Graph Algorithms - I

CS2800: Design and Analysis of Algorithms

Yadu Vasudev
yadu@cse.iitm.ac.in
IIT Madras

Module plan

1. Graph representations
2. Graph traversals and applications
3. Directed graphs
4. Shortest paths in graphs

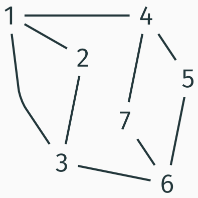
Graph representations

Graphs - basics

Graph: A set of vertices V, and a set of edges $E \subseteq V \times V$

Graphs - basics

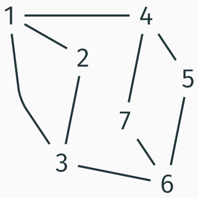
Graph: A set of vertices V, and a set of edges $E \subseteq V \times V$



- Transportation networks
- Social networks and the web graph
- Circuits

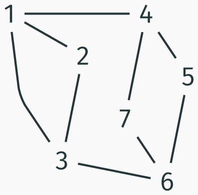
Graphs - basics

Graph: A set of vertices V, and a set of edges $E \subseteq V \times V$



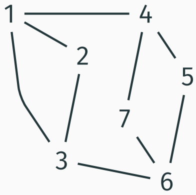
- Transportation networks
- Social networks and the web graph
- Circuits
- Ubiquitous abstraction for a variety of problems!

Graph representations - adjacency matrices



$$
\begin{array}{c c}
 & \begin{array}{ccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} &
\begin{pmatrix}
0 & 1 & 1 & 1 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 1 & 0
\end{pmatrix}
\end{array}
$$

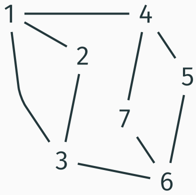
Graph representations - adjacency matrices



$$
\begin{array}{c c}
 & \begin{array}{ccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} &
\begin{pmatrix}
0 & 1 & 1 & 1 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 1 & 0
\end{pmatrix}
\end{array}
$$

- An n vertex graph requires $O(n^2)$ space for the adjacency matrix

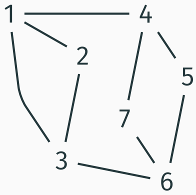
Graph representations - adjacency matrices



$$\begin{array}{c c} & \begin{array}{ccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} & \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

- An n vertex graph requires $O(n^2)$ space for the adjacency matrix
- Edge queries can be made in $O(1)$ time

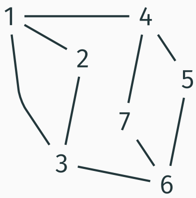
Graph representations - adjacency matrices



$$\begin{array}{cc} & \begin{array}{ccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} & \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

- An n vertex graph requires $O(n^2)$ space for the adjacency matrix
- Edge queries can be made in $O(1)$ time
- Finding all neighbors requires $O(n)$ time

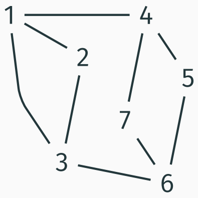
Graph representations - adjacency matrices

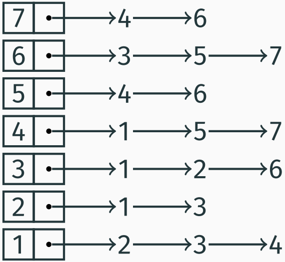


$$
\begin{array}{c c}
 & \begin{array}{ccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} &
\begin{pmatrix}
0 & 1 & 1 & 1 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 1 & 0
\end{pmatrix}
\end{array}
$$

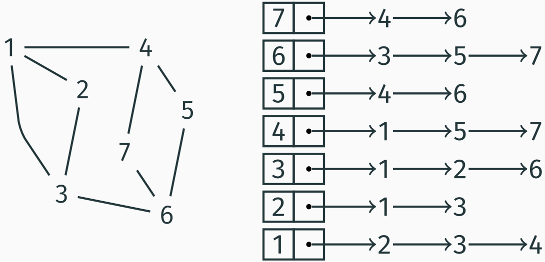
- An n vertex graph requires $O(n^2)$ space for the adjacency matrix
- Edge queries can be made in $O(1)$ time
- Finding all neighbors requires $O(n)$ time
- More useful for representing dense graphs $(|E| = \Omega(n^2))$

Graph representations - adjacency lists



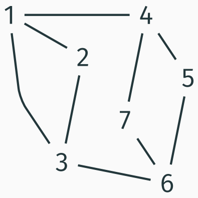


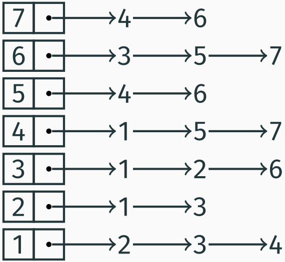
Graph representations - adjacency lists



- Requires $O(|V| + |E|)$ space to represent a graph
- Checking the existence of edge (u, v) requires time $O(\min\{d(u), d(v)\})$

Graph representations - adjacency lists





- Requires $O(|V| + |E|)$ space to represent a graph
- Checking the existence of edge (u, v) requires time $O(\min\{d(u), d(v)\})$
- Finding all neighbors of u requires $O(d(u))$ time

Graph traversals and applications

Reachability in graphs

Reachability: Given a vertex s, what are the vertices reachable from s via edges of the graph?

Reachability in graphs

Reachability: Given a vertex s, what are the vertices reachable from s via edges of the graph?

- Perhaps the most important primitive of graph operations

Reachability in graphs

Reachability: Given a vertex s, what are the vertices reachable from s via edges of the graph?

- Perhaps the most important primitive of graph operations
- Two important variants:

Reachability in graphs

Reachability: Given a vertex s, what are the vertices reachable from s via edges of the graph?

- Perhaps the most important primitive of graph operations
- Two important variants:
  - Depth-First Search (DFS)
  - Breadth-First Search (BFS)

Reachability in graphs

Reachability: Given a vertex s, what are the vertices reachable from s via edges of the graph?

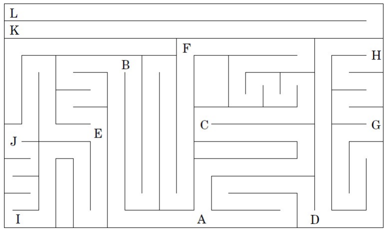
- Perhaps the most important primitive of graph operations
- Two important variants:
  - Depth-First Search (DFS)
  - Breadth-First Search (BFS)
- Can be used to collect a lot of auxilliary information about the structure of the graph, and
- both BFS and DFS are highly efficient in terms of time and space complexity

Depth-First Search

How do you traverse a maze?

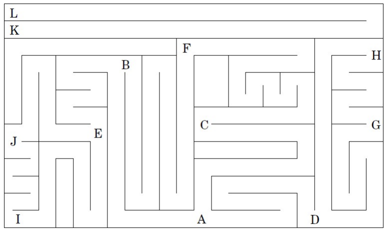
Depth-First Search

How do you traverse a maze?



Depth-First Search

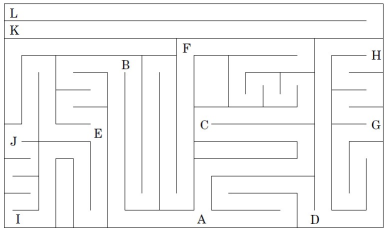
How do you traverse a maze?



- Mark the starting location with a chalk

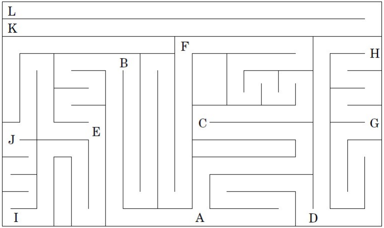
Depth-First Search

How do you traverse a maze?



- Mark the starting location with a chalk
- Walk along a path until you reach the next junction, mark the junction

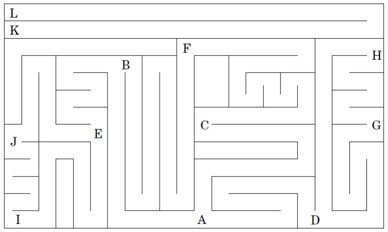
How do you traverse a maze?



- Mark the starting location with a chalk
- Walk along a path until you reach the next junction, mark the junction
- Keep continuing until you reach your destination or hit a dead-end

Depth-First Search

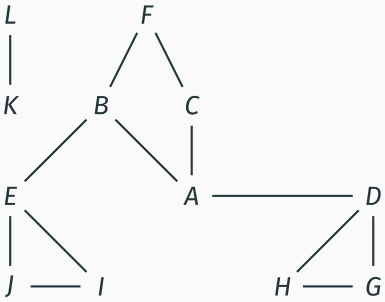
How do you traverse a maze?



- Mark the starting location with a chalk
- Walk along a path until you reach the next junction, mark the junction
- Keep continuing until you reach your destination or hit a dead-end
- If you hit a dead-end or an already marked junction, retrace your steps to the last marked junction and try out the next path ...

Depth-First Search

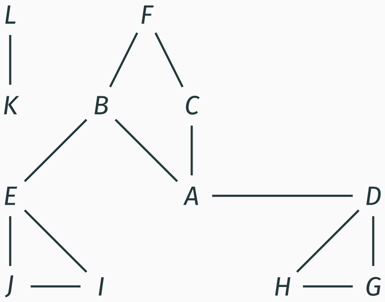
How do you traverse a maze?



- Mark the starting location with a chalk
- Walk along a path until you reach the next junction, mark the junction
- Keep continuing until you reach your destination or hit a dead-end
- If you hit a dead-end or an already marked junction, retrace your steps to the last marked junction and try out the next path …

Depth-First Search

How do you traverse a maze?

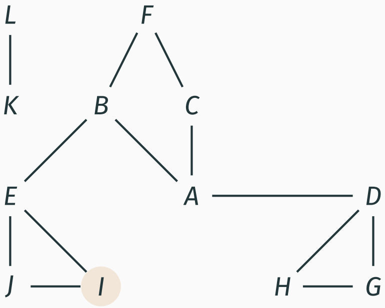


RecursiveDFS

if v is unmarked **then**
  mark v
  foreach edge vw **do**
    Recursively traverse starting from v

Depth-First Search

How do you traverse a maze?

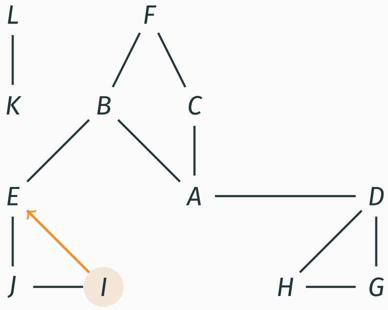


RecursiveDFS

if v is unmarked **then**
  mark v
  foreach edge vw **do**
    Recursively traverse
    starting from v

Depth-First Search

How do you traverse a maze?

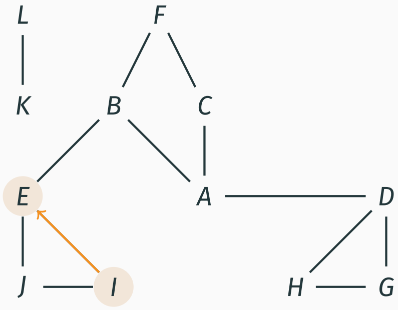


RecursiveDFS

if v is unmarked **then**
  mark v
  foreach edge vw **do**
    Recursively traverse starting from v

Depth-First Search

How do you traverse a maze?

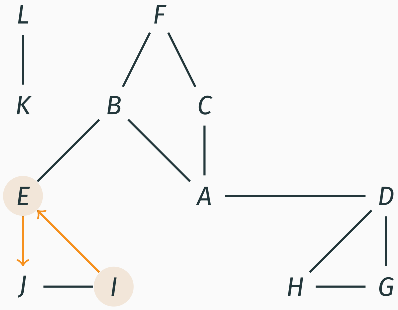


RecursiveDFS

if v is unmarked **then**
  mark v
  foreach edge vw **do**
    Recursively traverse starting from v

Depth-First Search

How do you traverse a maze?

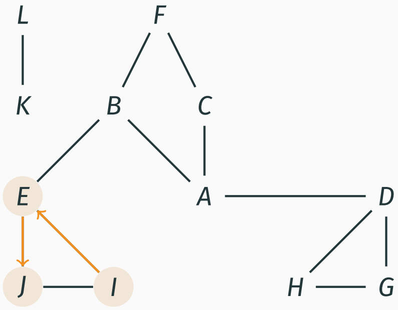


RecursiveDFS

if v is unmarked **then**
    mark v
    foreach edge vw **do**
        Recursively traverse
        starting from v

Depth-First Search

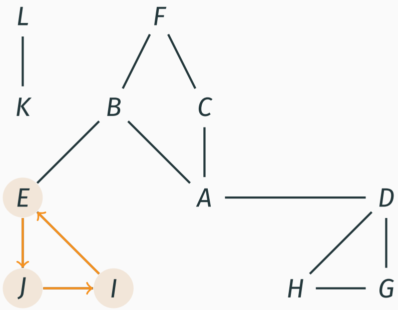
How do you traverse a maze?



RECURSIVEDFS

if *v* is unmarked **then**
  mark *v*
  foreach edge *vw* **do**
    Recursively traverse starting from *v*

How do you traverse a maze?

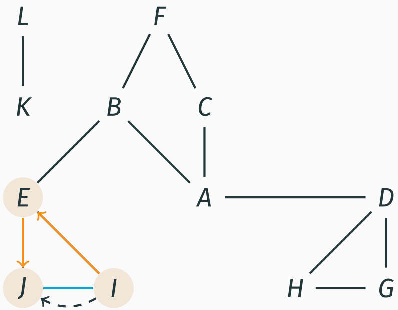


RecursiveDFS

if *v* is unmarked **then**
  mark *v*
  foreach edge *vw* **do**
    Recursively traverse starting from *v*

Depth-First Search

How do you traverse a maze?

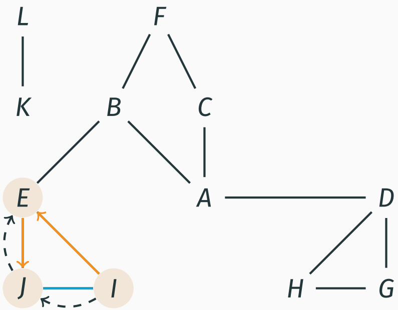


RecursiveDFS

if v is unmarked **then**
  mark v
  foreach edge vw **do**
    Recursively traverse
     starting from v

Depth-First Search

How do you traverse a maze?

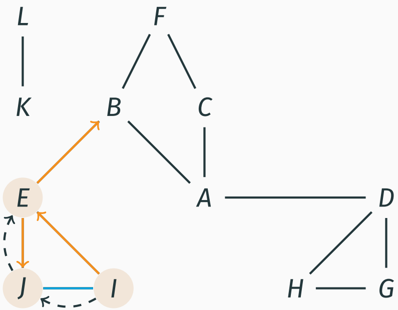


RECURSIVEDFS

if v is unmarked **then**
  mark v
  foreach edge vw **do**
    Recursively traverse
    starting from v

Depth-First Search

How do you traverse a maze?

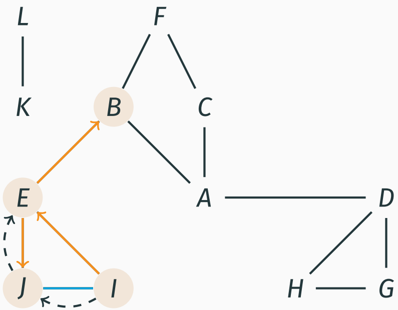


RECURSIVEDFS

if *v* is unmarked **then**
  mark *v*
  foreach edge *vw* **do**
    Recursively traverse starting from *v*

Depth-First Search

How do you traverse a maze?

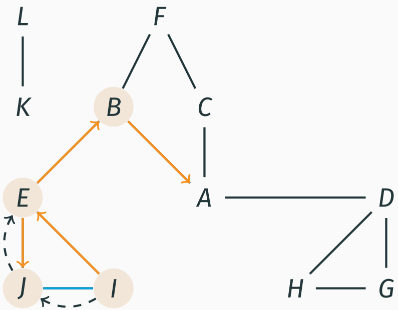


RecursiveDFS

if *v* is unmarked **then**
  mark *v*
  foreach edge *vw* **do**
    Recursively traverse
    starting from *v*

Depth-First Search

How do you traverse a maze?

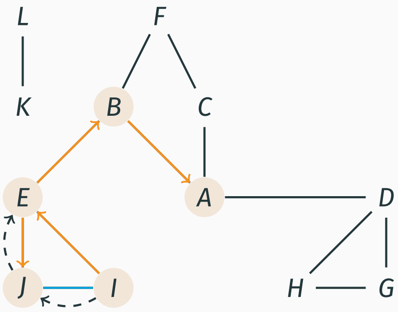


RecursiveDFS

if *v* is unmarked **then**
  mark *v*
  foreach edge *vw* **do**
    Recursively traverse
     starting from *v*

Depth-First Search

How do you traverse a maze?

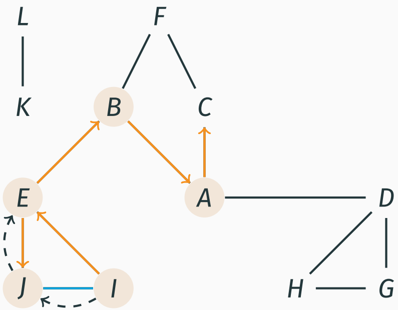


RecursiveDFS

if *v* is unmarked **then**
  mark *v*
  foreach edge *vw* **do**
    Recursively traverse
     starting from *v*

Depth-First Search

How do you traverse a maze?

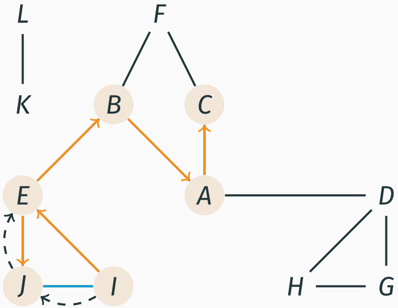


RecursiveDFS

if v is unmarked **then**
  mark v
  foreach edge vw **do**
    Recursively traverse
     starting from v

Depth-First Search

How do you traverse a maze?

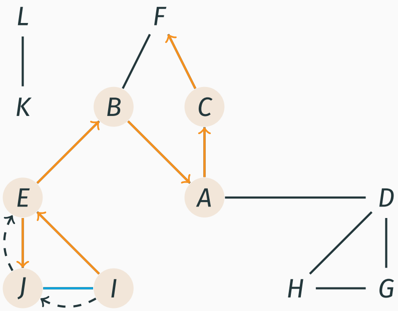


RECURSIVEDFS

if *v* is unmarked **then**
  mark *v*
  foreach edge *vw* **do**
    Recursively traverse
     starting from *v*

Depth-First Search

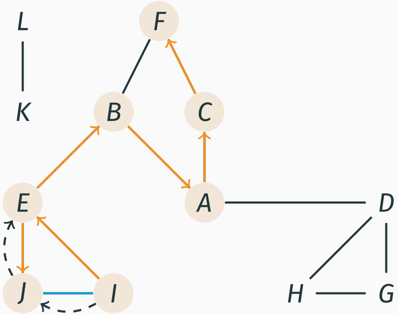
How do you traverse a maze?



RECURSIVEDFS

if v is unmarked **then**
mark v
foreach edge vw **do**
Recursively traverse starting from v

How do you traverse a maze?

