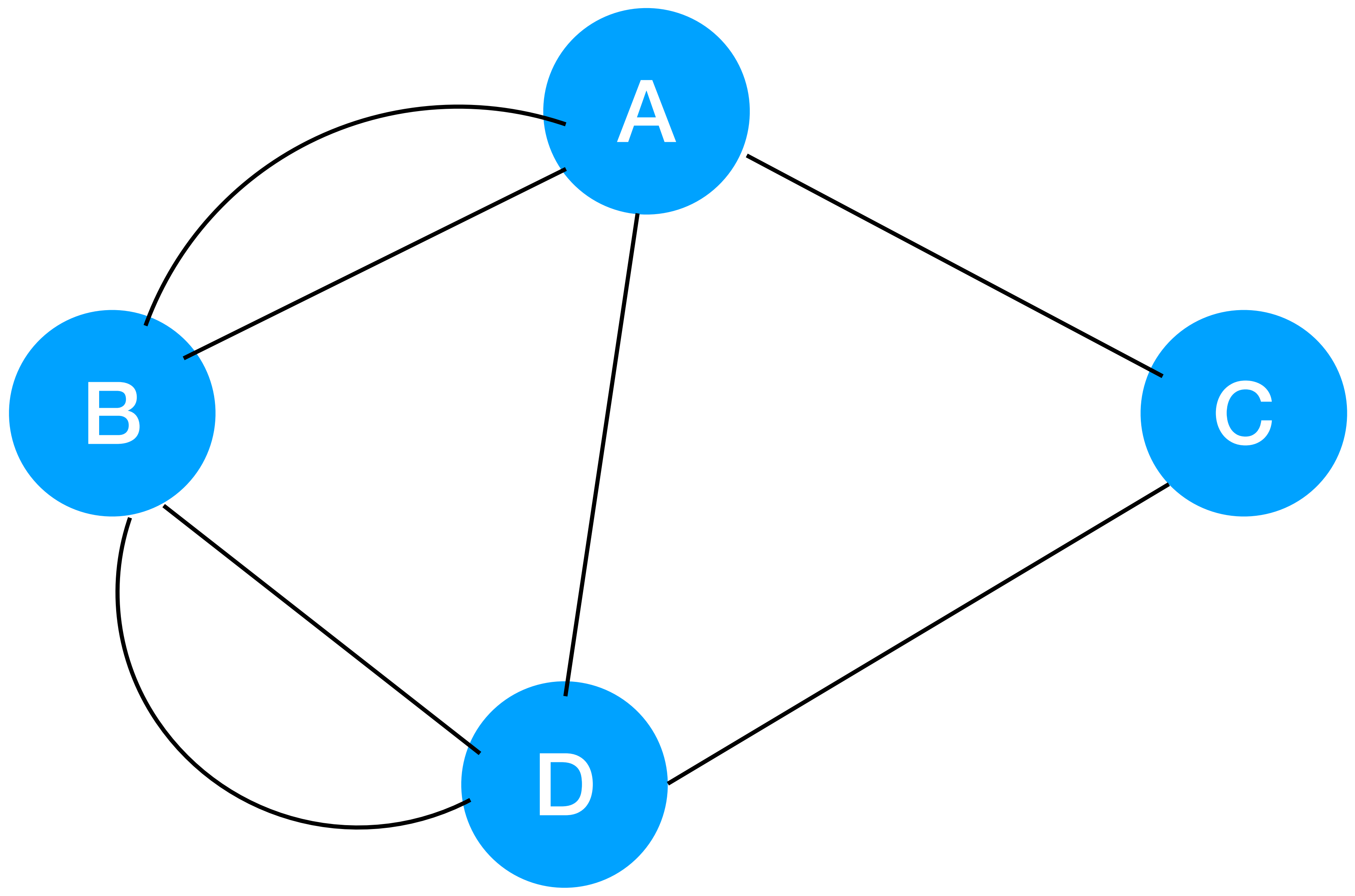
Prinzipien der Programmierung: Graphs and Catalan

Addie Jordon (he/they)
November 2024
addie.jordon@uni-bielefeld.de

ADT: Graphs

Graph Definition

- A **graph** $G = (V, E)$ is a set of V **vertices** (nodes) and a collection E of pairs from V, called **edges**



V **= {A, B, C, D}**

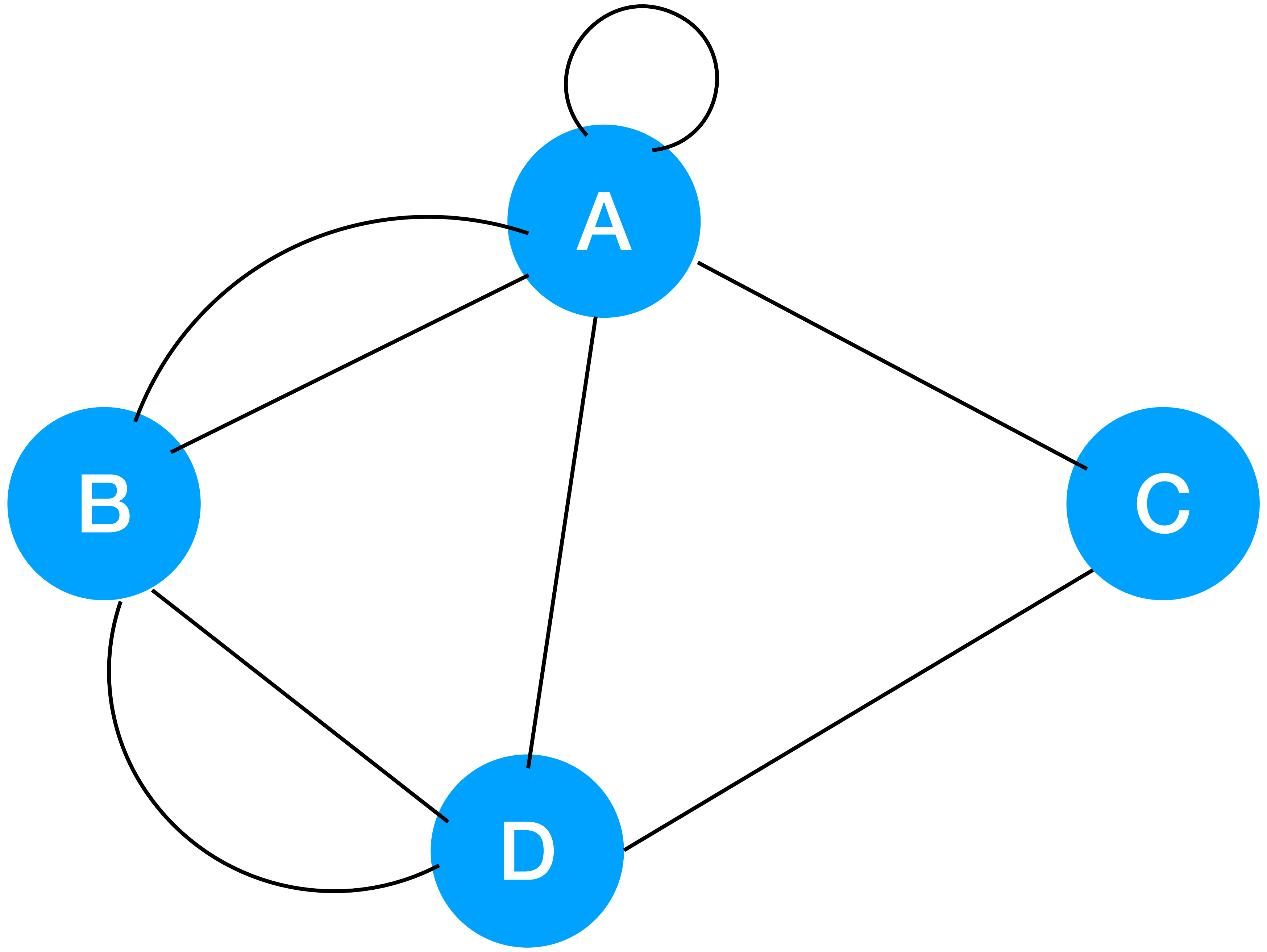
E **= {{A, B}, {A,B}, {B,D}, {B,D}, {A,D}, {D,C}, {A,C}}**

Different Types of Graphs

- *directed* graphs (digraphs)
- *undirected* graphs
- *simple* graphs
- *complete* graphs
- *connected* graphs
- *acyclic* graphs
- *bipartite* graphs
- *weighted* graphs
- trees

Undirected Graphs

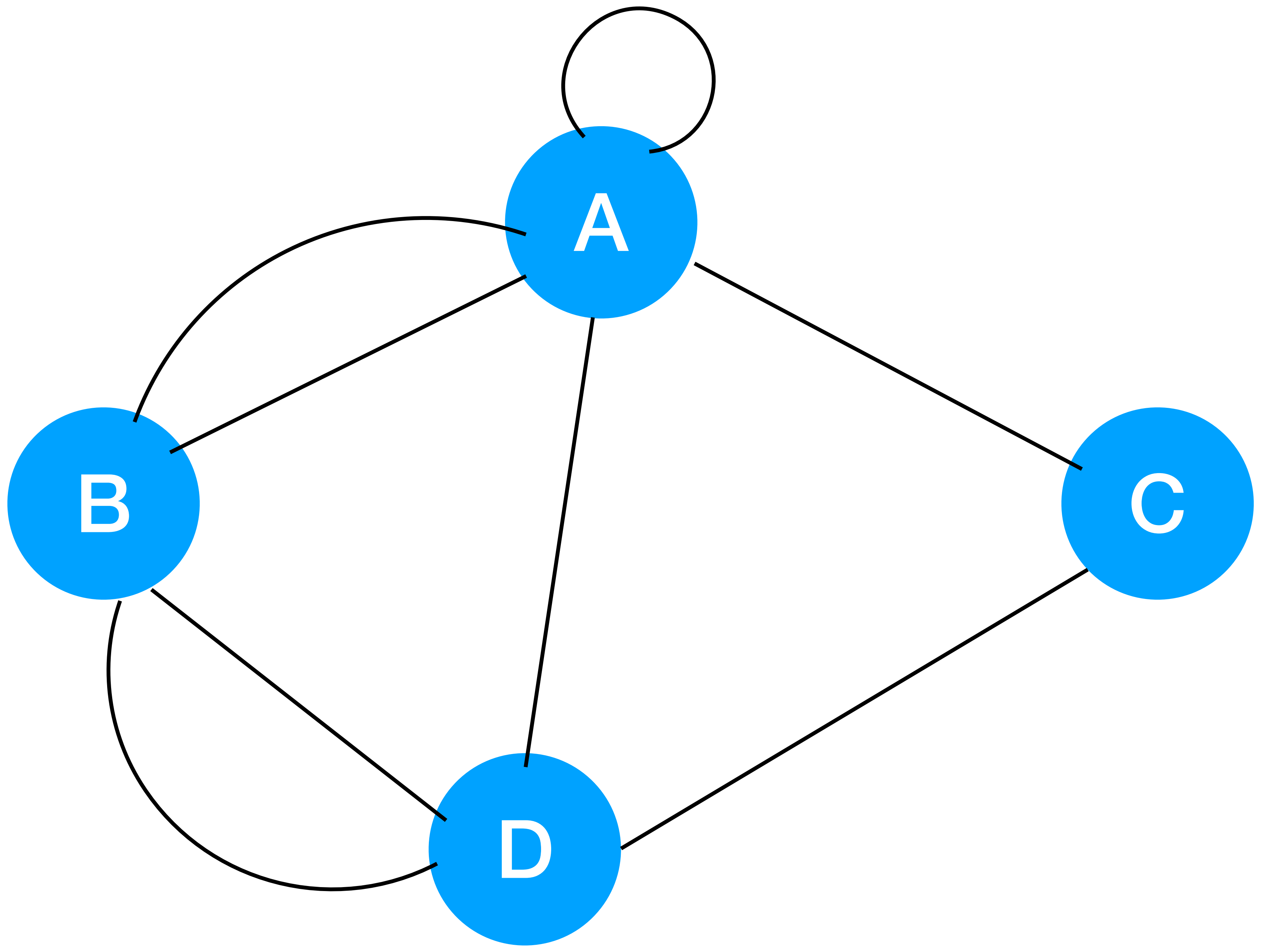
- An undirected edge e represents a symmetric relation between two vertices v and w.
    - $e = \{v, w\}$ where $\{v, w\}$ is an <u>unordered</u> pair
    - v, w are *endpoints*
    - v is *adjacent* to w
    - e is incident to both v, w
    - n: the number of vertices, $|V|$
    - m: the number of edges, $|E|$



