Announcements

- Homework 0 Due today
- Homework 1 online due next Friday
- Discussion section notes/podcast online
- Remember FinAid survey
- No class on Monday

Last Time

- Graphs
- explore/DFS
- Connected components

Graph Definition

Definition: A *graph* G = (V,E) consists of two things:

- A collection V of *vertices*, or objects to be connected.
- A collection E of *edges*, each of which connects a pair of vertices.

Runtime of DFS

```
explore(v)
  v.visited ← true
  For each edge (v,w)
     If not w.visited
          explore(w)
```

Run once per vertex

O(|V|) total

Run once per neighboring vertex

O(|E|) total

```
DFS(G)
  Mark all v ∈ G as unvisited
  For v ∈ G
    If not v.visited, explore(v)
```

O(|V|)

Final runtime: O(|V|+|E|)

Runtime of DFS

```
explore(v)
  v.visited ← true
  For each edge (v,w)
     If not w.visited
        explore(w)
```

Run once per vertex

O(|V|) total

Run once per neighboring vertex

O(|E|) total

Only Counting work from Non-recursive calls!

```
DFS(G)
  Mark all v ∈ G as unvisited
  For v ∈ G
    If not v.visited, explore(v)
```

O(|V|)

Final runtime: O(|V|+|E|)

Connected Components

Theorem: The vertices of a graph G can be partitioned into *connected components* so that v is reachable from w if and only if they are in the same connected component.

Today

- Computing connected components
- Pre- / Post- orders
- Directed graphs
- Topological orderings

Problem: Computing Connected Components

<u>Problem:</u> Given a graph G, compute its connected components.

Problem: Computing Connected Components

<u>Problem:</u> Given a graph G, compute its connected components.

<u>Easy:</u> For each v, run `explore(v)` to find vertices reachable from it. Group together into components.

Problem: Computing Connected Components

Problem: Given a graph G, compute its connected components.

Easy: For each v, run `explore(v)` to find vertices reachable from it. Group together into components.

Runtime: $O(|V|(|V|+|E|))$.

Problem: Computing Connected Components

Problem: Given a graph G, compute its connected components.

Easy: For each v, run `explore(v)` to find vertices reachable from it. Group together into components.

Runtime: $O(|V|(|V|+|E|))$.

Better: Run `explore(v)` to find the component of v. Repeat on unclassified vertices.

DFS lets us do this!

```
ConnectedComponents(G)

  For v ∈ G
    v.visited ← false
  For v ∈ G
    If not v.visited

      explore(v)
```

```
explore(v)
  v.visited ← true

  For each edge (v,w)
    If not w.visited
      explore(w)
```

DFS lets us do this!

```
ConnectedComponents(G)
  CCNum ← 0
  For v ∈ G
    v.visited ← false
  For v ∈ G
    If not v.visited
      CCNum++
      explore(v)
```

```
explore(v)
  v.visited ← true

  For each edge (v,w)
    If not w.visited
      explore(w)
```

DFS lets us do this!

```
ConnectedComponents(G)
  CCNum ← 0
  For v ∈ G
    v.visited ← false
  For v ∈ G
    If not v.visited
      CCNum++
      explore(v)
```

```
explore(v)
  v.visited ← true
  v.CC ← CCNum
  For each edge (v,w)
    If not w.visited
      explore(w)
```

DFS lets us do this!

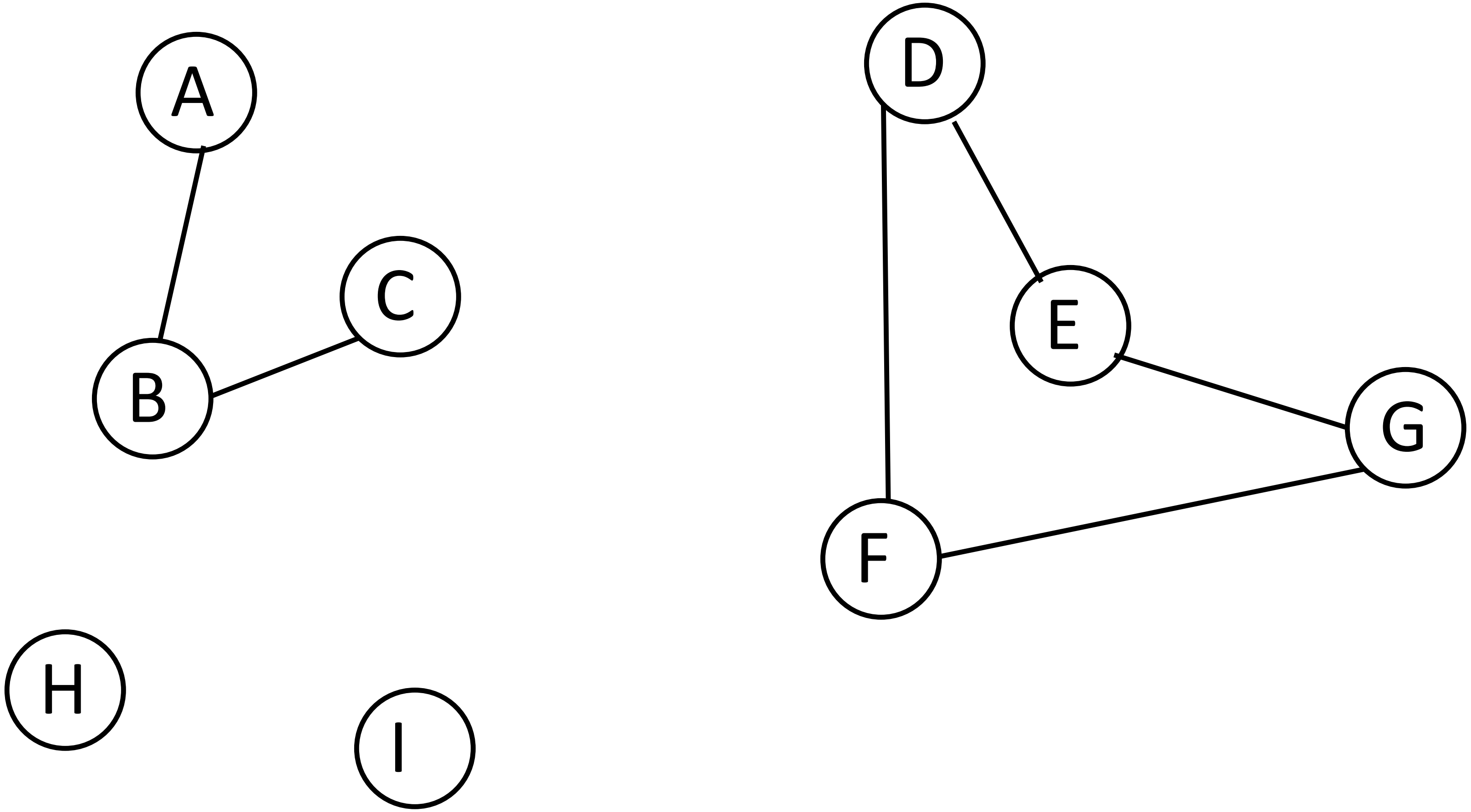
```
ConnectedComponents(G)
  CCNum ← 0
  For v ∈ G
    v.visited ← false
  For v ∈ G
    If not v.visited
      CCNum++
      explore(v)
```

```
explore(v)
  v.visited ← true
  v.CC ← CCNum
  For each edge (v,w)
    If not w.visited
      explore(w)
```

Runtime $O(|V|+|E|)$.

