Announcements

- Homework 1 online due today
- Homework 2 online due next Friday
- Remember FinAid survey Due Today on canvas!!!
- Minor office hour schedule changes *this week*
  – Akhila: Thursday 4-6pm -> Friday 7-9pm

Last Time

- Strongly connected components and metagraphs

Strongly Connected Components

Definition: In a directed graph G, two vertices v and w are in the same Strongly Connected Component (SCC) if v is reachable from w *and* w is reachable from v.

Metagraph

Definition: The metagraph of a directed graph G is a graph whose vertices are the SCCs of G, where there is an edge between C_1 and C_2 if and only if G has an edge between some vertex of C_1 and some vertex of C_2.

Result

Theorem: The metagraph is any directed graph is a DAG.

Computing SCCs

Problem: Given a directed graph G compute the SCCs of G and its metagraph.

Observation

Suppose that SCC(v) is a sink in the metagraph.

- G has no edges from SCC(v) to another SCC.
- Run `explore(v)` to find all vertices reachable from v.
  - Contains all vertices in SCC(v).
  - Contains no other vertices.
- If v in sink SCC, `explore(v)` finds *exactly* v's component.

Today

- Computing SCCs
- Shortest Paths in Graphs (Ch 4)
  - BFS

Strategy

- Find v in a sink SCC of G.
- Run `explore(v)` to find component C_1.
- Repeat process on $G-C_1$.

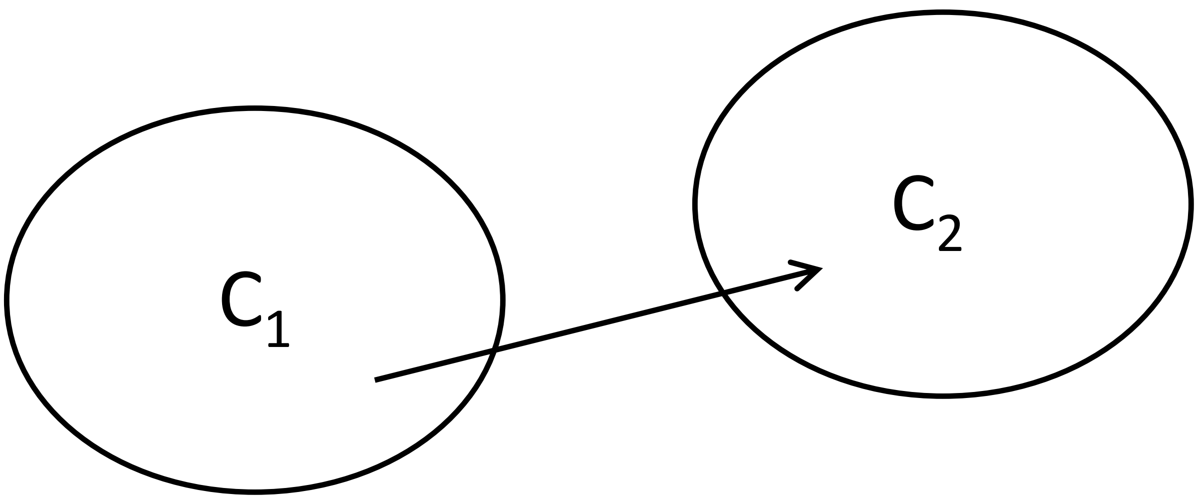
Strategy

- Find v in a sink SCC of G.
- Run `explore(v)` to find component C_1.
- Repeat process on G-C_1.

Problem: How do we find v?

Result

<u>Proposition:</u> Let C_1 and C_2 be SCCs of G with an edge from C_1 to C_2. If we run `DFS` on G, the largest postorder number of any vertex in C_1 will be larger than the largest postorder number in C_2.



Why do we Care?

- Let v be the vertex with the *largest* postorder number.

Why do we Care?

- Let v be the vertex with the *largest* postorder number.
  - There is no edge to SCC(v) from any other SCC.
  - SCC is a <u>source</u> SCC.

Why do we Care?

- Let v be the vertex with the *largest* postorder number.
  - There is no edge to SCC(v) from any other SCC.
  - SCC is a <u>source</u> SCC.
- But we want a *sink* SCC.

Why do we Care?

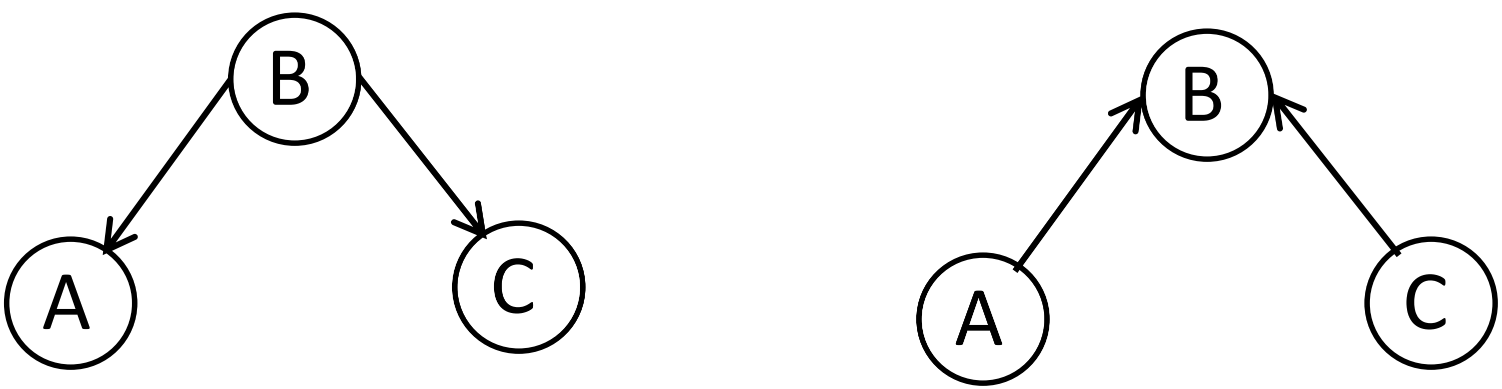
- Let v be the vertex with the *largest* postorder number.
  - There is no edge to SCC(v) from any other SCC.
  - SCC is a <u>source</u> SCC.
- But we want a *sink* SCC.
- A sink is like a source, only with the edges going in the opposite direction.

Reverse Graph

Definition: Given a directed graph G, the reverse graph of G (denoted G^R) is obtained by reversing the directions of all of the edges of G.

Reverse Graph

Definition: Given a directed graph G, the reverse graph of G (denoted G^R) is obtained by reversing the directions of all of the edges of G.



Question: Reverse Graph Properties

Which of the following are NOT true about reverse graphs?

A) G and G^R have the same number of vertices.

B) G and G^R have the same number of edges.

C) $G = (G^R)^R$.

D) A vertex has as many ingoing/outgoing edges in G as it does in G^R.

Question: Reverse Graph Properties

Which of the following are NOT true about reverse graphs?

A) G and G^R have the same number of vertices.

B) G and G^R have the same number of edges.

C) $G = (G^R)^R$.

D) A vertex has as many ingoing/outgoing edges in G as it does in G^R.

Other Properties of Reverse Graphs

Given a directed graph G and its reverse graph G^R:

- G and G^R have the *same* SCCs.
- The sink SCCs of G are the source SCCs of G^R.
- The source SCCs of G are the sink SCCs of G^R.

Other Properties of Reverse Graphs

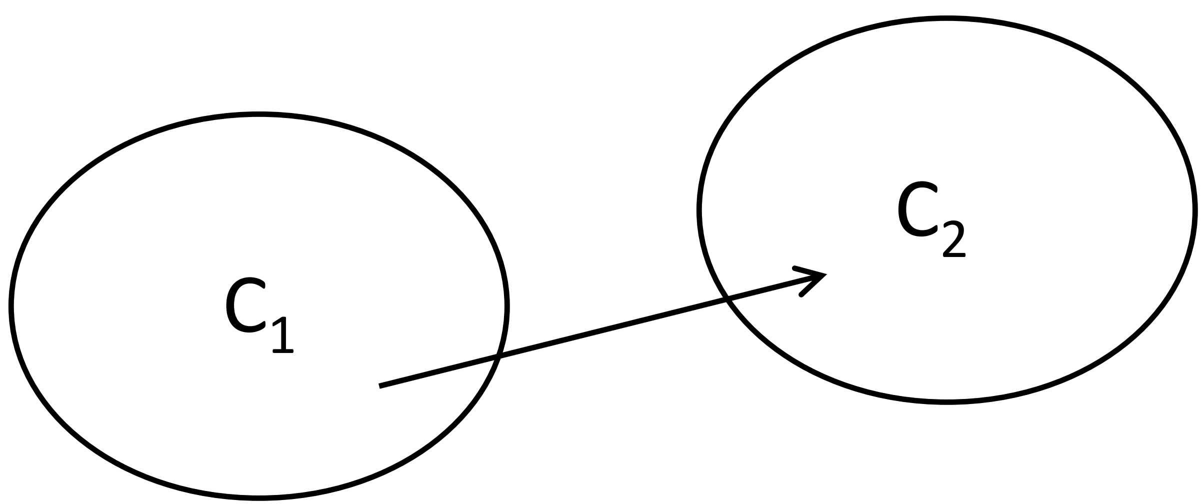
Given a directed graph G and its reverse graph G^R:

- G and G^R have the *same* SCCs.
- The sink SCCs of G are the source SCCs of G^R.
- The source SCCs of G are the sink SCCs of G^R.

So we can find a sink SCC of G, by finding a source SCC of G^R!

Proposition Reminder

Proposition: Let C_1 and C_2 be SCCs of G with an edge from C_1 to C_2. If we run `DFS` on G, the largest postorder number of any vertex in C_1 will be larger than the largest postorder number in C_2.



Proof I

If `DFS` discovers a vertex in C_1 before C_2:

- First vertex discovered is v.

Proof I

If `DFS` discovers a vertex in C_1 before C_2:

- First vertex discovered is v.
- All of C_1 and C_2 downstream of v.

Proof I

If `DFS` discovers a vertex in C_1 before C_2:

- First vertex discovered is v.
- All of C_1 and C_2 downstream of v.
- `DFS` will discover the rest of C_1 and C_2 while exploring v.

Proof I

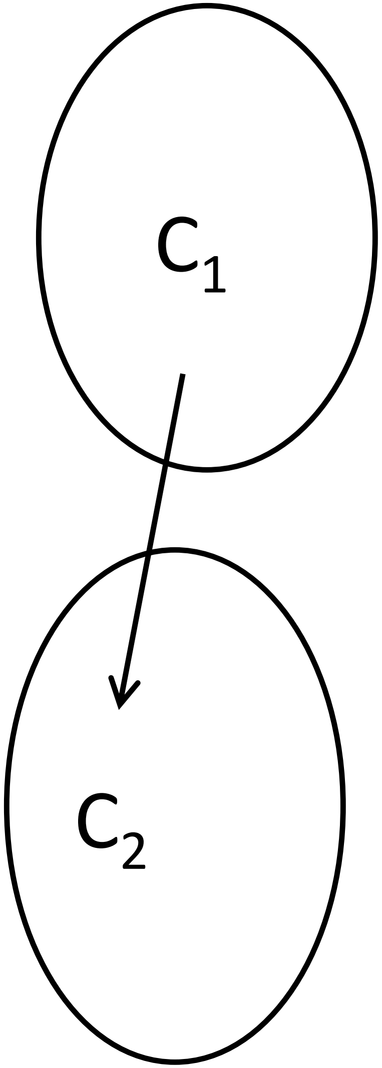
If `DFS` discovers a vertex in C_1 before C_2:

- First vertex discovered is v.
- All of C_1 and C_2 downstream of v.
- `DFS` will discover the rest of C_1 and C_2 while exploring v.
- v has largest postorder in C_1 or C_2.

Proof II

If `DFS` discovers a vertex in C_2 before C_1:

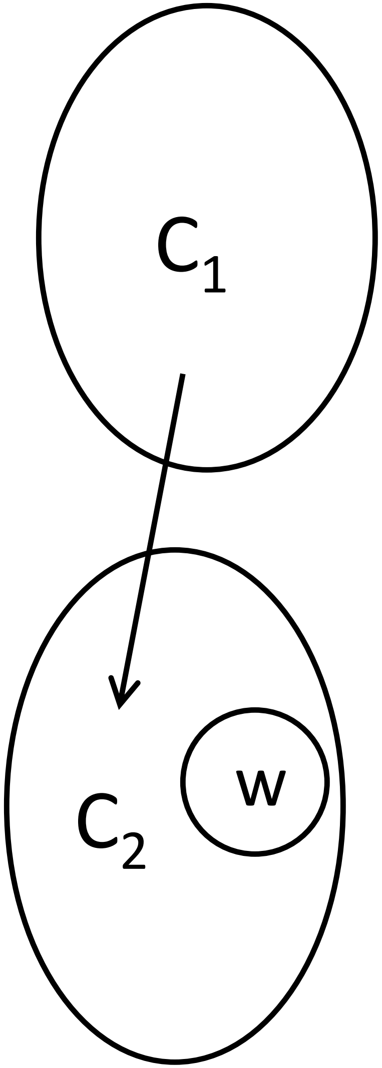
- First vertex discovered is w.



Proof II

If `DFS` discovers a vertex in C_2 before C_1:

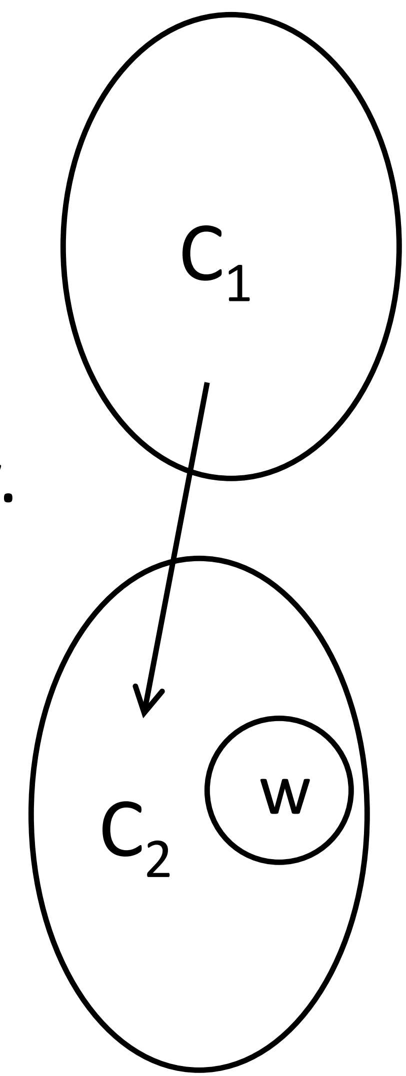
- First vertex discovered is w.



Proof II

If `DFS` discovers a vertex in C_2 before C_1:

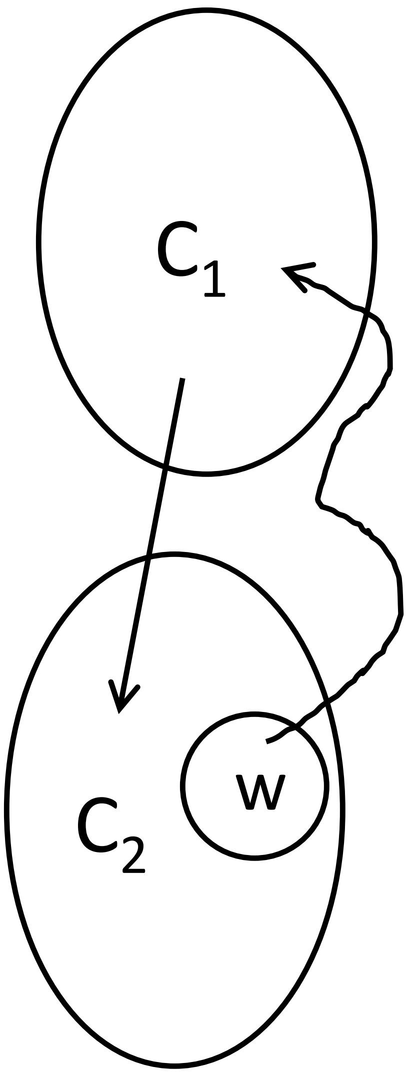
- First vertex discovered is w.
- `DFS` will find all of C_2 while exploring w.



Proof II

If `DFS` discovers a vertex in C_2 before C_1:

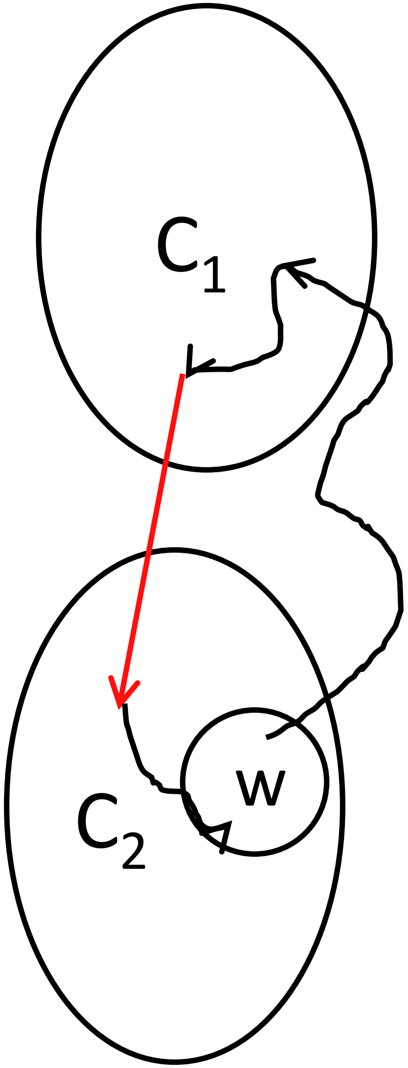
- First vertex discovered is w.
- `DFS` will find all of C_2 while exploring w.
- C_1 cannot be reached from w.



Proof II

If `DFS` discovers a vertex in C_2 before C_1:

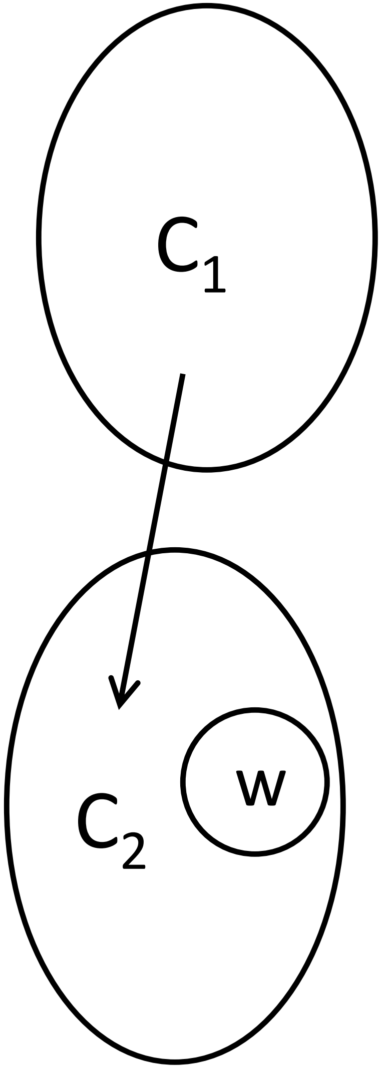
- First vertex discovered is w.
- `DFS` will find all of C_2 while exploring w.
- C_1 cannot be reached from w.
  - Otherwise they'd be the same SCC.



Proof II

If `DFS` discovers a vertex in C_2 before C_1:

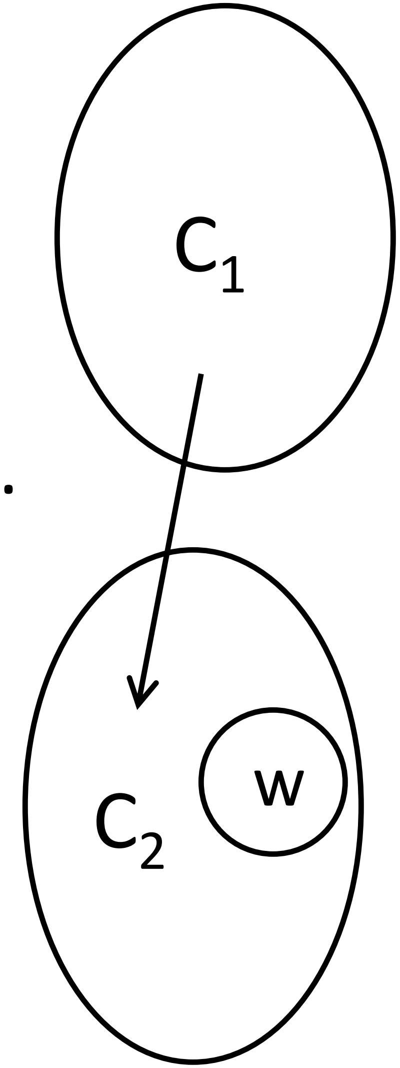
- First vertex discovered is w.
- `DFS` will find all of C_2 while exploring w.
- C_1 cannot be reached from w.
  - Otherwise they'd be the same SCC.
- Every vertex in C_1 discovered after w finished.