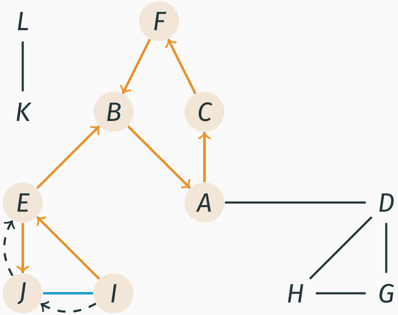


RecursiveDFS

if v is unmarked **then**
  mark v
  foreach edge vw **do**
    Recursively traverse
     starting from v

Depth-First Search

How do you traverse a maze?

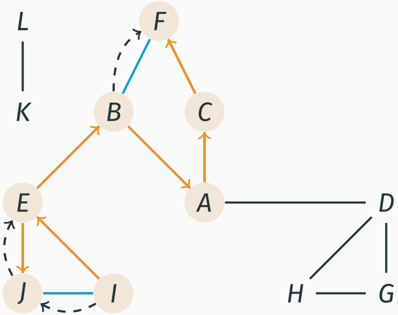


RecursiveDFS

if *v* is unmarked **then**
  mark *v*
  foreach edge *vw* **do**
    Recursively traverse
     starting from *v*

Depth-First Search

How do you traverse a maze?

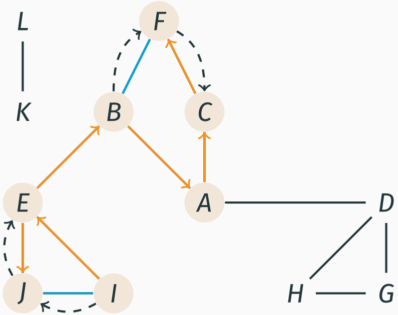


RECURSIVEDFS

if v is unmarked **then**
mark v
foreach edge vw **do**
Recursively traverse starting from v

Depth-First Search

How do you traverse a maze?

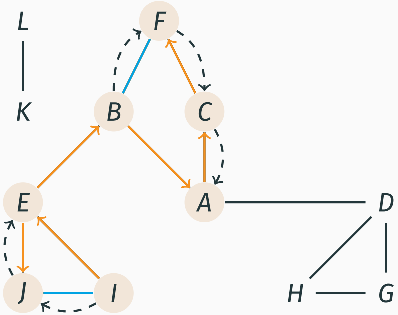


RECURSIVEDFS

if *v* is unmarked **then**
    mark *v*
    foreach edge *vw* **do**
        Recursively traverse
        starting from *v*

Depth-First Search

How do you traverse a maze?

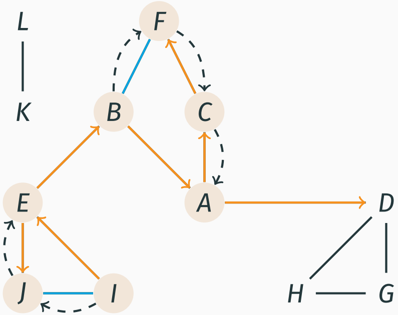


RecursiveDFS

if *v* is unmarked **then**
  mark *v*
  foreach edge *vw* **do**
    Recursively traverse
    starting from *v*

Depth-First Search

How do you traverse a maze?

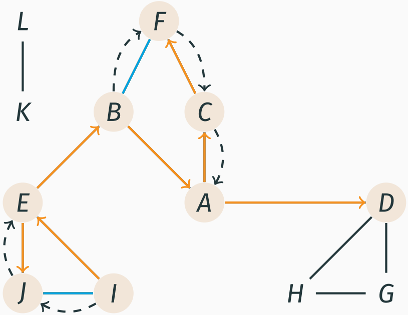


RecursiveDFS

if v is unmarked **then**
  mark v
  foreach edge vw **do**
    Recursively traverse
    starting from v

Depth-First Search

How do you traverse a maze?

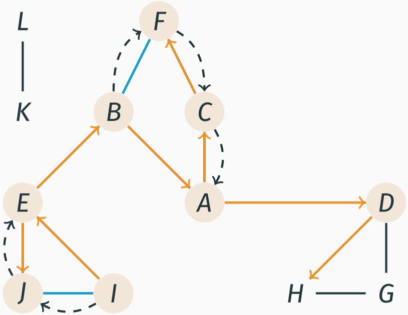


RecursiveDFS

if v is unmarked **then**
  mark v
  foreach edge vw **do**
    Recursively traverse
     starting from v

Depth-First Search

How do you traverse a maze?

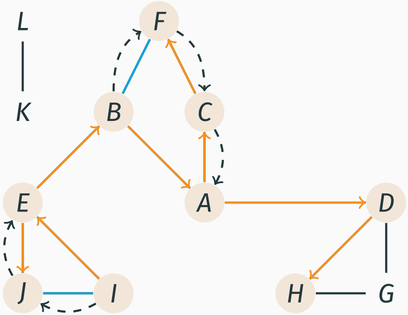


RecursiveDFS

if v is unmarked **then**
  mark v
  foreach edge vw **do**
    Recursively traverse
     starting from v

Depth-First Search

How do you traverse a maze?

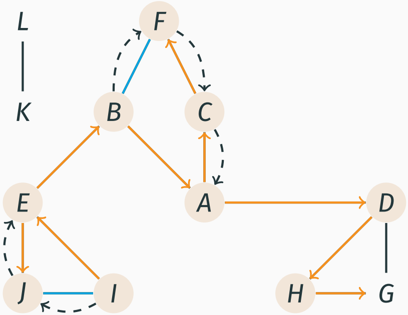


RecursiveDFS

if v is unmarked **then**
    mark v
    foreach edge vw **do**
        Recursively traverse
        starting from v

Depth-First Search

How do you traverse a maze?

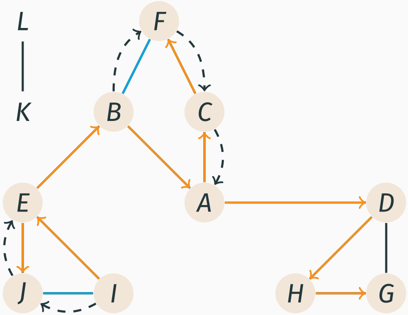


RecursiveDFS

if *v* is unmarked **then**
  mark *v*
  foreach edge *vw* **do**
    Recursively traverse
     starting from *v*

Depth-First Search

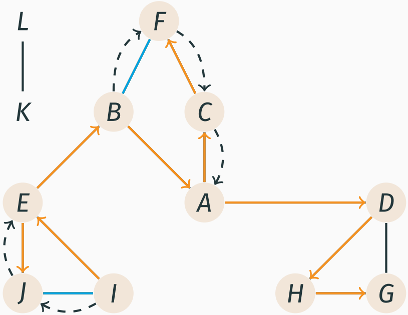
How do you traverse a maze?



RecursiveDFS

if v is unmarked **then**
  mark v
  foreach edge vw **do**
    Recursively traverse
     starting from v

How do you traverse a maze?

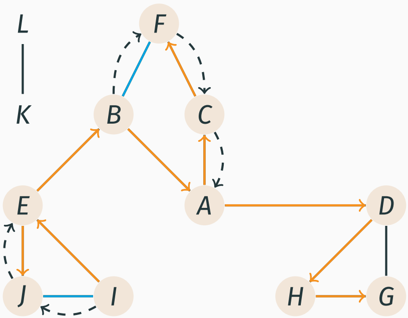


ITERATIVEDFS

push(s)
while stack is not empty **do**
  $v \leftarrow$ pop()
  if v is unmarked **then**
    mark v
    foreach edge vw **do**
      push(w)

Depth-First Search

How do you traverse a maze?

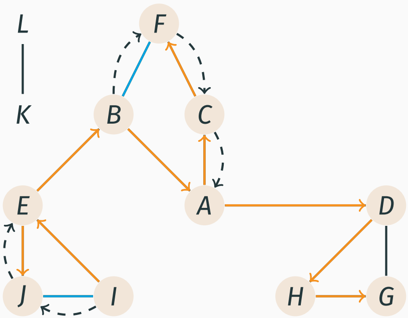


IterativeDFS

push($\varnothing, s$)
while stack is not empty **do**
    $(p, v) \leftarrow$ pop()
    if v is unmarked **then**
        mark v
        parent(v) $\leftarrow p$
        foreach edge vw **do**
            push(v, w)

Depth-First Search

How do you traverse a maze?



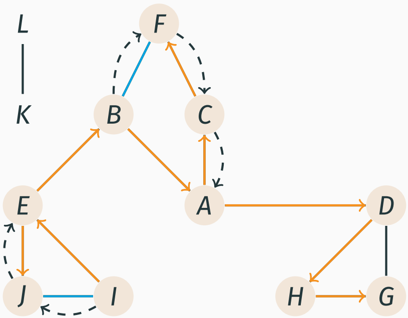
IterativeDFS

push($\varnothing, s$)
while stack is not empty **do**
    $(p, v) \leftarrow$ pop()
    if v is unmarked **then**
        mark v ⟶ $O(1)$/vertex
        parent(v) $\leftarrow p$
        foreach edge vw **do**
            push(v, w)

$O(d(v))$/vertex

Depth-First Search

How do you traverse a maze?



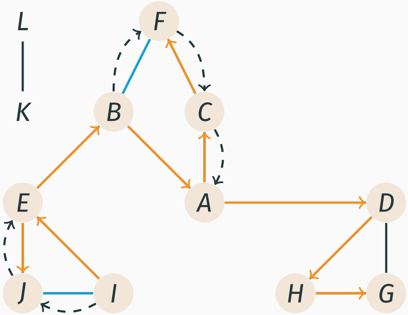
IterativeDFS

push($\varnothing, s$)
while stack is not empty **do**
    $(p, v) \leftarrow$ pop()
    if v is unmarked **then**
        mark v ⟶ $O(1)$/vertex
        parent(v) $\leftarrow p$
        foreach edge vw **do**
            push(v, w)

$O(d(v))$/vertex

Running time $O(|V| + |E|)$ in adjacency list model

How do you traverse a maze?



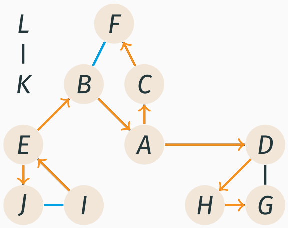
ITERATIVEDFS

push($\varnothing$, s)
while stack is not empty **do**
  (p, v) ← pop()
  if v is unmarked **then**
    mark v ⟶ $O(1)$/vertex
    parent(v) ← p
    foreach edge vw **do**
      push(v, w) ⟶ $O(d(v))$/vertex

Running time $O(|V| + |E|)$ in adjacency list model

Question What is the running time in the adjacency matrix model?

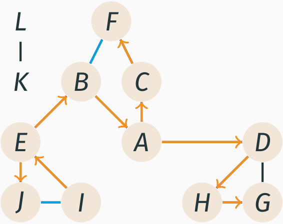
Depth-First Search



Lemma: The following statements are true for DFS:

1. DFS marks those vertices reachable from s and only those vertices.
2. The set of pairs $(v, \text{parent}(v))$ defines a spanning tree of the component containing s.

Depth-First Search

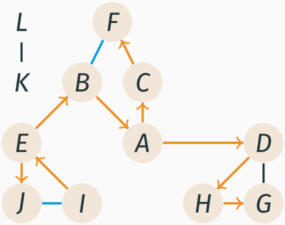


Lemma: The following statements are true for DFS:

1. DFS marks those vertices reachable from s and only those vertices.
2. The set of pairs $(v, \text{parent}(v))$ defines a spanning tree of the component containing s.

Proof of (1): Induction on the length of the shortest path

Depth-First Search



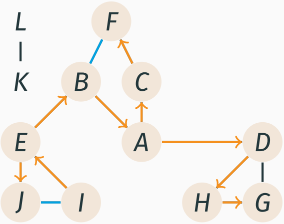
Lemma: The following statements are true for DFS:

1. DFS marks those vertices reachable from s and only those vertices.
2. The set of pairs $(v, \text{parent}(v))$ defines a spanning tree of the component containing s.

Proof of (1): Induction on the length of the shortest path

Let $s - u_1 - u_2 - \ldots - u\ - v$ be a shortest path from s to v

Depth-First Search



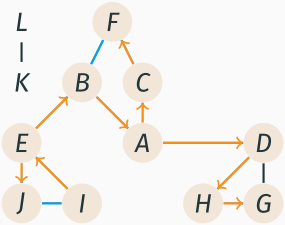
Lemma: The following statements are true for DFS:

1. DFS marks those vertices reachable from s and only those vertices.
2. The set of pairs $(v, \text{parent}(v))$ defines a spanning tree of the component containing s.

Proof of (1): Induction on the length of the shortest path

Let $s - u_1 - u_2 - \ldots - u$ $-v$ be a shortest path from s to v

Depth-First Search



Lemma: The following statements are true for DFS:

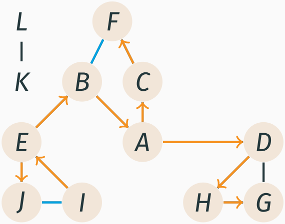
1. DFS marks those vertices reachable from s and only those vertices.
2. The set of pairs $(v, \text{parent}(v))$ defines a spanning tree of the component containing s.

Proof of (1): Induction on the length of the shortest path

Let $s - u_1 - u_2 - \ldots - u$ $-v$ be a shortest path from s to v

By the induction hypothesis, u is marked by the DFS

Depth-First Search



Lemma: The following statements are true for DFS:

1. DFS marks those vertices reachable from s and only those vertices.
2. The set of pairs $(v, \text{parent}(v))$ defines a spanning tree of the component containing s.

Proof of (1): Induction on the length of the shortest path

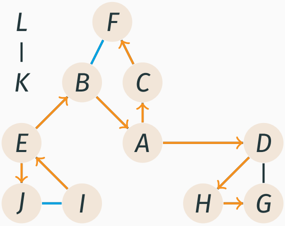
Let $s - u_1 - u_2 - \ldots - u$ $-v$ be a shortest path from s to v

By the induction hypothesis, u is marked by the DFS

If u is marked, then all the neighbors of u are placed in the stack. Therefore when they are taken out of the stack, either

- They are already marked, or
- They are unmarked and will get marked at that stage

Depth-First Search

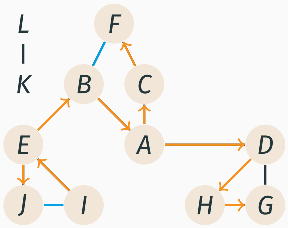


Lemma: The following statements are true for DFS:

1. DFS marks those vertices reachable from s and only those vertices.
2. The set of pairs $(v, \text{parent}(v))$ defines a spanning tree of the component containing s.

Proof of (2): Induction on the time at which v is marked

Depth-First Search

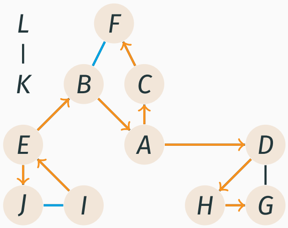


Lemma: The following statements are true for DFS:

1. DFS marks those vertices reachable from s and only those vertices.
2. The set of pairs $(v, \text{parent}(v))$ defines a spanning tree of the component containing s.

Proof of (2): Induction on the time at which v is marked
If $w = \text{parent}(v)$, then w is marked before v - There must be a path from w to s and hence from v to s

Depth-First Search



Lemma: The following statements are true for DFS:

1. DFS marks those vertices reachable from s and only those vertices.
2. The set of pairs $(v, \text{parent}(v))$ defines a spanning tree of the component containing s.

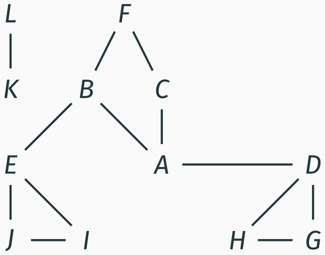
Proof of (2): Induction on the time at which v is marked
If $w = \text{parent}(v)$, then w is marked before v - There must be a path from w to s and hence from v to s
Every vertex v has a unique parent (except s that has no parent), and hence the subgraph has $n - 1$ edges $\Rightarrow$ spanning tree

Breadth-First Search

How do you find the shortest path in a maze?

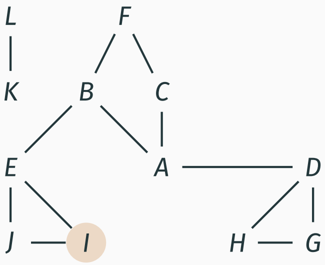
Breadth-First Search

How do you find the shortest path in a maze?



BFS
enqueue($\varnothing, s$)
while queue is not empty **do**
    (p, v) ← dequeue()
    if v is umarked **then**
        mark v
        parent(v) ← p
        foreach edge vw **do**
            enqueue(v, w)

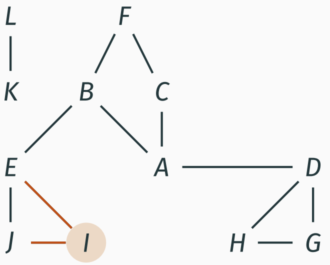
How do you find the shortest path in a maze?



BFS
enqueue($\varnothing, s$)
while queue is not empty **do**
  (p, v) ← dequeue()
  if v is umarked **then**
    mark v
    parent(v) ← p
    foreach edge vw **do**
      enqueue(v, w)

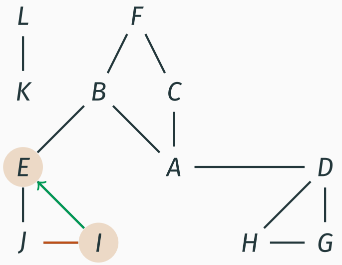
Breadth-First Search

How do you find the shortest path in a maze?



BFS
enqueue($\varnothing, s$)
while queue is not empty **do**
  $(p, v) \leftarrow$ dequeue()
  if v is umarked **then**
    mark v
    parent(v) $\leftarrow p$
    foreach edge vw **do**
      enqueue(v, w)

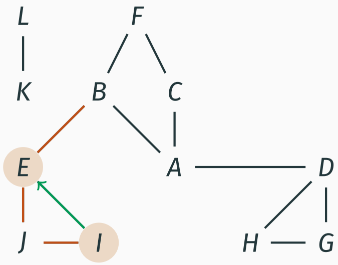
How do you find the shortest path in a maze?



BFS
enqueue($\varnothing, s$)
while queue is not empty **do**
    $(p, v) \leftarrow$ dequeue()
    if v is umarked **then**
        mark v
        parent(v) $\leftarrow p$
        foreach edge vw **do**
            enqueue(v, w)

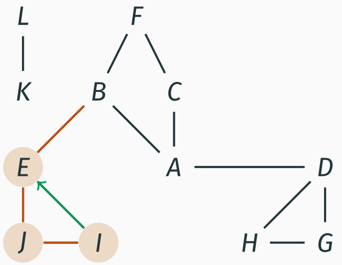
Breadth-First Search

How do you find the shortest path in a maze?



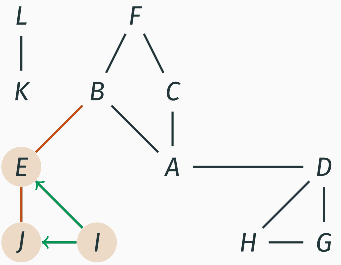
BFS
enqueue($\varnothing, s$)
while queue is not empty **do**
    (p, v) ← dequeue()
    if v is umarked **then**
        mark v
        parent(v) ← p
        foreach edge vw **do**
            enqueue(v, w)

How do you find the shortest path in a maze?