undirected

Undirected Graphs

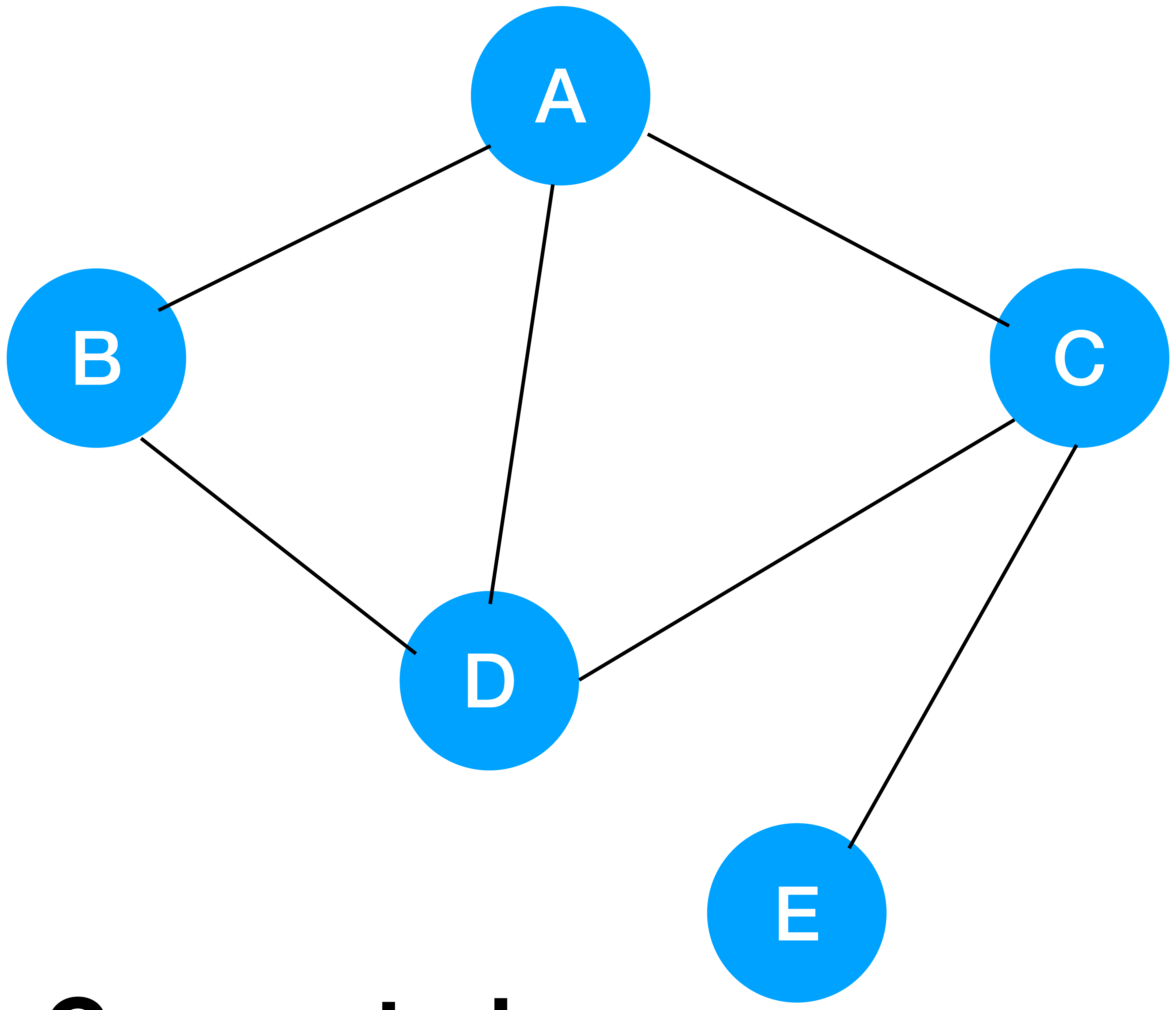
- An undirected edge e represents a symmetric relation between two vertices v and w.
  - *degree* of a vertex is the number of edges incident to it
  - eg. deg(A) = 4
- *parallel edges*: more than one edge between a pair of vertices (uncommon)
- *self-loop*: an edge that connects a vertex to itself
  - for this course, unless specified, you can assume the graph will not have parallel edges nor self-loops