Proof II

If `DFS` discovers a vertex in C_2 before C_1:

- First vertex discovered is w.
- `DFS` will find all of C_2 while exploring w.
- C_1 cannot be reached from w.
  - Otherwise they'd be the same SCC.
- Every vertex in C_1 discovered after w finished.
- C_1 has larger posts than C_2.



Algorithm

```
SCCs(G)
  Run DFS(G^R) record postorder
  Find v with largest v.post
  Set all vertices unvisited
  Run explore(v)
  Let C be the visited vertices
  Return SCCs(G-C) U {C}
```

Algorithm

```
SCCs(G)                                O(|V|+|E|)
  Run DFS(G^R) record postorder
  Find v with largest v.post
  Set all vertices unvisited
  Run explore(v)
  Let C be the visited vertices
  Return SCCs(G-C) U {C}
```

Algorithm

```
SCCs(G)                                    O(|V|+|E|)
  Run DFS(G^R) record postorder
  Find v with largest v.post
  Set all vertices unvisited
  Run explore(v)
  Let C be the visited vertices
  Return SCCs(G-C) U {C}
```

Final Runtime: $O((|V|+|E|)(\#SCCs))$

Still Too Slow

Problem: We recompute the postorder for every SCC we need to find.

Still Too Slow

Problem: We recompute the postorder for every SCC we need to find.

Solution: We don't have to do this. After removing some SCCs to get G', the largest postorder number of vertices in G' is *still* in a sink component of G'.

Algorithm II

```
SCCs(G)
  Run DFS(G^R) record postorders
  Mark all vertices unvisited
  For v ∈ V in reverse postorder

    If v not in a component yet
      explore(v) on
      G-components found,
      marking new component
```

Algorithm II

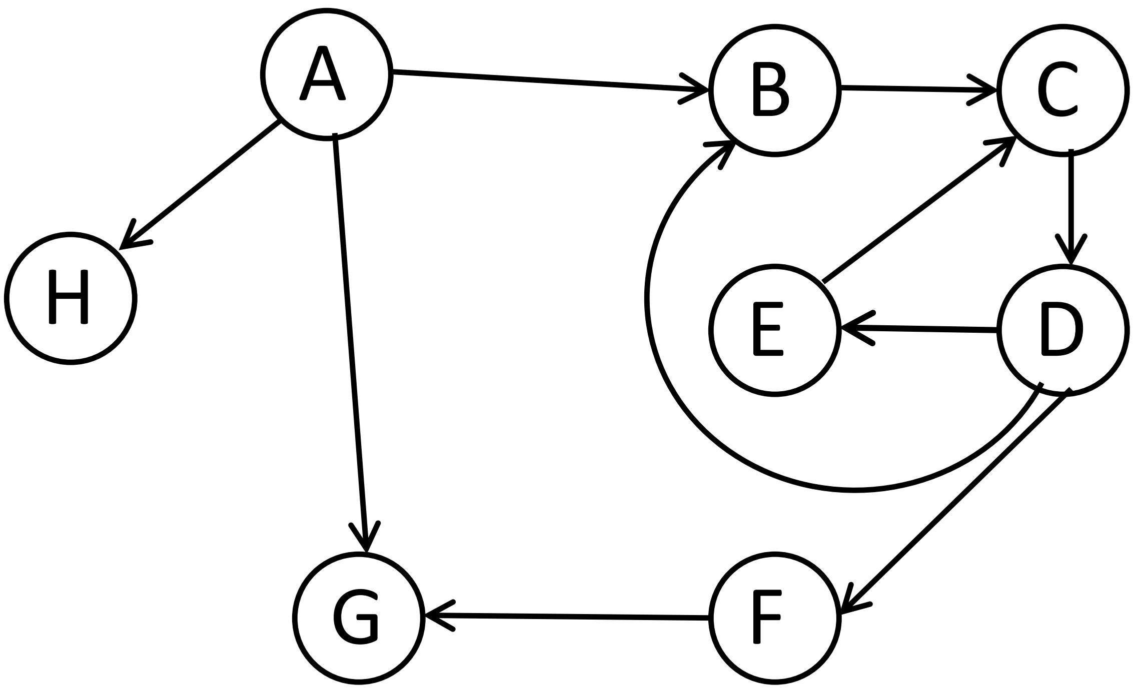
```
SCCs(G)
  Run DFS(G^R) record postorders
  Mark all vertices unvisited
  For v ∈ V in reverse postorder

    If not v.visited
      explore(v) mark component
```

Algorithm II

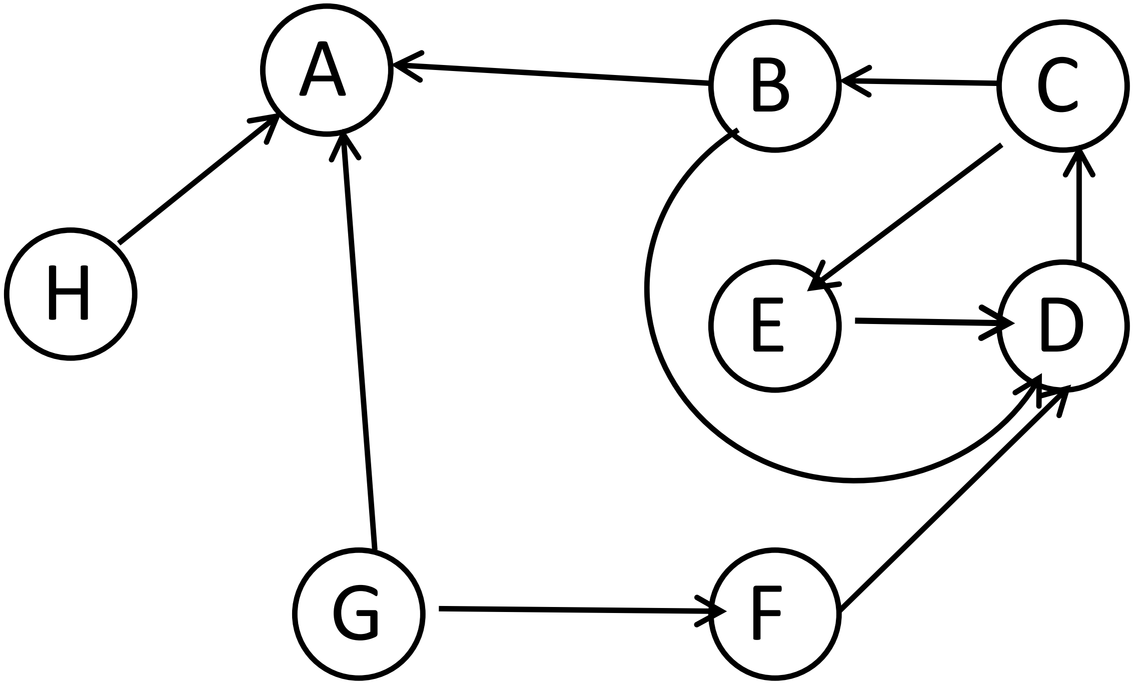
```
SCCs(G)
  Run DFS(GR) record postorders
  Mark all vertices unvisited
  For v ∈ V in reverse postorder

    If not v.visited
      explore(v) mark component
```

Just 2 DFSs! Runtime $O(|V|+|E|)$.