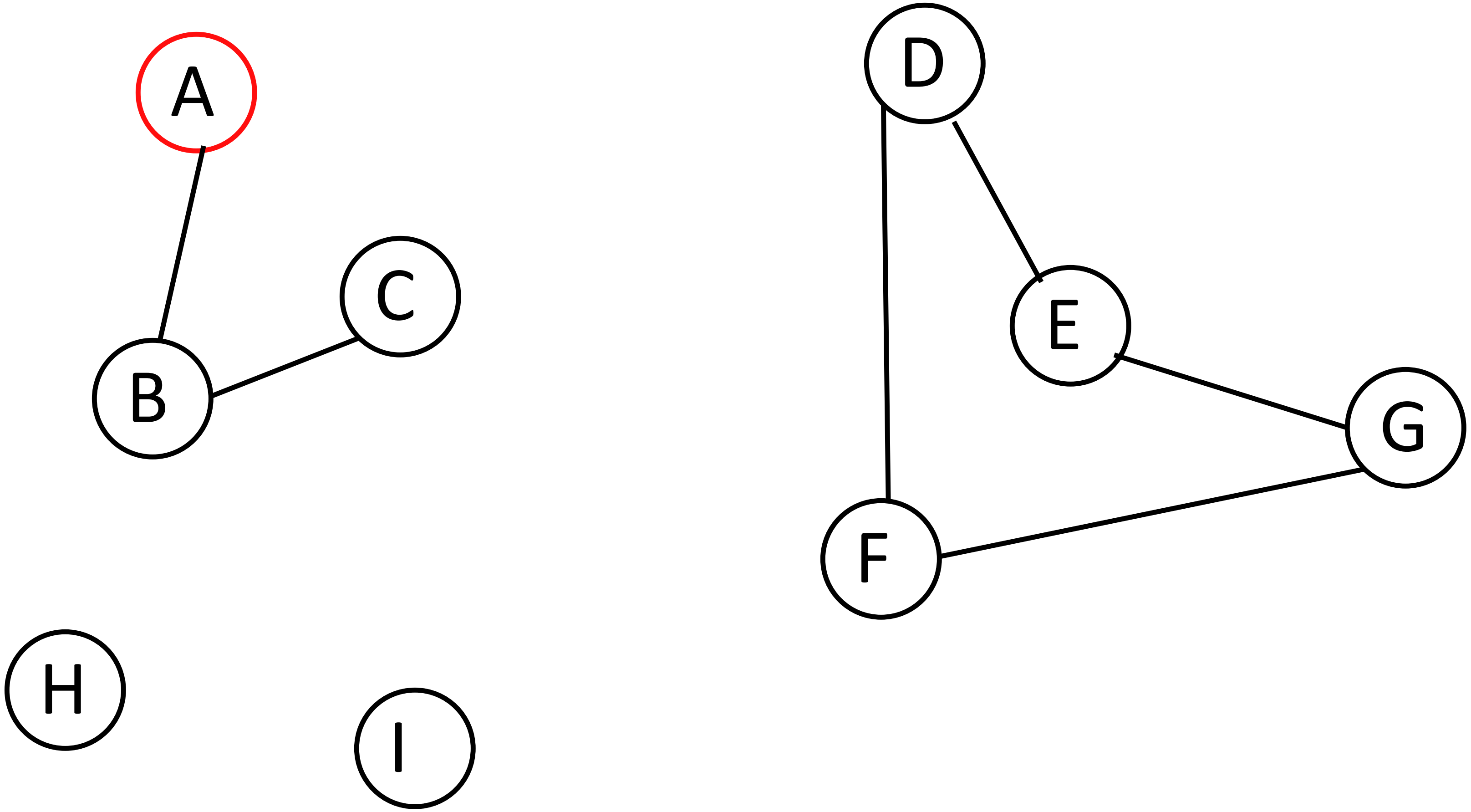
Example

`CCNum:`



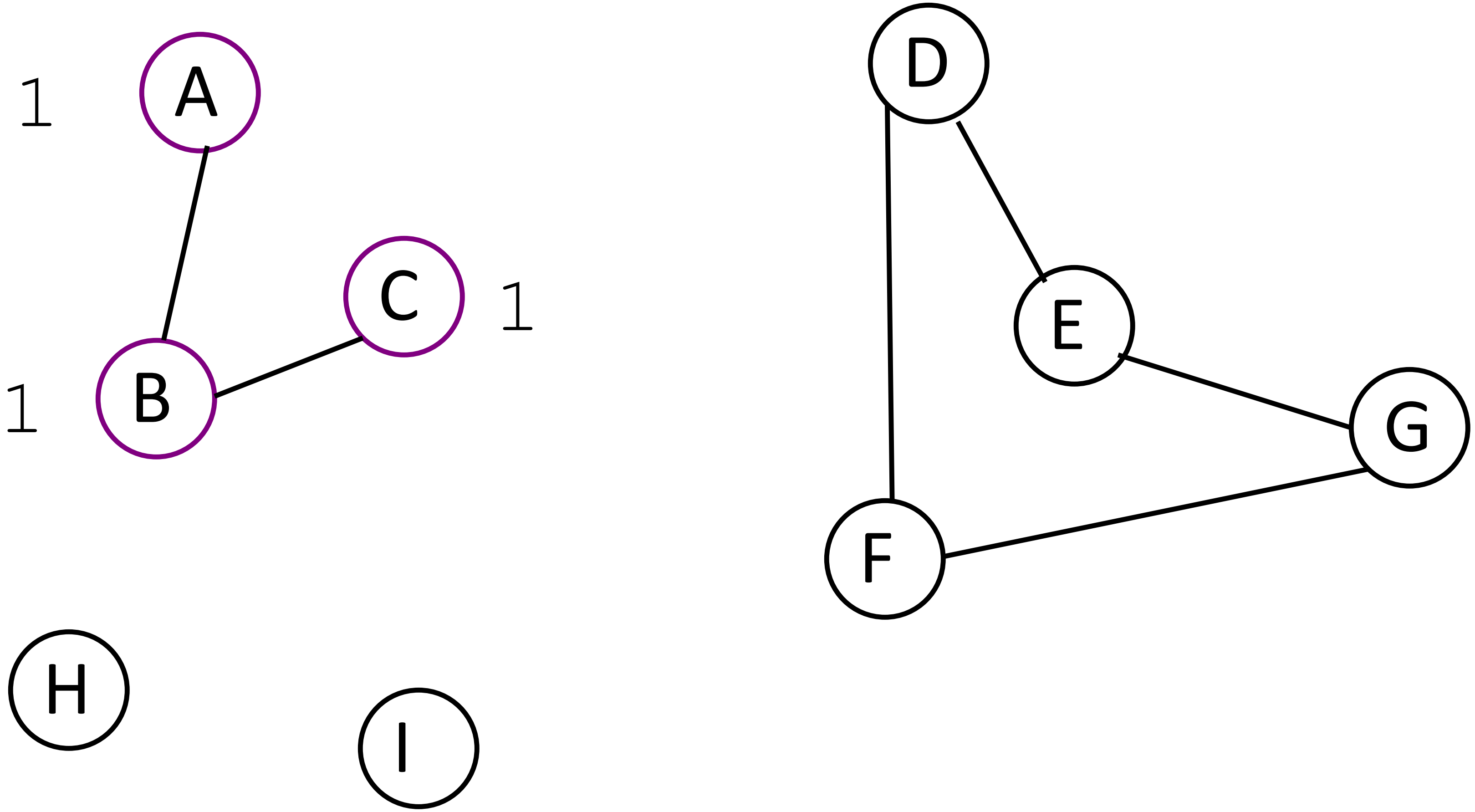
Example

CCNum: 1



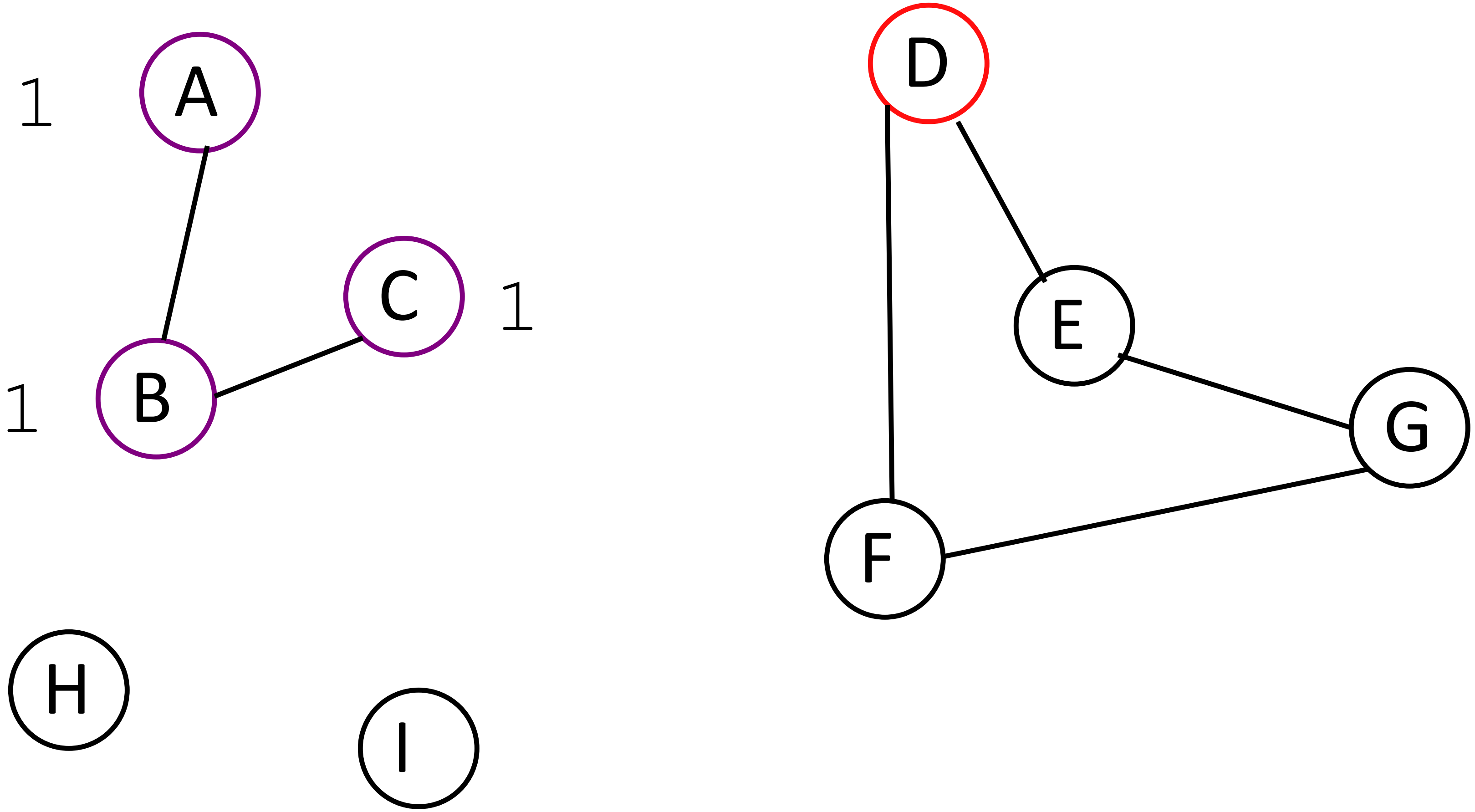
Example

CCNum: 1



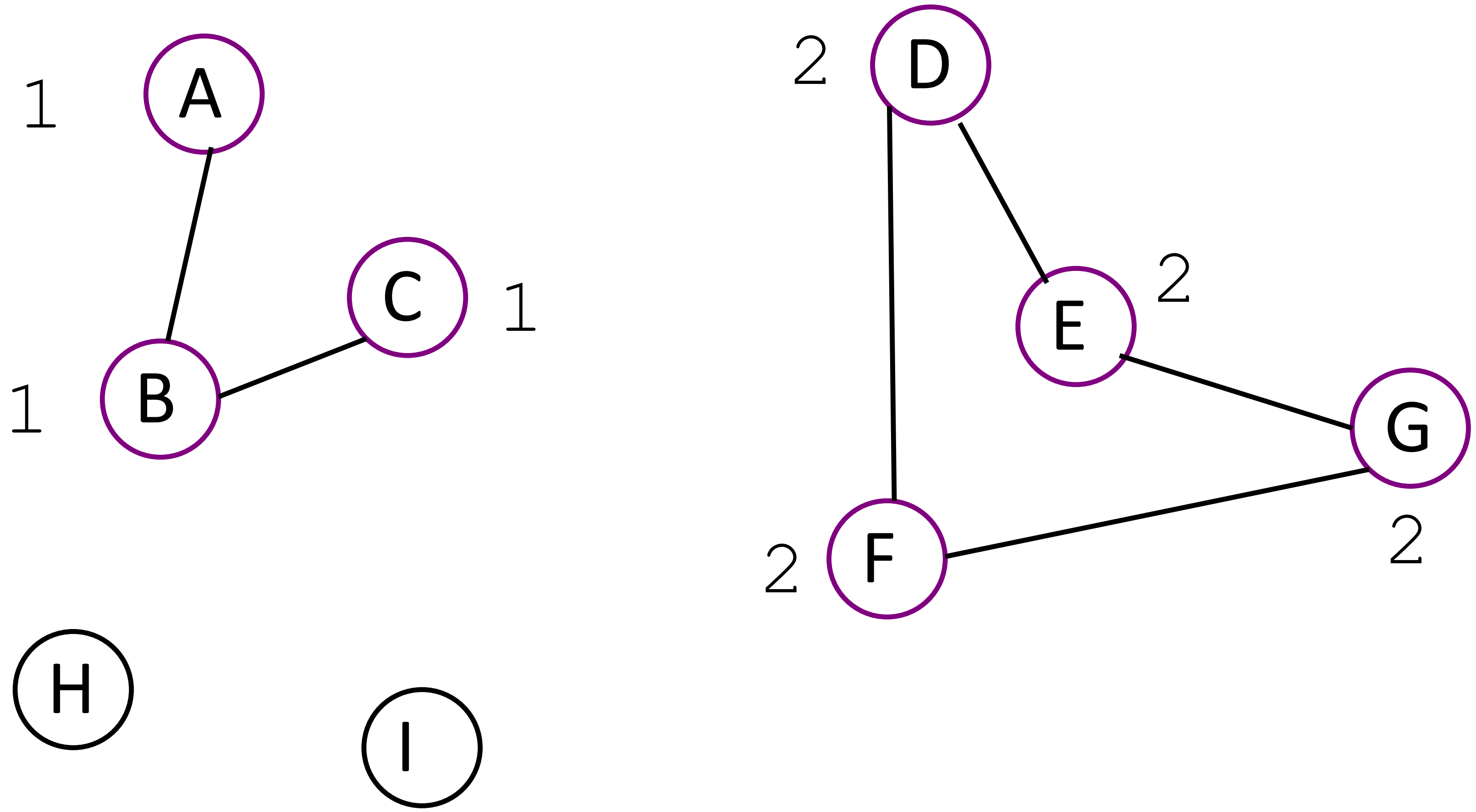
Example

CCNum: 2



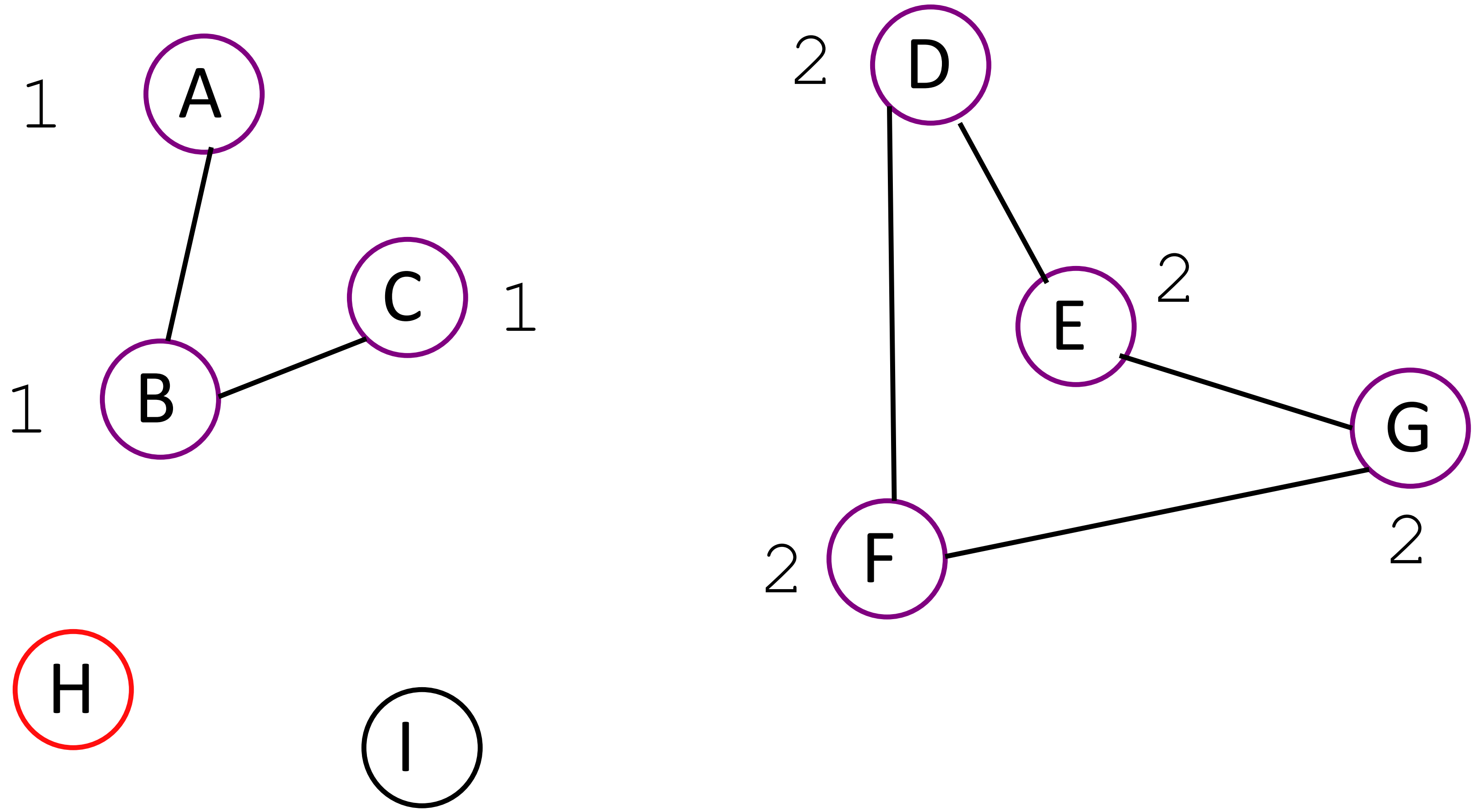
Example

CCNum: 2



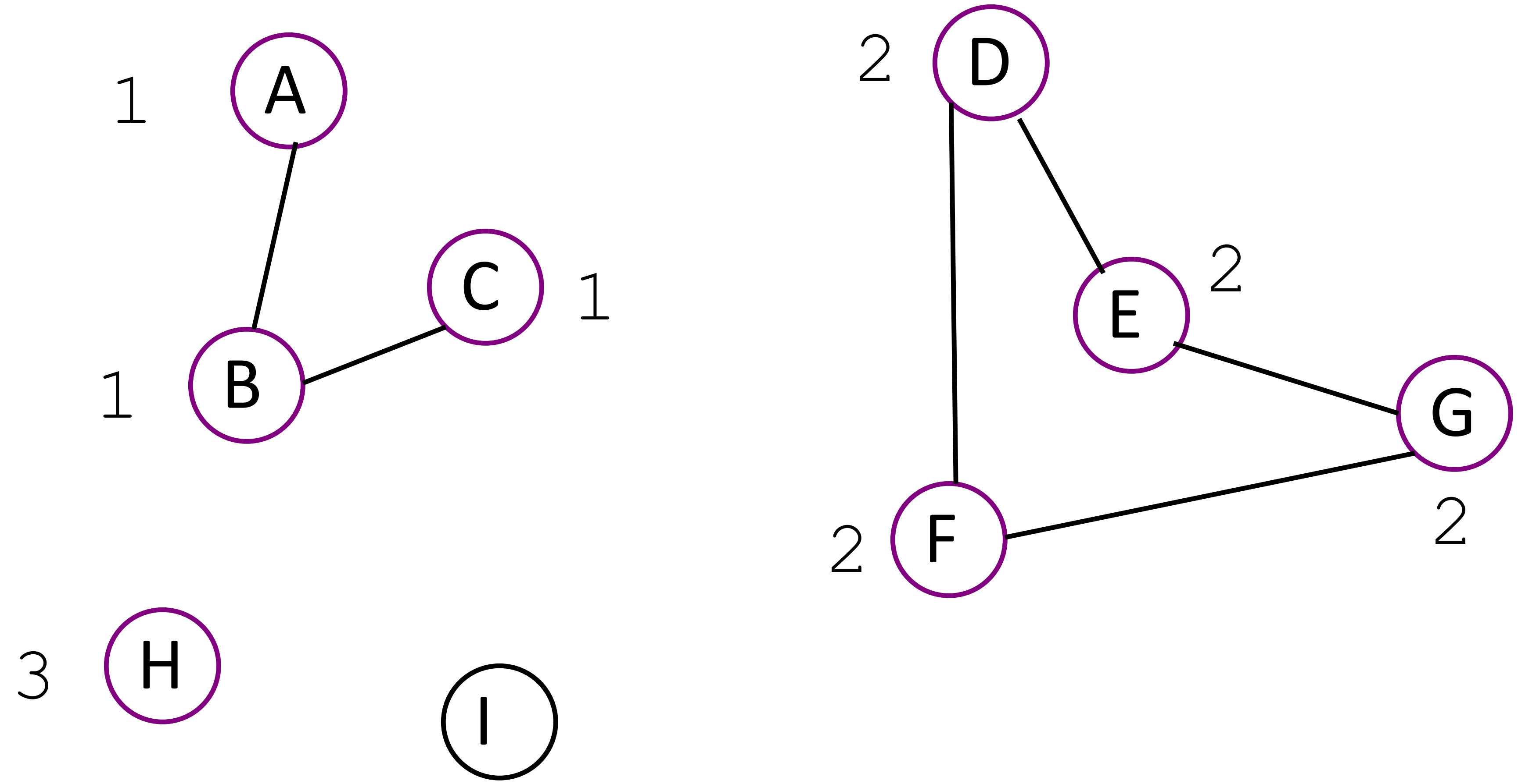
Example

CCNum: 3



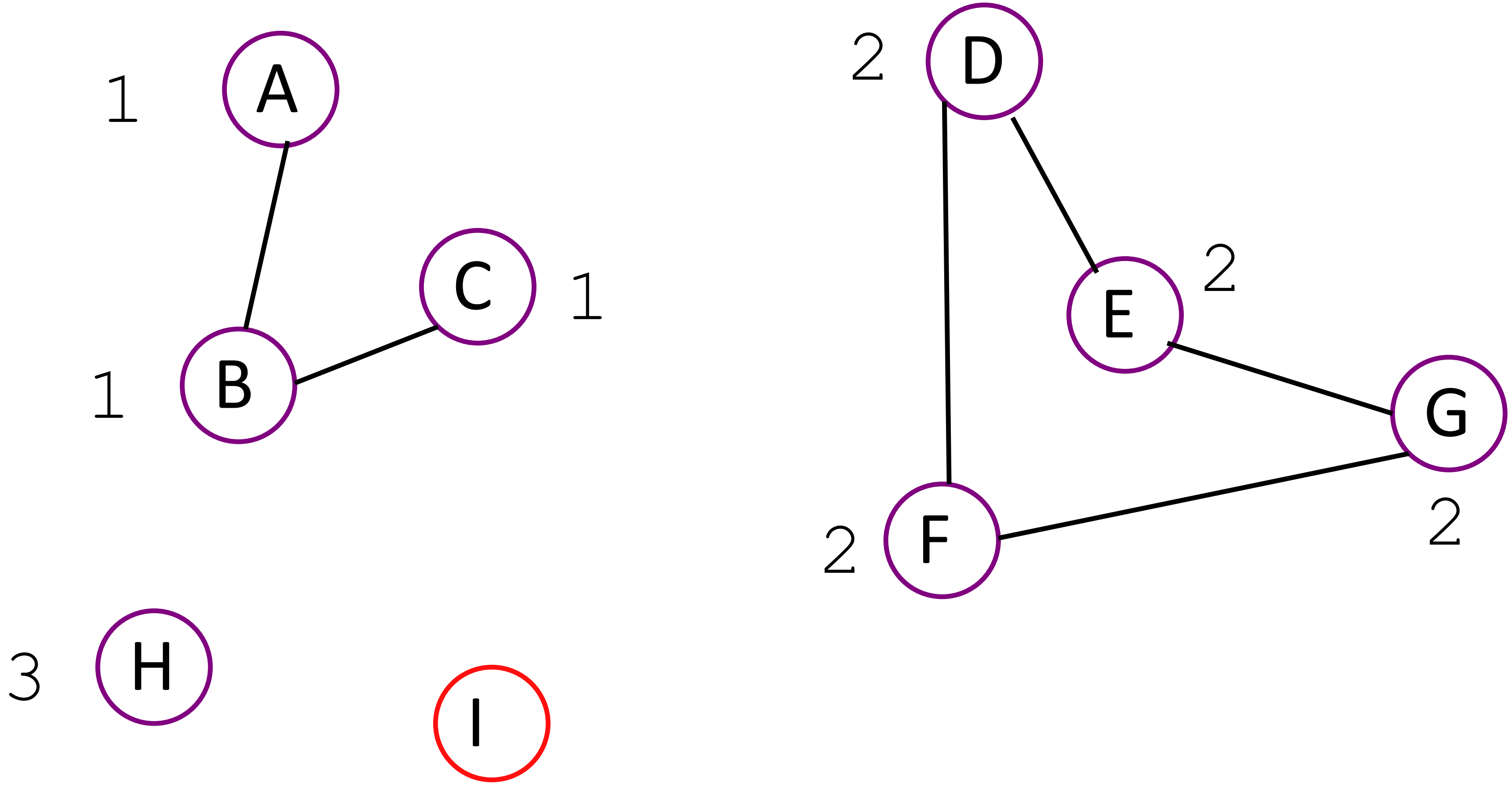
Example

CCNum: 3



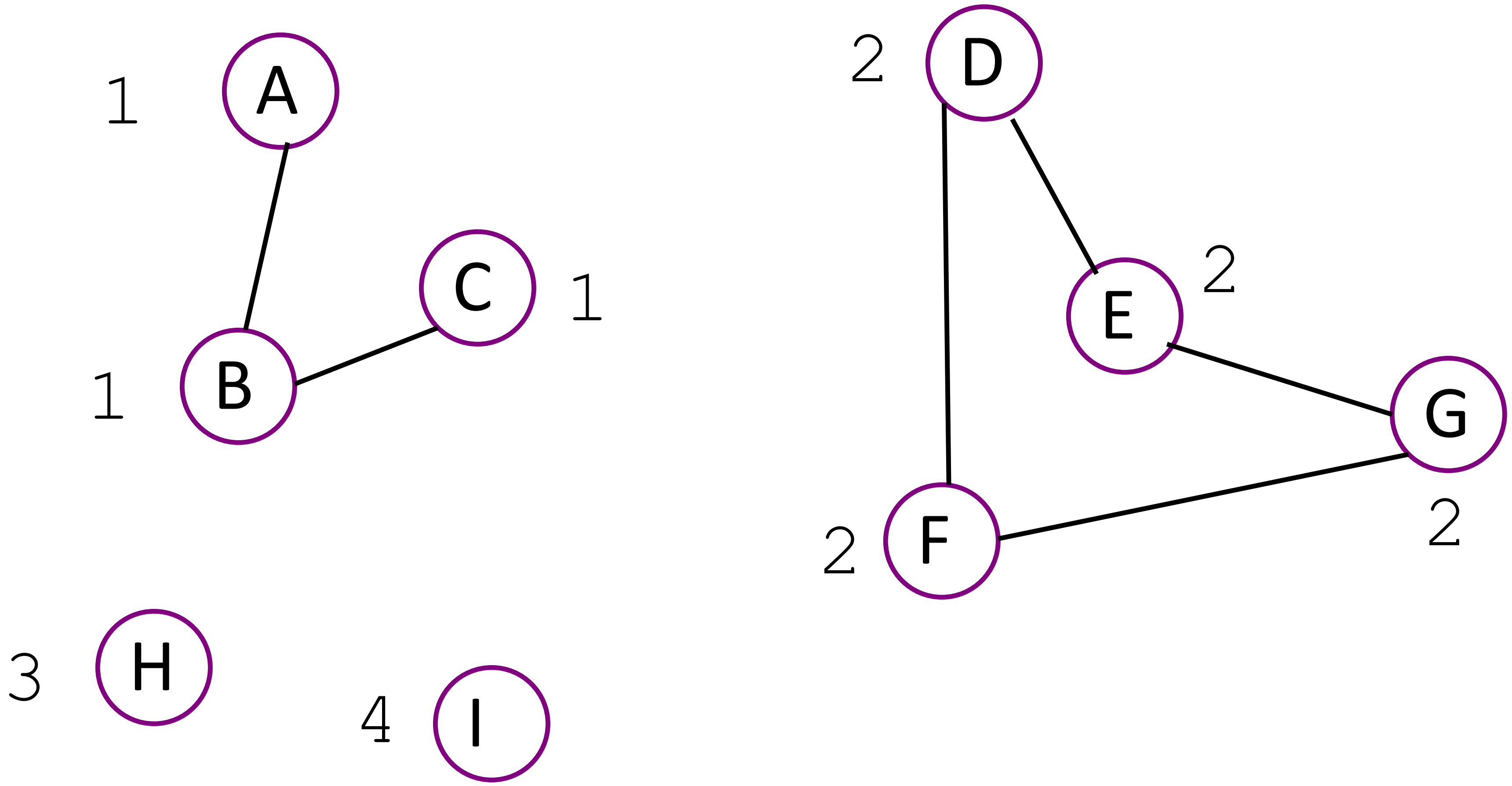
Example

CCNum: 4



Example

CCNum: 4



Discussion about DFS

What does DFS actually *do*?

Discussion about DFS

What does DFS actually *do*?

- No output.
- Marks all vertices as visited.
- Easier ways to do this.

Discussion about DFS

What does DFS actually *do*?

- No output.
- Marks all vertices as visited.
- Easier ways to do this.

However, DFS also is a useful way to explore the graph. By *augmenting* the algorithm a bit (like we did with the connected components algorithm), we can learn useful things.

Pre- and Post- Orders

Augment how?

- Keep track of what algorithm does & in what order.
- Have a "clock" and note time whenever:
  - Algorithm visits a new vertex for the first time.
  - Algorithm finishes processing a vertex.
- Record values as `v.pre` and `v.post`.

Computing Pre- & Post- Orders

```
ConnectedComponents(G)

  For v ∈ G
    v.visited ← false
  For v ∈ G
    If not v.visited