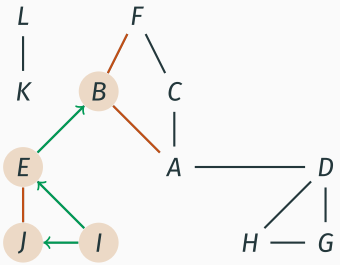
BFS
enqueue($\varnothing, s$)
while queue is not empty **do**
  (p, v) ← dequeue()
  if v is umarked **then**
    mark v
    parent(v) ← p
    foreach edge vw **do**
      enqueue(v, w)

How do you find the shortest path in a maze?



BFS
enqueue($\varnothing, s$)
while queue is not empty **do**
    (p, v) ← dequeue()
    if v is umarked **then**
        mark v
        parent(v) ← p
        foreach edge vw **do**
            enqueue(v, w)

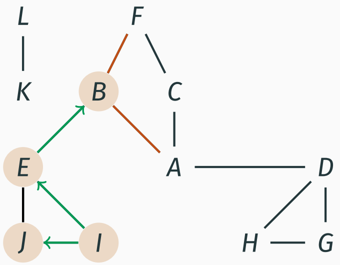
How do you find the shortest path in a maze?



BFS
enqueue($\varnothing, s$)
while queue is not empty **do**
    $(p, v) \leftarrow$ dequeue()
    if v is umarked **then**
        mark v
        parent(v) $\leftarrow p$
        foreach edge vw **do**
            enqueue(v, w)

Breadth-First Search

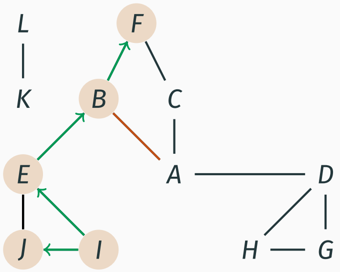
How do you find the shortest path in a maze?



BFS
enqueue($\varnothing, s$)
while queue is not empty **do**
  $(p, v) \leftarrow$ dequeue()
  if v is umarked **then**
    mark v
    parent(v) $\leftarrow p$
    foreach edge vw **do**
      enqueue(v, w)

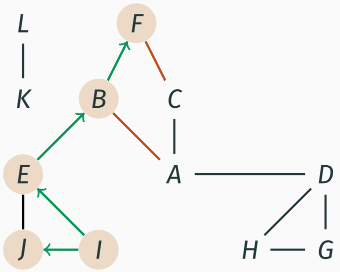
Breadth-First Search

How do you find the shortest path in a maze?



BFS
enqueue($\varnothing, s$)
while queue is not empty **do**
  $(p, v) \leftarrow$ dequeue()
  if v is umarked **then**
    mark v
    parent(v) $\leftarrow p$
    foreach edge vw **do**
      enqueue(v, w)

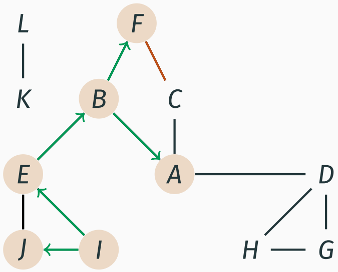
How do you find the shortest path in a maze?



BFS
enqueue(∅, s)
while queue is not empty **do**
  (p, v) ← dequeue()
  if v is umarked **then**
    mark v
    parent(v) ← p
    foreach edge vw **do**
      enqueue(v, w)

Breadth-First Search

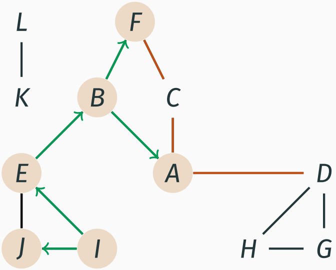
How do you find the shortest path in a maze?



BFS
enqueue($\varnothing, s$)
while queue is not empty **do**
    $(p, v) \leftarrow$ dequeue()
    if v is umarked **then**
        mark v
        parent(v) $\leftarrow p$
        foreach edge vw **do**
            enqueue(v, w)

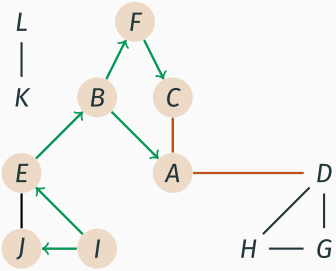
Breadth-First Search

How do you find the shortest path in a maze?



BFS
enqueue($\varnothing, s$)
while queue is not empty **do**
  (p, v) ← dequeue()
  if v is umarked **then**
    mark v
    parent(v) ← p
    foreach edge vw **do**
      enqueue(v, w)

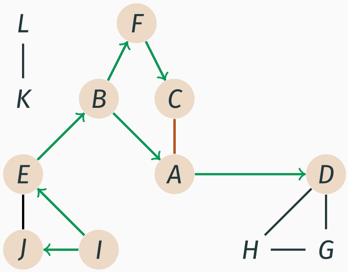
How do you find the shortest path in a maze?



BFS
enqueue($\varnothing, s$)
while queue is not empty **do**
    $(p, v) \leftarrow$ dequeue()
    if v is umarked **then**
        mark v
        parent(v) $\leftarrow p$
        foreach edge vw **do**
            enqueue(v, w)

Breadth-First Search

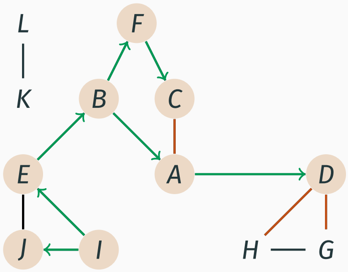
How do you find the shortest path in a maze?



BFS
enqueue($\varnothing, s$)
while queue is not empty **do**
    $(p, v) \leftarrow$ dequeue()
    if v is umarked **then**
        mark v
        parent(v) $\leftarrow p$
        foreach edge vw **do**
            enqueue(v, w)

Breadth-First Search

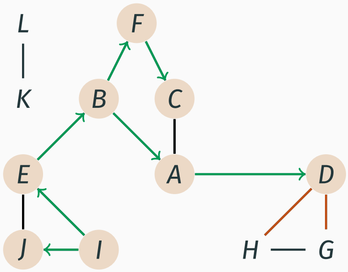
How do you find the shortest path in a maze?



BFS
enqueue($\varnothing, s$)
while queue is not empty **do**
  $(p, v) \leftarrow$ dequeue()
  if v is umarked **then**
    mark v
    parent(v) $\leftarrow p$
    foreach edge vw **do**
      enqueue(v, w)

Breadth-First Search

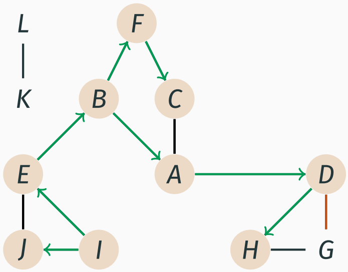
How do you find the shortest path in a maze?



BFS
enqueue($\varnothing, s$)
while queue is not empty **do**
    (p, v) ← dequeue()
    if v is umarked **then**
        mark v
        parent(v) ← p
        foreach edge vw **do**
            enqueue(v, w)

Breadth-First Search

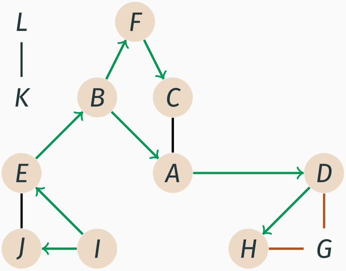
How do you find the shortest path in a maze?



BFS
enqueue($\varnothing, s$)
while queue is not empty **do**
    $(p, v) \leftarrow$ dequeue()
    if v is umarked **then**
        mark v
        parent(v) $\leftarrow p$
        foreach edge vw **do**
            enqueue(v, w)

Breadth-First Search

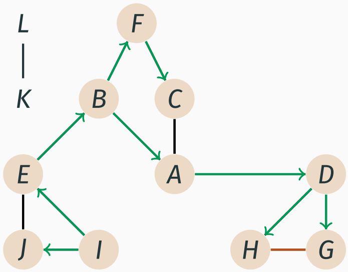
How do you find the shortest path in a maze?



BFS
enqueue($\emptyset, s$)
while queue is not empty **do**
    $(p, v) \leftarrow$ dequeue()
    if v is umarked **then**
        mark v
        parent(v) $\leftarrow p$
        foreach edge vw **do**
            enqueue(v, w)

Breadth-First Search

How do you find the shortest path in a maze?



BFS
enqueue($\emptyset, s$)
while queue is not empty **do**
    $(p, v) \leftarrow$ dequeue()
    if v is umarked **then**
        mark v
        parent(v) $\leftarrow p$
        foreach edge vw **do**
            enqueue(v, w)

DFS vs BFS

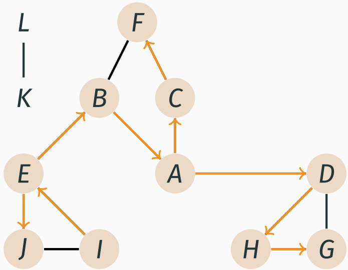


Figure 1: DFS tree

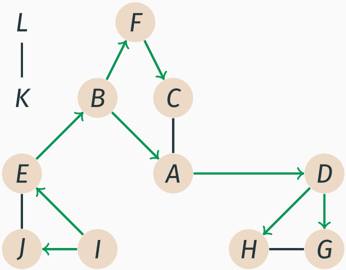


Figure 2: BFS tree

Applications: Connected compenents

Question: How many connected components are there in the graph?

Applications: Connected compenents

Question: How many connected components are there in the graph?

- DFS/BFS starting from a vertex s visits those and only those vertices that lie in the same connected component as s

Applications: Connected compenents

Question: How many connected components are there in the graph?

- DFS/BFS starting from a vertex s visits those and only those vertices that lie in the same connected component as s

DFSALL(G)

foreach vertex $v \in G$ **do**
    unmark v

foreach vertex $v \in G$ **do**
    if v is unmarked **then**

        DFS(v)

DFS(s)

push(s)
while stack is not empty **do**
    $v \leftarrow$ pop()
    if v is unmarked **then**
        mark v

        foreach edge vw **do**
            push(w)

Applications: Connected compenents

Question: How many connected components are there in the graph?

- DFS/BFS starting from a vertex s visits those and only those vertices that lie in the same connected component as s

DFSALL(G)

```
foreach vertex v ∈ G do
    unmark v
cc ← 0
foreach vertex v ∈ G do
    if v is unmarked then
        cc ← cc + 1
        DFS(v)
```

DFS(s)

```
push(s)
while stack is not empty do
    v ← pop()
    if v is unmarked then
        mark v

        foreach edge vw do
            push(w)
```

Applications: Connected compenents

Question: How many connected components are there in the graph?

- DFS/BFS starting from a vertex s visits those and only those vertices that lie in the same connected component as s

DFSAll(G)

foreach vertex $v \in G$ **do**
  unmark v
$cc \leftarrow 0$
foreach vertex $v \in G$ **do**
  if v is unmarked **then**
    $cc \leftarrow cc + 1$
    DFS(v)

unmarked vertex means new component

DFS(s)

push(s)
while stack is not empty **do**
  $v \leftarrow$ pop()
  if v is unmarked **then**
    mark v

    foreach edge vw **do**
      push(w)

Applications: Connected compenents

Question: How many connected components are there in the graph?

- DFS/BFS starting from a vertex s visits those and only those vertices that lie in the same connected component as s

```
DFSALL(G)
foreach vertex v ∈ G do
    unmark v
cc ← 0
foreach vertex v ∈ G do
    if v is unmarked then
        cc ← cc + 1
        DFS(v)
```

unmarked vertex means new component

```
DFS(s)
push(s)
while stack is not empty do
    v ← pop()
    if v is unmarked then
        mark v
        comp[v] = cc
        foreach edge vw do
            push(w)
```

Applications: Connected compenents

Question: How many connected components are there in the graph?

- DFS/BFS starting from a vertex s visits those and only those vertices that lie in the same connected component as s

DFSALL(G)

foreach vertex $v \in G$ **do**
  unmark v
$cc \leftarrow 0$
foreach vertex $v \in G$ **do**
  if v is unmarked **then**
    $cc \leftarrow cc + 1$
    DFS(v)

unmarked vertex means new component

DFS(s)

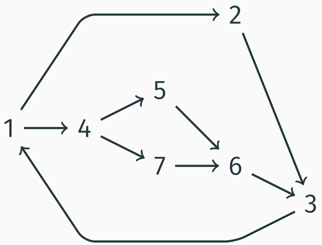
push(s)
while stack is not empty **do**
  $v \leftarrow$ pop()
  if v is unmarked **then**
    mark v
    $comp[v] = cc$ $\longrightarrow$ component label
    foreach edge vw **do**
      push(w)

Directed graphs

Directed graphs: Basics

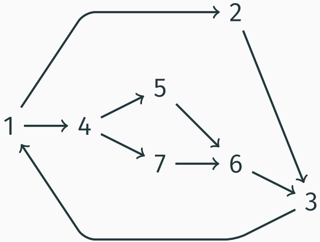
Directed graphs: Basics

- $G(V, E)$ – $E \subseteq V \times V$: Implicitly assumed a symmetric relation earlier



Directed graphs: Basics

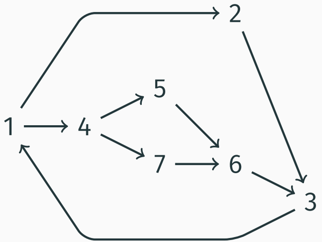
- $G(V, E)$ – $E \subseteq V \times V$: Implicitly assumed a symmetric relation earlier



- If you get pairwise movie preferences for a person, can you rank-order their favourite movies?

Directed graphs: Basics

- $G(V, E)$ – $E \subseteq V \times V$: Implicitly assumed a symmetric relation earlier



- If you get pairwise movie preferences for a person, can you rank-order their favourite movies?
    - Pairwise preferences - directed edges
    - Rank-ordering - Hamiltonian path

DFS in directed graphs: Classification of edges

DFS in directed graphs: Classification of edges

DFSAll(G)

foreach vertex $v \in G$ **do**
  unmark v
foreach vertex $v \in G$ **do**
  if v is unmarked **then**
    DFS(v)

DFS(s)

mark s
foreach edge $s \to w$ **do**
  if w is unmarked **then**
    parent[w] $\leftarrow s$
    DFS(w)

DFS in directed graphs: Classification of edges

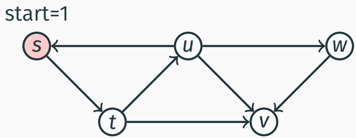
DFSALL(G)
clock ← 0
foreach vertex $v \in G$ **do**
    unmark v
foreach vertex $v \in G$ **do**
    if v is unmarked **then**
        DFS(v, clock)

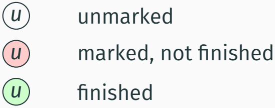
DFS(s, clock)
mark s
clock ← clock+1
start[s] ← clock
foreach edge $s \to w$ **do**
    if w is unmarked **then**
        parent[w] ← s
        clock ← DFS(w, clock)
clock ← clock+1
end[s] ← clock
return clock

DFS in directed graphs: Classification of edges

DFSALL(G)
clock ← 0
foreach vertex $v \in G$ **do**
  unmark v
foreach vertex $v \in G$ **do**
  if v is unmarked **then**
    DFS(v, clock)

DFS(s, clock)
mark s
clock ← clock+1
start[s] ← clock
foreach edge $s \to w$ **do**
  if w is unmarked **then**
    parent[w] ← s
    clock ← DFS(w, clock)
clock ← clock+1
end[s] ← clock
return clock

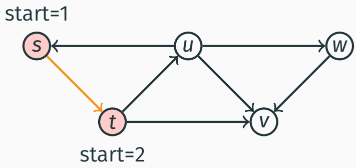


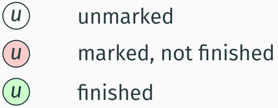


DFS in directed graphs: Classification of edges

DFSAll(G)
clock ← 0
foreach vertex $v \in G$ **do**
    unmark v
foreach vertex $v \in G$ **do**
    if v is unmarked **then**
        DFS(v, clock)

DFS(s, clock)
mark s
clock ← clock+1
start[s] ← clock
foreach edge $s \to w$ **do**
    if w is unmarked **then**
        parent[w] ← s
        clock ← DFS(w, clock)
clock ← clock+1
end[s] ← clock
return clock

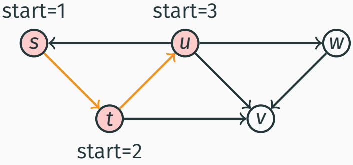


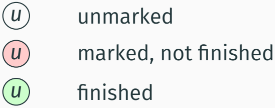


DFS in directed graphs: Classification of edges

DFSALL(G)
clock ← 0
foreach vertex $v \in G$ **do**
    unmark v
foreach vertex $v \in G$ **do**
    if v is unmarked **then**
        DFS(v, clock)

DFS(s, clock)
mark s
clock ← clock+1
start[s] ← clock
foreach edge $s \to w$ **do**
    if w is unmarked **then**
        parent[w] ← s
        clock ← DFS(w, clock)
clock ← clock+1
end[s] ← clock
return clock

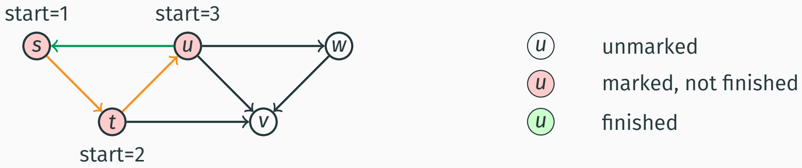




DFS in directed graphs: Classification of edges

DFSALL(G)
clock ← 0
foreach vertex $v \in G$ **do**
    unmark v
foreach vertex $v \in G$ **do**
    if v is unmarked **then**
        DFS(v, clock)

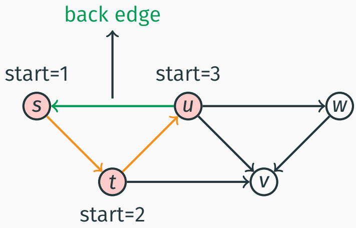
DFS(s, clock)
mark s
clock ← clock+1
start[s] ← clock
foreach edge $s \to w$ **do**
    if w is unmarked **then**
        parent[w] ← s
        clock ← DFS(w, clock)
clock ← clock+1
end[s] ← clock
return clock

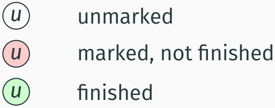


DFS in directed graphs: Classification of edges

DFSALL(G)
clock ← 0
foreach vertex $v \in G$ **do**
    unmark v
foreach vertex $v \in G$ **do**
    if v is unmarked **then**
        DFS(v, clock)

DFS(s, clock)
mark s
clock ← clock+1
start[s] ← clock
foreach edge $s \to w$ **do**
    if w is unmarked **then**
        parent[w] ← s
        clock ← DFS(w, clock)
clock ← clock+1
end[s] ← clock
return clock





DFS in directed graphs: Classification of edges

DFSAll(G)

clock ← 0

foreach vertex $v \in G$ **do**

  unmark v

foreach vertex $v \in G$ **do**

  if v is unmarked **then**

    DFS(v, clock)

