Graph Search

Sides based on material by Kani, Chekuri, Erickson et. al.

All mistakes are my own! - Ivan Abraham (Fall 2024)

Image by ChatGPT (probably collaborated with DALL-E)

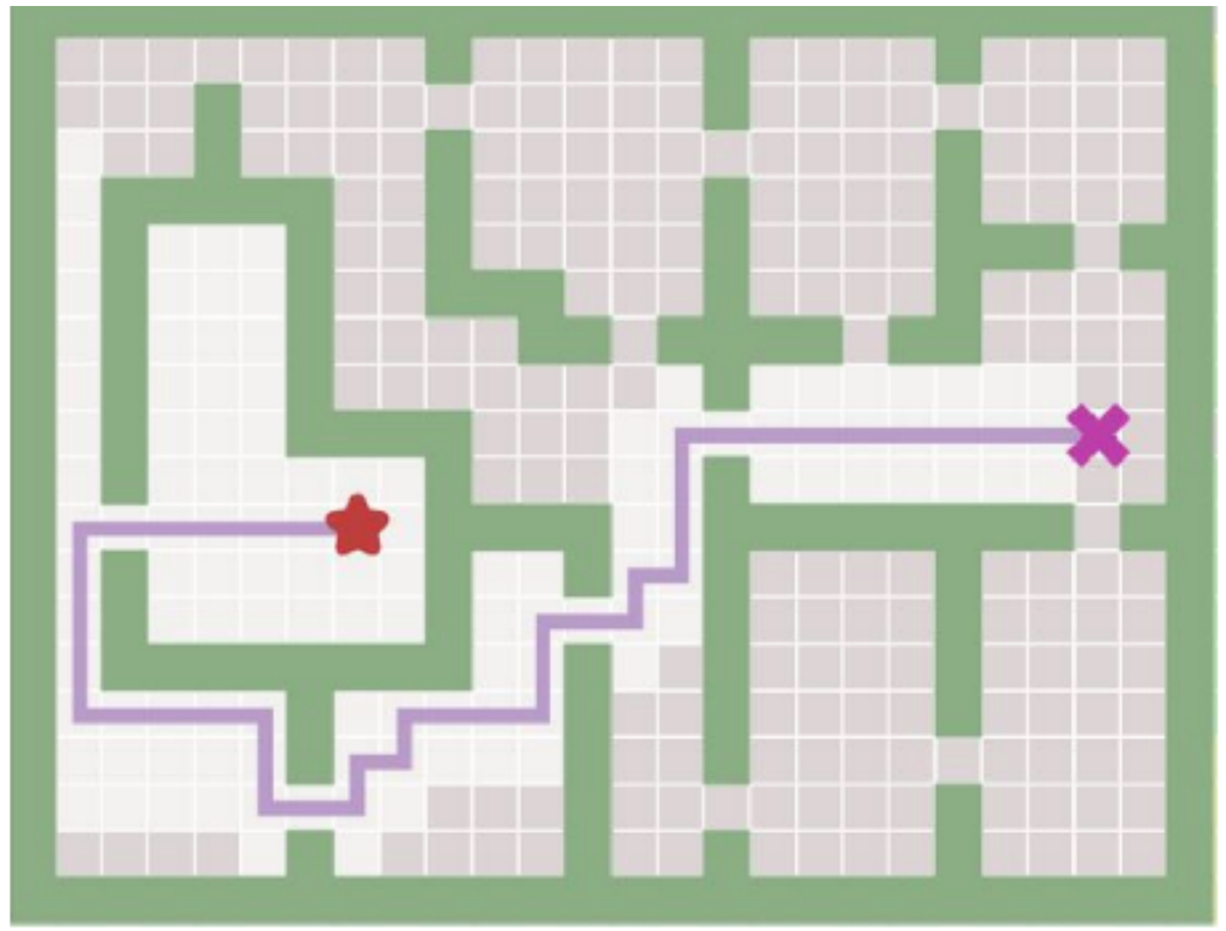
Why graphs?

- Graphs have **many applications**!
  - Graphs help model *networks* — which are ubiquitous: transportation networks (rail, roads, airways), social networks (interpersonal relationships), information networks (web page links), and many problems that don't even look like graph problems.
- Fundamental objects in CS, optimization, combinatorics
- Many important and useful optimization problems are graph problems
- Graph theory: elegant, fun and deep branch of mathematics

Why graphs?

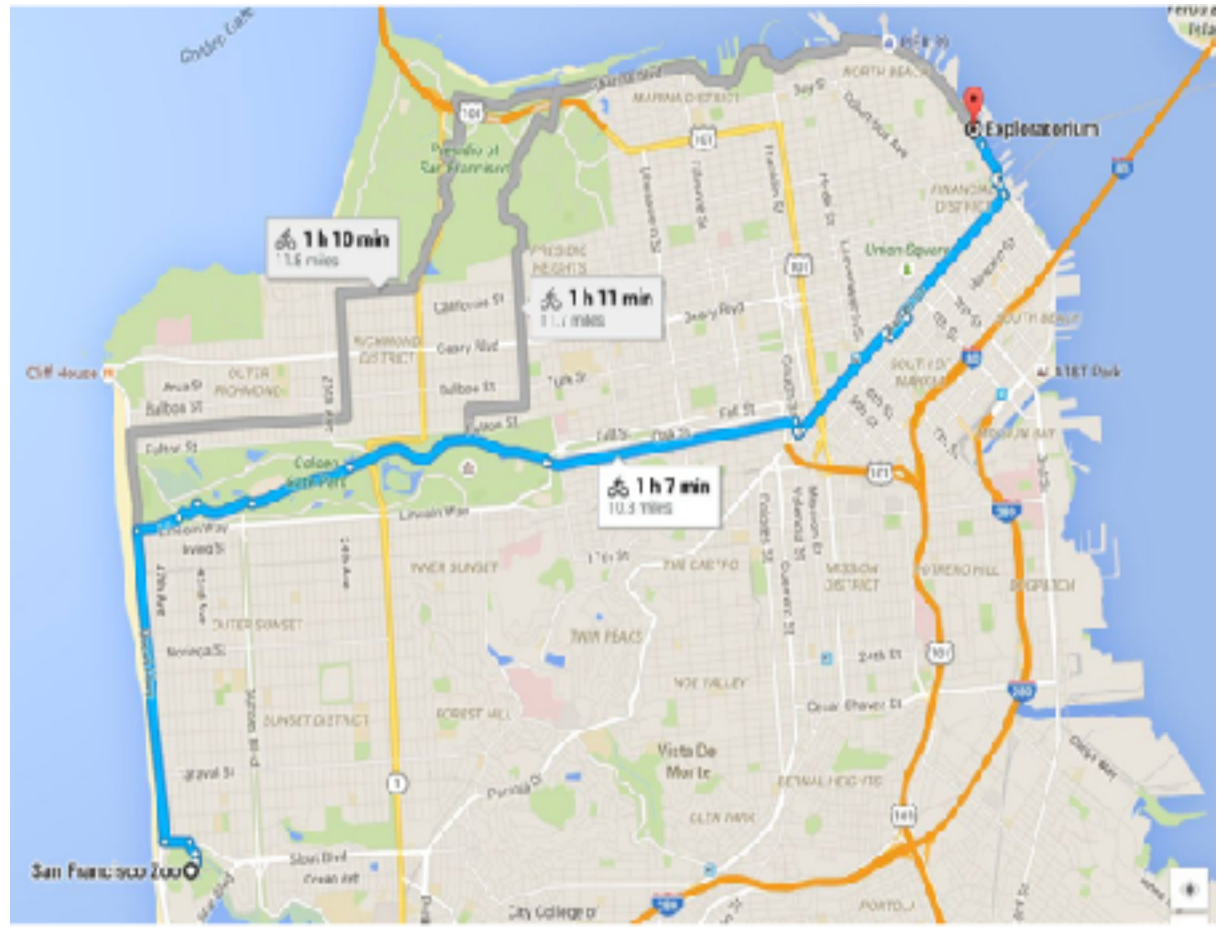
Real life applications

Shortest Path

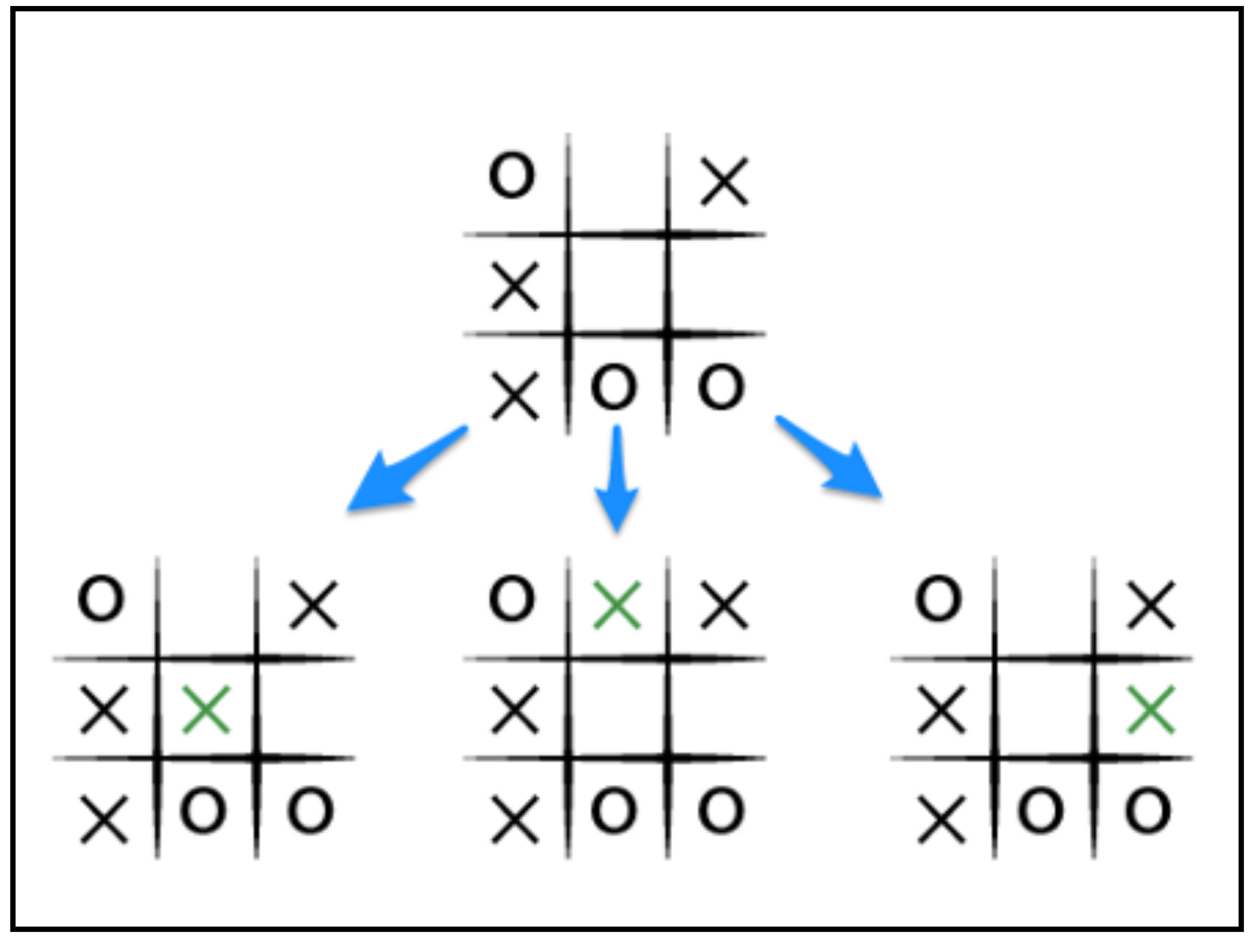


Search & Rescue

Route Planning



Game Playing

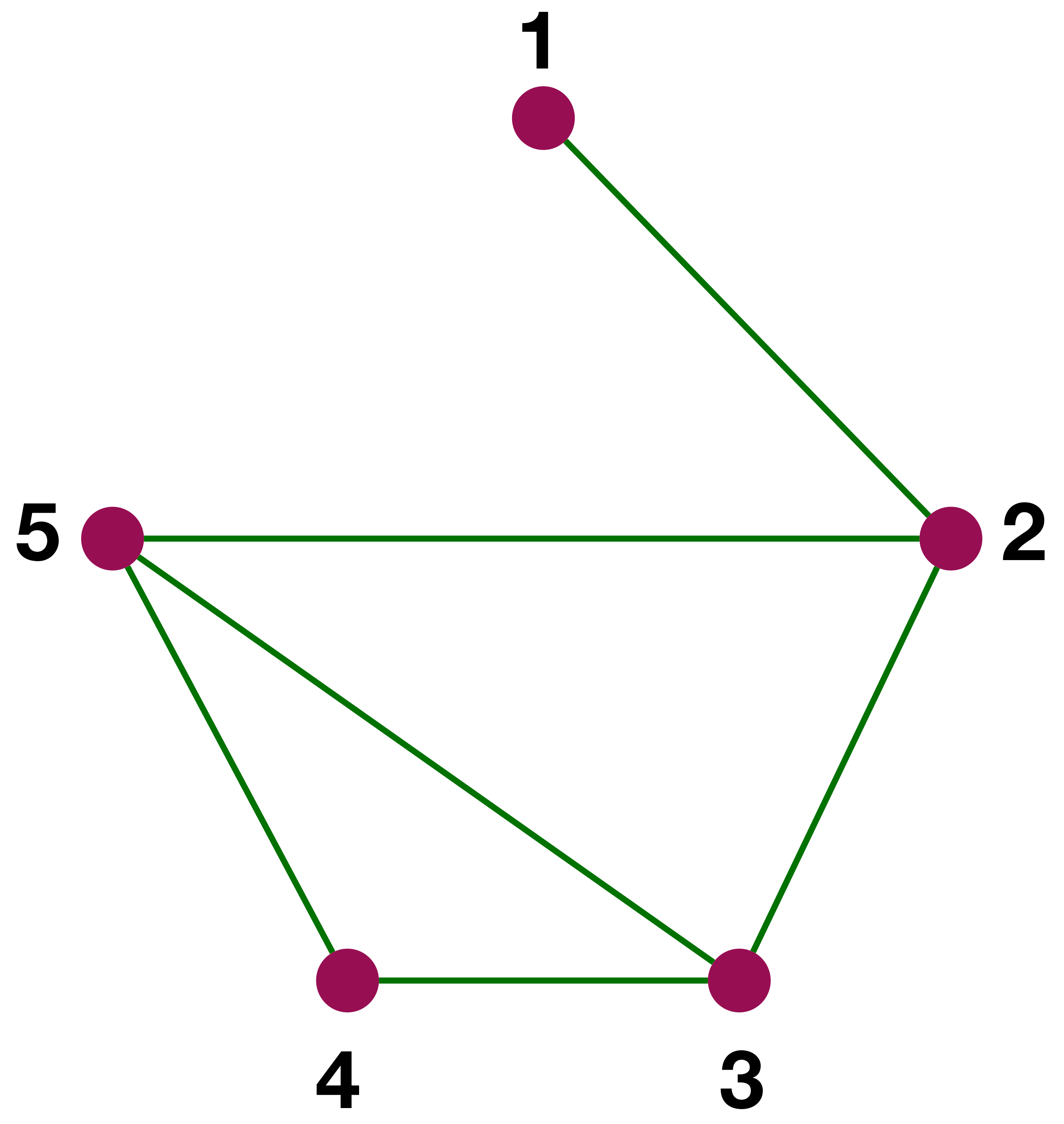


Introduction

What is a Graph?

- A graph is a collection of **nodes** and **edges**.
- The dots are called ***vertices*** or ***nodes***.
- The *connections* between nodes are called ***edges***
- An edge typically represented as a *set* $\{i, j\}$ of two vertices.

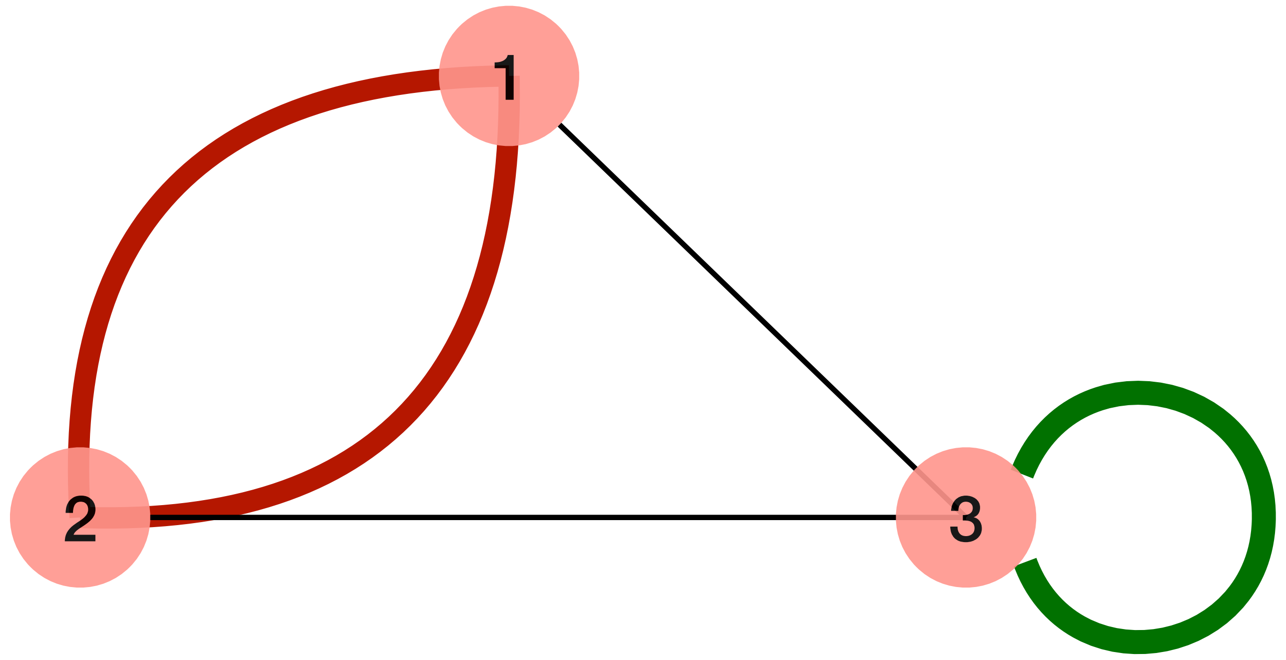
Eg: The edge between **2** and **5** is $\{2,5\} = \{5,2\}$



Notational convention

What is a Graph?

An edge in an undirected graph is an *unordered pair* of nodes and hence it is a set. We reserve the use of (u, v) (ordered pair) for the case of *directed* graphs.



- Generalizations
  - *Multi-graphs* allow
    - ***loops*** which are edges with the same node appearing as both end points
    - ***multi-edges***: *different* edges between same pairs of nodes
- In this class we will assume that a graph is a *simple graph* unless explicitly stated otherwise.

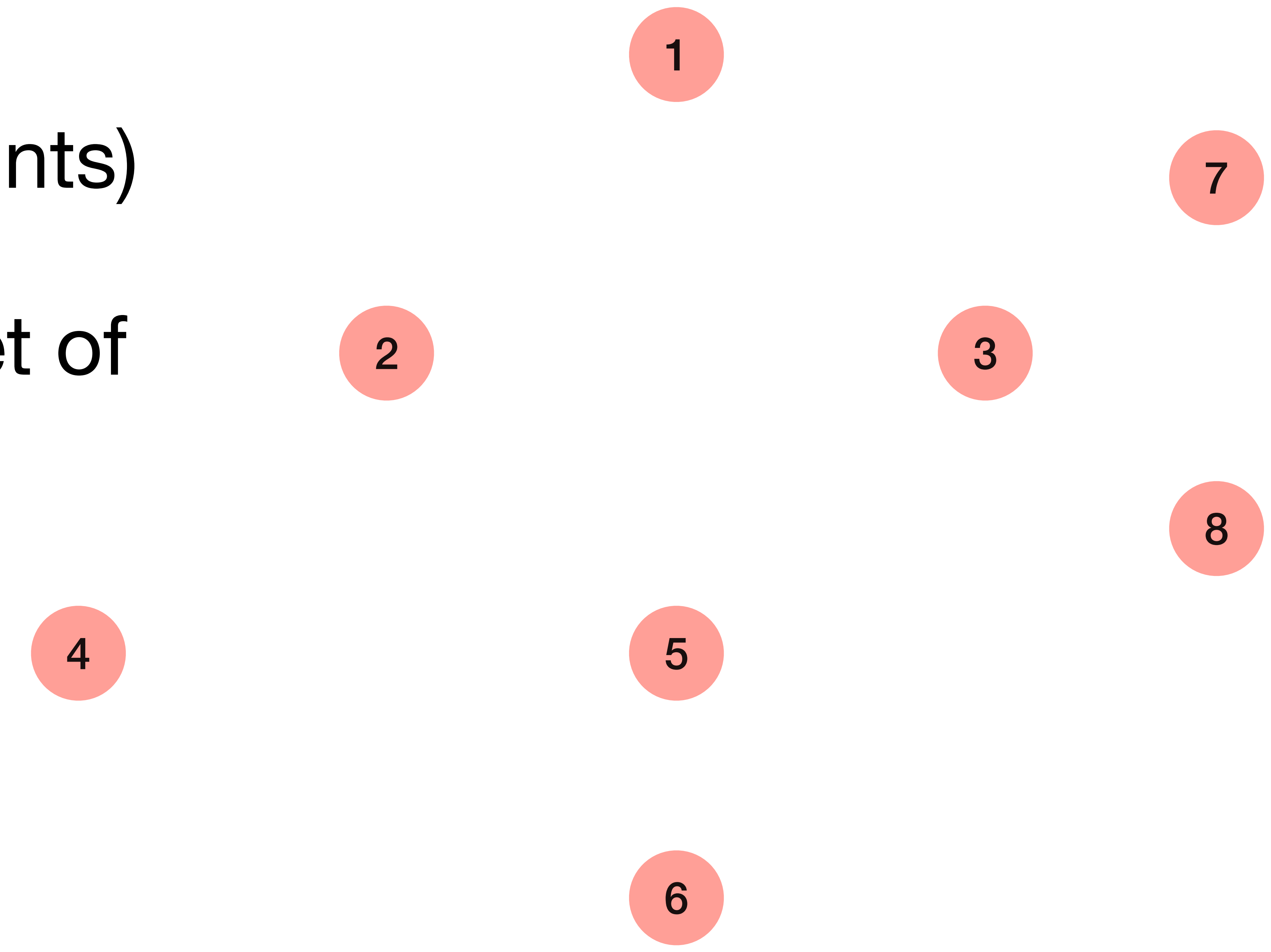
Introduction

Defintion

An undirected (simple) graph $G = (V, E)$ is a 2-tuple:

- V is a set of vertices (also referred to as nodes/points)
- E is a set of edges where each edge $e \in E$ is a set of the form $\{u, v\}$ with $u, v \in V$ and $u \neq v$.

Example:



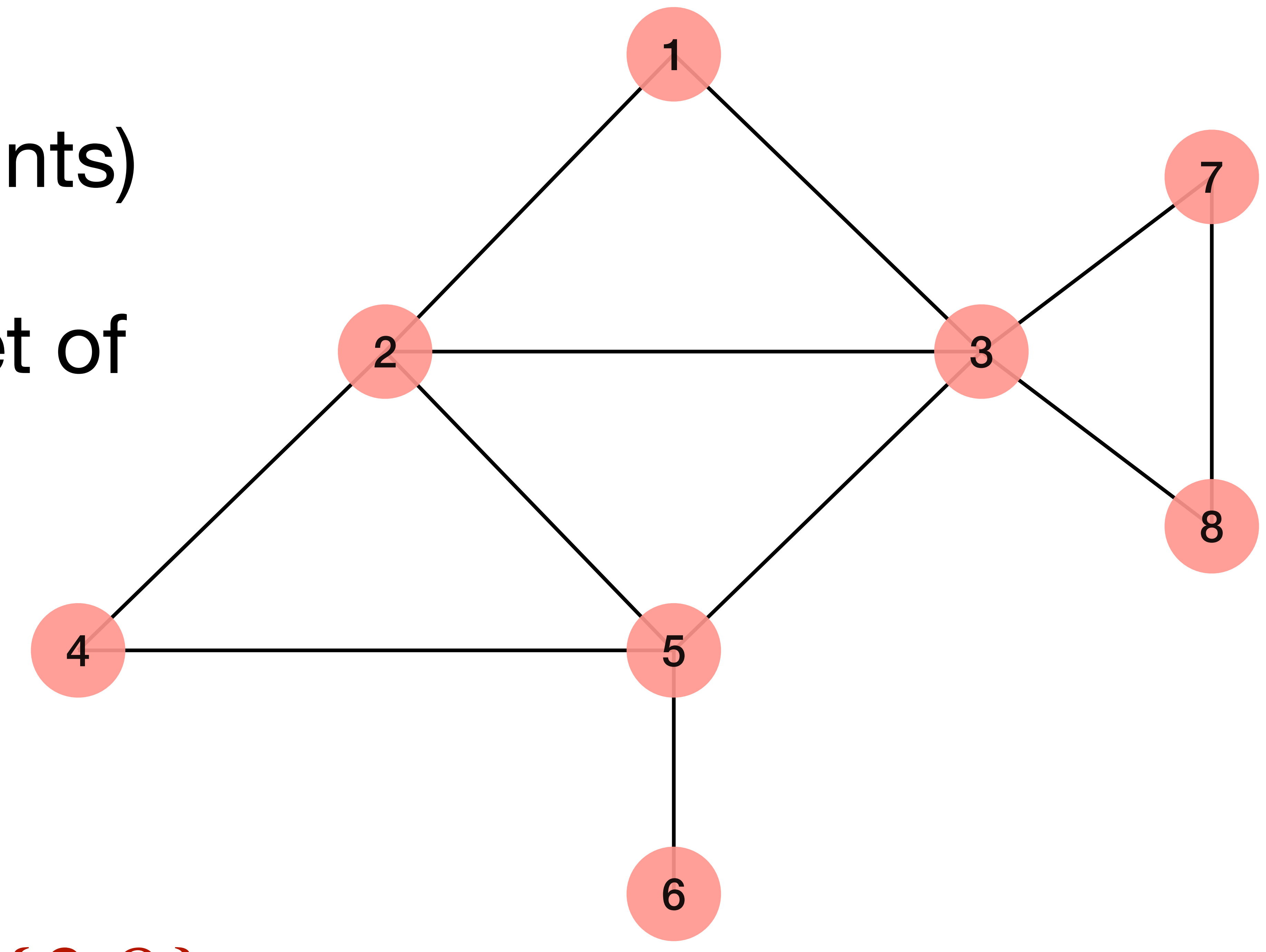
Graph $G = (V, E)$ where $V = \{1,2,3,4,5,6,7,8\}$ and

Introduction

Defintion

An undirected (simple) graph $G = (V, E)$ is a 2-tuple:

- V is a set of vertices (also referred to as nodes/points)
- E is a set of edges where each edge $e \in E$ is a set of the form $\{u, v\}$ with $u, v \in V$ and $u \neq v$.



Example:

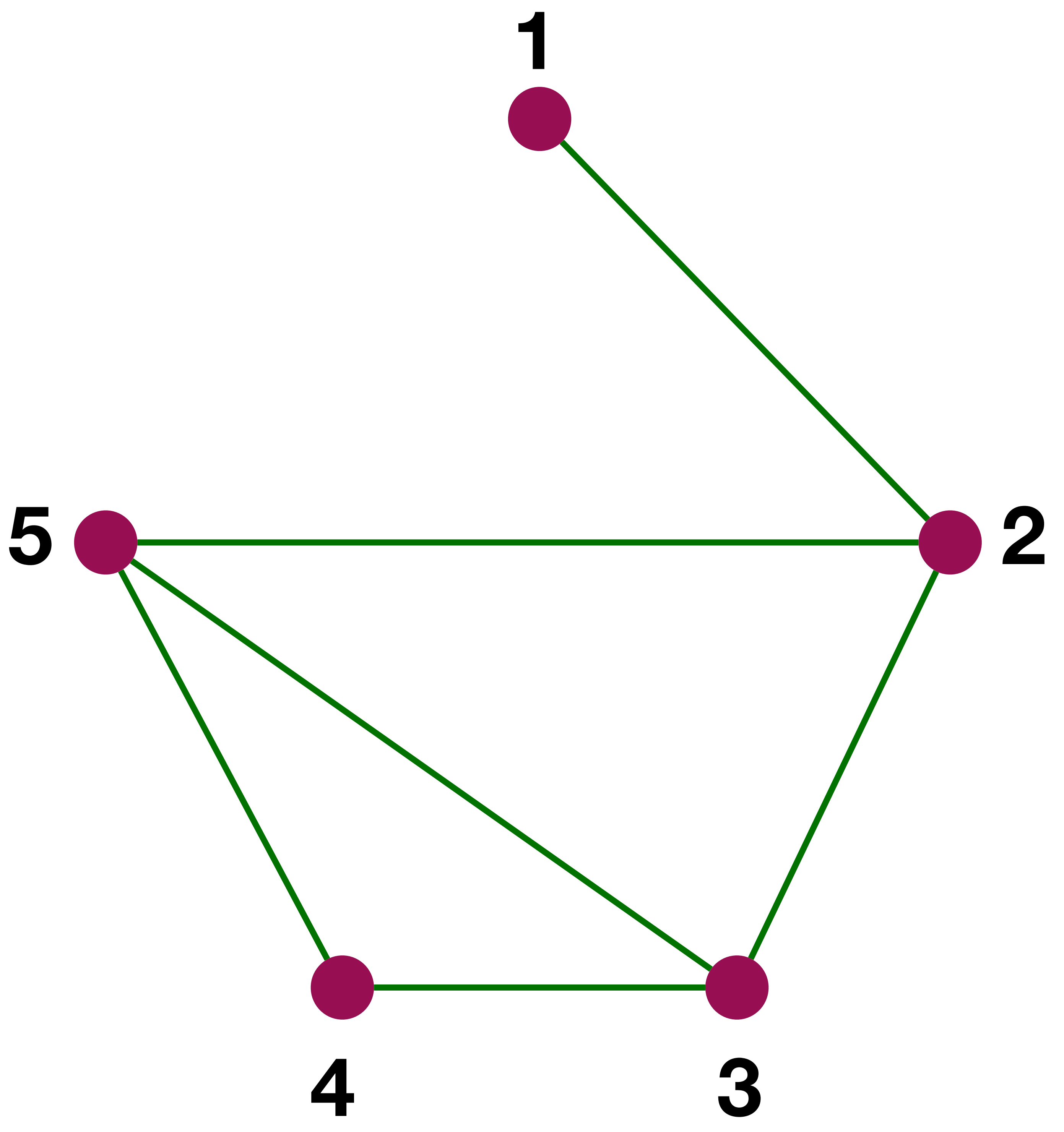
Graph $G = (V, E)$ where $V = \{1,2,3,4,5,6,7,8\}$ and

$$E = \{\{1,2\}, \{1,3\}, \{2,3\}, \{2,4\}, \{2,5\}, \{3,5\}, \{3,7\}, \{3,8\}, \{4,5\}, \{5,6\}, \{7,8\}\}$$

Basic notions

Degree

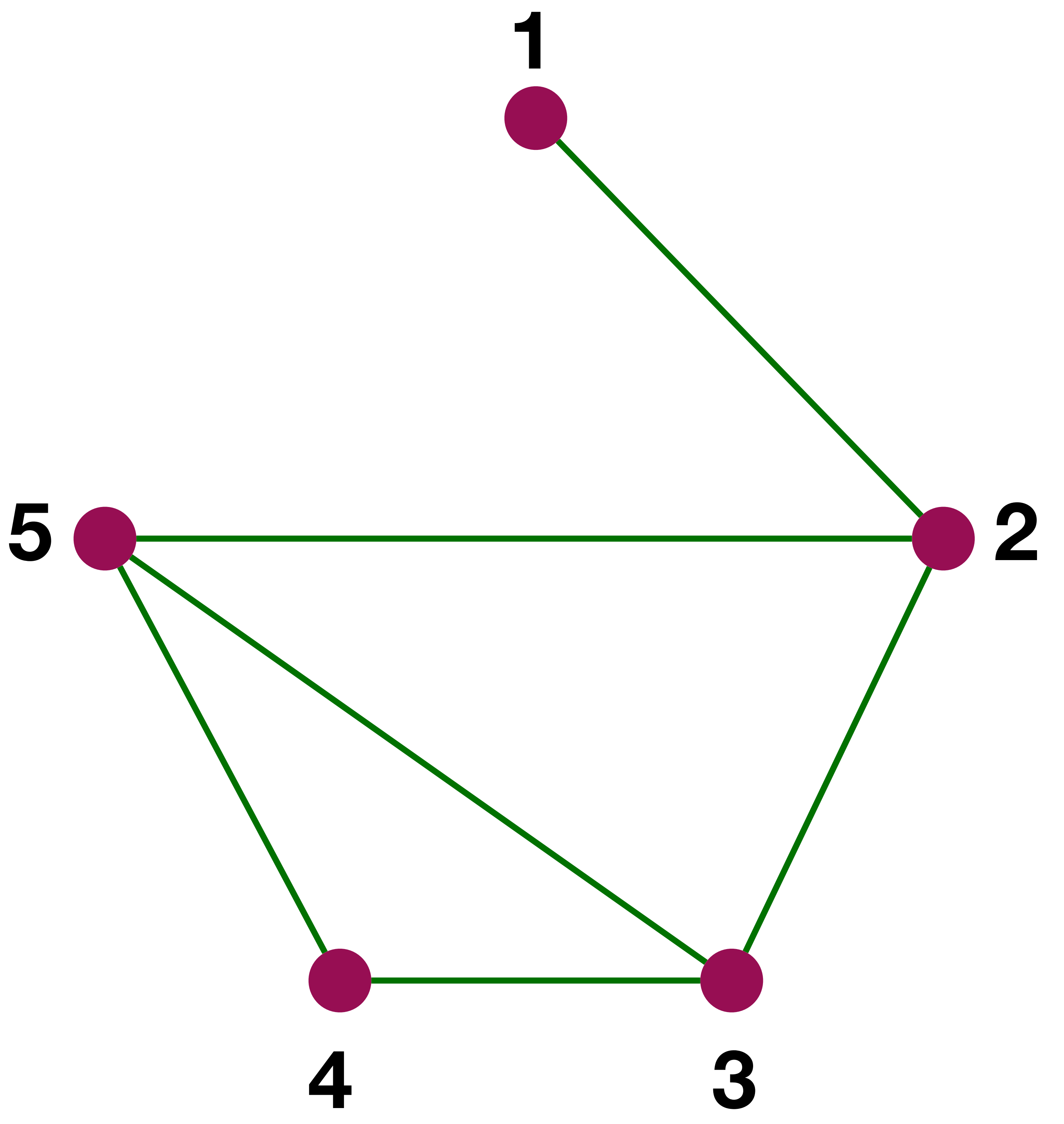
- Vertices connected by an edge are called *adjacent.*
- The *neighborhood* of a node v is the set of all vertices adjacent to v. It's denoted $N_G(v)$.
  - $N_G(2) = \{1,3,5\}$
- A vertex v is *incident* with an edge e when $v \in e$.