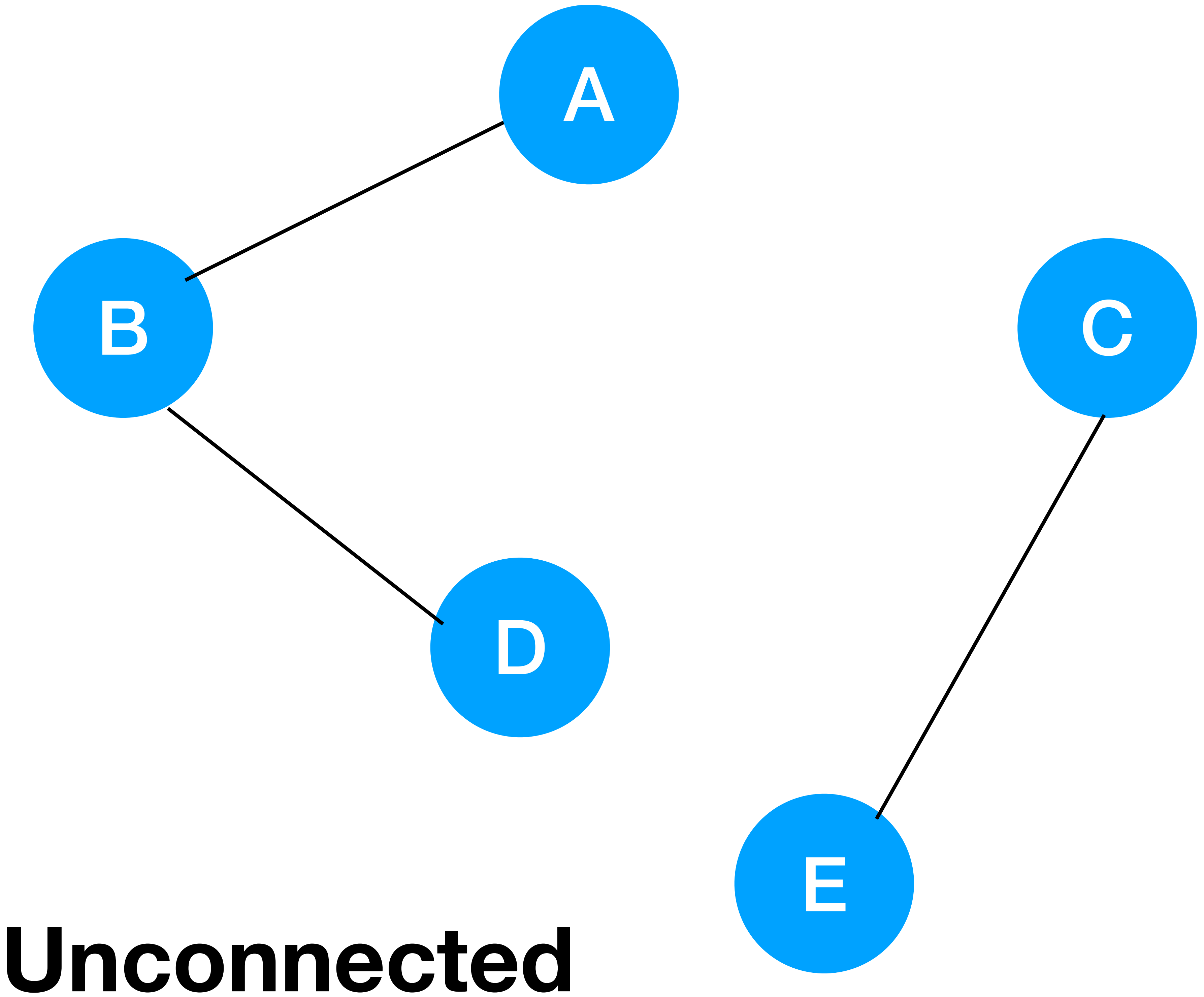
undirected

Connected Graphs

- A graph is *connected* if every pair of vertices is connected by a path



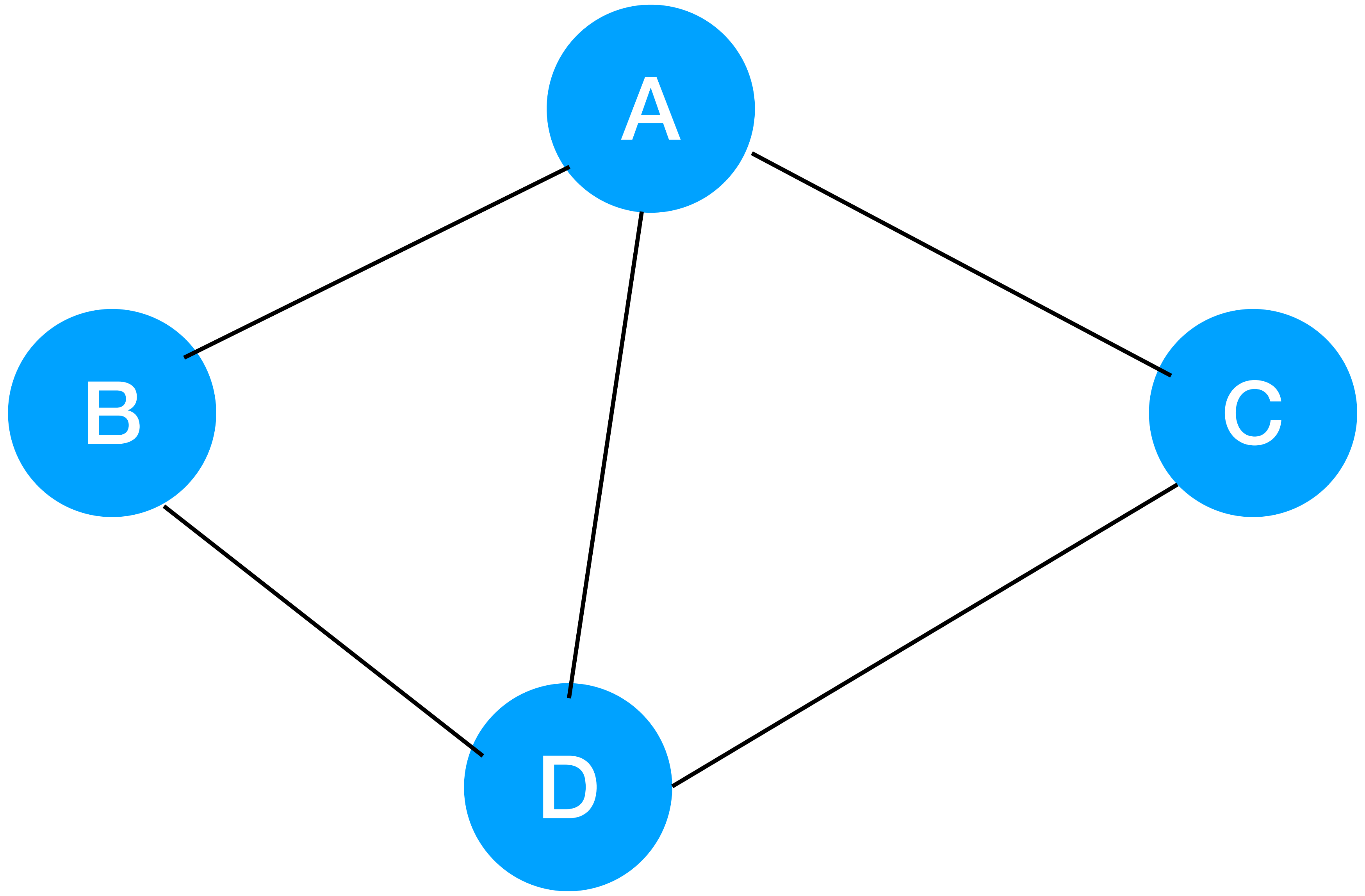
Connected



Unconnected

Graph ADT: Operations

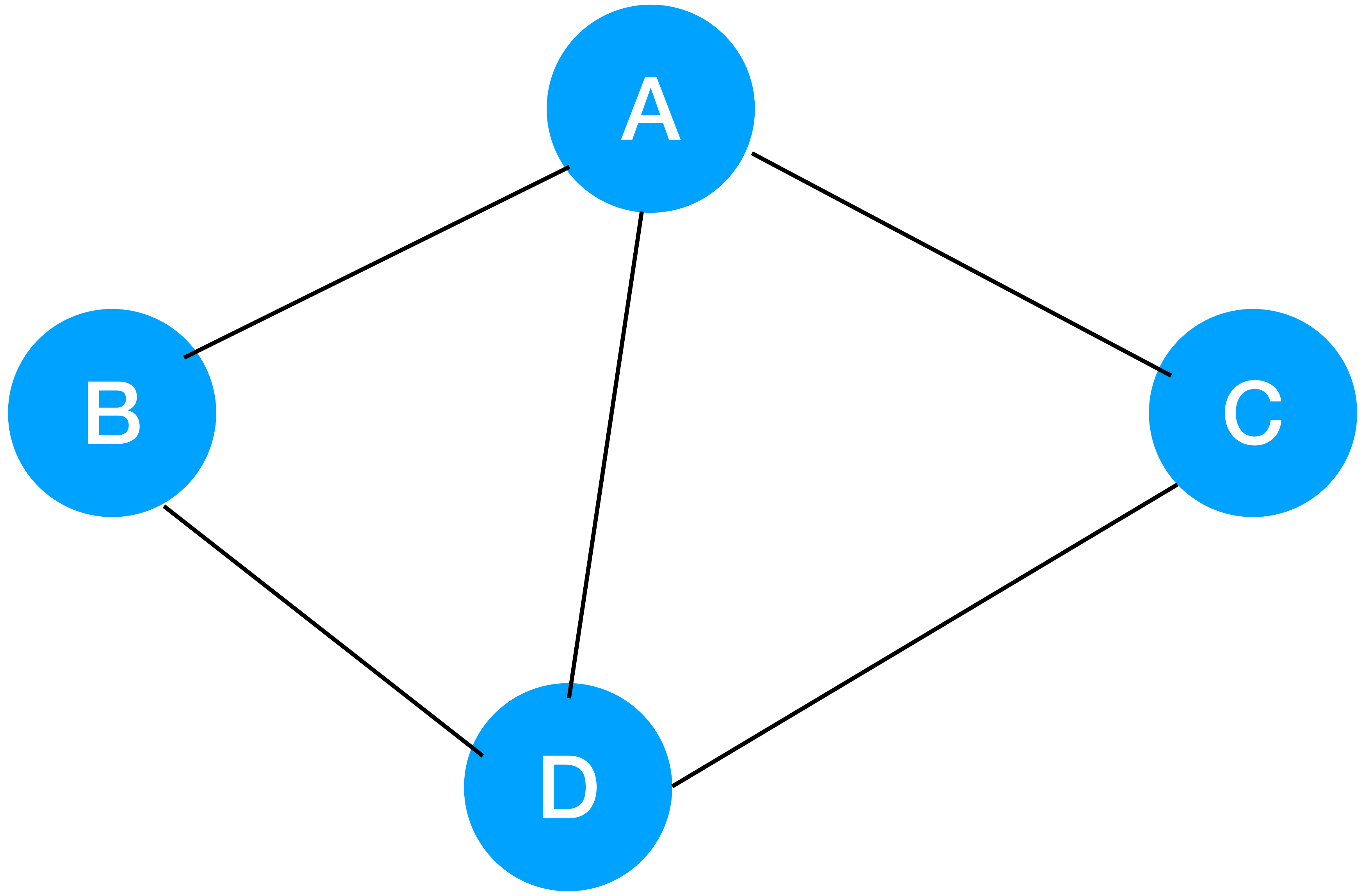
- `numVertices()`: returns the number of vertices in the graph, n
- `numEdges()`: returns the number of edges in the graph, m
- `vertices()`: returns an iterator of the vertices in G
- `edges()`: returns an iterator of the edges in G
- `degree(v)`: returns the degree of vertex v



undirected

Graph ADT: Operations

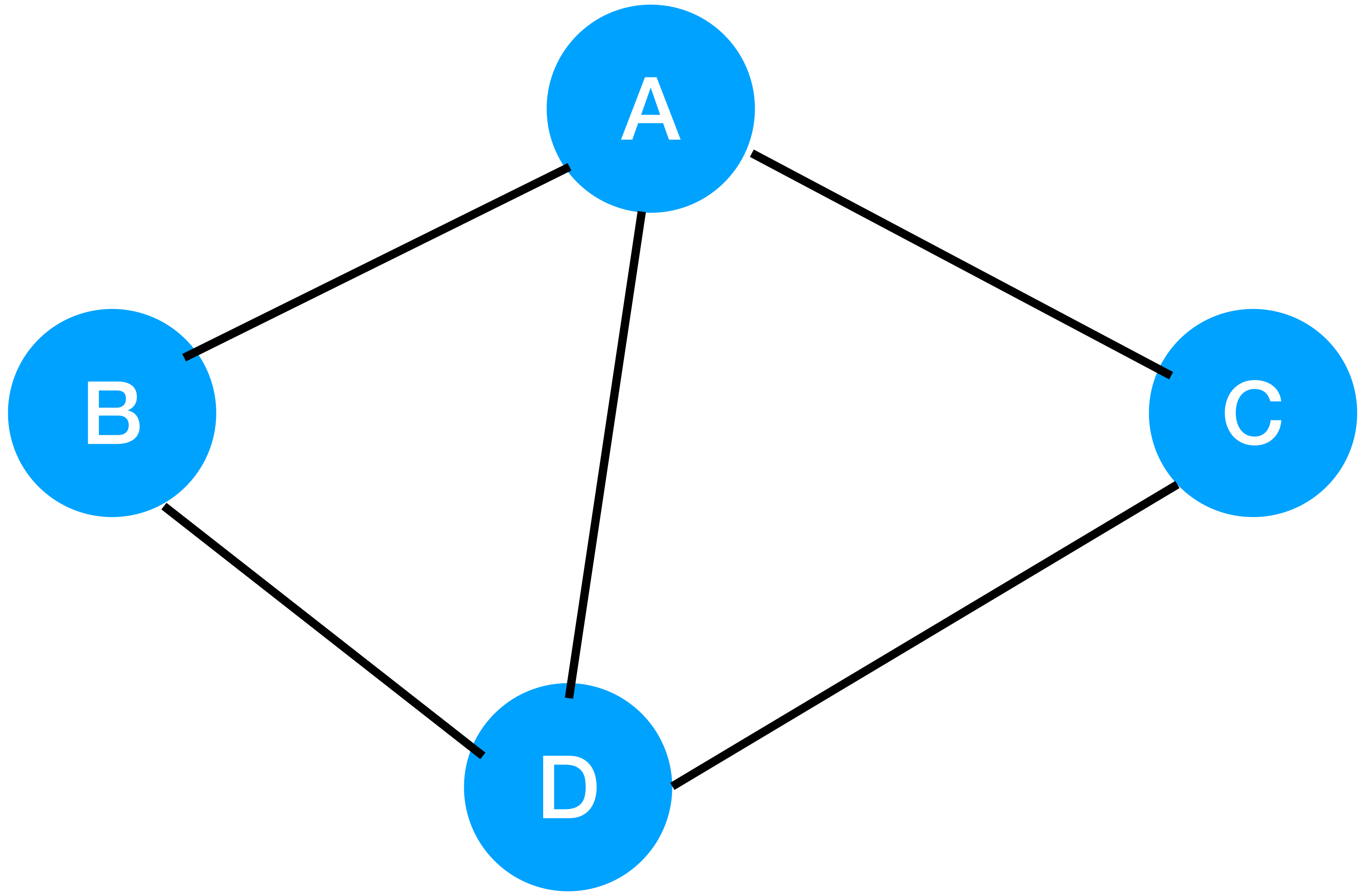
- `adjacentVertices(v)` : iterator of all neighbours of v
- `incidentEdges(v)`: iterator of all edges incident to v
- `endpoints(e)`: v, w that are endpoints of e
- `opposite(v,e)` : w, the other endpoint of e
- `areAdjacent(v,w)`: true if v, w are neighbours, false otherwise



undirected

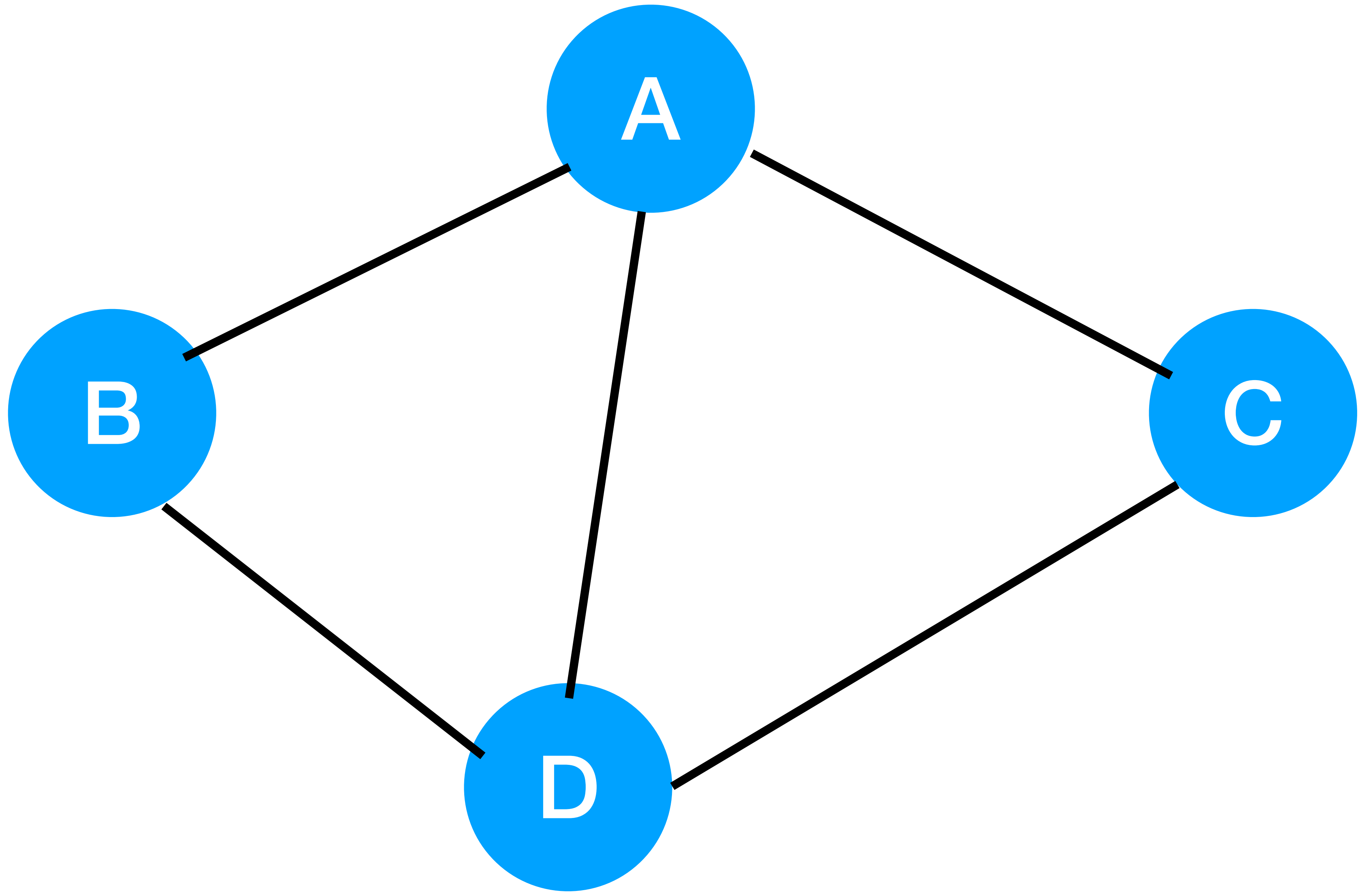
Graph modification methods

- `insertEdge(v,w)` : insert and return an undirected edge between vertices v and w
- `insertDirectedEdge(v,w)`: insert an return a directed edge between vertices v and w, with v as the source and w as the destination
- `insertVertex(v,o)`: insert and return an isolated vertex v with object o stored in the vertex

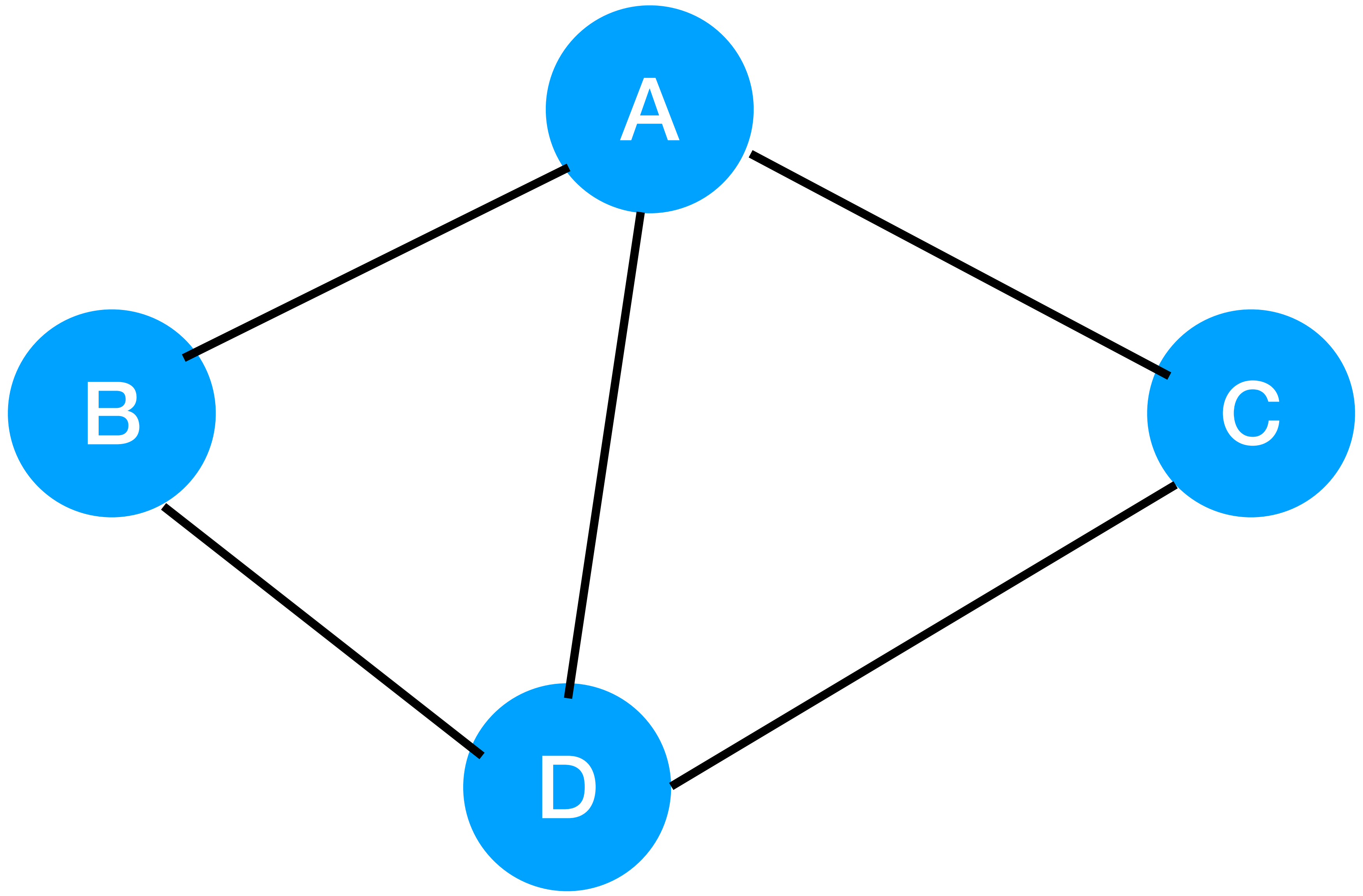


Graph modification methods

- `removeVertex(v)` : remove vertex v and all its edges
- `removeEdge(e)`: remove edge e
- `makeUndirected(e)`: make edge e undirected
- `reverseDirection(e)` : reverse direction of directed edge e
- `setDirectionFrom(e,v)`: make edge e directed away from v
- `setDirectionTo(e,v)`: make edge e directed into v