DFS(s, clock)

mark s

clock ← clock+1

start[s] ← clock

foreach edge $s \to w$ **do**

  if w is unmarked **then**

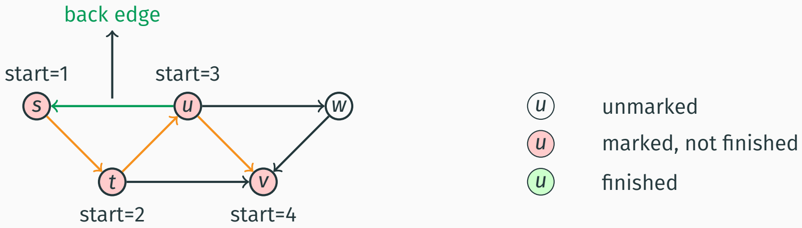
    parent[w] ← s

    clock ← DFS(w, clock)

clock ← clock+1

end[s] ← clock

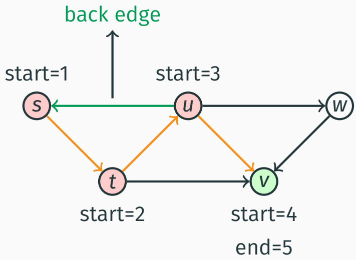
return clock

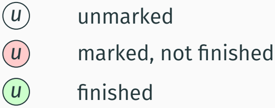


DFS in directed graphs: Classification of edges

DFSALL(G)
clock ← 0
foreach vertex $v \in G$ **do**
    unmark v
foreach vertex $v \in G$ **do**
    if v is unmarked **then**
        DFS(v, clock)

DFS(s, clock)
mark s
clock ← clock+1
start[s] ← clock
foreach edge $s \to w$ **do**
    if w is unmarked **then**
        parent[w] ← s
        clock ← DFS(w, clock)
clock ← clock+1
end[s] ← clock
return clock

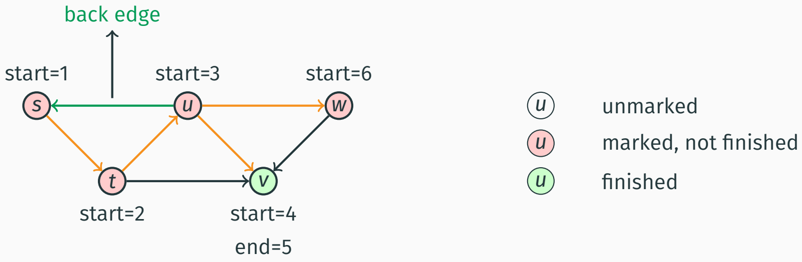




DFS in directed graphs: Classification of edges

DFSAll(G)
clock ← 0
foreach vertex $v \in G$ **do**
  unmark v
foreach vertex $v \in G$ **do**
  if v is unmarked **then**
    DFS(v, clock)

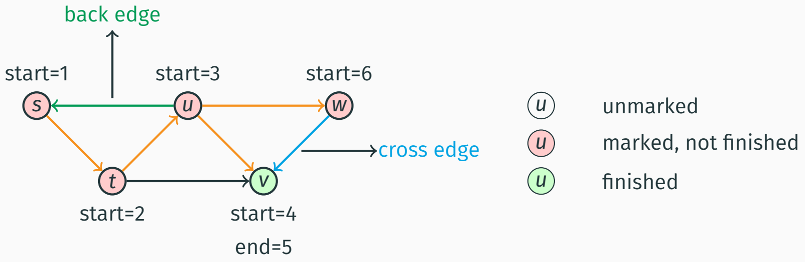
DFS(s, clock)
mark s
clock ← clock+1
start[s] ← clock
foreach edge $s \to w$ **do**
  if w is unmarked **then**
    parent[w] ← s
    clock ← DFS(w, clock)
clock ← clock+1
end[s] ← clock
return clock



DFS in directed graphs: Classification of edges

DFS(s, clock)
mark s
clock ← clock+1
start[s] ← clock
foreach edge $s \to w$ **do**
  if w is unmarked **then**
    parent[w] ← s
    clock ← DFS(w, clock)
clock ← clock+1
end[s] ← clock
return clock

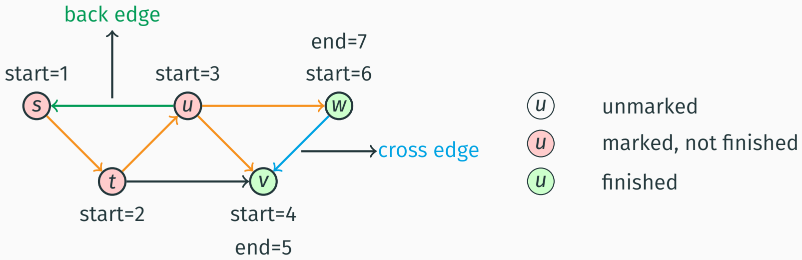
DFSAll(G)
clock ← 0
foreach vertex $v \in G$ **do**
  unmark v
foreach vertex $v \in G$ **do**
  if v is unmarked **then**
    DFS(v, clock)



DFS in directed graphs: Classification of edges

DFSALL(G)
clock ← 0
foreach vertex $v \in G$ **do**
    unmark v
foreach vertex $v \in G$ **do**
    if v is unmarked **then**
        DFS(v, clock)

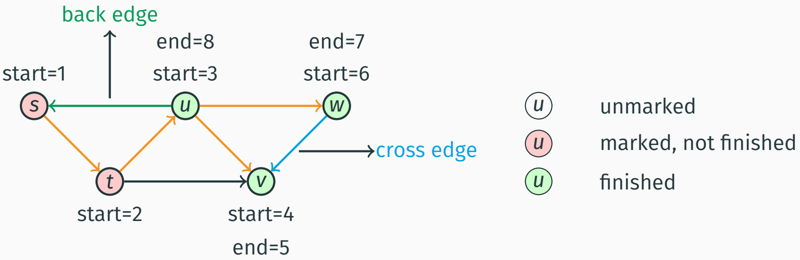
DFS(s, clock)
mark s
clock ← clock+1
start[s] ← clock
foreach edge $s \to w$ **do**
    if w is unmarked **then**
        parent[w] ← s
        clock ← DFS(w, clock)
clock ← clock+1
end[s] ← clock
return clock



DFS in directed graphs: Classification of edges

DFSALL(G)
clock ← 0
foreach vertex $v \in G$ **do**
    unmark v
foreach vertex $v \in G$ **do**
    if v is unmarked **then**
        DFS(v, clock)

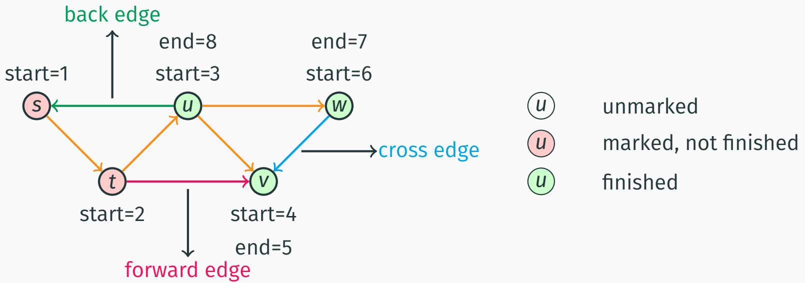
DFS(s, clock)
mark s
clock ← clock+1
start[s] ← clock
foreach edge $s \to w$ **do**
    if w is unmarked **then**
        parent[w] ← s
        clock ← DFS(w, clock)
clock ← clock+1
end[s] ← clock
return clock



DFS in directed graphs: Classification of edges

DFSALL(G)
clock ← 0
foreach vertex $v \in G$ **do**
    unmark v
foreach vertex $v \in G$ **do**
    if v is unmarked **then**
        DFS(v, clock)

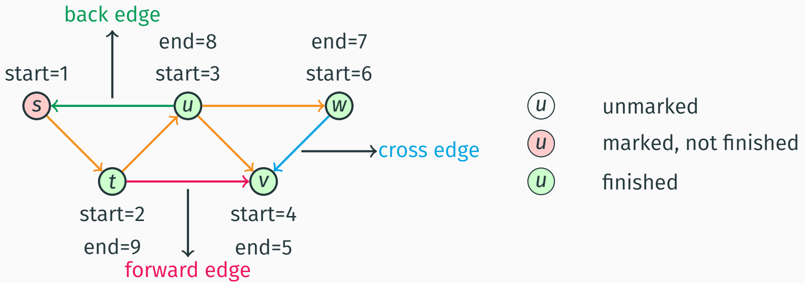
DFS(s, clock)
mark s
clock ← clock+1
start[s] ← clock
foreach edge $s \to w$ **do**
    if w is unmarked **then**
        parent[w] ← s
        clock ← DFS(w, clock)
clock ← clock+1
end[s] ← clock
return clock



DFS in directed graphs: Classification of edges

DFSALL(G)
clock ← 0
foreach vertex $v \in G$ **do**
  unmark v
foreach vertex $v \in G$ **do**
  if v is unmarked **then**
    DFS(v, clock)

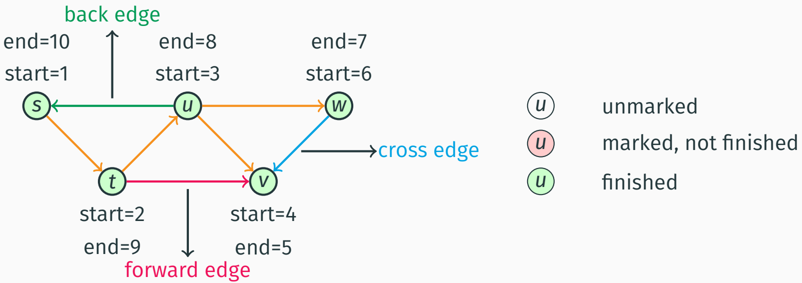
DFS(s, clock)
mark s
clock ← clock+1
start[s] ← clock
foreach edge $s \to w$ **do**
  if w is unmarked **then**
    parent[w] ← s
    clock ← DFS(w, clock)
clock ← clock+1
end[s] ← clock
return clock



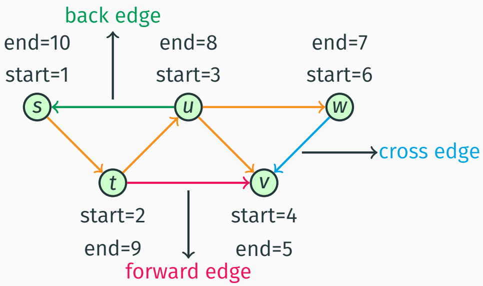
DFS in directed graphs: Classification of edges

DFSALL(G)
clock ← 0
foreach vertex $v \in G$ **do**
    unmark v
foreach vertex $v \in G$ **do**
    if v is unmarked **then**
        DFS(v, clock)

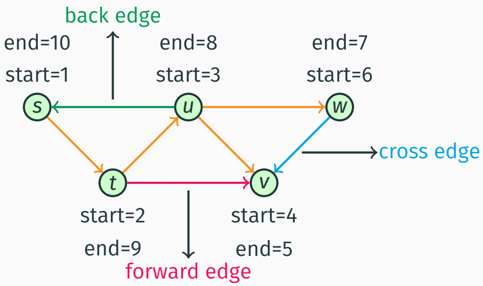
DFS(s, clock)
mark s
clock ← clock+1
start[s] ← clock
foreach edge $s \to w$ **do**
    if w is unmarked **then**
        parent[w] ← s
        clock ← DFS(w, clock)
clock ← clock+1
end[s] ← clock
return clock



DFS in directed graphs: Classification of edges

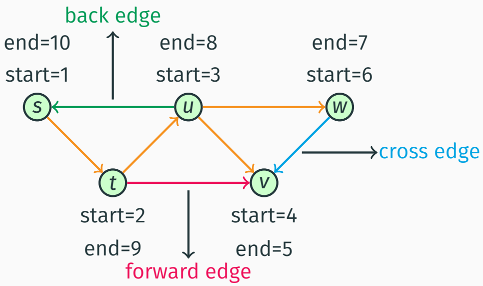


DFS in directed graphs: Classification of edges



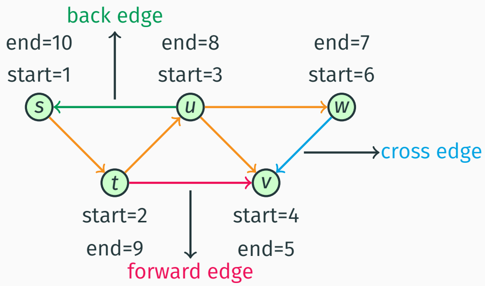
- **Tree edge:** start[s] < start[t] < end[t] < end[s], and DFS(t) is directly called by DFS(s)

DFS in directed graphs: Classification of edges



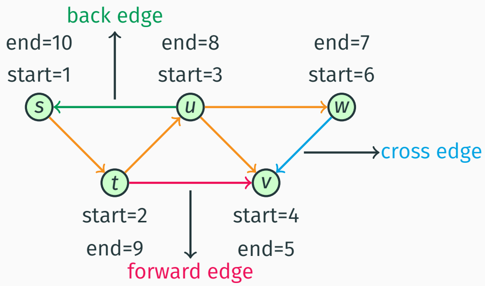
- **Tree edge:** start[s] < start[t] < end[t] < end[s], and DFS(t) is directly called by DFS(s)
- **Forward edge:** start[t] < start[v] < end[v] < end[t], but DFS(v) is not directly called by DFS(t)

DFS in directed graphs: Classification of edges



- **Tree edge:** start[s] < start[t] < end[t] < end[s], and DFS(t) is directly called by DFS(s)
- **Forward edge:** start[t] < start[v] < end[v] < end[t], but DFS(v) is not directly called by DFS(t)
- **Back edge:** start[s] < start[u] < end[u] < end[s], and there is a directed path from s to u

DFS in directed graphs: Classification of edges



- **Tree edge:** start[s] < start[t] < end[t] < end[s], and DFS(t) is directly called by DFS(s)
- **Forward edge:** start[t] < start[v] < end[v] < end[t], but DFS(v) is not directly called by DFS(t)
- **Back edge:** start[s] < start[u] < end[u] < end[s], and there is a directed path from s to u
- **Cross edge:** v is finished when DFS(w) starts - end[v] < start[w]

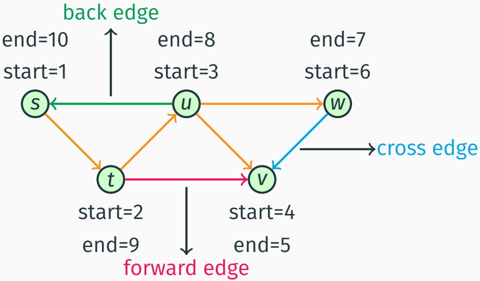
Cycles in directed graphs

Cycles in directed graphs

When does a digraph have a cycle?

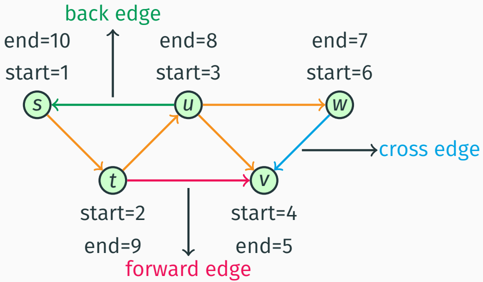
Cycles in directed graphs

When does a digraph have a cycle?



Cycles in directed graphs

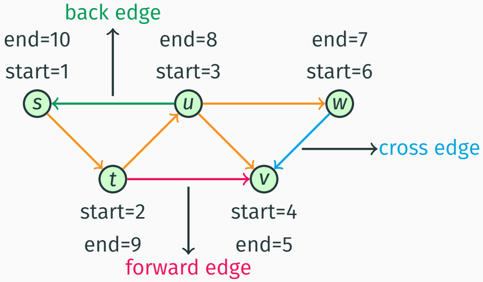
When does a digraph have a cycle?



- There must a back-edge in the DFS traversal

Cycles in directed graphs

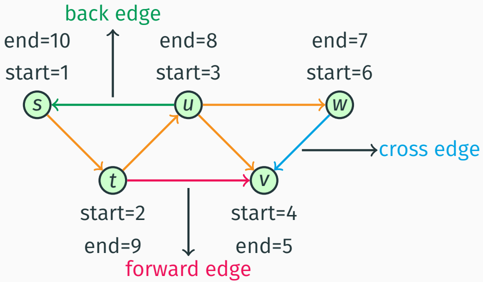
When does a digraph have a cycle?



- There must a back-edge in the DFS traversal
- There is an edge $u \to v$ such that end[u] < end[v]

Cycles in directed graphs

When does a digraph have a cycle?



- There must a back-edge in the DFS traversal
- There is an edge $u \to v$ such that end[u] < end[v]

Exercise: Write the pseudo-code for the $O(|V| + |E|)$ algorithm to check if G is a DAG (Directed Acyclic Graph)?

Recursion and DAGs

$\textsc{Fib}(n)$

if $n = 1$ or 2 **then return** 1
return $\textsc{Fib}(n-1) + \textsc{Fib}(n-2)$

Recursion and DAGs

Dependency digraph

FIB(n)

if n = 1 or 2 **then return** 1
return FIB(n – 1) + FIB(n – 2)

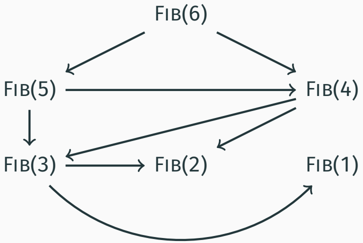
$$
\begin{tikzcd}
 & \textsc{Fib}(6) \arrow[dl] \arrow[dr] & \\
\textsc{Fib}(5) \arrow[rr] \arrow[d] & & \textsc{Fib}(4) \arrow[dll] \arrow[dl] \\
\textsc{Fib}(3) \arrow[r] \arrow[rr, bend right] & \textsc{Fib}(2) & \textsc{Fib}(1)
\end{tikzcd}
$$

Recursion and DAGs

Dependency digraph

FIB(n)

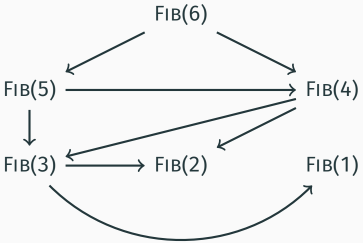
if $n = 1$ or 2 **then return** 1
return FIB($n-1$) + FIB($n-2$)



- Recursive function calls can be simulated by a DFS of the dependency graph – memoization

Recursion and DAGs

Dependency digraph



FIB(n)

if n = 1 or 2 **then return** 1
return FIB(n − 1) + FIB(n − 2)

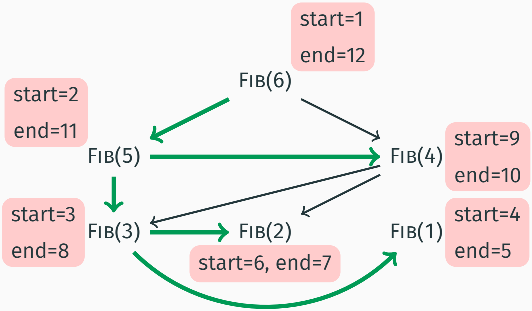
- Recursive function calls can be simulated by a DFS of the dependency graph – memoization
- What is the order in which the recursive calls are made?