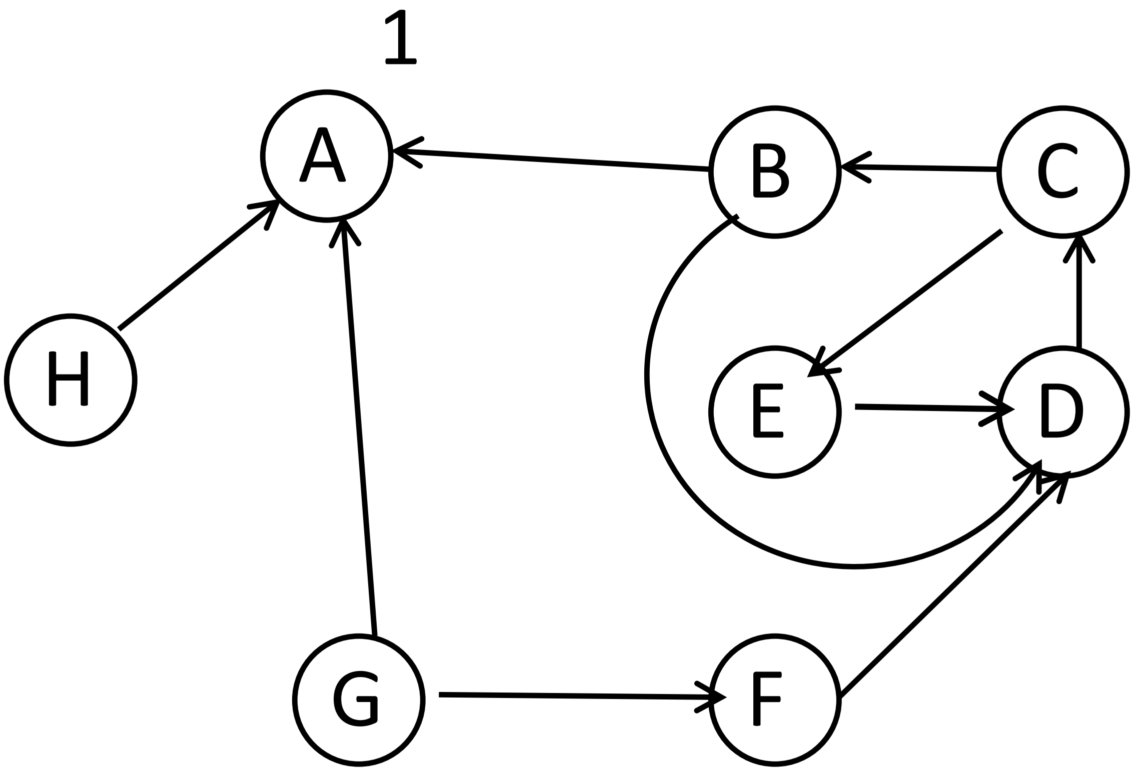
Example



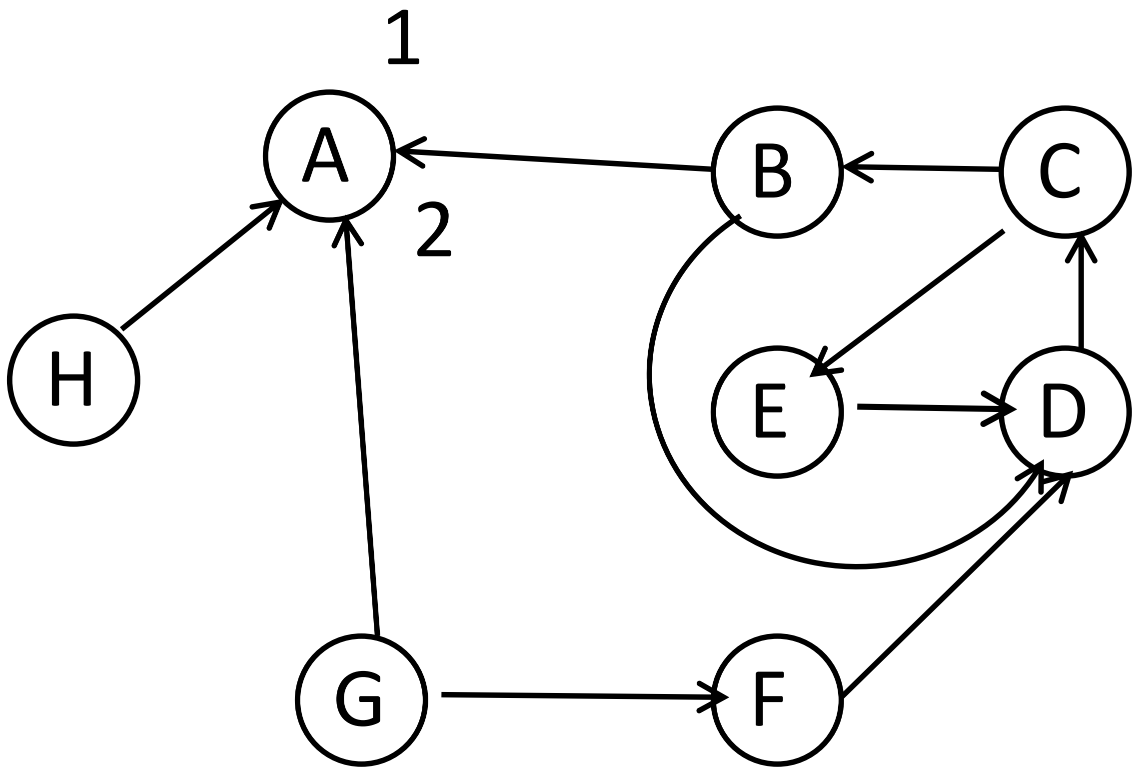
Example



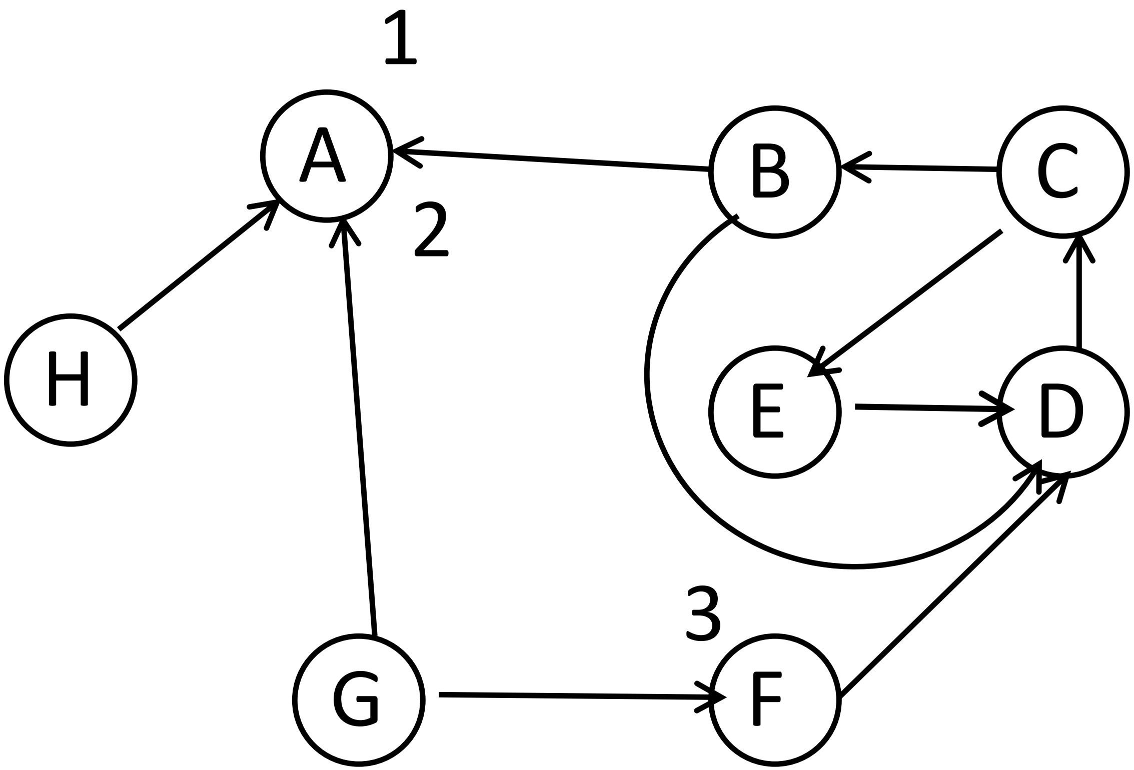
Example



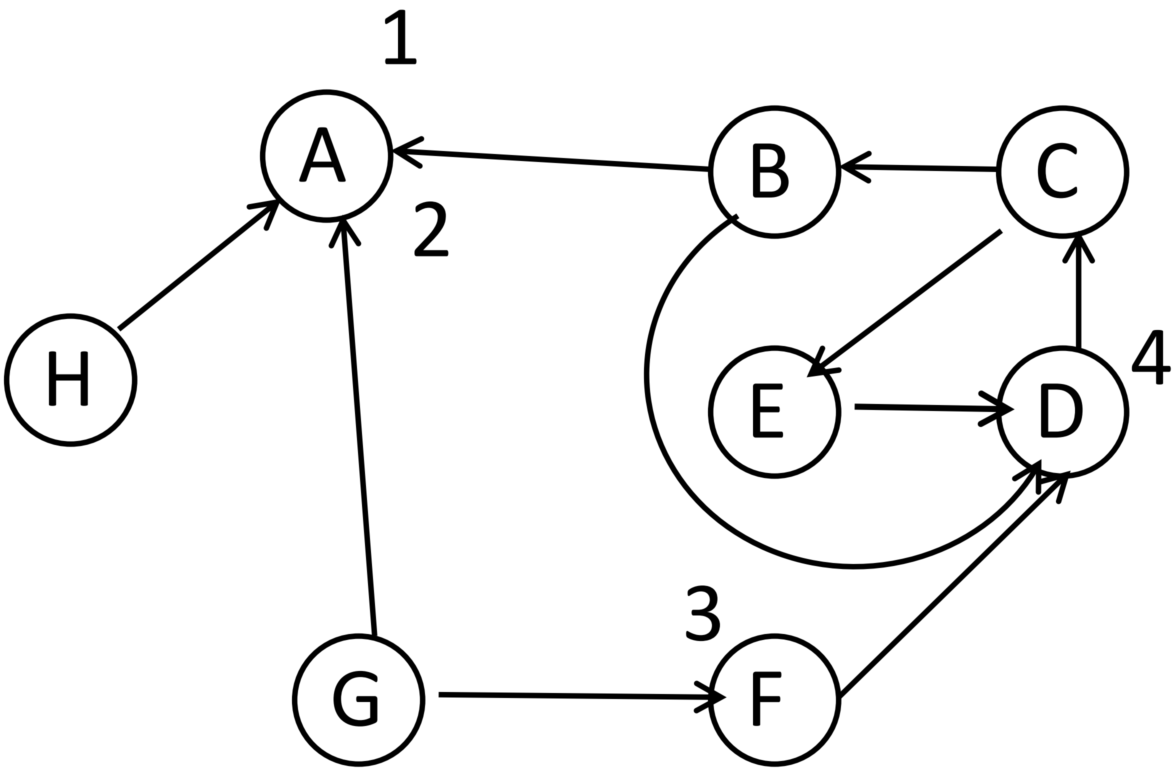
Example



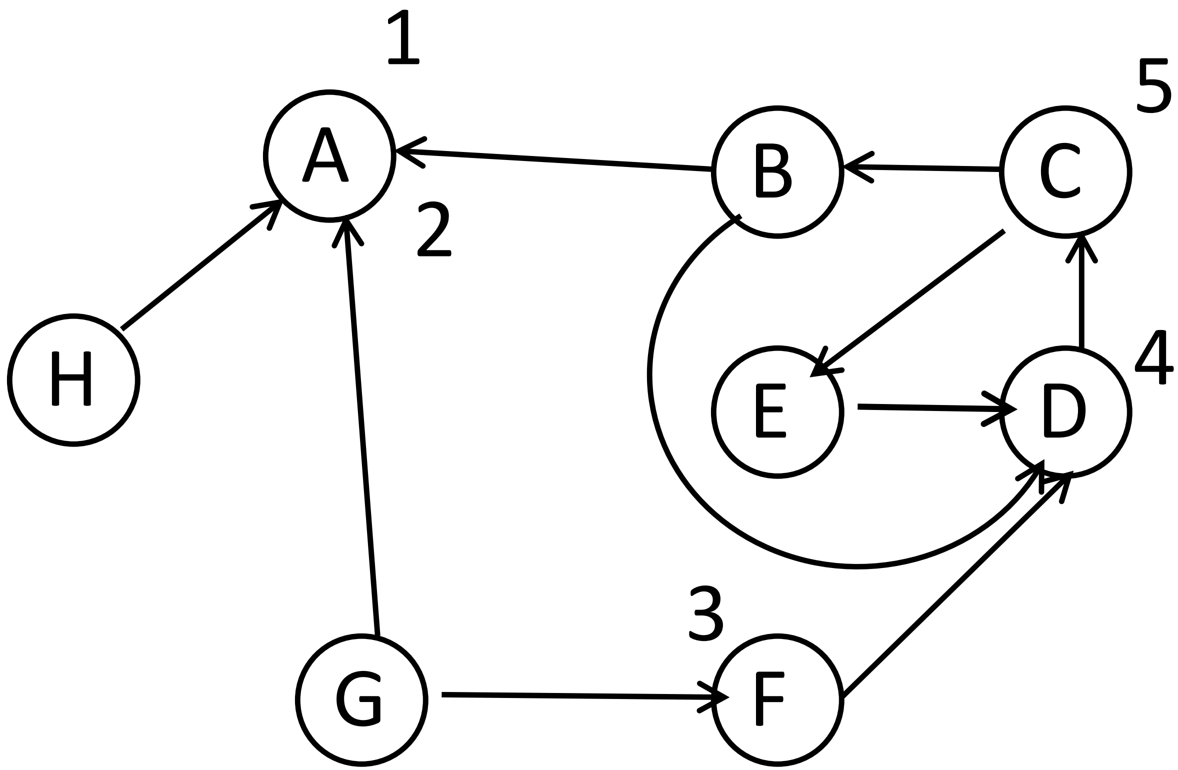
Example



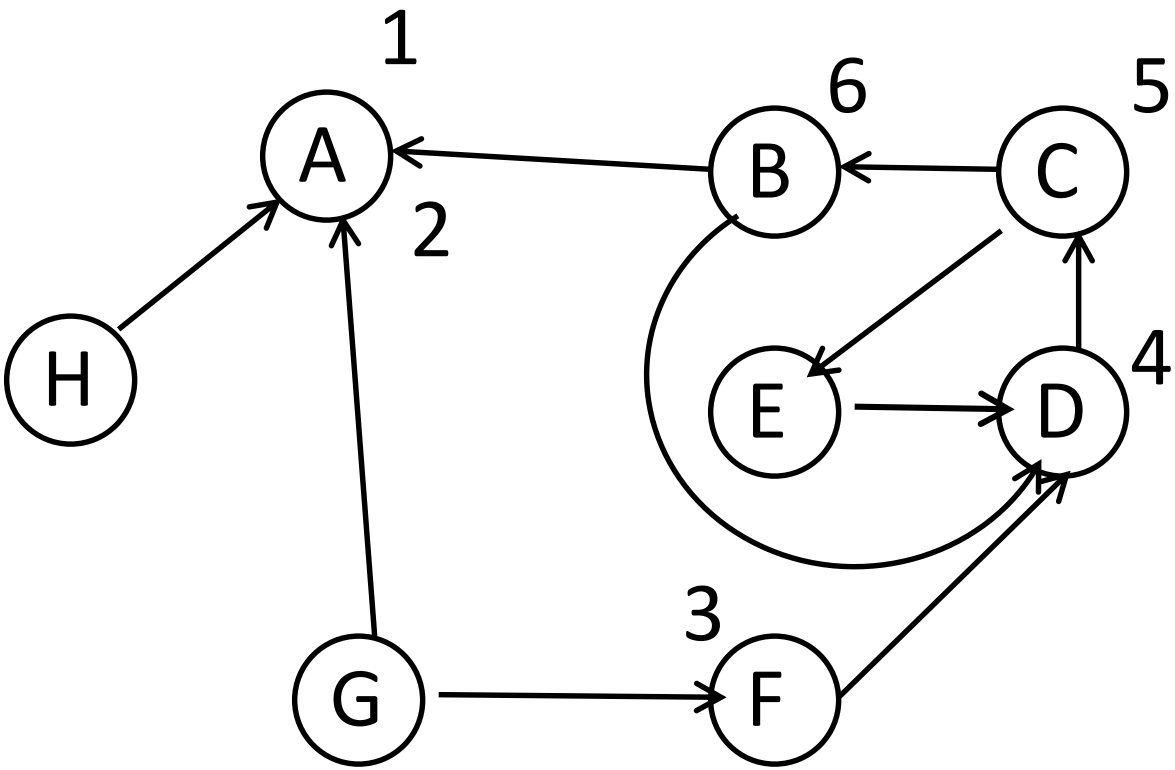
Example



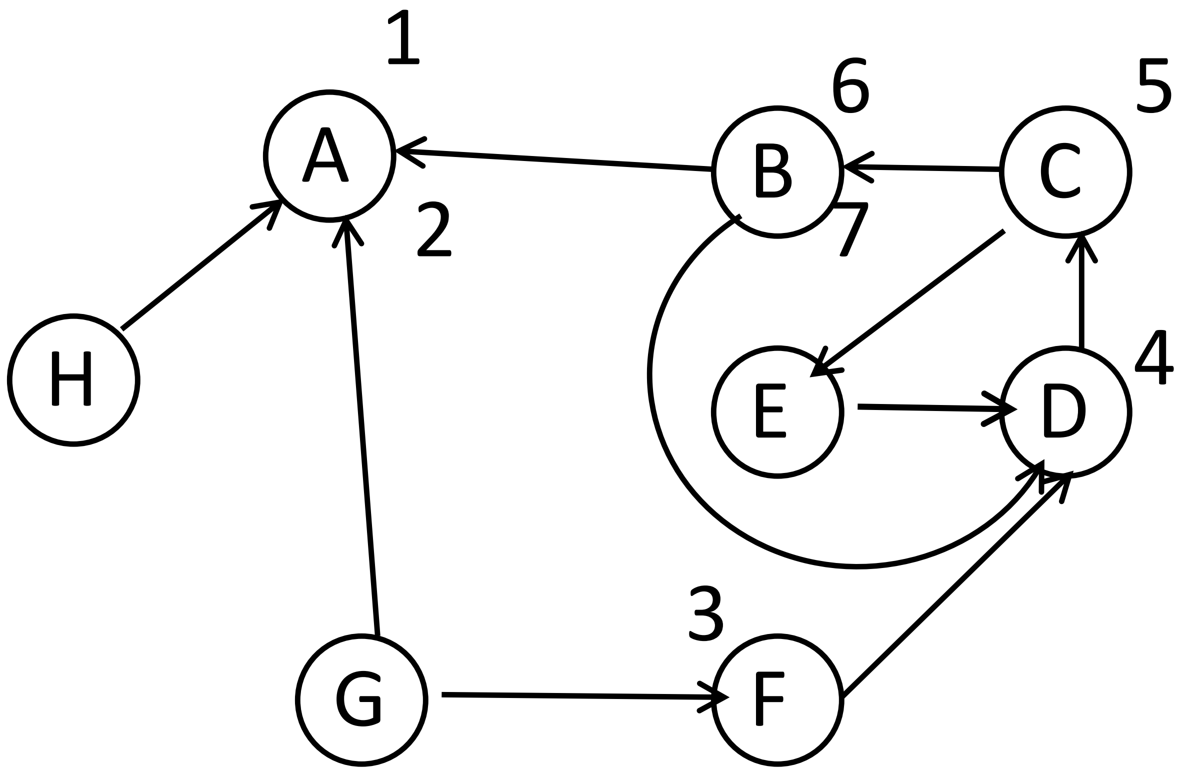
Example



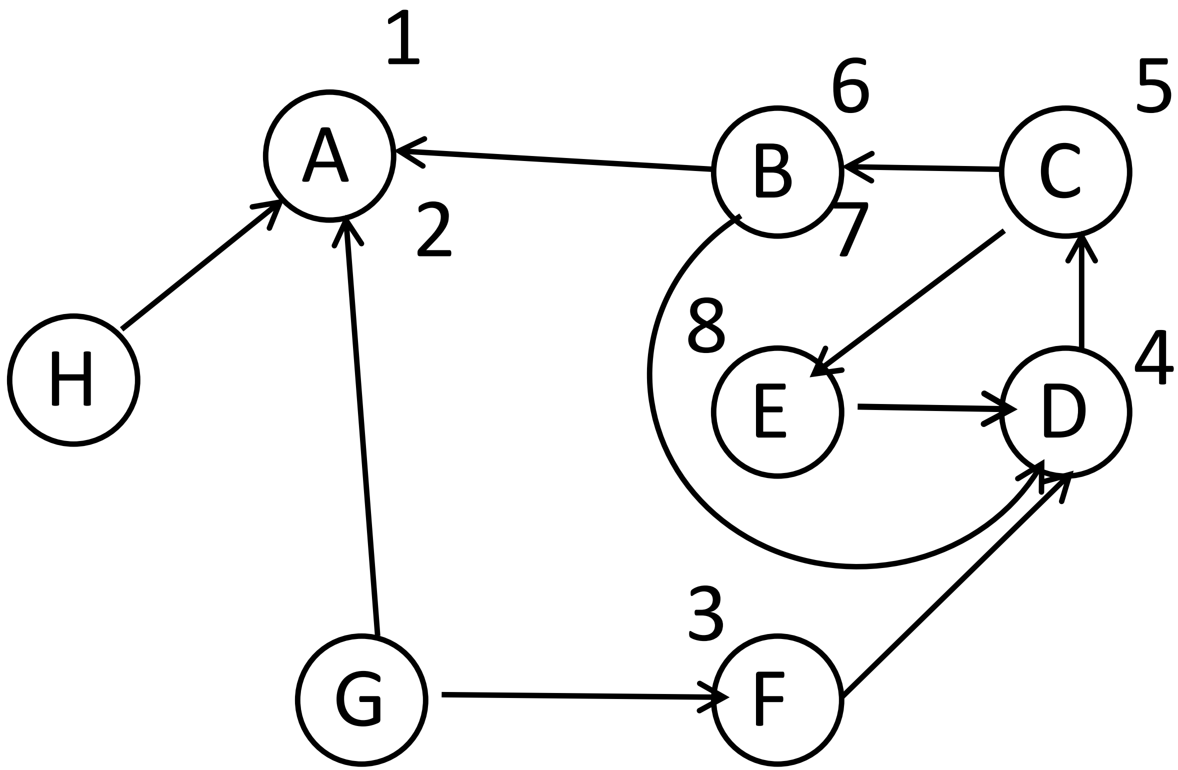
Example



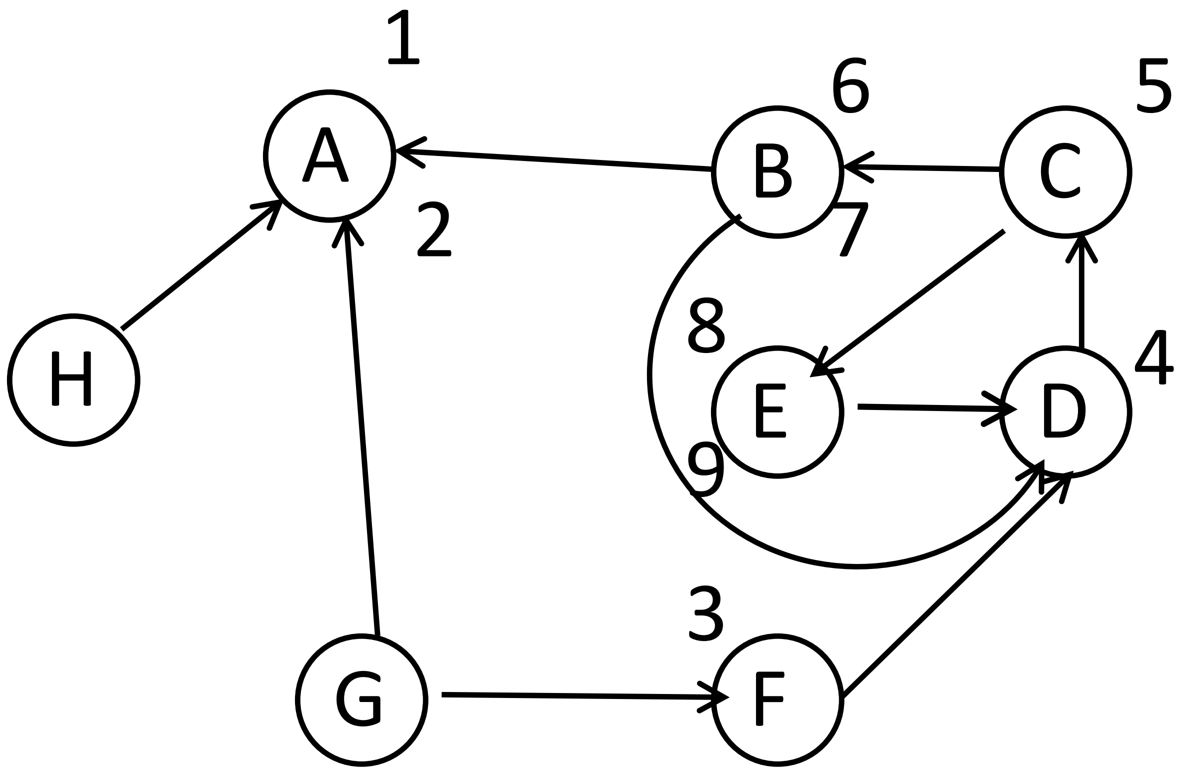
Example



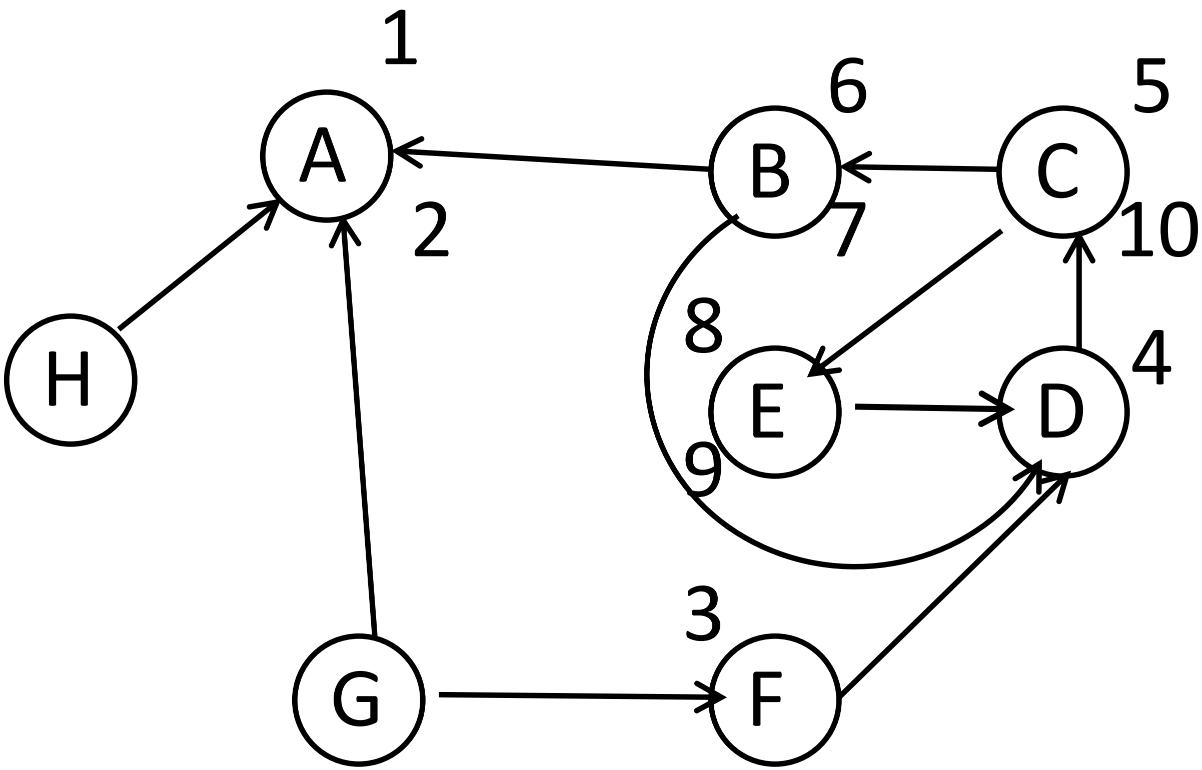
Example



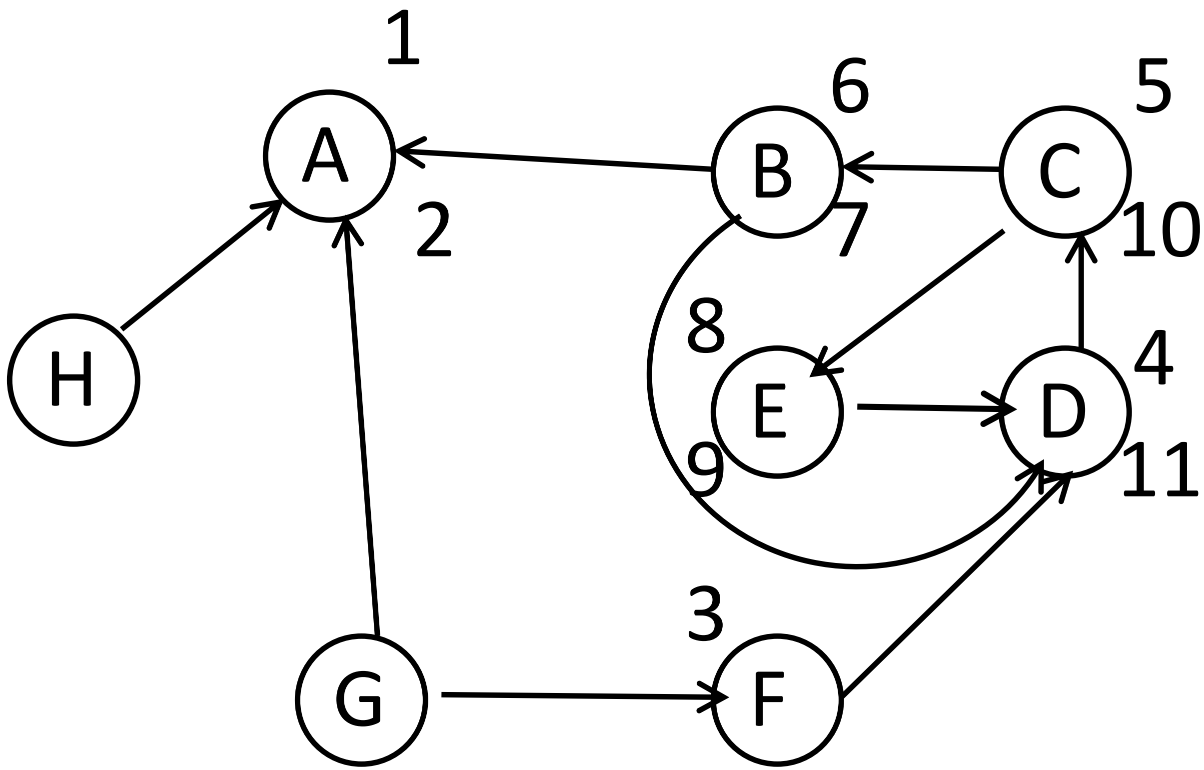
Example



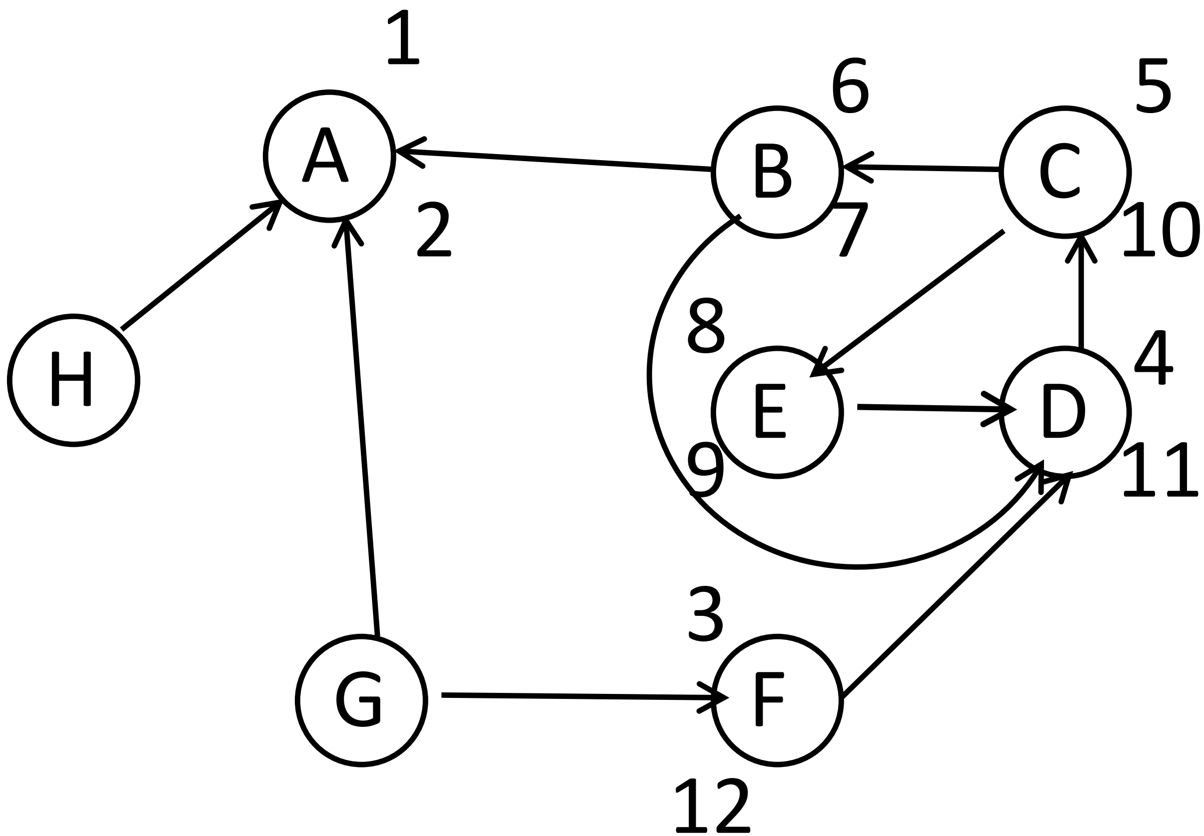
Example



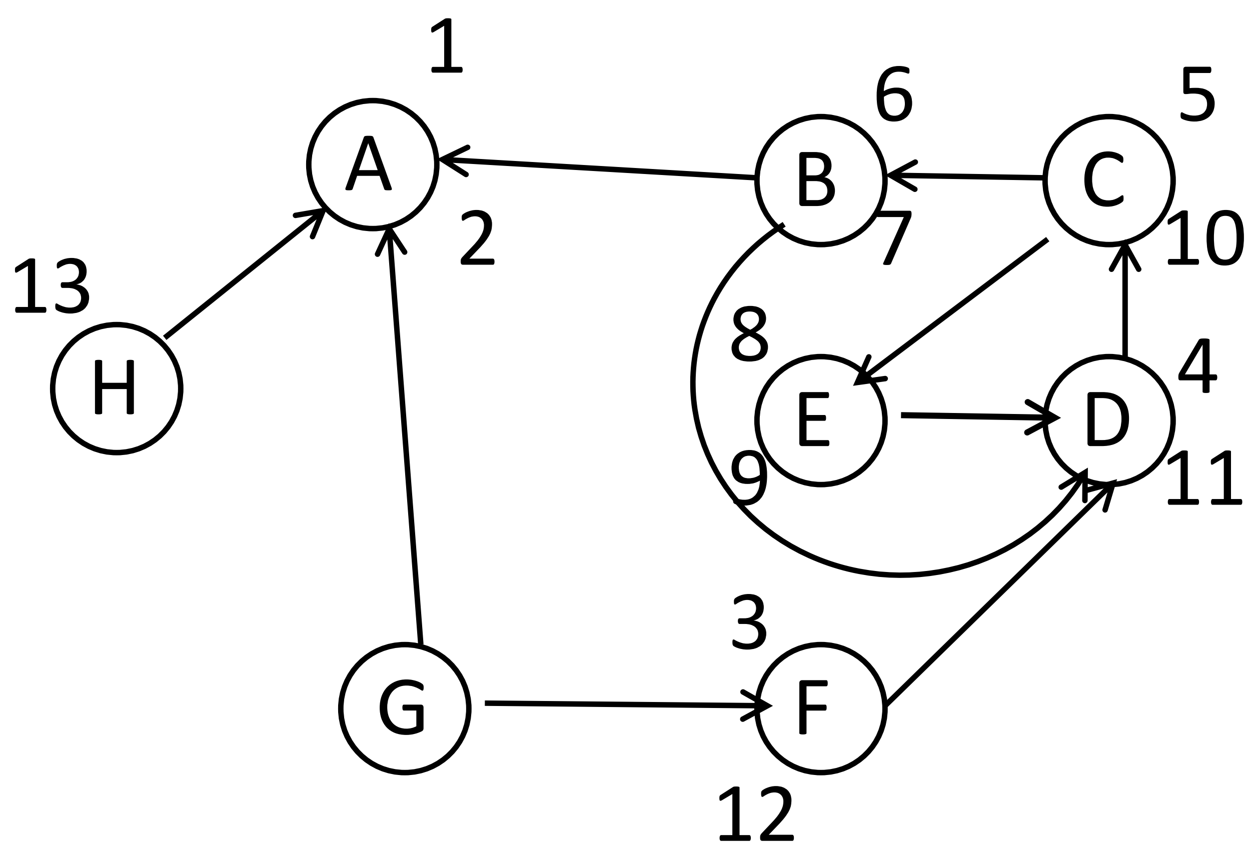
Example



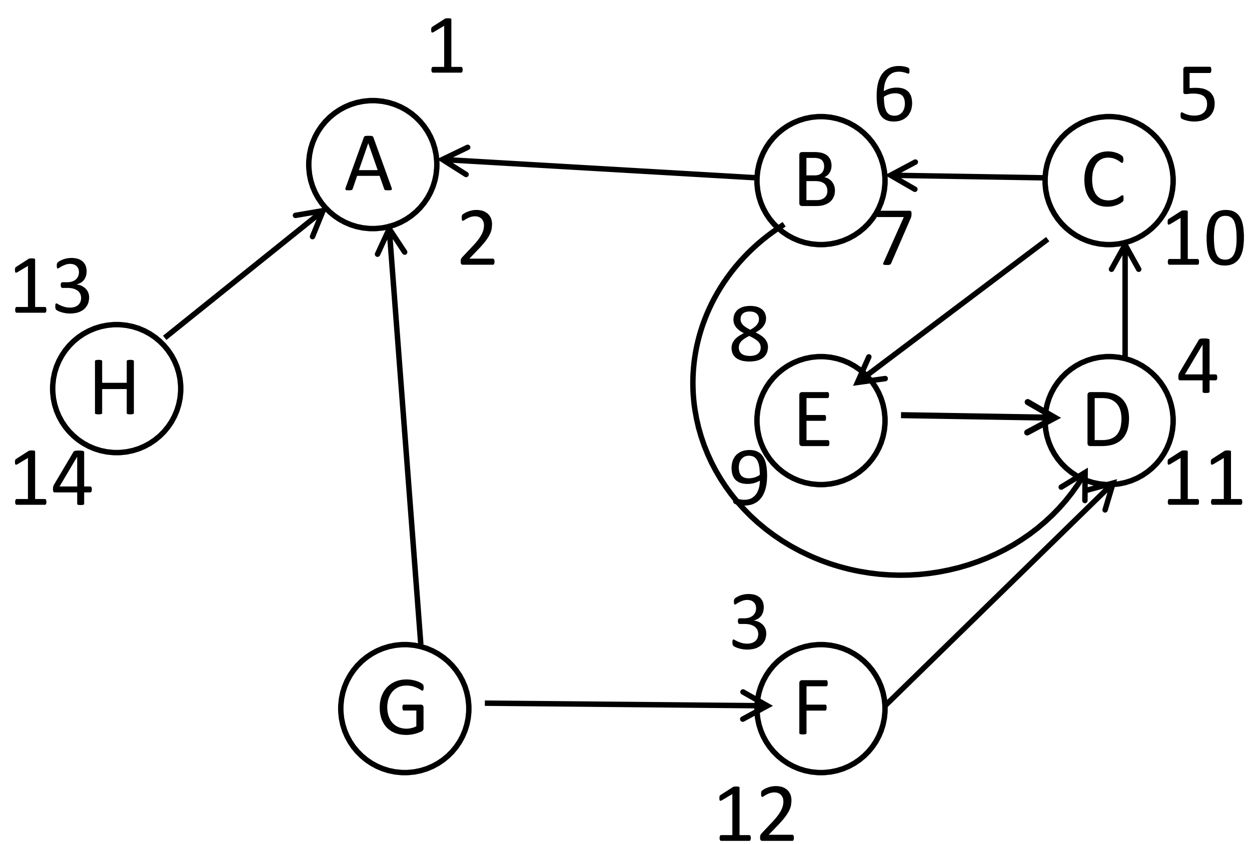
Example



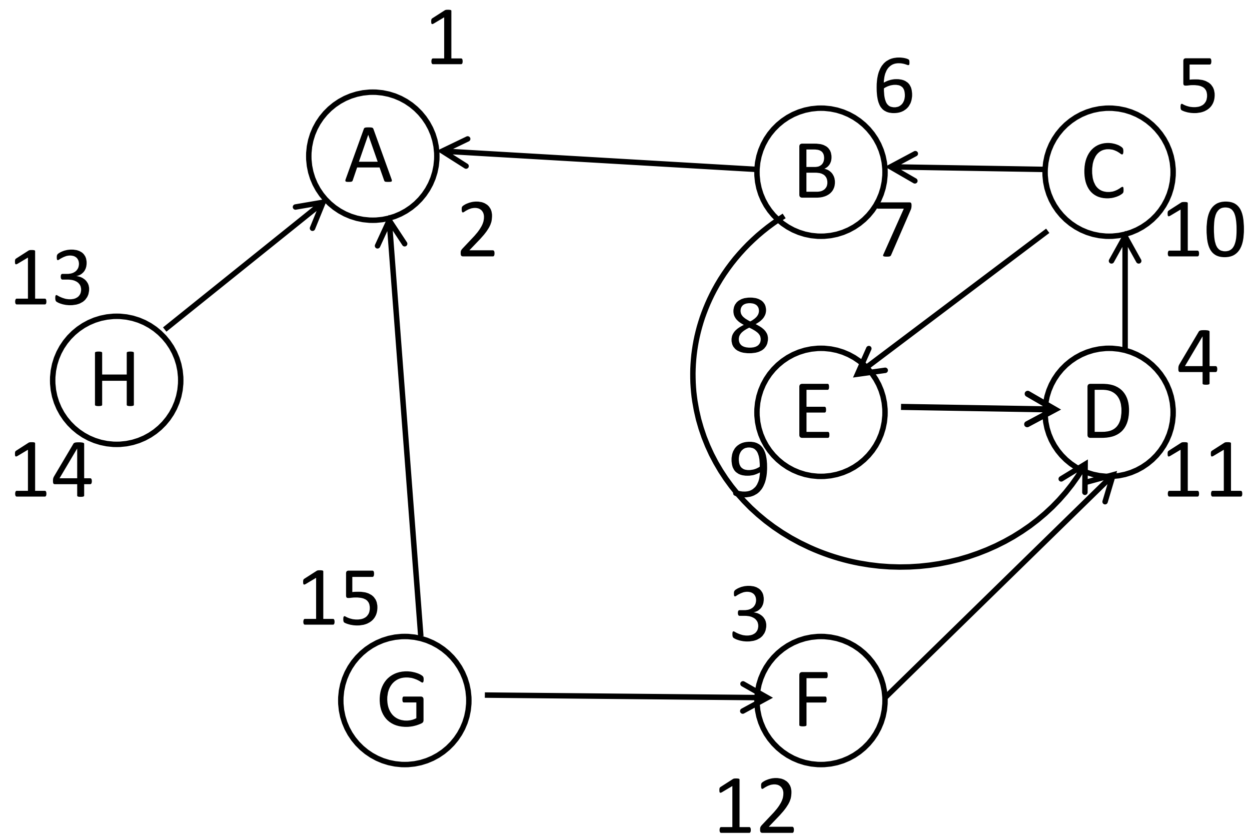