Basic notions

Degree

- Vertices connected by an edge are called *adjacent.*
- The *neighborhood* of a node v is the set of all vertices adjacent to v. It's denoted $N_G(v)$.
  - $N_G(2) = \{1,3,5\}$
- A vertex v is *incident* with an edge e when $v \in e$.
  - Vertex **2** is incident with edges $\{1,2\}$, $\{2,5\}$ and $\{2,3\}$



Basic notions

Degree

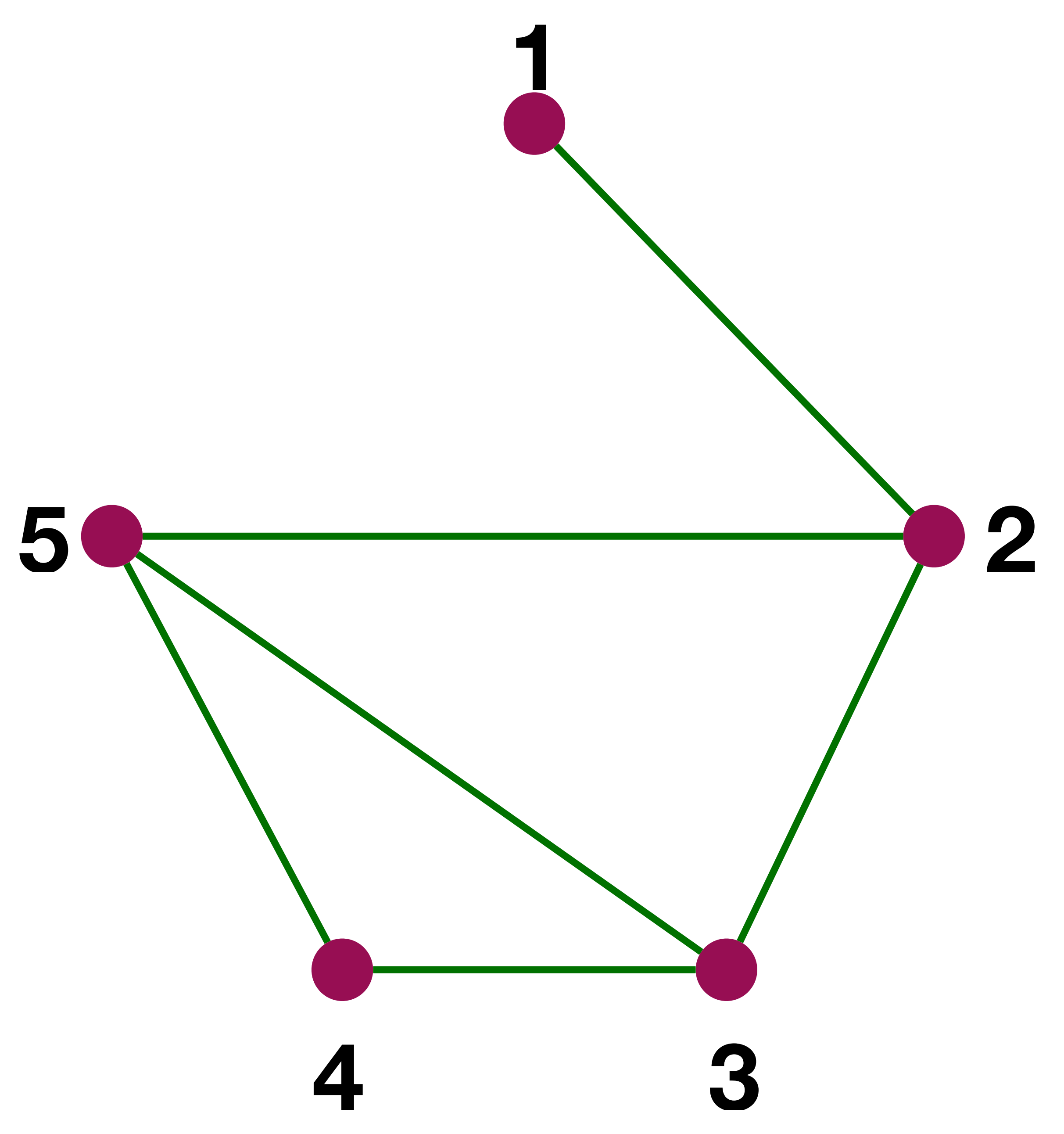
- The ***degree*** of a vertex is the number of edges incident to it:

Basic notions

Degree

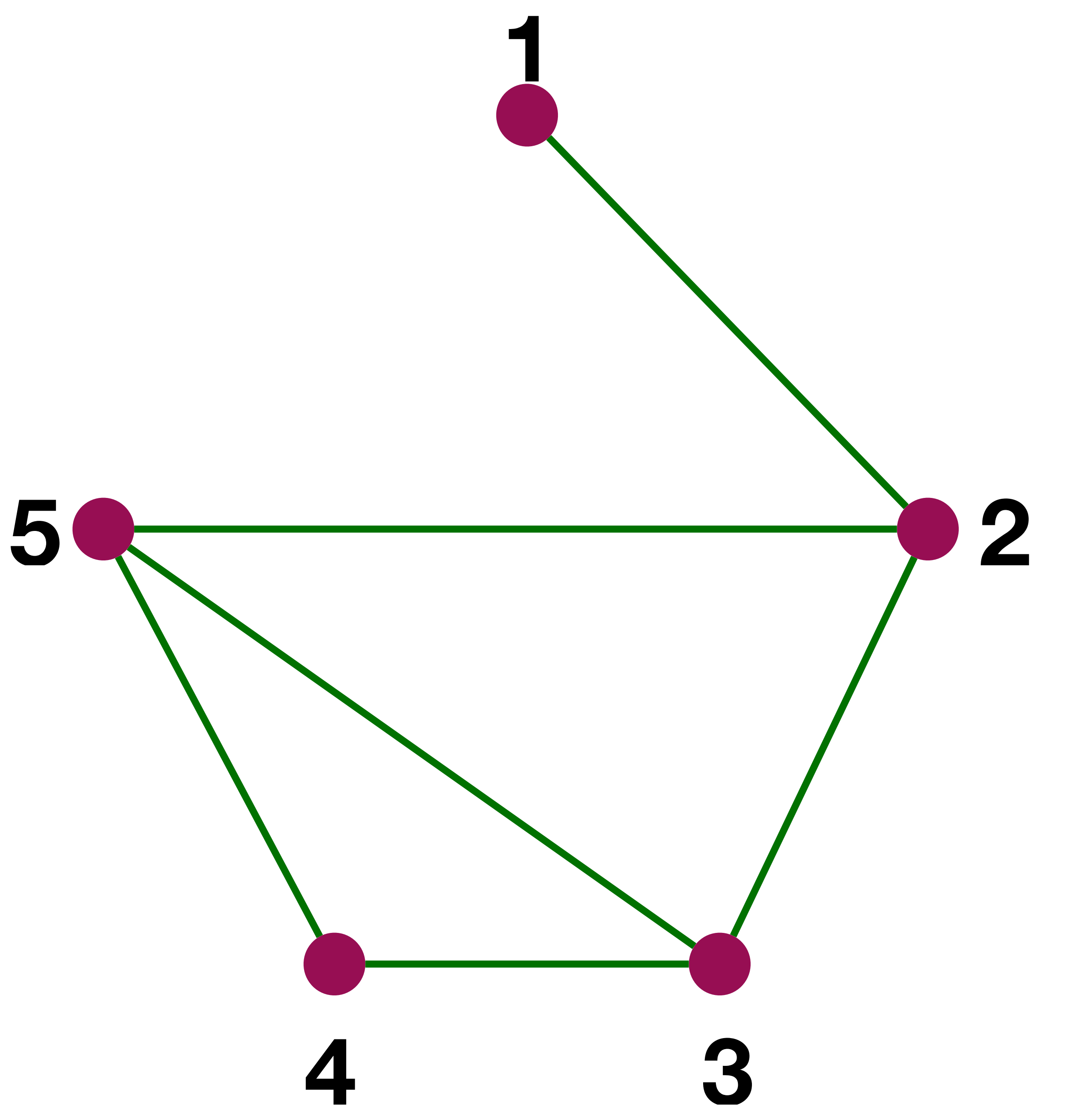
- The ***degree*** of a vertex is the number of edges incident to it:

$$d(1) = 1 \quad d(2) = 3 \quad d(3) = 3 \quad d(4) = 2 \quad d(5) = 3$$



Basic notions

Degree



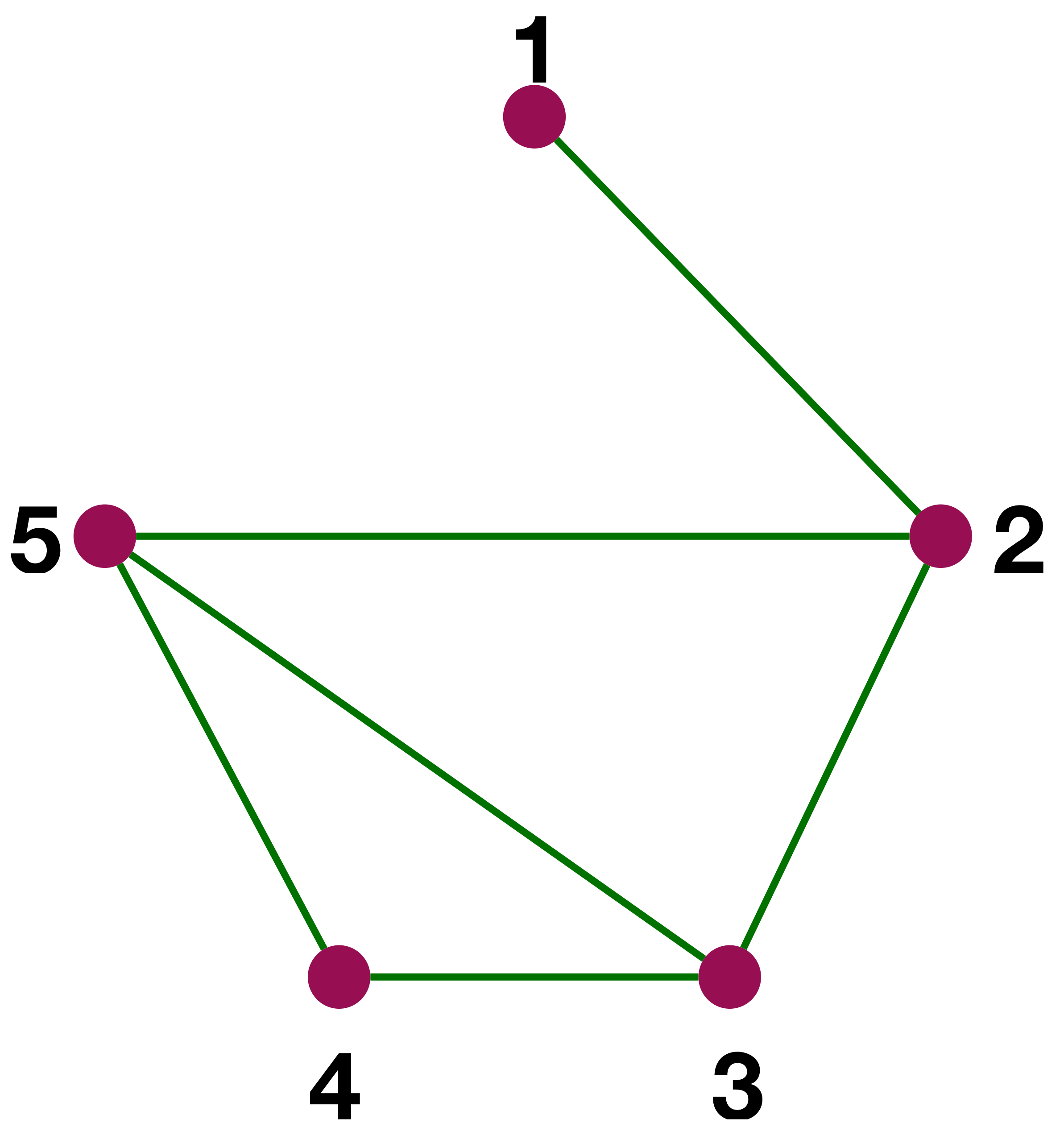
- The ***degree*** of a vertex is the number of edges incident to it:

$$d(1) = 1 \quad d(2) = 3 \quad d(3) = 3 \quad d(4) = 2 \quad d(5) = 3$$

- The ***degree sequence*** is to list the degrees listed in descending order:

Basic notions

Degree



- The ***degree*** of a vertex is the number of edges incident to it:

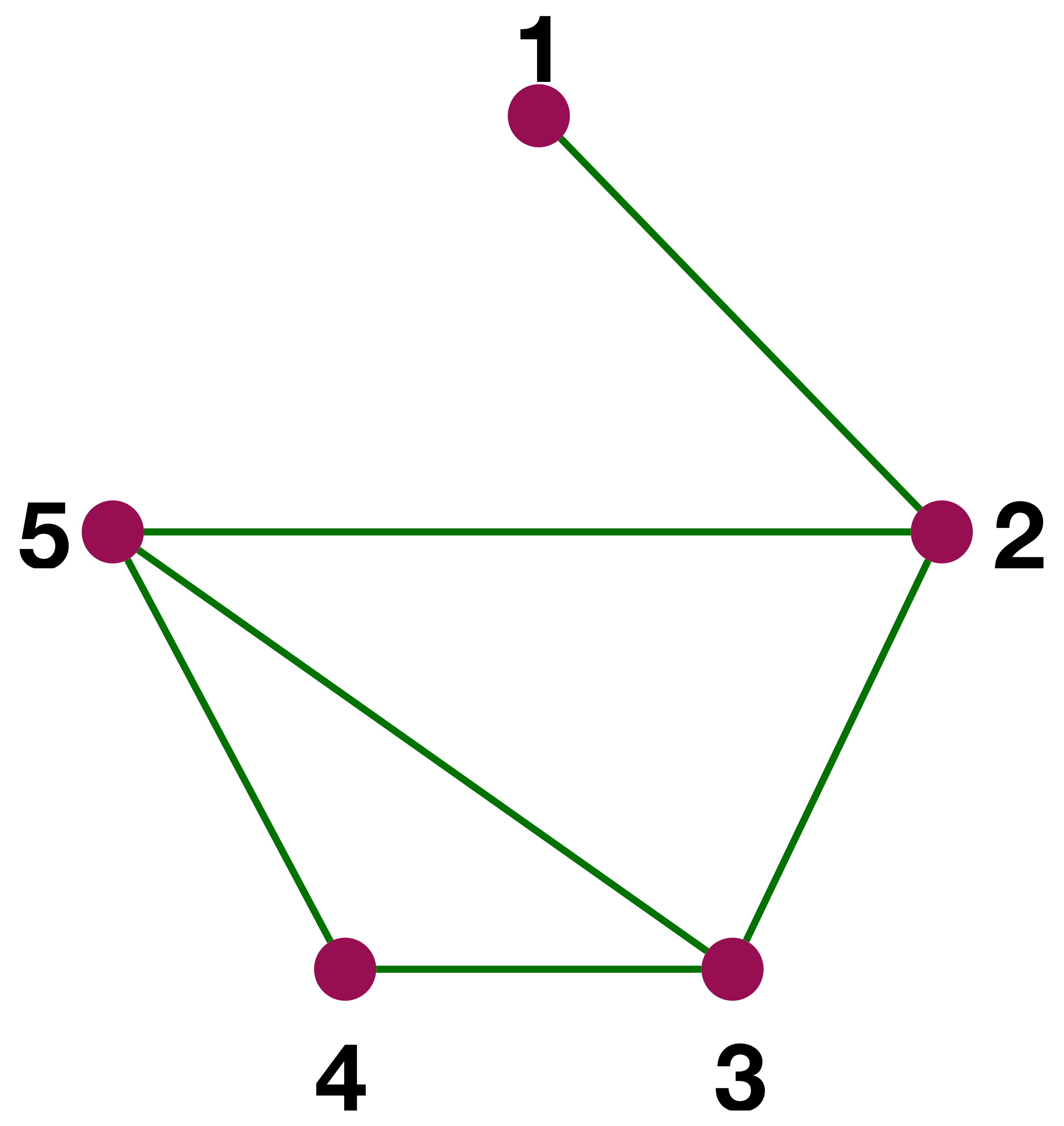
$$d(1) = 1 \quad d(2) = 3 \quad d(3) = 3 \quad d(4) = 2 \quad d(5) = 3$$

- The ***degree sequence*** is to list the degrees listed in descending order:

$$3,3,3,2,1$$

Basic notions

Degree



- The ***degree*** of a vertex is the number of edges incident to it:

$$d(1) = 1 \quad d(2) = 3 \quad d(3) = 3 \quad d(4) = 2 \quad d(5) = 3$$

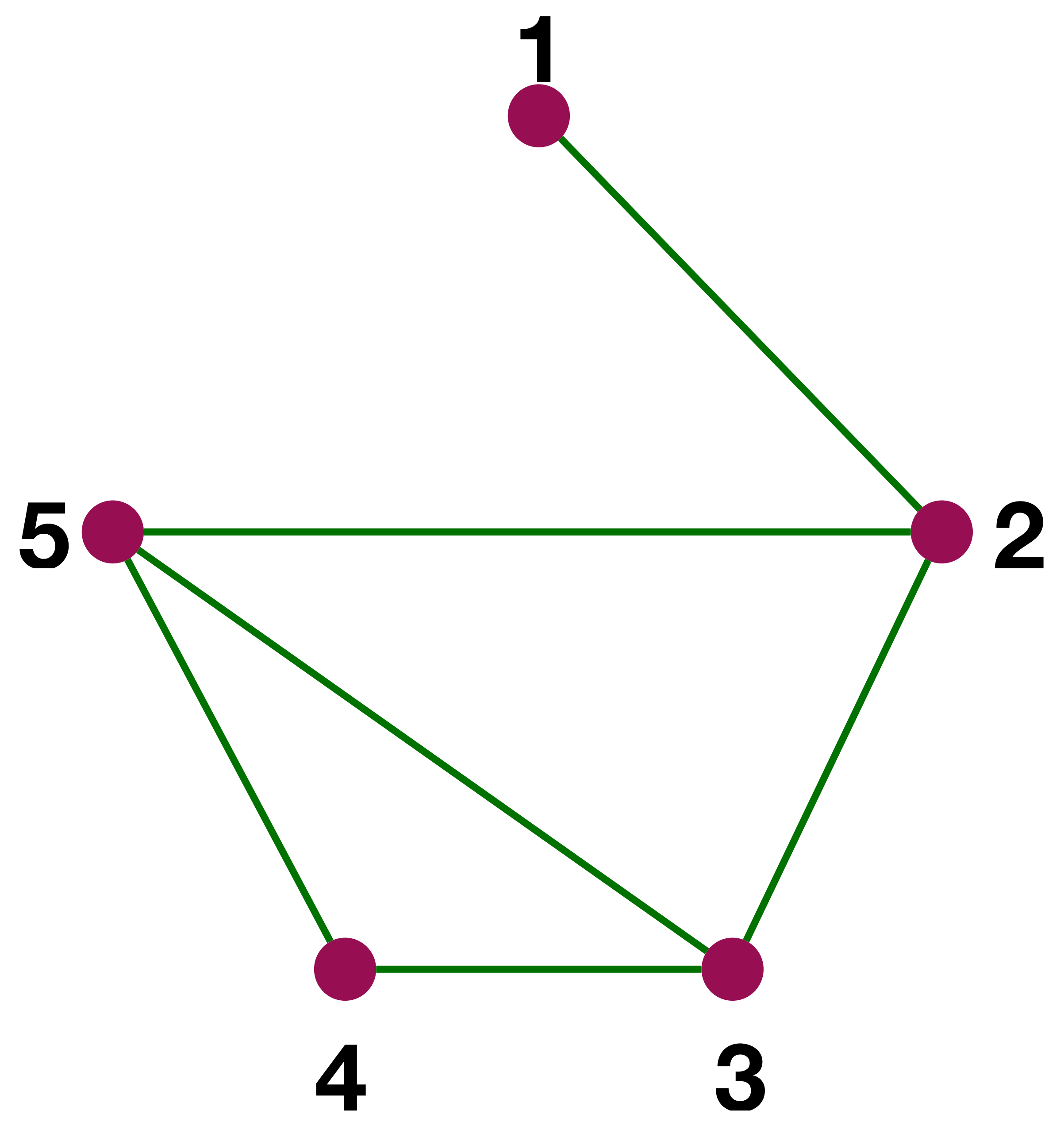
- The ***degree sequence*** is to list the degrees listed in descending order:

$$3,3,3,2,1$$

- The ***minimum degree*** is denoted $\delta(G)$. Here $\delta(G) = 1$

Basic notions

Degree



- The ***degree*** of a vertex is the number of edges incident to it:

$$d(1) = 1 \quad d(2) = 3 \quad d(3) = 3 \quad d(4) = 2 \quad d(5) = 3$$

- The ***degree sequence*** is to list the degrees listed in descending order:

$$3,3,3,2,1$$

- The ***minimum degree*** is denoted $\delta(G)$. Here $\delta(G) = 1$

- The ***maximum degree*** is denoted $\Delta(G)$. Here $\Delta(G) = 3$

Basic notions

Degree

- The ***degree*** of a vertex is the number of edges incident to it:

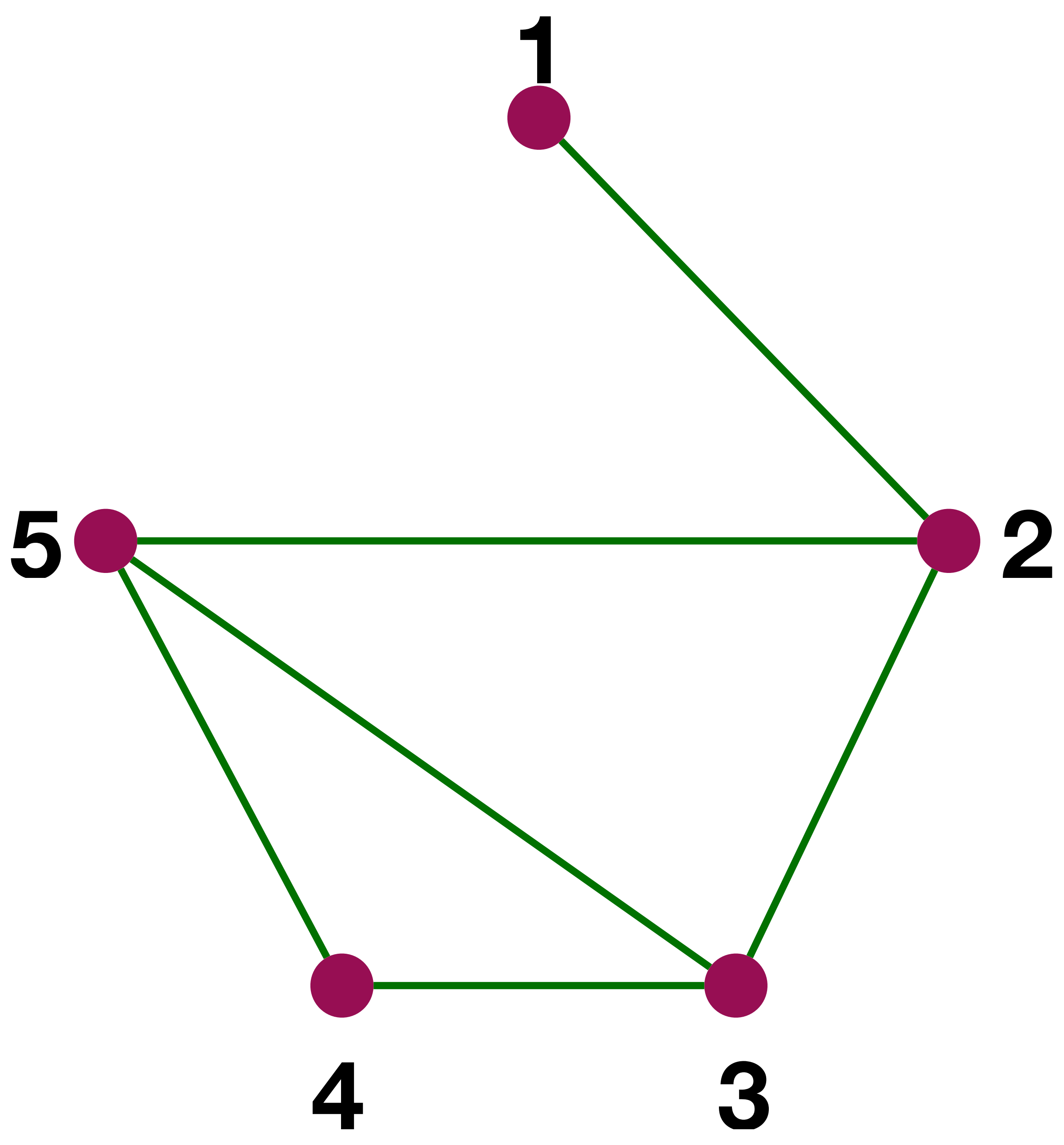
$$d(1) = 1 \quad d(2) = 3 \quad d(3) = 3 \quad d(4) = 2 \quad d(5) = 3$$

- The ***degree sequence*** is to list the degrees listed in descending order:

$$3,3,3,2,1$$

- The ***minimum degree*** is denoted $\delta(G)$. Here $\delta(G) = 1$

- The ***maximum degree*** is denoted $\Delta(G)$. Here $\Delta(G) = 3$



Handshaking lemma

$$\sum d(v) = 2|E|$$

Basic notions

Degree

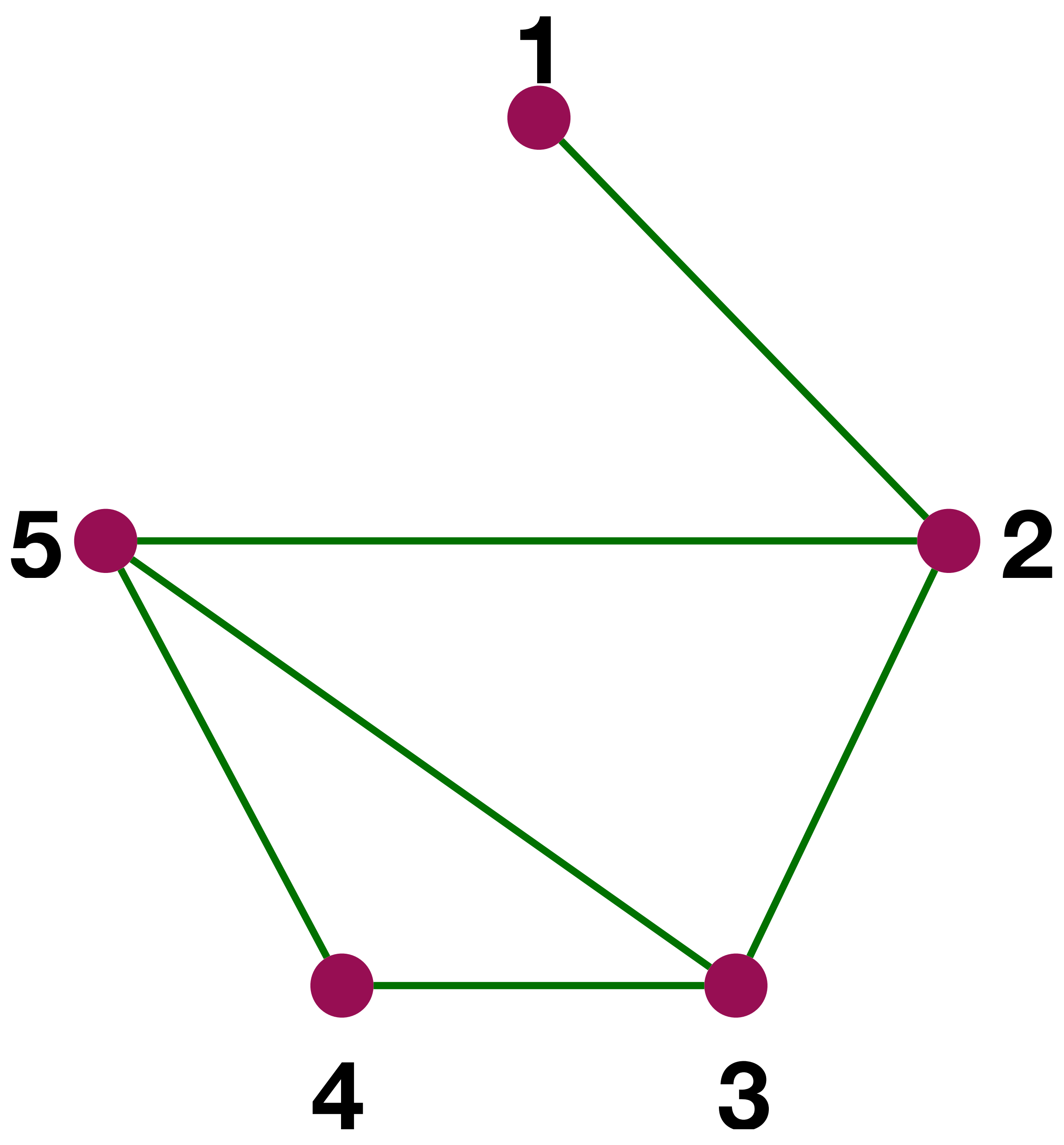
- The ***degree*** of a vertex is the number of edges incident to it:

$$d(1) = 1 \quad d(2) = 3 \quad d(3) = 3 \quad d(4) = 2 \quad d(5) = 3$$

- The ***degree sequence*** is to list the degrees listed in descending order:

$$3,3,3,2,1$$

- The ***minimum degree*** is denoted $\delta(G)$. Here $\delta(G) = 1$
- The ***maximum degree*** is denoted $\Delta(G)$. Here $\Delta(G) = 3$



Handshaking lemma

$$\sum d(v) = 2|E|$$

Sum of Degrees = 12
Number of Edges = 6

Graph representations

Adjacency matrix

Graph representation I

Represent $G = (V, E)$ with n vertices and m edges using a $n \times n$ ***adjacency matrix*** $A = (a_{ij})$ where

Adjacency matrix

Graph representation I

Represent $G = (V, E)$ with n vertices and m edges using a $n \times n$ ***adjacency matrix*** $A = (a_{ij})$ where

- $a_{ij} = a_{ji} = 1$ if $\{i, j\} \in E$ and $a_{ij} = a_{ji} = 0$ if $\{i, j\} \notin E$.

Adjacency matrix

Graph representation I

Represent $G = (V, E)$ with n vertices and m edges using a $n \times n$ ***adjacency matrix*** $A = (a_{ij})$ where

- $a_{ij} = a_{ji} = 1$ if $\{i, j\} \in E$ and $a_{ij} = a_{ji} = 0$ if $\{i, j\} \notin E$.

- Advantage: can check if $\{i, j\} \in E$ in $O(1)$ time

Adjacency matrix

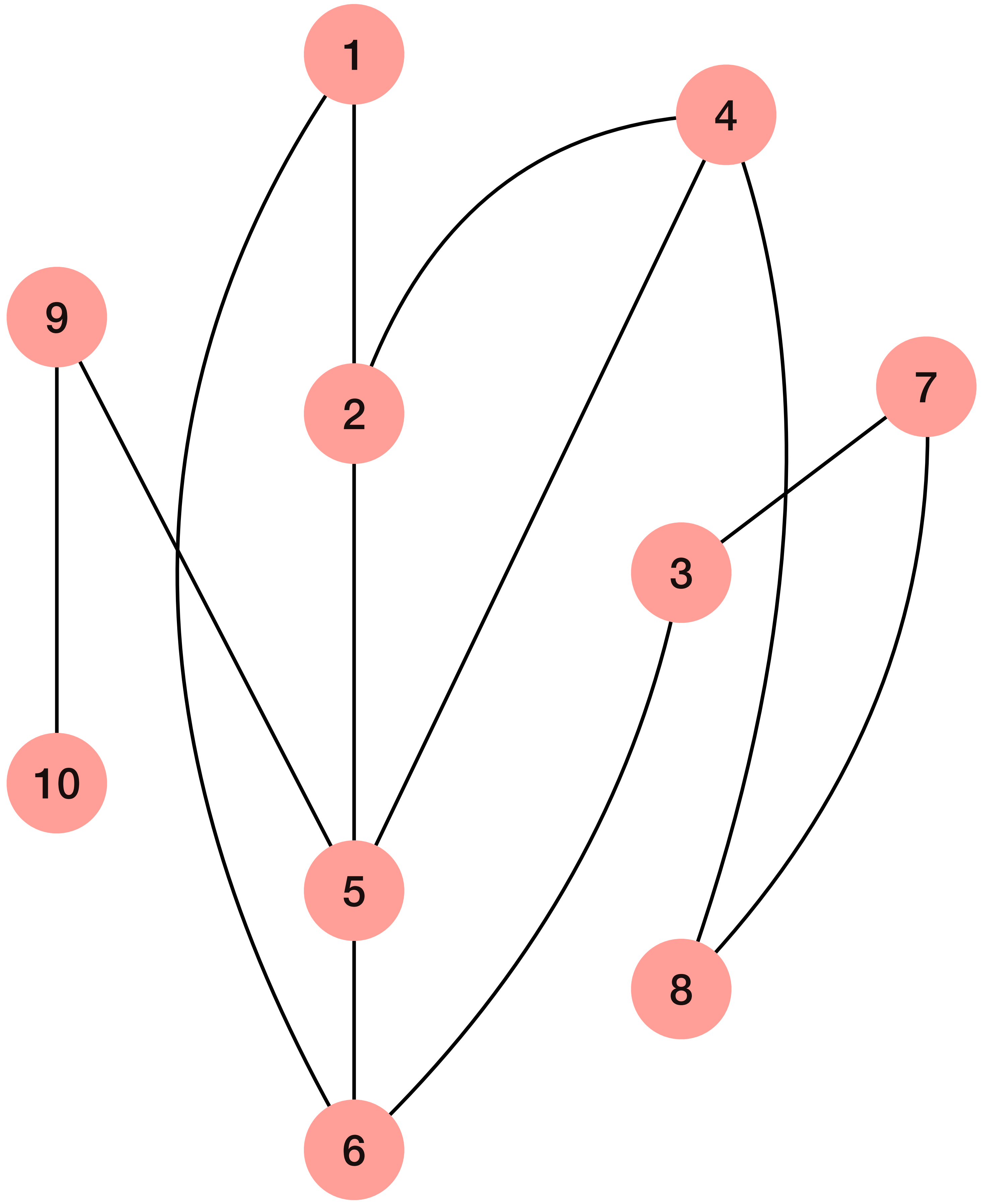
Graph representation I

Represent $G = (V, E)$ with n vertices and m edges using a $n \times n$ ***adjacency matrix*** $A = (a_{ij})$ where

- $a_{ij} = a_{ji} = 1$ if $\{i, j\} \in E$ and $a_{ij} = a_{ji} = 0$ if $\{i, j\} \notin E$.
- Advantage: can check if $\{i, j\} \in E$ in $O(1)$ time
- Disadvantage: needs $\Omega(n^2)$ space even when $m \ll n^2$

Graph adjacency matrix

Example



	1	2	3	4	5	6	7	8	9	10
1	0	1	0	0	0	1	0	0	0	0
2	1	0	0	1	1	0	0	0	0	0
3	0	0	0	0	0	1	1	0	0	0
4	0	1	0	0	1	0	0	1	0	0
5	0	1	0	1	0	1	0	0	1	0
6	1	0	1	0	1	0	0	0	0	0
7	0	0	1	0	0	0	0	1	0	0
8	0	0	0	1	0	0	1	0	0	0
9	0	0	0	0	1	0	0	0	0	1
10	0	0	0	0	0	0	0	0	1	0

Adjacency list

Graph representation II

Represent $G = (V, E)$ with n vertices and m edges using ***adjacency lists:***

Adjacency list

Graph representation II

Represent $G = (V, E)$ with n vertices and m edges using ***adjacency lists:***

- For each $u \in V$, $\mathrm{adj}(u) := N_G(u)$, that is neighbors of u.

Adjacency list

Graph representation II

Represent $G = (V, E)$ with n vertices and m edges using ***adjacency lists:***

- For each $u \in V$, $\text{adj}(u) := N_G(u)$, that is neighbors of u.
- Advantage: space is $O(m + n)$.

Adjacency list

Graph representation II

Represent $G = (V, E)$ with n vertices and m edges using ***adjacency lists:***

- For each $u \in V$, $\text{adj}(u) := N_G(u)$, that is neighbors of u.
- Advantage: space is $O(m + n)$.
- Disadvantage: cannot "easily" determine in $O(1)$ time whether $\{i, j\} \in E$

Adjacency list

Graph representation II

Represent $G = (V, E)$ with n vertices and m edges using ***adjacency lists:***

- For each $u \in V$, $\text{adj}(u) := N_G(u)$, that is neighbors of u.
- Advantage: space is $O(m + n)$.
- Disadvantage: cannot “easily” determine in $O(1)$ time whether $\{i, j\} \in E$

Note: In this class we will assume that by default, graphs are represented using plain vanilla (*unsorted*) adjacency lists.

Adjacency matrix vs. list

	1	2	3	4	5	6	7	8	9	10
1	0	1	0	0	0	1	0	0	0	0
2	1	0	0	1	1	0	0	0	0	0
3	0	0	0	0	0	1	1	0	0	0
4	0	1	0	0	1	0	0	1	0	0
5	0	1	0	1	0	1	0	0	1	0
6	1	0	1	0	1	0	0	0	0	0
7	0	0	1	0	0	0	0	1	0	0
8	0	0	0	1	0	0	1	0	0	0
9	0	0	0	0	1	0	0	0	0	1
10	0	0	0	0	0	0	0	0	1	0

Vertex	Adjacency List
1	2, 6
2	1, 4, 5
3	6, 7
4	2, 5, 8
5	2, 4, 6, 9
6	1, 3, 5
7	3, 8
8	4, 7
9	5, 10
10	9

Concrete representations

How might we represent this in a language?