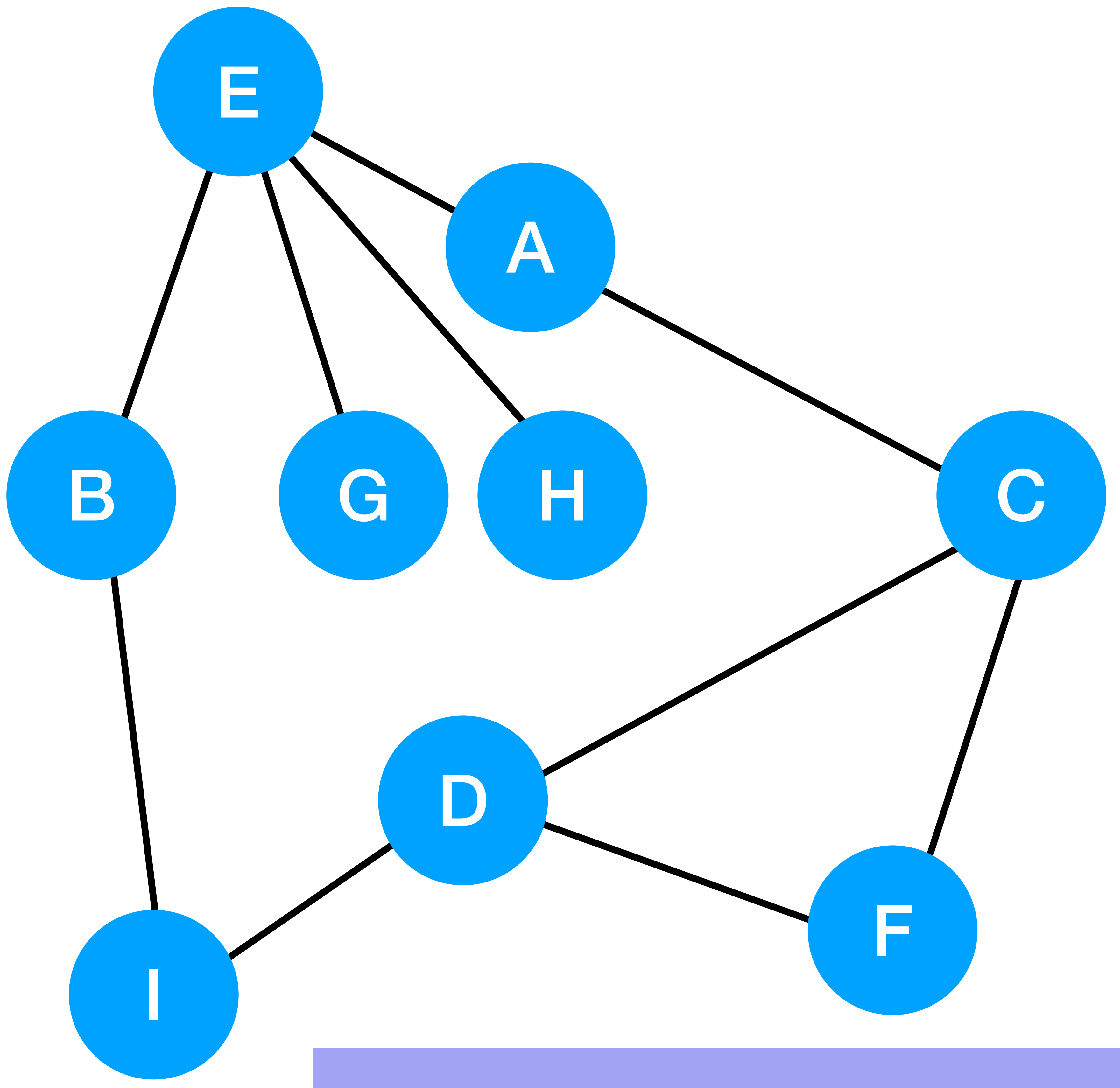


How to store a graph?



- Node-centric option
  - vertex and edge objects
  - adjacency-lists
  - labeled adjacency-lists
  - adjacency-matrix (0s and 1s)
  - labeled adjacency-matrix

Option 1: store a list of edges



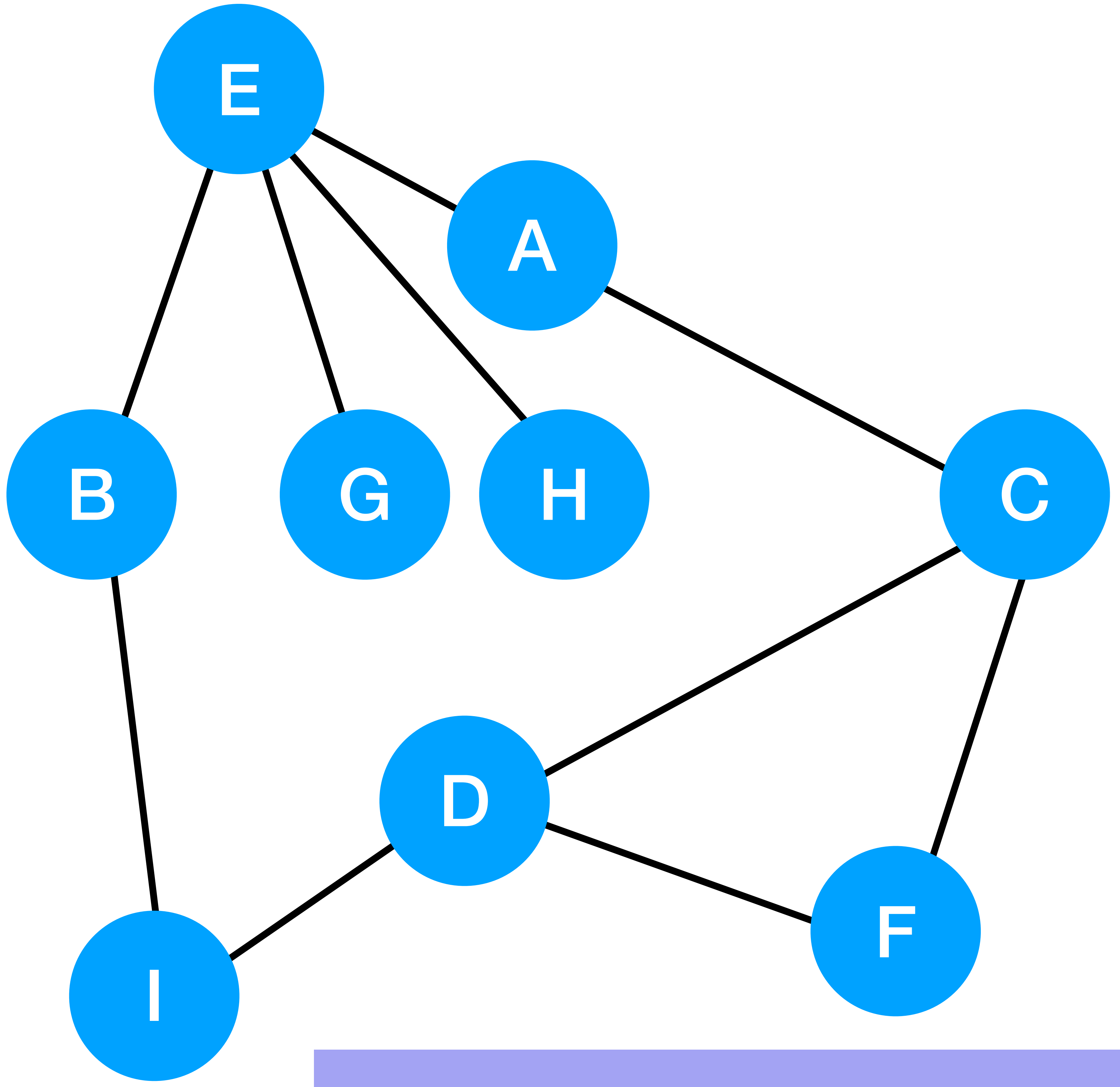
- Store a list of vertices and a list of edges (linked lists or arrays)
- Maintain the vertices
- Maintain the edges
- Pros
  - Simple, only need to store 2 things
- Cons
  - vertex-centric operations run in $O(m)$ time
  - any vertices without edges cannot be stored

A	C
A	E
B	E
B	I
C	D
C	F
D	F
D	I
E	G
E	H

how long to iterate over vertices adjacent to E?

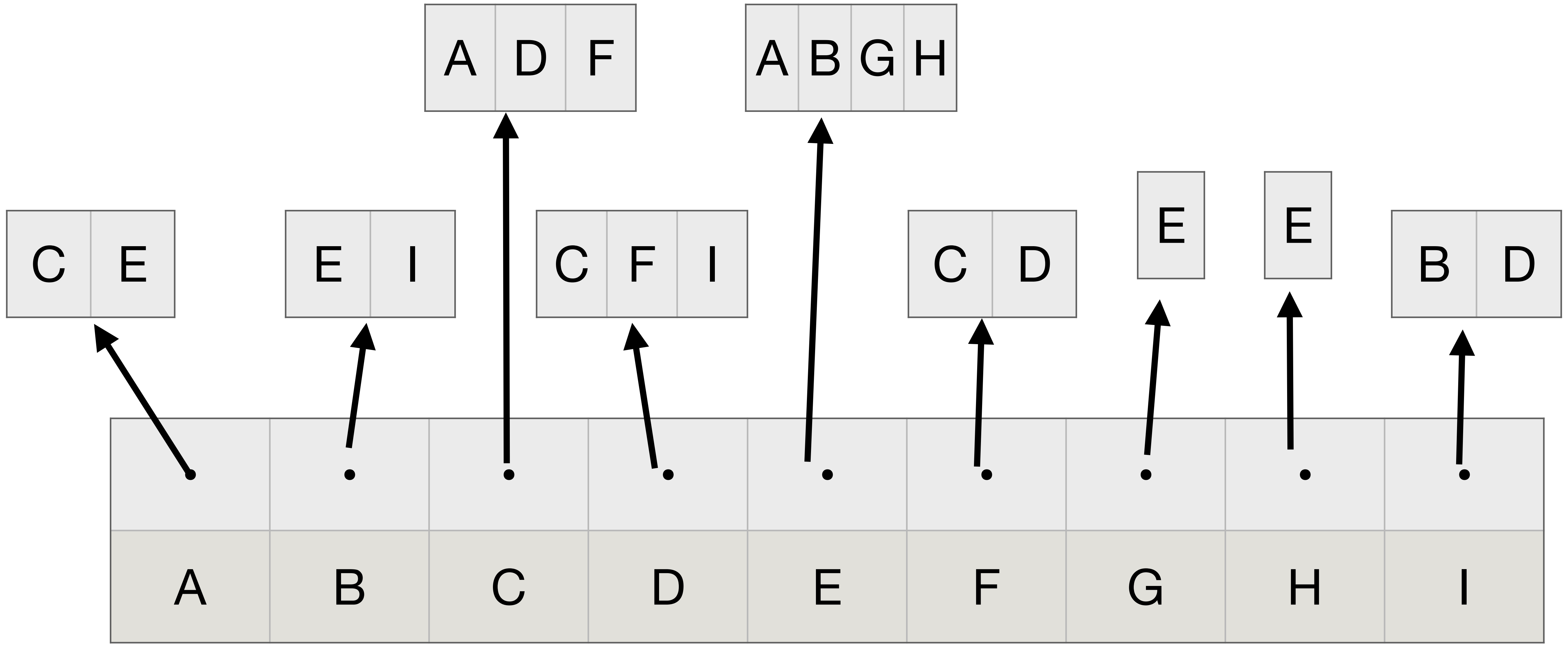
removeVertex(v), areAdjacent(v,w)