      explore(v)
```

```
explore(v)
  v.visited ← true


  For each edge (v,w)
    If not w.visited
      explore(w)
```

Computing Pre- & Post- Orders

```
ConnectedComponents(G)
  clock ← 1
  For v ∈ G
    v.visited ← false
  For v ∈ G
    If not v.visited
      explore(v)
```

```
explore(v)
  v.visited ← true


  For each edge (v,w)
    If not w.visited
      explore(w)
```

Computing Pre- & Post- Orders

```
ConnectedComponents(G)
  clock ← 1
  For v ∈ G
    v.visited ← false
  For v ∈ G
    If not v.visited
      explore(v)
```

```
explore(v)
  v.visited ← true
  v.pre ← clock
  clock++
  For each edge (v,w)
    If not w.visited
      explore(w)
```

Computing Pre- & Post- Orders

```
ConnectedComponents(G)
  clock ← 1
  For v ∈ G
    v.visited ← false
  For v ∈ G
    If not v.visited
      explore(v)
```

```
explore(v)
  v.visited ← true
  v.pre ← clock
  clock++
  For each edge (v,w)
    If not w.visited
      explore(w)
  v.post ← clock
  clock++
```

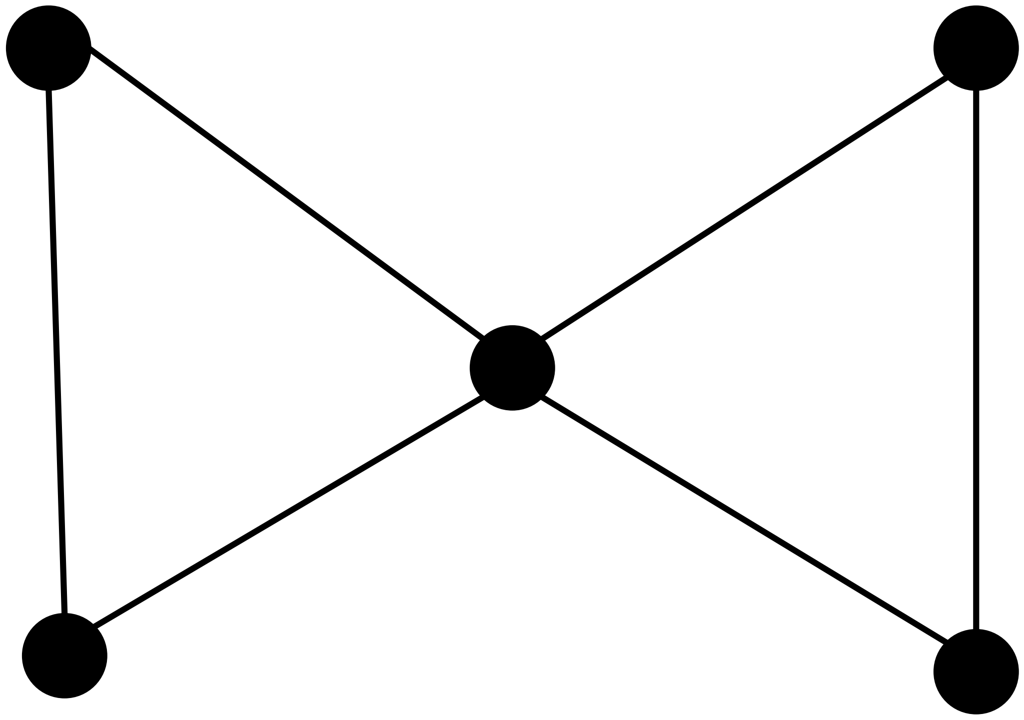
Computing Pre- & Post- Orders

```
ConnectedComponents(G)
  clock ← 1
  For v ∈ G
    v.visited ← false
  For v ∈ G
    If not v.visited
      explore(v)
```

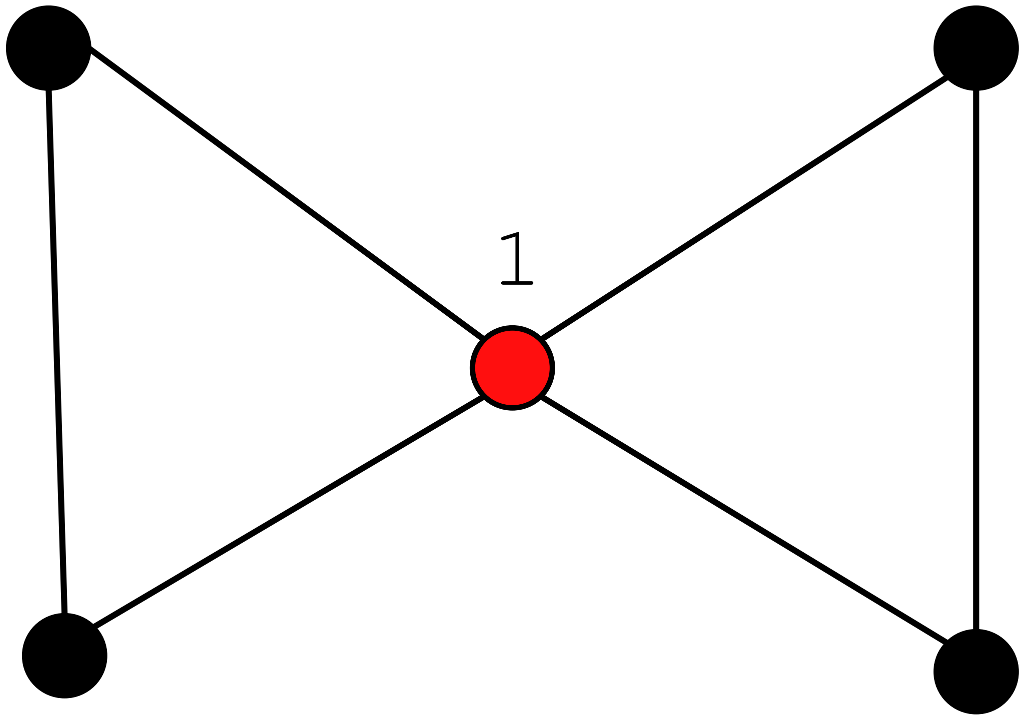
```
explore(v)
  v.visited ← true
  v.pre ← clock
  clock++
  For each edge (v,w)
    If not w.visited
      explore(w)
  v.post ← clock
  clock++
```

Runtime O(|V|+|E|).