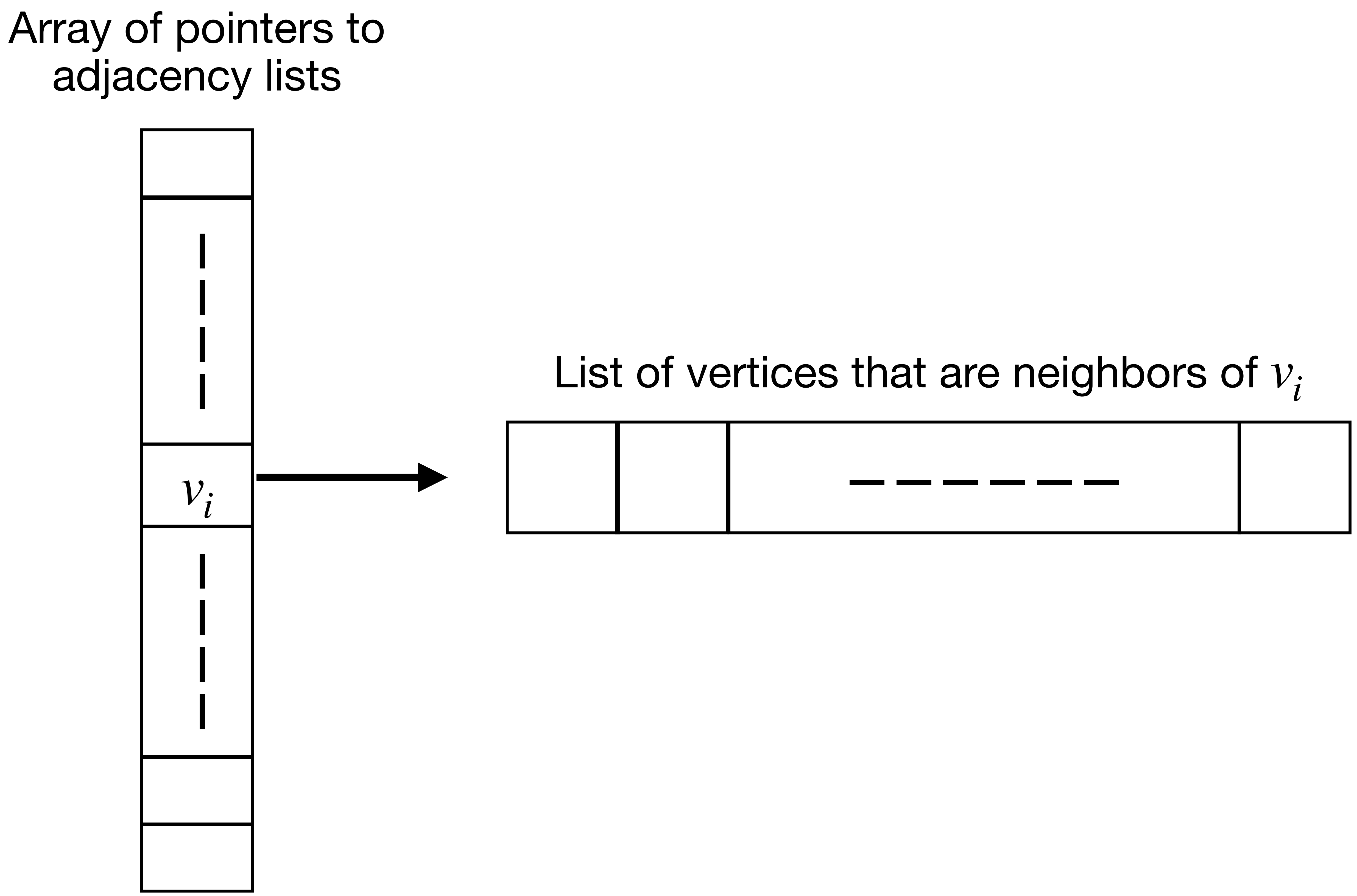
- Python-like (nested lists can be of different sizes)

```
alist = [[2,6],
         [1,4,5],
         [6,7],
         [2,5,8],
         [2,4,5,9],
         [1,3,5],
         [3,8],
         [4,7],
         [5,10],
         [9]]
```

Vertex	Adjacency List
1	2, 6
2	1, 4, 5
3	6, 7
4	2, 5, 8
5	2, 4, 6, 9
6	1, 3, 5
7	3, 8
8	4, 7
9	5, 10
10	9

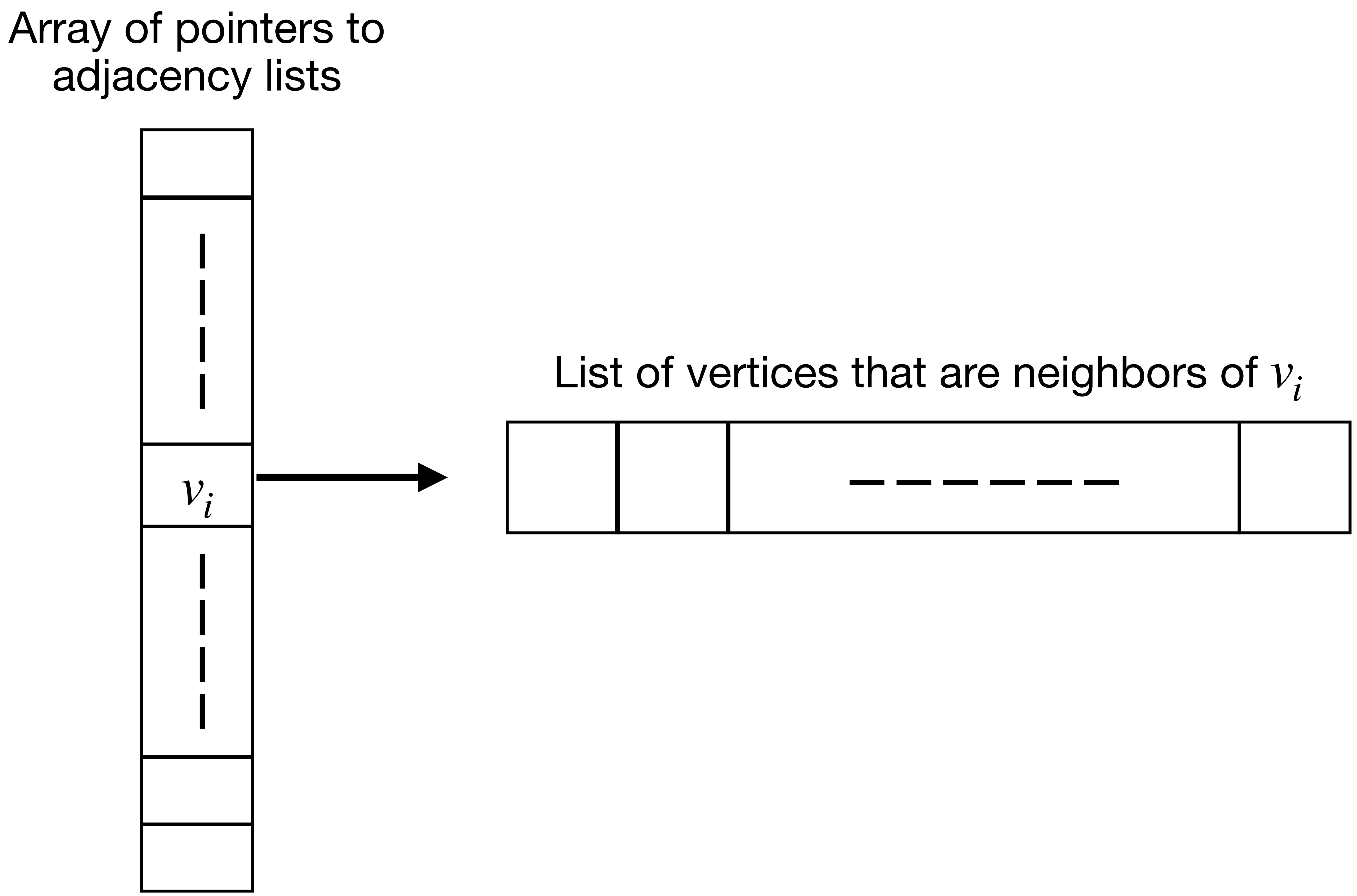
Concrete representations

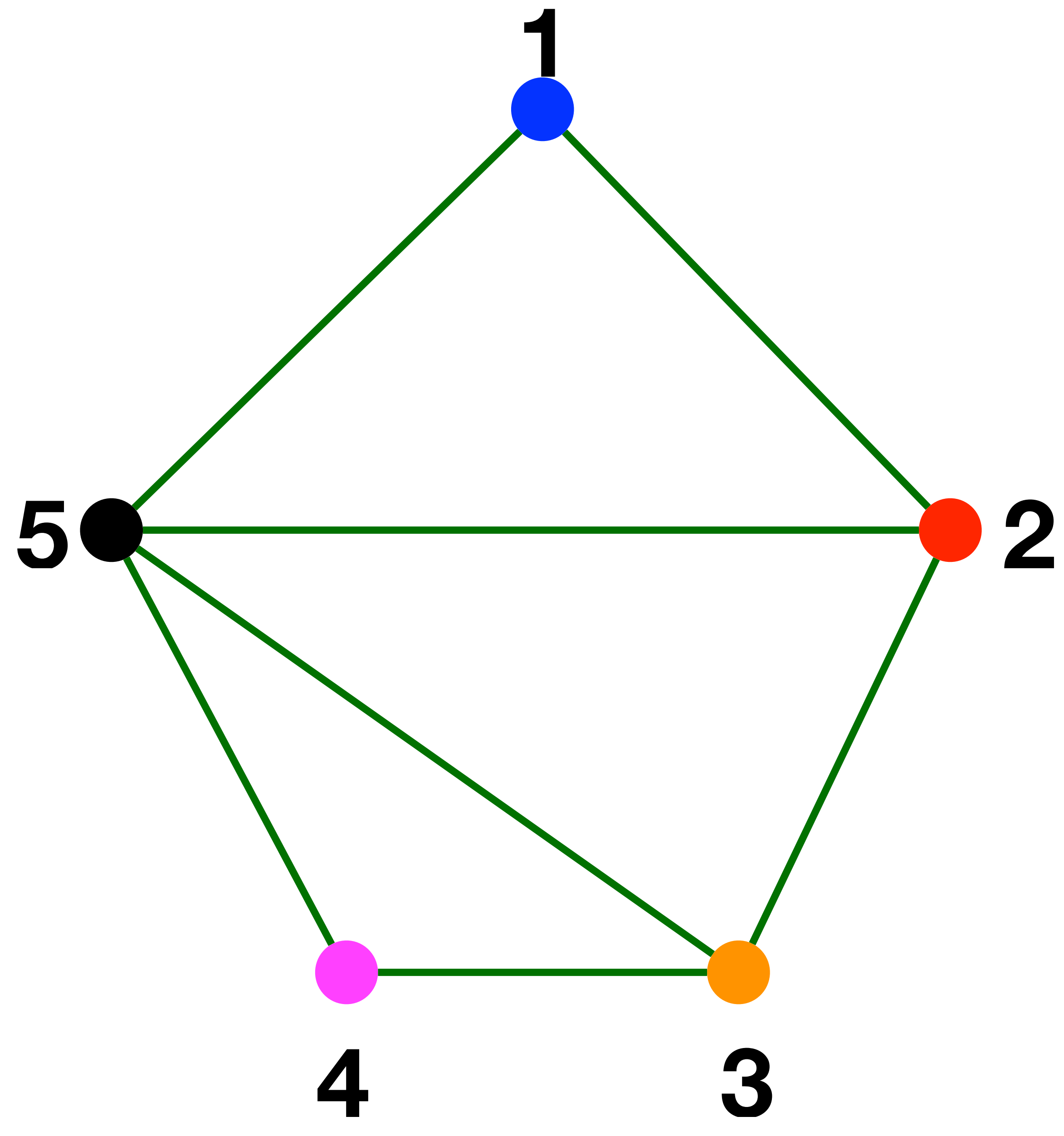
C-like: Can use pointers



Concrete representations

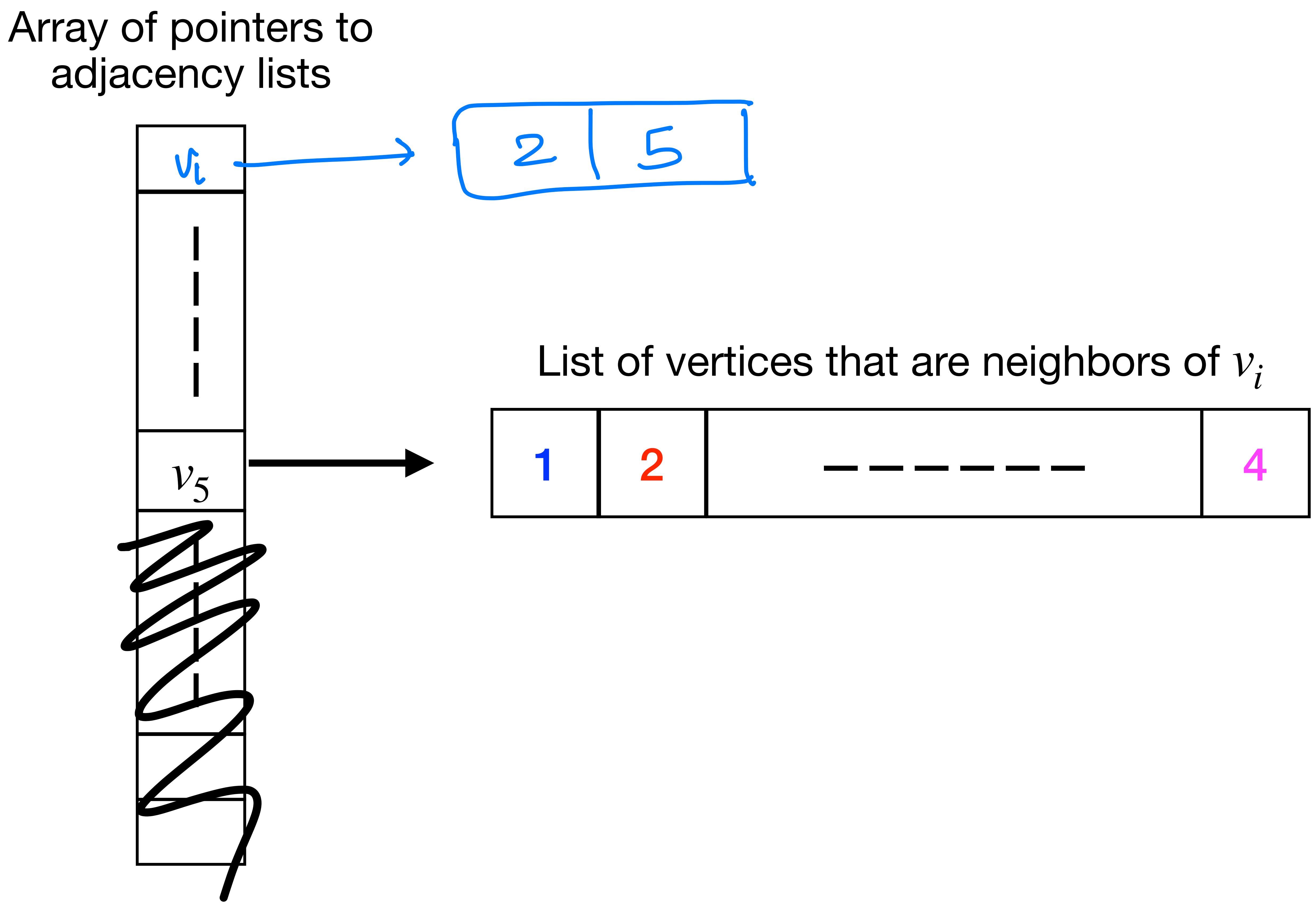
C-like: Can use pointers

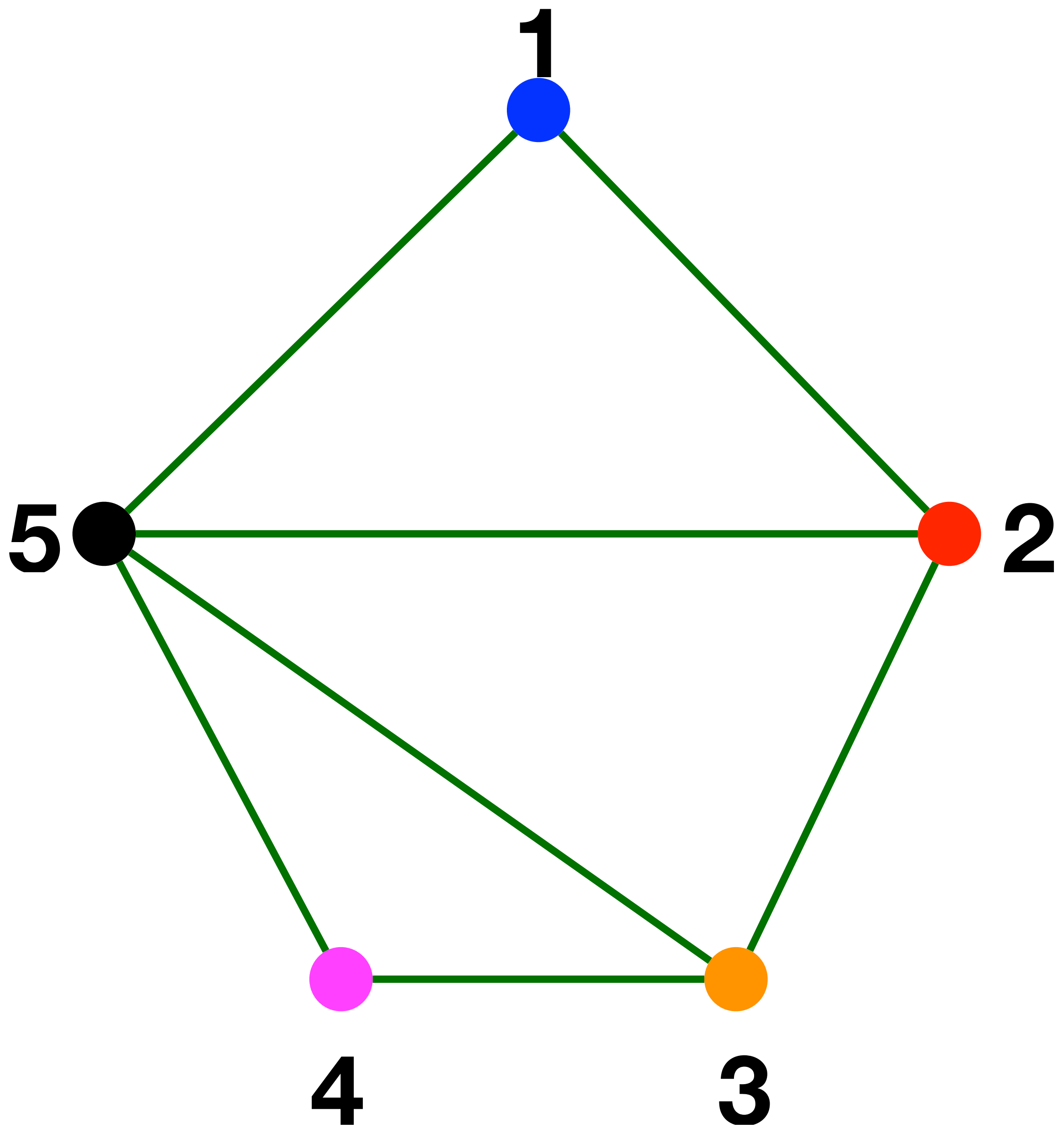




Concrete representations

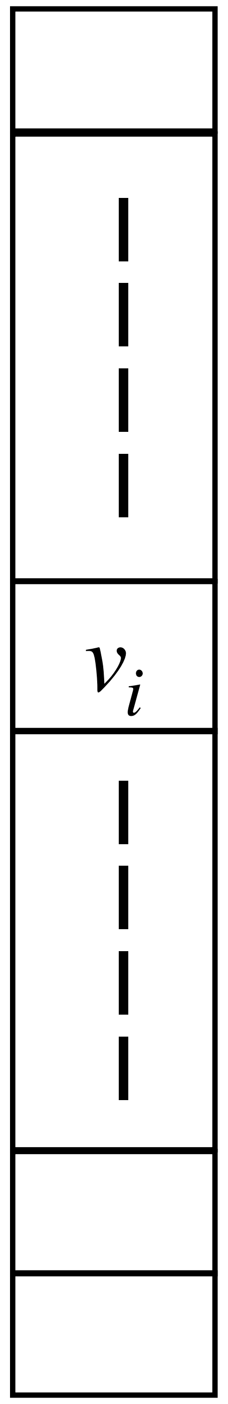
C-like: Can use pointers

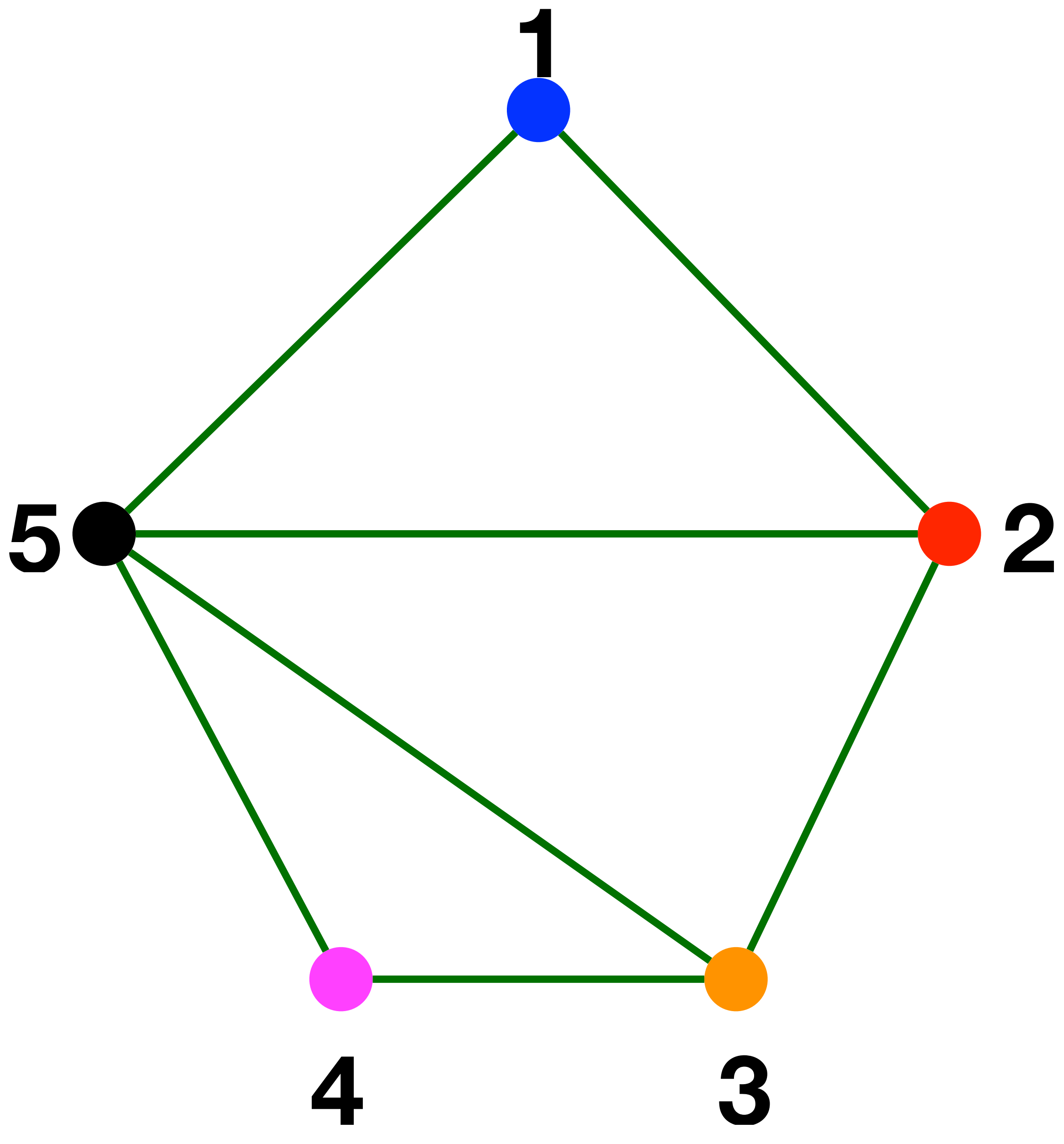




Concrete representations

How about using plain arrays?

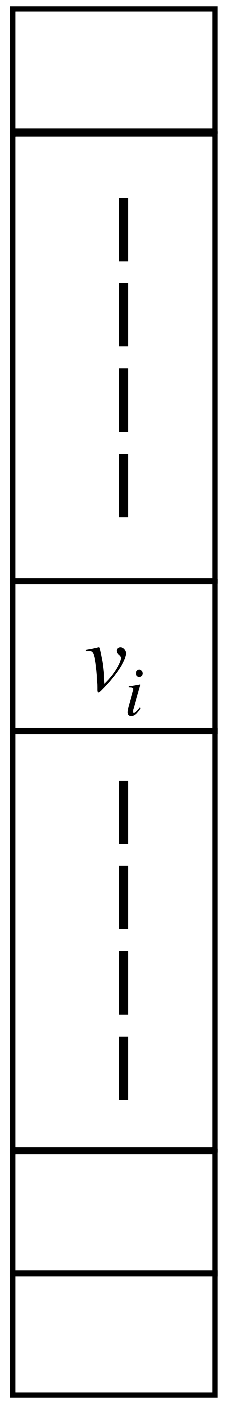


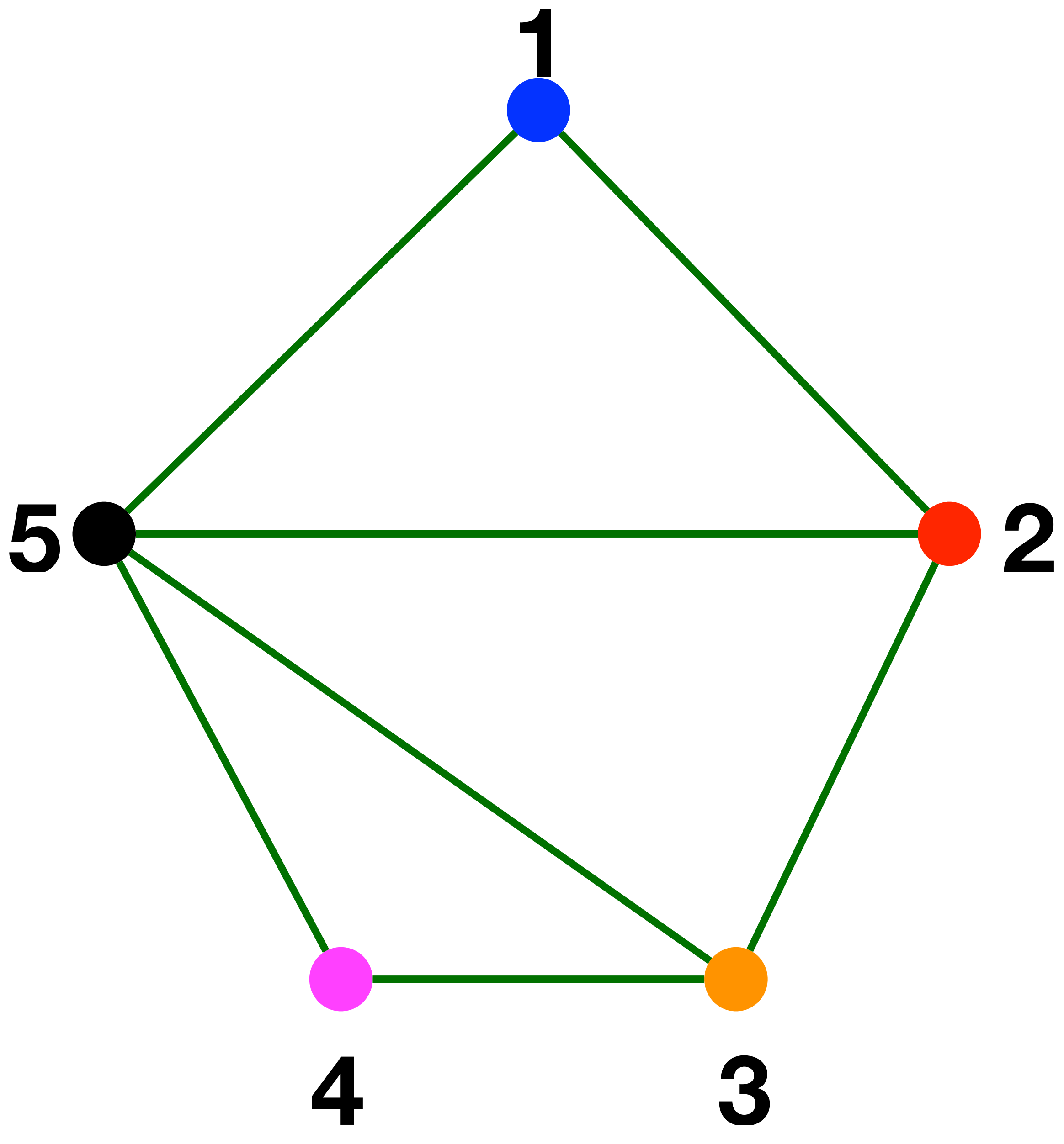


Concrete representations

How about using plain arrays?

Array of vertices, $\mathscr{V}$

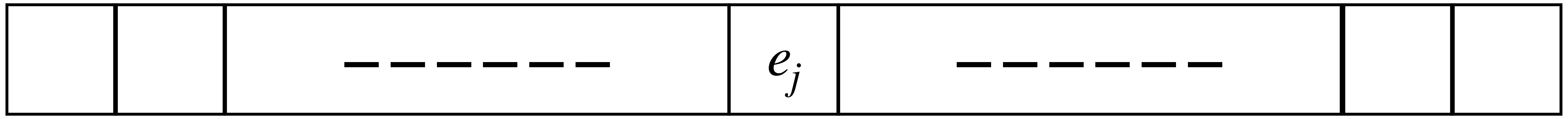




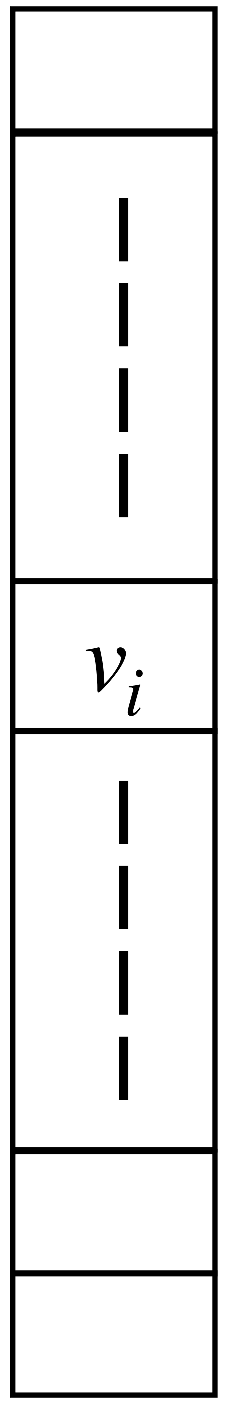
Concrete representations

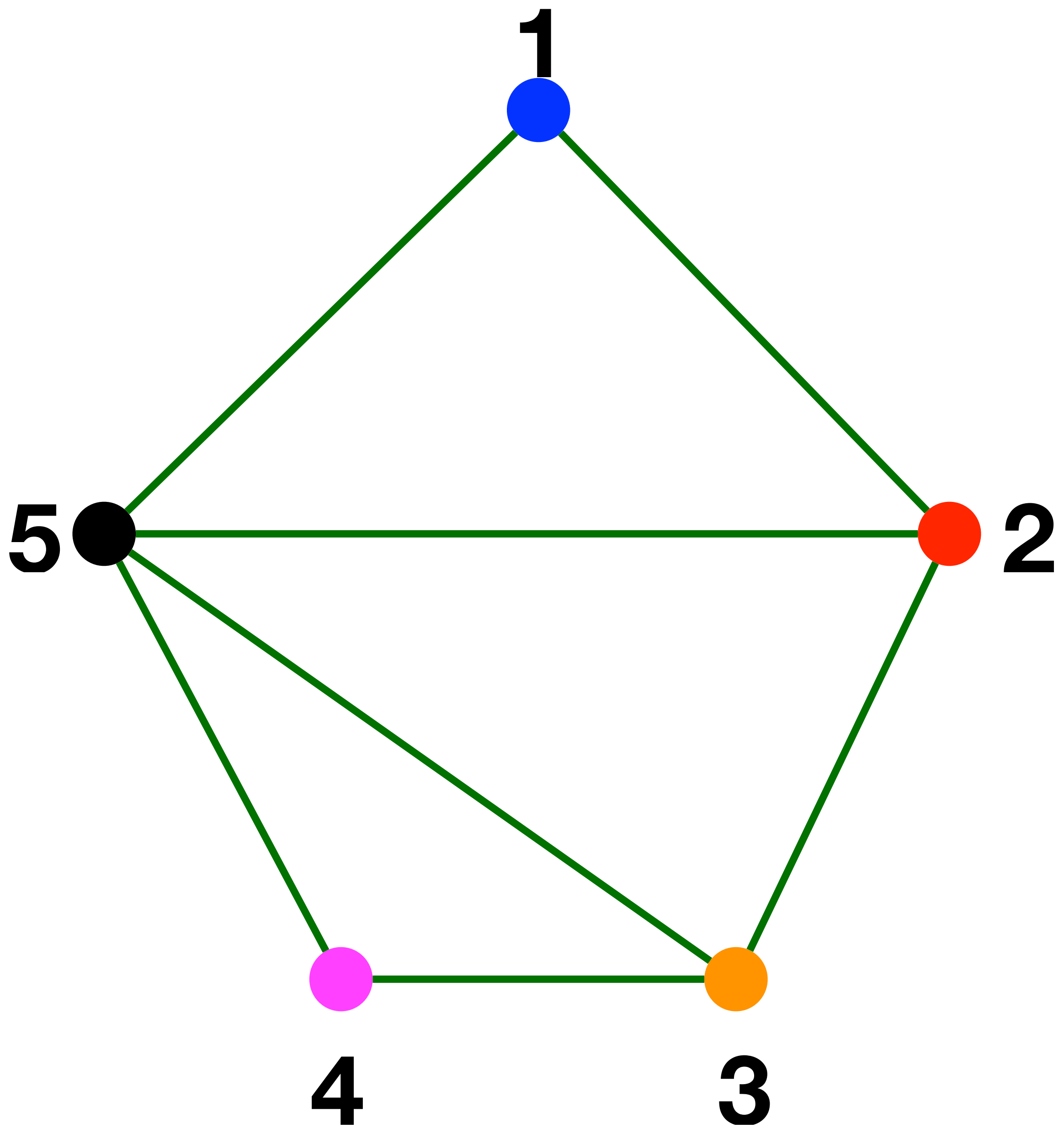
How about using plain arrays?

An edge array, $\mathscr{E}$ ← script to differentiate from E



Array of vertices, $\mathscr{V}$ ← script to differentiate from V

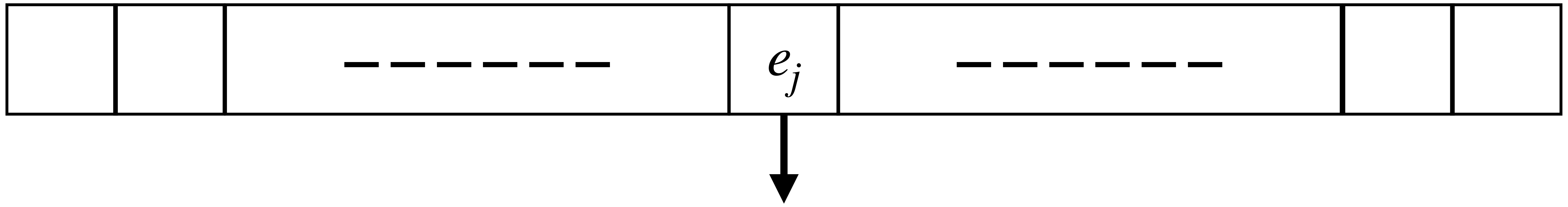




Concrete representations

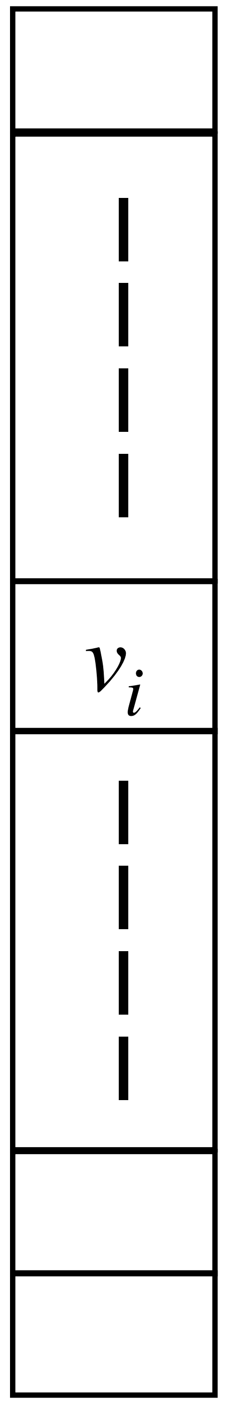
How about using plain arrays?

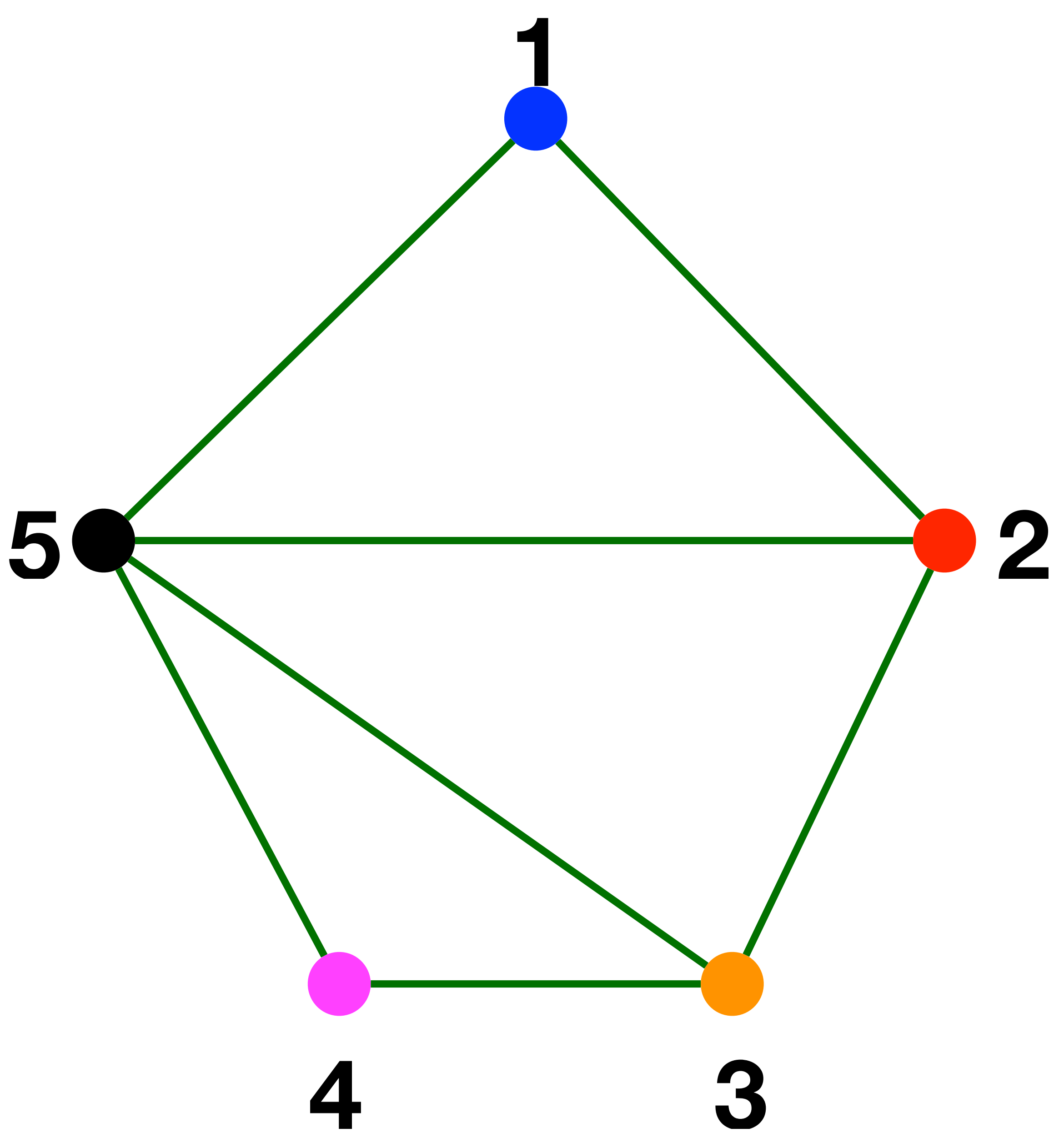
An edge array, $\mathscr{E}$



e_j is the *destination* vertex of the j-th edge

Array of vertices, $\mathscr{V}$

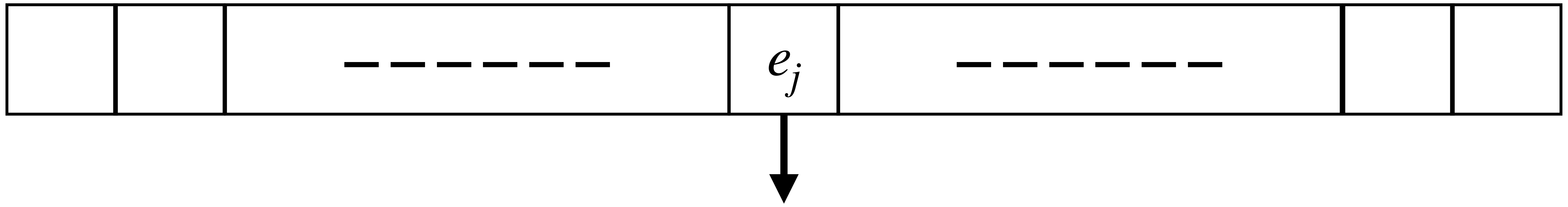




Concrete representations

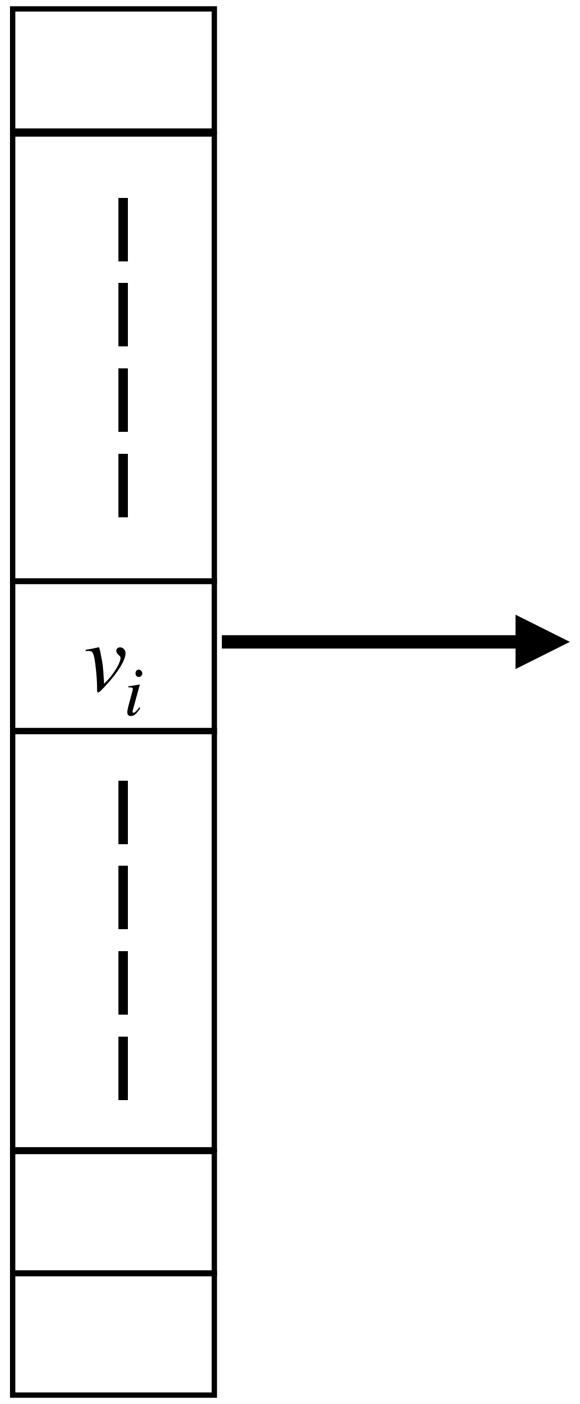
How about using plain arrays?

