4. Graphs and Digraphs

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

https://algs4.cs.princeton.edu
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?

- 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
Twitter followers

Vertex = Twitter account; edge = Twitter follower.
Protein-protein interaction network

Vertex = protein; edge = interaction.
## Graph applications

<table>
<thead>
<tr>
<th>graph</th>
<th>vertex</th>
<th>edge</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 optic 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>logic gate</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.

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

**Cycle.** Path (with \( \geq 1 \) 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.

**Reachable.** 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>s-t path</td>
<td>Find a path between s and t.</td>
</tr>
<tr>
<td>shortest s-t path</td>
<td>Find a path with the fewest edges between s to t.</td>
</tr>
<tr>
<td>cycle</td>
<td>Find a cycle.</td>
</tr>
<tr>
<td>Euler cycle</td>
<td>Find a cycle that uses each edge exactly once.</td>
</tr>
<tr>
<td>Hamilton cycle</td>
<td>Find a cycle that uses each vertex exactly once.</td>
</tr>
<tr>
<td>connected components</td>
<td>Find connected components.</td>
</tr>
<tr>
<td>graph isomorphism</td>
<td>Find an isomorphism between two graphs.</td>
</tr>
<tr>
<td>planarity</td>
<td>Draw in the plane with no crossing edges.</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

https://algs4.cs.princeton.edu
Vertex representation.

- This lecture: integers between $0$ and $V - 1$.
- Real-world 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 {
    public Digraph(int V) {
        // create an empty digraph with V vertices
    }
    void addEdge(int v, int w) {
        // add a directed edge v→w
        // our API allows self-loops and parallel edges
    }
    Iterable<Integer> adj(int v) {
        // vertices adjacent from v
    }
    int V() {
        // number of vertices
    }
    Digraph reverse() {
        // reverse digraph
    }
    // ...
}
```

// 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;
}

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

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

Memory. $\Theta(V^2)$ space.
Digraph representation: adjacency lists

Maintain vertex-indexed array of lists: \( \text{adj}[v] \) contains vertices adjacent from vertex \( v \).

**Memory.** \( \Theta(E + V) \) space.
What is the running time of the following code fragment?
Assume \textbf{adjacency-lists} representation, \( V = \# \text{ vertices}, E = \# \text{ edges}. \)

```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. \( \Theta(EV) \)
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]

<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>outdegree($v$)</td>
<td>outdegree($v$)</td>
</tr>
</tbody>
</table>

† disallows parallel edges
Digraph representation (adjacency lists): Java implementation

```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<>();
    }

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

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

- **adjacency lists**
  (could use a stack or queue instead of a bag)

- **create empty digraph with V vertices**

- **add edge v→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
Reachability problem in a digraph

Reachability problem. Given a digraph $G$ and vertex $s$, find all vertices \textit{reachable} from $s$. 
Reachability problem in a digraph

Reachability problem. Given a digraph $G$ and vertex $s$, find all vertices reachable from $s$.

Depth-first search. A systematic method to explore all vertices reachable from $s$.

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

Mark vertex v.
Recursively visit all unmarked vertices w adjacent from v.
Directed depth-first search demo

To visit a vertex $v$:

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

A directed graph

4→2
2→3
3→2
6→0
0→1
2→0
11→12
12→9
9→10
9→11
8→9
10→12
11→4
4→3
3→5
6→8
8→6
5→4
0→5
6→4
6→9
7→6
Directed depth-first search demo

To visit a vertex $v$:

- Mark vertex $v$.
- Recursively visit all unmarked vertices adjacent from $v$.
Run DFS using the given adjacency-lists representation of digraph G, starting at vertex 0. In which order is 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

adjacency-lists representation

digraph G
```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 isReachable(int v) {
        return marked[v];
    }
}
```

marked[v] = true if v is reachable from s
constructor marks vertices reachable from s
recursive DFS does the work
is v reachable from s?
Depth-first search: running time

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

**Pf.**
- Initializing the marked[] array takes \( \Theta(V) \) time.
- Each vertex is visited at most once.
- Visiting a vertex takes time proportional to its outdegree:

\[
\text{outdegree}(v_0) + \text{outdegree}(v_1) + \text{outdegree}(v_2) + \ldots = E
\]

\[\uparrow\]

*in worst case, all \( V \) vertices are reachable from \( s \)*

**Note.** If all vertices are reachable from \( s \), then \( E \geq V - 1 \) and running time simplifies to \( \Theta(E) \).
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);
    marked[v] = true;
}
```
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

https://algs4.cs.princeton.edu
Directed paths DFS demo

**Goal.** DFS determines which vertices are reachable from $s$. How to reconstruct paths?

**Solution.** Use parent-link representation.
Depth-first search: path finding

Parent–link representation of paths from $s$.