Example



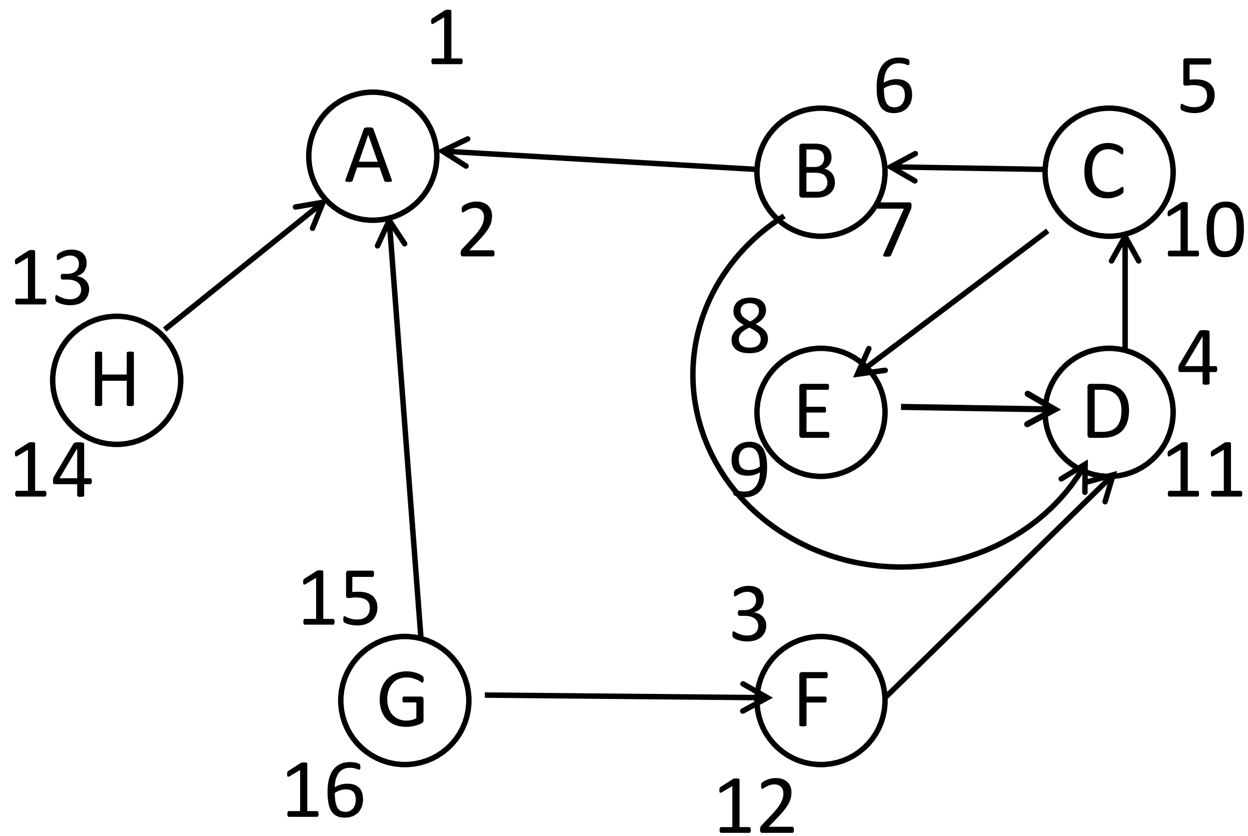
Example



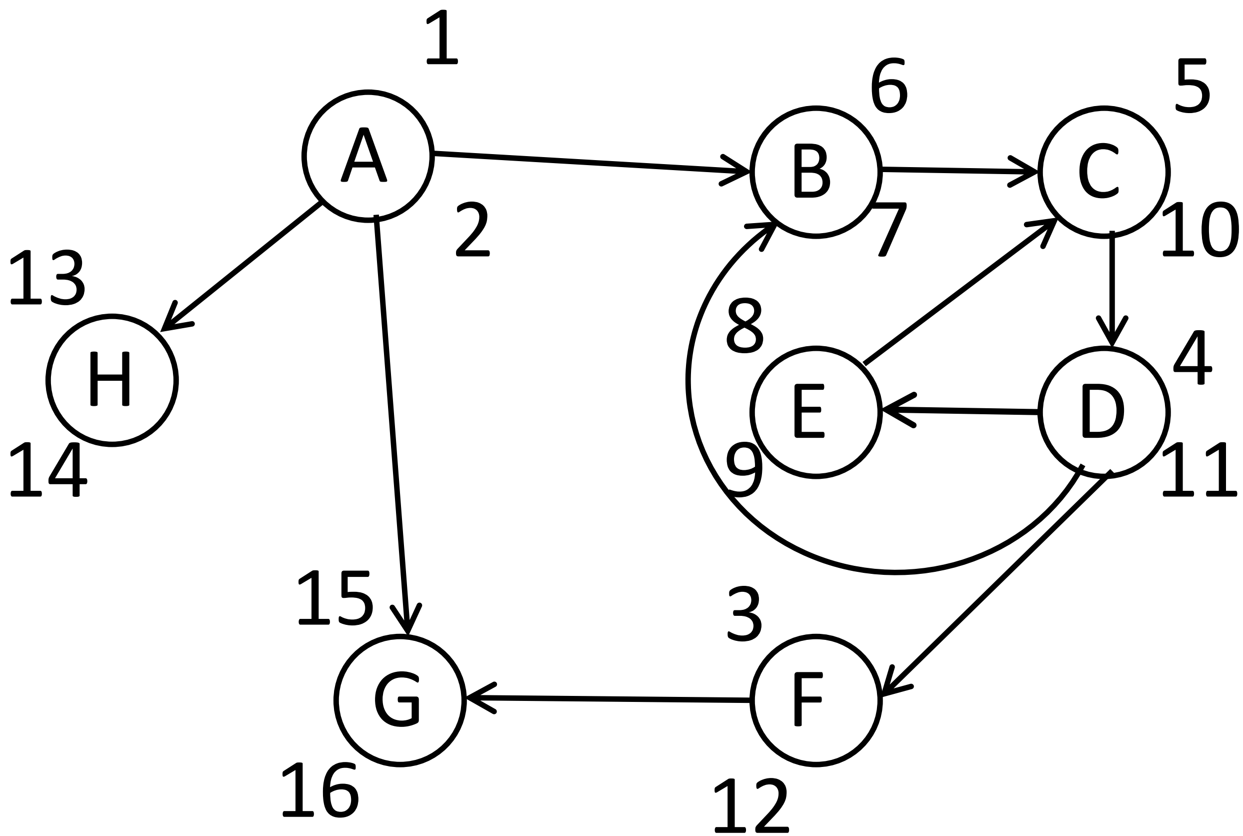
Example



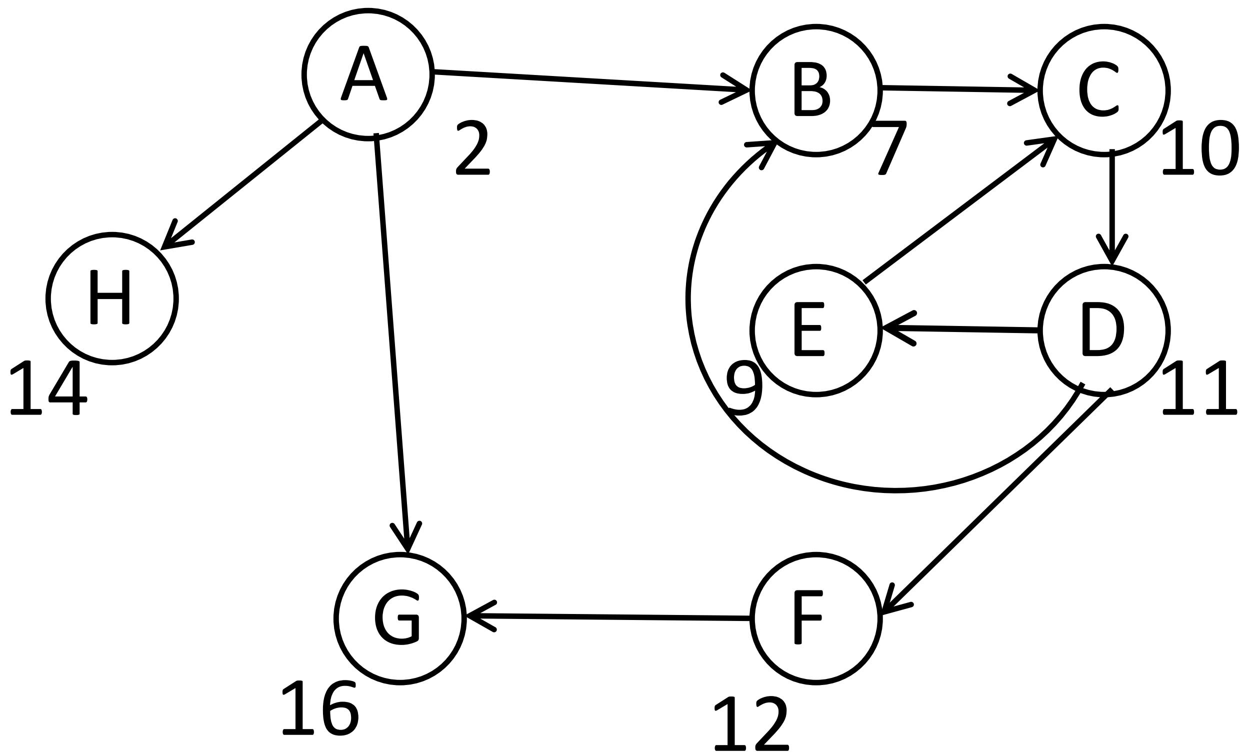
Example



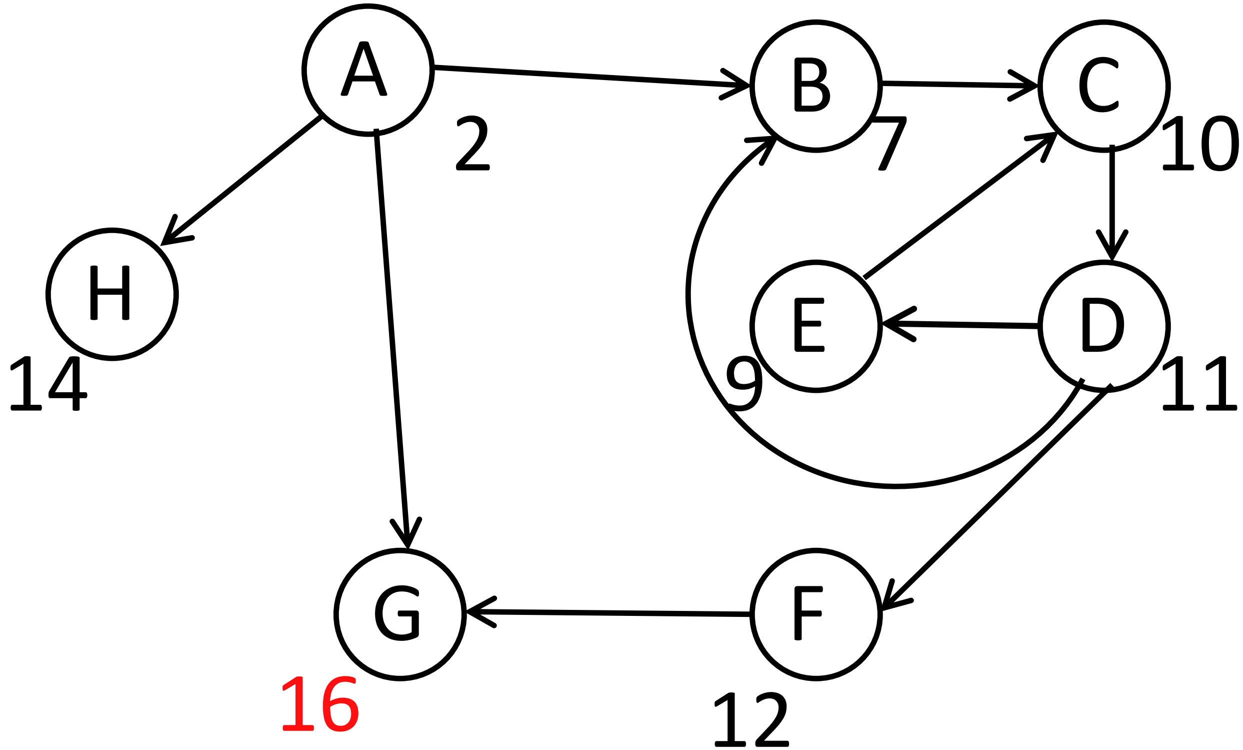
Example



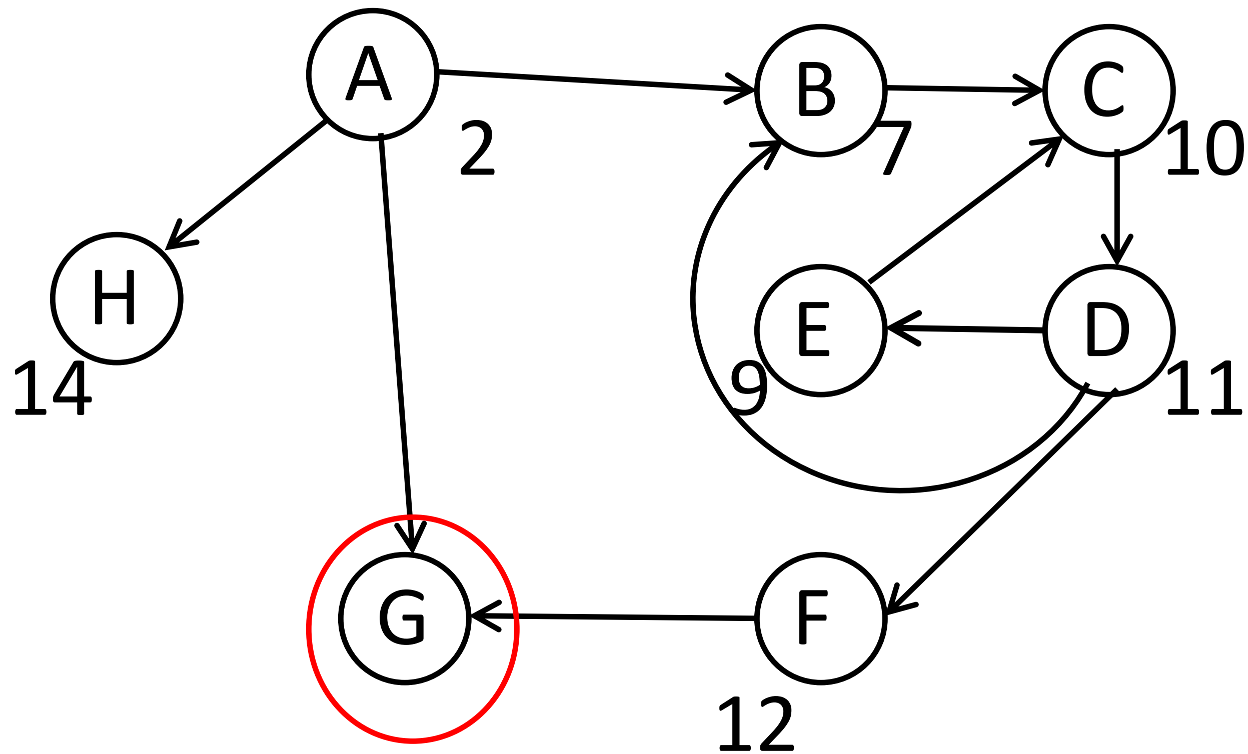
Example



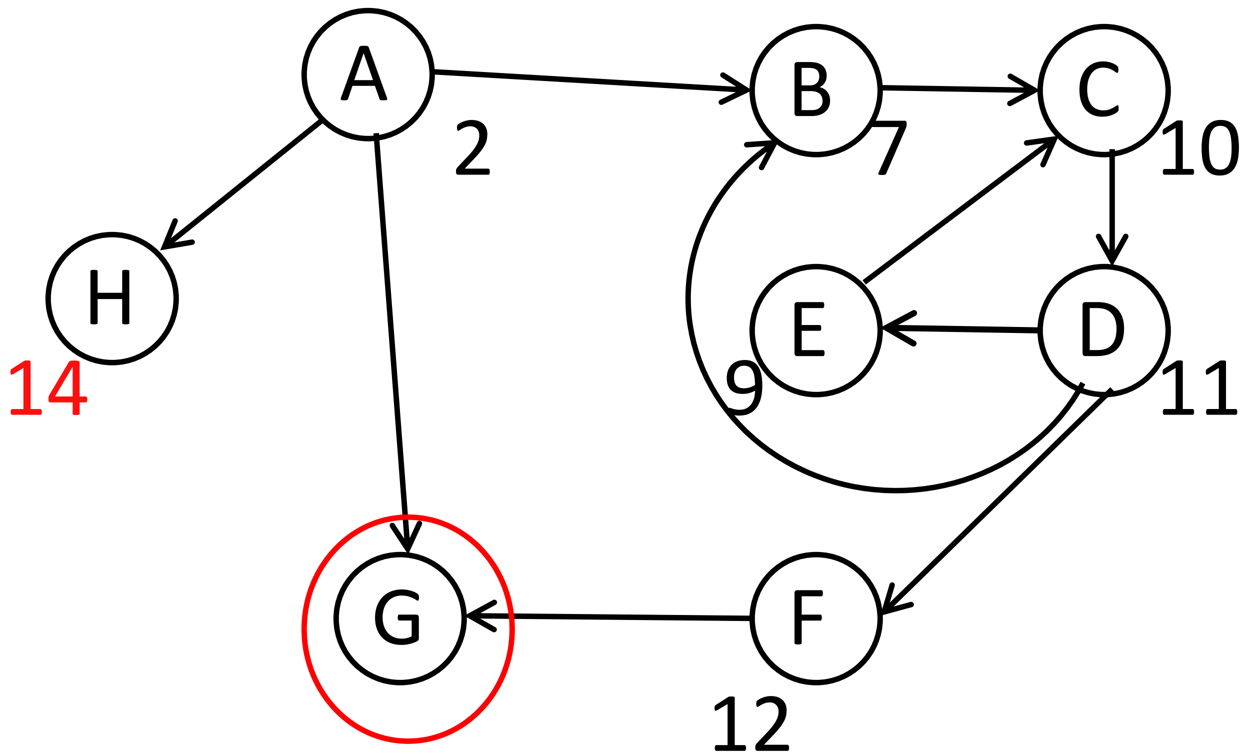
Example



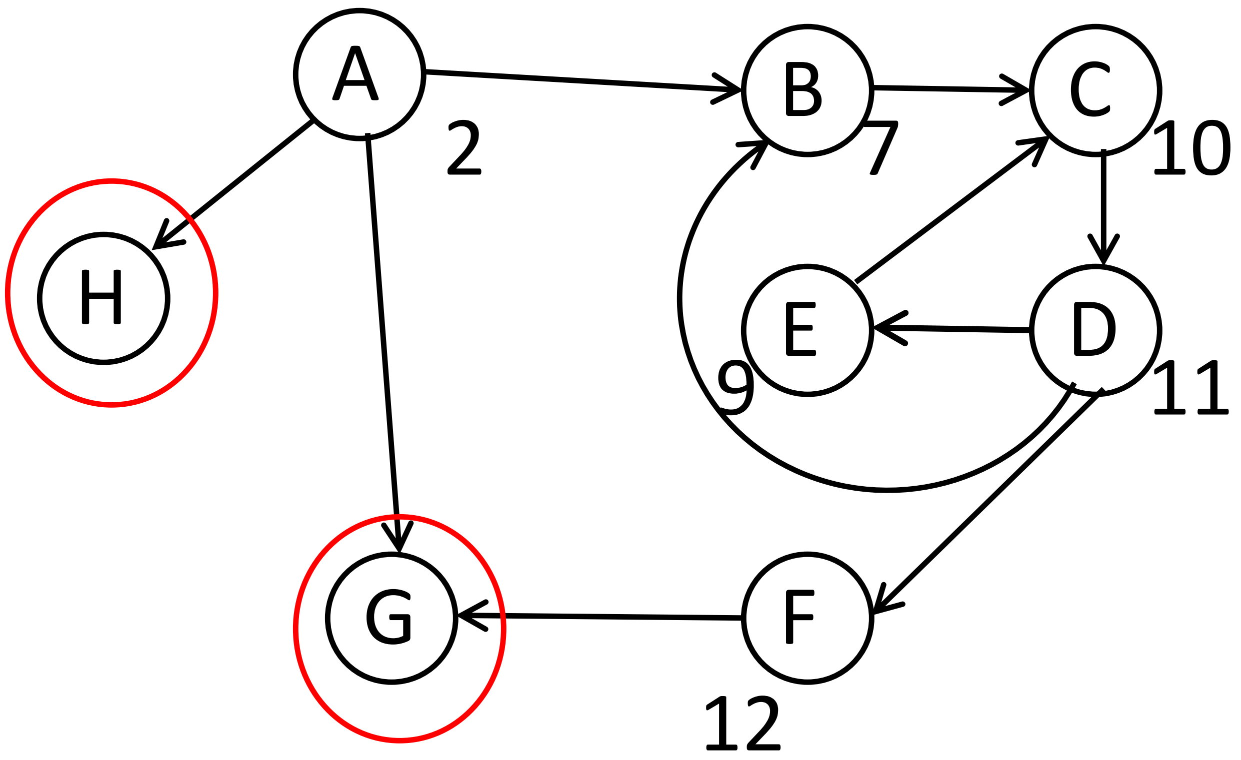
Example



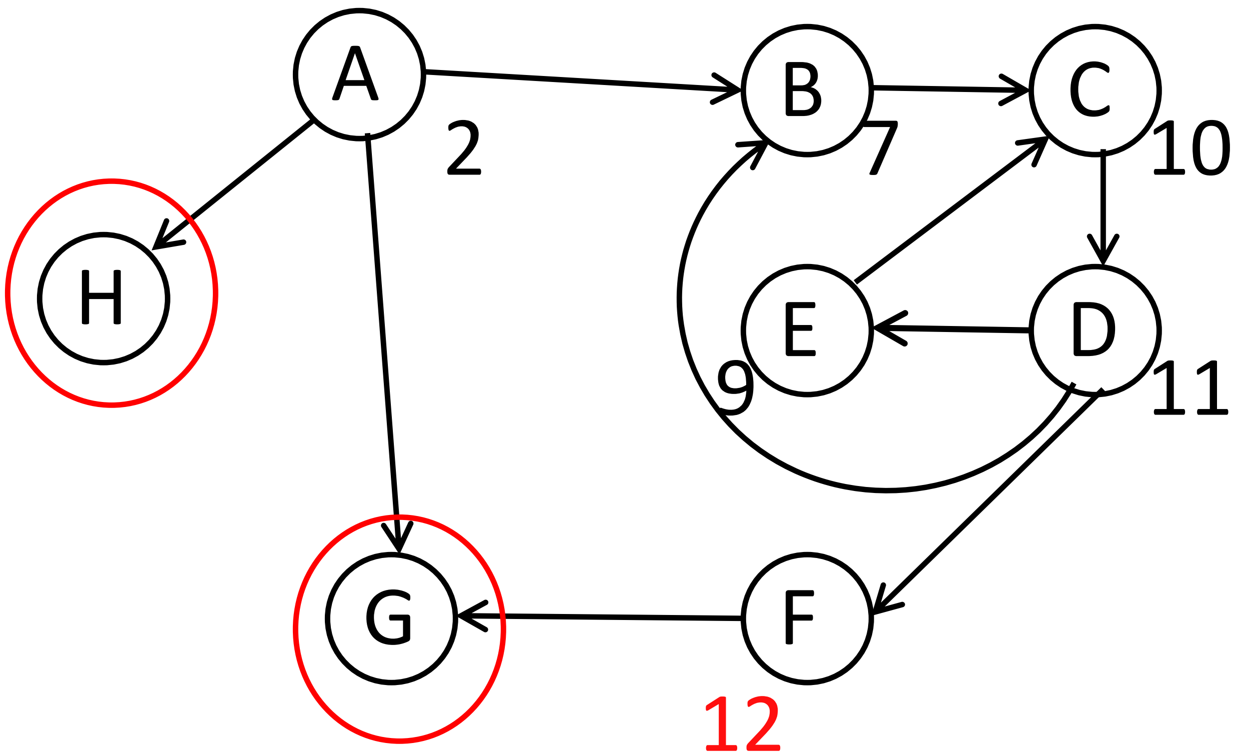
Example



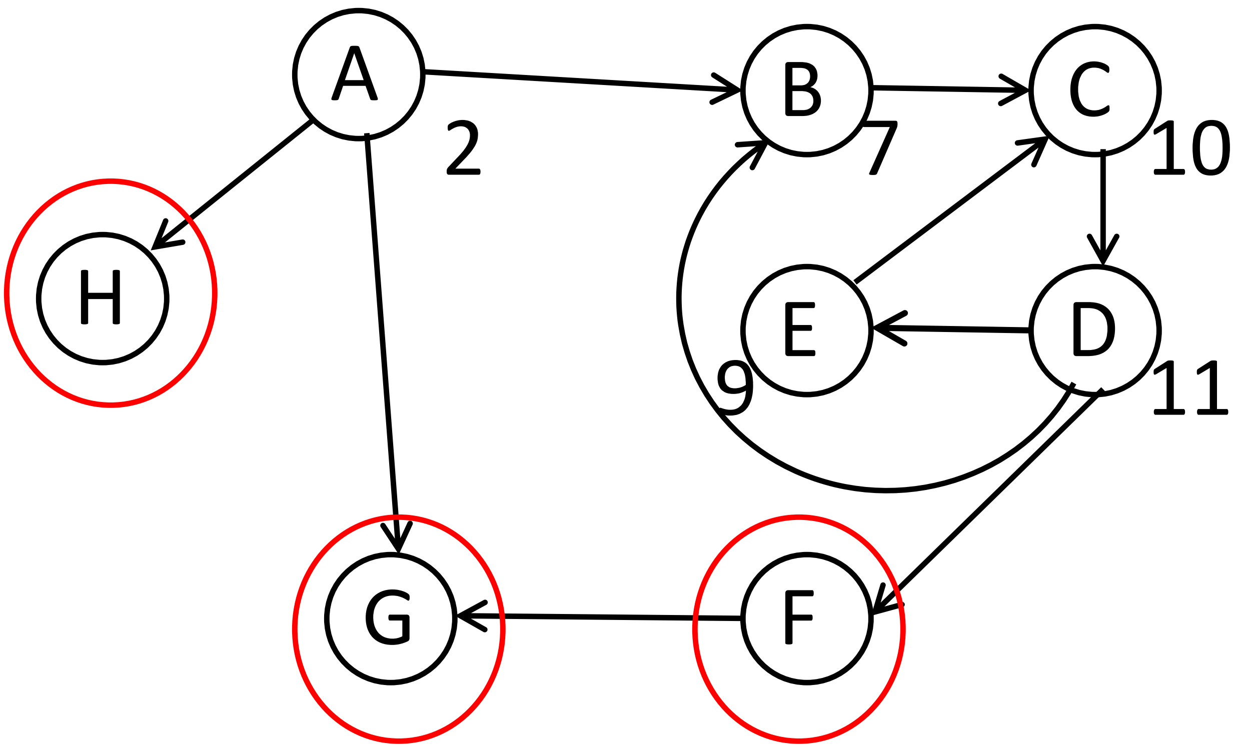
Example



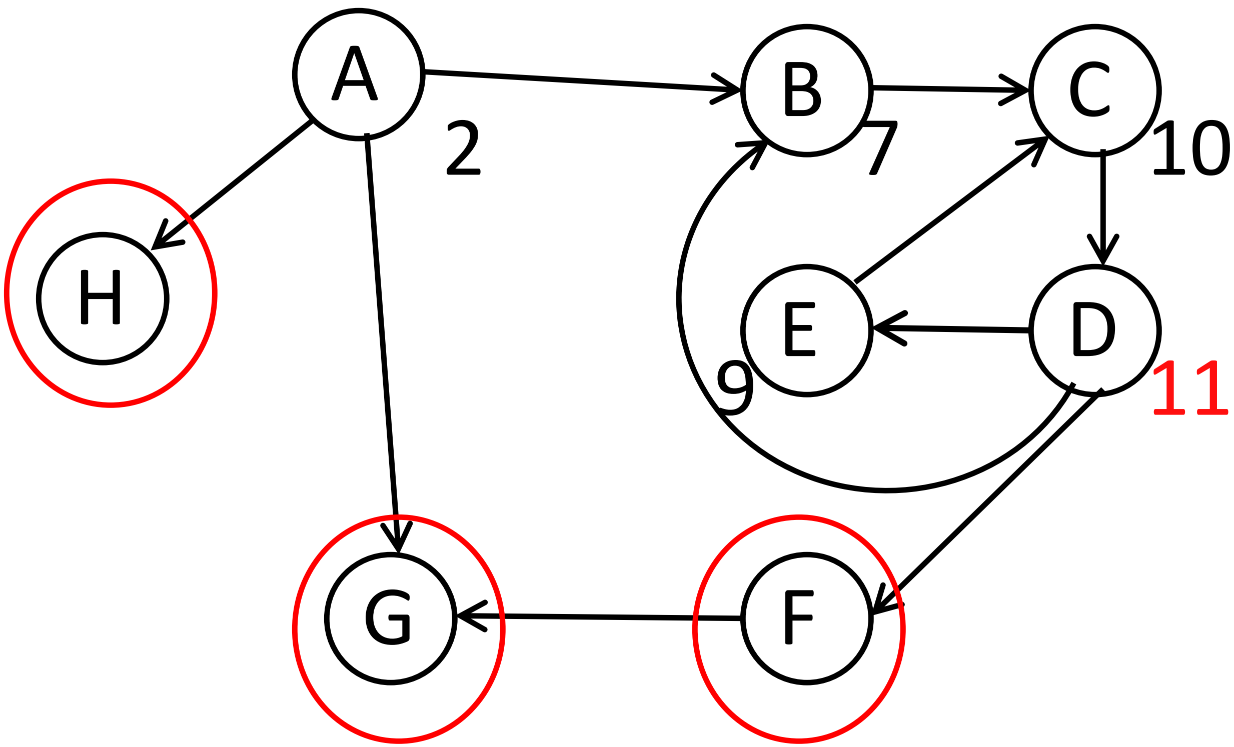
Example



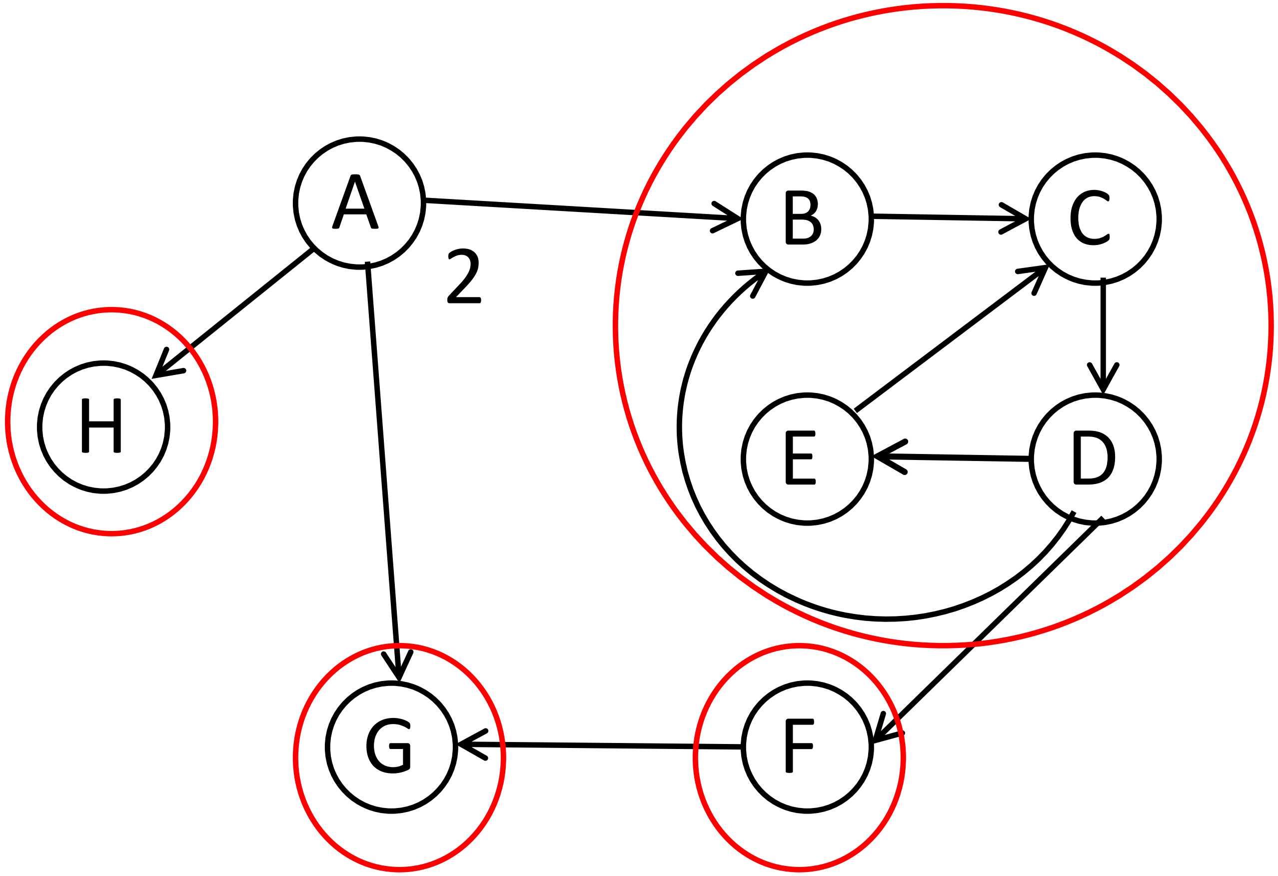
Example



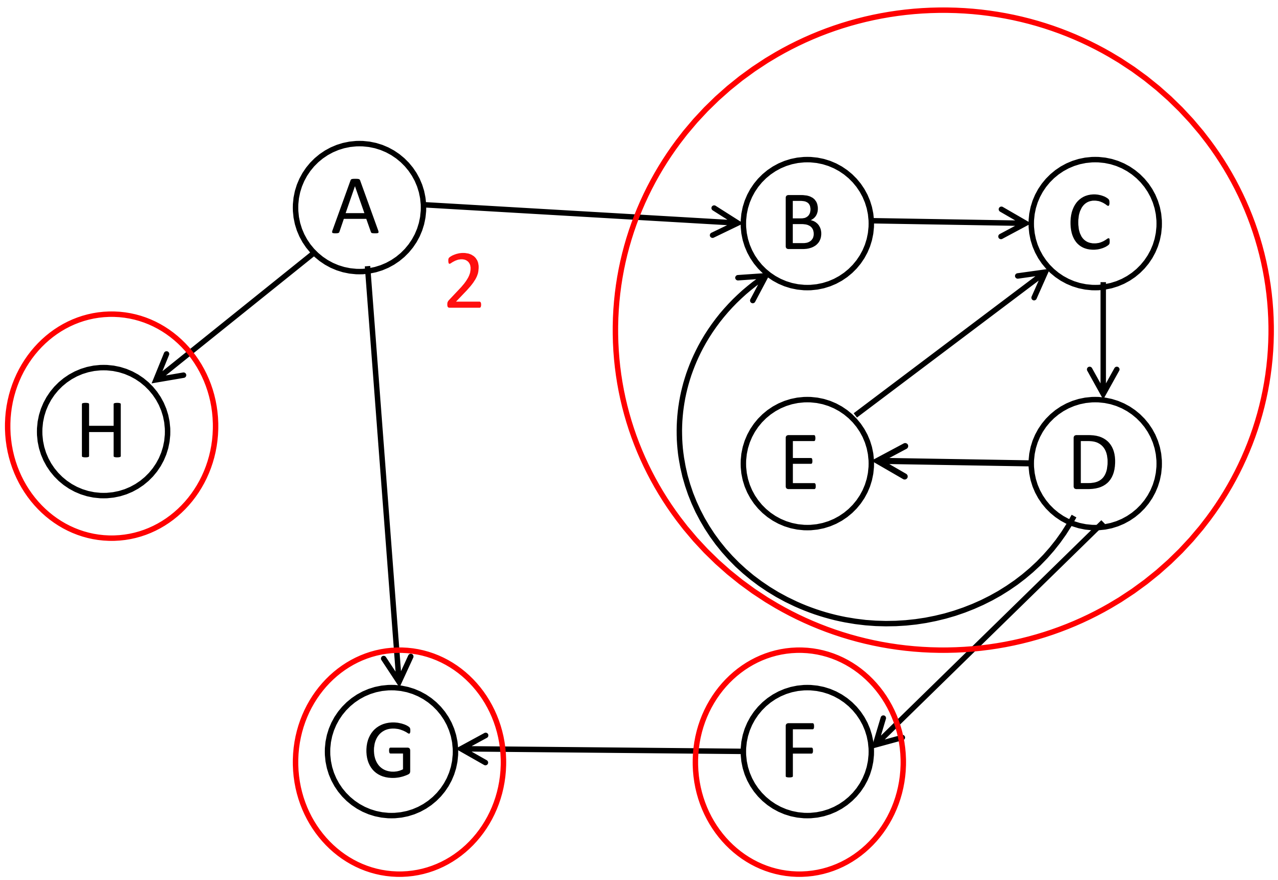
Example



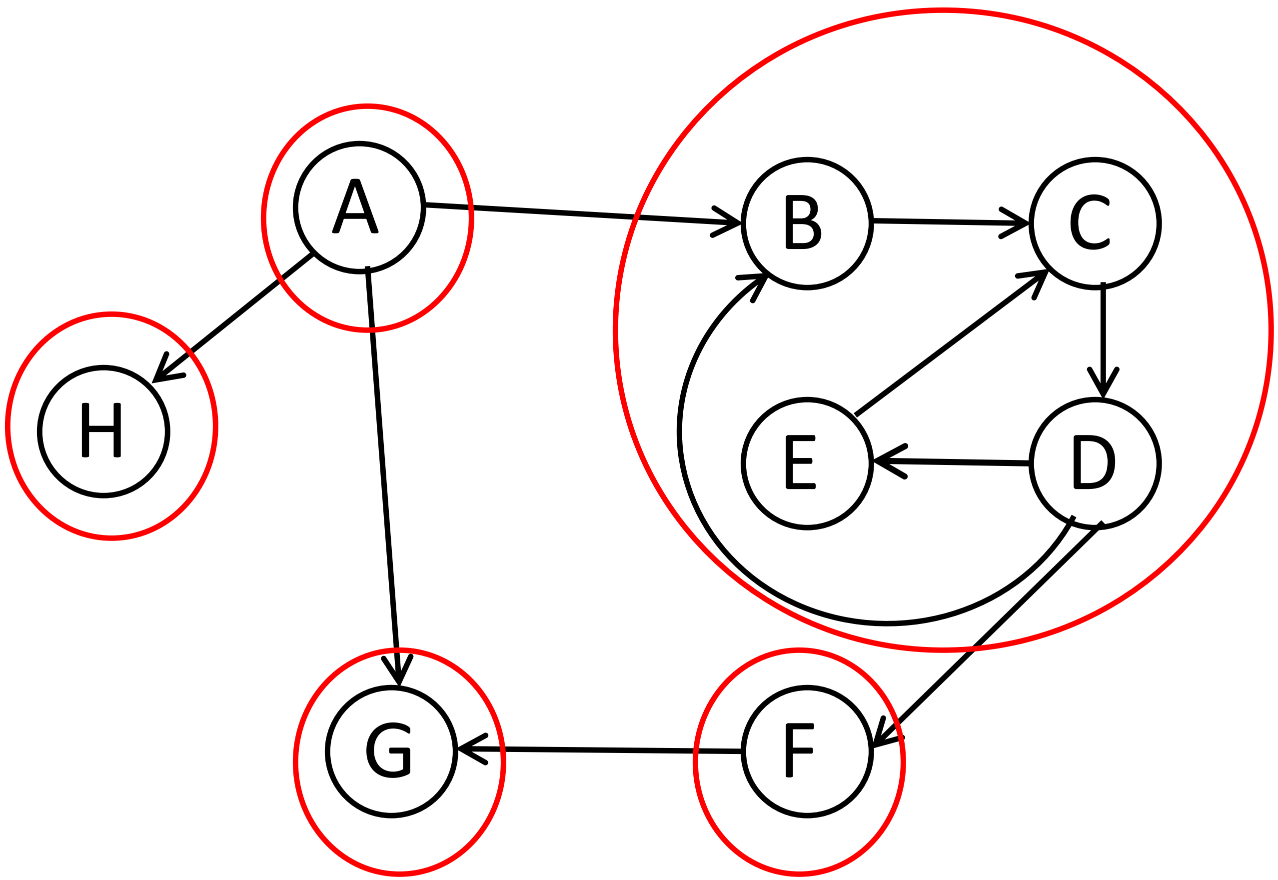
Example



Example



Example



Paths in Graphs (Ch 4)