An edge array, $\mathscr{E}$

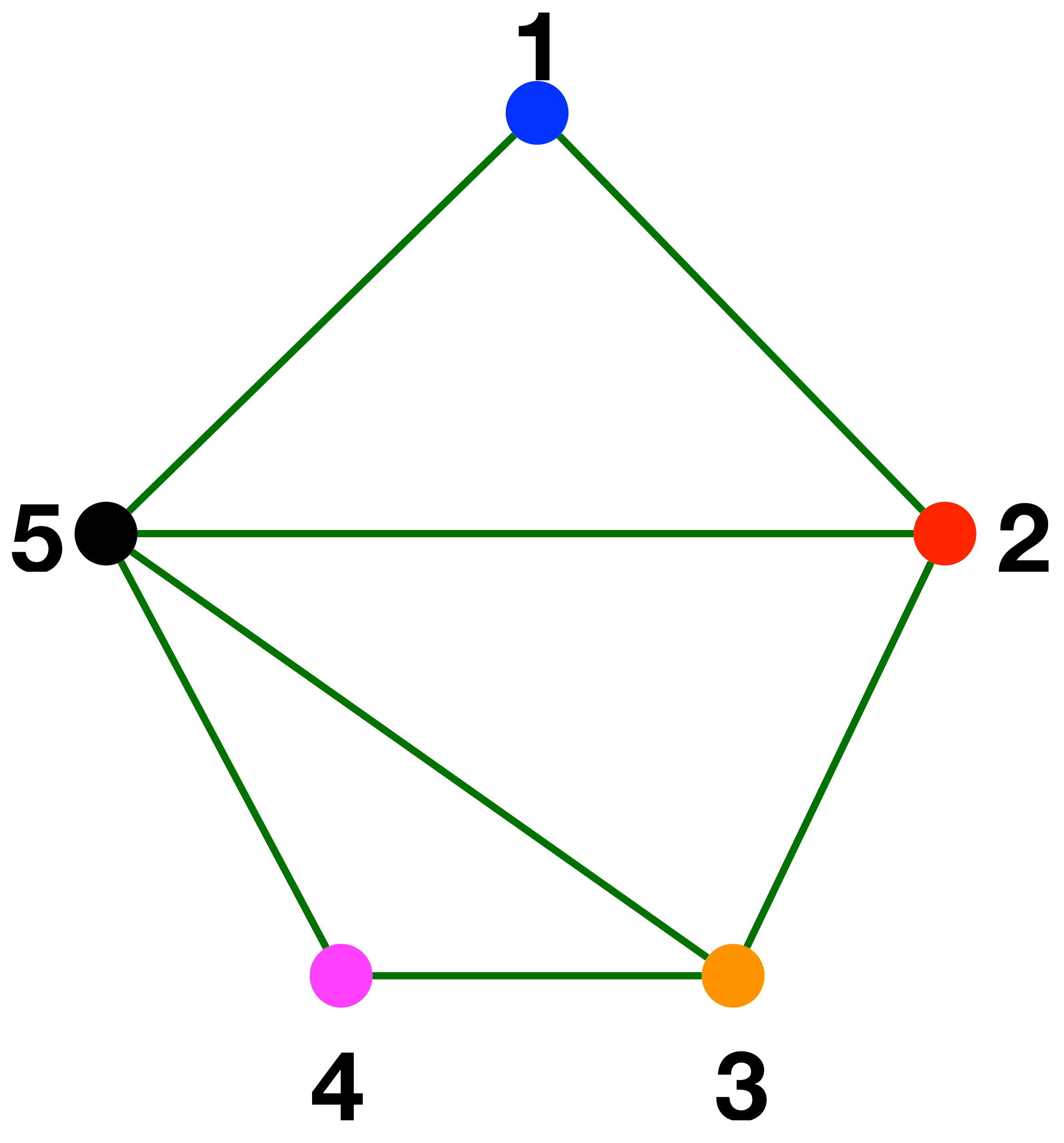


e_j is the *destination* vertex of the j-th edge

Array of vertices, $\mathscr{V}$



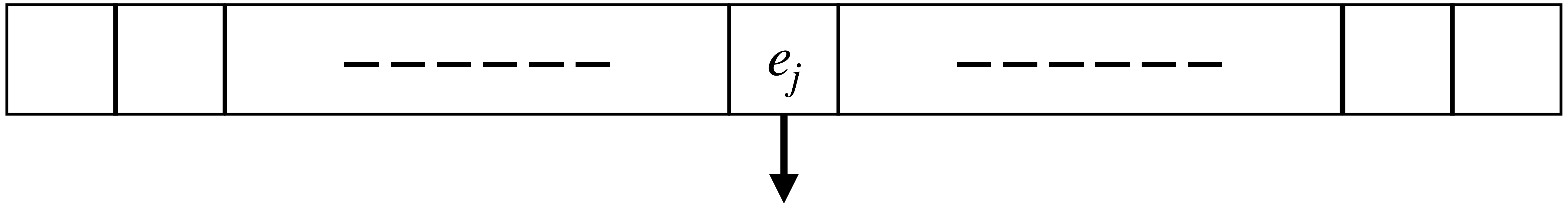
v_i is starting index (in $\mathscr{E}$) of vertices adjacent to v_i



Concrete representations

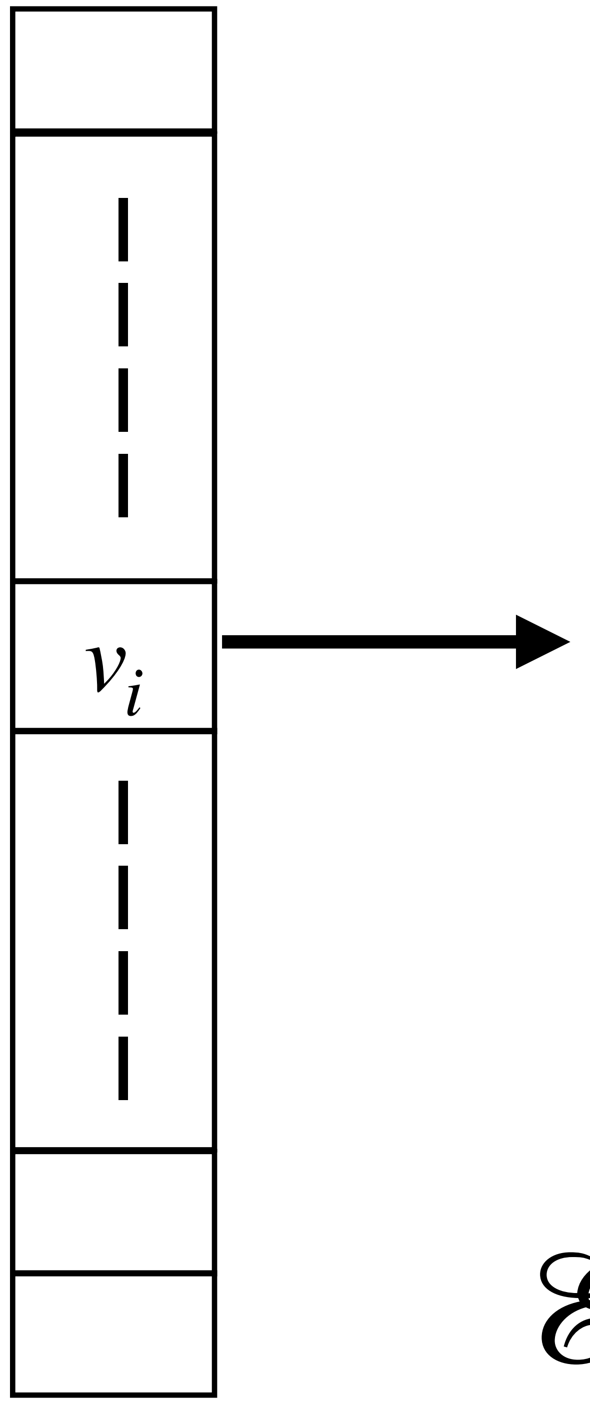
How about using plain arrays?

An edge array, $\mathscr{E}$



e_j is the *destination* vertex of the j-th edge

Array of vertices, $\mathscr{V}$



v_i is starting index (in $\mathscr{E}$) of vertices adjacent to v_i

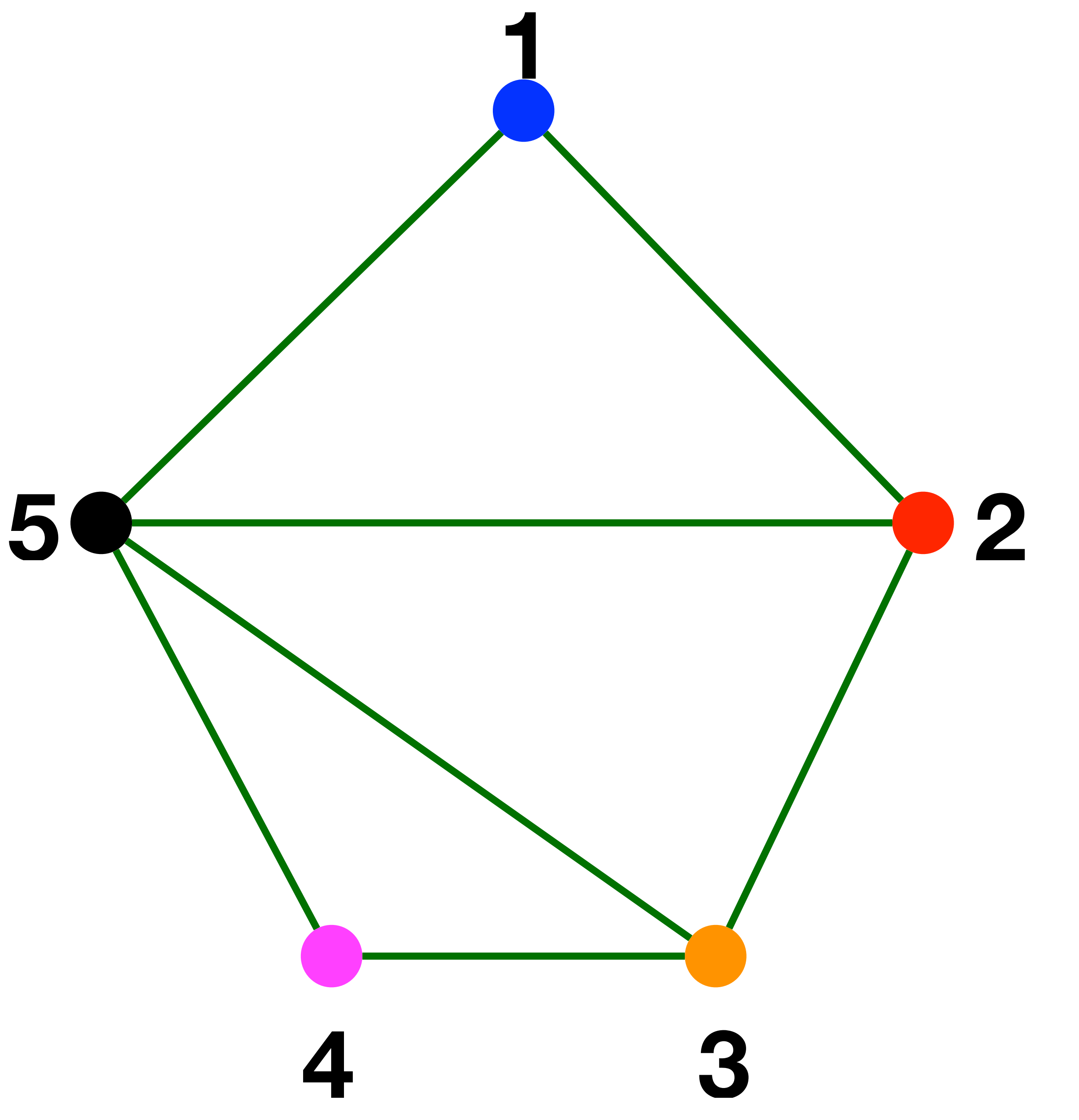
Assuming zero based indexing

v_1 v_2 v_3 v_4 v_5

$$\mathscr{V} = [0, 2, 5, 8, 10]$$

$$\mathscr{E} = [2,5, 1,3,5, 2,4,5, 3,5, 1,2,3,4]$$

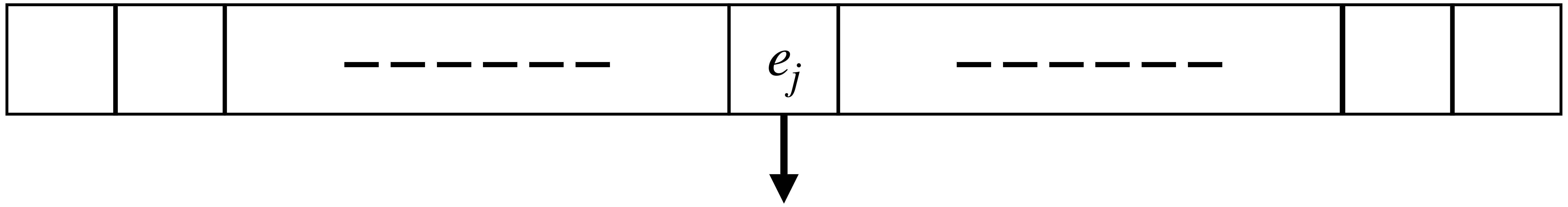
0 1 | 2 3 4 | 5 6 7 | 8 9 10 | 11 12 13



Concrete representations

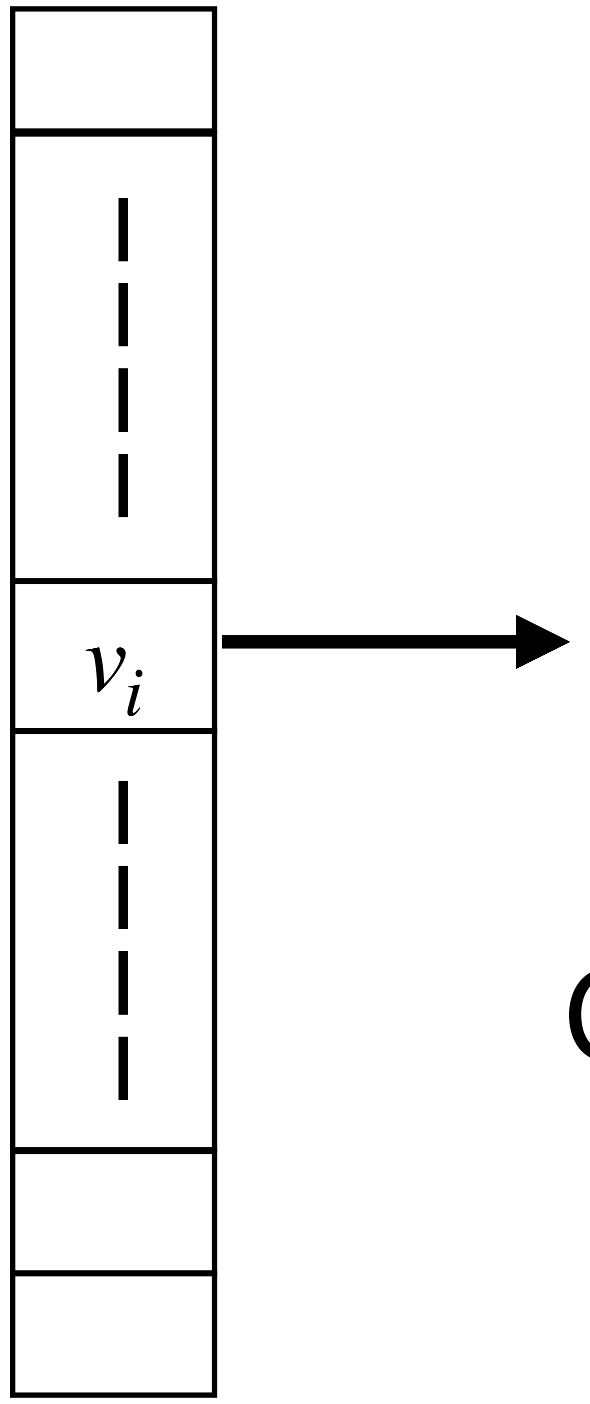
How about using plain arrays?

An edge array, $\mathscr{E}$



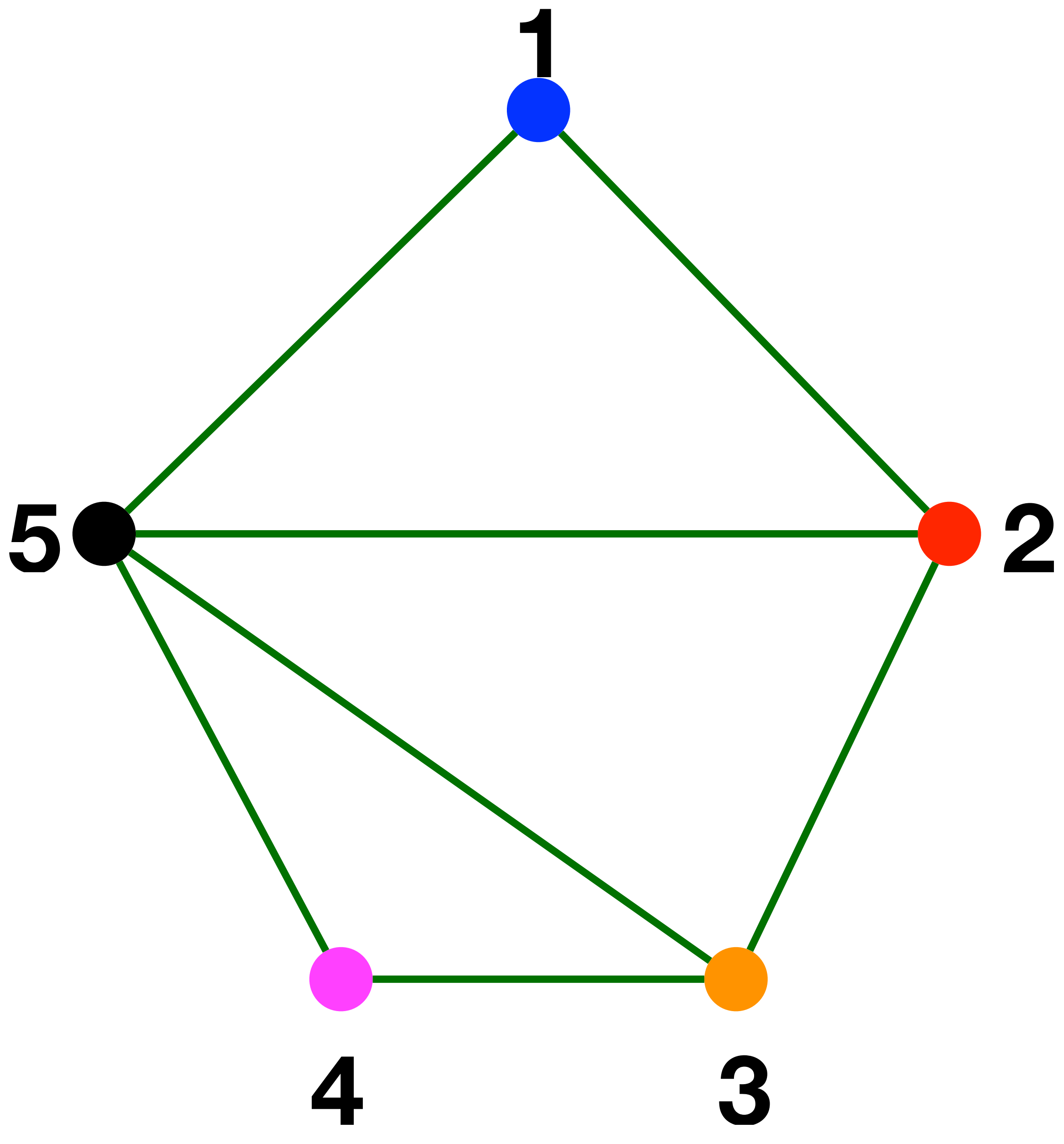
e_j is the *destination* vertex of the j-th edge

Array of vertices, $\mathscr{V}$



v_i is starting index (in $\mathscr{E}$) of vertices adjacent to v_i

Can get neighbors of v_i by examining
$\mathscr{E}\left[\mathscr{V}[i]\right]$ to $\mathscr{E}\left[\mathscr{V}[i+1]\right]$



Concrete representations
Advantages

- Edges are explicitly represented/numbered. Scanning/processing all edges easy to do.
- Representation easily supports multi-graphs including self-loops.
- Explicit numbering of vertices and edges allows use of arrays.
- Can also implement via pointer based lists for certain dynamic graph settings

Connectivity

Paths on a graph

Given a graph $G = (V, E)$:

Connectivity

Paths on a graph

Given a graph $G = (V, E)$:

- A *path* from v_1 to v_k is a sequence of distinct vertices $v_1, v_2, \ldots, v_k$ such that $\{v_i, v_{i+1}\} \in E$ for $1 \leq i \leq k-1$. The length of the path is $k - 1$.

Connectivity

Paths on a graph

Given a graph $G = (V, E)$:

- A *path* from v_1 to v_k is a sequence of distinct vertices $v_1, v_2, \ldots, v_k$ such that $\{v_i, v_{i+1}\} \in E$ for $1 \leq i \leq k - 1$. The length of the path is $k - 1$.
  - Note: A single vertex u is a path of length 0.

Connectivity

Paths on a graph

Given a graph $G = (V, E)$:

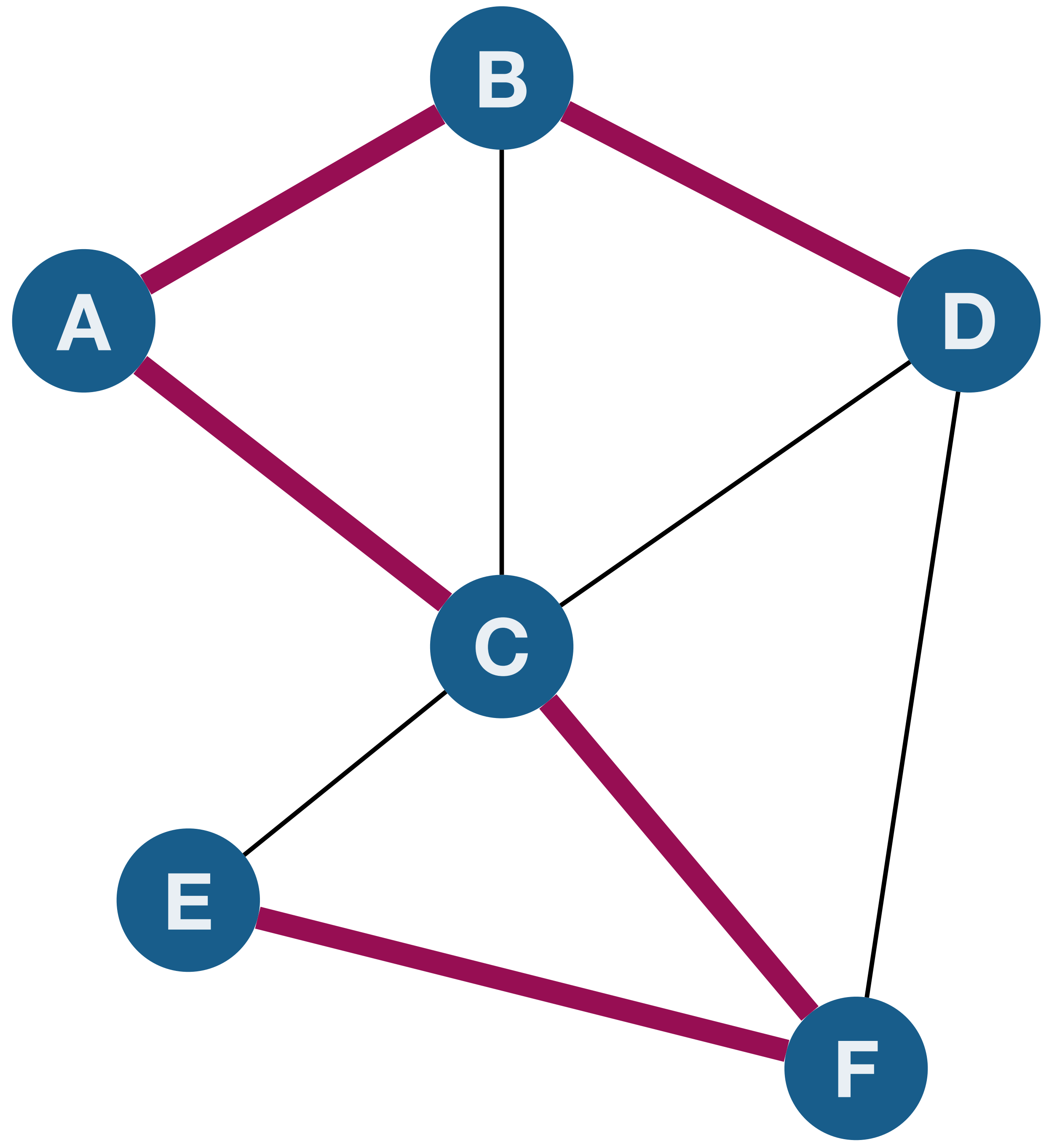
- A *path* from v_1 to v_k is a sequence of distinct vertices $v_1, v_2, \ldots, v_k$ such that $\{v_i, v_{i+1}\} \in E$ for $1 \leq i \leq k-1$. The length of the path is $k-1$.
  - Note: A single vertex u is a path of length 0.
- We say a vertex u is connected to a vertex v if there is a path from u to v.

Connectivity

Paths on a graph

Given a graph $G = (V, E)$:

- A *path* from v_1 to v_k is a sequence of distinct vertices $v_1, v_2, \ldots, v_k$ such that $\{v_i, v_{i+1}\} \in E$ for $1 \le i \le k-1$. The length of the path is $k - 1$.
  - Note: A single vertex u is a path of length 0.
- We say a vertex u is connected to a vertex v if there is a path from u to v.

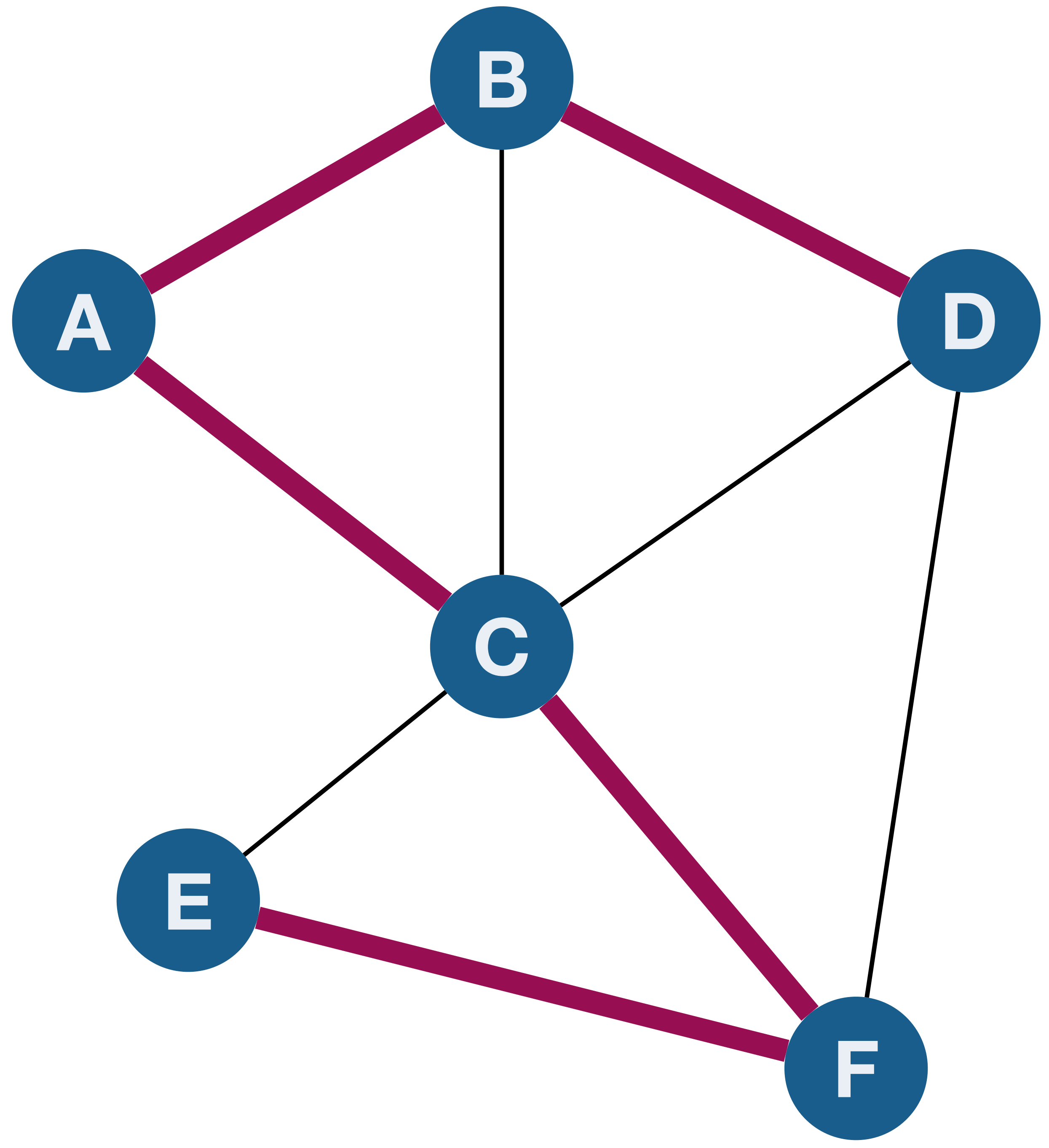


Connectivity

Paths on a graph

Given a graph $G = (V, E)$:

- A *path* from v_1 to v_k is a sequence of distinct vertices $v_1, v_2, \ldots, v_k$ such that $\{v_i, v_{i+1}\} \in E$ for $1 \leq i \leq k-1$. The length of the path is $k - 1$.
  - Note: A single vertex u is a path of length 0.
- We say a vertex u is connected to a vertex v if there is a path from u to v.
- Example: ***D, B, A, C, F, E***



Connectivity

Cycle

Given a graph $G = (V, E)$:

Connectivity

Cycle

Given a graph $G = (V, E)$:

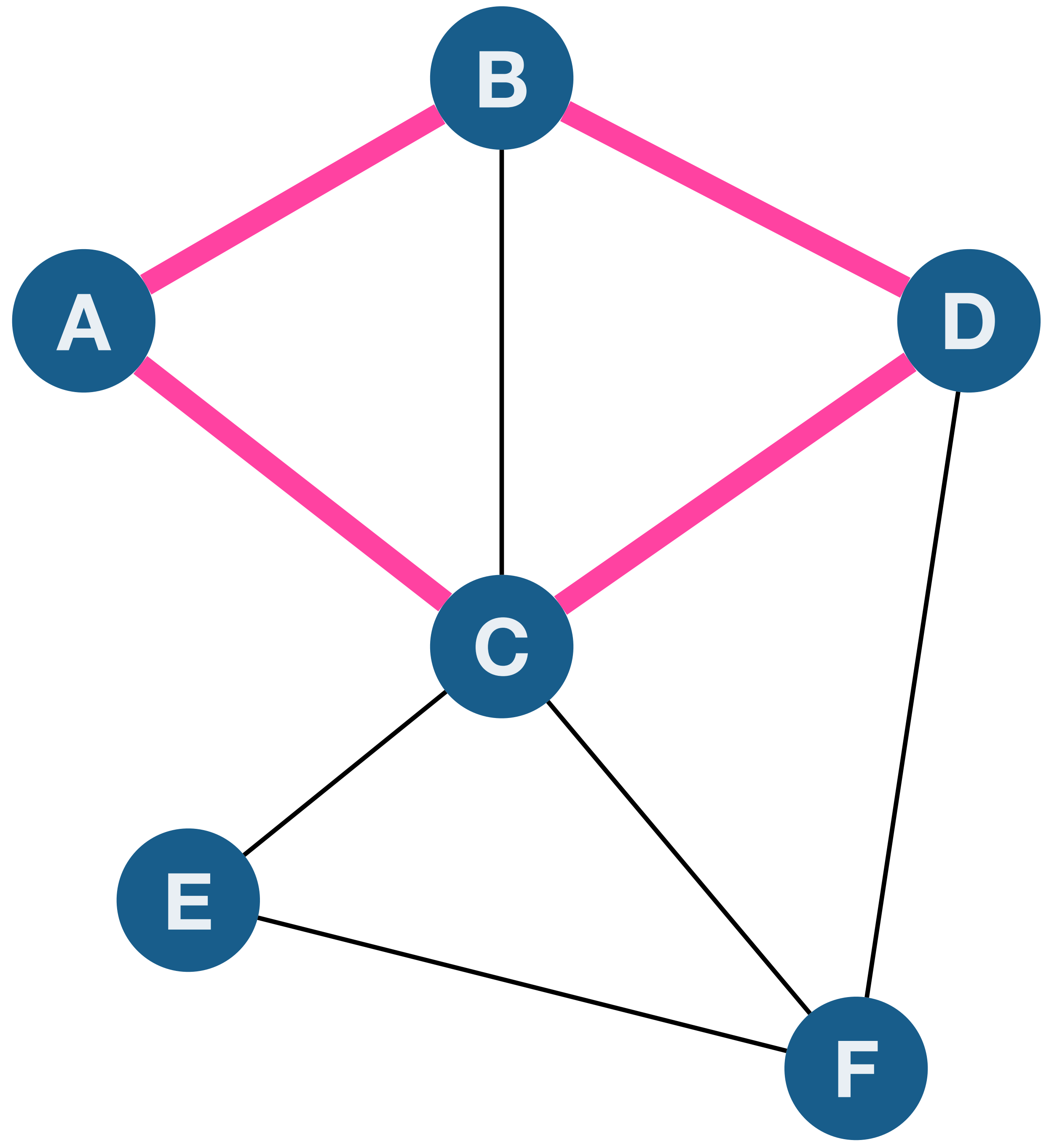
- A *cycle* is a sequence of distinct vertices $v_1, v_2, \ldots, v_k$ with $k \geq 3$ such that $\{v_i, v_{i+1}\} \in E$ for $1 \leq i \leq k - 1$ and $\{v_1, v_k\} \in E$.

Connectivity

Cycle

Given a graph $G = (V, E)$:

- A *cycle* is a sequence of distinct vertices $v_1, v_2, \ldots, v_k$ with $k \geq 3$ such that $\{v_i, v_{i+1}\} \in E$ for $1 \leq i \leq k-1$ and $\{v_1, v_k\} \in E$.