- Maintain an integer array edgeTo[].
- Interpretation: $\text{edgeTo}[v]$ is the next–to–last vertex on a path from $s$ to $v$.
- To reconstruct path from $s$ to $v$, trace $\text{edgeTo}[]$ 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>
Depth-first search (with path finding): Java implementation

```java
public class DepthFirstDirectedPaths {

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

    public DepthFirstDirectedPaths(Digraph 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]) {
                dfs(G, w);
                edgeTo[w] = v;
            }
        }
    }

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

    // More methods...
}
```

- `marked[v]` is a boolean array that marks if a vertex `v` has been visited.
- `edgeTo[w]` is an integer array that stores the previous vertex on the path from `s` to `v`.
- `dfs(G, s)` is the depth-first search function that starts from vertex `s`.
- `v → w` is the edge that led to the discovery of `w`.

This implementation follows the Breadth-First Search (BFS) approach but with an added edgeTo array to keep track of the previous vertex on the path. It is particularly useful for finding paths in a directed graph.
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

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

Solution. Use DFS. but now, for each undirected edge $v$–$w$:

$v$ is adjacent to $w$ and $w$ is adjacent to $v$

\begin{itemize}
  \item \textbf{Mark vertex $v$.}
  \item \textbf{Recursively visit all unmarked vertices $w$ adjacent to $v$.}
\end{itemize}

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

```
vertices connected to 0
(and associated paths)
```

```
<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.
Directed graph 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<>();
    }

    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→w**
- **iterator for vertices adjacent from v**

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<>();
    }

    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
iterator for vertices adjacent to v

https://algs4.cs.princeton.edu/41graph/Graph.java.html
Depth-first search (in directed graphs)

```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 isReachable(int v) {
        return marked[v];
    }
}
```

marked[v] = true if v is reachable from s
constructor marks vertices reachable from s
recursive DFS does the work
is v reachable from s?

https://algs4.cs.princeton.edu/42digraph/DirectedDFS.java.html
Depth-first search (in undirected graphs)

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

  public DirectedDFS(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 isConnected(int v) {
    return marked[v];
  }
}
```

- `marked[v] = true if v is connected to s`
- `constructor marks vertices connected to s`
- `recursive DFS does the work`
- `is v connected to s ?`

https://algs4.cs.princeton.edu/41graph/DepthFirstSearch.java.html
Depth-first search summary

DFS enables direct solution of several elementary graph and digraph problems.

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

DFS is also core of solution to more advanced problems.

- Euler cycle.
- Biconnectivity.
- 2-satisfiability.
- Planarity testing.
- Strong components.
- Nonbipartite matching.

...
<table>
<thead>
<tr>
<th>image</th>
<th>source</th>
<th>license</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pac–Man Graph</td>
<td>Oatzy</td>
<td></td>
</tr>
<tr>
<td>Pac–Man Game</td>
<td>Old Classic Retro Gaming</td>
<td></td>
</tr>
<tr>
<td>London Tube Map</td>
<td>Transport for London</td>
<td></td>
</tr>
<tr>
<td>London Tube Graph</td>
<td>visualize.org</td>
<td></td>
</tr>
<tr>
<td>Visualizing Friendships</td>
<td>Paul Butler / Facebook</td>
<td></td>
</tr>
<tr>
<td>Twitter Graph</td>
<td>allthingsgraphed.com</td>
<td></td>
</tr>
<tr>
<td>Protein Interaction Graph</td>
<td>Hawing Jeong / KAIST</td>
<td></td>
</tr>
<tr>
<td>PageRank</td>
<td>Wikipedia</td>
<td>public domain</td>
</tr>
<tr>
<td>Control Flow Graph</td>
<td>Stack Exchange</td>
<td></td>
</tr>
<tr>
<td>DFS Graph Visualization</td>
<td>Gerry Jenkins</td>
<td></td>
</tr>
</tbody>
</table>