists.
What is the running time of the following code fragment?
Assume **adjacency-lists** representation, \( V = \# \) vertices, \( E = \# \) edges.

```java
for (int v = 0; v < G.V(); v++)
    for (int w : G.adj(v))
        StdOut.println(v + "->" + w);
```

print each edge once

A. \( \Theta(V) \)
B. \( \Theta(E + V) \)
C. \( \Theta(V^2) \)
D. \( \Theta(E \ V) \)
### Digraph representations

**In practice.** Use adjacency-lists representation.

- Algorithms based on iterating over vertices adjacent from $v$.
- Real-world graphs tend to be sparse (not dense).

![Adjacency matrix and adjacency lists comparison](image)

<table>
<thead>
<tr>
<th>representation</th>
<th>space</th>
<th>add edge from $v$ to $w$</th>
<th>has edge from $v$ to $w$?</th>
<th>iterate over vertices adjacent from $v$?</th>
</tr>
</thead>
<tbody>
<tr>
<td>adjacency matrix</td>
<td>$V^2$</td>
<td>1 †</td>
<td>1</td>
<td>$V$</td>
</tr>
<tr>
<td>adjacency lists</td>
<td>$E + V$</td>
<td>1</td>
<td>$\text{outdegree}(v)$</td>
<td>$\text{outdegree}(v)$</td>
</tr>
</tbody>
</table>

† disallows parallel edges
public class Digraph
{
    private final int V;
    private 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<Integer>();
    }

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

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

adjacency lists
create empty digraph with V vertices
add edge \( v \rightarrow w \)
(parallel edges and self-loops allowed)
iterator for vertices adjacent from \( v \)
4. **Graphs and Digraphs I**

- introduction
- graph representation
- depth-first search
- path finding
- undirected graphs

[https://algs4.cs.princeton.edu](https://algs4.cs.princeton.edu)
Problem. Given a digraph $G$ and vertex $s$, find all vertices reachable from $s$. 
Depth-first search

**Goal.** Systematically traverse a digraph.

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

- Mark vertex v.
- Recursively visit all unmarked vertices w adjacent from v.

**Typical applications.**

- Reachability: find all vertices reachable from a given vertex.
- Path finding: find a directed path from one vertex to another vertex.
Directed depth-first search demo

To visit a vertex \( v \):

- Mark vertex \( v \).
- Recursively visit all unmarked vertices adjacent from \( v \).

a directed graph
Directed depth-first search demo

To visit a vertex $v$:

- Mark vertex $v$.
- Recursively visit all unmarked vertices adjacent from $v$.

<table>
<thead>
<tr>
<th>$v$</th>
<th>marked[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>T</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
</tr>
<tr>
<td>9</td>
<td>F</td>
</tr>
<tr>
<td>10</td>
<td>F</td>
</tr>
<tr>
<td>11</td>
<td>F</td>
</tr>
<tr>
<td>12</td>
<td>F</td>
</tr>
</tbody>
</table>

reachable from vertex 0
Run DFS using the following adjacency-lists representation of digraph $G$, starting at vertex 0. In which order is $\text{dfs}(G, v)$ called?

A. 0 1 2 4 5 3 6
B. 0 1 2 4 5 6 3
C. 0 1 3 2 6 4 5
D. 0 1 2 6 4 5 3
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 isReachable(int v)
    {
        return marked[v];
    }
}
Depth-first search: properties

Proposition. DFS marks all vertices reachable from $s$ in $\Theta(E + V)$ time in the worst case.

Pf.

• Initializing an array of length $V$ takes $\Theta(V)$ time.
• Each vertex is visited at most once.
• Visiting a vertex takes time proportional to its outdegree:

$$outdegree(v_0) + outdegree(v_1) + outdegree(v_2) + \ldots = E$$

in worst case, all $V$ vertices reachable from $s$

Note. If all vertices are reachable from $s$, then $E \geq V - 1$, so $V$ is a lower-order term.
Graphs and digraphs: quiz 5

What could happen if we marked a vertex at the end of the DFS call (instead of beginning)?

A. Marks a vertex not reachable from $s$.
B. Compile–time error.
C. Infinite loop / stack overflow.
D. None of the above.

```java
private void dfs(Digraph G, int v) {
    marked[v] = true;
    for (int w : G.adj(v))
        if (!marked[w])
            dfs(G, w);
}
```
Reachability application: program control-flow analysis

Every program is a digraph.
- Vertex = basic block of instructions (straight-line program).
- Edge = jump.

Dead-code elimination.
Find (and remove) unreachable code.

Infinite-loop detection.
Determine whether exit is unreachable.
Reachability application: mark–sweep garbage collector

Every data structure is a digraph.
- Vertex = object.
- Edge = reference/pointer.

Roots. Objects known to be directly accessible by program (e.g., stack frame).