Option 2: store a list of vertices

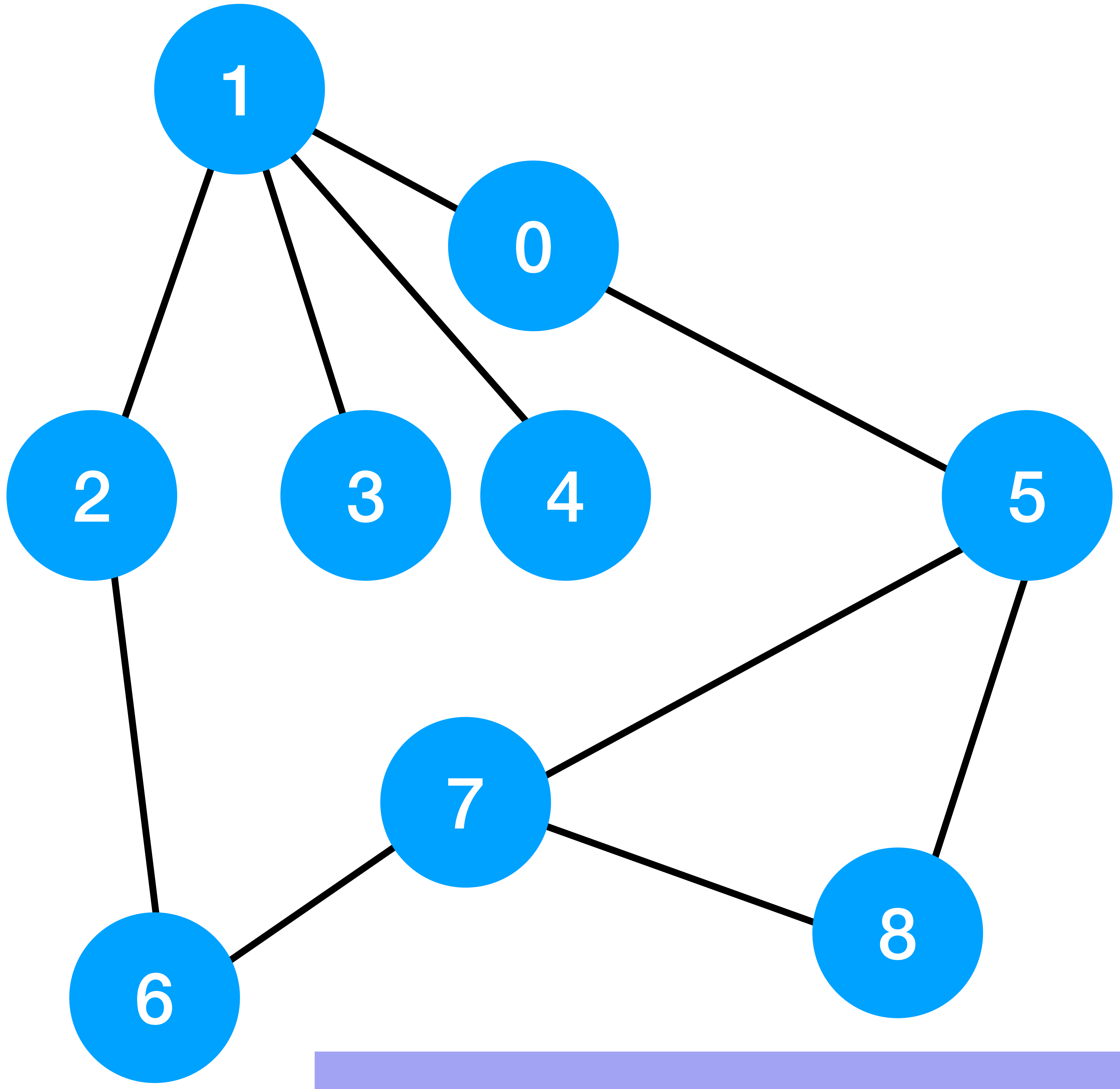


- Store a single list of vertices with links to adjacent edges
- Vertices are mapped to 0, ..., n-1
- Pros
  - Index directly into the desired vertex
  - In sparse graphs, faster than option 1
    - $O(\text{deg}(v))$ time

how long to iterate over vertices adjacent to E?



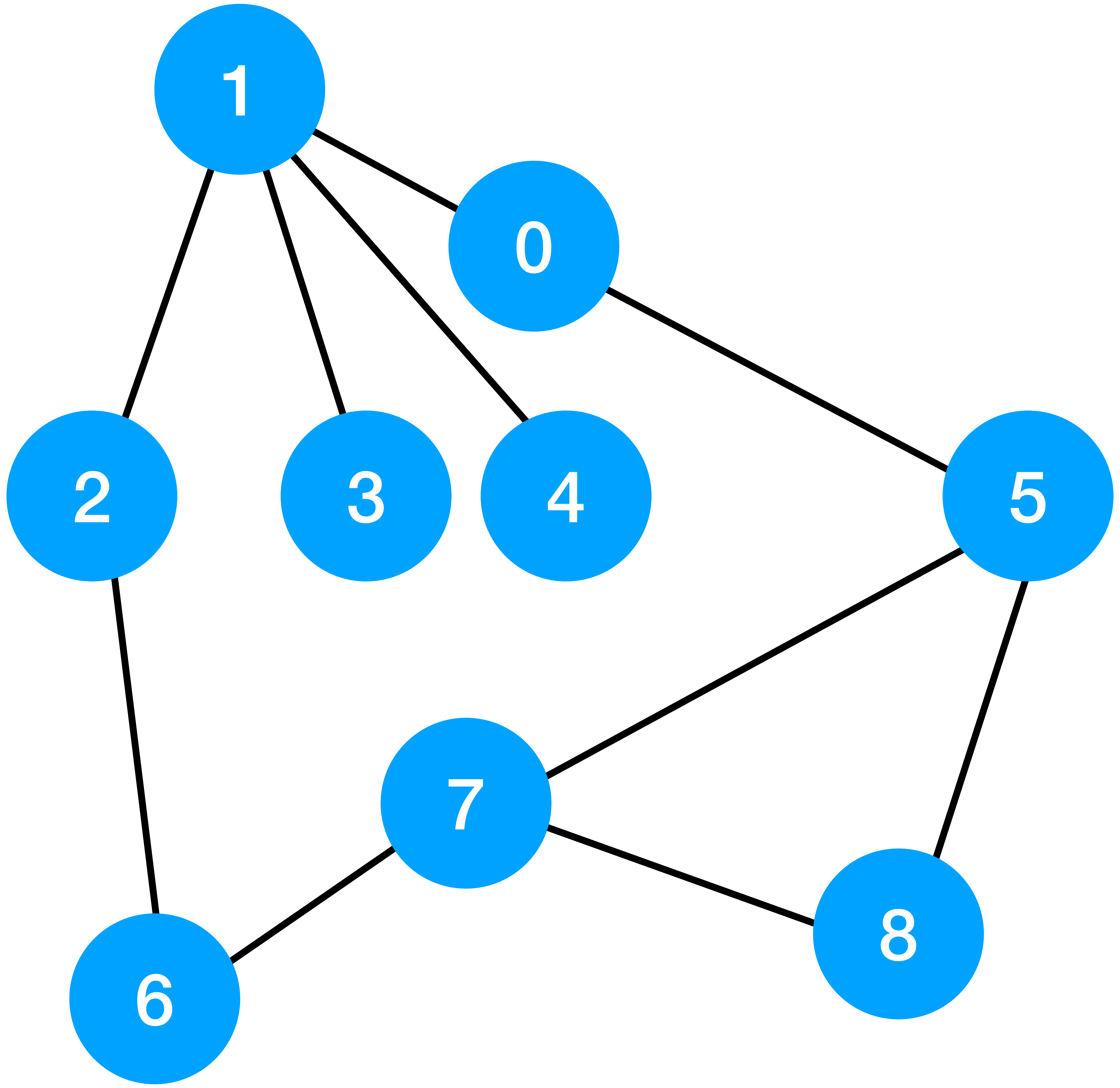
Option 3: adjacency-matrix



- vertices are numbered 0 to n-1
- all edge information stored in a 2D matrix A
- `A[i,j] = 1` if there exists edge $\{i,j\}$ in G
  - `0` otherwise

how long to iterate over vertices adjacent to E?

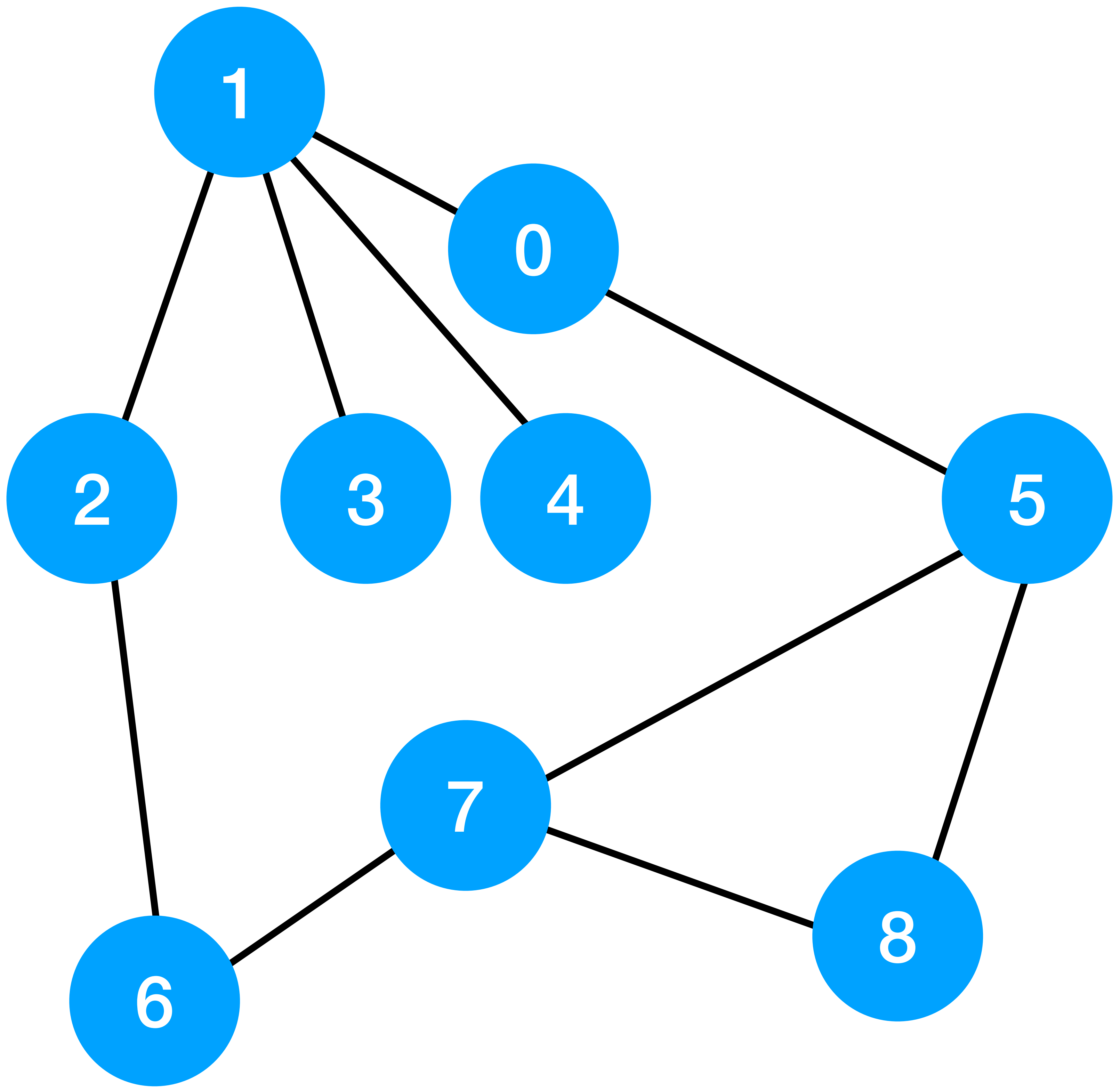
Option 3: adjacency-matrix



how long to iterate over vertices adjacent to E?