Recursion and DAGs

FIB(n)

if n = 1 or 2 **then return** 1
return FIB(n − 1) + FIB(n − 2)

Dependency digraph

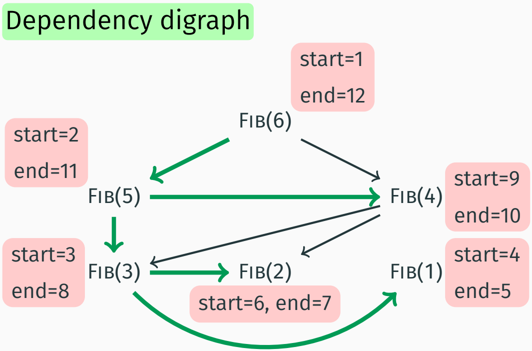


- Recursive function calls can be simulated by a DFS of the dependency graph – memoization
- What is the order in which the recursive calls are made?
  - DFS defines an order on the vertices of the graph

Recursion and DAGs

Fɪʙ(n)

if n = 1 or 2 **then return** 1
return Fɪʙ(n – 1) + Fɪʙ(n – 2)



- Recursive function calls can be simulated by a DFS of the dependency graph – memoization
- What is the order in which the recursive calls are made?
    - DFS defines an order on the vertices of the graph
      Fɪʙ(6) → Fɪʙ(5) → Fɪʙ(4) → Fɪʙ(3) → Fɪʙ(2) → Fɪʙ(1)

Topological ordering of DAGs

$$\begin{tikzcd}
\textsc{Fib}_6 \arrow[r] \arrow[rr, bend left] & \textsc{Fib}_5 \arrow[r] \arrow[rr, bend left] & \textsc{Fib}_4 \arrow[r] \arrow[rr, bend left] & \textsc{Fib}_3 \arrow[r] \arrow[rr, bend left] & \textsc{Fib}_2 & \textsc{Fib}_1
\end{tikzcd}$$

Topological ordering of DAGs

An ordering $<$ of vertices is said to be a topological order if

$$(u, v) \in E \Rightarrow u < v$$

Topological ordering of DAGs

An ordering $<$ of vertices is said to be a topological order if

$$(u, v) \in E \Rightarrow u < v$$

Observation: If the digraph contains a cycle, then there is no topological ordering

Topological ordering of DAGs

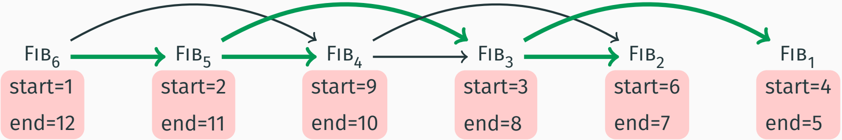
An ordering $<$ of vertices is said to be a topological order if

$$(u, v) \in E \Rightarrow u < v$$

Observation: If the digraph contains a cycle, then there is no topological ordering

Question: Do all DAGs have a topological ordering?

Topological ordering of DAGs



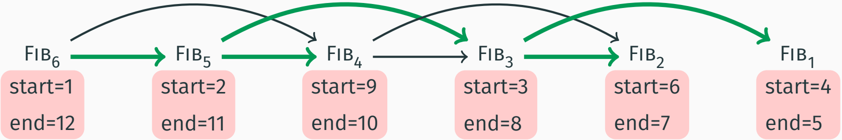
An ordering $<$ of vertices is said to be a topological order if

$$(u,v) \in E \Rightarrow u<v$$

Observation: If the digraph contains a cycle, then there is no topological ordering

Question: Do all DAGs have a topological ordering?

Topological ordering of DAGs



An ordering $<$ of vertices is said to be a topological order if

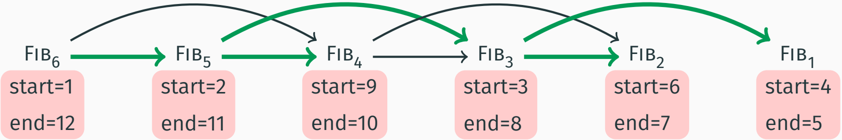
$$(u, v) \in E \Rightarrow u < v$$

end(u) > end(v)

Observation: If the digraph contains a cycle, then there is no topological ordering

Question: Do all DAGs have a topological ordering?

Topological ordering of DAGs



An ordering $<$ of vertices is said to be a topological order if

$$(u, v) \in E \Rightarrow u < v$$

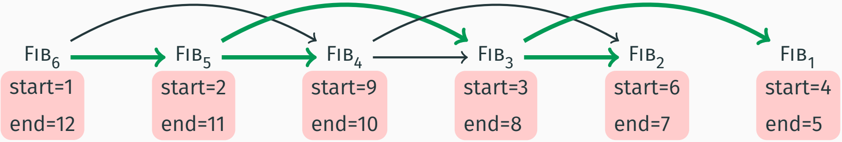
end(u) > end(v)

Observation: If the digraph contains a cycle, then there is no topological ordering

Question: Do all DAGs have a topological ordering?

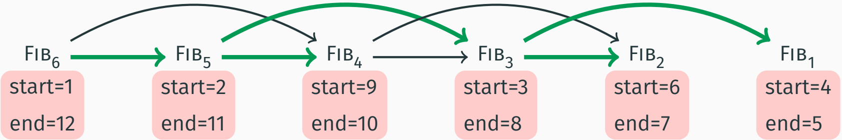
Theorem: The ordering of vertices in the decreasing order of their DFS end-times is a topological order

Topological ordering of DAGs



Theorem: The ordering of vertices in the decreasing order of their DFS end-times is a topological order

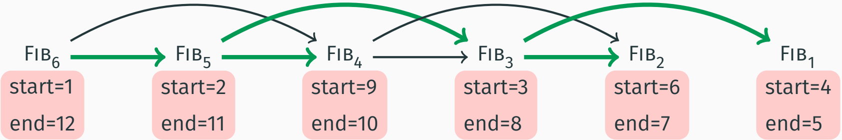
Topological ordering of DAGs



Theorem: The ordering of vertices in the decreasing order of their DFS end-times is a topological order

Proof: Suppose not. Let $(u, v) \in E$ and $\text{end}[u] < \text{end}[v]$.

Topological ordering of DAGs

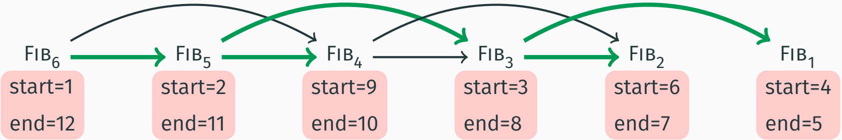


Theorem: The ordering of vertices in the decreasing order of their DFS end-times is a topological order

Proof: Suppose not. Let $(u, v) \in E$ and $\text{end}[u] < \text{end}[v]$.

Three cases possible about their start times

Topological ordering of DAGs



Theorem: The ordering of vertices in the decreasing order of their DFS end-times is a topological order

Proof: Suppose not. Let $(u, v) \in E$ and end$[u]$ < end$[v]$.

Three cases possible about their start times

- start$[u]$ < end$[u]$ < start$[v]$ < end$[v]$: not possible because if DFS from u starts before v, then it cannot end before the DFS of v since $(u, v) \in E$
- start$[u]$ < start$[v]$ < end$[u]$ < end$[v]$: not possible for any DFS
- start$[v]$ < start$[u]$ < end$[u]$ < end$[v]$: path from v to u $\Rightarrow$ there must be a cycle that includes the edge (u, v)

Topological ordering of DAGs

Observation: In a DAG G, if $(u, v) \in E$ then $\mathrm{end}[u] > \mathrm{end}[v]$

Topological ordering of DAGs

Observation: In a DAG G, if $(u, v) \in E$ then end$[u] >$ end$[v]$

Observation: Every DAG contains a source (vertex with in-degree=0) and a sink (vertex with out-degree=0)

Topological ordering of DAGs

Observation: In a DAG G, if $(u, v) \in E$ then end[u] > end[v]

Observation: Every DAG contains a source (vertex with in-degree=0) and a sink (vertex with out-degree=0)

> The vertex u with highest end[u] must be a source, and v with the lowest end[v] must be a sink.

Topological ordering of DAGs

Observation: In a DAG G, if $(u, v) \in E$ then end$[u]$ > end$[v]$

Observation: Every DAG contains a source (vertex with in-degree=0) and a sink (vertex with out-degree=0)

> The vertex u with highest end$[u]$ must be a source, and v with the lowest end$[v]$ must be a sink.

Algorithm: Find a source, append it to the topological ordering, delete the vertex and its edges, continue ...

Topological ordering of DAGs

Observation: In a DAG G, if $(u, v) \in E$ then end$[u]$ > end$[v]$

Observation: Every DAG contains a source (vertex with in-degree=0) and a sink (vertex with out-degree=0)

> The vertex u with highest end$[u]$ must be a source, and v with the lowest end$[v]$ must be a sink.

Algorithm: Find a source, append it to the topological ordering, delete the vertex and its edges, continue ...

Exercise: Implement this algorithm to run in time $O(|V| + |E|)$

DAGs and the structure of digraphs

DAGs and the structure of digraphs

DFS from v on a digraph covers all vertices u that are reachable from v - reach(v)

DAGs and the structure of digraphs

DFS from v on a digraph covers all vertices u that are reachable from v - reach(v)

- A strongly connected component is a maximal subset $V' \subseteq V$ such that for every $u, v \in V'$: $u \in \text{reach}(v)$ and $v \in \text{reach}(u)$
  - scc(u) - strongly connected component containing u

DAGs and the structure of digraphs

DFS from v on a digraph covers all vertices u that are reachable from v - reach(v)

- A strongly connected component is a maximal subset $V' \subseteq V$ such that for every $u, v \in V'$: $u \in \text{reach}(v)$ and $v \in \text{reach}(u)$

  scc(u) - strongly connected component containing u

Observation: For every u and v, either

$$\left.\begin{array}{l} \cdot \quad \text{scc}(u) = \text{scc}(v), \text{ or} \\ \cdot \quad \text{scc}(u) \cap \text{scc}(v) = \emptyset \end{array}\right\}$$

DAGs and the structure of digraphs

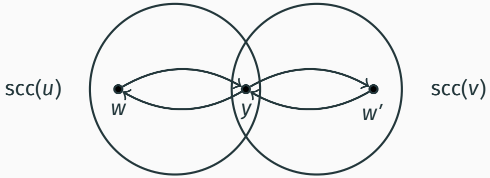
DFS from v on a digraph covers all vertices u that are reachable from v - reach(v)

- A strongly connected component is a maximal subset $V' \subseteq V$ such that for every $u, v \in V'$: $u \in \text{reach}(v)$ and $v \in \text{reach}(u)$

    scc(u) - strongly connected component containing u

Observation: For every u and v, either

$$\left.\begin{array}{l} \cdot \quad \text{scc}(u) = \text{scc}(v), \text{ or} \\ \cdot \quad \text{scc}(u) \cap \text{scc}(v) = \varnothing \end{array}\right\} \text{equivalence relation}$$



DAGs and the structure of digraphs

A strongly connected component is a maximal subset $V' \subseteq V$ such that for every $u, v \in V'$: $u \in \text{reach}(v)$ and $v \in \text{reach}(u)$

scc(u) - strongly connected component containing u

DAGs and the structure of digraphs

A **strongly connected component** is a maximal subset $V' \subseteq V$ such that for every $u, v \in V'$: $u \in \mathrm{reach}(v)$ and $v \in \mathrm{reach}(u)$

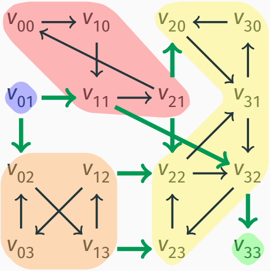
> $\mathrm{scc}(u)$ - strongly connected component containing u

$$
\begin{tikzcd}
v_{00} \arrow[r] & v_{10} \arrow[d] & v_{20} \arrow[dr] & v_{30} \arrow[l] \\
v_{01} \arrow[r] \arrow[d] & v_{11} \arrow[r] \arrow[drr] & v_{21} \arrow[u] \arrow[d] \arrow[ull] & v_{31} \arrow[u] \arrow[d] \\
v_{02} \arrow[dr] & v_{12} \arrow[r] \arrow[dl] & v_{22} \arrow[r] \arrow[ur] & v_{32} \arrow[d] \arrow[dl] \\
v_{03} \arrow[u] & v_{13} \arrow[u] \arrow[r] & v_{23} \arrow[u] & v_{33}
\end{tikzcd}
$$

DAGs and the structure of digraphs

A strongly connected component is a maximal subset $V' \subseteq V$ such that for every $u, v \in V'$: $u \in \text{reach}(v)$ and $v \in \text{reach}(u)$

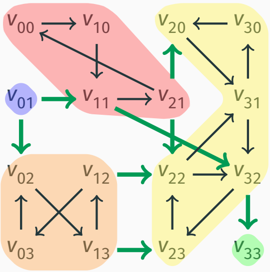
> $\text{scc}(u)$ - strongly connected component containing u



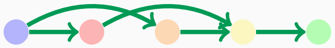
DAGs and the structure of digraphs

A strongly connected component is a maximal subset $V' \subseteq V$ such that for every $u, v \in V'$: $u \in \text{reach}(v)$ and $v \in \text{reach}(u)$

scc(u) - strongly connected component containing u



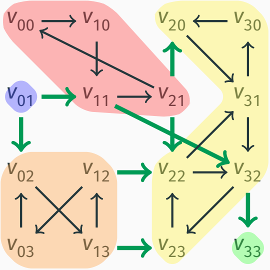
Digraph of strongly connected components: SCC(G)



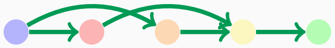
DAGs and the structure of digraphs

A strongly connected component is a maximal subset $V' \subseteq V$ such that for every $u, v \in V'$: $u \in \text{reach}(v)$ and $v \in \text{reach}(u)$

$\text{scc}(u)$ - strongly connected component containing u



Digraph of strongly connected components: $\text{SCC}(G)$



Question: Is $\text{SCC}(G)$ always a DAG?

Finding the strongly connected components

Question: Given a $v \in G$, find $\mathrm{scc}(v)$

Finding the strongly connected components

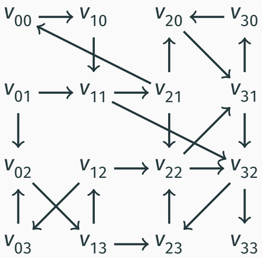
Question: Given a $v \in G$, find $\mathrm{scc}(v)$

- Find the set $\mathrm{reach}(v)$ using DFS

Finding the strongly connected components

Question: Given a $v \in G$, find $\mathrm{scc}(v)$

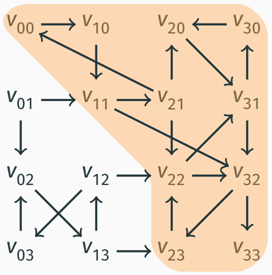
- Find the set $\mathrm{reach}(v)$ using DFS



Finding the strongly connected components

Question: Given a $v \in G$, find $\text{scc}(v)$

- Find the set $\text{reach}(v)$ using DFS

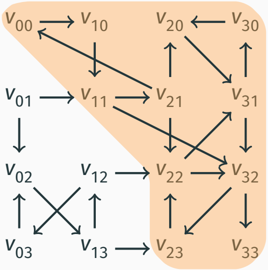


Finding the strongly connected components

Question: Given a $v \in G$, find $\mathrm{scc}(v)$

- Find the set $\mathrm{reach}(v)$ using DFS

  Which of these vertices are in $\mathrm{scc}(v)$?



Finding the strongly connected components

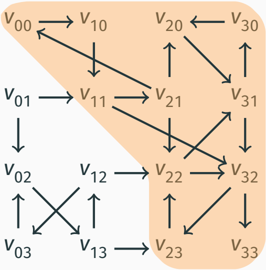
Question: Given a $v \in G$, find $\mathrm{scc}(v)$

- Find the set $\mathrm{reach}(v)$ using DFS

  Which of these vertices are in $\mathrm{scc}(v)$?

- All the vertices in the set

$$\mathrm{reach}^{-1}(v) := \{u \mid v \in \mathrm{reach}(u)\}$$



Finding the strongly connected components

Question: Given a $v \in G$, find $\mathrm{scc}(v)$

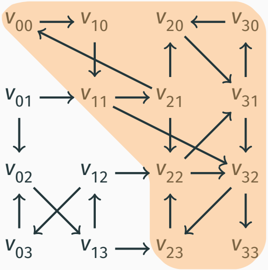
- Find the set $\mathrm{reach}(v)$ using DFS

  Which of these vertices are in $\mathrm{scc}(v)$?

- All the vertices in the set

$$\mathrm{reach}^{-1}(v) := \{u \mid v \in \mathrm{reach}(u)\}$$

How do you find this set?



Finding the strongly connected components

Question: Given a $v \in G$, find $\mathrm{scc}(v)$

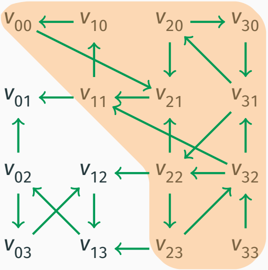
- Find the set $\mathrm{reach}(v)$ using DFS

  Which of these vertices are in $\mathrm{scc}(v)$?

- All the vertices in the set

$$\mathrm{reach}^{-1}(v) := \{u \mid v \in \mathrm{reach}(u)\}$$

How do you find this set?



$\mathrm{rev}(G)$

Finding the strongly connected components

Question: Given a $v \in G$, find $\text{scc}(v)$

- Find the set $\text{reach}(v)$ using DFS

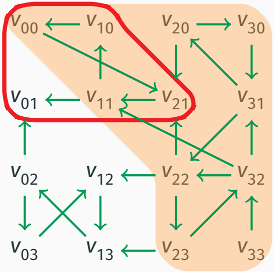
  Which of these vertices are in $\text{scc}(v)$?

- All the vertices in the set

$$\text{reach}^{-1}(v) := \{u \mid v \in \text{reach}(u)\}$$

How do you find this set?

$\text{reach}(v)$ in $\text{rev}(G)$



$\text{rev}(G)$

Finding the strongly connected components

Question: Given a $v \in G$, find $\mathrm{scc}(v)$

- Find the set $\mathrm{reach}(v)$ using DFS

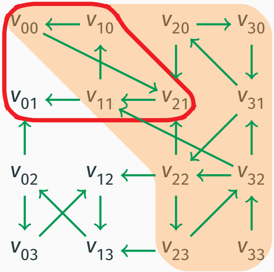
  Which of these vertices are in $\mathrm{scc}(v)$?

- All the vertices in the set

$$\mathrm{reach}^{-1}(v) := \{u \mid v \in \mathrm{reach}(u)\}$$

How do you find this set?

$\mathrm{reach}(v)$ in $\mathrm{rev}(G)$



$\mathrm{rev}(G)$

$$\mathrm{scc}(v) = \mathrm{reach}(v) \cap \mathrm{reach}^{-1}(v)$$

Finding the strongly connected components

Question: Given a $v \in G$, find $\mathrm{scc}(v)$

- Find the set $\mathrm{reach}(v)$ using DFS

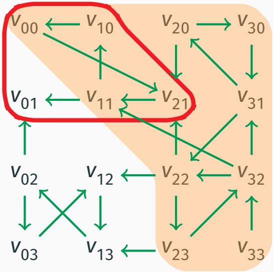
  Which of these vertices are in $\mathrm{scc}(v)$?