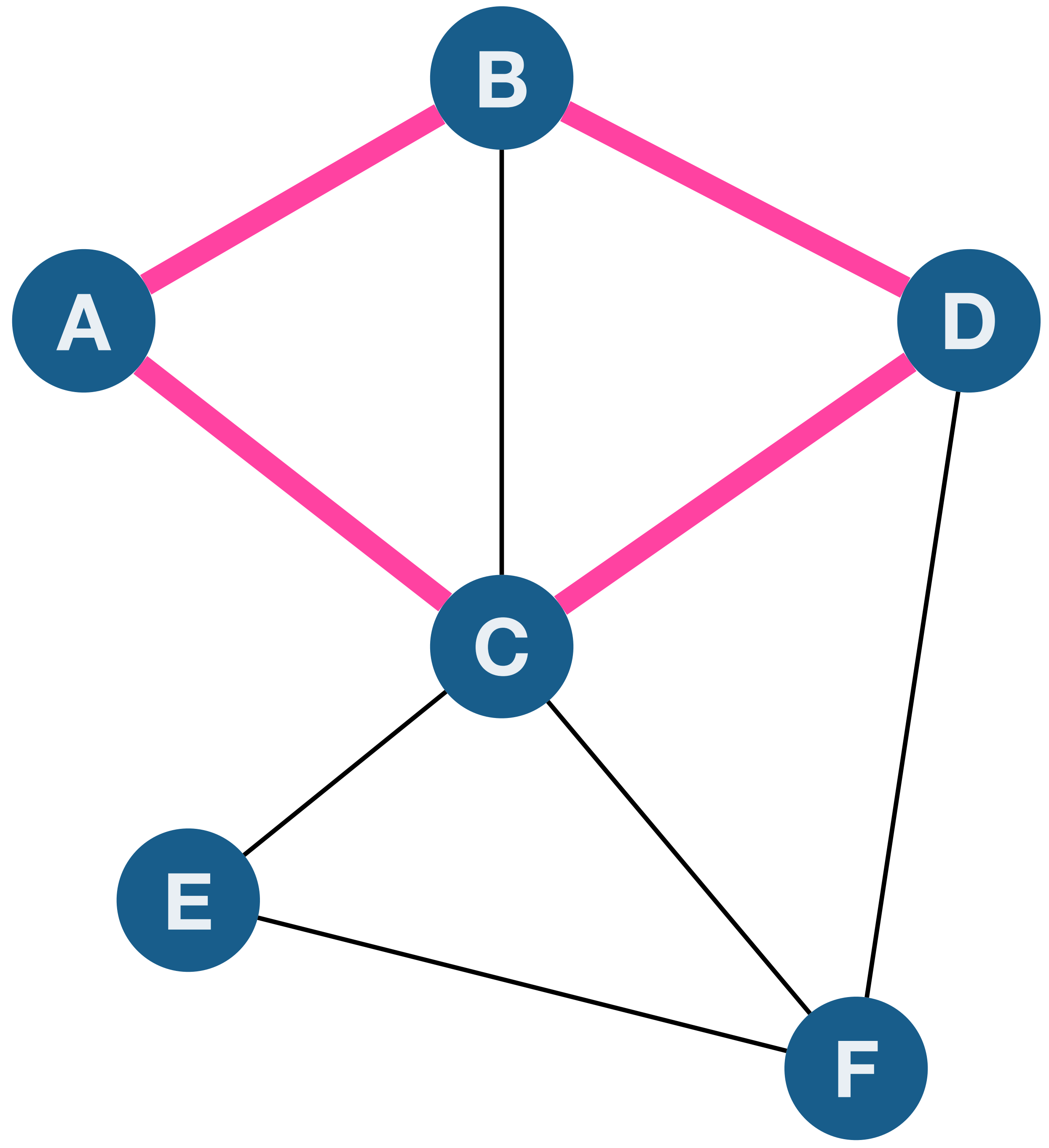


Connectivity

Cycle

Given a graph $G = (V, E)$:

- A *cycle* is a sequence of distinct vertices $v_1, v_2, \ldots, v_k$ with $k \geq 3$ such that $\{v_i, v_{i+1}\} \in E$ for $1 \leq i \leq k-1$ and $\{v_1, v_k\} \in E$.
- Example: **A, B, D, C, A**



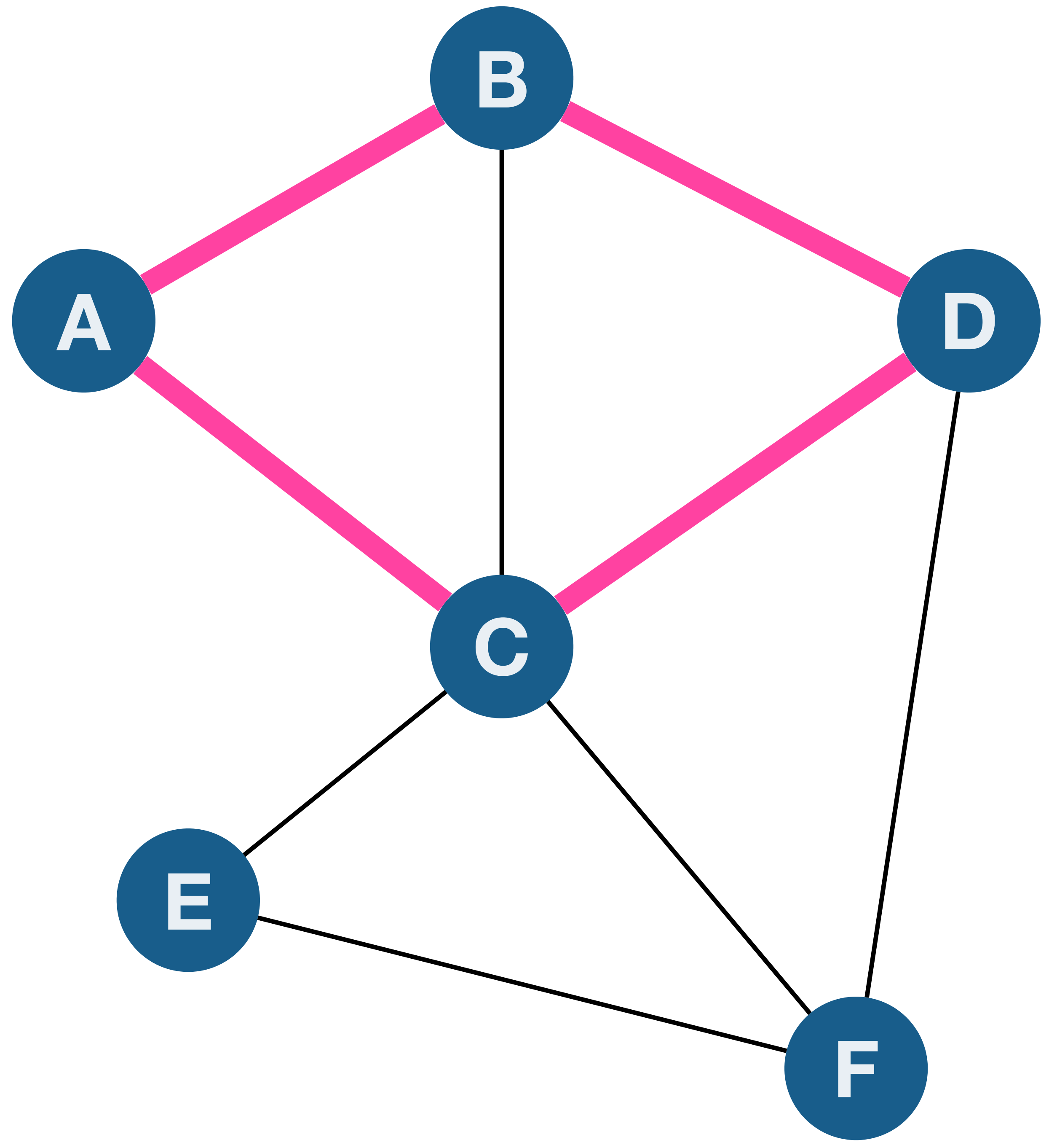
Connectivity

Cycle

Given a graph $G = (V, E)$:

- A *cycle* is a sequence of distinct vertices $v_1, v_2, \ldots, v_k$ with $k \geq 3$ such that $\{v_i, v_{i+1}\} \in E$ for $1 \leq i \leq k-1$ and $\{v_1, v_k\} \in E$.
- Example: **A, B, D, C, A**

Caveat: Some times people use the term *cycle* to also allow vertices to be repeated; we will use the term ***tour.***

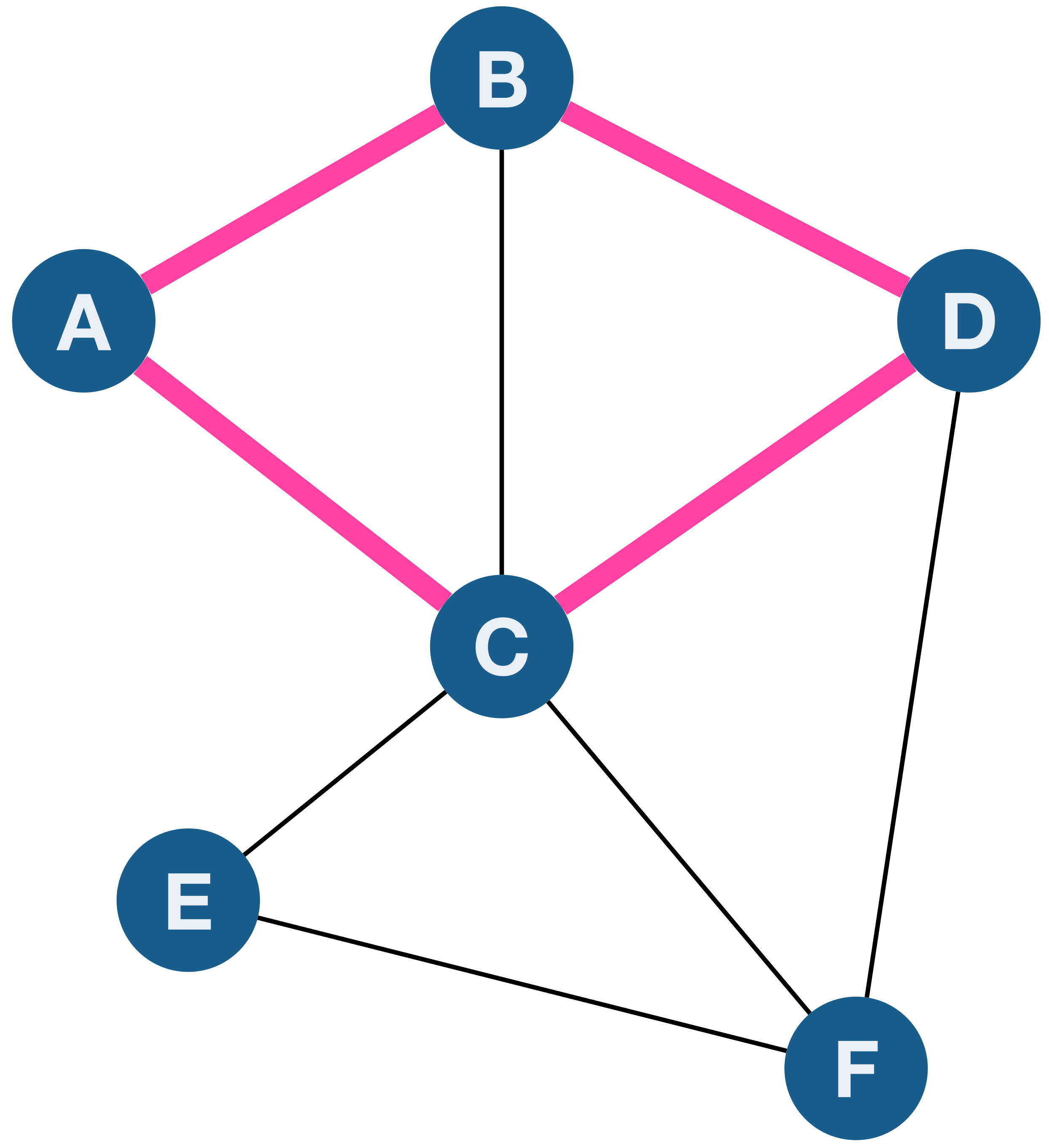


Connectivity

Cycle

Given a graph $G = (V, E)$:

- A *cycle* is a sequence of distinct vertices $v_1, v_2, \ldots, v_k$ with $k \geq 3$ such that $\{v_i, v_{i+1}\} \in E$ for $1 \leq i \leq k-1$ and $\{v_1, v_k\} \in E$.
- Example: **A, B, D, C, A**



Caveat: Some times people use the term *cycle* to also allow vertices to be repeated; we will use the term ***tour.***

Note: A *single* vertex or *an* edge are not cycles according to this definition

Connectivity

Connected components

Define a *relation* C on $V \times V$ as uCv if u is connected to v

vertex is connected to itself

- **Proposition:** In undirected graphs, connectivity is a *reflexive*, *symmetric*, and *transitive* relation.

a∼b and b∼c ⇒ a∼c

a∼b ⇔ b∼a

Connectivity
Connected components

Define a *relation* C on $V \times V$ as uCv if u is connected to v

- **Proposition:** In undirected graphs, connectivity is a *reflexive*, *symmetric*, and *transitive* relation.
- We say that the ***connected components*** of a graph are the *equivalence classes* of C.

Connectivity

Connected components

Define a *relation* C on $V \times V$ as uCv if u is connected to v

- **Proposition:** In undirected graphs, connectivity is a *reflexive*, *symmetric*, and *transitive* relation.
- We say that the ***connected components*** of a graph are the *equivalence classes* of C.
  - "Analogous to ε-reach"

Connectivity

Connected components

Define a *relation* C on $V \times V$ as uCv if u is connected to v

- **Proposition:** In undirected graphs, connectivity is a *reflexive*, *symmetric*, and *transitive* relation.
- We say that the ***connected components*** of a graph are the *equivalence classes* of C.
  - "Analogous to ε-reach" → but that was following ε transitions (NOT necessarily one-hop!!)
- Graph is said to be connected if there is only ***one*** connected component.

Connectivity

Connected components

Define a *relation* C on $V \times V$ as uCv if u is connected to v

- **Proposition:** In undirected graphs, connectivity is a *reflexive*, *symmetric*, and *transitive* relation.
- We say that the ***connected components*** of a graph are the *equivalence classes* of C.
  - "Analogous to ε-reach"
- Graph is said to be connected if there is only ***one*** connected component.
  - In English: starting from any node can reach any other node.

Connectivity problems

Algorithmic problems

- Given graph G and nodes u and v, is u connected to v?
- Given G and node u, find all nodes that are connected to u.
- Find all connected components of G.

Connectivity problems

Algorithmic problems

- Given graph G and nodes u and v, is u connected to v?
- Given G and node u, find all nodes that are connected to u.
- Find all connected components of G.

Can be accomplished in $O(m + n)$ time using **BFS** or **DFS**.

Connectivity problems

Algorithmic problems

- Given graph G and nodes u and v, is u connected to v?
- Given G and node u, find all nodes that are connected to u.
- Find all connected components of G.

Can be accomplished in $O(m+n)$ time using **BFS** or **DFS**.

BFS and **DFS** are flavors of an natural graph exploration algorithm we will call *Basic Search.*

Search on graph

Basic search

Search on graph

Basic search

Not quite list, some people call it a "dispenser". Jeff E calls it a "bag"

Essentially a data structure to track which nodes to explore next.

```
Explore(G,u):
  Initialize: Set Visited[I] ← FALSE for 1 ≤ i ≤ n
  Lists: ToExplore, S
  Add u to ToExplore and to S,
  Visited[u] ← TRUE
  while (ToExplore is non-empty) do
      Remove node x from ToExplore
      for each vertex y in Adj(x) do
          if (Visited[y] = FALSE)
              Visited[y] ← TRUE
              Add y to ToExplore
              Add y to S
  Output S
```

array

list/array → will be all vertices connected to u.

Search on graph

Basic search

- BFS and DFS are special case of the following algorithm.

```
Explore(G,u):
  Initialize: Set Visited[I]← FALSE for 1 ≤ i ≤ n
  Lists: ToExplore, S
  Add u to ToExplore and to S,
  Visited[u] ← TRUE
  while (ToExplore is non-empty) do
        Remove node x from ToExplore
        for each vertex y in Adj(x) do
            if (Visited[y] = FALSE)
                Visited[y] ← TRUE
                Add y to ToExplore
                Add y to S
  Output S
```

Search on graph

Basic search

- BFS and DFS are special case of the following algorithm.
  - BFS maintains *ToExplore* using a **queue** data structure

```
Explore(G,u):
  Initialize: Set Visited[I]← FALSE for 1 ≤ i ≤ n
  Lists: ToExplore, S
  Add u to ToExplore and to S,
  Visited[u] ← TRUE
  while (ToExplore is non-empty) do
       Remove node x from ToExplore
       for each vertex y in Adj(x) do
          if (Visited[y] = FALSE)
              Visited[y] ← TRUE
              Add y to ToExplore
              Add y to S
  Output S
```

Search on graph

Basic search

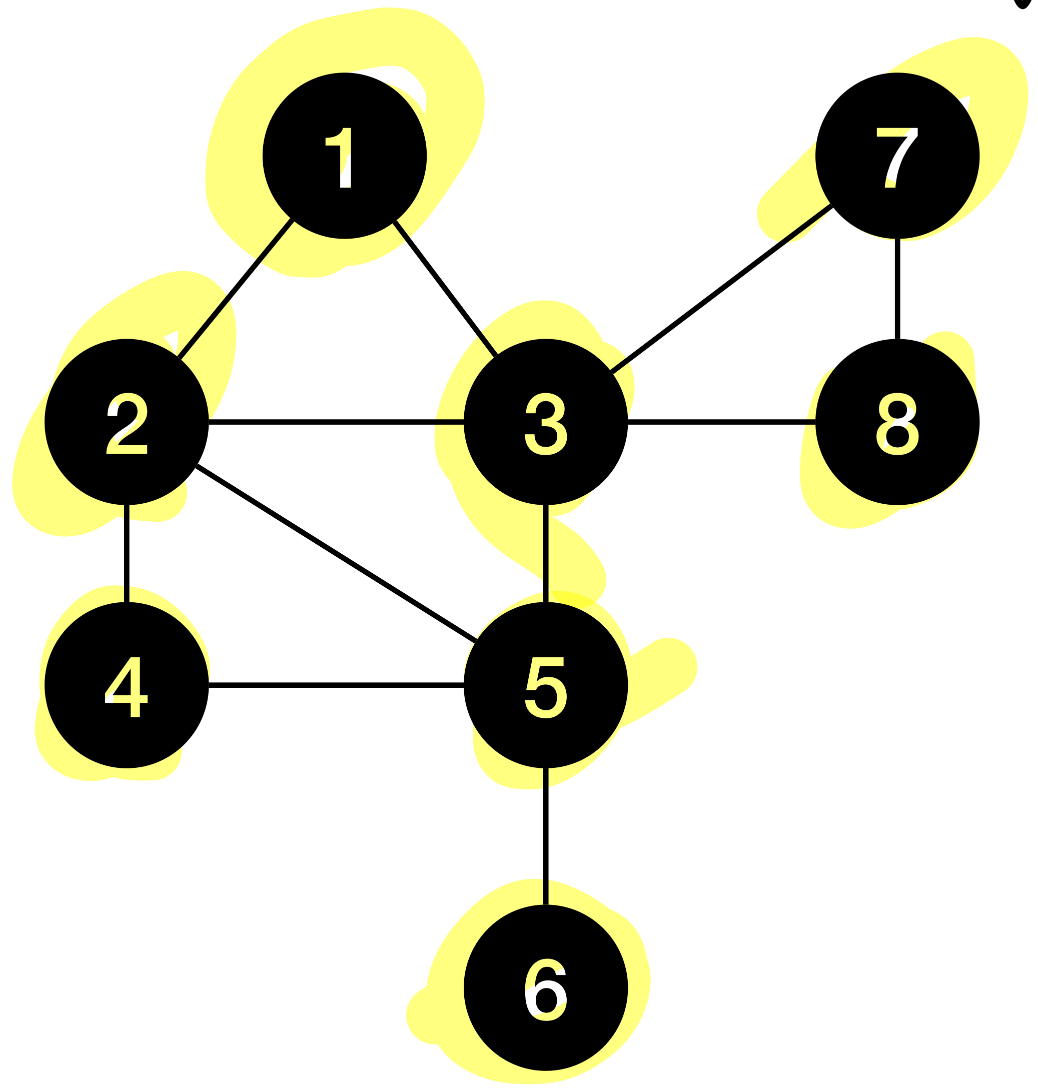
- BFS and DFS are special case of the following algorithm.
  - BFS maintains *ToExplore* using a **queue** data structure
  - DFS maintains *ToExplore* using a **stack** data structure

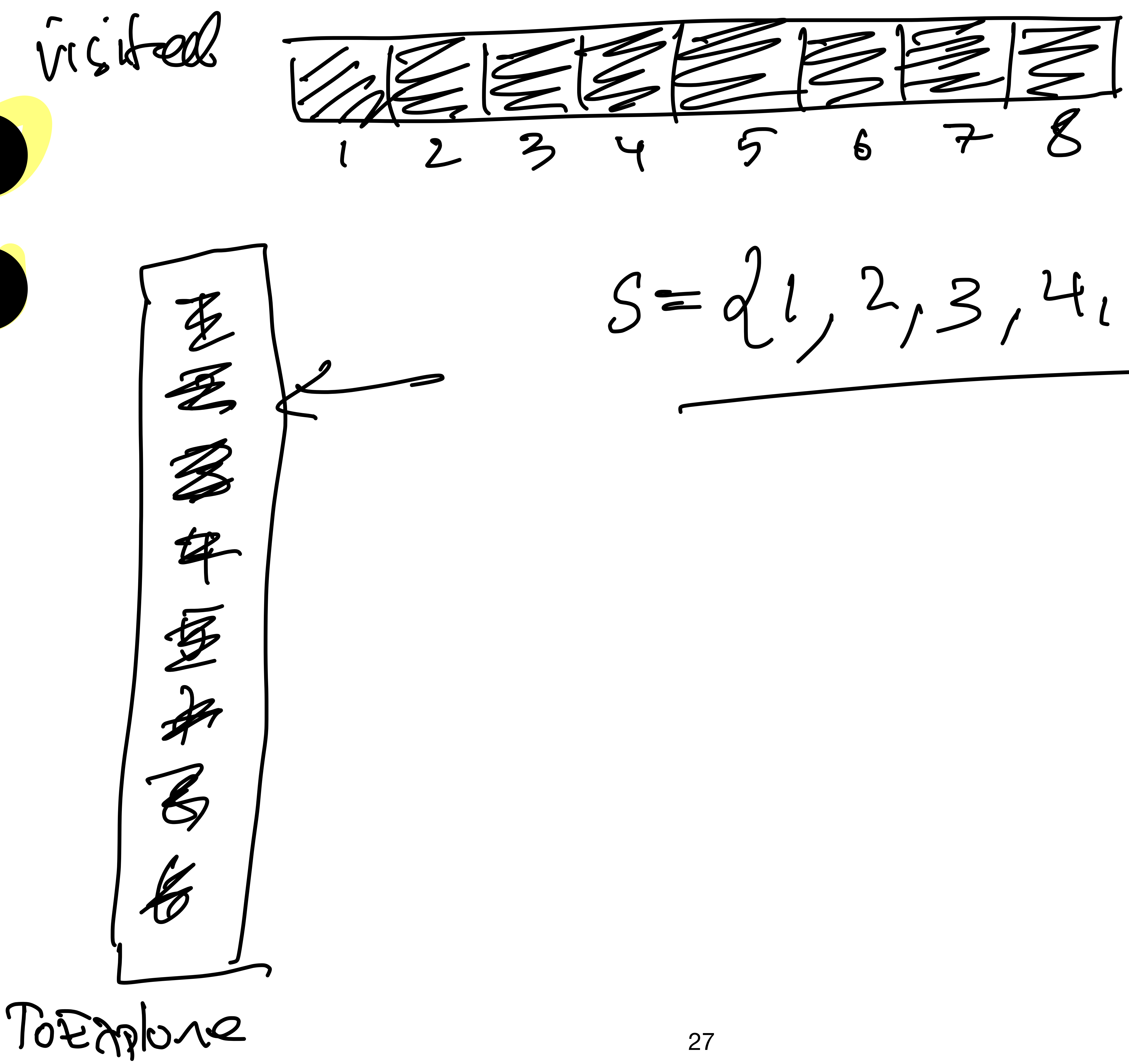
```
Explore(G,u):
  Initialize: Set Visited[I]← FALSE for 1 ≤ i ≤ n
  Lists: ToExplore, S
  Add u to ToExplore and to S,
  Visited[u] ← TRUE
  while (ToExplore is non-empty) do
       Remove node x from ToExplore
       for each vertex y in Adj(x) do
          if (Visited[y] = FALSE)
              Visited[y] ← TRUE
              Add y to ToExplore
              Add y to S
  Output S
```

Search on graph

Example - maintain *ToExplore* as a queue

```
Explore(G,u):
  Initialize: Set Visited[I]← FALSE for 1≤i≤n
  Lists: ToExplore, S
  Add u to ToExplore and to S,
  Visited[u] ← TRUE
  while (ToExplore is non-empty) do
       Remove node x from ToExplore
       for each vertex y in Adj(x) do
          if (Visited[y] = FALSE)
              Visited[y] ← TRUE
              Add y to ToExplore
              Add y to S
  Output S
```





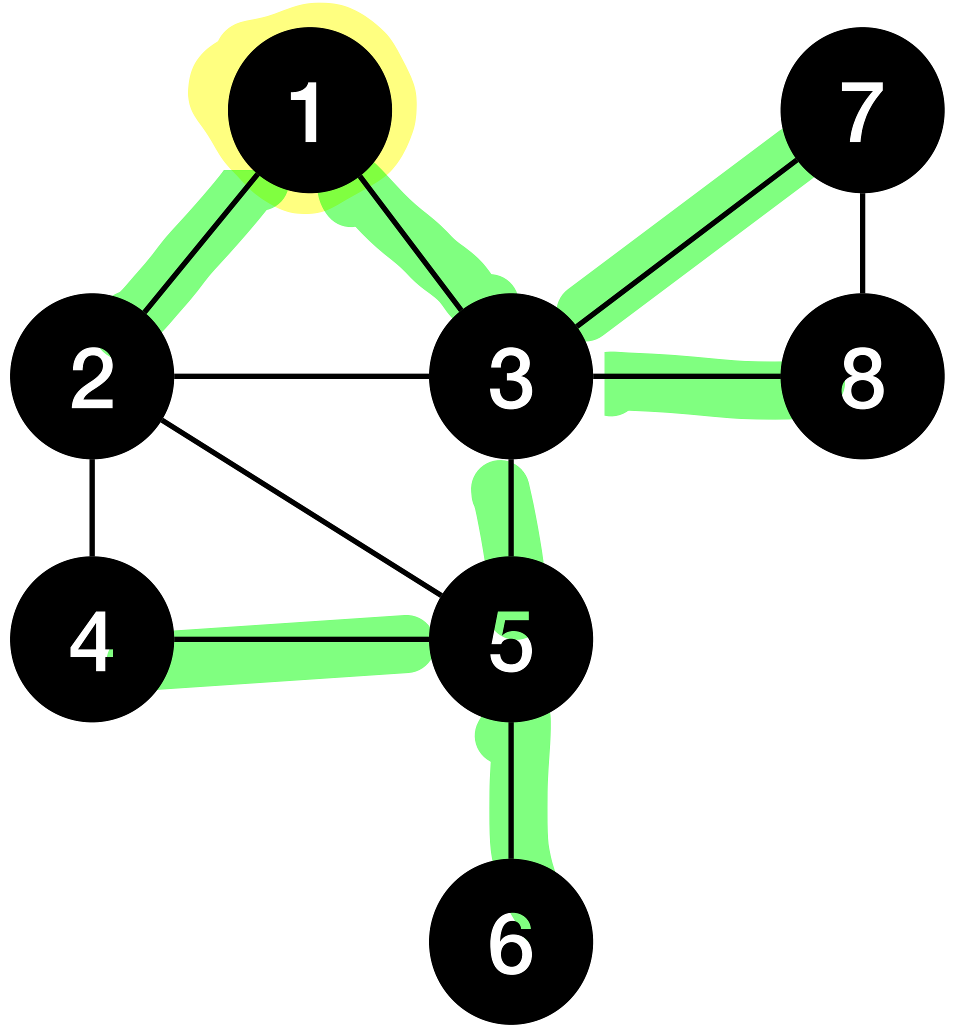
Let u = 1

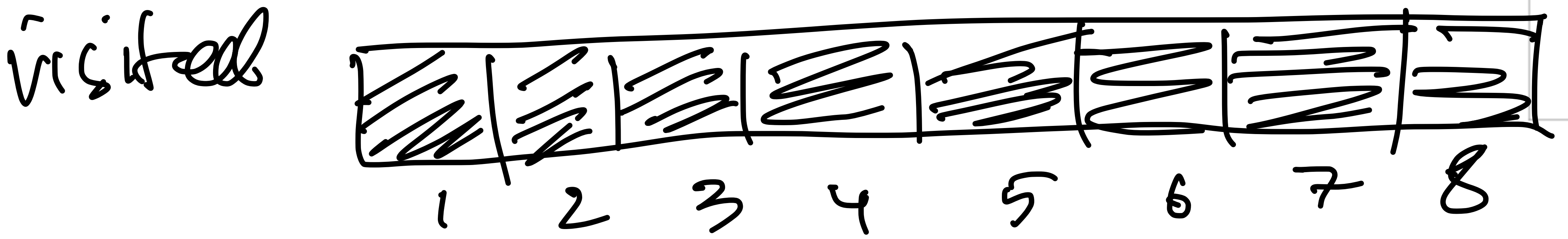
S = {1, 2, 3, 4, 5, 7, 8, 6}

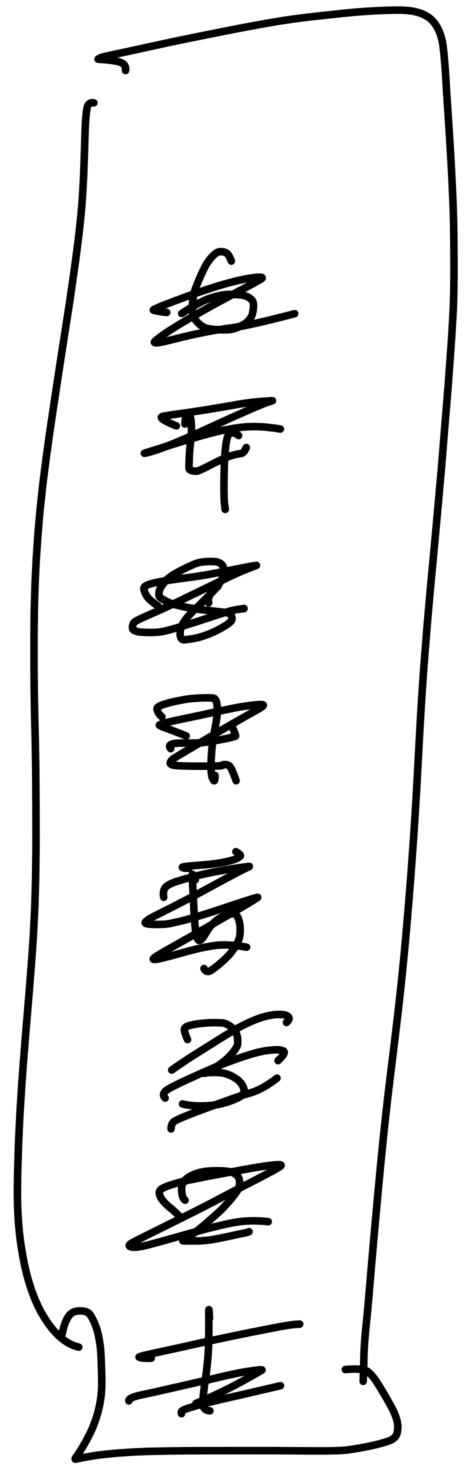
Search on graph

Exercise - maintain *ToExplore* as a stack

```
Explore(G,u):
  Initialize: Set Visited[I]← FALSE for 1≤i≤n
  Lists: ToExplore, S
  Add u to ToExplore and to S,
  Visited[u] ← TRUE
  while (ToExplore is non-empty) do
        Remove node x from ToExplore
        for each vertex y in Adj(x) do
           if (Visited[y] = FALSE)
                Visited[y] ← TRUE
                Add y to ToExplore
                Add y to S
  Output S
```







To Explore .

$$S = \{1, 2, 3, 5, 7, 8, 4, 6\}$$

Search on graph

Basic search - modified to get search tree

```
Explore(G,u):
  array Visited[1..n]
  Initialize: Set Visited[I]← FALSE for 1 ≤ i ≤ n
  List: ToExplore, S
  Add u to ToExplore and to S, Visited[u] ← TRUE
  Make tree T with root as u
  while (ToExplore is non-empty) do
        Remove node x from ToExplore
        for each vertex y in Adj(x) do
            if (Visited[y] = FALSE)
                Visited[y] ← TRUE
                Add y to ToExplore
                Add y to S
                Add y to T with x as parent
  Output S, T
```

Search on graph

Basic search - modified to get search tree

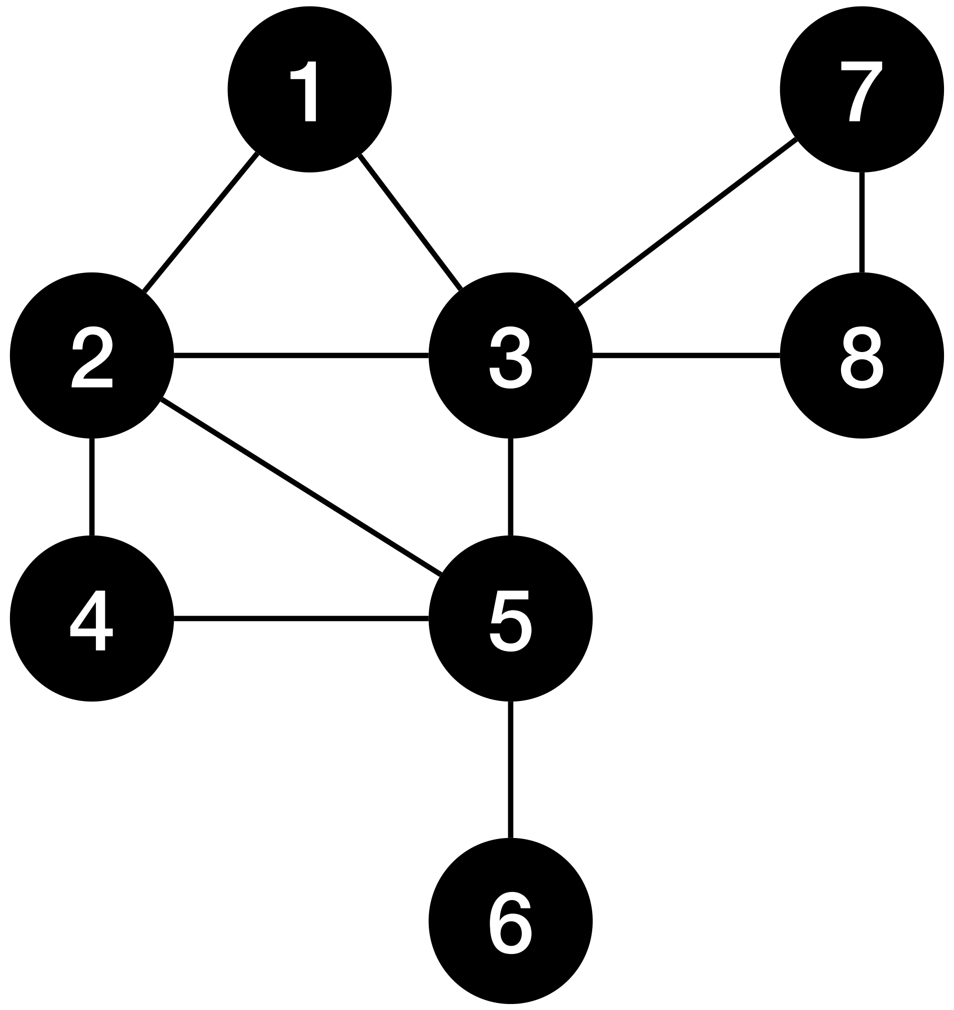
- The *search tree* for **Explore(G, u)** is tree *rooted* at **u** that spans the connected component of u.

```
Explore(G,u):
  array Visited[1..n]
  Initialize: Set Visited[I]← FALSE for 1 ≤ i ≤ n
  List: ToExplore, S
  Add u to ToExplore and to S, Visited[u] ← TRUE
  Make tree T with root as u
  while (ToExplore is non-empty) do
       Remove node x from ToExplore
       for each vertex y in Adj(x) do
          if (Visited[y] = FALSE)
              Visited[y] ← TRUE
              Add y to ToExplore
              Add y to S
              Add y to T with x as parent
  Output S, T
```

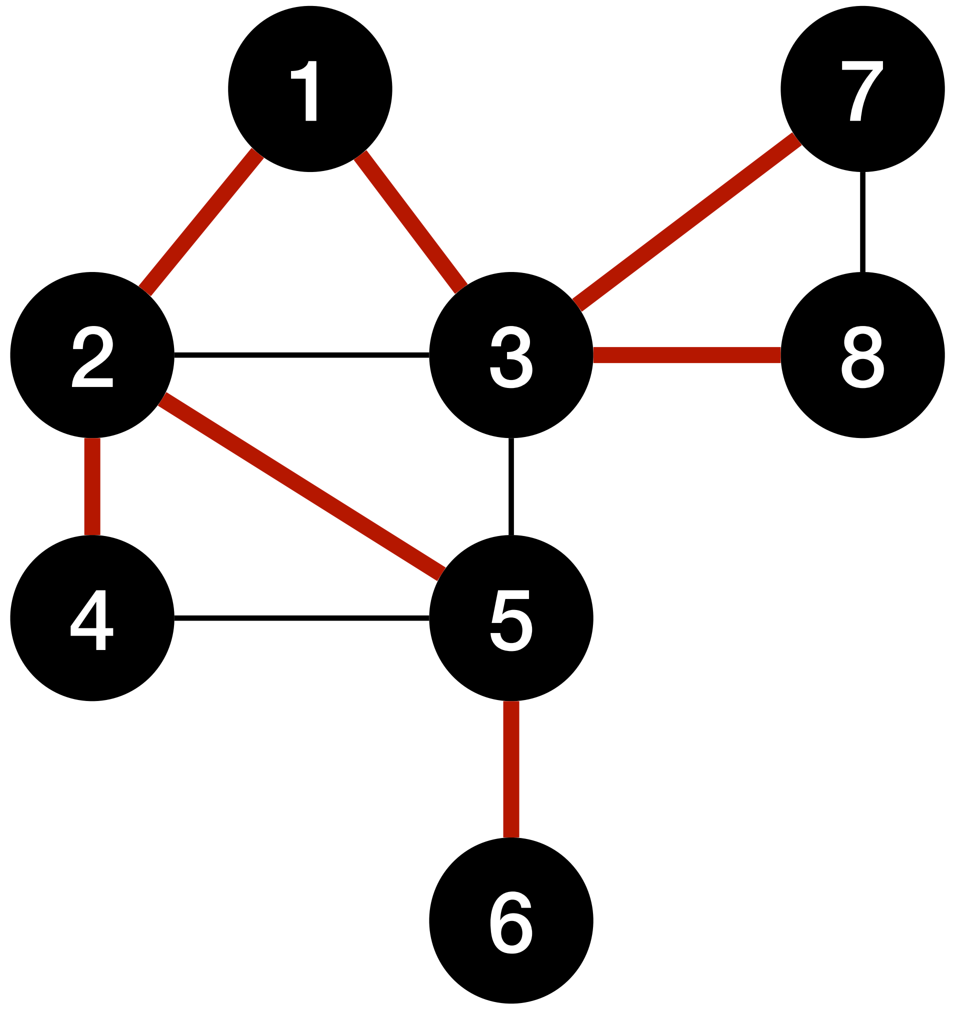
Search on graph

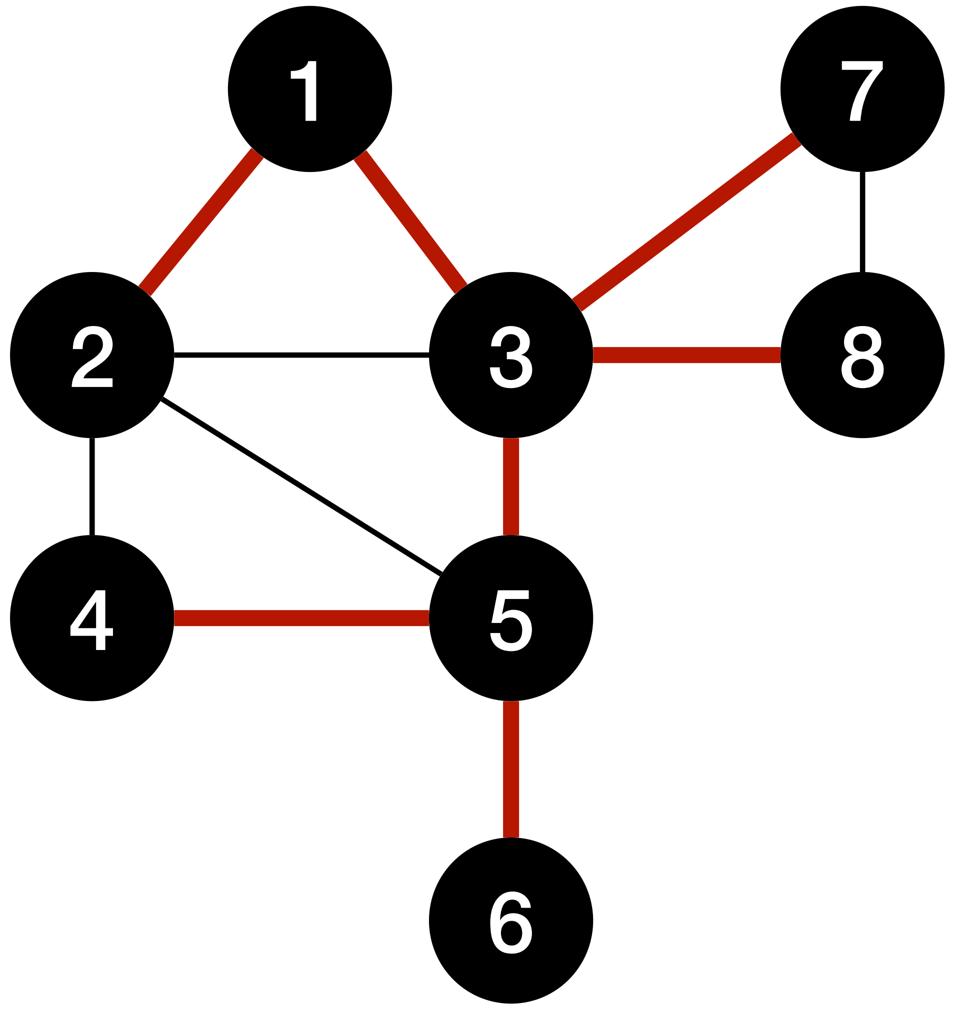
Basic search - modified to get search tree

- BFS and DFS will return different search trees on the following graph



Verify these !! Which is BFS? DFS?