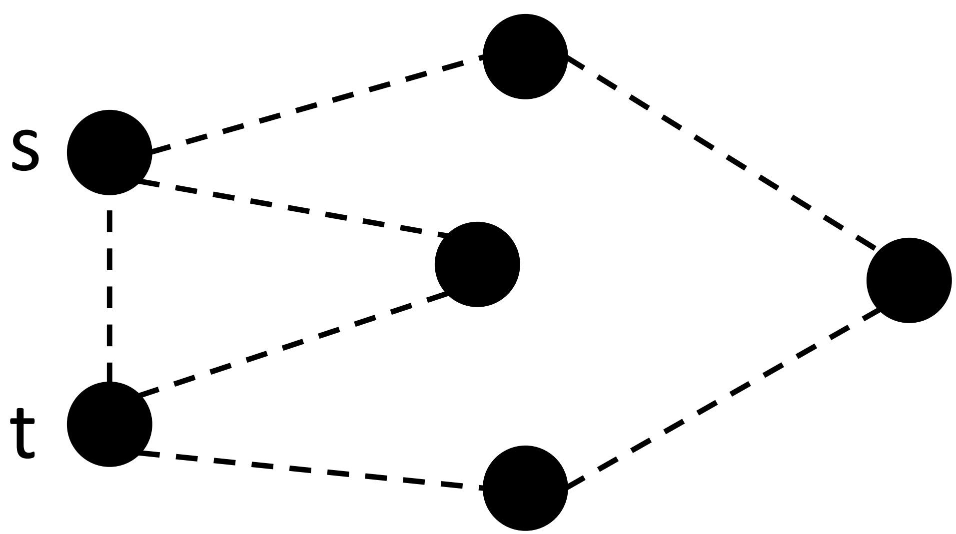
- Breadth First Search
- Dijkstra
  - Priority Queues
- Bellman-Ford
- Shortest Paths in DAGs

Motivation

`DFS/explore` allow us to determine *if* it is possible to get from one vertex to another, and using the DFS tree, you can also find *a* path. But this often is not an efficient path.

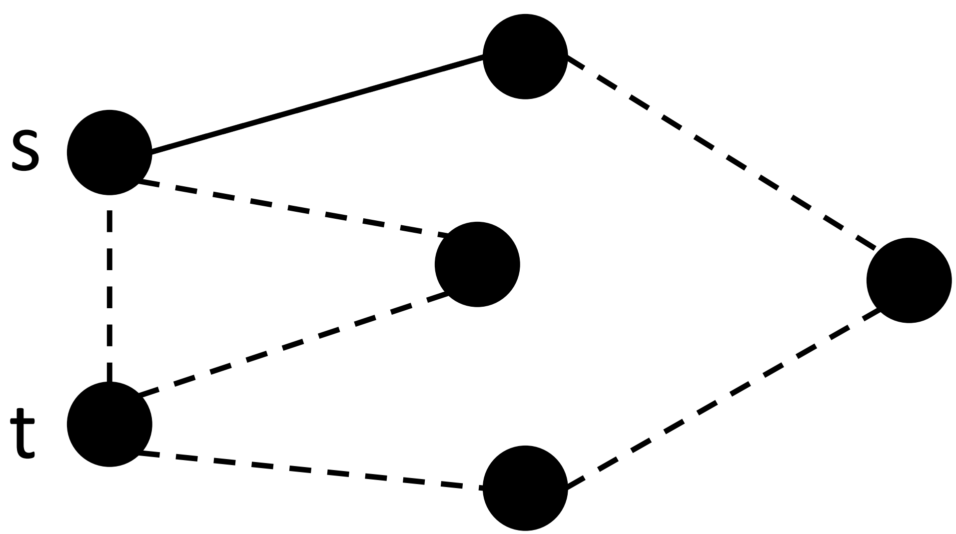
Motivation

`DFS`/`explore` allow us to determine *if* it is possible to get from one vertex to another, and using the DFS tree, you can also find *a* path. But this often is not an efficient path.



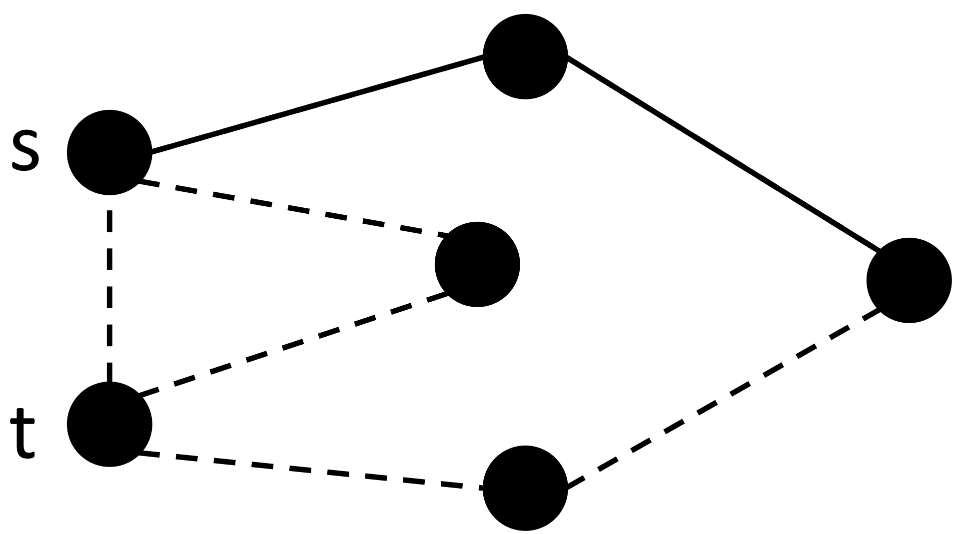
Motivation

`DFS/explore` allow us to determine *if* it is possible to get from one vertex to another, and using the DFS tree, you can also find *a* path. But this often is not an efficient path.