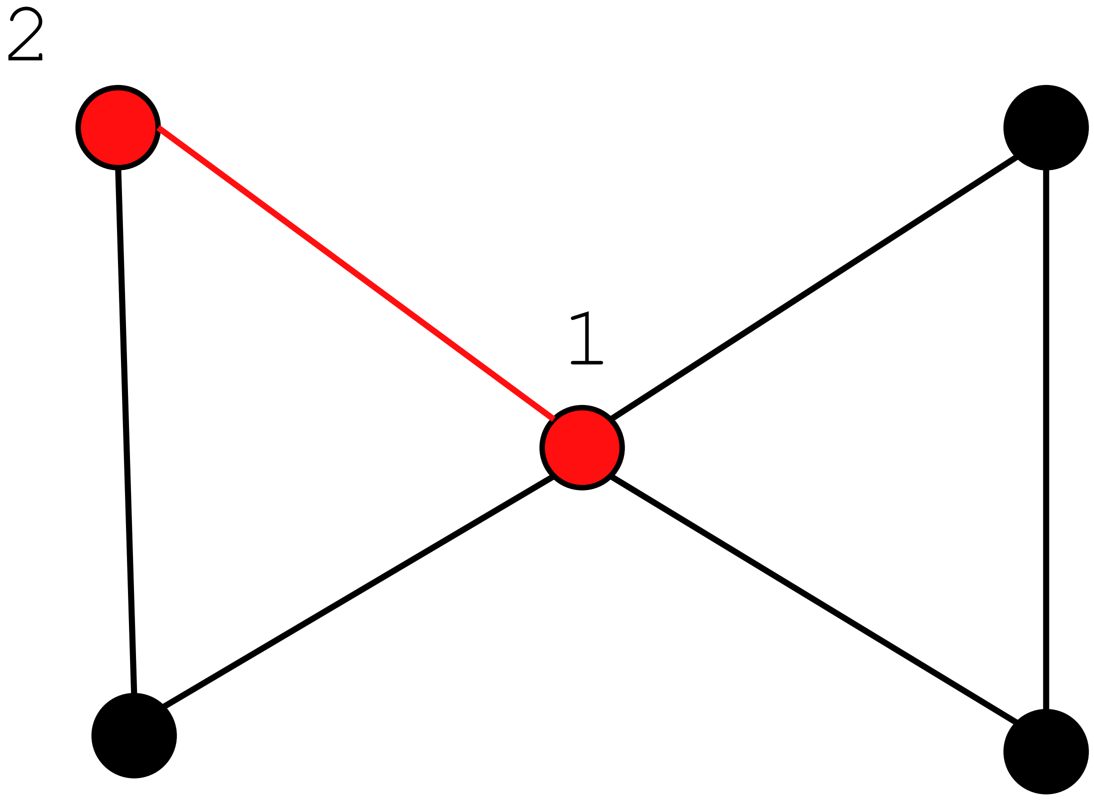
Example



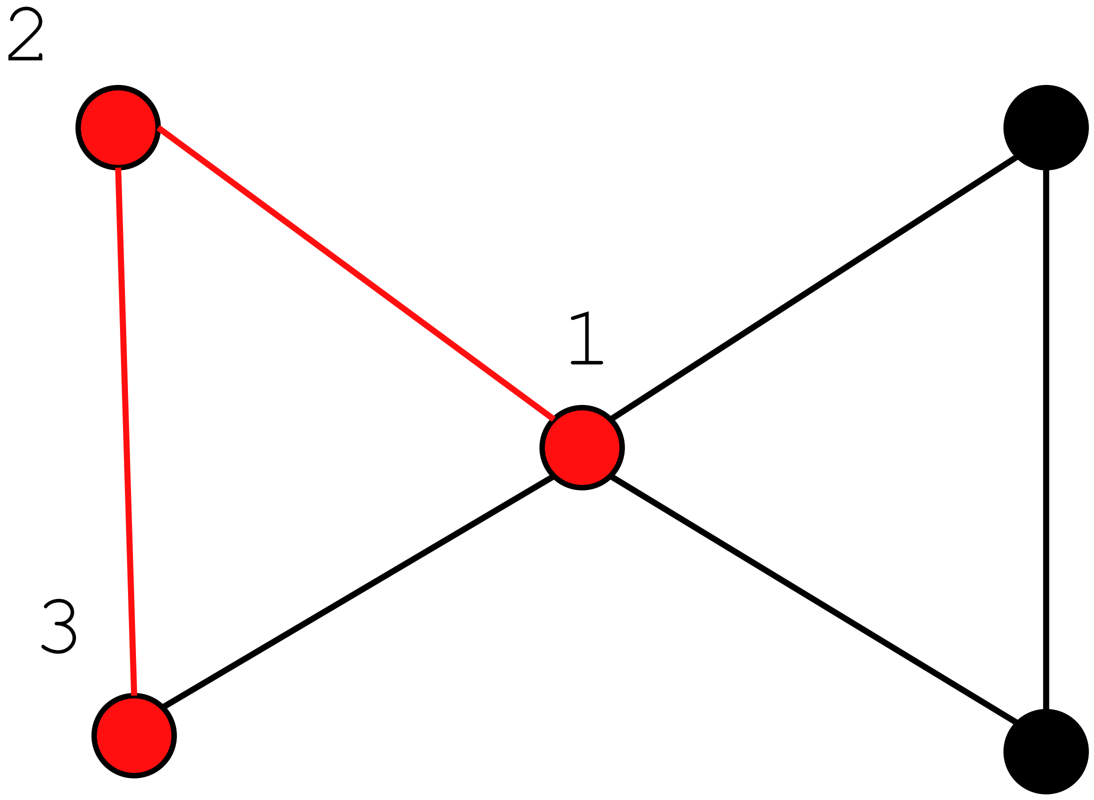
Example



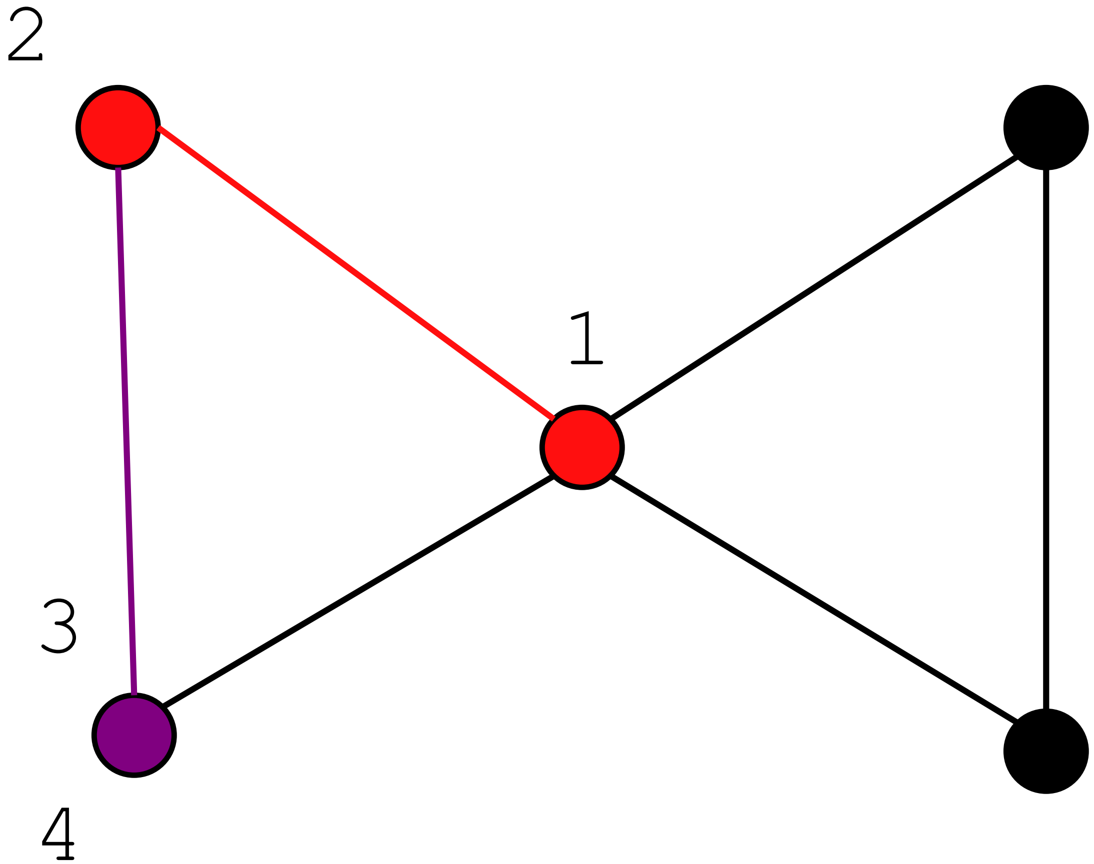
Example



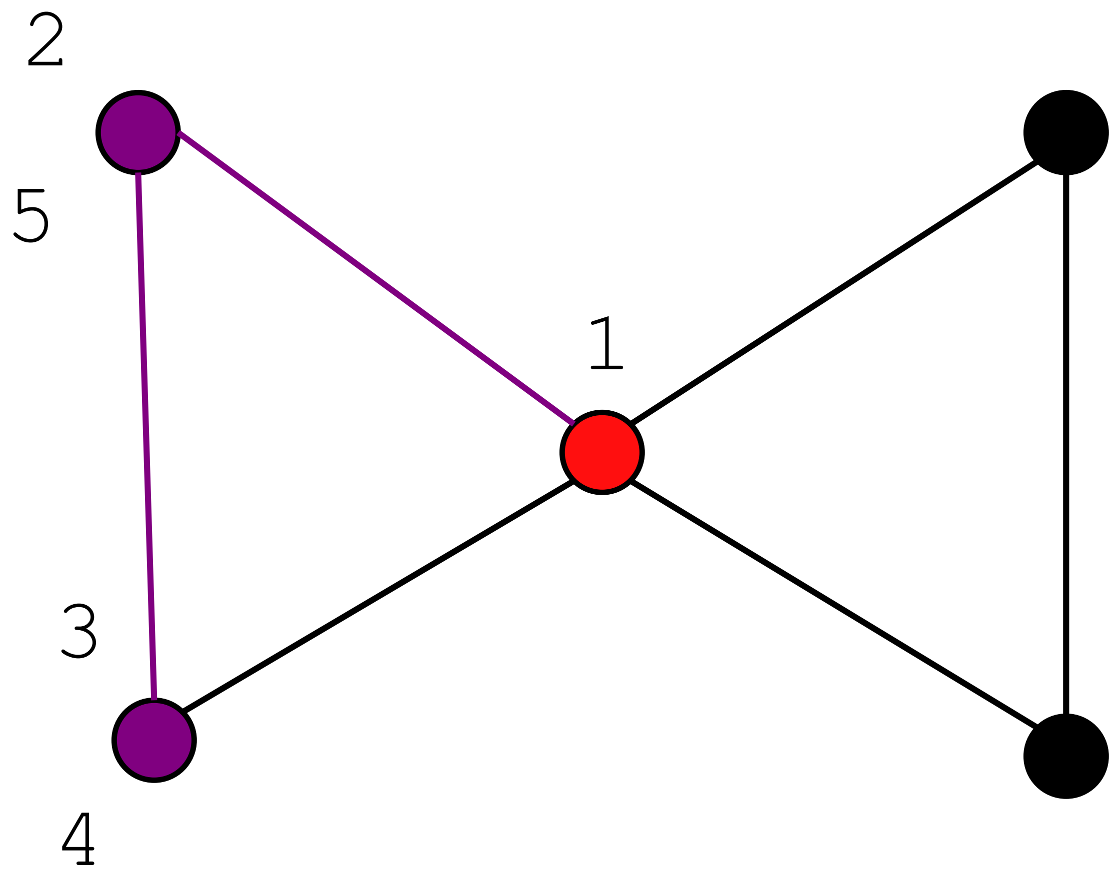
Example



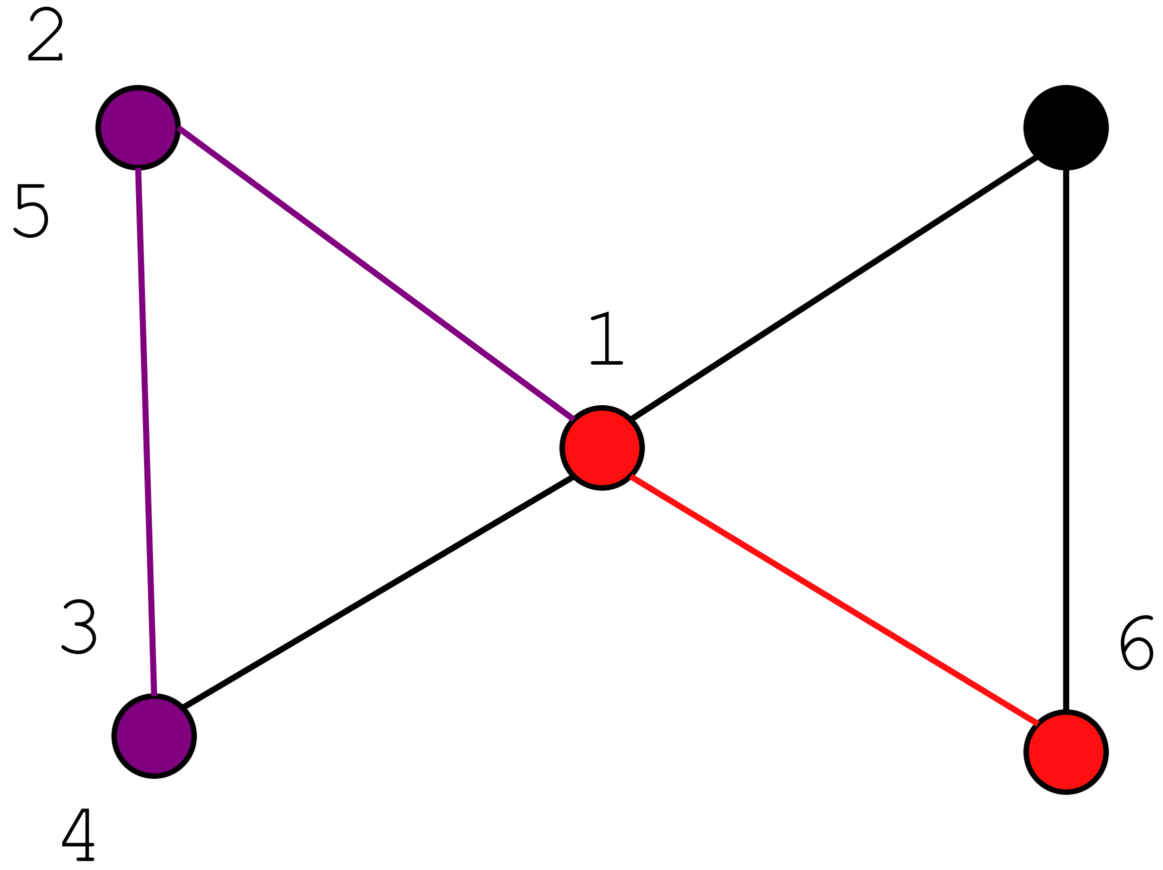
Example



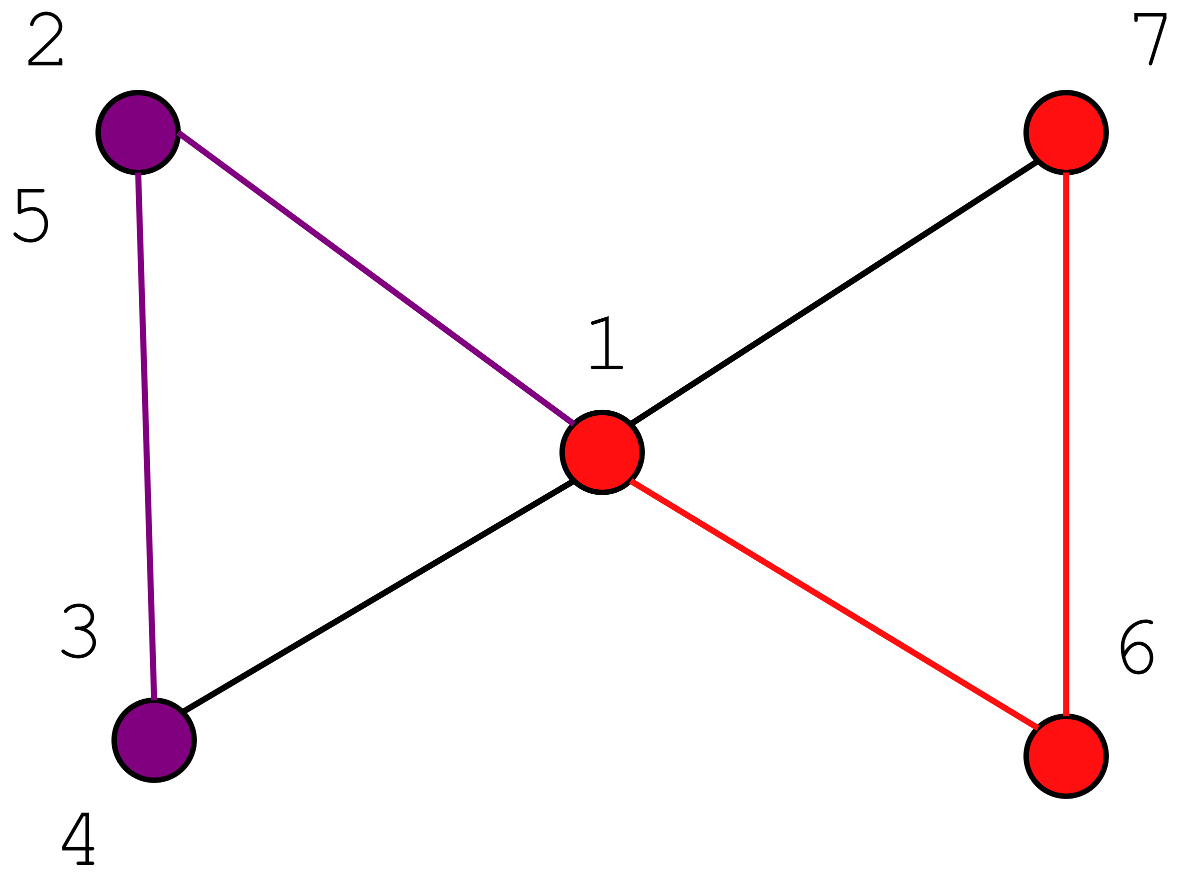
Example



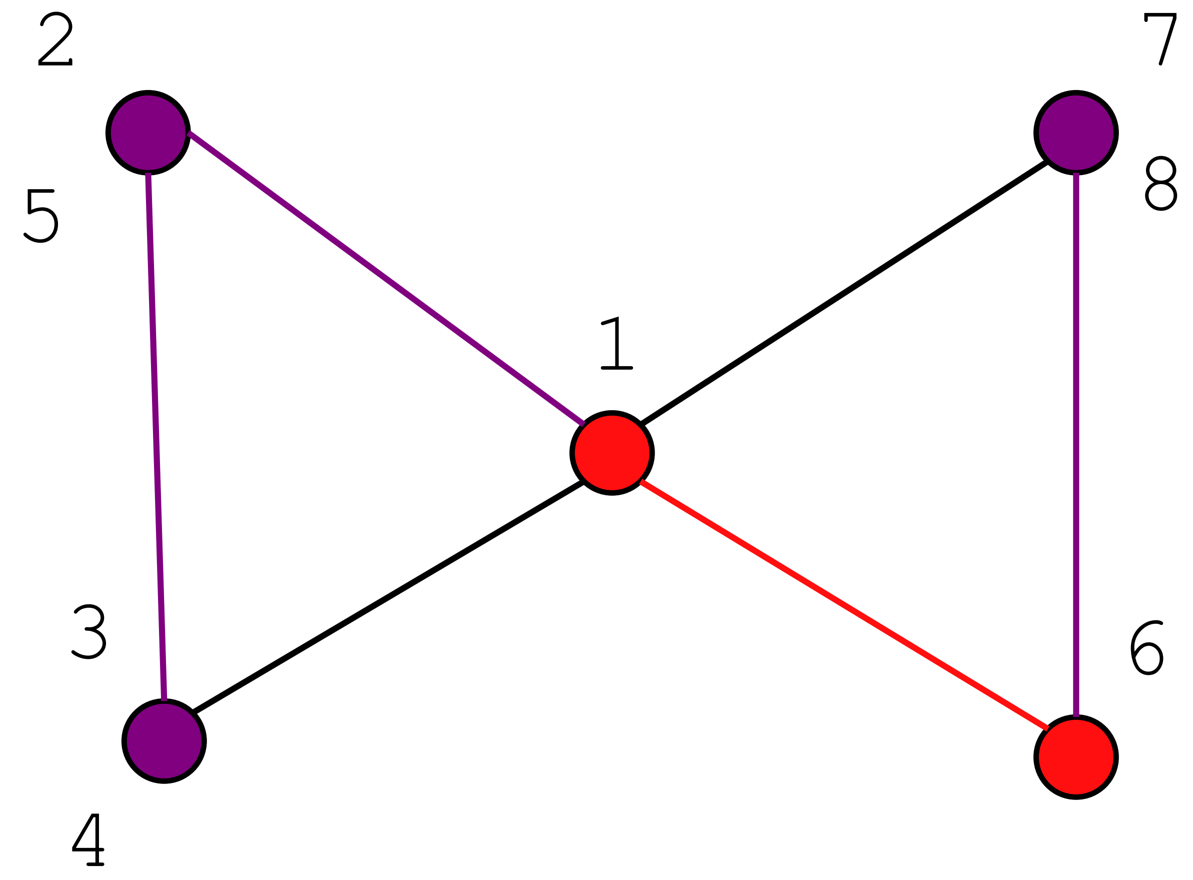
Example



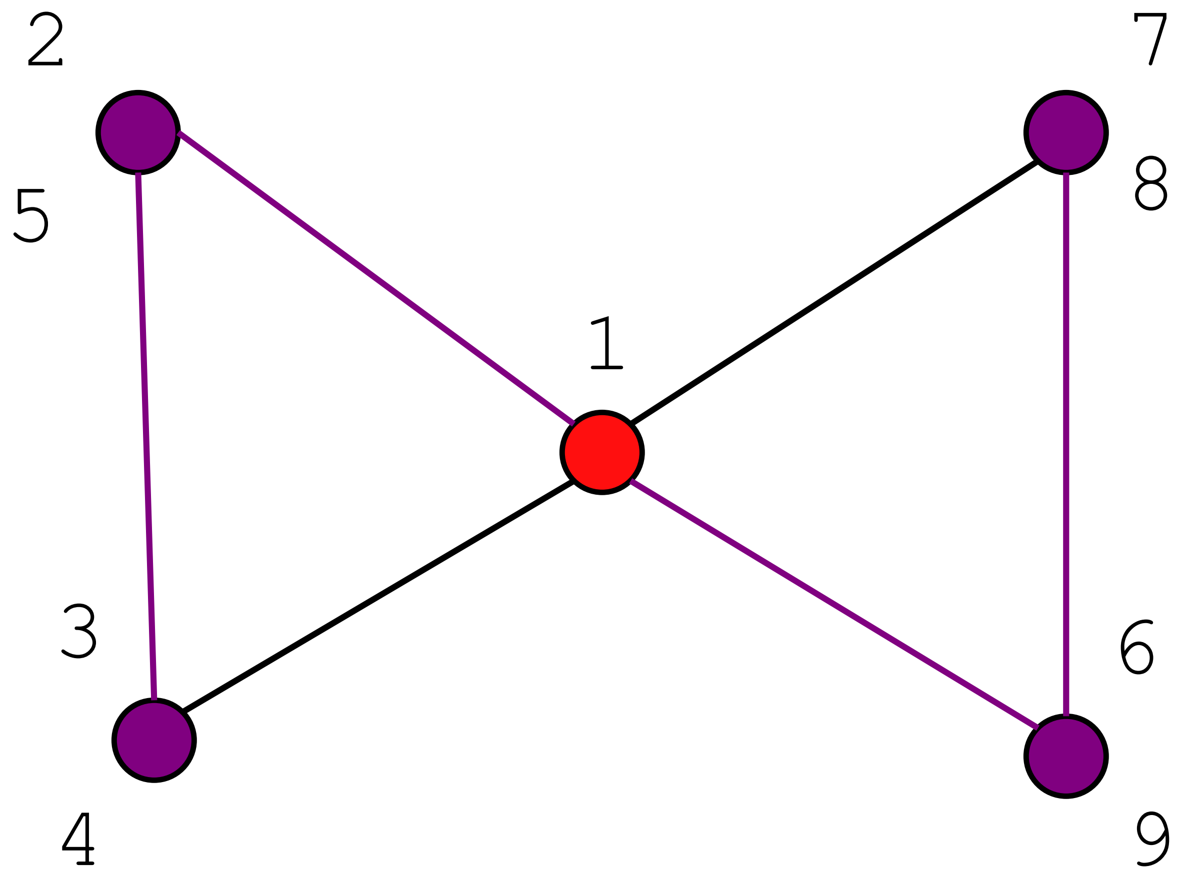
Example



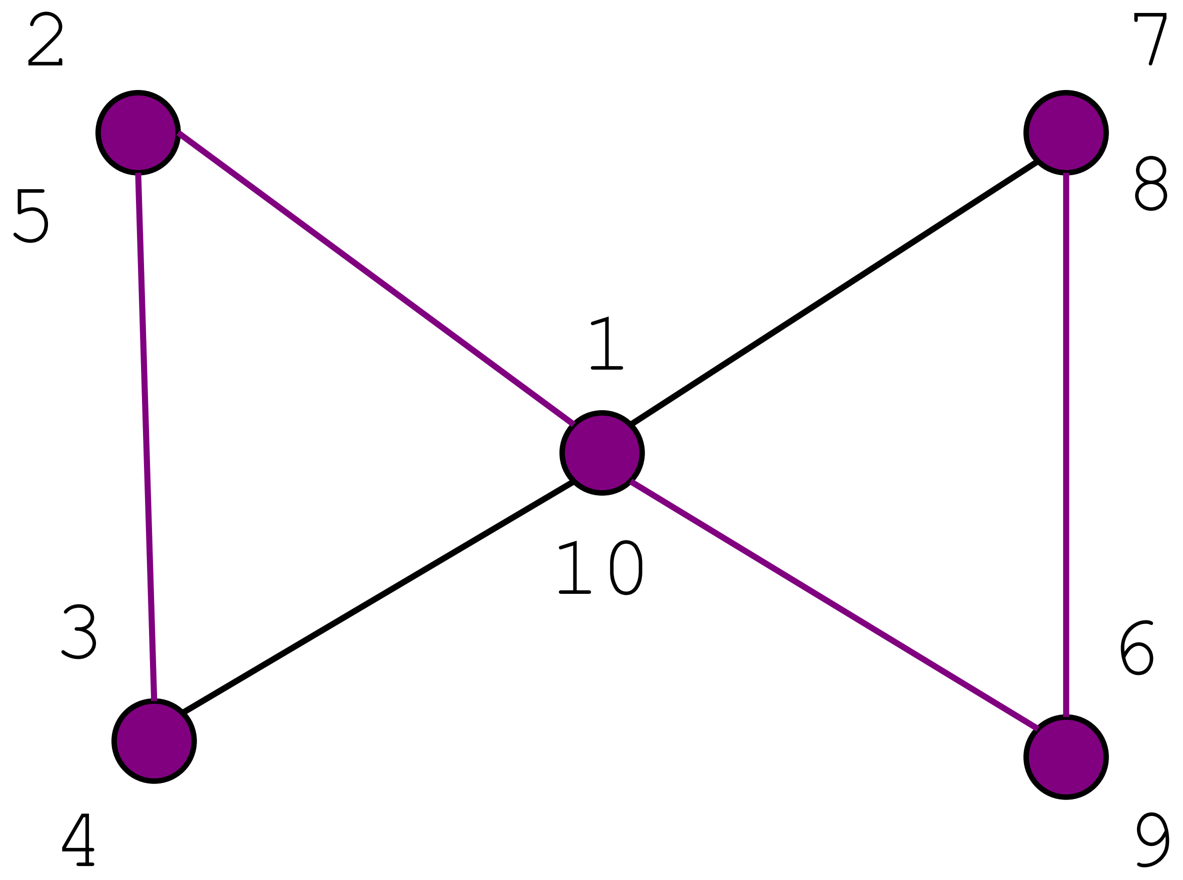
Example



Example



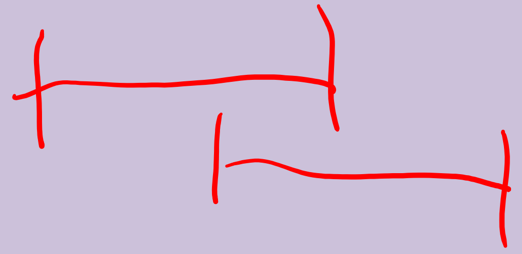
Example



What do these orders tell us?

Prop: For vertices v, w consider intervals [`v.pre`, `v.post`] and [`w.pre`, `w.post`].

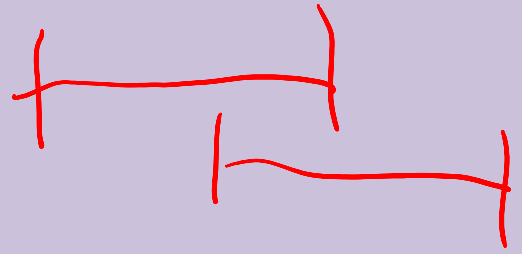
These intervals:



What do these orders tell us?

Prop: For vertices v, w consider intervals `[v.pre, v.post]` and `[w.pre, w.post]`. These intervals:

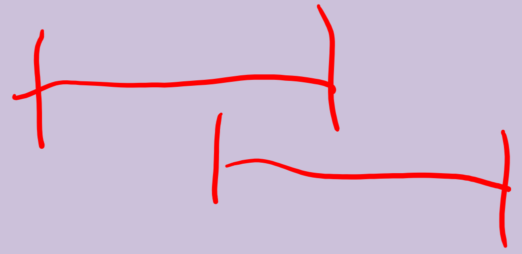
1. Contain each other if v is an ancestor/descendant of w in the DFS tree.



What do these orders tell us?

<u>Prop:</u> For vertices v, w consider intervals [`v.pre`, `v.post`] and [`w.pre`, `w.post`]. These intervals:

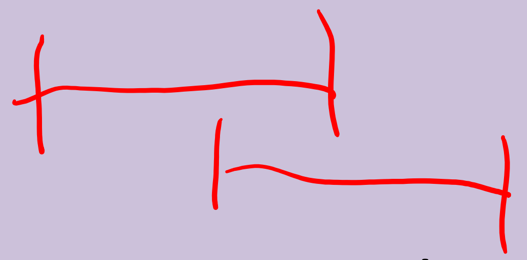
1. Contain each other if v is an ancestor/descendant of w in the DFS tree.
2. Are disjoint if v and w are cousins in the DFS tree.



What do these orders tell us?

Prop: For vertices v, w consider intervals [`v.pre`, `v.post`] and [`w.pre`, `w.post`]. These intervals:

1. Contain each other if v is an ancestor/descendant of w in the DFS tree.
2. Are disjoint if v and w are cousins in the DFS tree.
3. Never interleave (`v.pre` < `w.pre` < `v.post` < `w.post`)



Proof

- Assume algorithm finds v before w
  (`v.pre` < `w.pre`)

Proof

- Assume algorithm finds v before w
  (`v.pre` < `w.pre`)
- If algorithm discovers w *after* fully processing v:
  - `v.post` < `w.pre`
  - Intervals disjoint
  - v and w are cousins

Proof

- Assume algorithm finds v before w (`v.pre` < `w.pre`)
- If algorithm discovers w *after* fully processing v:
  - `v.post` < `w.pre`
  - Intervals disjoint
  - v and w are cousins
- If algorithm discovers w *before* fully processing v:
  - Algorithm finishes processing w before it finishes v
  - `v.pre` < `w.pre` < `w.post` < `v.post`
  - Nested intervals
  - v is ancestor of w

Question: Possible Intervals

Which pairs of pre-post intervals are not possible for DFS? (multiple correct answers)

A) [1,2] & [3,4]

B) [1,3] & [2,4]

C) [1,4] & [2,3]

D) [1,5] & [2,4]

E) [1,6] & [2,5]

Question: Possible Intervals

Which pairs of pre-post intervals are not possible for DFS? (multiple correct answers)

A) [1,2] & [3,4]
B) [1,3] & [2,4]
C) [1,4] & [2,3]
D) [1,5] & [2,4]
E) [1,6] & [2,5]

Directed Graphs

Often an edge makes sense both ways, but sometimes streets are one directional.

Directed Graphs

Often an edge makes sense both ways, but sometimes streets are one directional.

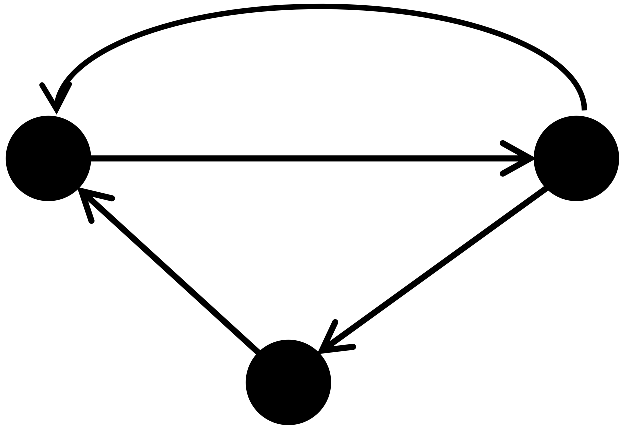
<u>Definition:</u> A <u>directed graph</u> is a graph where each edge has a direction. Goes *from* v *to* w.

Directed Graphs

Often an edge makes sense both ways, but sometimes streets are one directional.

Definition: A directed graph is a graph where each edge has a direction. Goes *from* v *to* w.

Draw edges with arrows to denote direction.



Question: Directed Graphs

Which of the following does NOT need to be modeled as a directed rather than undirected graph:

A) The Internet (links connecting webpages)

B) Facebook (friendships connecting people)

C) Twitter (followings connecting people)

D) Maps (roads connecting intersections)

Question: Directed Graphs

Which of the following does NOT need to be modeled as a directed rather than undirected graph:

A) The Internet (links connecting webpages)
B) Facebook (friendships connecting people)
C) Twitter (followings connecting people)
D) Maps (roads connecting intersections)

DFS on Directed Graphs

DFS on Directed Graphs

- Same code

DFS on Directed Graphs

- Same code
- Only follow *directed* edges from v to w.