- All the vertices in the set

$$\mathrm{reach}^{-1}(v) := \{u \mid v \in \mathrm{reach}(u)\}$$

How do you find this set?

$\mathrm{reach}(v)$ in $\mathrm{rev}(G)$

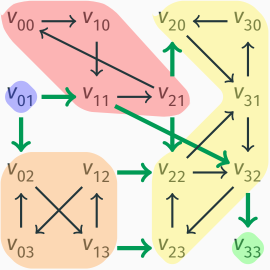


$\mathrm{rev}(G)$

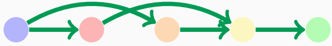
$$\mathrm{scc}(v) = \mathrm{reach}(v) \cap \mathrm{reach}^{-1}(v)$$

Exercise: How do we find $\mathrm{rev}(G)$ in $O(|V| + |E|)$ time?

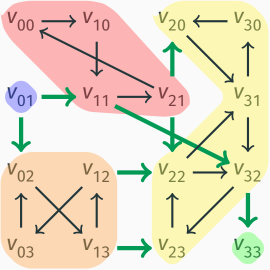
Finding the strongly connected components



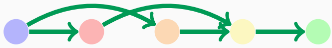
Digraph of strongly connected components: SCC(G)



Finding the strongly connected components

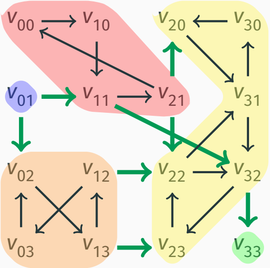


Digraph of strongly connected components: SCC(G)

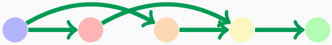


Question: For which vertices v, do we have scc(v) = reach(v)?

Finding the strongly connected components



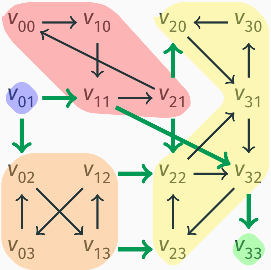
Digraph of strongly connected components: SCC(G)



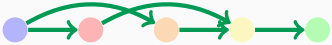
Question: For which vertices v, do we have scc(v) = reach(v)?

iff v is a vertex in a sink component of SCC(G)

Finding the strongly connected components



Digraph of strongly connected components: SCC(G)



Question: For which vertices v, do we have scc(v) = reach(v)?

iff v is a vertex in a sink component of SCC(G)

Algorithm idea:

- Find a vertex v in a sink component of SCC(G)
- Find reach(v)
- Remove it from the graph and continue...

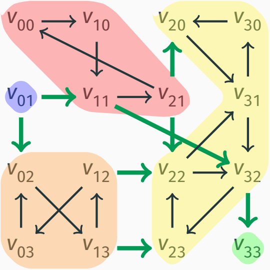
Finding the strongly connected components

Question: How do we find a vertex v in a sink component of $\mathrm{SCC}(G)$?

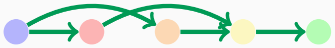
Finding the strongly connected components

Question: How do we find a vertex v in a sink component of $\mathrm{SCC}(G)$?

A different question: How do you find a vertex v in a source component of $\mathrm{SCC}(G)$?



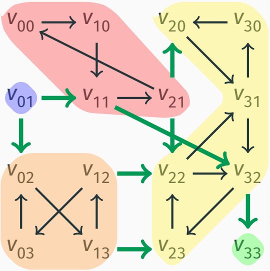
Digraph of strongly connected components: $\mathrm{SCC}(G)$



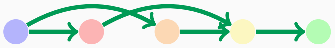
Finding the strongly connected components

Question: How do we find a vertex v in a sink component of SCC(G)?

A different question: How do you find a vertex v in a source component of SCC(G)?



Digraph of strongly connected components: SCC(G)



Which vertex v has the largest end[v] among all the vertices?

Finding the strongly connected components

Theorem: The vertex v with the largest value of end[v] must lie in a source component of SCC(G)

Finding the strongly connected components

Theorem: The vertex v with the largest value of end[v] must lie in a source component of SCC(G)

u - vertex in C_1 with largest end[] value
v - vertex in C_2 with largest end[] value, then

Finding the strongly connected components

Theorem: The vertex v with the largest value of end[v] must lie in a source component of SCC(G)

u - vertex in C_1 with largest end[] value
v - vertex in C_2 with largest end[] value,
then end[u] > end[v]

Finding the strongly connected components

Theorem: The vertex v with the largest value of end[v] must lie in a source component of SCC(G)

u - vertex in C_1 with largest end[] value
v - vertex in C_2 with largest end[] value,
then end[u] > end[v]

- If DFS visits vertex in C_1 before C_2, then the first vertex w visited in C_1 has end[v] greater than all the vertices in C_2

Finding the strongly connected components

Theorem: The vertex v with the largest value of end[v] must lie in a source component of SCC(G)

u - vertex in C_1 with largest end[] value
v - vertex in C_2 with largest end[] value,
then end[u] > end[v]

- If DFS visits vertex in C_1 before C_2, then the first vertex w visited in C_1 has end[v] greater than all the vertices in C_2
- If DFS visits a vertex in C_2 first, then first vertex w visited in C_1 has start[w] > end time of all vertices in C_2

Finding the strongly connected components

But, how do we get a vertex in the sink component?

Finding the strongly connected components

But, how do we get a vertex in the sink component?

- The sink components in SCC(G) are source components in rev(SCC(G))

Finding the strongly connected components

But, how do we get a vertex in the sink component?

- The sink components in SCC(G) are source components in rev(SCC(G))

Lemma: rev(SCC(G)) = SCC(rev(G))

Finding the strongly connected components

But, how do we get a vertex in the sink component?

- The sink components in SCC(G) are source components in rev(SCC(G))

Lemma: rev(SCC(G)) = SCC(rev(G))

Proof sketch:

- A subset $C \subseteq V$ is a strongly connected component of G iff it is a strongly connected component of rev(G)
- SCC(G) and SCC(rev(G)) have the same set of vertices
- An edge goes from component C_i to C_j in rev(SCC(G)) iff an edge goes from C_i to C_j in rev(G)

Finding the strongly connected components

But, how do we get a vertex in the sink component?

- The sink components in SCC(G) are source components in rev(SCC(G))

Lemma: rev(SCC(G)) = SCC(rev(G))

Proof sketch:

- A subset $C \subseteq V$ is a strongly connected component of G iff it is a strongly connected component of rev(G)
- SCC(G) and SCC(rev(G)) have the same set of vertices
- An edge goes from component C_i to C_j in rev(SCC(G)) iff an edge goes from C_i to C_j in rev(G)

To obtain the source components of rev(SCC(G)), it is sufficient to perform DFS on rev(G)

The Kosaraju-Sharir algorithm

The Kosaraju-Sharir algorithm

Phase 1: Perform DFS on rev(G)

DFS(v)
mark v
foreach edge $u \to w$ **do**
    if w is unmarked **then**
        DFS(w)
push(v)

The Kosaraju-Sharir algorithm

Phase 1: Perform DFS on rev(G)

DFS(v)
mark v
foreach edge $u \to w$ **do**
    if w is unmarked **then**
        DFS(w)
push(v)

vertices pushed in the order of ending times

The Kosaraju-Sharir algorithm

Phase 1: Perform DFS on rev(G)

DFS(v)
mark v
foreach edge $u \to w$ **do**
    if w is unmarked **then**
        DFS(w)
push(v)

vertices pushed in the order of ending times

Phase 2: Perform DFS in the order of the stack

DFS(G)
count $\leftarrow$ 1
foreach $v \in G$ **do**
    set cc[v] $\leftarrow$ 0
while stack is non-empty **do**
    $v \leftarrow$ pop()
    if cc[v] = 0 **then**
        LabelCC(v, count)
        count $\leftarrow$ count +1

The Kosaraju-Sharir algorithm

Phase 1: Perform DFS on rev(G)

DFS(v)
mark v
foreach edge $u \to w$ **do**
    if w is unmarked **then**
        DFS(w)
push(v)

vertices pushed in the order of ending times

Phase 2: Perform DFS in the order of the stack

DFS(G)
count $\leftarrow$ 1
foreach $v \in G$ **do**
    set cc[v] $\leftarrow$ 0
while stack is non-empty **do**
    $v \leftarrow$ pop()
    if cc[v] = 0 **then**
        LabelCC(v, count)
        count $\leftarrow$ count +1

LabelCC(v, count)
cc[v] = count
foreach edge $v \to w$ **do**
    if $cc[w] = 0$ **then**
        LabelCC(w, count)

The Kosaraju-Sharir algorithm

Phase 1: Perform DFS on rev(G)

```
DFS(v)
mark v
foreach edge u → w do
    if w is unmarked then
        DFS(w)
push(v)
```

vertices pushed in the order of ending times

- Perform DFS on rev(G), and order according to end times
- Perform DFS on G in this order and label components
- Total running time: $O(|V| + |E|)$

Phase 2: Perform DFS in the order of the stack

```
DFS(G)
count ← 1
foreach v ∈ G do
    set cc[v] ← 0
while stack is non-empty do
    v ← pop()
    if cc[v] = 0 then
        LabelCC(v, count)
        count ← count +1

LabelCC(v, count)
cc[v] = count
foreach edge v → w do
    if cc[w] = 0 then
        LabelCC(w, count)
```

Shortest paths in graphs

Finding shortest paths in graphs

Question: Given a (di)graph G, and two vertices s and t, find the shortest path from s to t in G

Finding shortest paths in graphs

Question: Given a (di)graph G, and two vertices s and t, find the shortest path from s to t in G

- The graphs could be weighted or unweighted - shortest path in terms of edge-weights

Finding shortest paths in graphs

Question: Given a (di)graph G, and two vertices s and t, find the shortest path from s to t in G

- The graphs could be weighted or unweighted - shortest path in terms of edge-weights
- All the algorithms actually solve a more general problem - compute the shortest distances from s to all the other nodes (Single-Source Shortest Path)

Finding shortest paths in graphs

Question: Given a (di)graph G, and two vertices s and t, find the shortest path from s to t in G

- The graphs could be weighted or unweighted - shortest path in terms of edge-weights
- All the algorithms actually solve a more general problem - compute the shortest distances from s to all the other nodes (Single-Source Shortest Path)
- Unweighted case - already seen. But, let's recall...

Unweighted graphs: Breadth-First Search

$\textsc{ShortestPath}(G)$

foreach $v \in G$ **do**
    $\text{dist}(v) \leftarrow \infty$
    $p[v] \leftarrow \emptyset$
$\text{dist}(s) \leftarrow 0$
$\text{enqueue}(s)$

Unweighted graphs: Breadth-First Search

```
SHORTESTPATH(G)
foreach v ∈ G do
    dist(v) ← ∞
    p[v] ← ∅
dist(s) ← 0
enqueue(s)
while queue is not empty do
    v ← dequeue()
    foreach edge v → w do
        if dist(w) = ∞ then
            dist(w) ← dist(v) + 1
            p[w] ← v
            enqueue(w)
```

Unweighted graphs: Breadth-First Search

```
SHORTESTPATH(G)
foreach v ∈ G do
    dist(v) ← ∞
    p[v] ← ∅
dist(s) ← 0
enqueue(s)
while queue is not empty do
    v ← dequeue()
    foreach edge v → w do
        if dist(w) = ∞ then
            dist(w) ← dist(v) + 1
            p[w] ← v
            enqueue(w)
```

- Every vertex is inserted in the queue only once

Unweighted graphs: Breadth-First Search

SHORTESTPATH(G)

```
foreach v ∈ G do
    dist(v) ← ∞
    p[v] ← ∅
dist(s) ← 0
enqueue(s)
while queue is not empty do
    v ← dequeue()
    foreach edge v → w do
        if dist(w) = ∞ then
            dist(w) ← dist(v) + 1
            p[w] ← v
            enqueue(w)
```

- Every vertex is inserted in the queue only once
- The value of dist(v) changes exactly once in the entire run of the algorithm

Unweighted graphs: Breadth-First Search

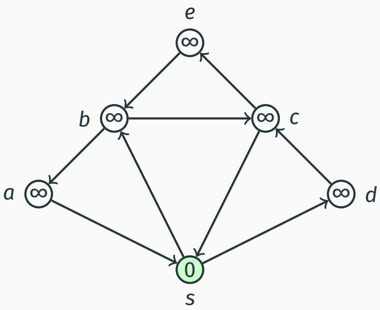
```
ShortestPath(G)
foreach v ∈ G do
    dist(v) ← ∞
    p[v] ← ∅
dist(s) ← 0
enqueue(s)
while queue is not empty do
    v ← dequeue()
    foreach edge v → w do
        if dist(w) = ∞ then
            dist(w) ← dist(v) + 1
            p[w] ← v
            enqueue(w)
```

- Every vertex is inserted in the queue only once
- The value of dist(v) changes exactly once in the entire run of the algorithm

Unweighted graphs: Breadth-First Search

SHORTESTPATH(G)

foreach $v \in G$ **do**
  dist(v) ← ∞
  $p[v]$ ← ∅
dist(s) ← 0
enqueue(s)
while queue is not empty **do**
  v ← dequeue()
  foreach edge $v \to w$ **do**
    if dist(w) = ∞ **then**
      dist(w) ← $dist(v) + 1$
      $p[w]$ ← v
      enqueue(w)



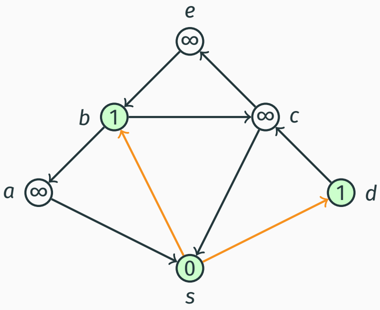
s

- Every vertex is inserted in the queue only once
- The value of dist(v) changes exactly once in the entire run of the algorithm

Unweighted graphs: Breadth-First Search

ShortestPath(G)

foreach $v \in G$ **do**
  dist(v) ← ∞
  $p[v]$ ← ∅
dist(s) ← 0
enqueue(s)
while queue is not empty **do**
  v ← dequeue()
  foreach edge $v \to w$ **do**
    if dist(w) = ∞ **then**
      dist(w) ← $dist(v) + 1$
      $p[w]$ ← v
      enqueue(w)



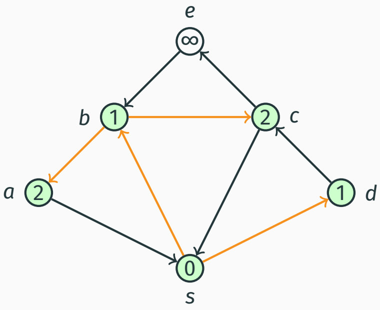
b d

- Every vertex is inserted in the queue only once
- The value of dist(v) changes exactly once in the entire run of the algorithm

Unweighted graphs: Breadth-First Search

SHORTESTPATH(G)

foreach $v \in G$ **do**
  dist(v) $\leftarrow \infty$
  $p[v] \leftarrow \emptyset$
dist(s) $\leftarrow 0$
enqueue(s)
while queue is not empty **do**
  $v \leftarrow$ dequeue()
  foreach edge $v \rightarrow w$ **do**
    if dist(w) = ∞ **then**
      dist(w) $\leftarrow dist(v) + 1$
      $p[w] \leftarrow v$
      enqueue(w)



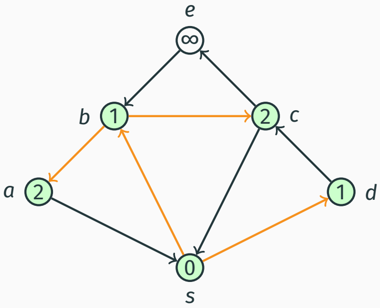
$d\ a\ c$

- Every vertex is inserted in the queue only once
- The value of dist(v) changes exactly once in the entire run of the algorithm

Unweighted graphs: Breadth-First Search

SHORTESTPATH(G)

foreach $v \in G$ **do**
    dist(v) ← ∞
    $p[v]$ ← ∅
dist(s) ← 0
enqueue(s)
while queue is not empty **do**
    v ← dequeue()
    foreach edge $v \to w$ **do**
        if dist(w) = ∞ **then**
            dist(w) ← $dist(v) + 1$
            $p[w]$ ← v
            enqueue(w)



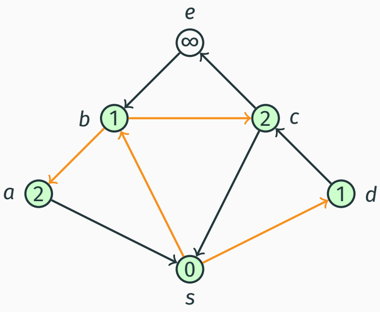
$a\ c$

- Every vertex is inserted in the queue only once
- The value of dist(v) changes exactly once in the entire run of the algorithm

Unweighted graphs: Breadth-First Search

SHORTESTPATH(G)

foreach $v \in G$ **do**
    dist(v) $\leftarrow \infty$
    $p[v] \leftarrow \emptyset$
dist(s) $\leftarrow 0$
enqueue(s)
while queue is not empty **do**
    $v \leftarrow$ dequeue()
    foreach edge $v \rightarrow w$ **do**
        if dist(w) $= \infty$ **then**
            dist(w) $\leftarrow dist(v) + 1$
            $p[w] \leftarrow v$
            enqueue(w)



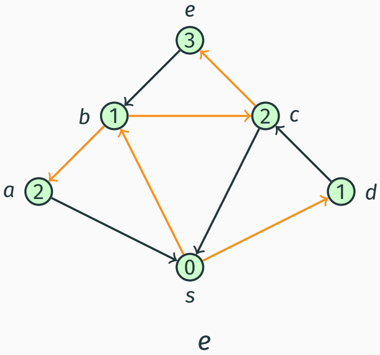
C

- Every vertex is inserted in the queue only once
- The value of dist(v) changes exactly once in the entire run of the algorithm

Unweighted graphs: Breadth-First Search

SHORTESTPATH(G)

```
foreach v ∈ G do
    dist(v) ← ∞
    p[v] ← ∅
dist(s) ← 0
enqueue(s)
while queue is not empty do
    v ← dequeue()
    foreach edge v → w do
        if dist(w) = ∞ then
            dist(w) ← dist(v) + 1
            p[w] ← v
            enqueue(w)
```

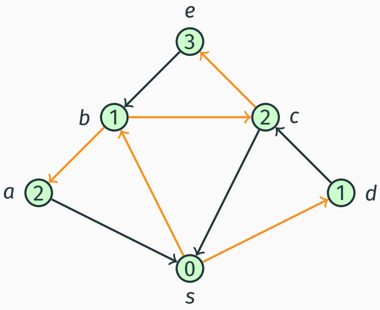


- Every vertex is inserted in the queue only once
- The value of dist(v) changes exactly once in the entire run of the algorithm

Unweighted graphs: Breadth-First Search

SHORTESTPATH(G)

foreach $v \in G$ **do**
    dist(v) $\leftarrow \infty$
    $p[v] \leftarrow \emptyset$
dist(s) $\leftarrow 0$
enqueue(s)
while queue is not empty **do**
    $v \leftarrow$ dequeue()
    foreach edge $v \to w$ **do**
        if dist(w) $= \infty$ **then**
            dist(w) $\leftarrow dist(v) + 1$
            $p[w] \leftarrow v$
            enqueue(w)