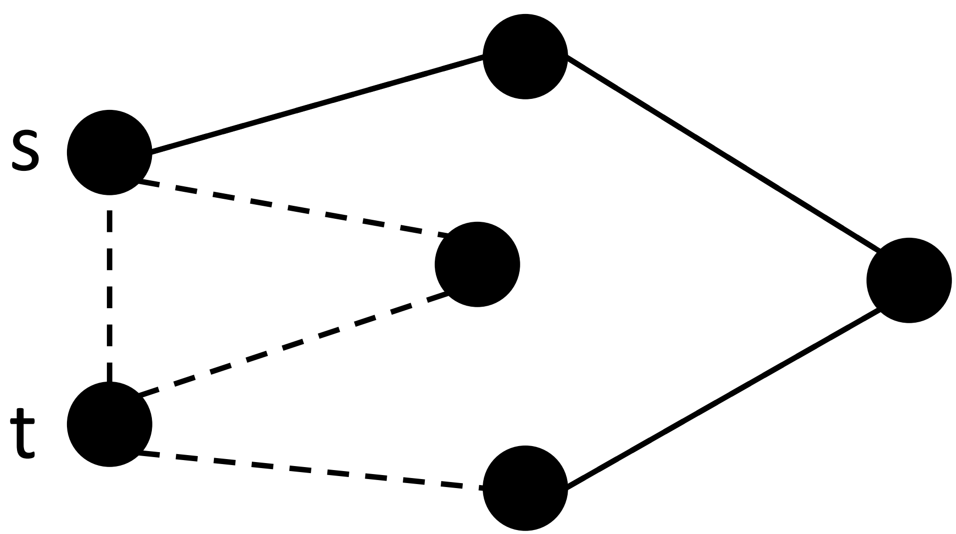
Motivation

`DFS/explore` allow us to determine *if* it is possible to get from one vertex to another, and using the DFS tree, you can also find *a* path. But this often is not an efficient path.



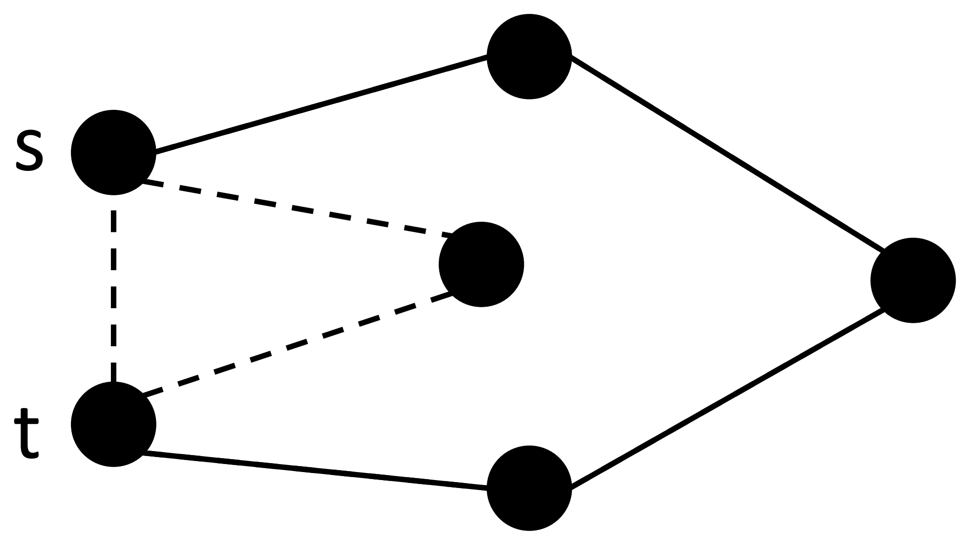
Motivation

`DFS/explore` allow us to determine *if* it is possible to get from one vertex to another, and using the DFS tree, you can also find *a* path. But this often is not an efficient path.



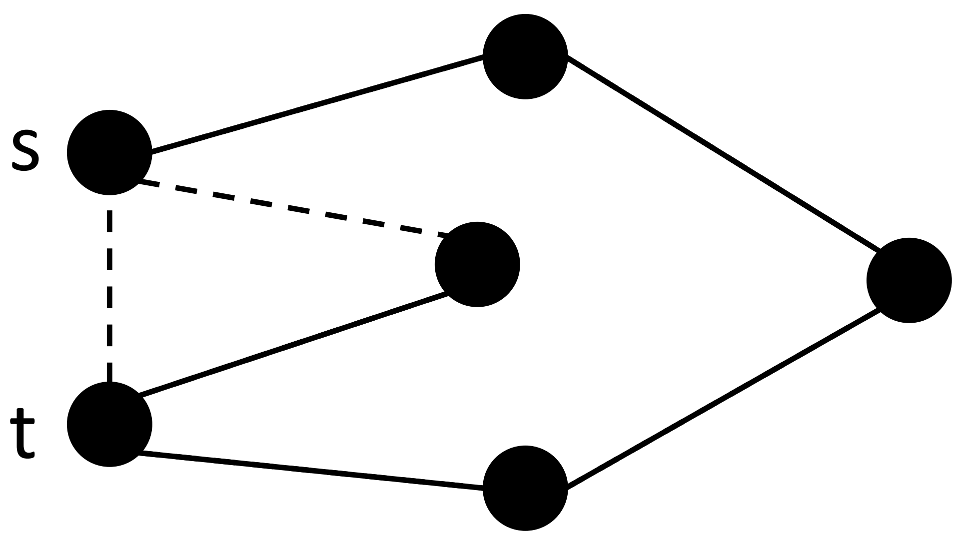
Motivation

`DFS/explore` allow us to determine *if* it is possible to get from one vertex to another, and using the DFS tree, you can also find *a* path. But this often is not an efficient path.