	0	1	2	3	4	5	6	7	8
0	0	1	0	0	0	1	0	0	0
1	1	0	1	1	1	0	0	0	0
2	0	1	0	0	0	0	1	0	0
3	0	1	0	0	0	0	0	0	0
4	0	1	0	0	0	0	0	0	0
5	1	0	0	0	0	0	0	1	1
6	0	0	1	0	0	0	0	1	0
7	0	0	0	0	0	1	1	0	1
8	0	0	0	0	0	1	0	1	0

Option 3: adjacency-matrix

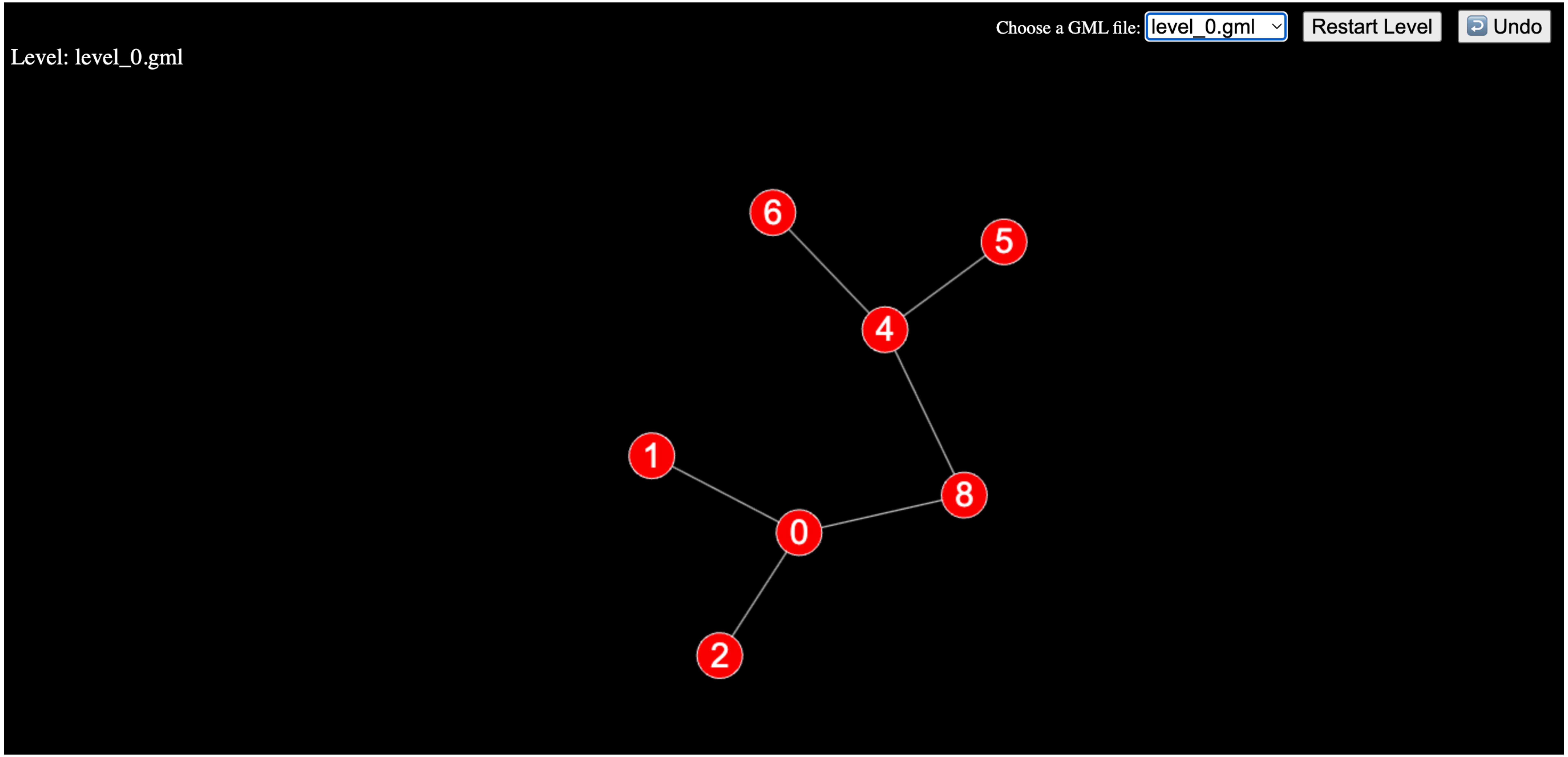


- Useful computations on adjacency matrices
  - `deg(v)` is the sum of column or row v
- `adjacent(v,w)` now runs in $O(1)$ time
- stored in $O(n^2)$ space
- `incidentEdges` and `adjacentVertices` now run in $O(n)$ time (which could often be slower than $O(\text{deg}(v))$)

Asymptotic Performance

• n vertices, m edges • big-Oh time	Edge List	Adjacency List	Adjacency Matrix
Space	n+m	n+m	n^2
incidentEdges(v)	m	deg(v)	n
adjacent(v,w)	m	min(deg(v), deg(w))	1
insertVertex(v,o)	1	1	n^2
insertEdge(v,w)	1	1	1
removeVertex(v)	m	deg(v)	n^2
removeEdge(e)	1	1	1

Catalan Assignment



Demo

How to play Catalan

https://catalan.algochem.techfak.de/game.html

GML Files

- special syntax for describing graphs
- you will need to
  - get the file path from command-line arguments
  - read the file contents
  - construct the graph

```
graph [
    node [ id 0 label "0" ]
    node [ id 1 label "1" ]
    node [ id 2 label "2" ]
    node [ id 3 label "3" ]

    edge [ source 0 target 1 label "-" ]
    edge [ source 0 target 2 label "-" ]
    edge [ source 0 target 3 label "-" ]
]
```

```
graph [
    node [ id 1 label "0" ]
    node [ id 2 label "1" ]
    node [ id 3 label "2" ]
    node [ id 4 label "3" ]
    node [ id 5 label "x" ]
    node [ id 6 label "x" ]
    node [ id 7 label "x" ]
    node [ id 8 label "x" ]
    node [ id 9 label "x" ]
    node [ id 10 label "x" ]
    node [ id 11 label "x" ]
    node [ id 13 label "x" ]
    node [ id 200 label "x" ]

    edge [ source 1 target 2 label "-" ]
    edge [ source 2 target 3 label "-" ]
    edge [ source 2 target 4 label "-" ]
    edge [ source 3 target 5 label "-" ]
    edge [ source 4 target 5 label "-" ]
    edge [ source 5 target 6 label "-" ]
    edge [ source 5 target 7 label "-" ]
    edge [ source 6 target 8 label "-" ]
    edge [ source 7 target 8 label "-" ]
    edge [ source 8 target 9 label "-" ]
    edge [ source 5 target 10 label "-" ]
    edge [ source 5 target 11 label "-" ]
    edge [ source 10 target 13 label "-" ]
    edge [ source 11 target 13 label "-" ]
    edge [ source 13 target 200 label "-" ]
]
```

Your work

- Vertex
- Graph
- Move
- Catalan

Your work

- Vertex
- Graph
- Move
- Catalan

`getID()`: each vertex has an id that is specified in the GML file

```
graph [
    node [ id 0 label "0" ]
    node [ id 1 label "1" ]
    node [ id 2 label "2" ]
    node [ id 3 label "3" ]

    edge [ source 0 target 1 label "-" ]
    edge [ source 0 target 2 label "-" ]
    edge [ source 0 target 3 label "-" ]
]
```

Your work

- Vertex
- Graph
- Move
- Catalan

`readGraphFromFile(String filepath)`

`numVertices()`

`areNeighbours(Vertex u, Vertex v)`

default constructor must work

Your work

- Vertex
- Graph
- Move
- Catalan

`getVertices()`

returns an

`ArrayList<Vertex>`

Your work

- Vertex
- Graph
- Move
- Catalan

`getVertices()`

returns an
ArrayList<Vertex>

"couldn't we just use a
Vertex[]?"

Your work

- Vertex
- Graph
- Move
- Catalan

getVertices()

returns an
ArrayList<Vertex>

"couldn't we just use a Vertex[]?"

yes, but ArrayList<Vertex> is very useful to know

Your work

- Vertex
- Graph
- Move
- Catalan

`getVertices()`

returns an `ArrayList<Vertex>`

"couldn't we just use a `Vertex[]`?"

yes, but `ArrayList<Vertex>` is very useful to know

import java.util.ArrayList to use!

Your work

- Vertex
- Graph
- Move
- Catalan

`getNeighbours(Vertex u)`

returns an
`ArrayList<Vertex>`

"couldn't we just use a `Vertex[]`?"

yes, but `ArrayList<Vertex>` is very useful to know

import java.util.ArrayList to use!

Your work

- Vertex
- Graph
- Move
- Catalan

```
collapseNeighbours(Vertex
u)
```

collapses all the neighbours of u.

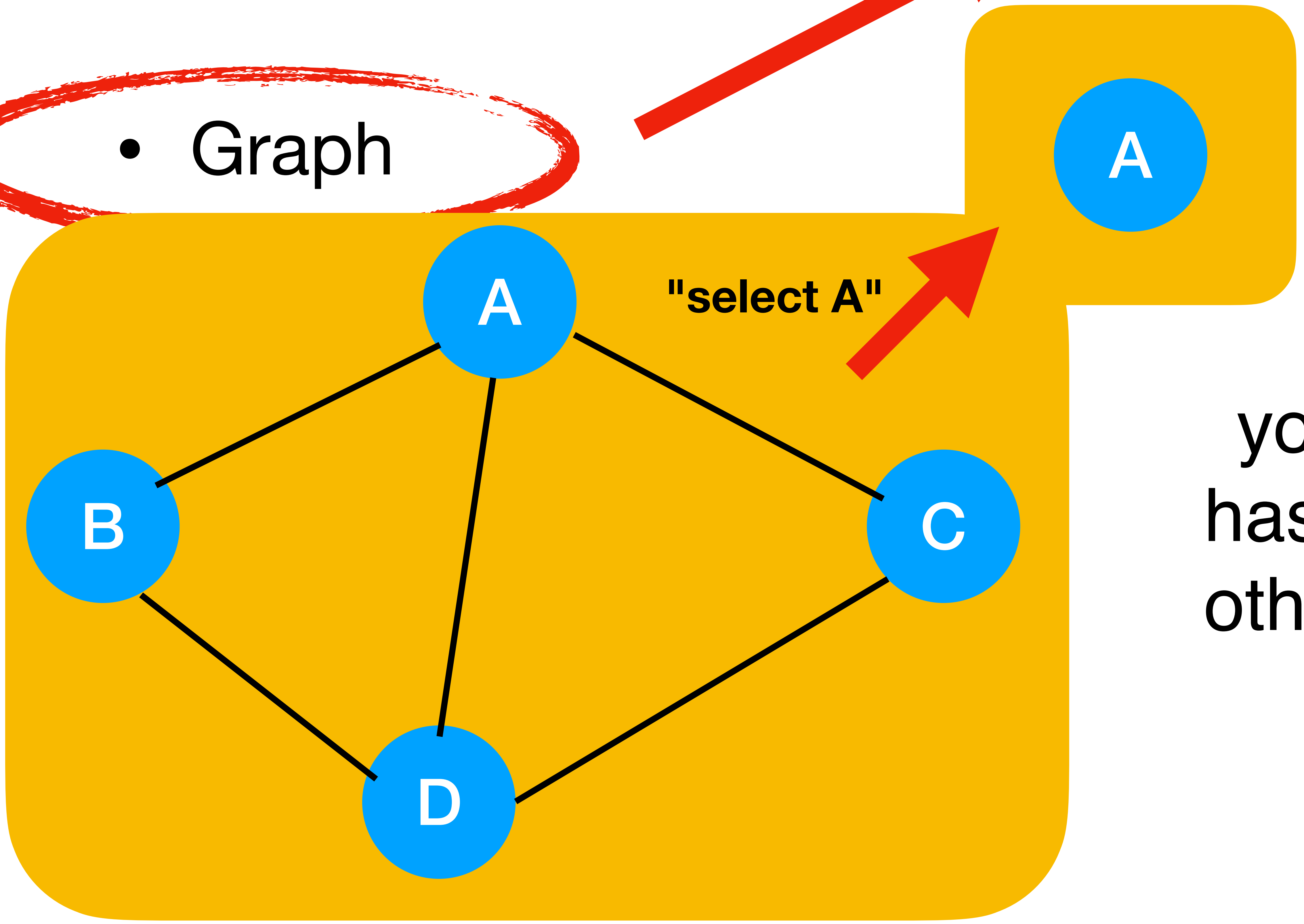
you can only do this if u has exactly 3 neighbours! otherwise, nothing should happen.

MUST MAKE A COPY!!!!!

Your work

```
collapseNeighbours(Vertex u)
```

- Vertex
- Graph

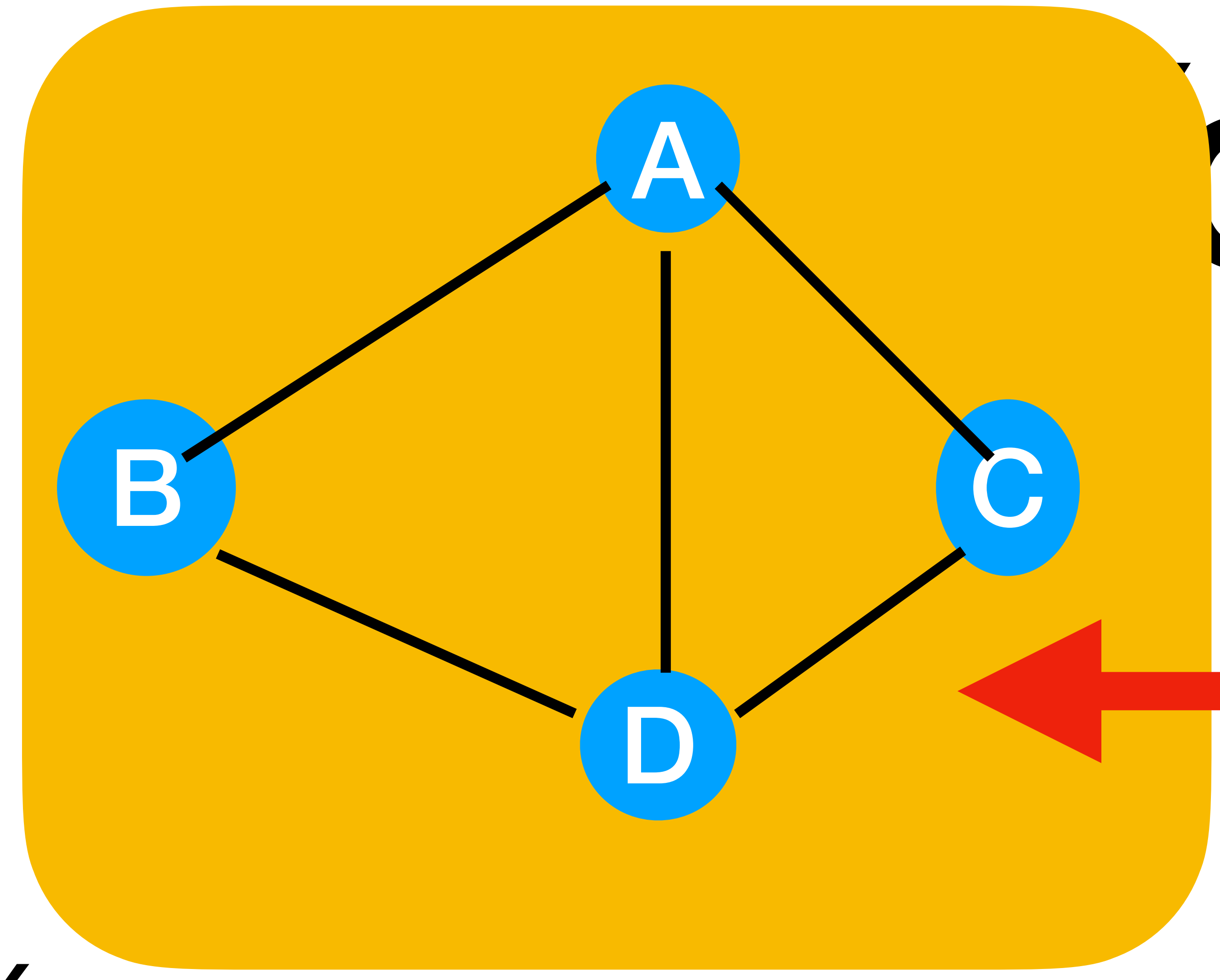


collapses all the neighbours of u.

you can only do this if u has exactly 3 neighbours! otherwise, nothing should happen.