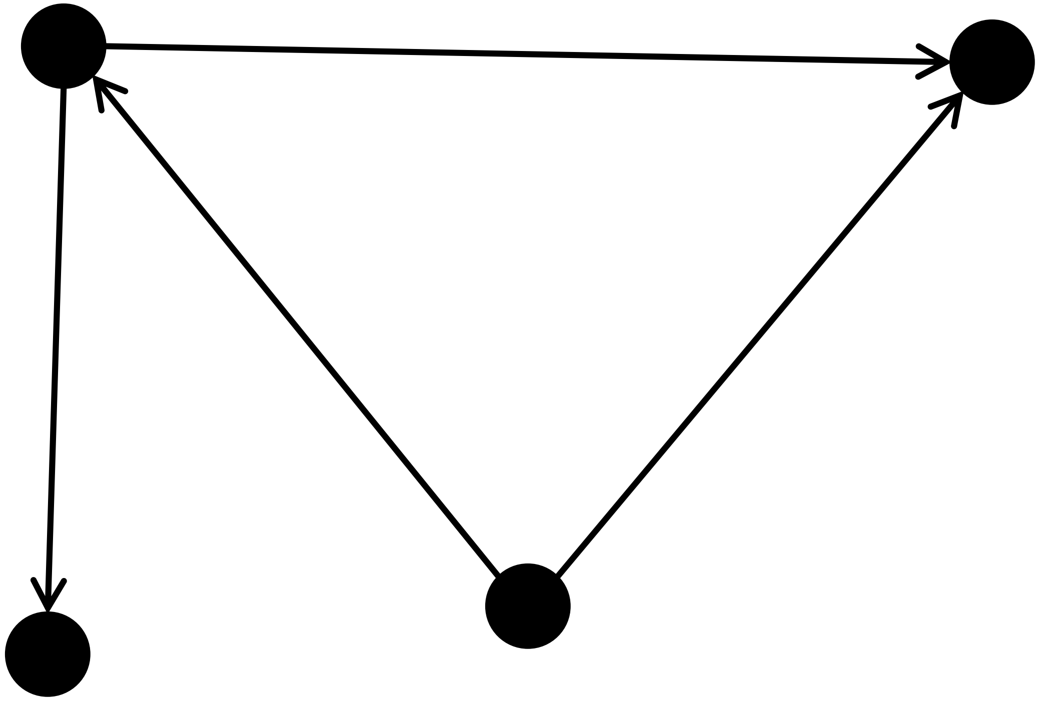
DFS on Directed Graphs

- Same code
- Only follow *directed* edges from v to w.
- Runtime still $O(|V|+|E|)$

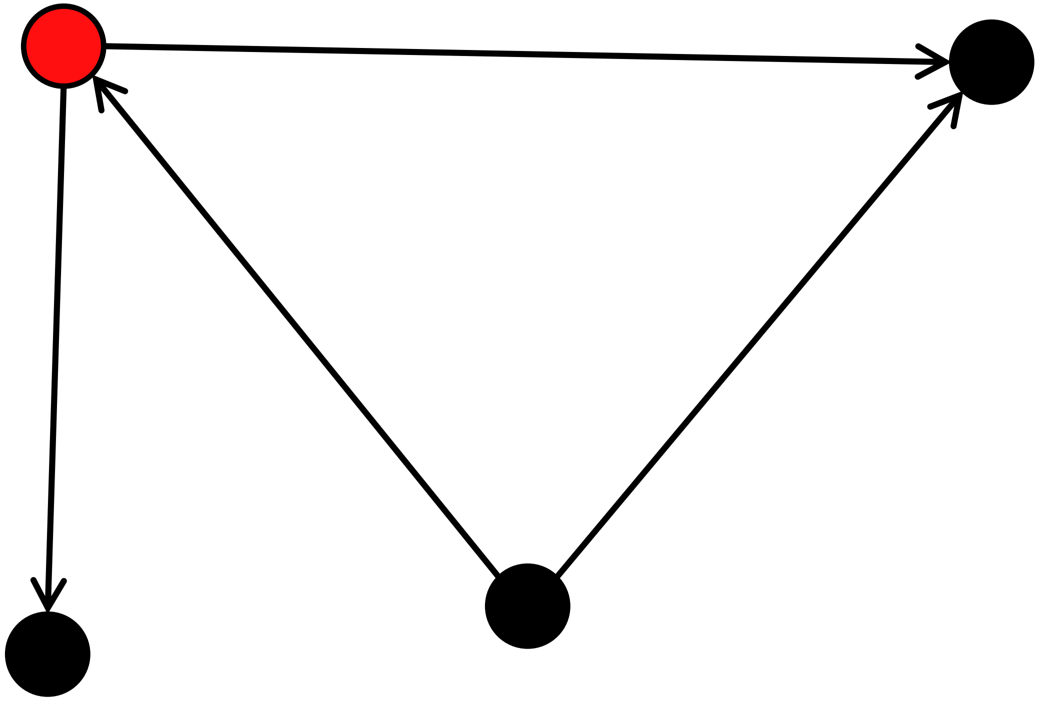
DFS on Directed Graphs

- Same code
- Only follow *directed* edges from v to w.
- Runtime still O(|V|+|E|)
- `explore(v)` discovers all vertices reachable from v following only directed edges.