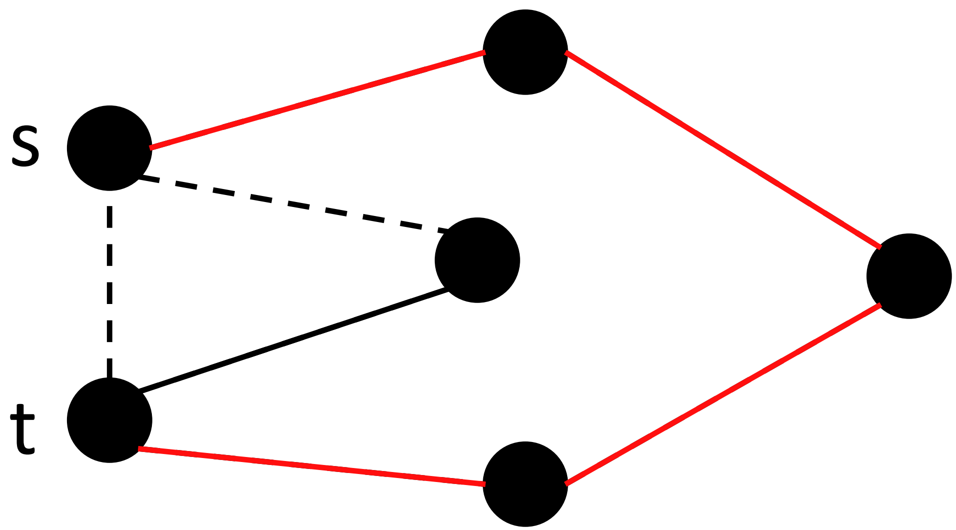
Motivation

`DFS`/`explore` allow us to determine *if* it is possible to get from one vertex to another, and using the DFS tree, you can also find *a* path. But this often is not an efficient path.



Motivation

`DFS/explore` allow us to determine *if* it is possible to get from one vertex to another, and using the DFS tree, you can also find *a* path. But this often is not an efficient path.



Goal

<u>Problem:</u> Given a graph G with two vertices s and t in the same connected component, find the *best* path from s to t.

Goal

Problem: Given a graph G with two vertices s and t in the same connected component, find the *best* path from s to t.

What do we mean by best?

Goal

Problem: Given a graph G with two vertices s and t in the same connected component, find the *best* path from s to t.

What do we mean by best?

- Least expensive

Goal

<u>Problem:</u> Given a graph G with two vertices s and t in the same connected component, find the *best* path from s to t.

What do we mean by best?

- Least expensive
- Best scenery

Goal

<u>Problem:</u> Given a graph G with two vertices s and t in the same connected component, find the *best* path from s to t.

What do we mean by best?

- Least expensive
- Best scenery
- Shortest

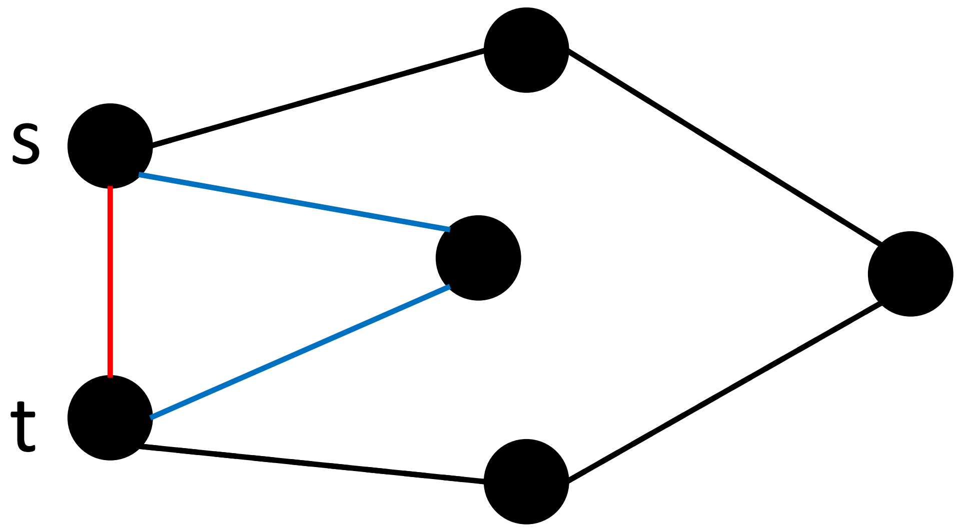
Goal

Problem: Given a graph G with two vertices s and t in the same connected component, find the *best* path from s to t.

What do we mean by best?

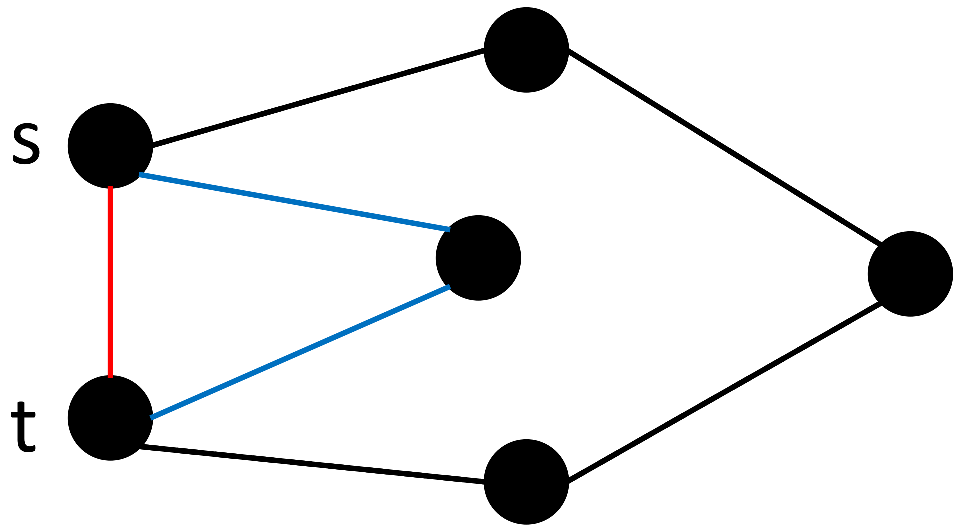
- Least expensive
- Best scenery
- Shortest
- For now: fewest edges

Example



4 edges
2 edges
1 edge

Example



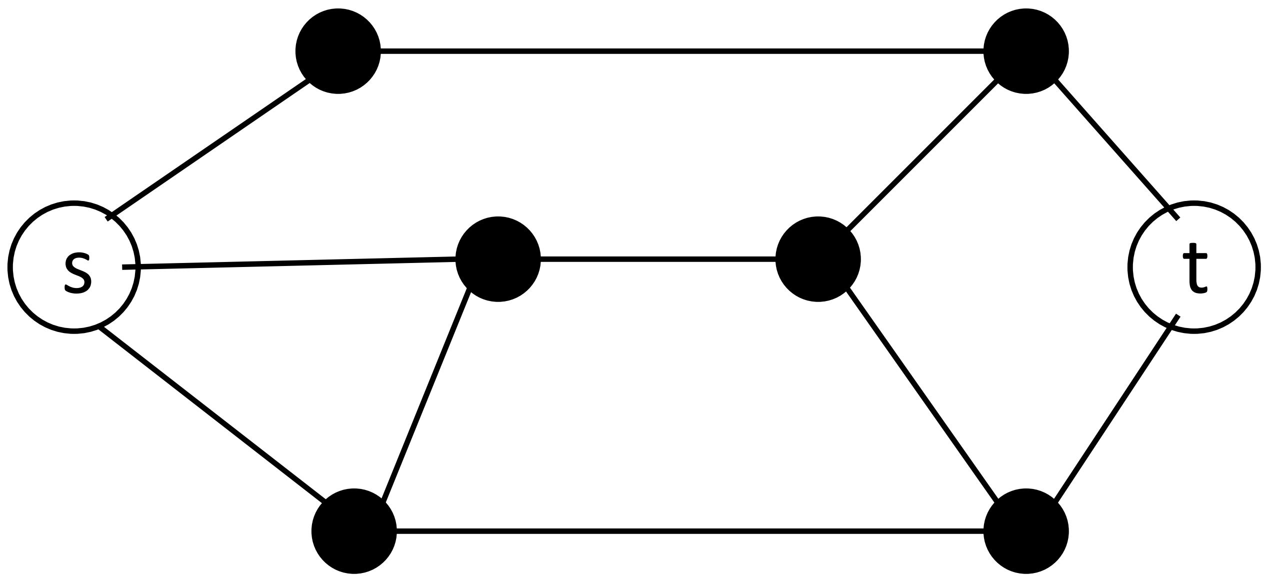
4 edges
2 edges
1 edge

Best is 1 edge.

Question: Shortest Path Length

What is the shortest path length from s to t in the graph below?

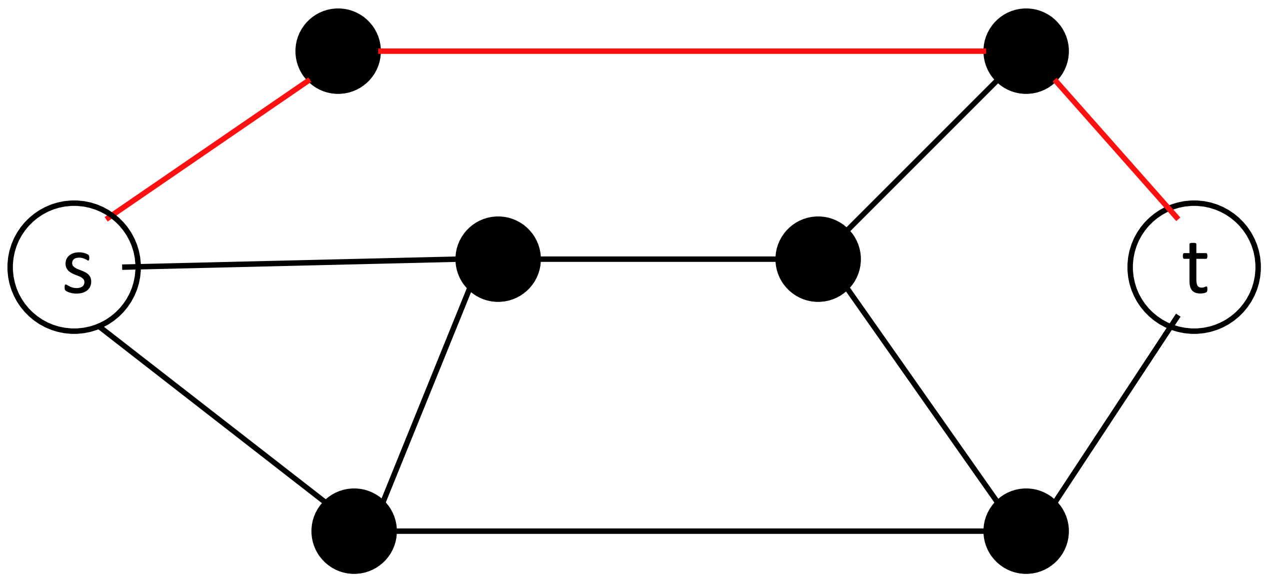
A) 2
B) 3
C) 4
D) 5
E) 6



Question: Shortest Path Length

What is the shortest path length from s to t in the graph below?

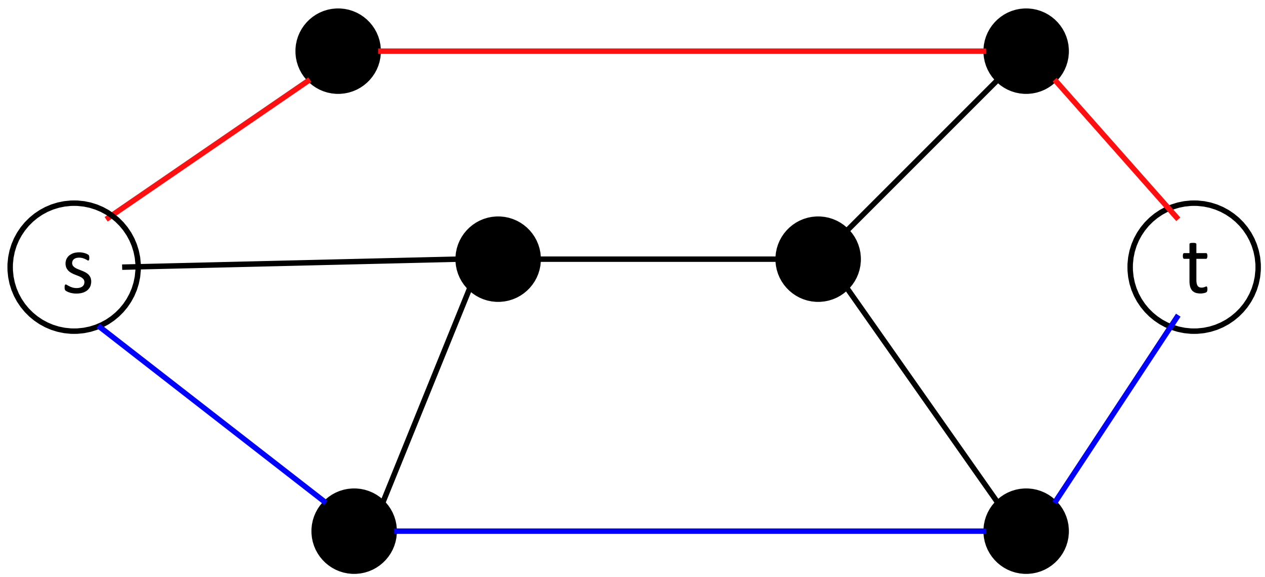
A) 2
B) 3
C) 4
D) 5
E) 6



Question: Shortest Path Length

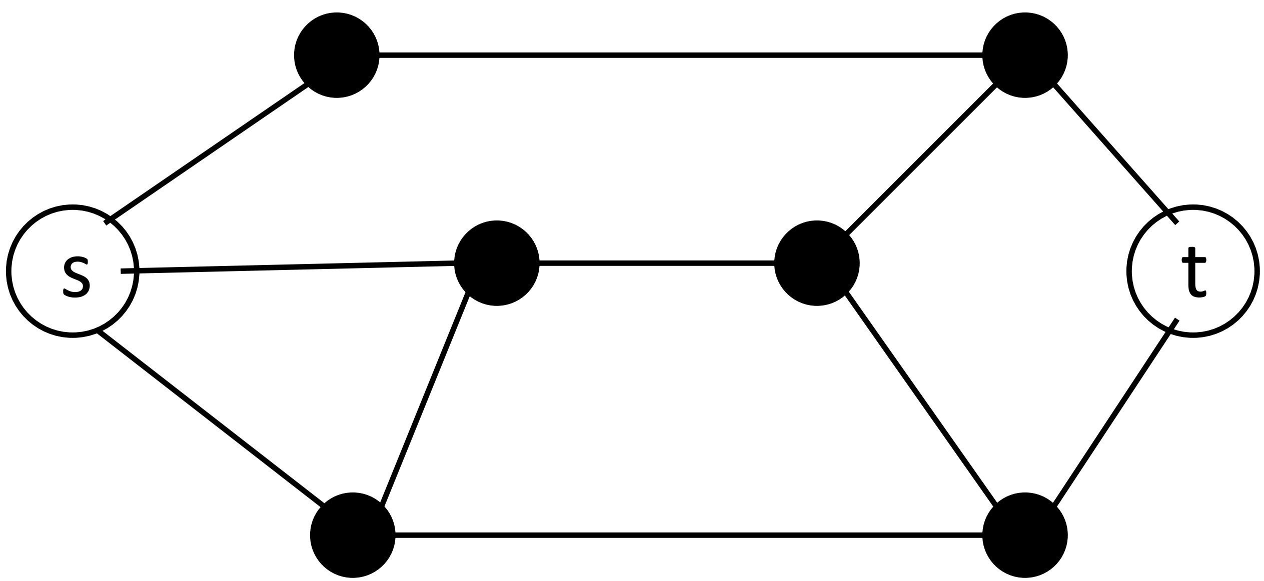
What is the shortest path length from s to t in the graph below?

A) 2
B) 3
C) 4
D) 5
E) 6



How do you *know*?

It is not hard to convince yourself that the shortest s-t path below has 3 edges, but how do we *know* there is nothing better?



Observation

If there is a length $\leq d$ s-v path, then there is some w adjacent to v with a length $\leq(d-1)$ s-w path.

Observation

If there is a length $\leq$d s-v path, then there is some w adjacent to v with a length $\leq$(d-1) s-w path.

Proof: w is the next to last vertex on the path.

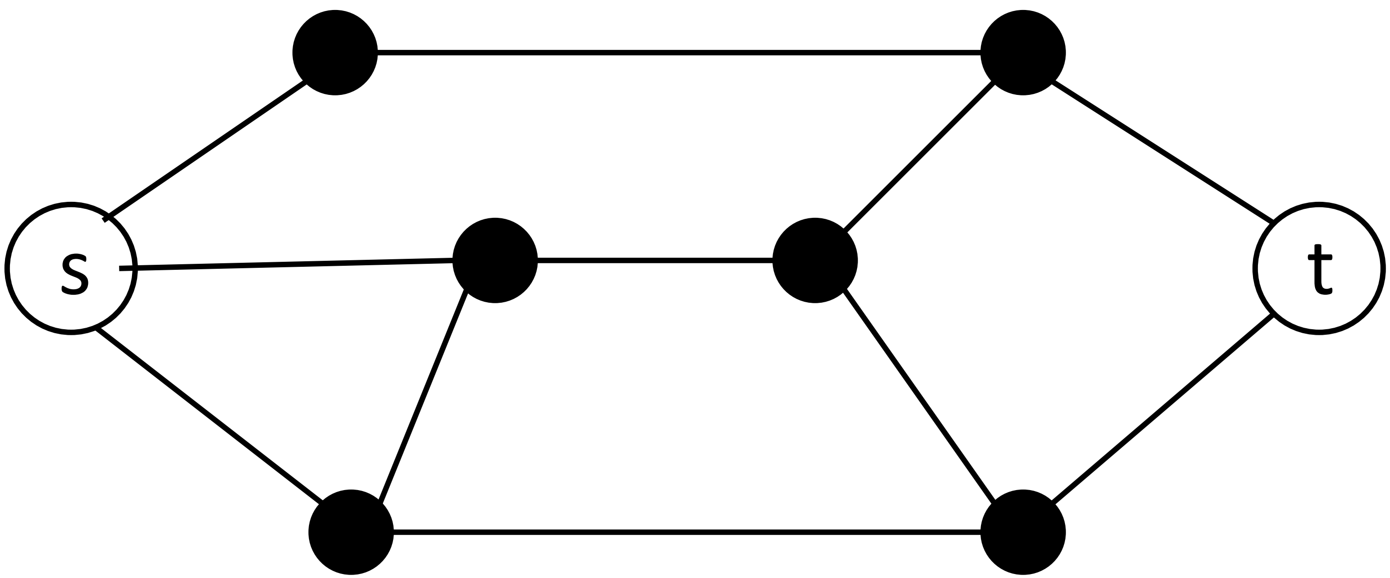
Observation

If there is a length ≤d s-v path, then there is some w adjacent to v with a length ≤(d-1) s-w path.

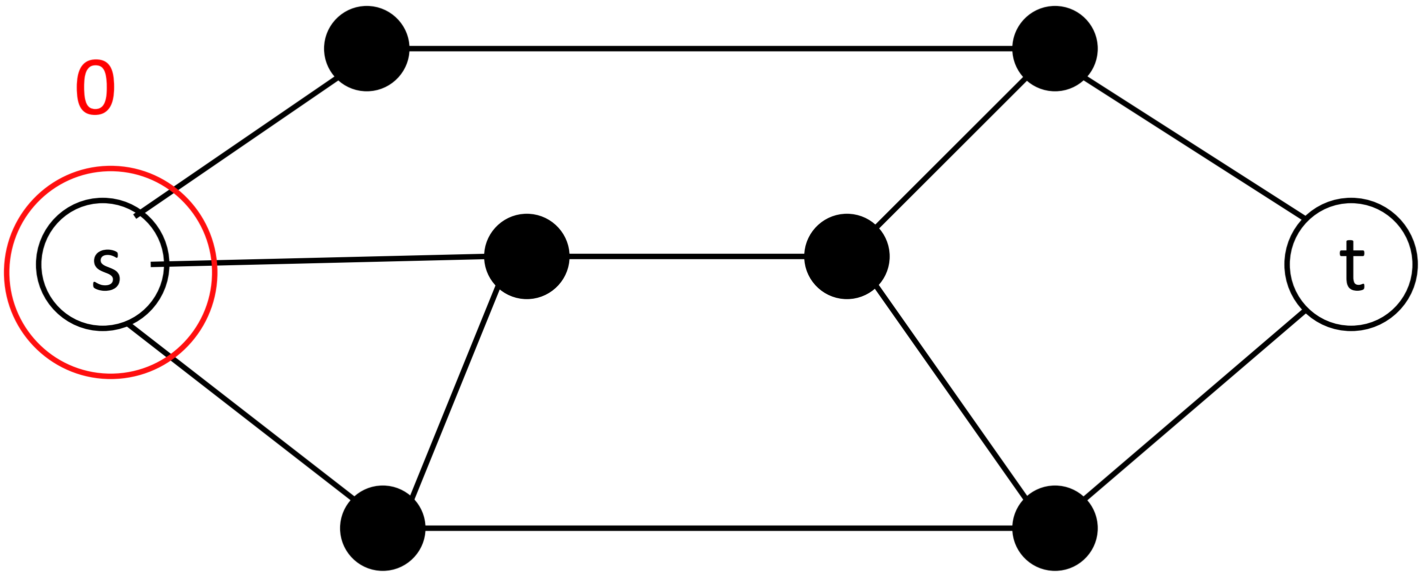
Proof: w is the next to last vertex on the path.

This means that if we know all of the vertices at distance ≤(d-1), we can find all of the vertices at distance ≤d.