MUST MAKE A COPY!!!!!

our work



- Vertex
- Graph
- Move
- Catalan

`getBeforeState()`

`getAfterState()`

`getSelectedVertex()`

"select A"

Your work

- Vertex
- Graph
- Move
- Catalan

`toString()`

must override the default `toString()`; again, useful to know how to do

Your work

- Vertex
- Graph
- Move
- Catalan

```
public Catalan(String
filepath)
```

a constructor which takes
in the filepath to a GML file

Your work

- Vertex
- Graph
- Move
- Catalan

`solve()`

the culmination of this assignment!!!
use all of the previous classes you just wrote to solve the game in the **fewest** moves possible.

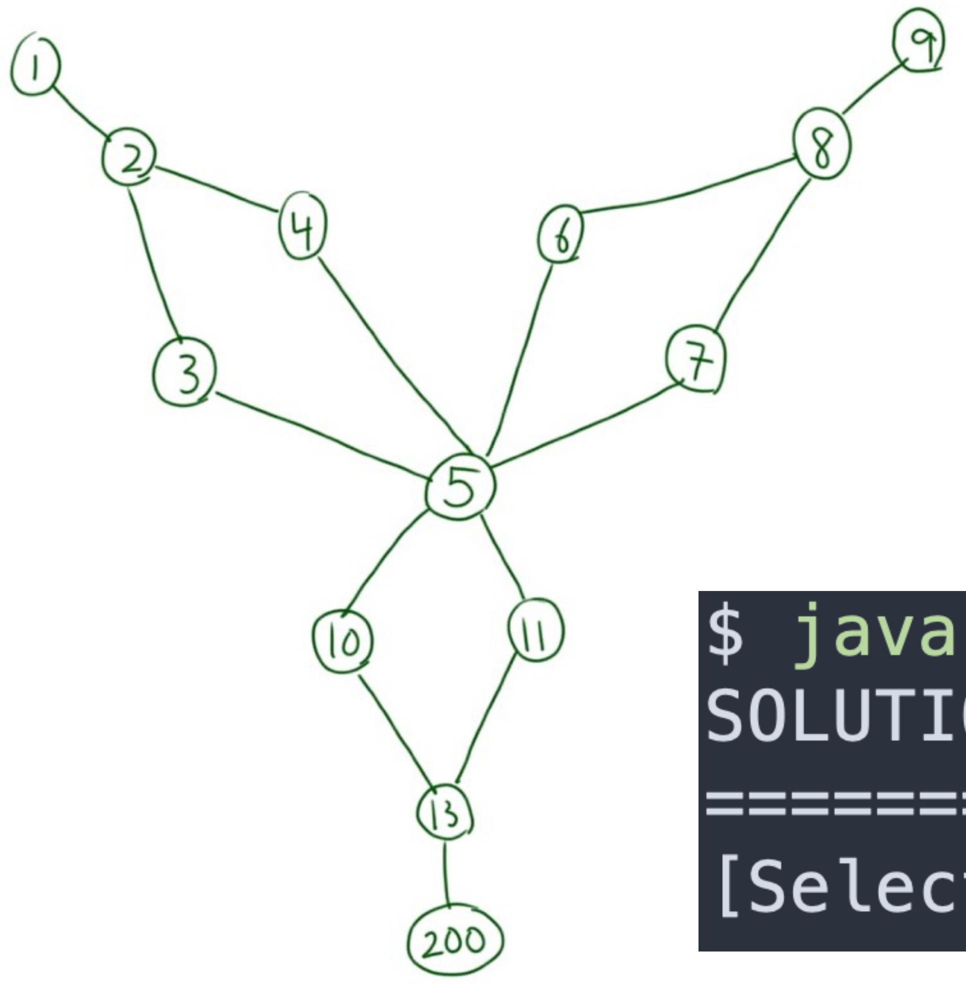
returns an ordered list of
`ArrayList<Move>`

Your work

- Vertex
- Graph
- Move
- Catalan

```
public static void main
(String[] args)
```

this file will also house the main method.




```
$ java Catalan ./solveable/graph2.gml
SOLUTION
===========
[Select vertex 13, Select vertex 8, Select vertex 2, Select vertex 5]
```

Your work

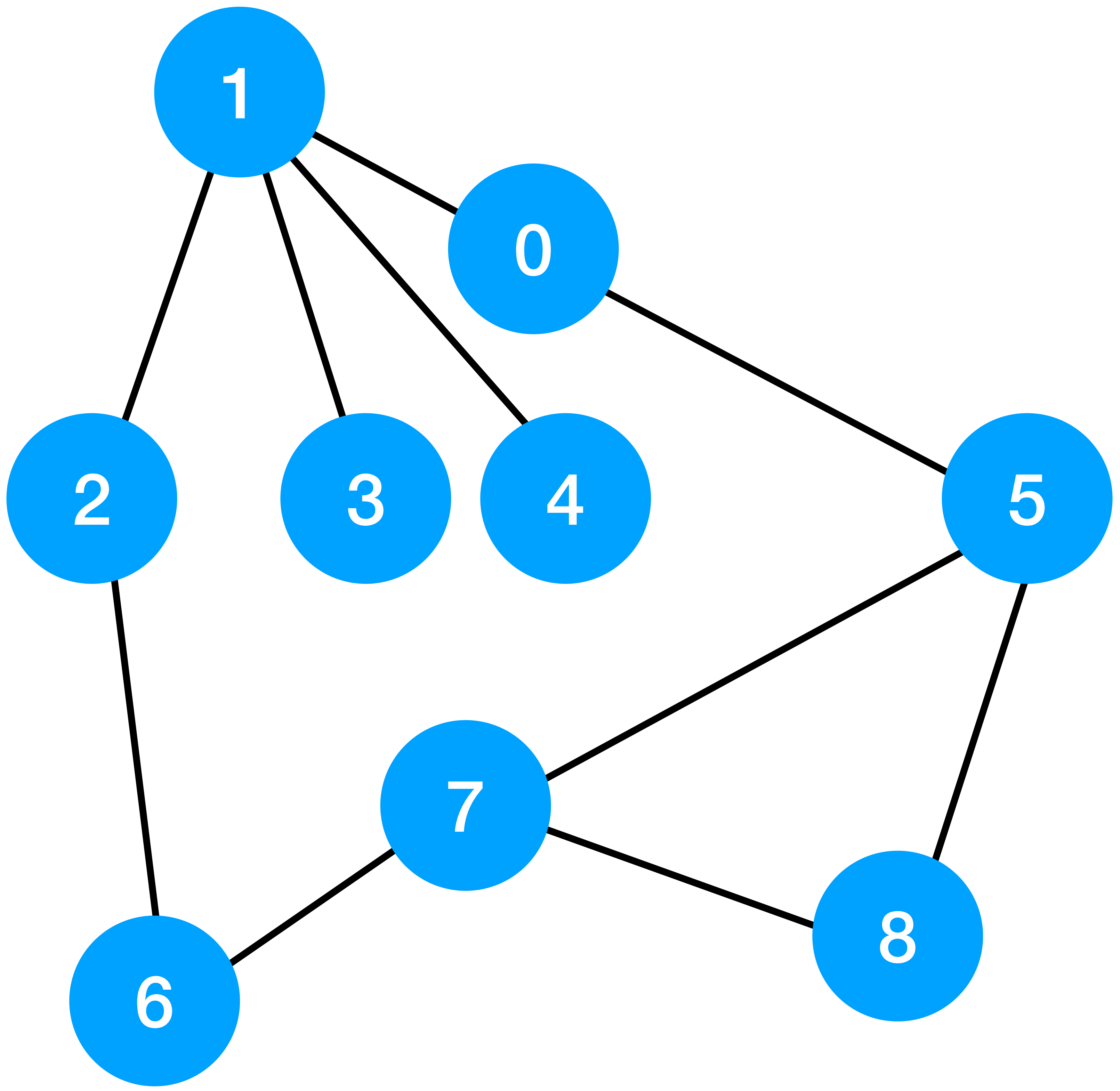
- Vertex
- Graph
- Move
- Catalan

```
public static void main
(String[] args)
```

DEMO: running your program from command-line and program output

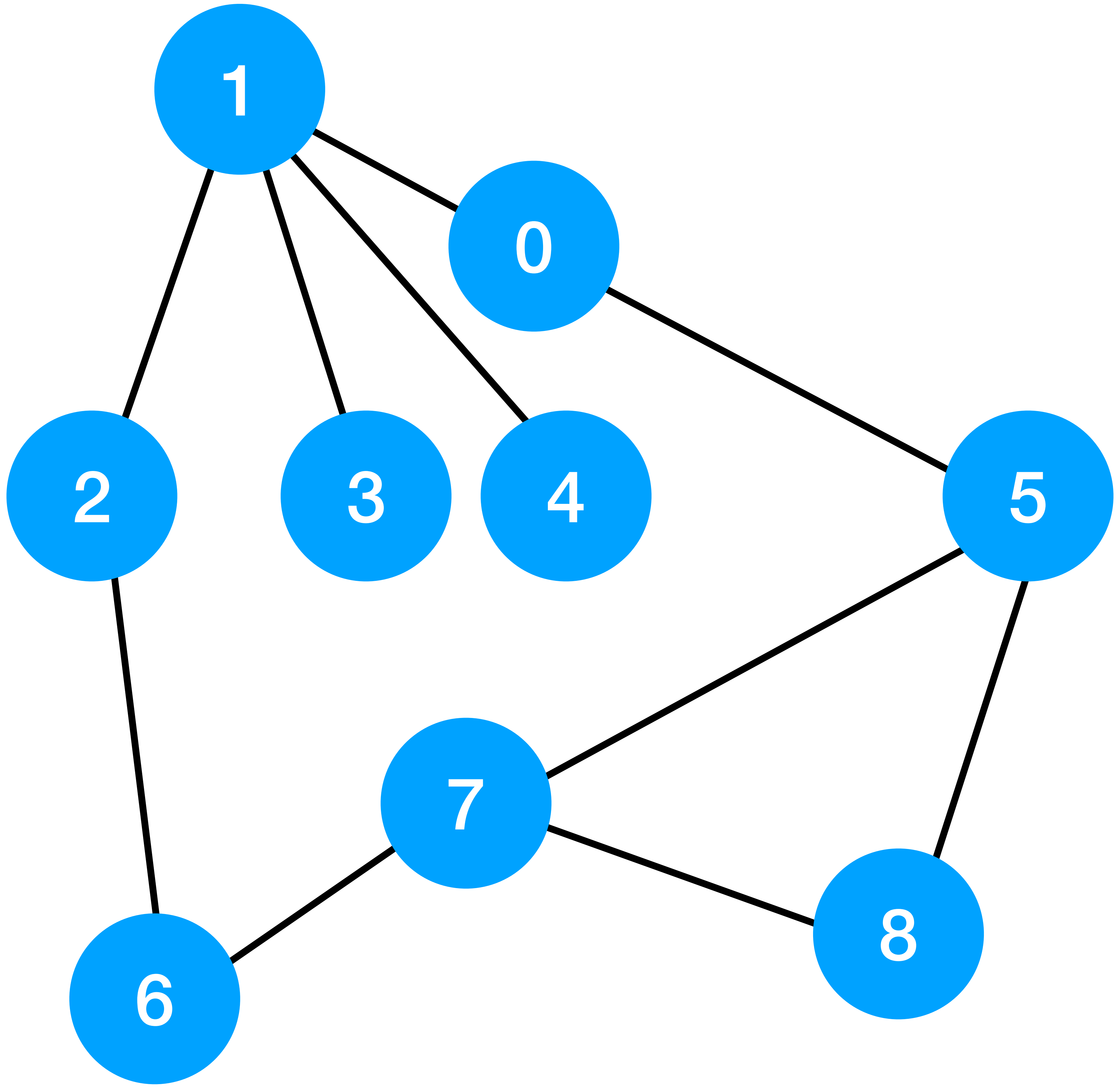
```
$ java Catalan ./solveable/graph2.gml
SOLUTION
===========
[Select vertex 13, Select vertex 8, Select vertex 2, Select vertex 5]
```