Directed graphs

Directed graphs

Definition

A directed graph $G = (V, E)$ consists of

- A set of vertices/nodes V and
- A set of edges $E \subseteq V \times V$.

Directed graphs

Definition

A directed graph $G = (V, E)$ consists of

- A set of vertices/nodes V and
- A set of edges $E \subseteq V \times V$.

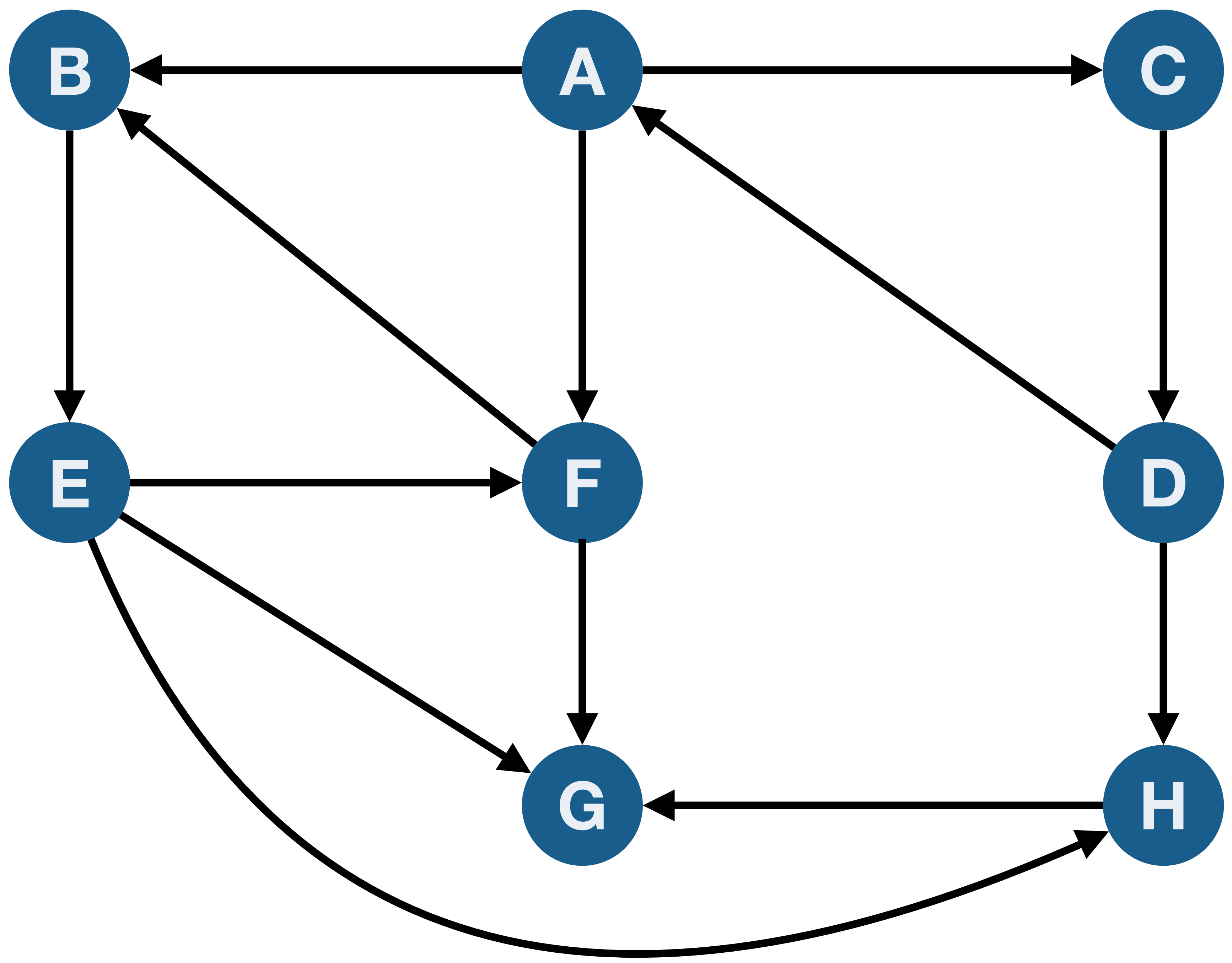
An edge is an **ordered pair** of vertices: (u, v) different from (v, u)

Directed graphs
Definition

A directed graph $G = (V, E)$ consists of

- A set of vertices/nodes V and
- A set of edges $E \subseteq V \times V$.

An edge is an **ordered pair** of vertices: (u, v) different from (v, u)



Directed graphs

Examples

In many situations relationship between vertices is asymmetric:

Directed graphs

Examples

In many situations relationship between vertices is asymmetric:

- **Road networks** with one-way streets.

Directed graphs
Examples

In many situations relationship between vertices is asymmetric:

- **Road networks** with one-way streets.
- **Web-link graph** where vertices are web-pages and there is an edge from page p to page p' if p has a link to p' .

Directed graphs

Examples

In many situations relationship between vertices is asymmetric:

- **Road networks** with one-way streets.
- **Web-link graph** where vertices are web-pages and there is an edge from page p to page p' if p has a link to p' .
- **Dependency graphs** in variety of applications: link from x to y if y depends on x. E.g. Make files for compiling programs.

Directed graphs

Examples

In many situations relationship between vertices is asymmetric:

- **Road networks** with one-way streets.
- **Web-link graph** where vertices are web-pages and there is an edge from page p to page p' if p has a link to p' .
- **Dependency graphs** in variety of applications: link from x to y if y depends on x. E.g. Make files for compiling programs.
- **Program analysis:** functions/procedures are vertices and there is an edge from x to y if x calls y.

Directed graphs

Representation

Graph $G = (V, E)$ with n vertices and m edges:

$A_{uG} = A_{uG}^T \iff a_{ij} = a_{ji}$

Directed graphs

Representation

Graph $G = (V, E)$ with n vertices and m edges:

- **Adjacency matrix**: $n \times n$ asymmetric matrix A. $a_{ij} = 1$ if $(i, j) \in E$ and $a_{ij} = 0$ if $(i, j) \notin E$.

Directed graphs

Representation

Graph $G = (V, E)$ with n vertices and m edges:

- **Adjacency matrix**: $n \times n$ asymmetric matrix A. $a_{ij} = 1$ if $(i, j) \in E$ and $a_{ij} = 0$ if $(i, j) \notin E$.
- **Adjacency lists**: For each node u, $\mathrm{Out}(u)$ (also referred to as $\mathrm{Adj}(u)$ by default) stores out-going edges from u .

Directed graphs

Representation

Graph $G = (V, E)$ with n vertices and m edges:

- **Adjacency matrix**: $n \times n$ asymmetric matrix A. $a_{ij} = 1$ if $(i, j) \in E$ and $a_{ij} = 0$ if $(i, j) \notin E$.

- **Adjacency lists**: For each node u, $\mathrm{Out}(u)$ (also referred to as $\mathrm{Adj}(u)$ by default) stores out-going edges from u .

  - Can also have $\mathrm{In}(u)$ and store in-coming edges to u.

Directed graphs

Representation

Graph $G = (V, E)$ with n vertices and m edges:

- **Adjacency matrix**: $n \times n$ asymmetric matrix A. $a_{ij} = 1$ if $(i, j) \in E$ and $a_{ij} = 0$ if $(i, j) \notin E$.
- **Adjacency lists**: For each node u, $\mathrm{Out}(u)$ (also referred to as $\mathrm{Adj}(u)$ by default) stores out-going edges from u .
  - Can also have $\mathrm{In}(u)$ and store in-coming edges to u.

Default representation is adjacency lists ($\mathrm{Adj}(u) \sim \mathrm{Out}(u)$).

Directed connectivity

Given a graph $G = (V, E)$:

Directed connectivity

$e \in E$ is an ordered tuple now.

Given a graph $G = (V, E)$:

- A *(directed) path* is a sequence of distinct vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k - 1$. The length of the path is $k - 1$ and the path is from v_1 to v_k. By convention, a single node u is a path of length 0.

Directed connectivity

Given a graph $G = (V, E)$:

- A *(directed) path* is a sequence of distinct vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k-1$. The length of the path is $k-1$ and the path is from v_1 to v_k. By convention, a single node u is a path of length 0.

- A *cycle* is a sequence of distinct vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k-1$ and $(v_k, v_1) \in E$. By convention, a single node u is not a cycle.

Q: is there such a thing as "undirected" path on a directed graph?

Directed connectivity

Given a graph $G = (V, E)$:

- A *(directed) path* is a sequence of distinct vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k-1$. The length of the path is $k-1$ and the path is from v_1 to v_k. By convention, a single node u is a path of length 0.
- A *cycle* is a sequence of distinct vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k-1$ and $(v_k, v_1) \in E$. By convention, a single node u is not a cycle.
- A vertex u can reach v if there is a path from u to v. Alternatively, we say v can be reached from u.

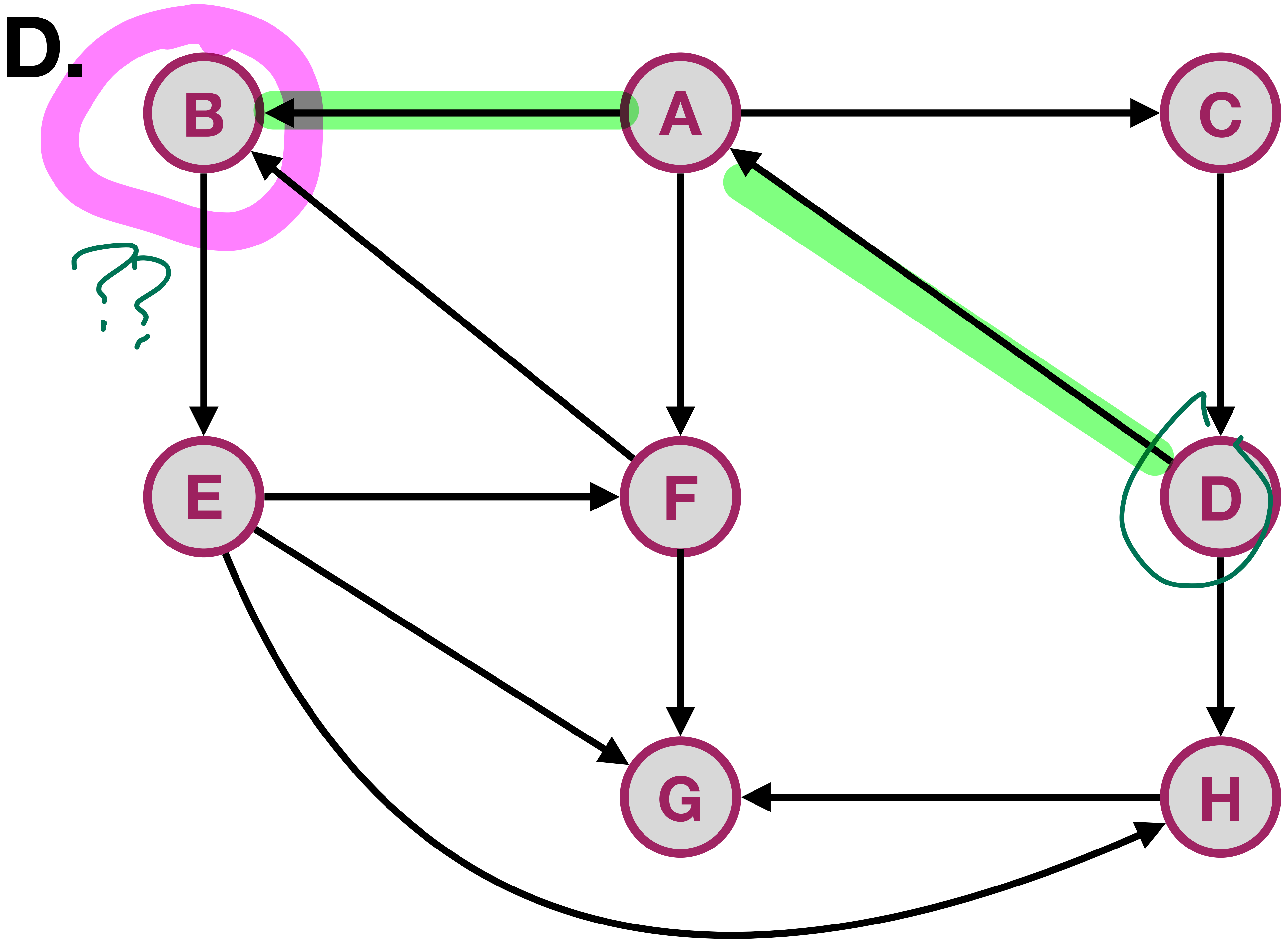
Directed connectivity

Given a graph $G = (V, E)$:

- A *(directed) path* is a sequence of distinct vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k - 1$. The length of the path is $k - 1$ and the path is from v_1 to v_k. By convention, a single node u is a path of length 0.
- A *cycle* is a sequence of distinct vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k - 1$ and $(v_k, v_1) \in E$. By convention, a single node u is not a cycle.
- A vertex u can reach v if there is a path from u to v. Alternatively, we say v can be reached from u.
- We denote with $\mathrm{rch}(u)$ the set of all vertices *reachable* from u.

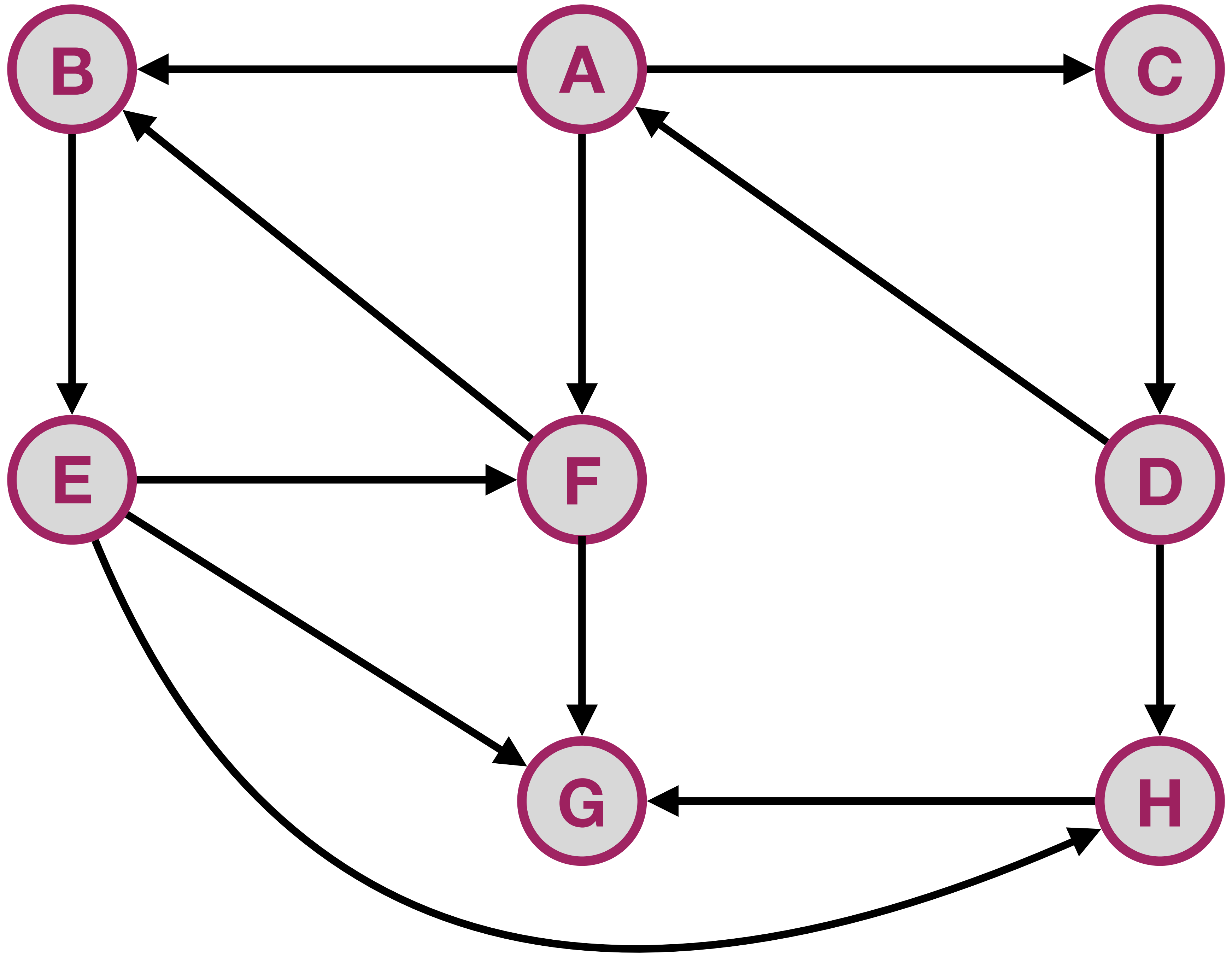
Directed connectivity

Asymmetricity: **D** can reach **B** but **B** cannot reach **D.**



Directed connectivity

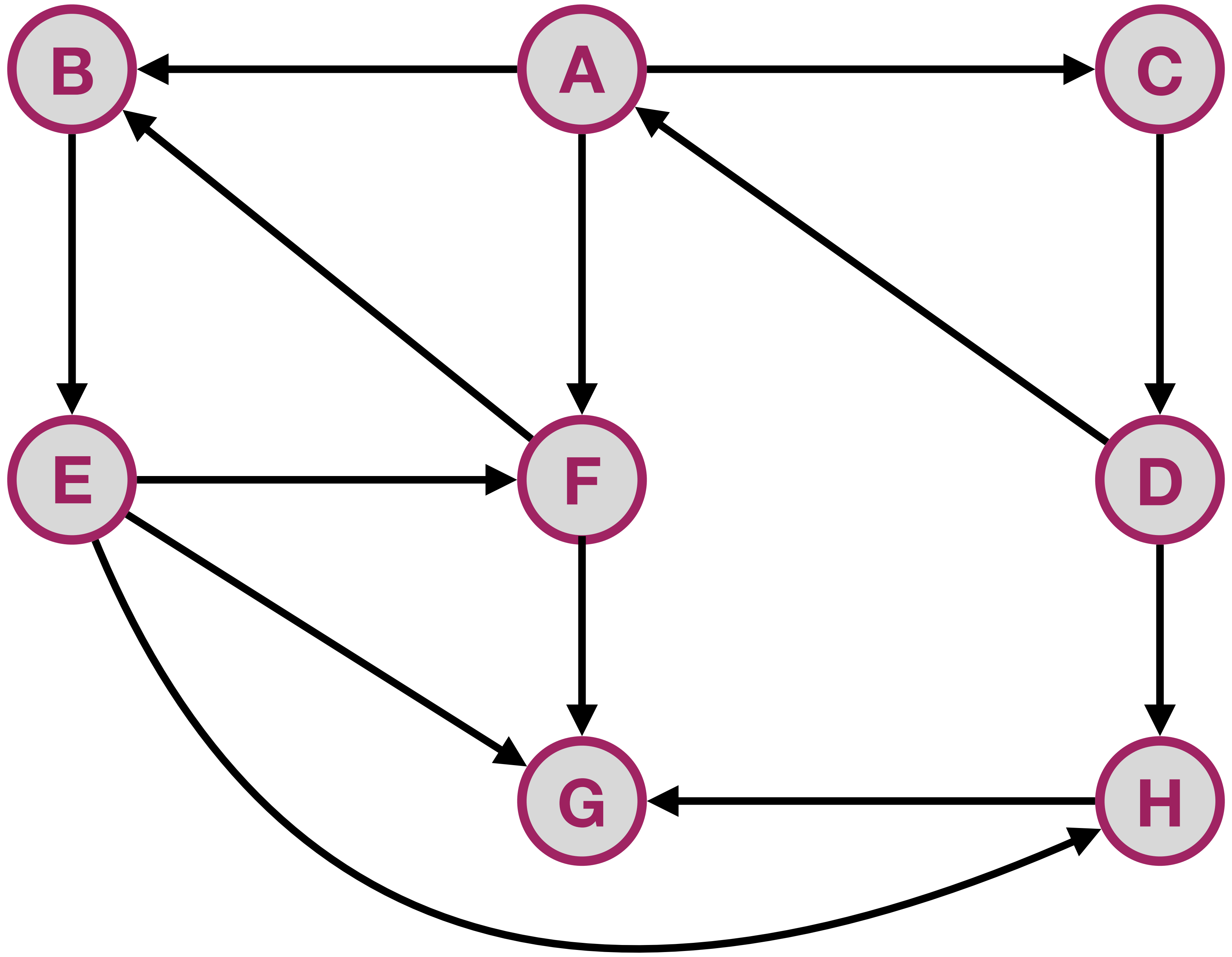
Asymmetricity: **D** can reach **B** but **B** cannot reach **D.**



Questions:

Directed connectivity

Asymmetricity: **D** can reach **B** but **B** cannot reach **D.**

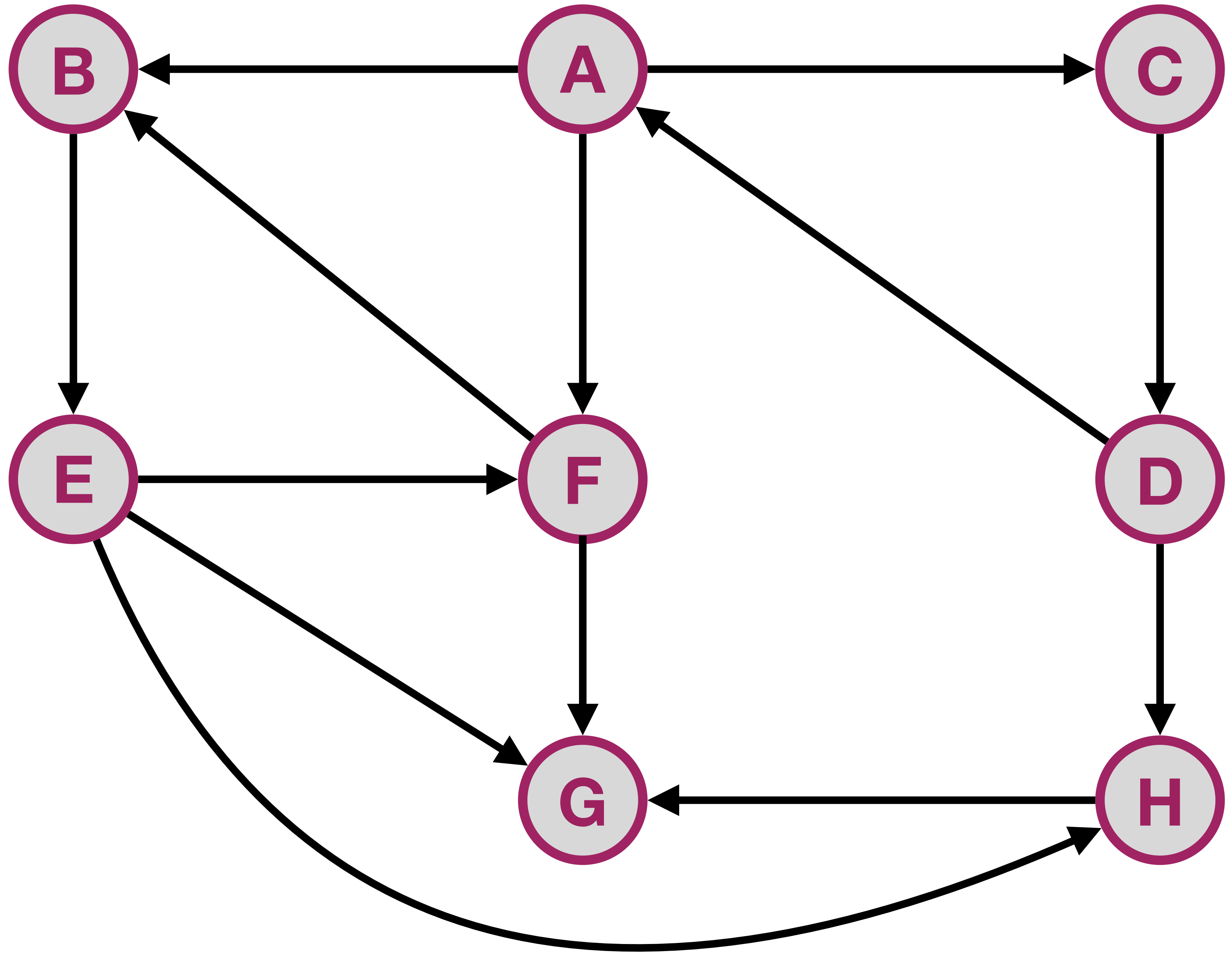


Questions:

Is there a notion of connected components?

Directed connectivity

Asymmetricity: **D** can reach **B** but **B** cannot reach **D.**



Questions:

Is there a notion of connected components?

How do we understand connectivity in directed graphs?

Connectivity and strongly connected components

Definition: Given a directed graph G, u is ***strongly connected*** to v if u can reach v **and** v can reach u. In other words $v \in \text{rch}(u)$ and $u \in \text{rch}(v)$.

Connectivity and strongly connected components

Definition: Given a directed graph G, u is ***strongly connected*** to v if u can reach v **and** v can reach u. In other words $v \in \text{rch}(u)$ and $u \in \text{rch}(v)$.

Proposition: Define relation C where uCv if u is (strongly) connected to v. Then C is an equivalence relation, that is *reflexive*, *symmetric* & *transitive.*

Connectivity and strongly connected components

Definition: Given a directed graph G, u is ***strongly connected*** to v if u can reach v **and** v can reach u. In other words $v \in \text{rch}(u)$ and $u \in \text{rch}(v)$.

Proposition: Define relation C where uCv if u is (strongly) connected to v. Then C is an equivalence relation, that is *reflexive*, *symmetric* & *transitive.*

Equivalence classes of C are the strongly connected components of G and they partition the vertices of G.

Connectivity and strongly connected components

Definition: Given a directed graph G, u is ***strongly connected*** to v if u can reach v **and** v can reach u. In other words $v \in \text{rch}(u)$ and $u \in \text{rch}(v)$.

Proposition: Define relation C where uCv if u is (strongly) connected to v. Then C is an equivalence relation, that is *reflexive*, *symmetric* & *transitive.*

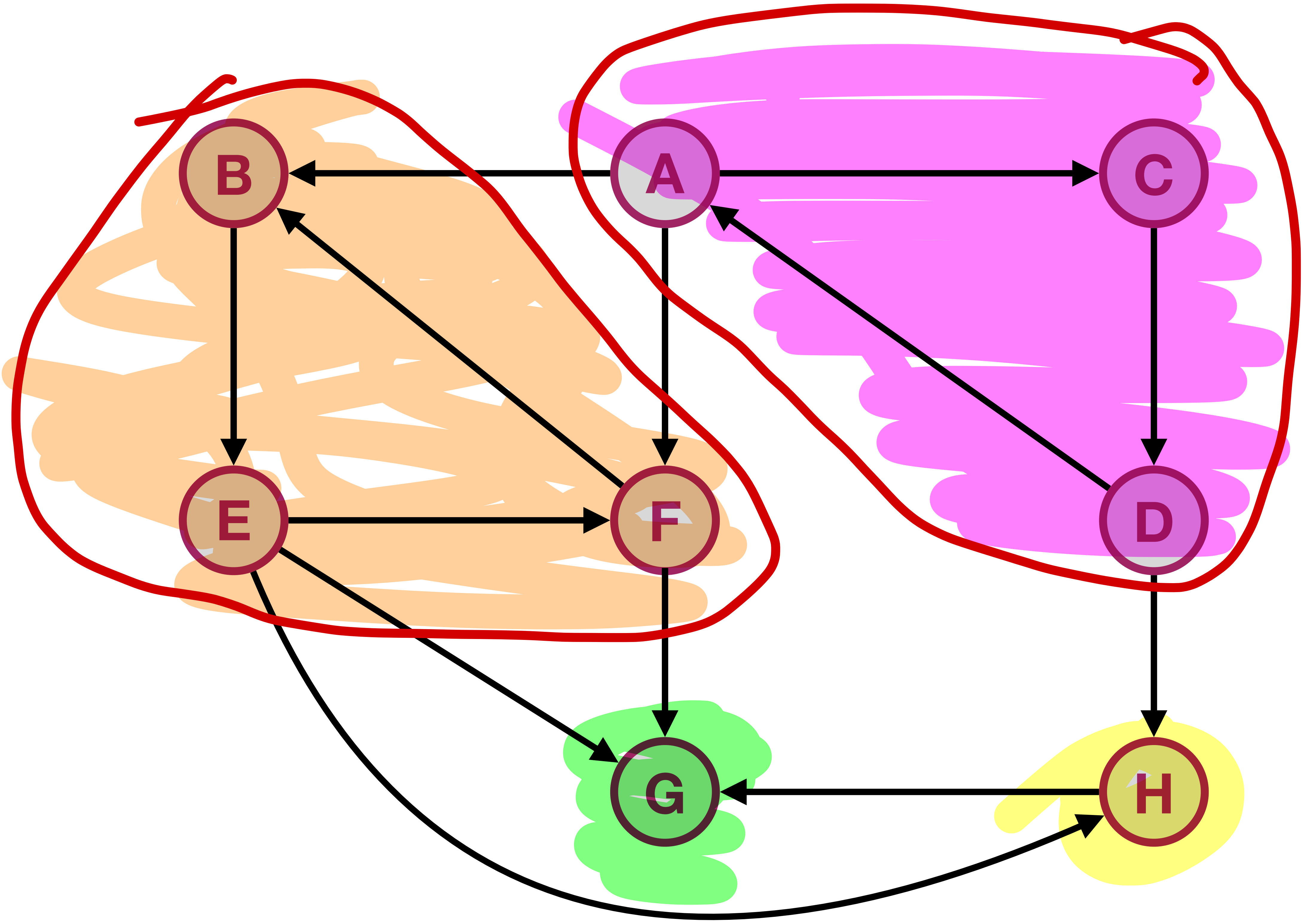
Equivalence classes of C are the strongly connected components of G and they partition the vertices of G.

We denote with $SCC(u)$ the strongly connected component containing u.

Connectivity and strongly connected components

Exercise

- Partition vertices of given graph under strong connectivity.



Directed graph connectivity problems

1. Given G and nodes u and v, can u reach v?
2. Given G and u, compute rch(u).
3. Given G and u, compute all v that can reach u, that is all v such that $u \in$ rch(v).
4. Find the strongly connected component containing node u, that is $SCC(u)$.
5. Is G strongly connected (a single strong component)?
6. Compute all strongly connected components of G.

sort of reverse question

Graph exploration in directed graphs

Directed graph search

Given $G = (V, E)$
a directed graph and
vertex $u \in V$.
Let $n = |V|$.

Directed graph search

Given $G = (V, E)$ a directed graph and vertex $u \in V$.

Let $n = |V|$.

```
Explore(G,u):
  array Visited[1..n]
  Initialize: Set Visited[I]← FALSE for 1 ≤ i ≤ n
  List: ToExplore, S
  Add u to ToExplore and to S, Visited[u] ← TRUE
  Make tree T with root as u
  while (ToExplore is non-empty) do
       Remove node x from ToExplore
       for each vertex y in Adj(x) do
           if (Visited[y] = FALSE)
               Visited[y] ← TRUE
               Add y to ToExplore
               Add y to S
               Add y to T with x as parent
  Output S, T
```

Directed graph search

Given $G = (V, E)$ a directed graph and vertex $u \in V$.

Let $n = |V|$.

We seek to find all nodes that can be reached from u (represented as a ***spanning*** tree).

```
Explore(G,u):
  array Visited[1..n]
  Initialize: Set Visited[I]← FALSE for 1 ≤ i ≤ n
  List: ToExplore, S
  Add u to ToExplore and to S, Visited[u] ← TRUE
  Make tree T with root as u
  while (ToExplore is non-empty) do
      Remove node x from ToExplore
      for each vertex y in Adj(x) do
         if (Visited[y] = FALSE)
            Visited[y] ← TRUE
            Add y to ToExplore
            Add y to S
            Add y to T with x as parent
  Output S, T
```

→ previously S was nodes "connected" to u.