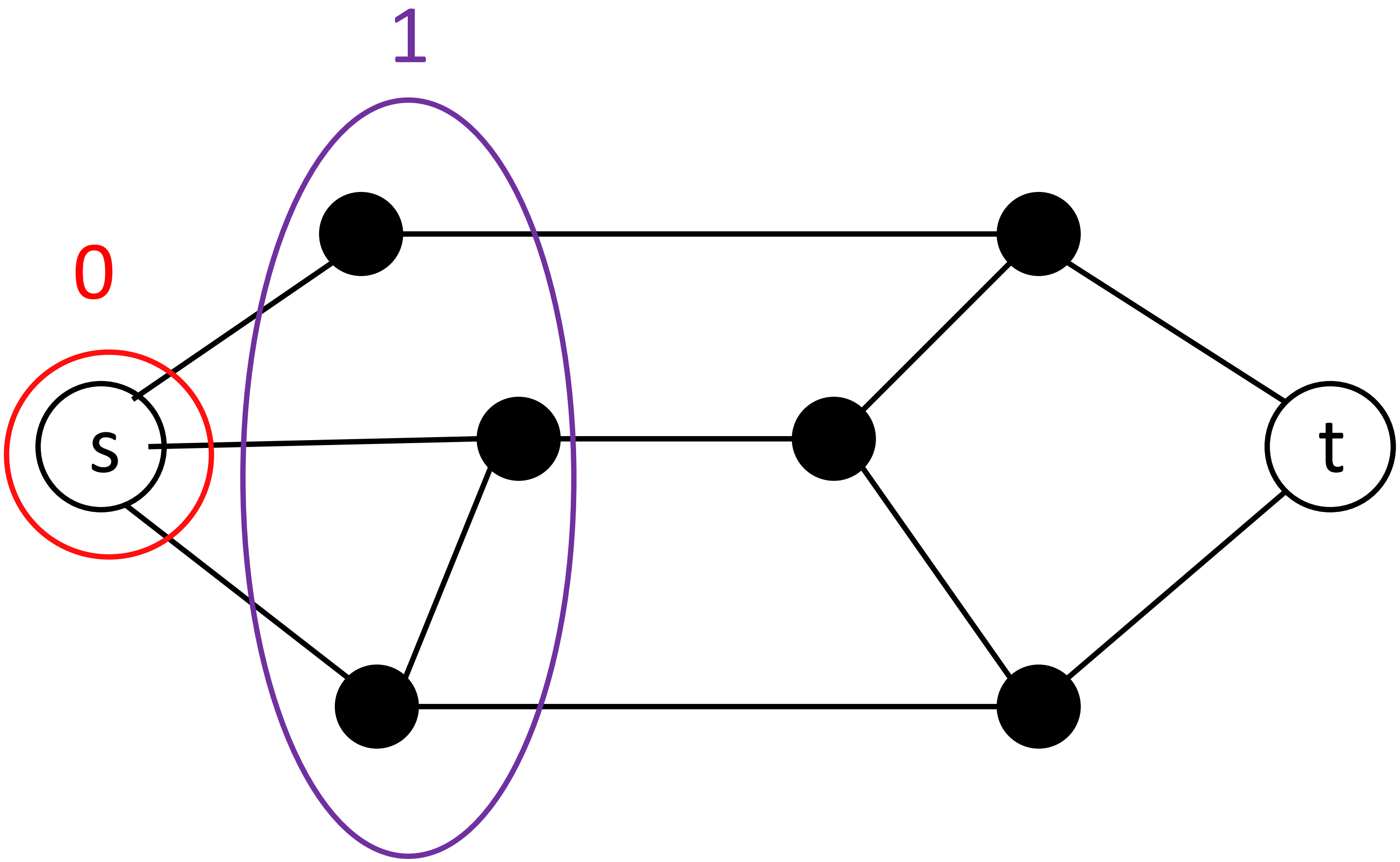
Example



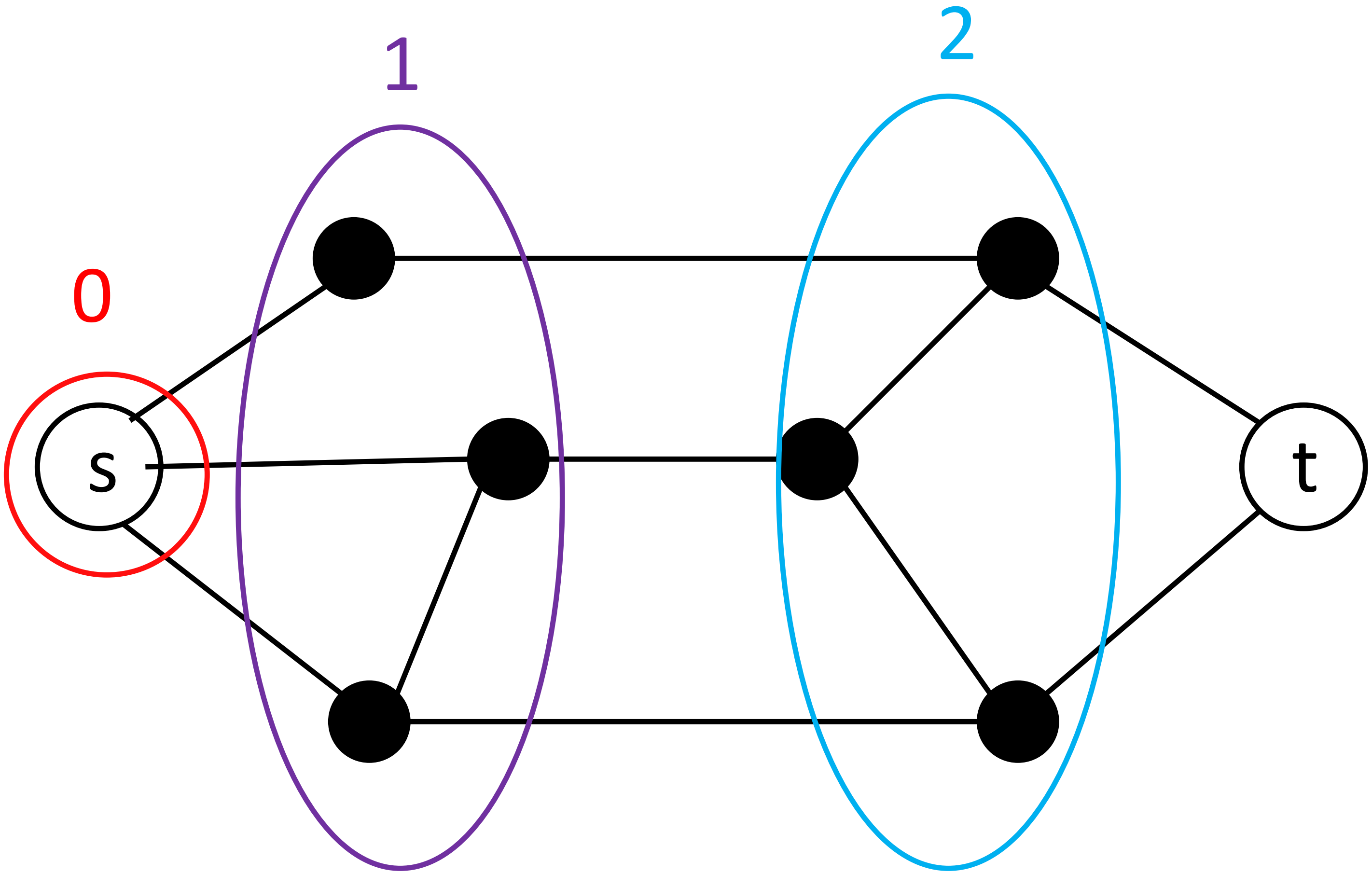
Example



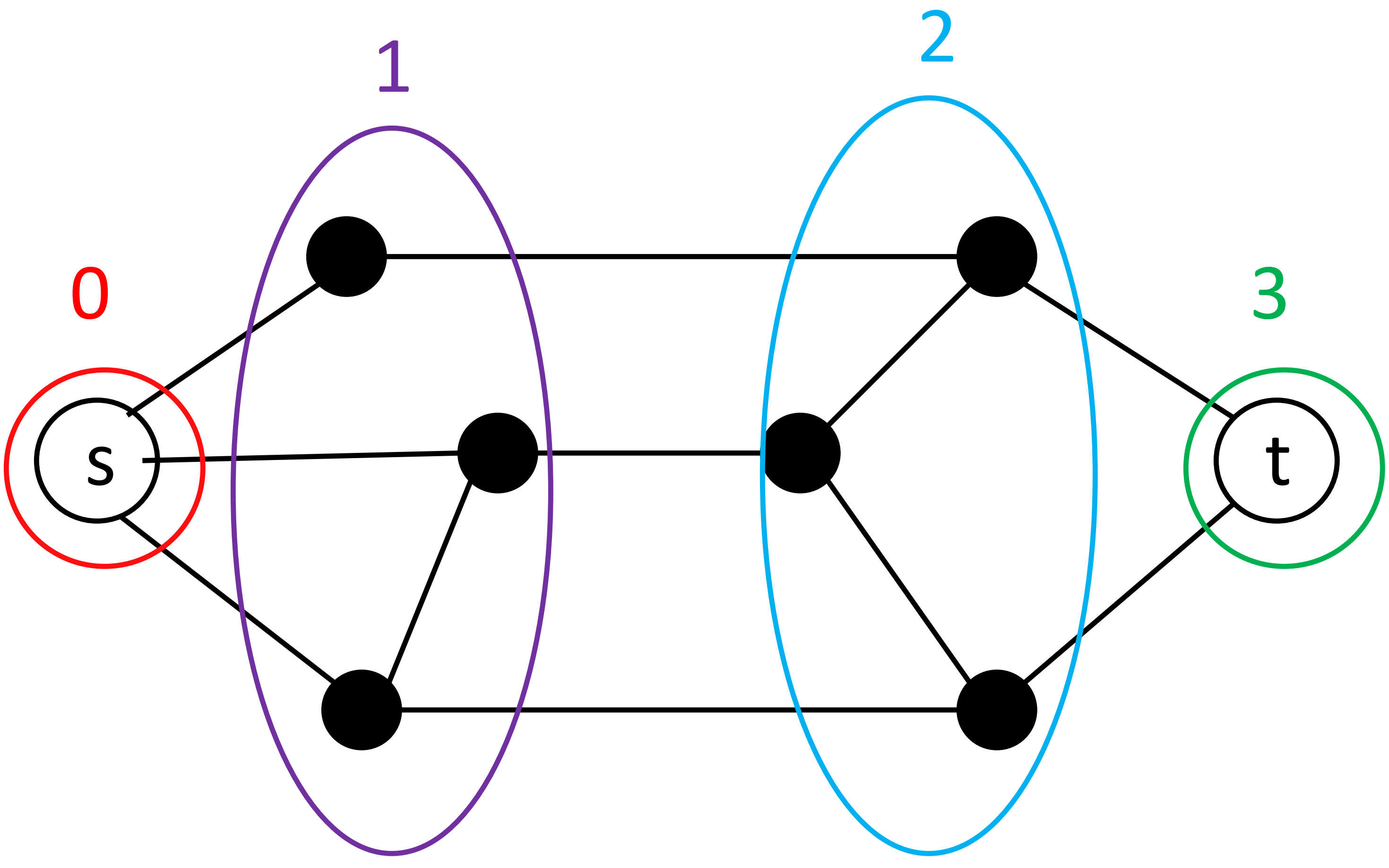
Example



Example



Example



Algorithm Idea

For each d create a list of all vertices at distance d from s.

Algorithm Idea

For each d create a list of all vertices at distance d from s.

- For d=0, this list is just {s}.
- For larger d, we want all new vertices adjacent to vertices at distance d-1.

ShortestPaths(G,s)

```
Initialize Array A
A[0] ← {s}
dist(s) ← 0
For d = 0 to n
  For u ∈ A[d]
    For (u,v) ∈ E
      If dist(v) undefined
        dist(v) ← d+1
        add v to A[d+1]
```

ShortestPaths(G,s)

What if dist(v) undefined at end?

```
Initialize Array A
A[0] ← {s}
dist(s) ← 0
For d = 0 to n
  For u ∈ A[d]
    For (u,v) ∈ E
      If dist(v) undefined
        dist(v) ← d+1
        add v to A[d+1]
```

ShortestPaths(G,s)

```
For v ∈ V, dist(v) ← ∞
Initialize Array A
A[0] ← {s}
dist(s) ← 0
For d = 0 to n
   For u ∈ A[d]
      For (u,v) ∈ E
         If dist(v) = ∞
            dist(v) ← d+1
            add v to A[d+1]
```

What if dist(v) undefined at end?

```
ShortestPaths(G,s)
  For v ∈ V, dist(v) ← ∞
  Initialize Array A
  A[0] ← {s}
  dist(s) ← 0
  For d = 0 to n
    For u ∈ A[d]
      For (u,v) ∈ E
        If dist(v) = ∞
          dist(v) ← d+1
          add v to A[d+1]
```

```
ShortestPaths(G,s)
  For v ∈ V, dist(v) ← ∞
  Initialize Array A
  A[0] ← {s}
  dist(s) ← 0
  For d = 0 to n
    For u ∈ A[d]
      For (u,v) ∈ E
        If dist(v) = ∞
          dist(v) ← d+1
          add v to A[d+1]
```

Algorithm goes through A[0], A[1],… in order. Can just use a queue.

```
ShortestPaths(G,s)
  For v ∈ V, dist(v) ← ∞
  Initialize Queue Q
  Q.enqueue(s)
  dist(s) ← 0
  While Q not empty
    u ← front(Q)
      For (u,v) ∈ E
        If dist(v) = ∞
          dist(v) ← dist(u)+1
          Q.enqueue(v)
```

Algorithm goes through A[0], A[1],… in order. Can just use a queue.

ShortestPaths(G,s)

```
For v ∈ V, dist(v) ← ∞
Initialize Queue Q
Q.enqueue(s)
dist(s) ← 0
While Q not empty
  u ← front(Q)
    For (u,v) ∈ E
      If dist(v) = ∞
        dist(v) ← dist(u)+1
        Q.enqueue(v)
```

```
ShortestPaths(G,s)
  For v ∈ V, dist(v) ← ∞
  Initialize Queue Q
  Q.enqueue(s)
  dist(s) ← 0
  While Q not empty
    u ← front(Q)
      For (u,v) ∈ E
        If dist(v) = ∞
          dist(v) ← dist(u)+1
          Q.enqueue(v)
```

Keep track of paths

```
ShortestPaths(G,s)
  For v ∈ V, dist(v) ← ∞
  Initialize Queue Q
  Q.enqueue(s)
  dist(s) ← 0
  While Q not empty
    u ← front(Q)
      For (u,v) ∈ E
        If dist(v) = ∞
          dist(v) ← dist(u)+1
          Q.enqueue(v)
          v.prev ← u
```

Keep track of paths

```
ShortestPaths(G,s)
  For v ∈ V, dist(v) ← ∞
  Initialize Queue Q
  Q.enqueue(s)
  dist(s) ← 0
  While Q not empty
    u ← front(Q)
      For (u,v) ∈ E
        If dist(v) = ∞
          dist(v) ← dist(u)+1
          Q.enqueue(v)
          v.prev ← u
```

```
BreadthFirstSearch(G,s)
  For v ∈ V, dist(v) ← ∞
  Initialize Queue Q
  Q.enqueue(s)
  dist(s) ← 0
  While Q not empty
    u ← front(Q)
      For (u,v) ∈ E
        If dist(v) = ∞
          dist(v) ← dist(u)+1
          Q.enqueue(v)
          v.prev ← u
```

```
BreadthFirstSearch(G,s)
  For v ∈ V, dist(v) ← ∞
  Initialize Queue Q
  Q.enqueue(s)
  dist(s) ← 0
  While Q not empty
    u ← front(Q)
      For (u,v) ∈ E
        If dist(v) = ∞
          dist(v) ← dist(u)+1
          Q.enqueue(v)
          v.prev ← u
```

Runtime

```
BFS(G,s)
   For v ∈ V, dist(v) ← ∞
   Initialize Queue Q
   Q.enqueue(s)
   dist(s) ← 0
   While(Q nonempty)
      u ← front(Q)
      For (u,v) ∈ E
         If dist(v) = ∞
            dist(v) ← dist(u)+1
            Q.enqueue(v)
            v.prev ← u
```

Runtime

```
BFS(G,s)
  For v ∈ V, dist(v) ← ∞
  Initialize Queue Q
  Q.enqueue(s)
  dist(s) ← 0
  While(Q nonempty)
    u ← front(Q)
    For (u,v) ∈ E
      If dist(v) = ∞
        dist(v) ← dist(u)+1
        Q.enqueue(v)
        v.prev ← u
```

O(|V|)

Runtime

```
BFS(G,s)
  For v ∈ V, dist(v) ← ∞
  Initialize Queue Q
  Q.enqueue(s)
  dist(s) ← 0
  While(Q nonempty)
    u ← front(Q)
    For (u,v) ∈ E
      If dist(v) = ∞
        dist(v) ← dist(u)+1
        Q.enqueue(v)
        v.prev ← u
```

O(|V|)

O(|V|) iterations

Runtime

```
BFS(G,s)
  For v ∈ V, dist(v) ← ∞
  Initialize Queue Q
  Q.enqueue(s)
  dist(s) ← 0
  While(Q nonempty)
    u ← front(Q)
    For (u,v) ∈ E
      If dist(v) = ∞
        dist(v) ← dist(u)+1
        Q.enqueue(v)
        v.prev ← u
```

O(|V|)

O(|V|) iterations

O(|E|) total iterations

Runtime

```
BFS(G,s)
  For v ∈ V, dist(v) ← ∞
  Initialize Queue Q
  Q.enqueue(s)
  dist(s) ← 0
  While(Q nonempty)
    u ← front(Q)
    For (u,v) ∈ E
      If dist(v) = ∞
        dist(v) ← dist(u)+1
        Q.enqueue(v)
        v.prev ← u
```

$O(|V|)$

$O(|V|)$ iterations

$O(|E|)$ total iterations

Total runtime:

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

DFS vs BFS

- Processed vertices (visited, dist $< \infty$)
- For each vertex, process all unprocessed neighbors

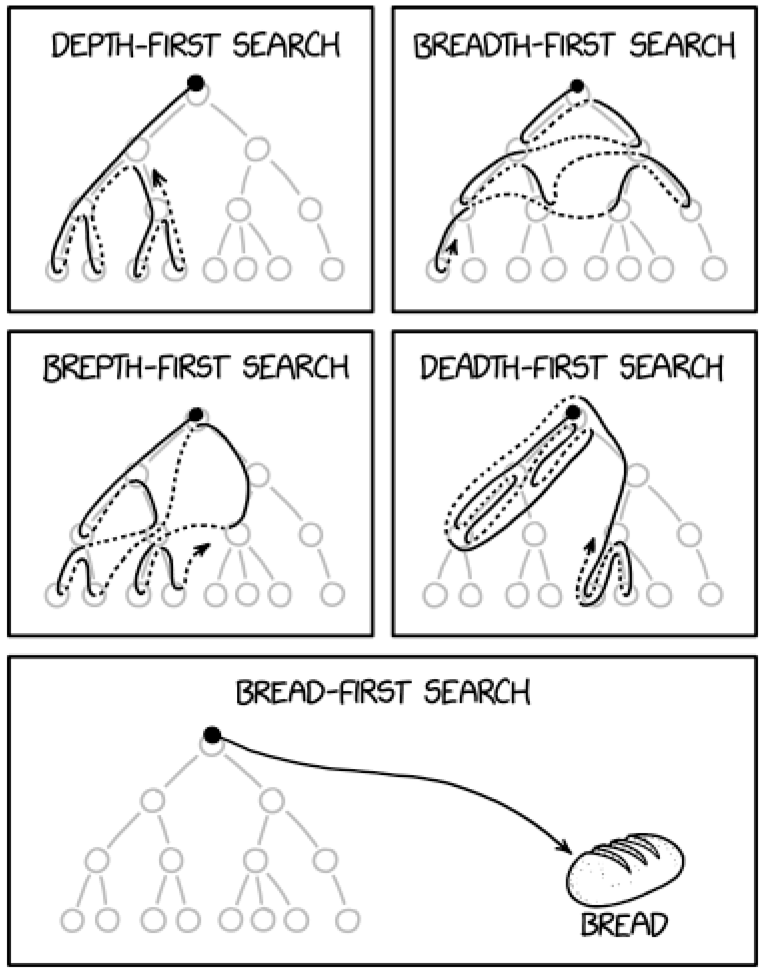
DFS vs BFS

- Processed vertices (visited, dist $< \infty$)
- For each vertex, process all unprocessed neighbors
- Difference:
  - DFS uses a stack to store vertices waiting to be processed
  - BFS uses a queue

DFS vs BFS

- Processed vertices (visited, dist < ∞)
- For each vertex, process all unprocessed neighbors
- Difference:
  - DFS uses a stack to store vertices waiting to be processed
  - BFS uses a queue
- Big effect
  - DFS goes depth first – very long path
  - BFS is breadth first – visits all side paths

DFS vs BFS vs Others?



https://xkcd.com/2407/