Reachable objects. Objects indirectly accessible by program (starting at a root and following a chain of pointers).
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 function-call stack).
4. **Graphs and Digraphs I**

- introduction
- graph representation
- depth-first search
- path finding
- undirected graphs
**Goal.** DFS determines which vertices are reachable from \( s \). How to reconstruct paths?

**Solution.** Use parent-link representation.

### Parent-Link Representation

<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>

The table above shows the marked and edgeTo[] values for each vertex, indicating which vertices are marked and the index of the next vertex in the path. The diagram illustrates the parent-link representation of paths from vertex 0.
Depth-first search: path finding

Parent-link representation of paths from $s$.

- Maintain an integer array `edgeTo[]`.
- Interpretation: `edgeTo[v]` is the next-to-last vertex on a path from $s$ to $v$.
- To reconstruct path from $s$ to $v$, trace `edgeTo[]` backward from $v$ to $s$ (and reverse).

<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>
</tbody>
</table>

```java
class public Iterable<Integer> pathTo(int v)
{
    if (!marked[v]) return null;
    Stack<Integer> path = new Stack<>();
    for (int x = v; x != s; x = edgeTo[x])
        path.push(x);
    path.push(s);
    return path;
}
```
Depth-first search (with path finding): Java implementation

```java
public class DepthFirstDirectedPaths {

    private boolean[] marked;
    private int[] edgeTo;
    private int s;

    public DepthFirstDirectedPaths(Graph G, int s) {
        ... 
        dfs(G, s);
    }

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

    // Other methods...

    // edgeTo[v] = previous vertex on path from s to v
    // v→w is edge that led to w
}
```

Graphs and digraphs: quiz 6

Suppose there are many paths from s to v. Which one does DepthFirstDirectedPaths find?

A. A shortest path (fewest edges).
B. A longest path (most edges).
C. Depends on digraph representation.
4. Graphs and Digraphs I

- introduction
- graph representation
- depth-first search
- path finding
- undirected graphs
**Problem.** Implement flood fill (Photoshop magic wand).
Depth-first search in undirected graphs

Problem. Given an undirected graph $G$ and vertex $s$, find all vertices connected to $s$.

Solution. Treat undirected graph as a digraph, replacing each edge with two antiparallel edges.

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

Mark vertex $v$.
Recursively visit all unmarked vertices $w$ adjacent to $v$.

Typical applications.

- Find all vertices connected to a given vertex.
- Find a path between two vertices.
Depth-first search demo

To visit a vertex $v$:

- Mark vertex $v$.
- Recursively visit all unmarked vertices adjacent to $v$. 

*graph G*
Depth-first search demo

To visit a vertex $v$:

- Mark vertex $v$.
- Recursively visit all unmarked vertices adjacent to $v$.

<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>0</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>T</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>T</td>
<td>0</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>
How to represent an undirected edge $v$–$w$ using adjacency lists?

A. Add $w$ to adjacency list for $v$.

B. Add $v$ to adjacency list for $w$.

C. Both A and B.

D. None of the above.
**Digraph representation (review)**

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

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

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

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

- **adjacency lists**: Represents the graph as an array of bags where each bag represents the neighbors of a vertex.
- **create empty digraph with V vertices**: Initializes the digraph with V vertices.
- **add edge v→w**: Adds an edge from vertex v to vertex w. (parallel edges and self-loops allowed)
- **iterator for vertices adjacent from v**: Returns an iterator for the vertices adjacent to vertex v.
Graph representation

```java
public class Graph {
    private final int V;
    private 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

create empty graph with $V$ vertices

add edge $v$–$w$
(parallel edges and self-loops allowed)

iterator for vertices adjacent to $v$

https://algs4.cs.princeton.edu/41undirected/Graph.java.html
Depth-first search (in digraphs)

Recall code for digraphs.

```java
public class DirectedFS {
    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];
    }
}
```

- `marked[v] = true if v reachable from s`
- Constructor marks vertices reachable from `s`
- Recursive DFS does the work
- Is vertex `v` is reachable from `s`?
Depth-first search (in undirected graphs)

Code for undirected graphs is essentially identical to code for digraphs.

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

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

    private void dfs(Graph 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];
    }
}
```

- `marked[v] = true` if `v` connected to `s`
- Constructor marks vertices connected to `s`
- Recursive DFS does the work
- Is vertex `v` is connected to `s`?
Depth-first search summary

DFS enables direct solution of simple graph and digraph problems.

- Reachability (in a digraph).
- Connectivity (in a graph).
- Path finding (in a graph or digraph).
- Topological sort.
- Directed cycle detection.

Next lecture: precept

DFS provides basis for solving difficult graph problems.

- Euler cycle.
- 2-satisfiability.
- Planarity testing.
- Strong components.

---

Dealing with graphs other than trees.

DFS provides basis for solving difficult graph problems.

- Euler cycle.
- 2-satisfiability.
- Planarity testing.
- Strong components.

---

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 $k_1 V + k_2 E + k_3$, where $V$ is the number of vertices and $E$ is the number of edges of the graph being examined.