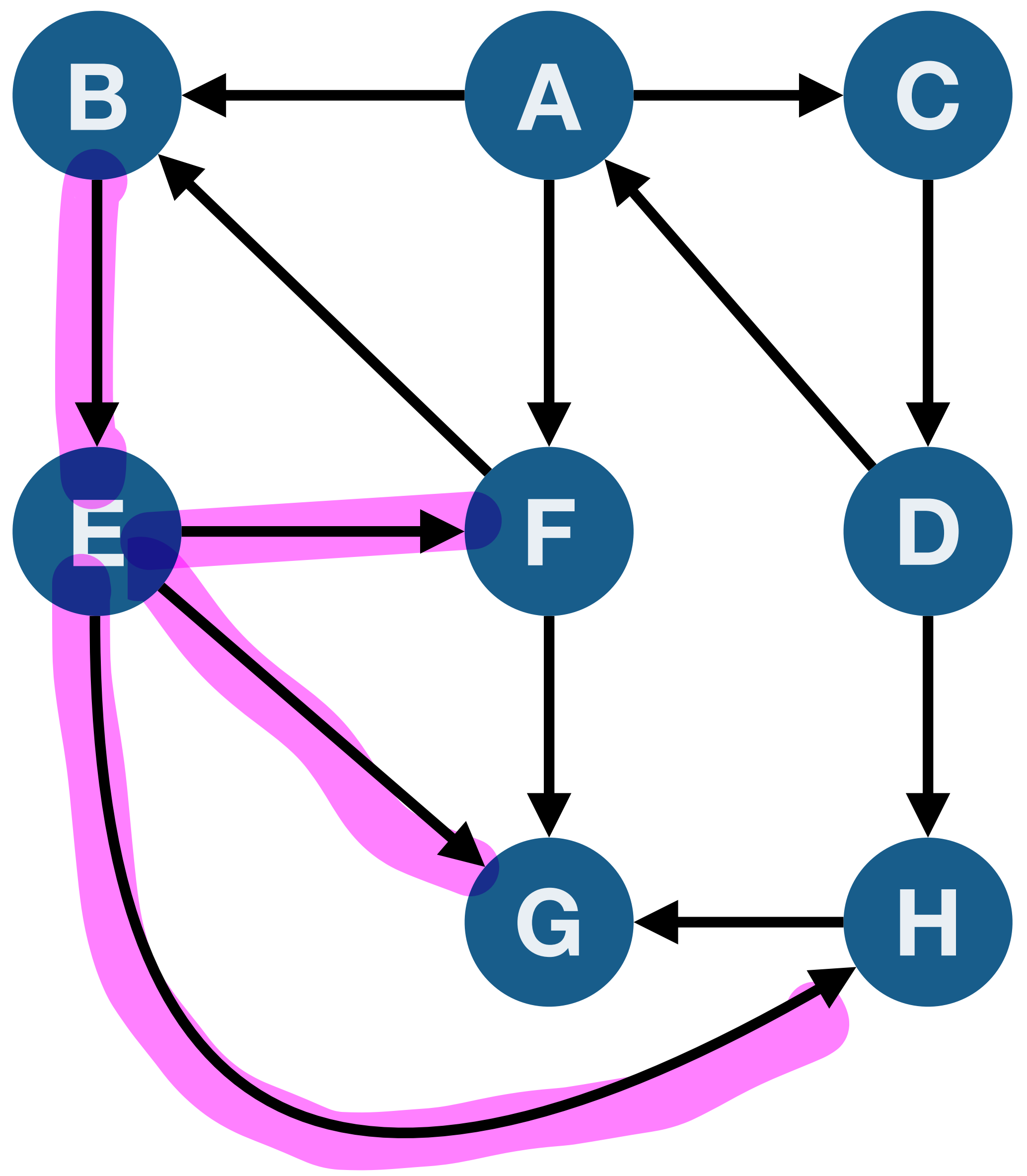
→ now it is nodes "reachable" from u.

Directed graph search

Example

```
Explore(G,u):
  array Visited[1..n]
  Initialize: Set Visited[I]← FALSE for 1 ≤ i ≤ n
  List: ToExplore, S
  Add u to ToExplore and to S, Visited[u] ← TRUE
  Make tree T with root as u
  while (ToExplore is non-empty) do
      Remove node x from ToExplore
      for each vertex y in Adj(x) do
          if (Visited[y] = FALSE)
              Visited[y] ← TRUE
              Add y to ToExplore
              Add y to S
              Add y to T with x as parent
  Output S, T
```

visited: A B C D E F G H



S = {B, E, F, G, H}

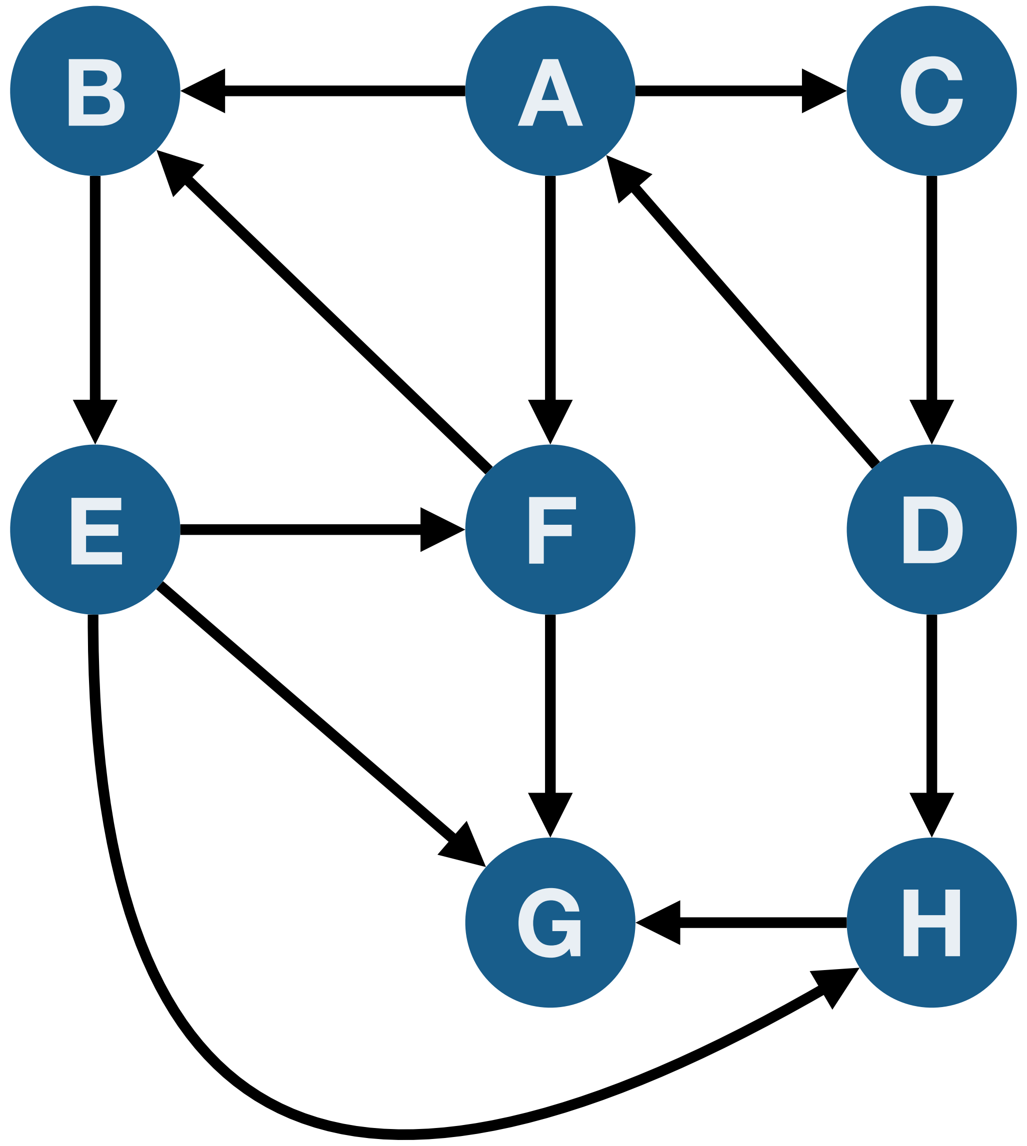
stack(B).

wrap the another while loop to visit all nodes

```
Explore(G,u):
  array Visited[1..n]
  Initialize: Set Visited[I]← FALSE for 1 ≤ i ≤ n
  List: ToExplore, S
  Add u to ToExplore and to S, Visited[u] ← TRUE
  Make tree T with root as u
  while (ToExplore is non-empty) do
       Remove node x from ToExplore
       for each vertex y in Adj(x) do
          if (Visited[y] = FALSE)
              Visited[y] ← TRUE
              Add y to ToExplore
              Add y to S
              Add y to T with x as parent
  Output S, T
```

Directed graph search

Example



proof skipped.
(see Prof. Kani's old slides for a sketch)

Proposition: *Explore(G,u)* terminates with S being $\mathrm{rch}(u)$.

Directed graph connectivity problems

1. Given G and nodes u and v, can u reach v?
2. Given G and u, compute rch(u).
3. Given G and u, compute all v that can reach u, that is all v such that $u \in$ rch(v).
4. Find the strongly connected component containing node u, that is $SCC(u)$.
5. Is G strongly connected (a single strong component)?
6. Compute all strongly connected components of G.

Directed graph connectivity problems

already discussed.

1. Given G and nodes u and v, can u reach v?
2. Given G and u, compute rch(u).

Use Explore(G, u) to compute rch(u) in $O(n + m)$ time.

3. Given G and u, compute all v that can reach u, that is all v such that $u \in$ rch(v).
4. Find the strongly connected component containing node u, that is $SCC(u)$.
5. Is G strongly connected (a single strong component)?
6. Compute all strongly connected components of G.

Directed graph connectivity problems

1. Given G and nodes u and v, can u reach v?
2. Given G and u, compute rch(u).

Use Explore(G, u) to compute rch(u) in $O(n + m)$ time.

3. Given G and u, compute all v that can reach u, that is all v such that $u \in$ rch(v).
4. Find the strongly connected component containing node u, that is $SCC(u)$.
5. Is G strongly connected (a single strong component)?

Uses G^{rev}

6. Compute all strongly connected components of G.

Algorithms via Basic Search - 1, 2

- Given G and nodes u and v, can u reach v?
- Given G and u, compute rch(u).

} already discussed

Use Explore(G, u) to compute rch(u) in $O(n + m)$ time.

Algorithms via Basic Search - 3

- Given G and u, compute all v, that can reach u, that is all v such that $u \in \text{rch}(u)$.
  Naive: $O(n(n+m))$

  ↳ run Explore from every vertex

Algorithms via Basic Search - 3

- Given G and u, compute all v, that can reach u, that is all v such that $u \in \text{rch}(v)$.
  Naive: $O(n(n+m))$

Definition (Reverse graph):

Given $G = (V, E)$, G^{rev} is the graph with edge directions reversed $G^{rev} = (V, E')$ where $E' = \{(y, x) \,|\, (x, y) \in E\}$

Algorithms via Basic Search - 3

- Given G and u, compute all v, that can reach u, that is all v such that $u \in \text{rch}(u)$. Naive: $O(n(n+m))$

Definition (Reverse graph):

Given $G = (V, E)$, G^{rev} is the graph with edge directions reversed $G^{rev} = (V, E')$ where $E' = \{(y, x) \mid (x, y) \in E\}$

Compute $\text{rch}(u)$ in G^{rev}.

Algorithms via Basic Search - 3

- Given G and u, compute all v, that can reach u, that is all v such that $u \in \text{rch}(u)$. Naive: $O(n(n+m))$

Definition (Reverse graph):

Given $G = (V, E)$, G^{rev} is the graph with edge directions reversed $G^{rev} = (V, E')$ where $E' = \{(y, x) \mid (x, y) \in E\}$

Compute rch(u) in G^{rev}. → *will be solution to all v that can reach u on original G.*

Running time: $O(n+m)$ to obtain G^{rev} from G and $O(n+m)$ time to compute rch(u)via Basic Search.

Algorithms via Basic Search - 4

$SCC(G, u) = \{v \,|\, u$ is strongly connected to v$\}$

Algorithms via Basic Search - 4

$SCC(G, u) = \{v \mid u$ is strongly connected to v$\}$

Find the strongly connected component containing node u. That is, compute $SCC(G, u)$.

Algorithms via Basic Search - 4

$SCC(G, u) = \{v \mid u$ is strongly connected to v$\}$

Find the strongly connected component containing node u. That is, compute $SCC(G, u)$.

$SCC(G, u) = \text{rch}(G, u) \cap \text{rch}(G^{rev}, u)$ → we will only "prove by example"

Algorithms via Basic Search - 4

$SCC(G, u) = \{v \mid u$ is strongly connected to v$\}$

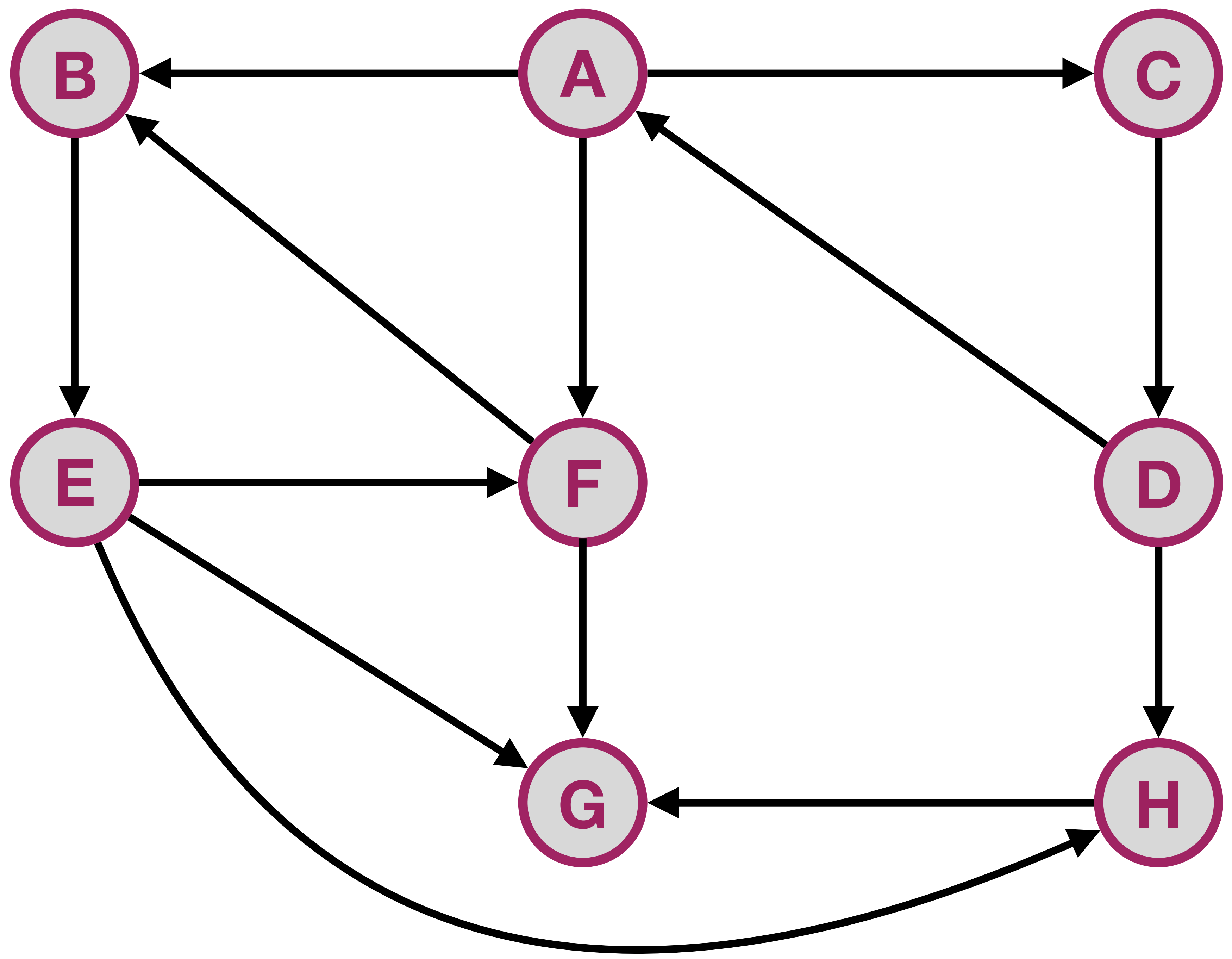
Find the strongly connected component containing node u. That is, compute $SCC(G, u)$.

$SCC(G, u) = \text{rch}(G, u) \cap \text{rch}(G^{rev}, u)$

Hence, $SCC(G, u)$ can be computed with Explore(G, u) and Explore(G^{rev}, u). Total $O(n + m)$ time

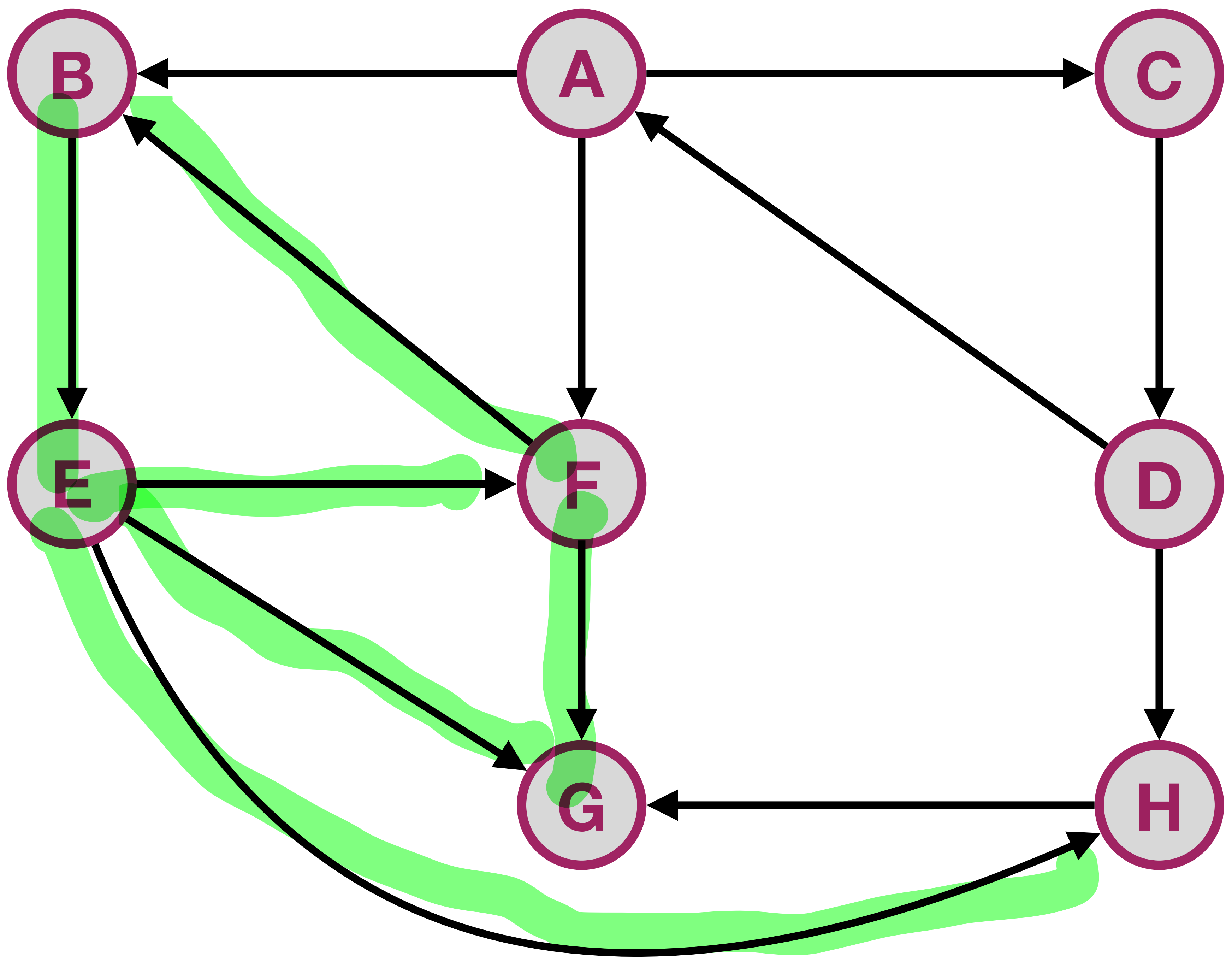
Algorithms via Basic Search - 4

Given a graph G, and a vertex F…



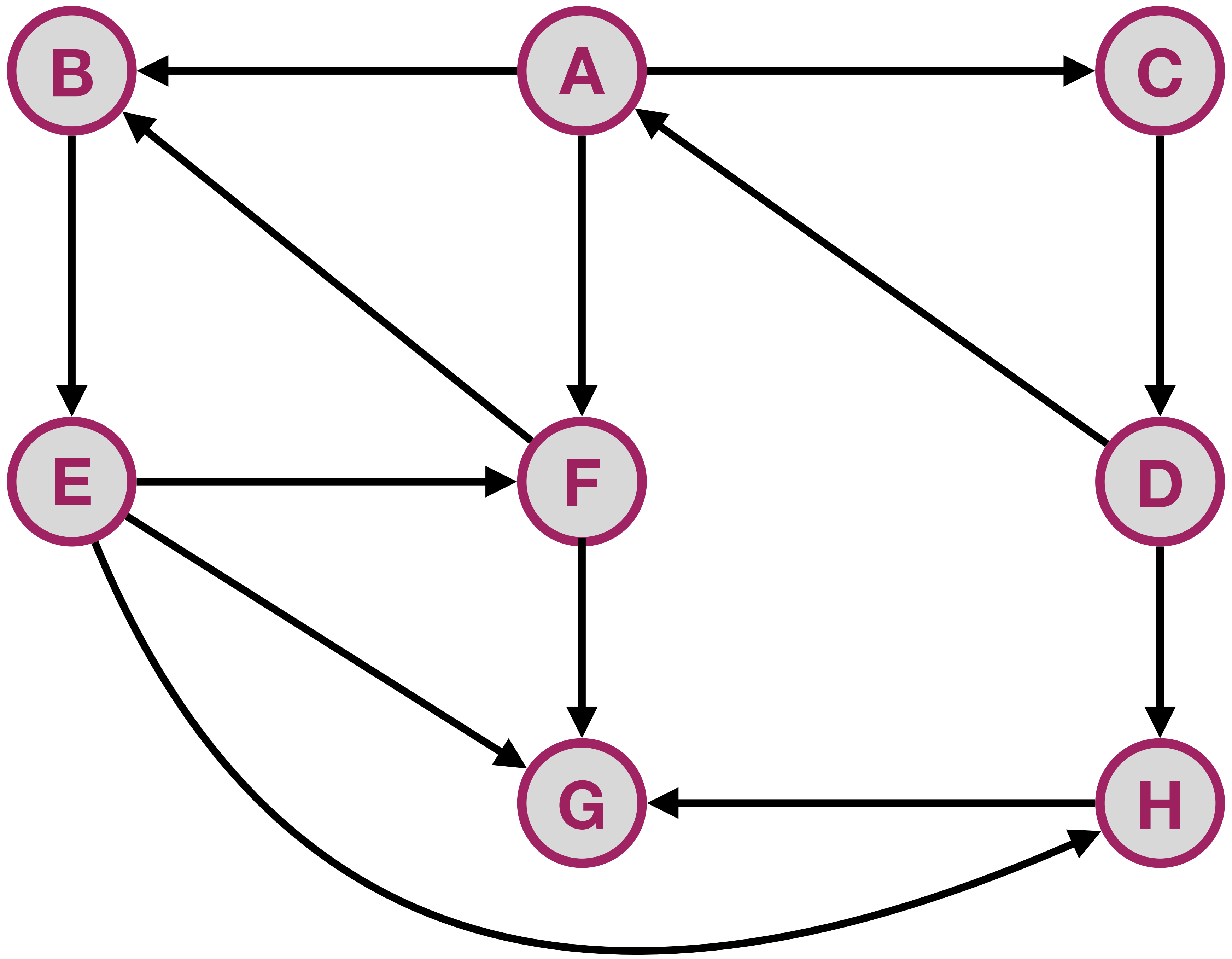
Graph G

… its reachable set rch(G, F)



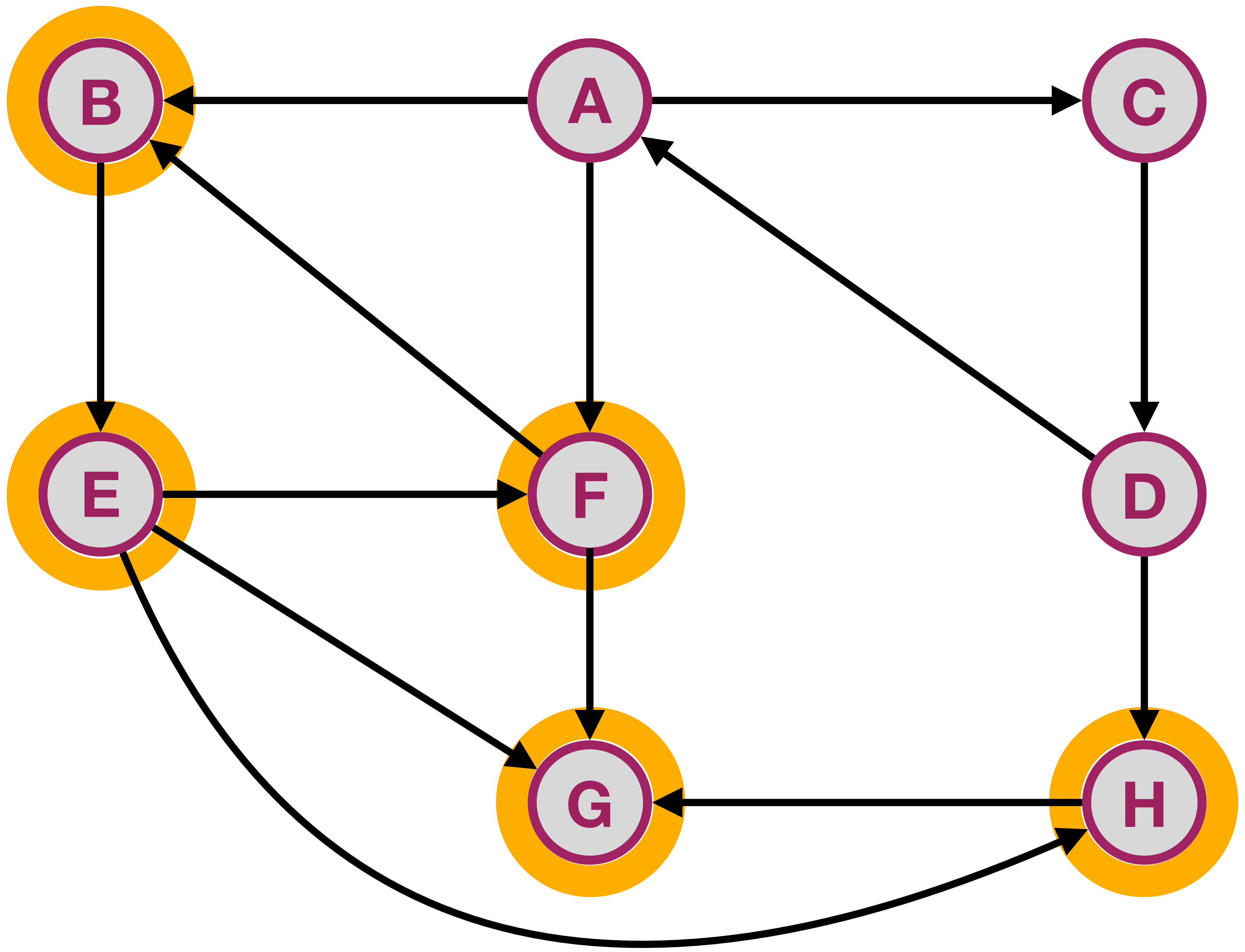
Algorithms via Basic Search - 4

Given a graph G, and a vertex F…



Graph G

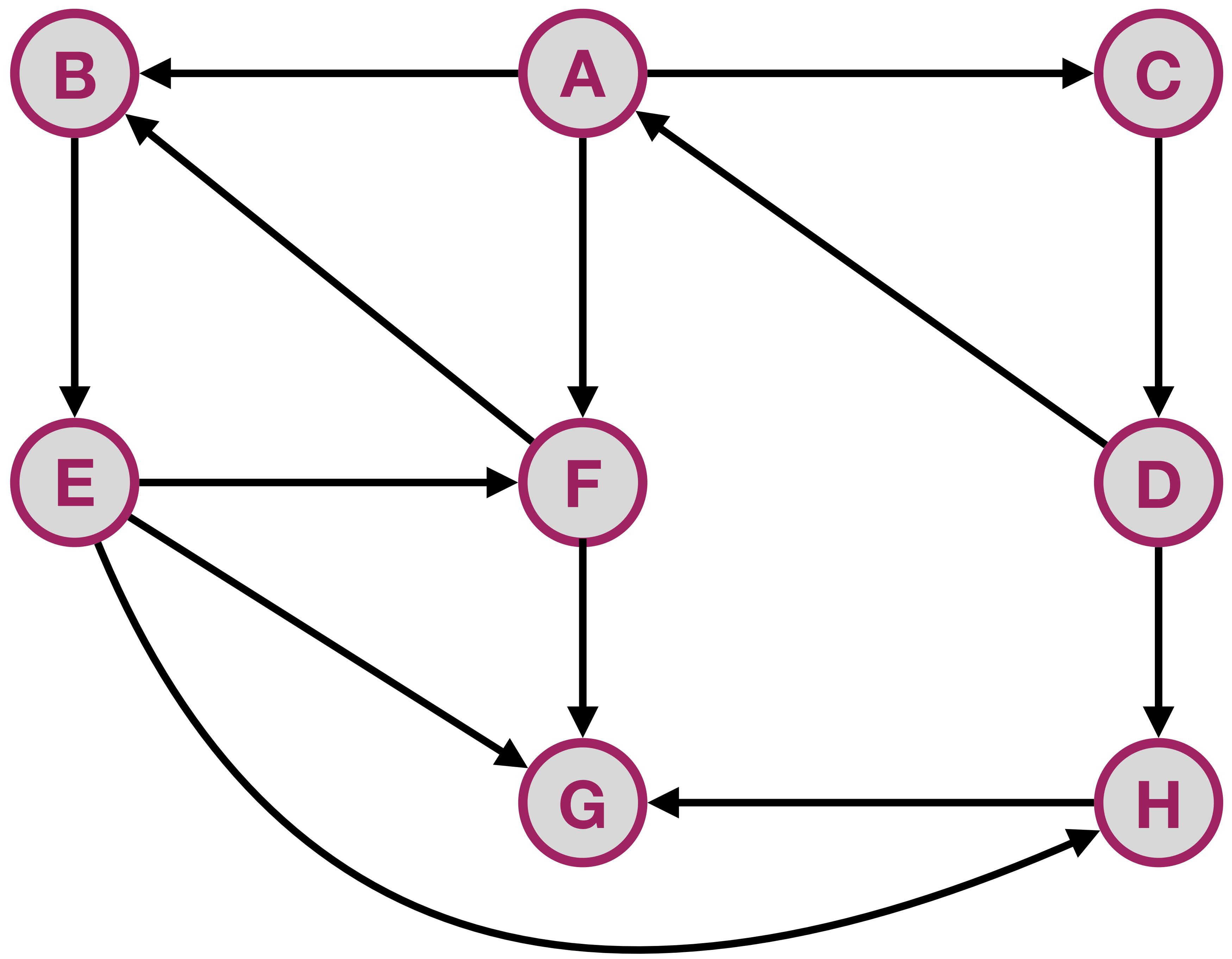
… its reachable set $\text{rch}(G, F)$



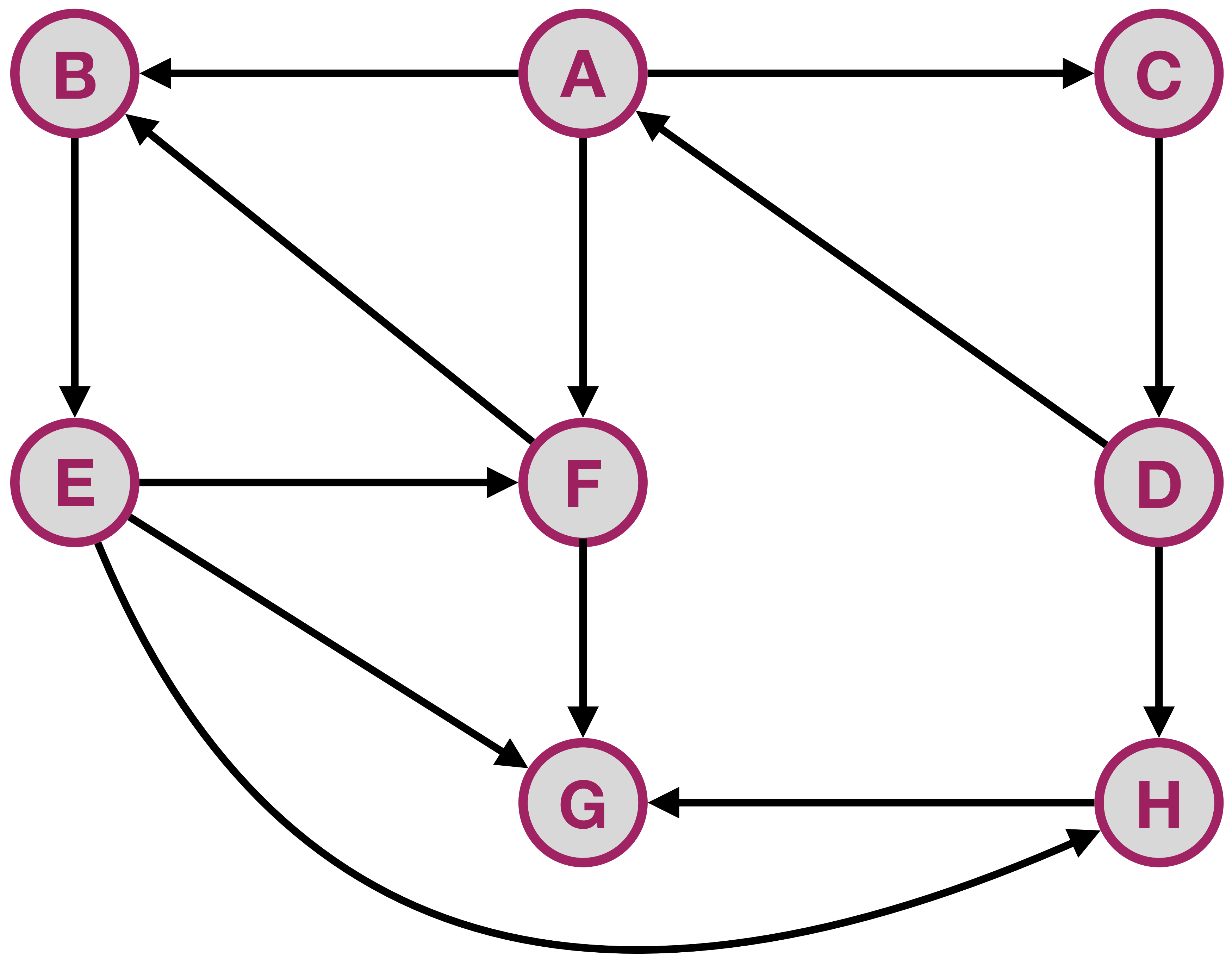
is set of vertices reachable from F .