Graph Traversals



- Similar to tree traversals, but for graphs
  - Systematically "explore" every vertex in the graph
- Two main flavours
  - Depth-First-Search (DFS)
    - iterative, using a stack
    - recursively, using call stack
  - Breadth-First-Search (BFS)
    - iterative, using a queue

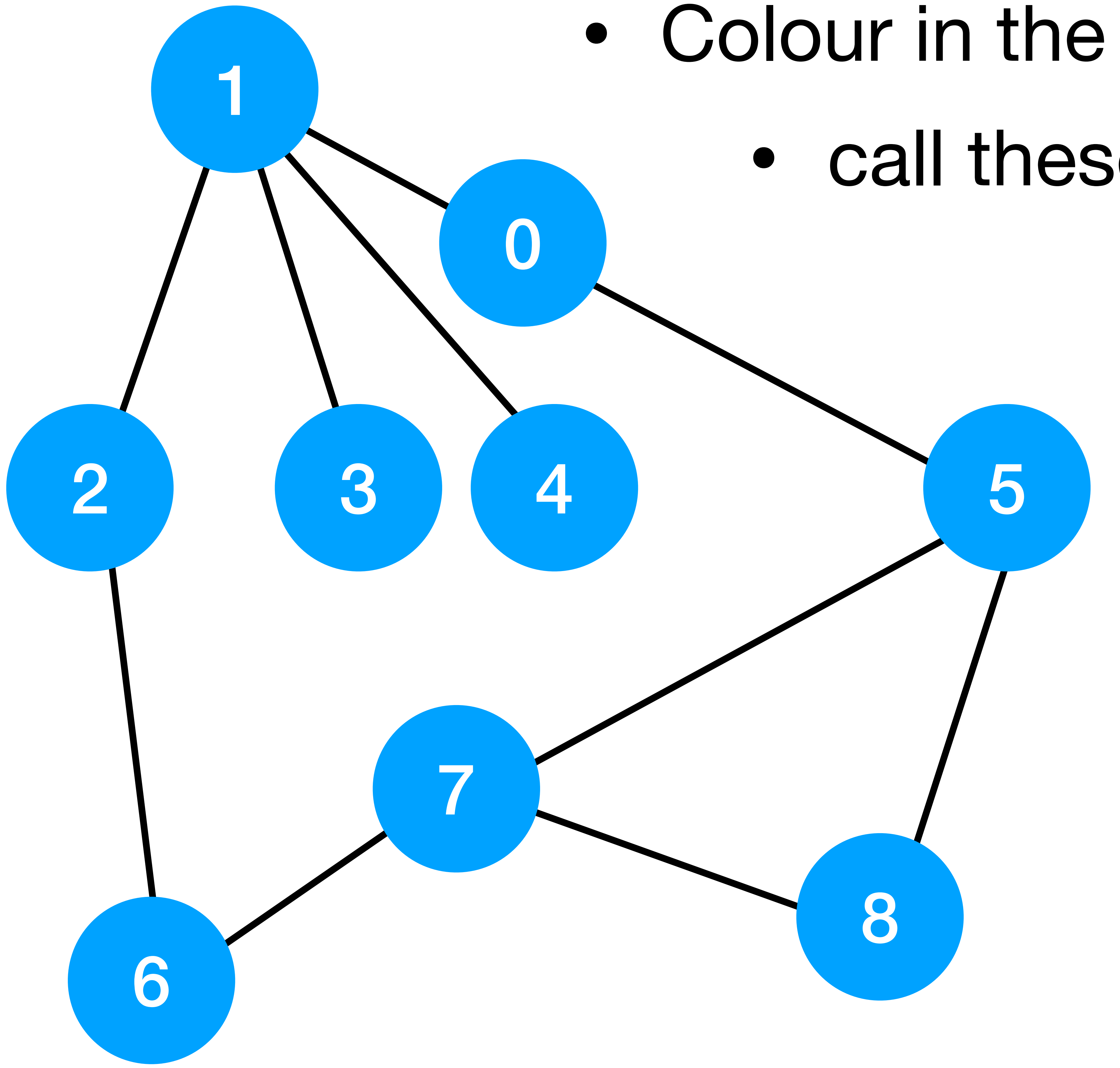
Depth-First-Search



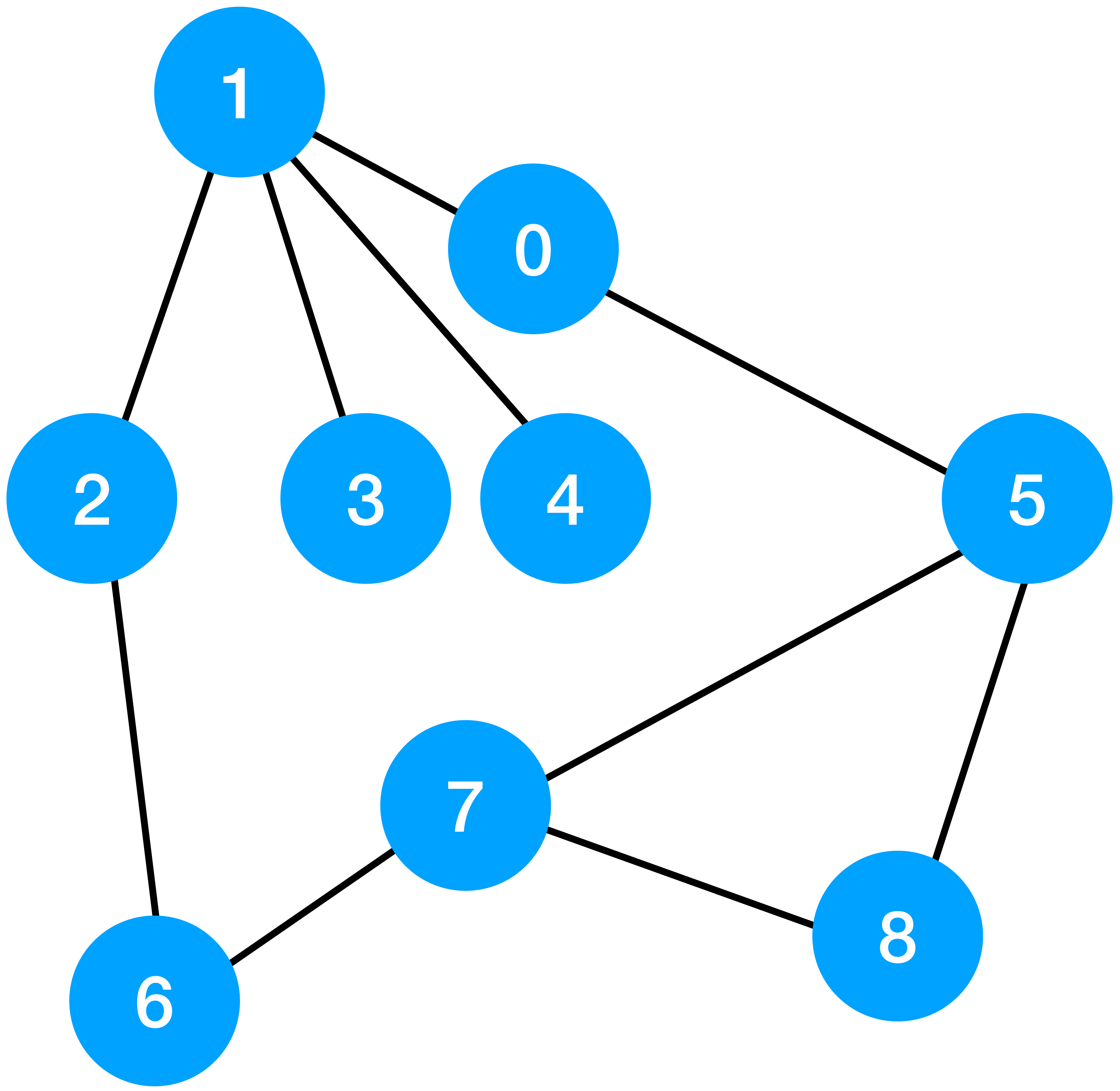
- Start at some vertex v and mark v as visited
- Add all unvisited neighbours of v to stack
- Visit next vertex on the stack and add all of its neighbours
  - repeat
- If a node has no unvisited neighbours, backtrack (pop off the stack)

Example: run DFS on G

- Start at vertex 5 and when add neighbours in order from smallest to largest
- Colour in the edges that DFS takes
  - call these coloured edges: "discovery edges"

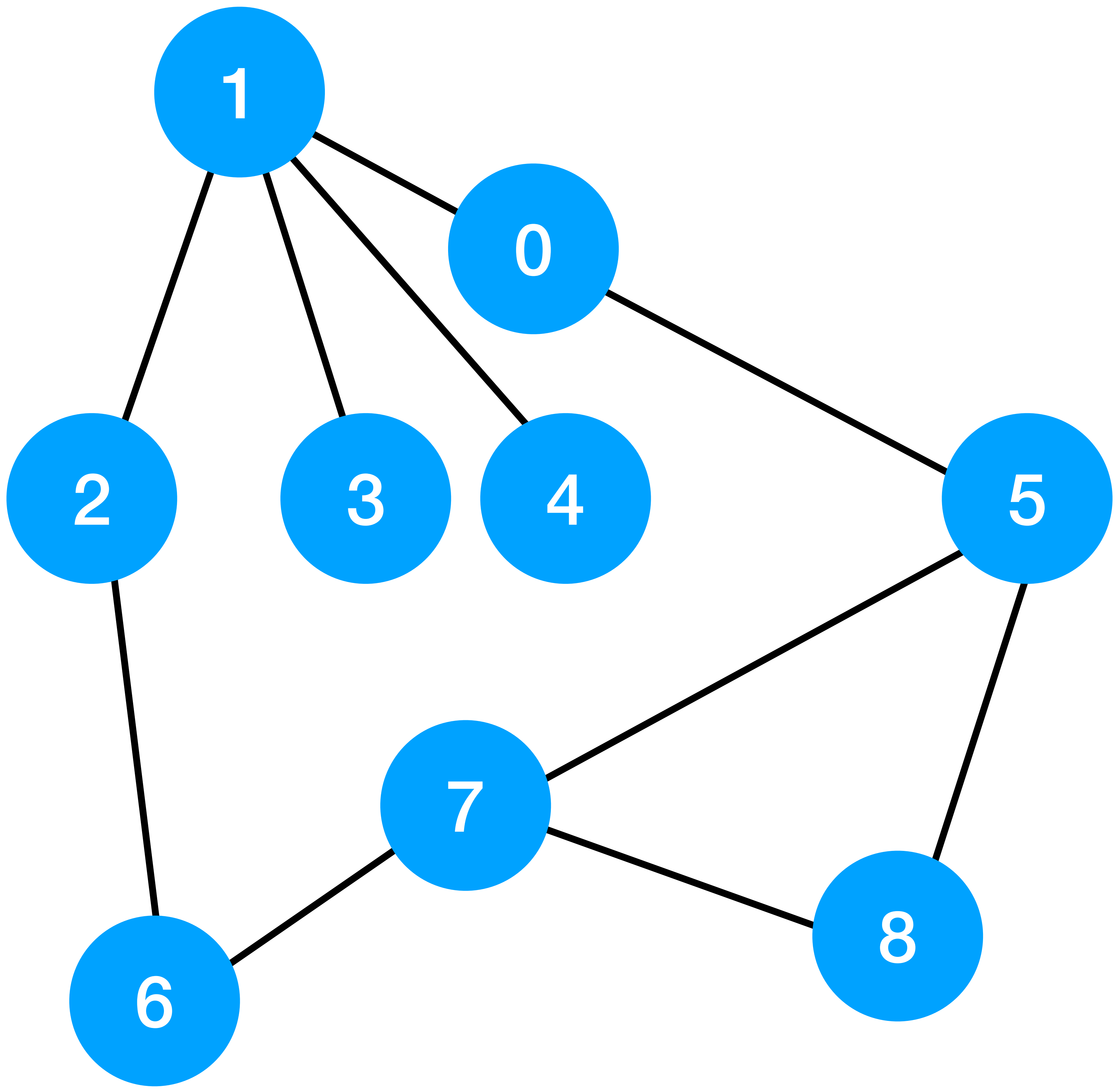


Breadth-First-Search



- Run BFS starting at vertex 3.
  - shade in the edges
  - when choosing neighbours, visit smallest to largest

BFS Properties



- BFS(G,v) visits all vertices in the connected component containing v
- The discovery edges create a spanning tree in the connected component containing v
- BFS finds the shortest path from v to any other vertex in the tree
- Each vertex is visited exactly once
- Each edge is visited at least once and at most twice
- **Theorem:** The time complexity for BFS is $O(n+m)$ where $|V|=n$ and $|E|=m$

Questions?