$\emptyset$

- Every vertex is inserted in the queue only once
- The value of dist(v) changes exactly once in the entire run of the algorithm

Unweighted graphs: Breadth-First Search

Theorem: When ShortestPath(G) ends, dist(v) is the length of the shortest path from s to v for every $v \in G$

Proof:

ShortestPath(G)

foreach $v \in G$ **do**
  dist(v) $\leftarrow \infty$
  $p[v] \leftarrow \emptyset$
dist(s) $\leftarrow 0$
enqueue(s)
while queue is not empty **do**
  $v \leftarrow$ dequeue()
  foreach edge $v \rightarrow w$ **do**
    if dist(w) $= \infty$ **then**
      dist(w) $\leftarrow dist(v) + 1$
      $p[w] \leftarrow v$
      enqueue(w)

Unweighted graphs: Breadth-First Search

Theorem: When SHORTESTPATH(G) ends, dist(v) is the length of the shortest path from s to v for every $v \in G$

Proof: For any path $P : s = v_0 \to v_1 \to v_2 \to \ldots \to v_t = v$, we will show that dist(v_j) $\leq j$

SHORTESTPATH(G)

```
foreach v ∈ G do
    dist(v) ← ∞
    p[v] ← ∅
dist(s) ← 0
enqueue(s)
while queue is not empty do
    v ← dequeue()
    foreach edge v → w do
        if dist(w) = ∞ then
            dist(w) ← dist(v) + 1
            p[w] ← v
            enqueue(w)
```

Unweighted graphs: Breadth-First Search

SHORTESTPATH(G)

foreach $v \in G$ **do**
  dist(v) ← ∞
  $p[v] \leftarrow \emptyset$
dist(s) ← 0
enqueue(s)
while queue is not empty **do**
  v ← dequeue()
  foreach edge $v \to w$ **do**
    if dist(w) = ∞ **then**
      dist(w) ← $dist(v) + 1$
      $p[w] \leftarrow v$
      enqueue(w)

Theorem: When SHORTESTPATH(G) ends, dist(v) is the length of the shortest path from s to v for every $v \in G$

Proof: For any path $P : s = v_0 \to v_1 \to v_2 \to \ldots \to v_t = v$, we will show that dist(v_j) $\leq j$

Proof by induction

- Base case: dist(v_0) = dist(s) = 0

Unweighted graphs: Breadth-First Search

Theorem: When ShortestPath(G) ends, dist(v) is the length of the shortest path from s to v for every $v \in G$

Proof: For any path $P : s = v_0 \to v_1 \to v_2 \to \ldots \to v_t = v$, we will show that dist(v_j) $\leq j$

Proof by induction

- Base case: dist(v_0) = dist(s) = 0
- Induction hypothesis: dist(v_i) $\leq i$ for every $i \leq j - 1$

```
ShortestPath(G)

foreach v ∈ G do
    dist(v) ← ∞
    p[v] ← ∅
dist(s) ← 0
enqueue(s)
while queue is not empty do
    v ← dequeue()
    foreach edge v → w do
        if dist(w) = ∞ then
            dist(w) ← dist(v) + 1
            p[w] ← v
            enqueue(w)
```

Unweighted graphs: Breadth-First Search

ShortestPath(G)

foreach $v \in G$ **do**
  dist(v) $\leftarrow \infty$
  $p[v] \leftarrow \emptyset$
dist(s) $\leftarrow 0$
enqueue(s)
while queue is not empty **do**
  $v \leftarrow$ dequeue()
  foreach edge $v \rightarrow w$ **do**
    if dist(w) $= \infty$ **then**
      dist(w) $\leftarrow dist(v) + 1$
      $p[w] \leftarrow v$
      enqueue(w)

Theorem: When ShortestPath(G) ends, dist(v) is the length of the shortest path from s to v for every $v \in G$

Proof: For any path $P : s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_t = v$, we will show that dist(v_j) $\leq j$

Proof by induction

- Base case: dist(v_0) = dist(s) = 0
- Induction hypothesis: dist(v_i) $\leq i$ for every $i \leq j - 1$
- Induction step: When v_{j-1} is dequeued, then two things can happen

Unweighted graphs: Breadth-First Search

ShortestPath(G)

foreach $v \in G$ **do**
    dist(v) $\leftarrow \infty$
    $p[v] \leftarrow \emptyset$
dist(s) $\leftarrow 0$
enqueue(s)
while queue is not empty **do**
    $v \leftarrow$ dequeue()
    foreach edge $v \rightarrow w$ **do**
        if dist(w) = ∞ **then**
            dist(w) $\leftarrow dist(v) + 1$
            $p[w] \leftarrow v$
            enqueue(w)

Theorem: When ShortestPath(G) ends, dist(v) is the length of the shortest path from s to v for every $v \in G$

Proof: For any path $P : s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_t = v$, we will show that dist(v_j) $\leq j$

Proof by induction

- Base case: dist(v_0) = dist(s) = 0
- Induction hypothesis: dist(v_i) $\leq i$ for every $i \leq j - 1$
- Induction step: When v_{j-1} is dequeued, then two things can happen
    - dist(v_j) $\leq$ dist(v_{j-1}) + 1 = j, or
    - v_j is enqueued, and dist(v_j) $\leftarrow$ dist(v_{j-1}) + 1 = j

Unweighted graphs: Breadth-First Search

ShortestPath(G)

foreach $v \in G$ **do**
  dist(v) $\leftarrow \infty$
  $p[v] \leftarrow \emptyset$
dist(s) $\leftarrow 0$
enqueue(s)
while queue is not empty **do**
  $v \leftarrow$ dequeue()
  foreach edge $v \to w$ **do**
    if dist(w) $= \infty$ **then**
      dist(w) $\leftarrow dist(v) + 1$
      $p[w] \leftarrow v$
      enqueue(w)

Theorem: When ShortestPath(G) ends, dist(v) is the length of the shortest path from s to v for every $v \in G$

Proof: For any path $P : s = v_0 \to v_1 \to v_2 \to \ldots \to v_t = v$, we will show that dist(v_j) $\leq j$

Proof by induction

- Base case: dist(v_0) = dist(s) = 0
- Induction hypothesis: dist(v_i) $\leq i$ for every $i \leq j - 1$
- Induction step: When v_{j-1} is dequeued, then two things can happen
  - dist(v_j) $\leq$ dist(v_{j-1}) + 1 = j, or
  - v_j is enqueued, and dist(v_j) $\leftarrow$ dist(v_{j-1}) + 1 = j

dist(v) must be at most the length of the shortest path from s to v, and
dist(v) is the length of an actual path from s to v

Weighted DAGs

Weighted DAGs

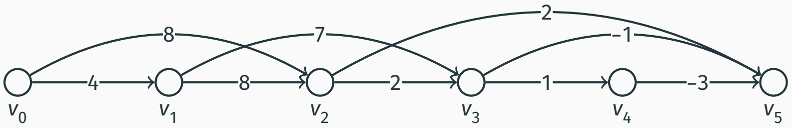
- Only need to compute for vertices v after s in the topological order

Weighted DAGs

- Only need to compute for vertices v after s in the topological order
- Multiple iterations before $\text{dist}(v)$ is correctly computed

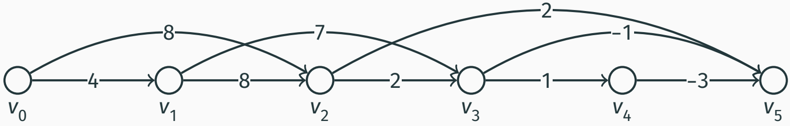
Weighted DAGs

- Only need to compute for vertices v after s in the topological order
- Multiple iterations before $\text{dist}(v)$ is correctly computed



Weighted DAGs

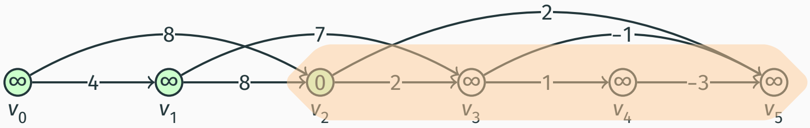
- Only need to compute for vertices v after s in the topological order
- Multiple iterations before $\text{dist}(v)$ is correctly computed



$s = v_2$

Weighted DAGs

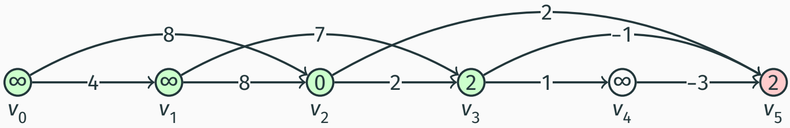
- Only need to compute for vertices v after s in the topological order
- Multiple iterations before $\text{dist}(v)$ is correctly computed



$s = v_2$

Weighted DAGs

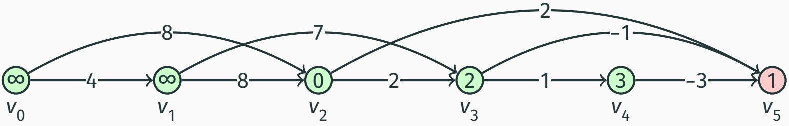
- Only need to compute for vertices v after s in the topological order
- Multiple iterations before $\mathrm{dist}(v)$ is correctly computed



$s = v_2$

Weighted DAGs

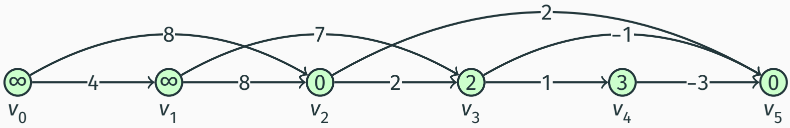
- Only need to compute for vertices v after s in the topological order
- Multiple iterations before $\text{dist}(v)$ is correctly computed



$s = v_2$

Weighted DAGs

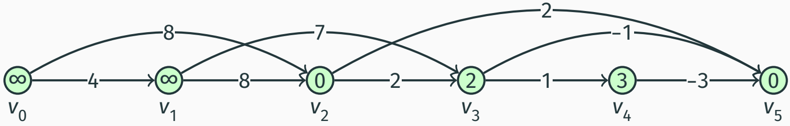
- Only need to compute for vertices v after s in the topological order
- Multiple iterations before $\mathrm{dist}(v)$ is correctly computed



$s = v_2$

Weighted DAGs

- Only need to compute for vertices v after s in the topological order
- Multiple iterations before $\text{dist}(v)$ is correctly computed

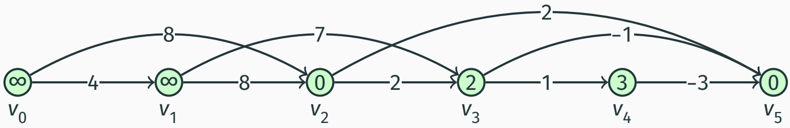


$s = v_2$

$$\text{dist}(v) = \min_{(u,v)\in E} \{\text{dist}(u) + w(u,v)\}$$

Weighted DAGs

- Only need to compute for vertices v after s in the topological order
- Multiple iterations before $\mathrm{dist}(v)$ is correctly computed



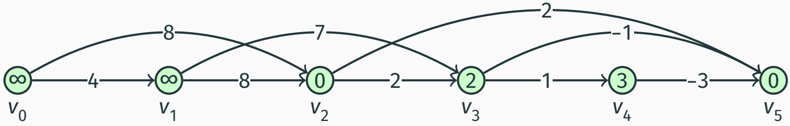
$s = v_2$

$$\mathrm{dist}(v) = \min_{(u,v)\in E} \{\mathrm{dist}(u) + w(u,v)\}$$

What is the dependency digraph?

Weighted DAGs

- Only need to compute for vertices v after s in the topological order
- Multiple iterations before $\text{dist}(v)$ is correctly computed



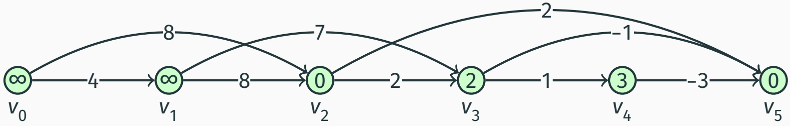
$s = v_2$

$$\text{dist}(v) = \min_{(u,v)\in E} \{\text{dist}(u) + w(u, v)\}$$

- The dependency digraph of the recurrence is $\text{rev}(G)$
- The order in which the recursive solutions are computed corresponds to the topological order of G

Weighted DAGs

- Only need to compute for vertices v after s in the topological order
- Multiple iterations before dist(v) is correctly computed



$s = v_2$

SSSP-DAG(G, s)

foreach $v \in G$ **do**
  dist(v) $\leftarrow \infty$
dist(s) $\leftarrow 0$
foreach $v \in G$ in topological order **do**
  foreach edge $v \to w$ **do**
    if dist(w) > dist(v) + wt(v, w) **then**
      dist(w) $\leftarrow$ dist(v) + wt(v, w)

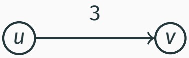
General weighted (di)graphs

Points to remember:

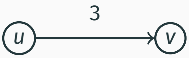
- Graphs cannot contain negative weight cycles
- For now, assume that there are no negative weight edges also

A BFS-based idea

A BFS-based idea



A BFS-based idea



A BFS-based idea

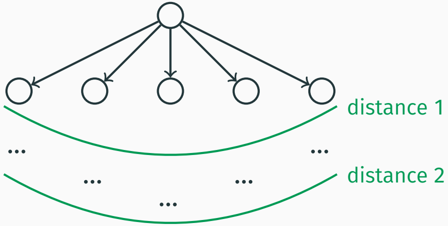
- Replace an edge of weight w with a path of length w with edge weight 1
- Perform BFS on the new graph

A BFS-based idea

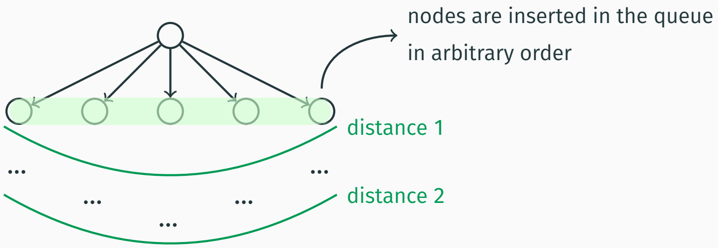
- Replace an edge of weight w with a path of length w with edge weight 1
- Perform BFS on the new graph

Question: Does this algorithm compute the shortest paths correctly?

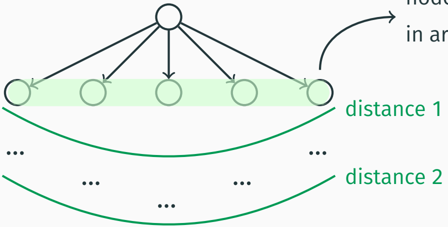
Generalizing the BFS idea



Generalizing the BFS idea



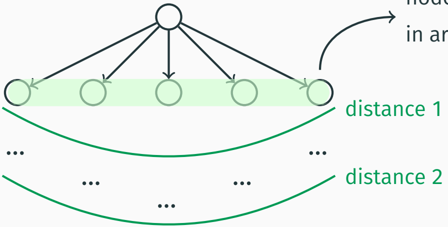
Generalizing the BFS idea



nodes are inserted in the queue
in arbitrary order

What happens when edges have weights?

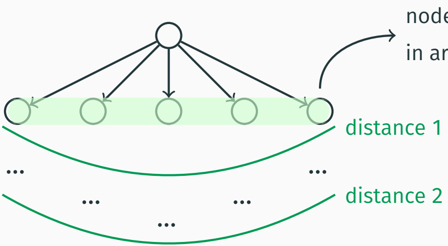
Generalizing the BFS idea



What happens when edges have weights?

- Maintain a set S of vertices whose shortest distances are computed, and expand this set one vertex at a time.

Generalizing the BFS idea



What happens when edges have weights?

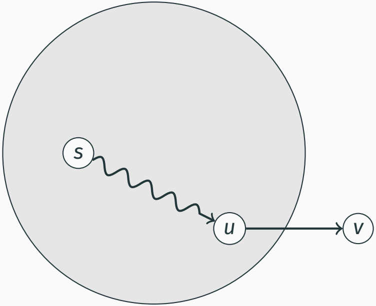
- Maintain a set S of vertices whose shortest distances are computed, and expand this set one vertex at a time.
- For BFS - add a neighbor of a vertex in S
- What should we do for weighted graphs?

Dijkstra's algorithm for SSSP

How do we choose the next vertex to add to the set S?

Dijkstra's algorithm for SSSP

How do we choose the next vertex to add to the set S?

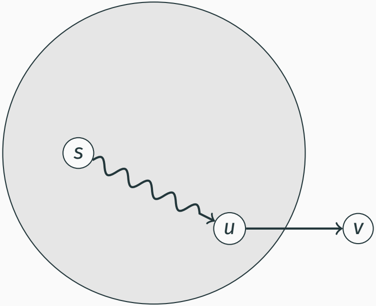


Choose v that minimizes

$\text{dist}(u) + \text{wt}(u, v)$ over all $u \in S$

Dijkstra's algorithm for SSSP

How do we choose the next vertex to add to the set S?

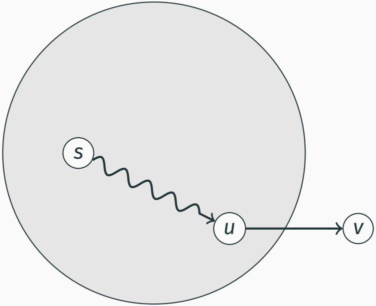


Choose v that minimizes
$\text{dist}(u) + \text{wt}(u, v)$ over all $u \in S$

Lemma: $\text{dist}(v)$ for the vertex obtained is the shortest distance from s to v.

Dijkstra's algorithm for SSSP

How do we choose the next vertex to add to the set S?



Choose v that minimizes

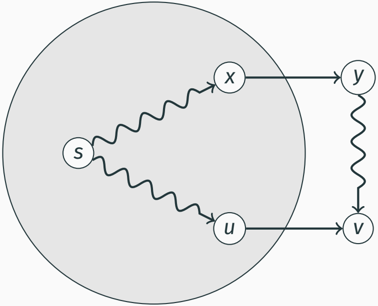
$\mathrm{dist}(u) + \mathrm{wt}(u, v)$ over all $u \in S$

Lemma: $\mathrm{dist}(v)$ for the vertex obtained is the shortest distance from s to v.

Proof: Suppose not!

Dijkstra's algorithm for SSSP

How do we choose the next vertex to add to the set S?



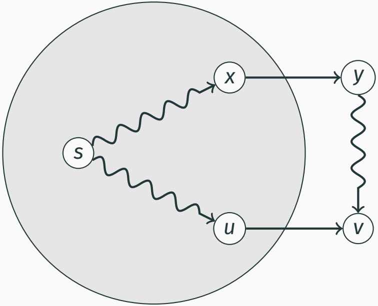
Choose v that minimizes
$\text{dist}(u) + \text{wt}(u, v)$ over all $u \in S$

Lemma: $\text{dist}(v)$ for the vertex obtained is the shortest distance from s to v.

Proof: Suppose not!

Dijkstra's algorithm for SSSP

How do we choose the next vertex to add to the set S?