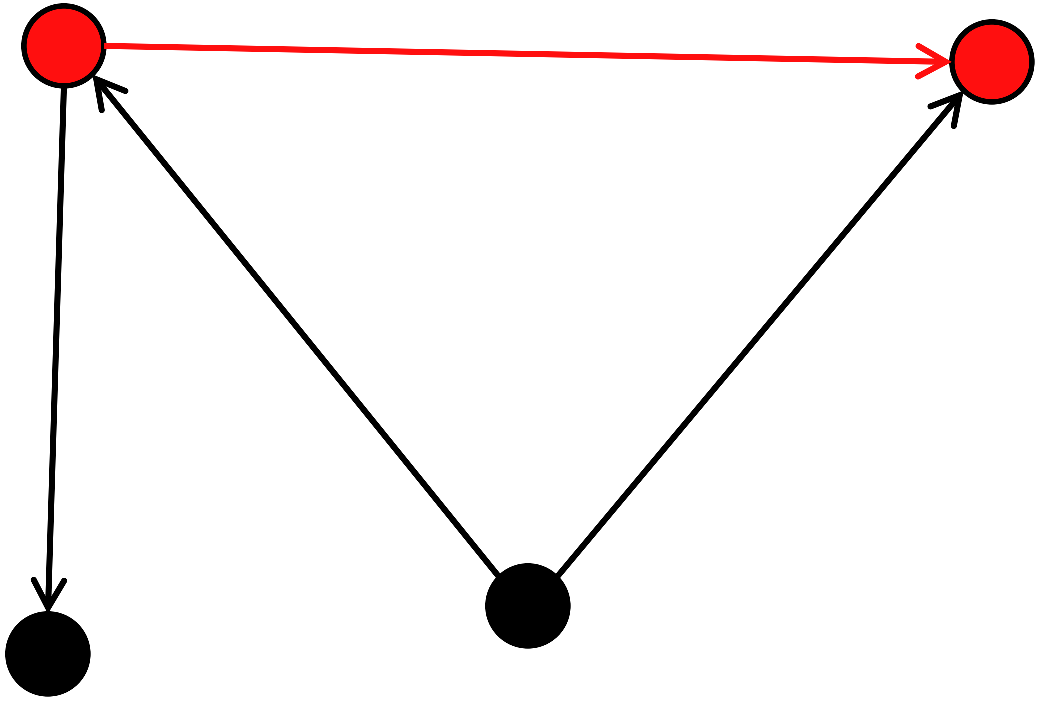
Example



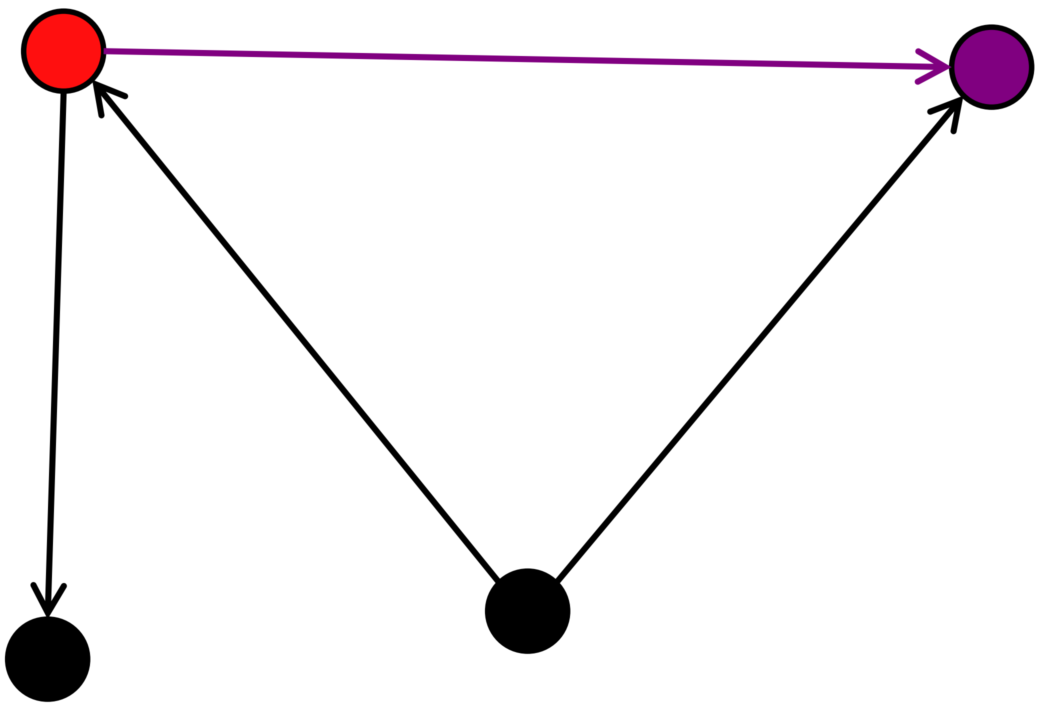
Example



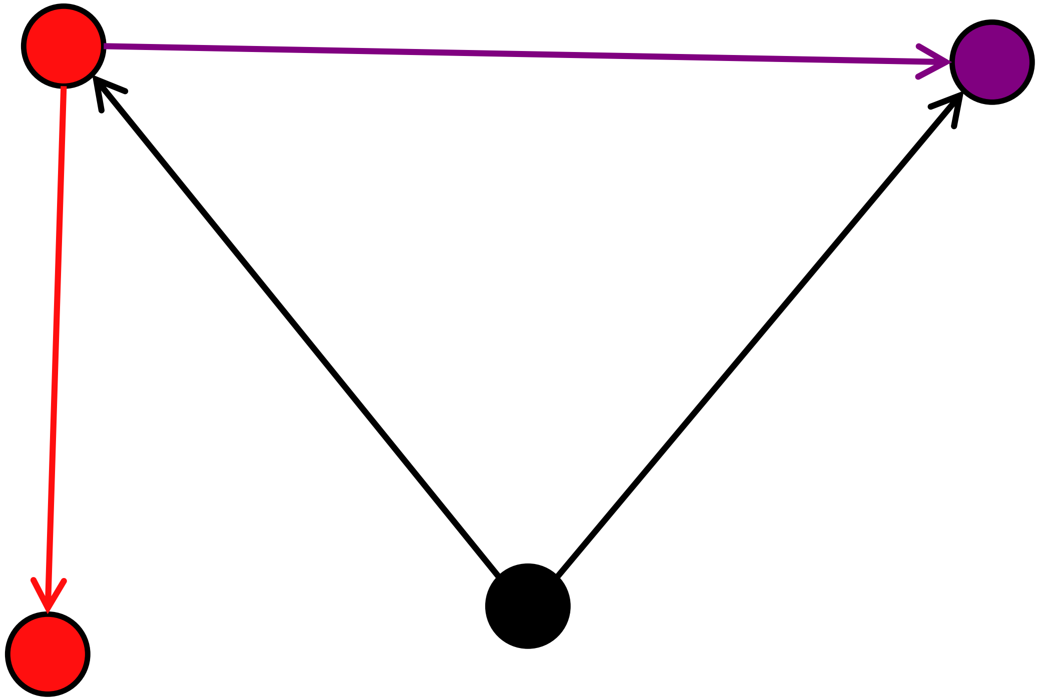
Example



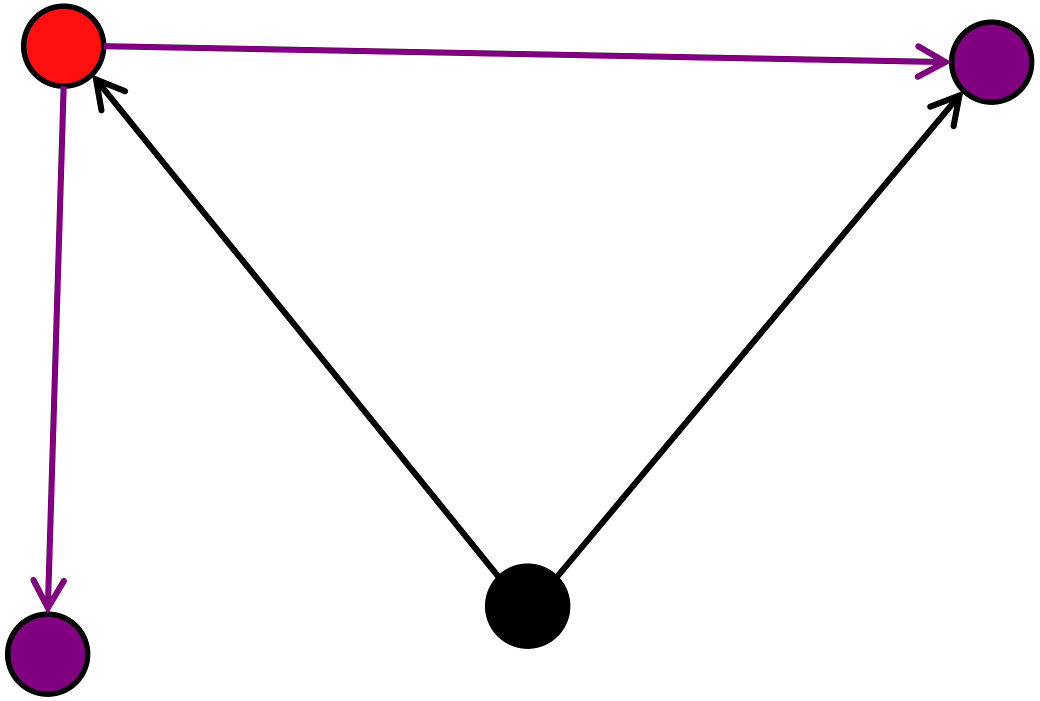
Example



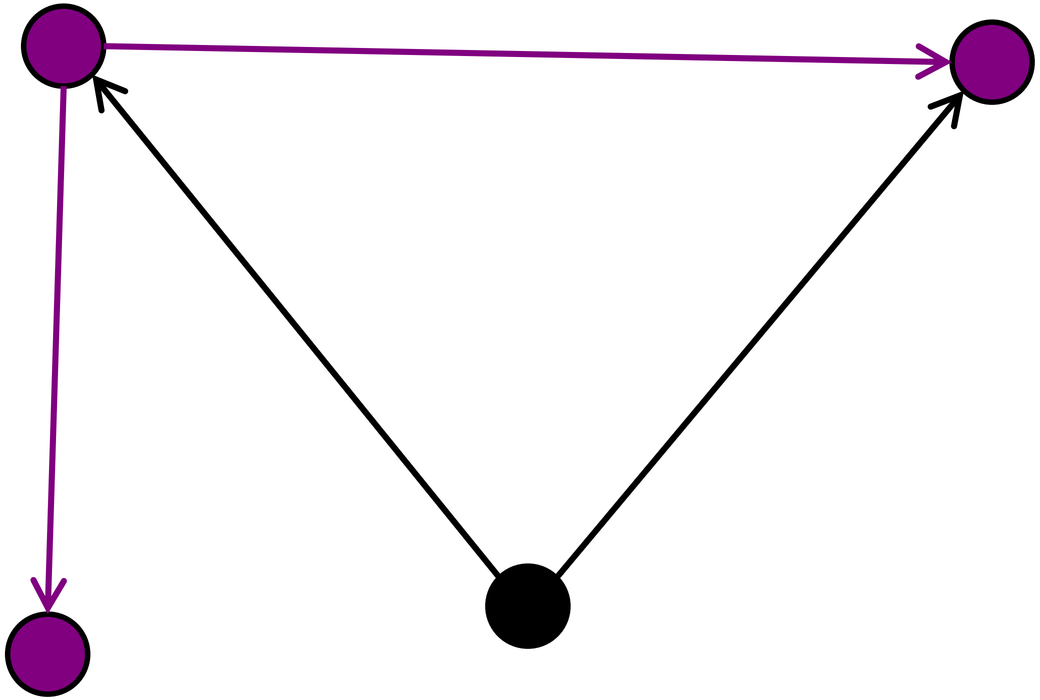
Example



Example



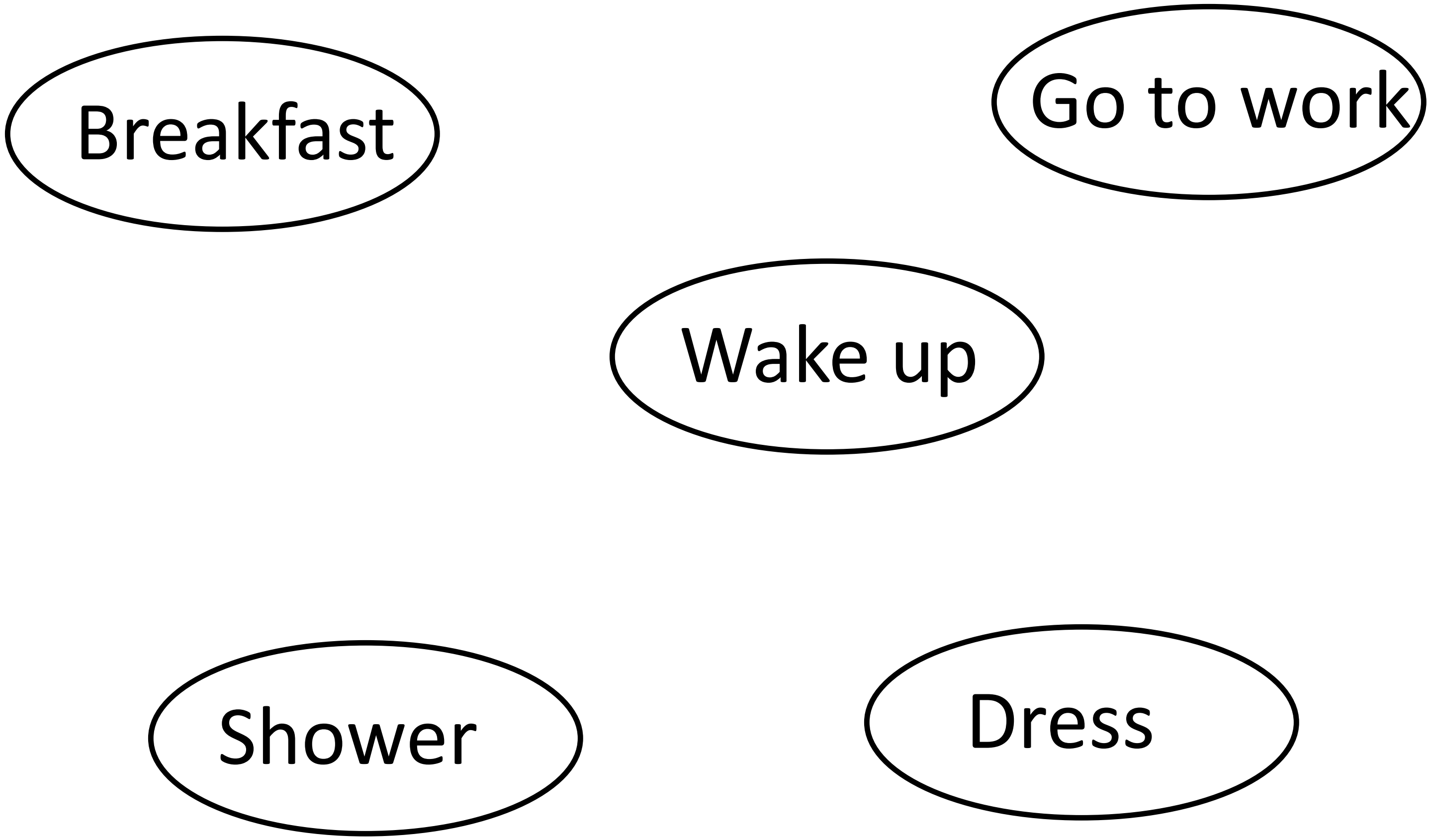
Example



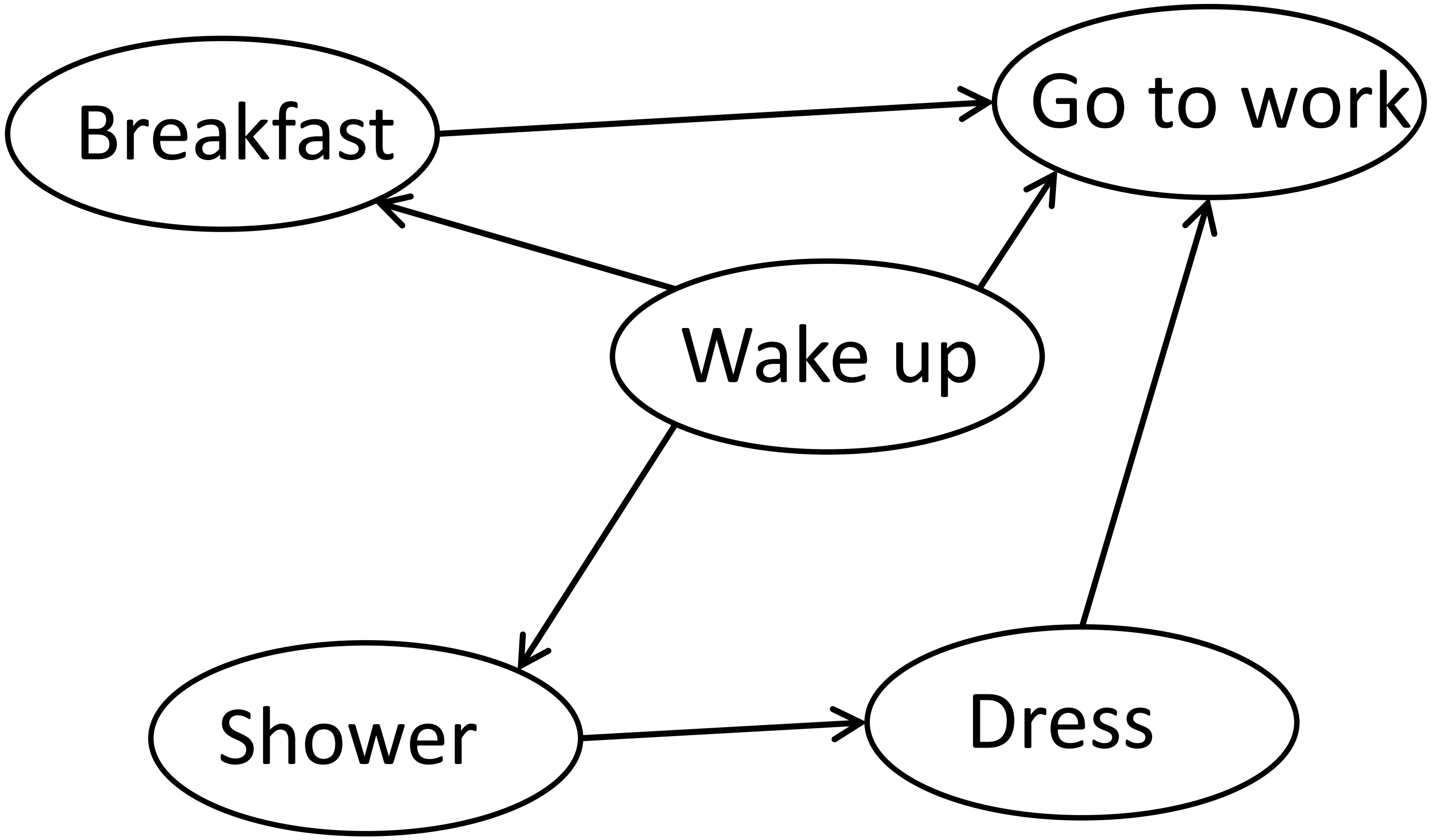
Directed Acyclic Graphs

- Directed graphs as dependencies
- Linear orderings
- DAGs definition
- Topological sort

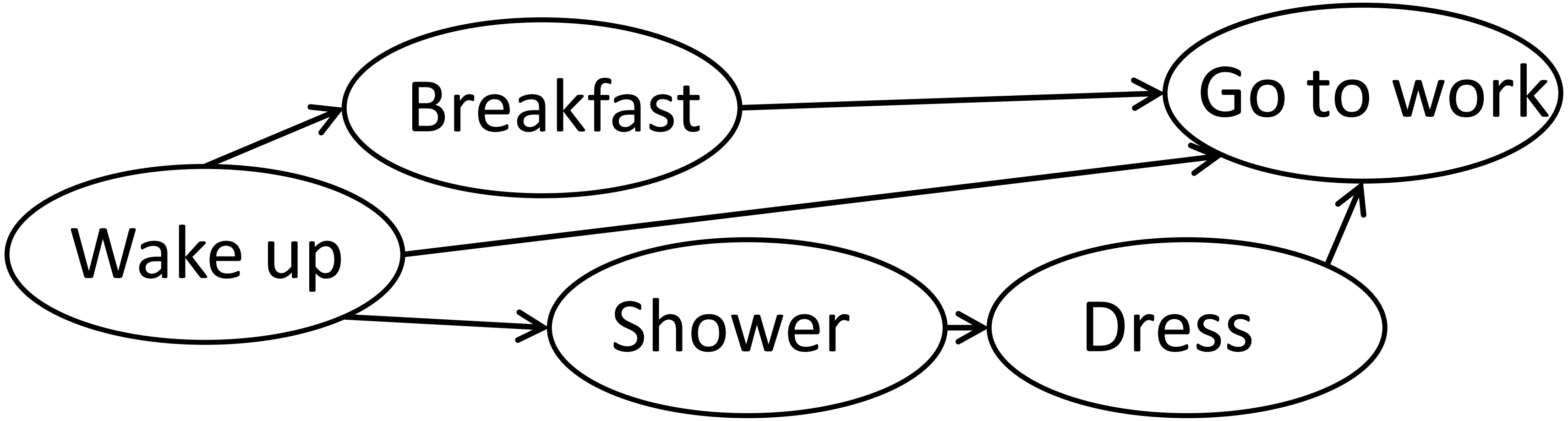
Dependency Graphs



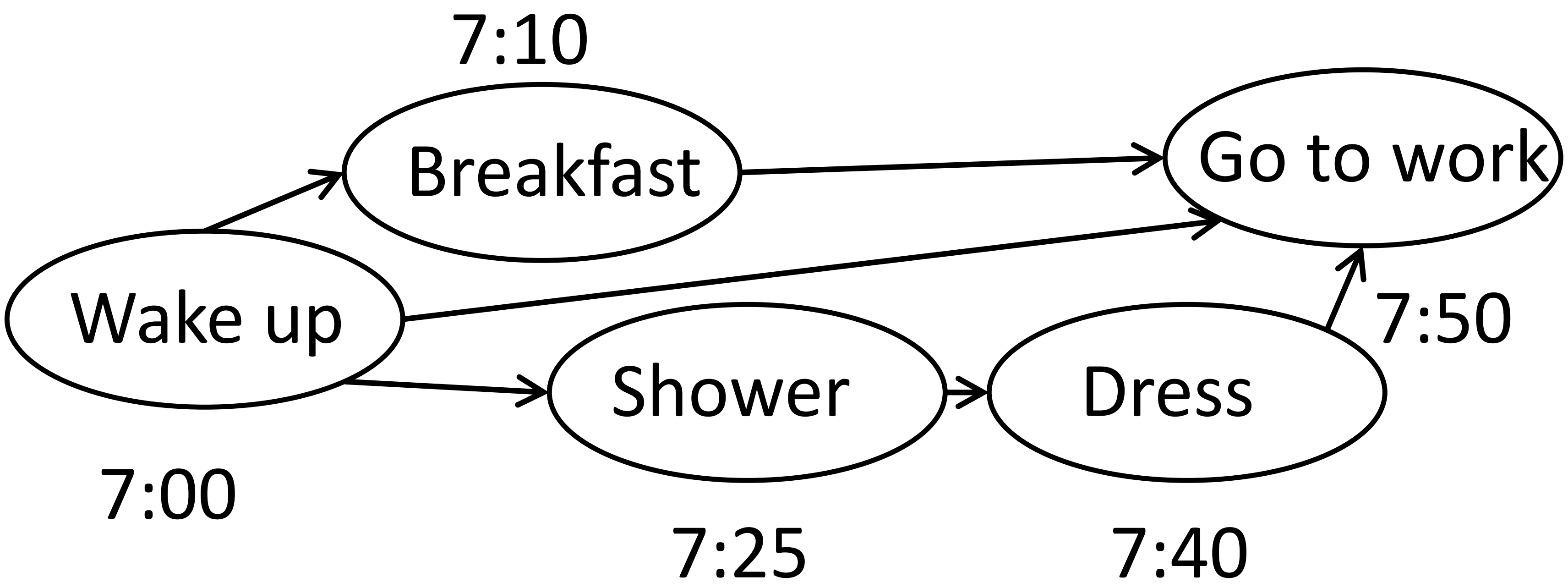
Dependency Graphs



Dependency Graphs



Dependency Graphs



Dependency Graphs

A directed graph can be thought of as a graph of dependencies.
Where an edge v→w means that v should come before w.

Dependency Graphs

A directed graph can be thought of as a graph of dependencies. Where an edge v→w means that v should come before w.

Definition: A topological ordering of a directed graph is an ordering of the vertices so that for each edge (v,w), v comes before w in the ordering.

Question: Existence of Orderings

Does every directed graph have a topological ordering?