Algorithms via Basic Search - 4

Given a graph G …

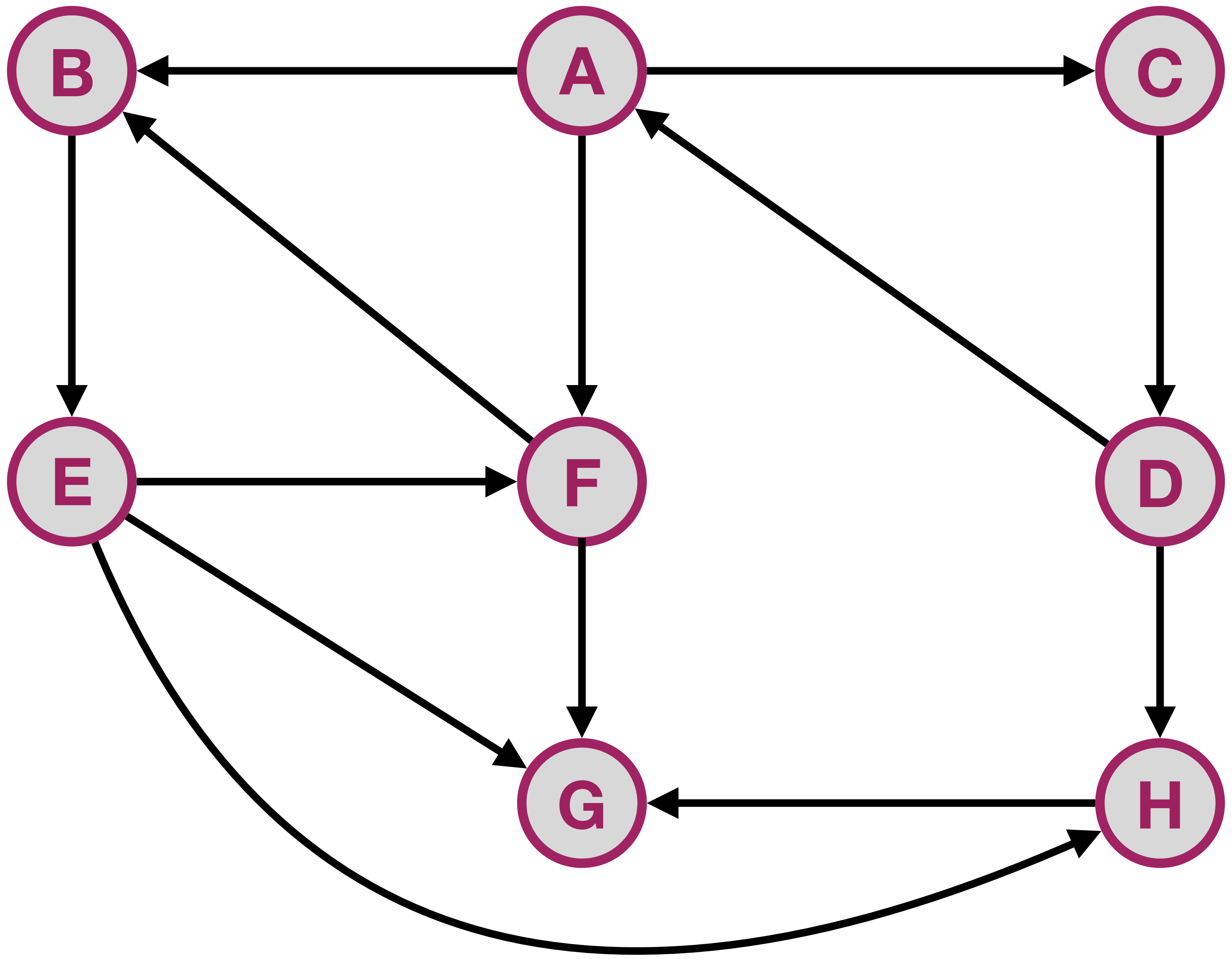


its reverse graph G^{rev} …

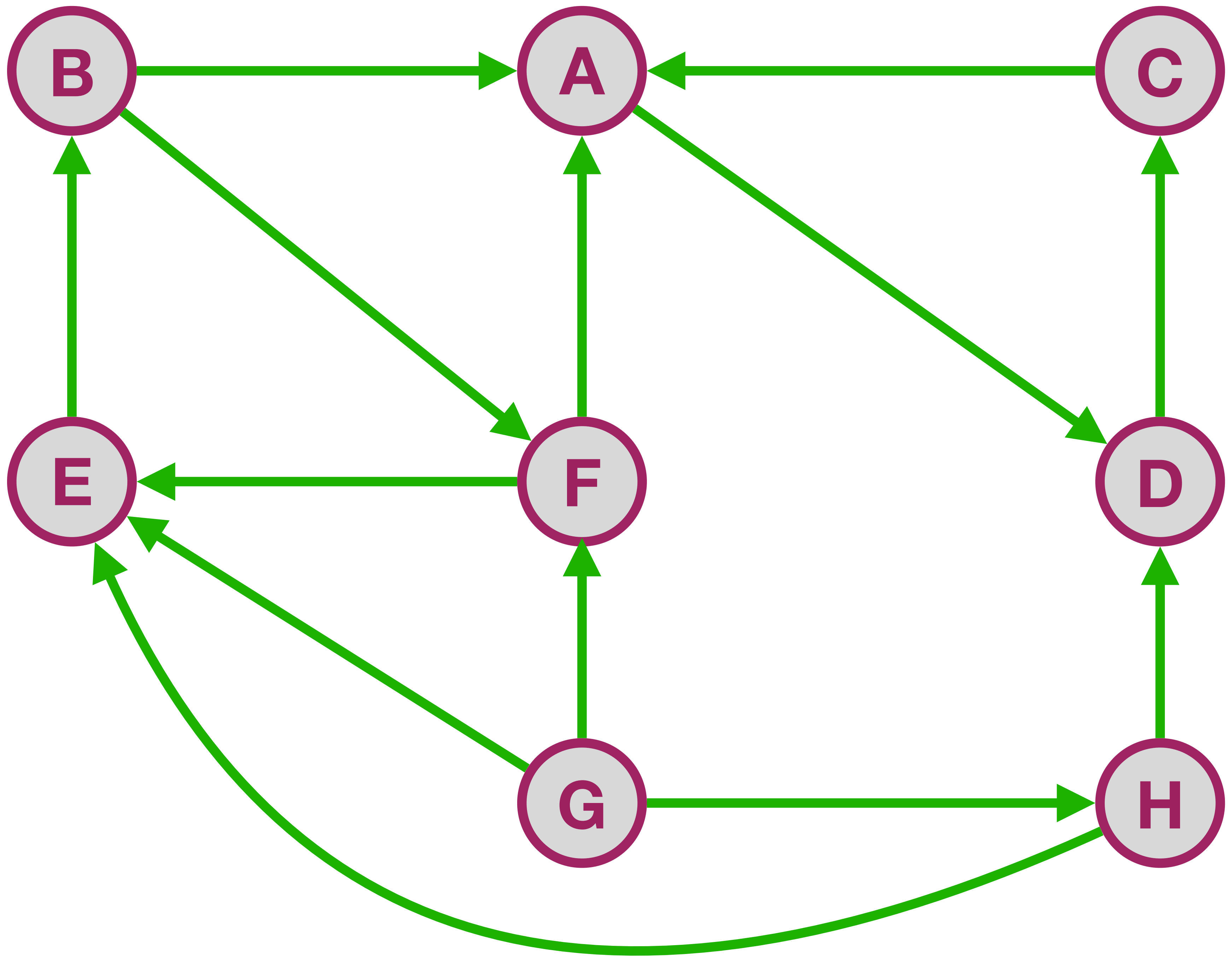


Algorithms via Basic Search - 4

Given a graph G …



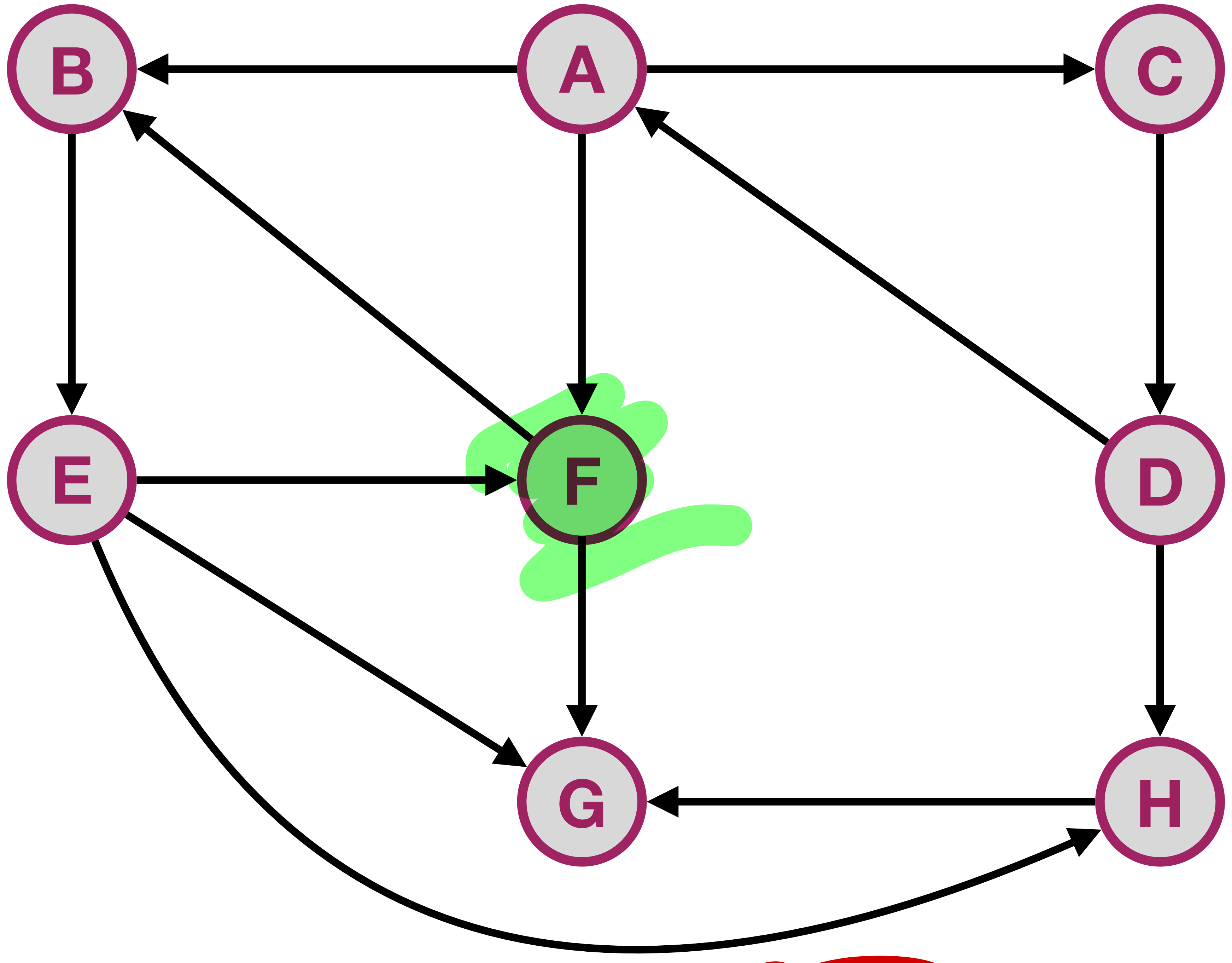
its reverse graph G^{rev} …



… has all edges reversed.

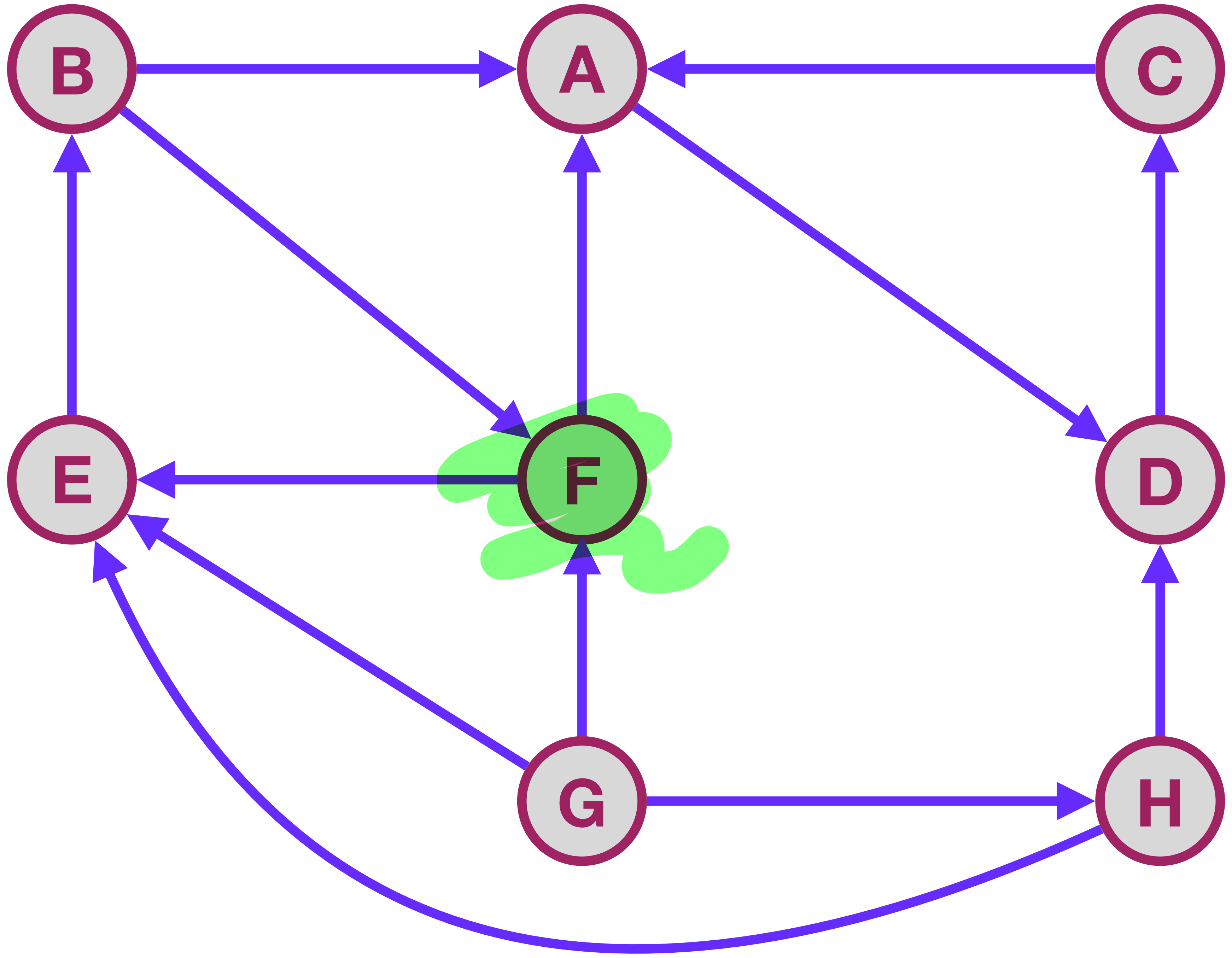
Algorithms via Basic Search - 4

Given a graph G, and a vertex F ...



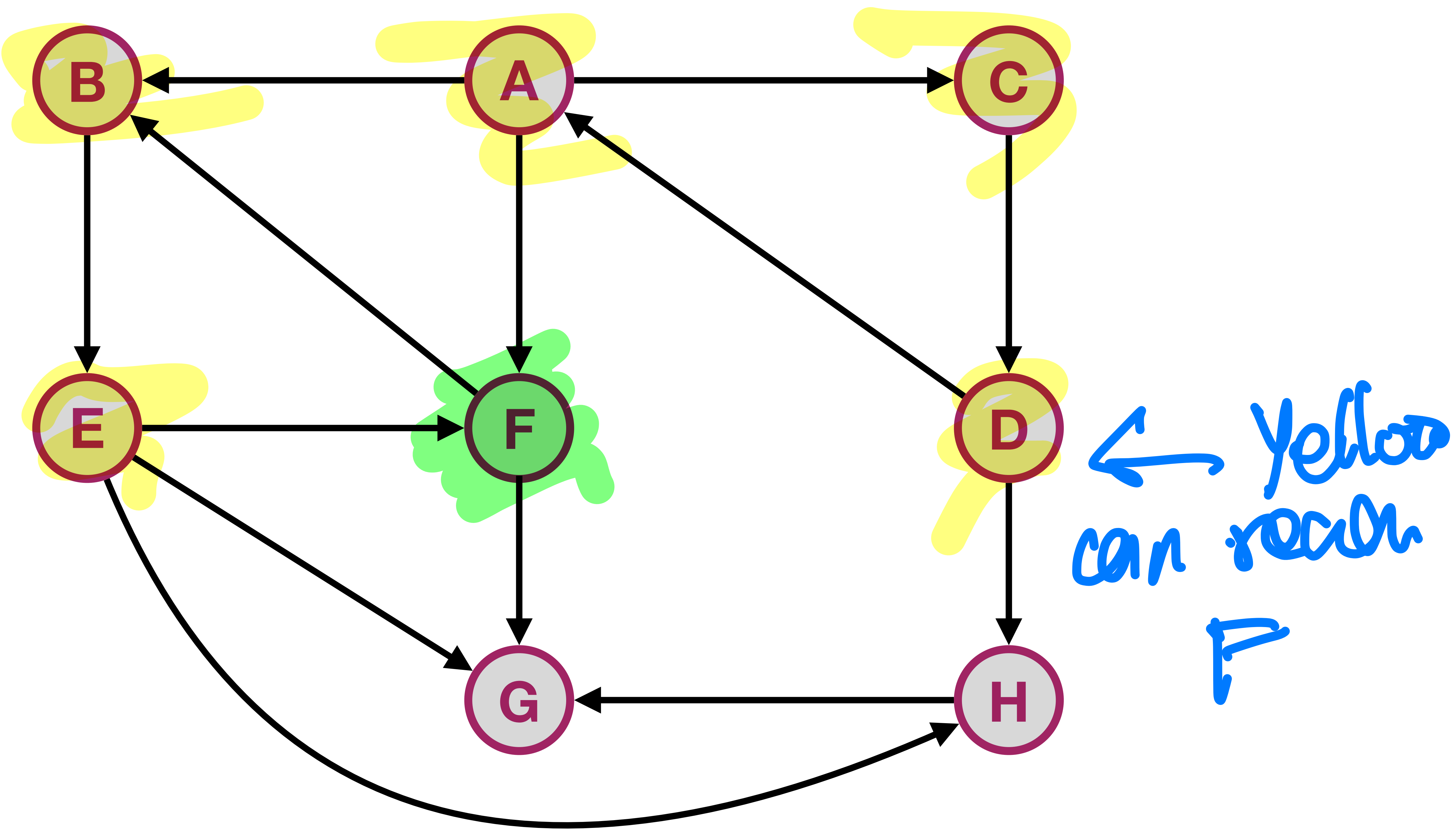
Graph G

.. the set of vertices that can reach it in G ...



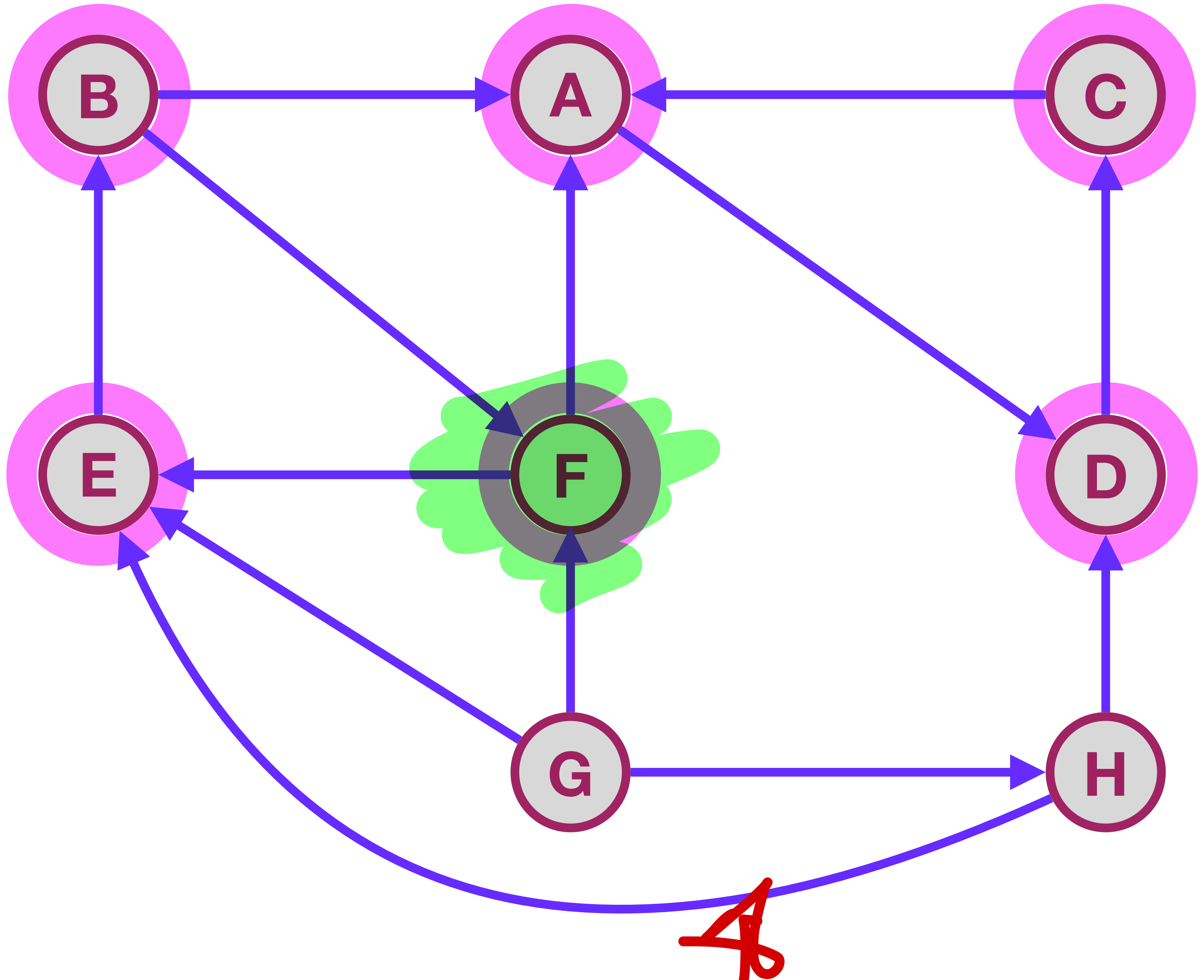
Algorithms via Basic Search - 4

Given a graph G, and a vertex F…



Graph G

.. the set of vertices that can reach it in G …



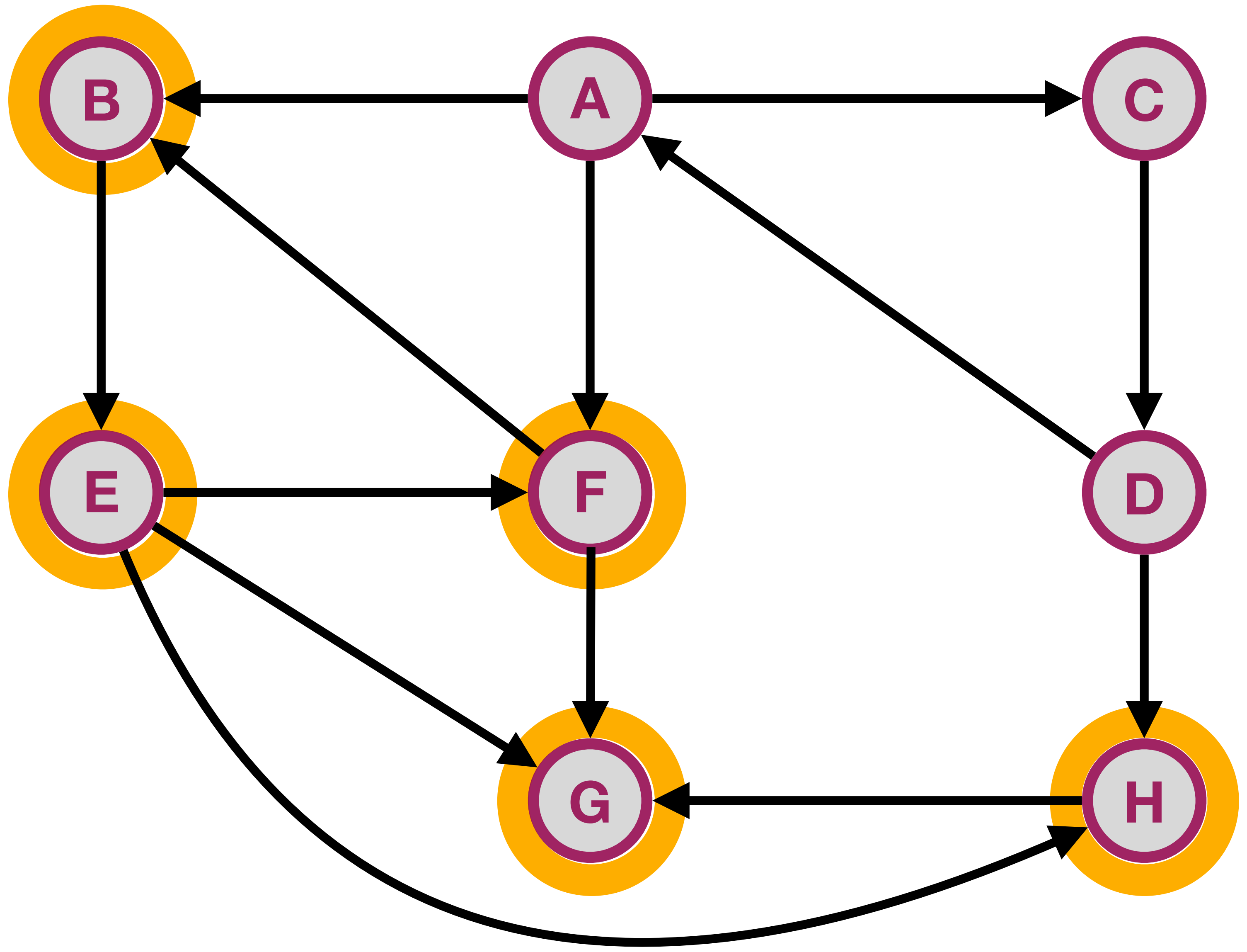
… is $\mathrm{rch}(G^{rev}, F)$

Algorithms via Basic Search - 4

i.e F's

Given a graph G, and a vertex F ~~and~~, its strongly connected component in G is …

$\text{rch}(G, F)$

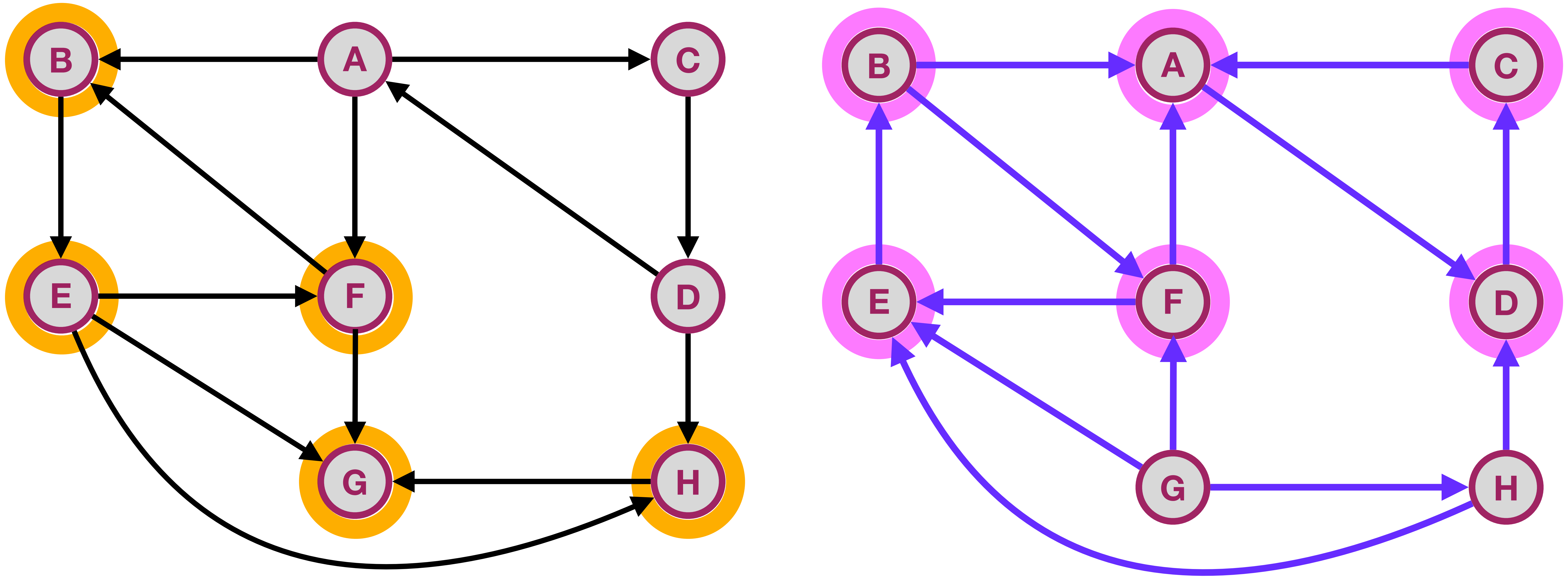


$\text{rch}(G, F)$

Algorithms via Basic Search - 4

Given a graph G, and a vertex F and its strongly connected component in G is …

$\text{rch}(G, F) \cap \text{rch}(G^{rev}, F)$



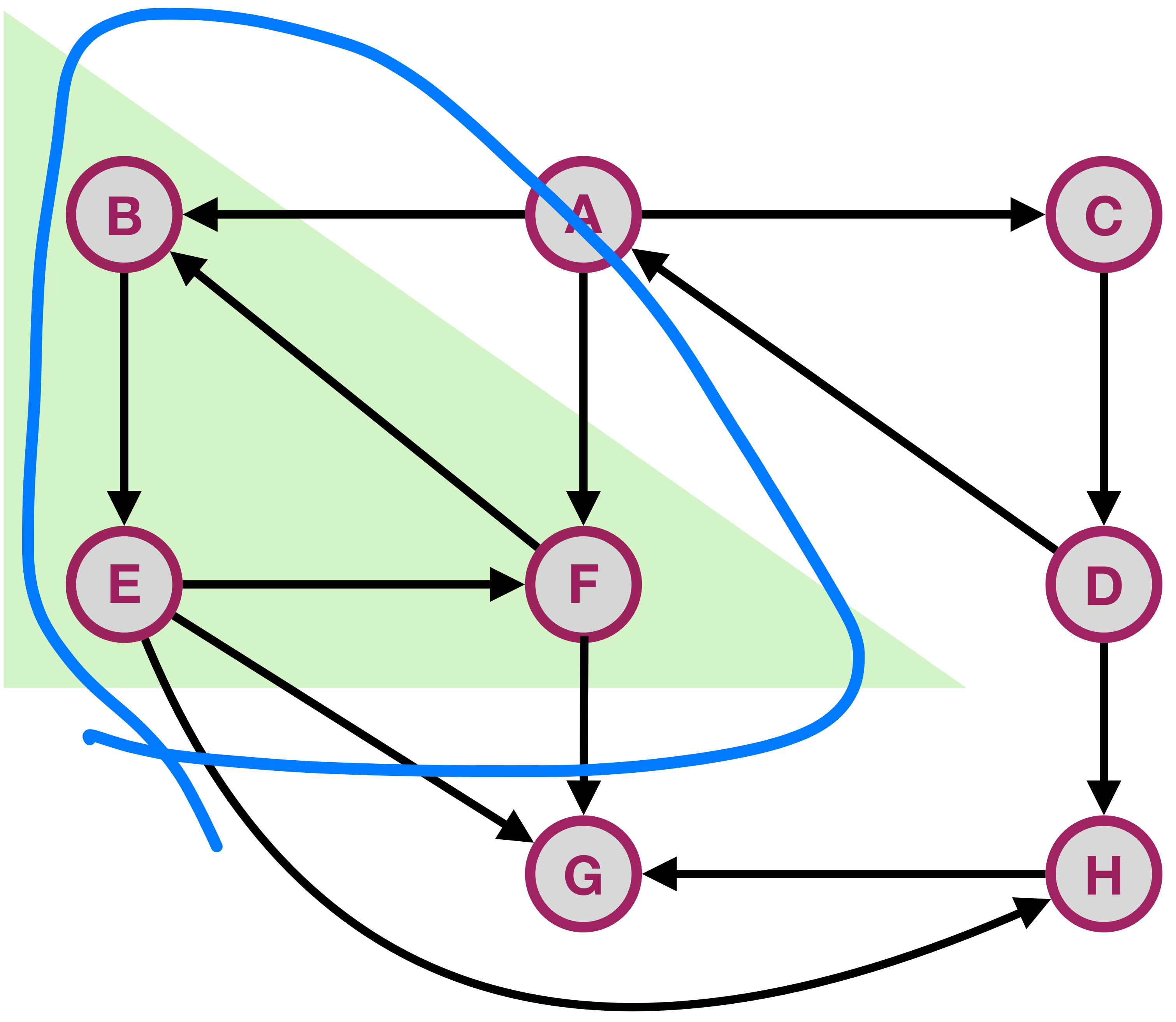
$\text{rch}(G, F)$

$\text{rch}(G^{rev}, F)$

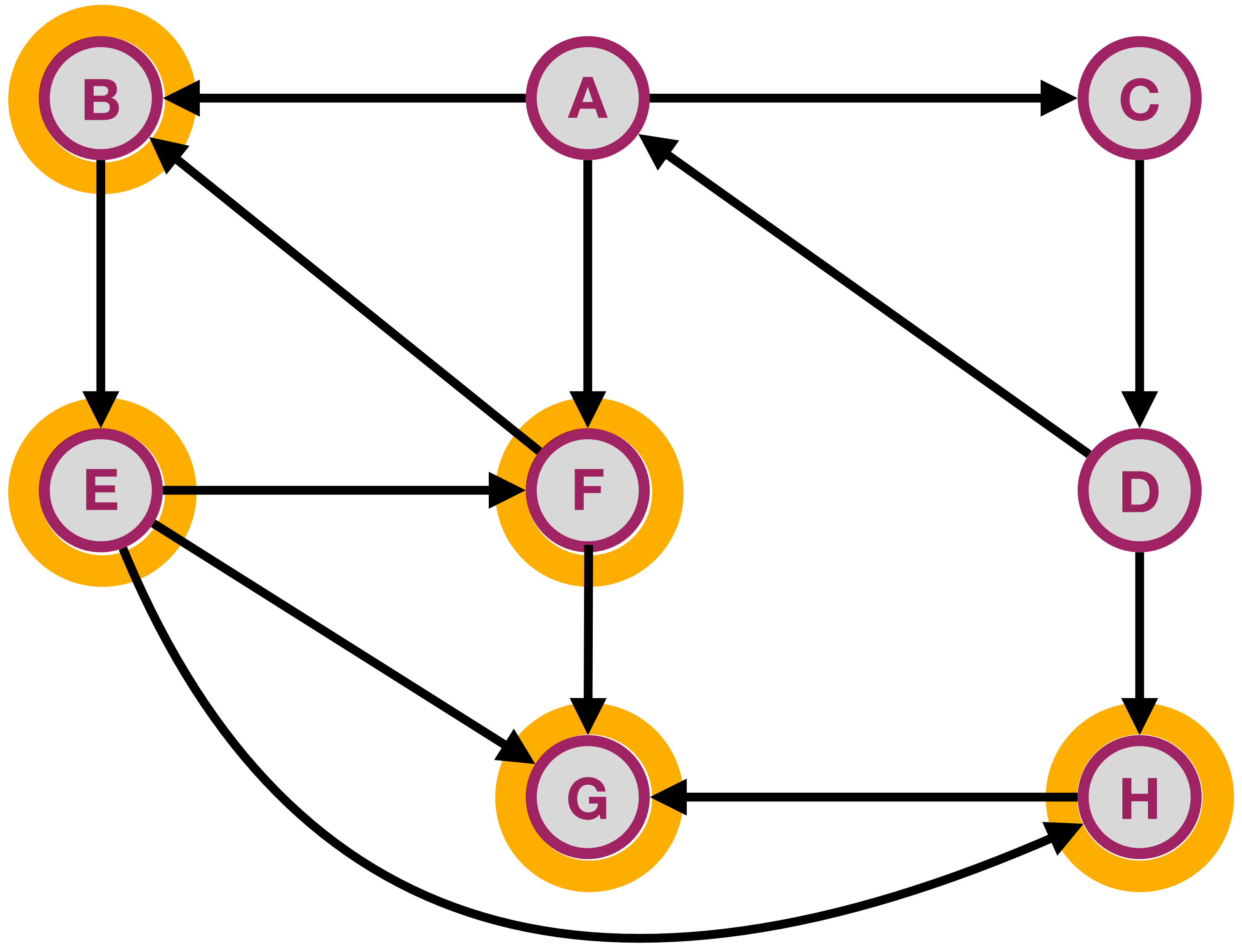
Algorithms via Basic Search - 4

Given a graph G, and a vertex F and its strongly connected component in G is …

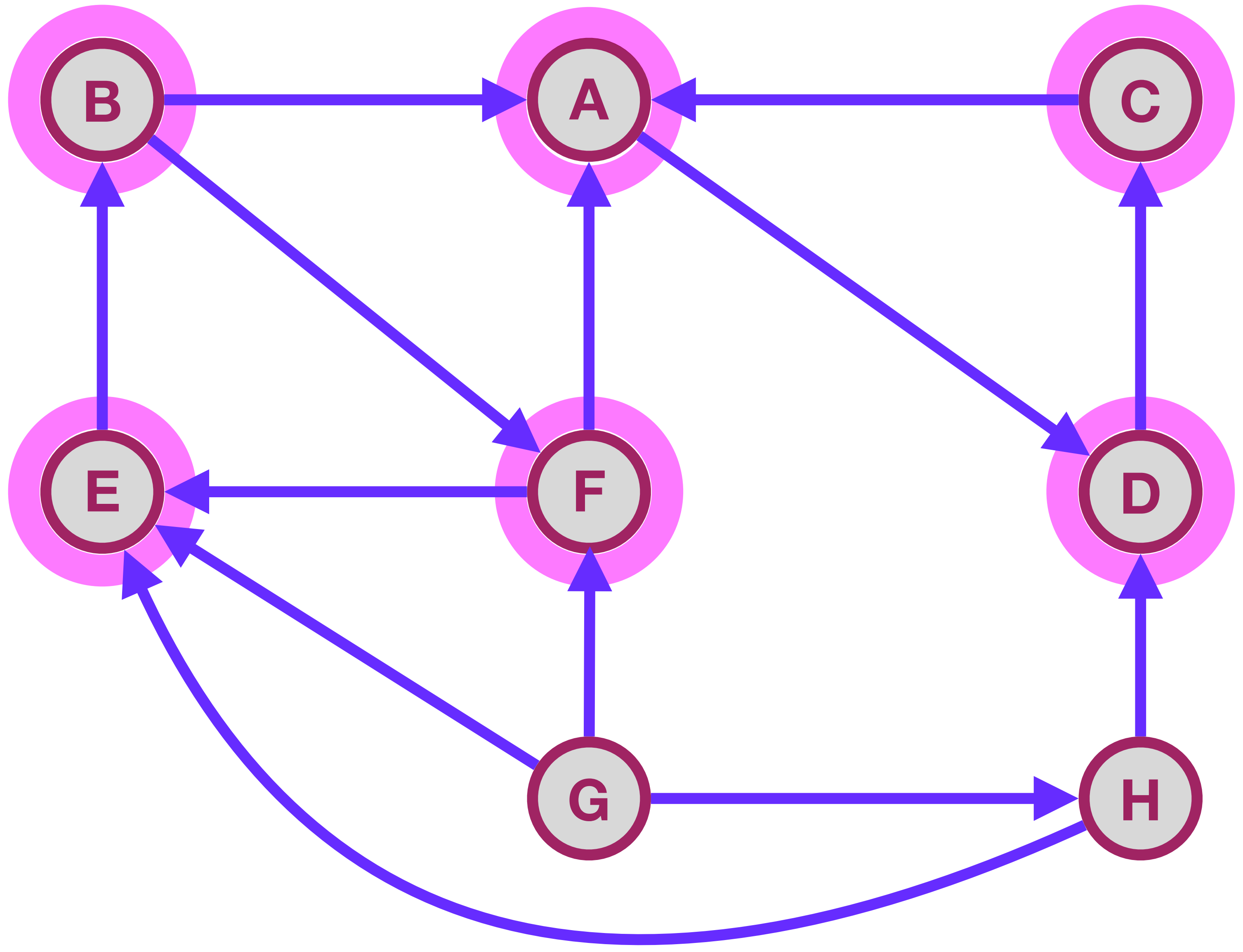
$$SCC(G, F) = \mathrm{rch}(G, F) \cap \mathrm{rch}(G^{rev}, F)$$



Graph G



rch(G, F)



rch(G^{rev}, F)

Algorithms via Basic Search - 5

- Is G strongly connected?
- Pick arbitrary vertex u.
- Check if $SCC(G, u) = V$.