A) Yes

B) No

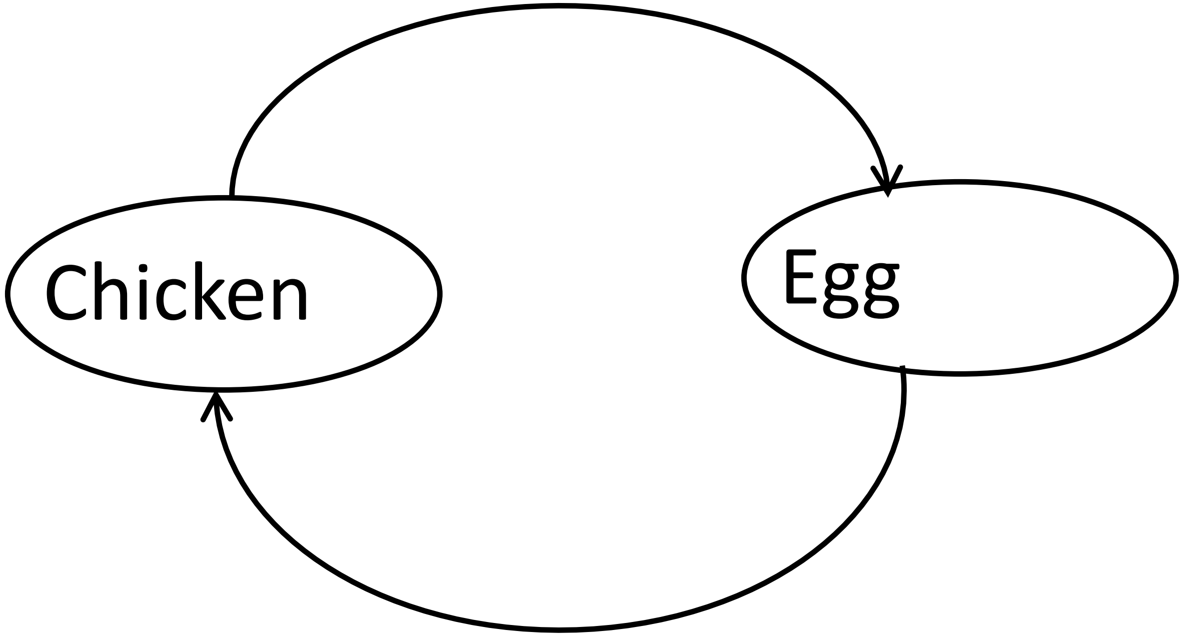
Question: Existence of Orderings

Does every directed graph have a topological ordering?

A) Yes

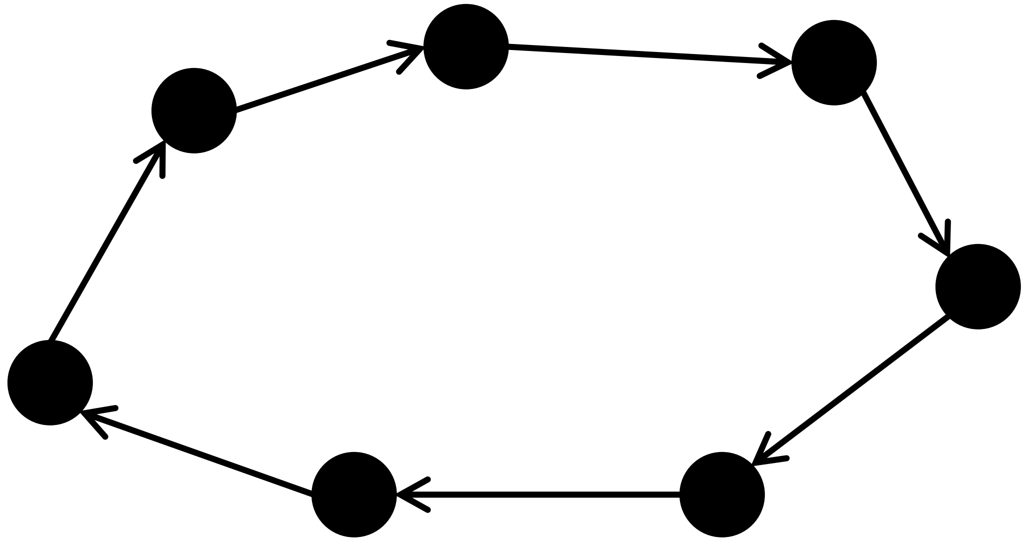
B) No

Counterexample



Cycles

<u>Definition:</u> A <u>cycle</u> in a directed graph is a sequence of vertices $v_1, v_2, v_3, \ldots, v_n$ so that there are edges $(v_1,v_2), (v_2,v_3), \ldots, (v_{n-1},v_n), (v_n,v_1)$



Obstacle

Proposition: If G is a directed graph with a cycle, then G has no topological ordering.

Obstacle

Proposition: If G is a directed graph with a cycle, then G has no topological ordering.

Proof:

- Have cycle $v_1, v_2, \ldots, v_n$.

Obstacle

Proposition: If G is a directed graph with a cycle, then G has no topological ordering.

Proof:

- Have cycle v_1, v_2,…, v_n.
- AFSOC we have an ordering.

Obstacle

Proposition: If G is a directed graph with a cycle, then G has no topological ordering.

Proof:

- Have cycle $v_1, v_2, \ldots, v_n$.
- AFSOC we have an ordering.
- Find earliest v_i in the ordering.

Obstacle

Proposition: If G is a directed graph with a cycle, then G has no topological ordering.

Proof:

- Have cycle $v_1, v_2, \ldots, v_n$.
- AFSOC we have an ordering.
- Find earliest v_i in the ordering.
- Note that v_i comes before v_{i-1}, in contradiction to the order property.

DAGs

Definition: A Directed Acyclic Graph (DAG) is a directed graph which contains no cycles.

DAGs

Definition: A Directed Acyclic Graph (DAG) is a directed graph which contains no cycles.

Previous result said that *only* DAGs can be topologically ordered. Is the reverse true?

DAGs

Definition: A Directed Acyclic Graph (DAG) is a directed graph which contains no cycles.

Previous result said that *only* DAGs can be topologically ordered. Is the reverse true?

Surprisingly, yes.

Existence of Orderings

Theorem: Let G be a (finite) DAG. Then G has a topological ordering.

Existence of Orderings

Theorem: Let G be a (finite) DAG. Then G has a topological ordering.

Proof:

- Consider the *last* vertex in the ordering.

Existence of Orderings

<u>Theorem:</u> Let G be a (finite) DAG. Then G has a topological ordering.

<u>Proof:</u>

- Consider the *last* vertex in the ordering.
  - Must be a *sink* (vertex with no outgoing edges).

Existence of Orderings

<u>Theorem:</u> Let G be a (finite) DAG. Then G has a topological ordering.

<u>Proof:</u>

- Consider the *last* vertex in the ordering.
  - Must be a *sink* (vertex with no outgoing edges).
- <u>Idea:</u> find a sink, put at end, order remaining.

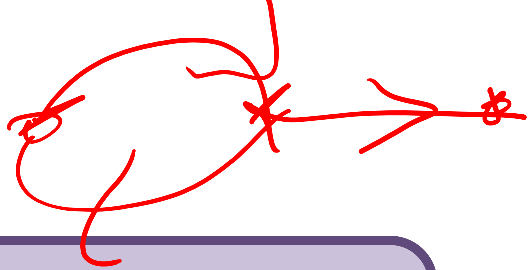
Existence of Orderings

Theorem: Let G be a (finite) DAG. Then G has a topological ordering.

Proof:

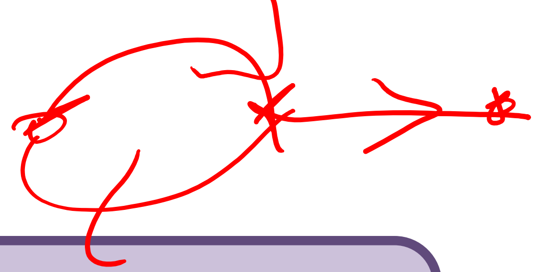
- Consider the *last* vertex in the ordering.
  - Must be a *sink* (vertex with no outgoing edges).
- Idea: find a sink, put at end, order remaining.
- Question: Does G have a sink?

Sinks



Lemma: Every finite DAG contains at least one sink.

Sinks

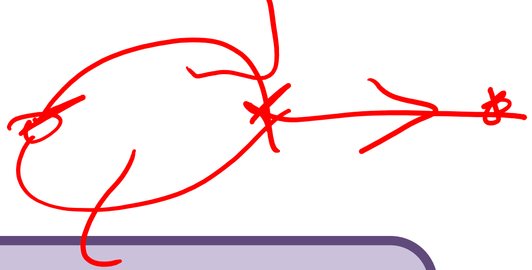


> **Lemma:** Every finite DAG contains at least one sink.

Proof:

- Start at vertex $v = v_1$

Sinks

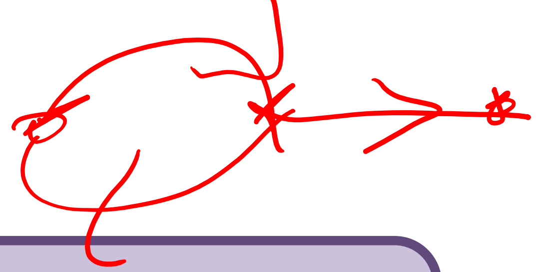


Lemma: Every finite DAG contains at least one sink.

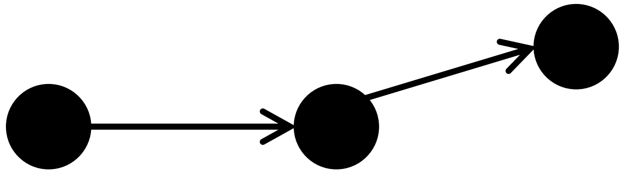
Proof:

- Start at vertex $v = v_1$
- Find edges (v_1,v_2), (v_2,v_3), (v_3, v_4)…

Sinks



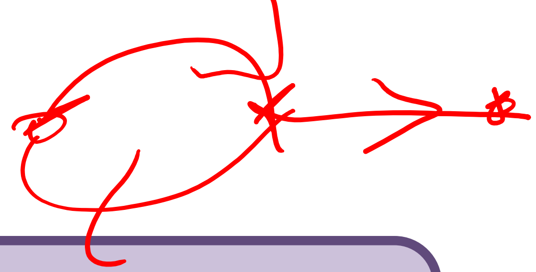
Lemma: Every finite DAG contains at least one sink.



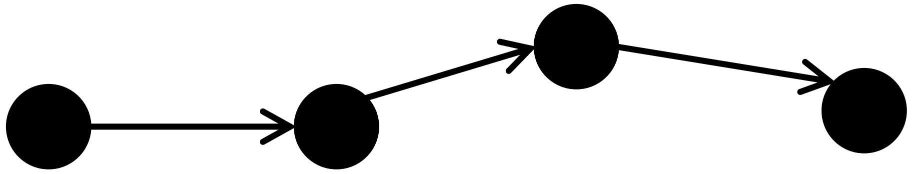
Proof:

- Start at vertex $v = v_1$
- Find edges (v_1,v_2), (v_2,v_3), (v_3, v_4)…

Sinks



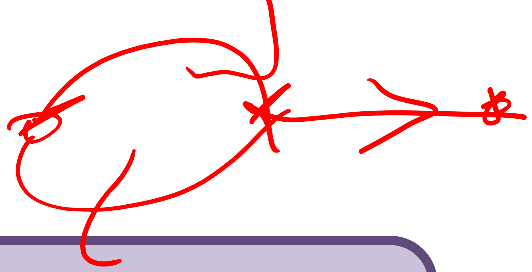
Lemma: Every finite DAG contains at least one sink.



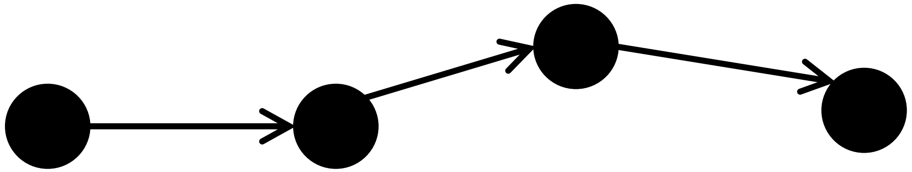
Proof:

- Start at vertex $v = v_1$
- Find edges (v_1,v_2), (v_2,v_3), (v_3, v_4)…

Sinks



Lemma: Every finite DAG contains at least one sink.

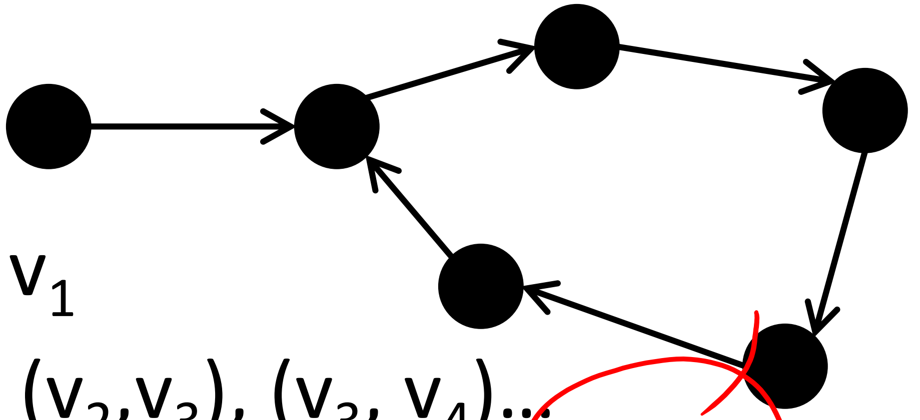


Proof:

- Start at vertex $v = v_1$
- Find edges (v_1,v_2), (v_2,v_3), (v_3, v_4)…
- Eventually either:

Sinks

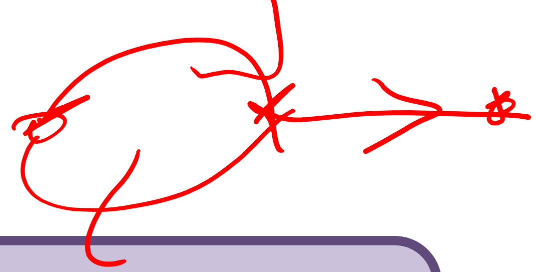
Lemma: Every finite DAG contains at least one sink.



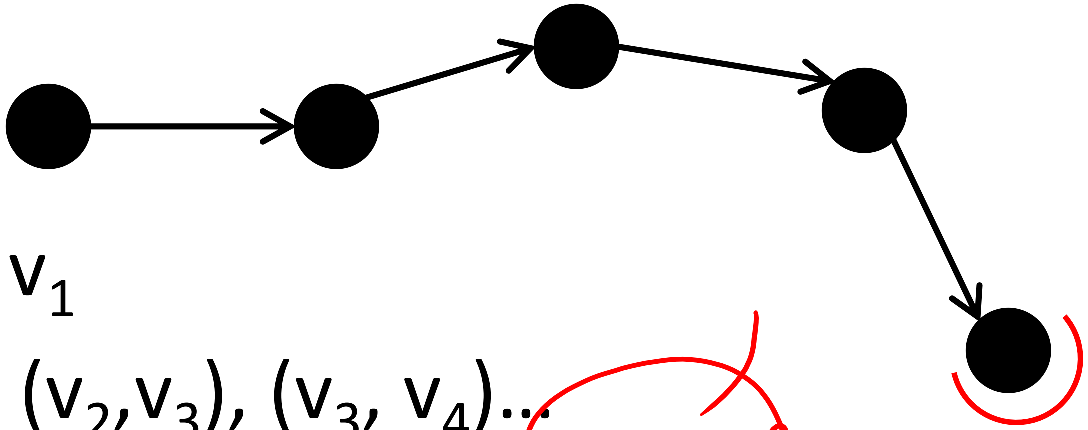
Proof:

- Start at vertex $v = v_1$
- Find edges (v_1,v_2), (v_2,v_3), (v_3, v_4)…
- Eventually either:
  - – Some vertex repeats (create cycle)

Sinks



Lemma: Every finite DAG contains at least one sink.



Proof:

- Start at vertex v = v_1
- Find edges (v_1,v_2), (v_2,v_3), (v_3, v_4)…
- Eventually either:
  - Some vertex repeats (create cycle)
  - Get stuck (found a sink)

Proof of Theorem

- Induction on $|G|$.

Proof of Theorem

- Induction on $|G|$.
- Find sink v.

Proof of Theorem

- Induction on $|G|$.
- Find sink v.
- Let G’ = G-v.

Proof of Theorem

- Induction on |G|.
- Find sink v.
- Let G’ = G-v.
- Inductively order G’ (still a DAG).

Proof of Theorem

- Induction on |G|.
- Find sink v.
- Let G’ = G-v.
- Inductively order G’ (still a DAG).
- Add v to the end of the ordering.

Algorithm

Problem: Design an algorithm that given a DAG G computes a topological ordering on G.