Choose v that minimizes

$\mathrm{dist}(u) + \mathrm{wt}(u, v)$ over all $u \in S$

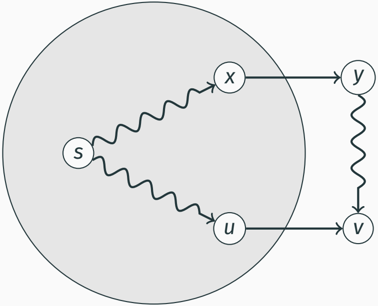
Lemma: $\mathrm{dist}(v)$ for the vertex obtained is the shortest distance from s to v.

Proof: Suppose not!

$\mathrm{dist}(x) + \mathrm{wt}(x, y) \geq dist(u) + \mathrm{wt}(u, v)$ and $\mathrm{dist}(y, v) \geq 0$

Dijkstra's algorithm for SSSP

How do we choose the next vertex to add to the set S?



Choose v that minimizes

$dist(u) + wt(u, v)$ over all $u \in S$

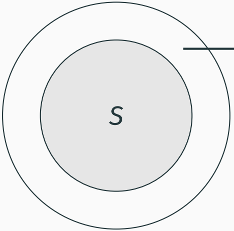
Lemma: $dist(v)$ for the vertex obtained is the shortest distance from s to v.

Proof: Suppose not!

$dist(x) + wt(x, y) \geq dist(u) + wt(u, v)$ and $dist(y, v) \geq 0$

Question: How do we find the vertex v efficiently?

Dijkstra's algorithm for SSSP



precomputed $\text{dist}(v) + w(u, v)$ values

- Choose the v with the smallest $\text{dist}(v) + w(u, v)$ where $u \in S$
- Once a v is chosen, recompute the $\text{dist}(w) + w(u, w)$ values for vertices lying outside $S \cup \{v\}$

need to recompute this
for only neighbors of v

Dijkstra's algorithm for SSSP

```
ShortestPath(G, s)
foreach v ∈ G do
    dist(v) ← ∞
dist(s) ← 0
enqueue(s)
while  queue is not empty do
    v ← dequeue()
    foreach edge v → w do
        if dist(w) = ∞  then
            dist(w) ← dist(v) + 1
            enqueue(w)
```

Dijkstra's algorithm for SSSP

```
DIJKSTRA(G, s)

foreach v ∈ G do
    dist(v) ← ∞
dist(s) ← 0
INSERT(s, 0)
while priority queue is not empty do
    v ← EXTRACTMIN()
    foreach edge v → w do
        if dist(w) > dist(v) + wt(v, w) then
            dist(w) ← dist(v) + wt(v, w)
            if w is in the priority queue then
                DECREASEKEY(w, dist(w))
            else
                INSERT(w, dist(w))
```

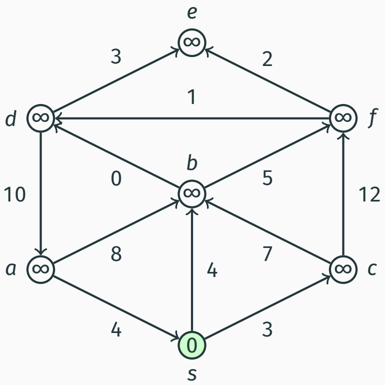
Dijkstra's algorithm for SSSP

```
Dijkstra(G, s)
foreach v ∈ G do
    dist(v) ← ∞
dist(s) ← 0
foreach v ∈ G do
    Insert(v, dist(v))
while priority queue is not empty do
    v ← ExtractMin()
    foreach edge v → w do
        if dist(w) > dist(v) + wt(v, w) then
            dist(w) ← dist(v) + wt(v, w)
            DecreaseKey(w, dist(w))
```

Dijkstra's algorithm for SSSP

DIJKSTRA(G, s)

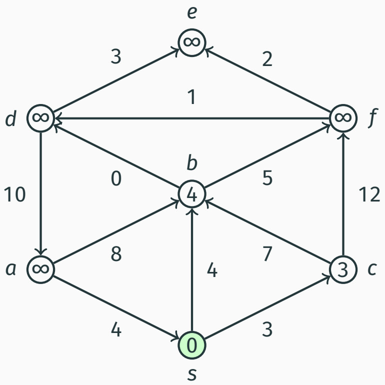
foreach $v \in G$ **do**
  dist(v) ← ∞
dist(s) ← 0
foreach $v \in G$ **do**
  INSERT(v, dist(v))
while priority queue is not empty **do**
  v ← EXTRACTMIN()
  foreach edge $v \to w$ **do**
    if dist(w) > dist(v) + wt(v, w) **then**
      dist(w) ← dist(v) + wt(v, w)
      DECREASEKEY(w, dist(w))



Dijkstra's algorithm for SSSP

DIJKSTRA(G, s)

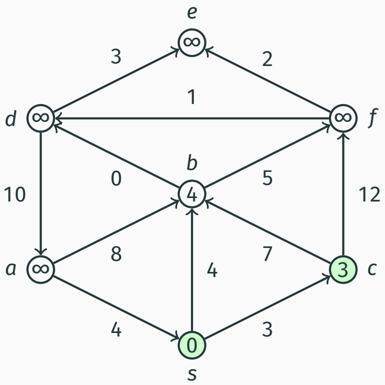
foreach $v \in G$ **do**
    dist(v) $\leftarrow \infty$
dist(s) $\leftarrow 0$
foreach $v \in G$ **do**
    INSERT(v, dist(v))
while priority queue is not empty **do**
    $v \leftarrow$ EXTRACTMIN()
    foreach edge $v \rightarrow w$ **do**
        if dist(w) > dist(v) + wt(v, w) **then**
            dist(w) $\leftarrow$ dist(v) + wt(v, w)
            DECREASEKEY(w, dist(w))



Dijkstra's algorithm for SSSP

DIJKSTRA(G, s)

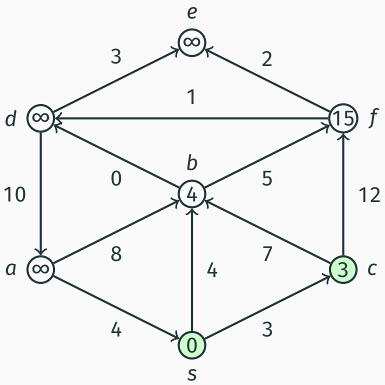
foreach $v \in G$ **do**
  dist(v) ← ∞
dist(s) ← 0
foreach $v \in G$ **do**
  INSERT(v, dist(v))
while priority queue is not empty **do**
  v ← EXTRACTMIN()
  foreach edge $v \to w$ **do**
    if dist(w) > dist(v) + wt(v, w) **then**
      dist(w) ← dist(v) + wt(v, w)
      DECREASEKEY(w, dist(w))



Dijkstra's algorithm for SSSP

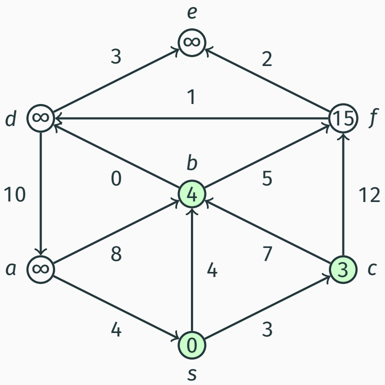
DIJKSTRA(G, s)

foreach $v \in G$ **do**
    dist(v) $\leftarrow \infty$
dist(s) $\leftarrow 0$
foreach $v \in G$ **do**
    INSERT(v, dist(v))
while priority queue is not empty **do**
    $v \leftarrow$ EXTRACTMIN()
    foreach edge $v \rightarrow w$ **do**
        if dist(w) > dist(v) + wt(v, w) **then**
            dist(w) $\leftarrow$ dist(v) + wt(v, w)
            DECREASEKEY(w, dist(w))



Dijkstra's algorithm for SSSP

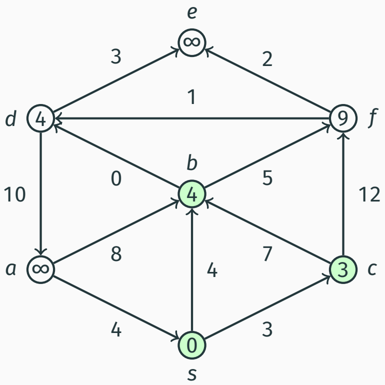
```
Dijkstra(G, s)
foreach v ∈ G do
    dist(v) ← ∞
dist(s) ← 0
foreach v ∈ G do
    Insert(v, dist(v))
while priority queue is not empty do
    v ← ExtractMin()
    foreach edge v → w do
        if dist(w) > dist(v) + wt(v, w) then
            dist(w) ← dist(v) + wt(v, w)
            DecreaseKey(w, dist(w))
```



Dijkstra's algorithm for SSSP

DIJKSTRA(G, s)

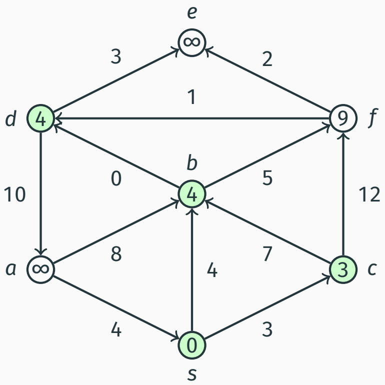
foreach $v \in G$ **do**
    dist(v) $\leftarrow \infty$
dist(s) $\leftarrow 0$
foreach $v \in G$ **do**
    INSERT(v, dist(v))
while priority queue is not empty **do**
    $v \leftarrow$ EXTRACTMIN()
    foreach edge $v \rightarrow w$ **do**
        if dist(w) > dist(v) + wt(v, w) **then**
            dist(w) $\leftarrow$ dist(v) + wt(v, w)
            DECREASEKEY(w, dist(w))



Dijkstra's algorithm for SSSP

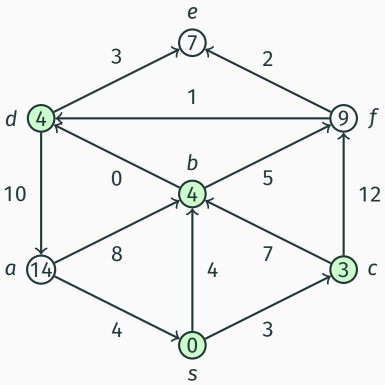
DIJKSTRA(G, s)

foreach $v \in G$ **do**
    dist(v) $\leftarrow \infty$
dist(s) $\leftarrow 0$
foreach $v \in G$ **do**
    INSERT(v, dist(v))
while priority queue is not empty **do**
    $v \leftarrow$ EXTRACTMIN()
    foreach edge $v \rightarrow w$ **do**
        if dist(w) > dist(v) + wt(v, w) **then**
            dist(w) $\leftarrow$ dist(v) + wt(v, w)
            DECREASEKEY(w, dist(w))



Dijkstra's algorithm for SSSP

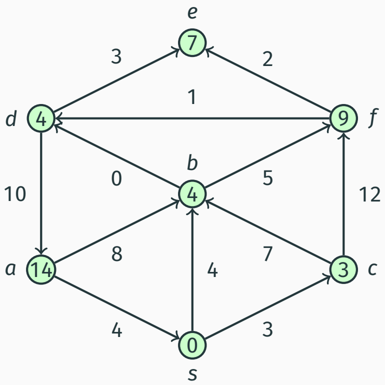
```
DIJKSTRA(G, s)
foreach v ∈ G do
    dist(v) ← ∞
dist(s) ← 0
foreach v ∈ G do
    INSERT(v, dist(v))
while priority queue is not empty do
    v ← EXTRACTMIN()
    foreach edge v → w do
        if dist(w) > dist(v) + wt(v, w) then
            dist(w) ← dist(v) + wt(v, w)
            DECREASEKEY(w, dist(w))
```



Dijkstra's algorithm for SSSP

DIJKSTRA(G, s)

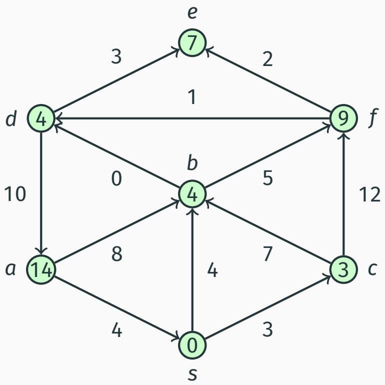
foreach $v \in G$ **do**
  dist(v) ← ∞
dist(s) ← 0
foreach $v \in G$ **do**
  INSERT(v, dist(v))
while priority queue is not empty **do**
  v ← EXTRACTMIN()
  foreach edge $v \rightarrow w$ **do**
    if dist(w) > dist(v) + wt(v, w) **then**
      dist(w) ← dist(v) + wt(v, w)
      DECREASEKEY(w, dist(w))



Dijkstra's algorithm for SSSP

DIJKSTRA(G, s)

foreach $v \in G$ **do**
    dist(v) $\leftarrow \infty$
dist(s) $\leftarrow 0$
foreach $v \in G$ **do**
    INSERT(v, dist(v))
while priority queue is not empty **do**
    $v \leftarrow$ EXTRACTMIN()
    foreach edge $v \rightarrow w$ **do**
        if dist(w) > dist(v) + wt(v, w) **then**
            dist(w) $\leftarrow$ dist(v) + wt(v, w)
            DECREASEKEY(w, dist(w))

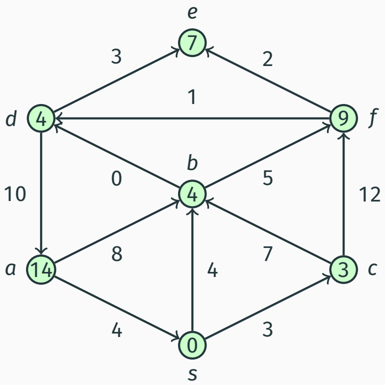


- Each vertex is inserted in the priority queue, and extracted exactly once

Dijkstra's algorithm for SSSP

$\textsc{Dijkstra}(G, s)$

foreach $v \in G$ **do**
    $\text{dist}(v) \leftarrow \infty$
$\text{dist}(s) \leftarrow 0$
foreach $v \in G$ **do**
    $\textsc{Insert}(v, \text{dist}(v))$
while priority queue is not empty **do**
    $v \leftarrow \textsc{ExtractMin}()$
    foreach edge $v \rightarrow w$ **do**
        if $\text{dist}(w) > \text{dist}(v) + \text{wt}(v, w)$ **then**
            $\text{dist}(w) \leftarrow \text{dist}(v) + \text{wt}(v, w)$
            $\textsc{DecreaseKey}(w, \text{dist}(w))$

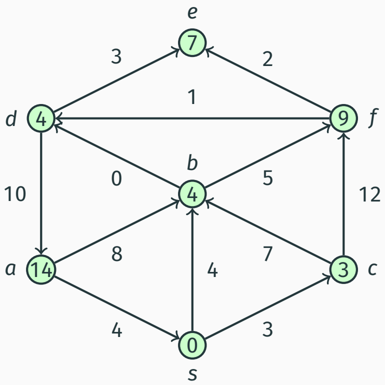


- Each vertex is inserted in the priority queue, and extracted exactly once
- Each edge is relaxed at most once

Dijkstra's algorithm for SSSP

DIJKSTRA(G, s)

foreach $v \in G$ **do**
  dist(v) $\leftarrow \infty$
dist(s) $\leftarrow 0$
foreach $v \in G$ **do**
  INSERT(v, dist(v))
while priority queue is not empty **do**
  $v \leftarrow$ EXTRACTMIN()
  foreach edge $v \rightarrow w$ **do**
    if dist(w) > dist(v) + wt(v, w) **then**
      dist(w) $\leftarrow$ dist(v) + wt(v, w)
      DECREASEKEY(w, dist(w))



- Each vertex is inserted in the priority queue, and extracted exactly once
- Each edge is relaxed at most once

Running time

- $O((|V| + |E|) \log |V|)$ (using binary heaps)
- $O(|V| \log |V| + |E|)$ (using Fibonacci heaps)

SSSP with negative-weight edges

SSSP with negative-weight edges

- Already seen in the case of DAGs

SSSP with negative-weight edges

- Already seen in the case of DAGs
- Dijkstra's algorithm fails when there are negative-weight edges - can you explain why?

SSSP with negative-weight edges

- Already seen in the case of DAGs
- Dijkstra's algorithm fails when there are negative-weight edges - can you explain why?
- Problem is ill-posed when the graph contains negative-weight cycles
- But, can you identify this situation and fail gracefully?

SSSP with negative-weight edges

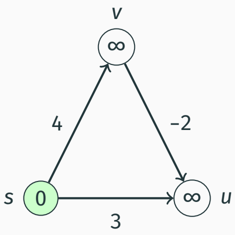
Observations

- dist(v) is always at least as large as the shortest distance to v
- Every edge is relaxed at most once in a run of Dijkstra's algorithm

SSSP with negative-weight edges

Observations

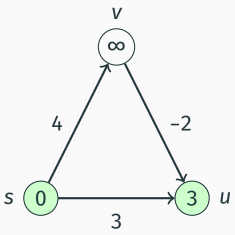
- dist(v) is always at least as large as the shortest distance to v
- Every edge is relaxed at most once in a run of Dijkstra's algorithm



SSSP with negative-weight edges

Observations

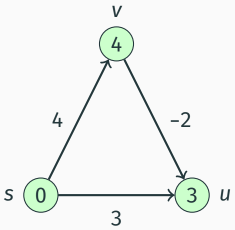
- dist(v) is always at least as large as the shortest distance to v
- Every edge is relaxed at most once in a run of Dijkstra's algorithm



SSSP with negative-weight edges

Observations

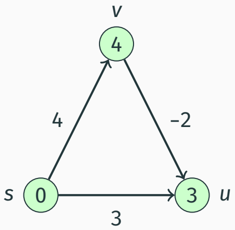
- dist(v) is always at least as large as the shortest distance to v
- Every edge is relaxed at most once in a run of Dijkstra's algorithm



SSSP with negative-weight edges

Observations

- dist(v) is always at least as large as the shortest distance to v
- Every edge is relaxed at most once in a run of Dijkstra's algorithm



one more update will set the distances correctly

The Bellman-Ford algorithm

Recall the recurrence

$$\text{dist}(v) = \min_{(u,v) \in E} \{\text{dist}(u) + w(u, v)\}$$

The Bellman-Ford algorithm

Modified recurrence

$$\mathrm{dist}_{\leq i}(v) = \begin{cases} 0 & \text{if } i = 0 \text{ and } v = s \\ \infty & \text{if } i = 0 \text{ and } v \neq s \\ & \\ & \end{cases}$$

shortest distance from s to v
by paths with $\leq i$ edges

The Bellman-Ford algorithm

Modified recurrence

$$\mathrm{dist}_{\le i}(v) = \begin{cases} 0 & \text{if } i = 0 \text{ and } v = s \\ \infty & \text{if } i = 0 \text{ and } v \neq s \\ \min \left\{ \begin{array}{l} \mathrm{dist}_{\le i-1}(v) \\ \min_{(u,v)\in E} \mathrm{dist}_{\le i-1}(u) + w(u,v) \end{array} \right\} & \text{otherwise} \end{cases}$$

shortest distance from s to v

by paths with $\le i$ edges

The Bellman-Ford algorithm

Modified recurrence

$$\mathrm{dist}_{\leq i}(v) = \begin{cases} 0 & \text{if } i = 0 \text{ and } v = s \\ \infty & \text{if } i = 0 \text{ and } v \neq s \\ \min \begin{Bmatrix} \mathrm{dist}_{\leq i-1}(v) \\ \min_{(u,v)\in E} \mathrm{dist}_{\leq i-1}(u) + w(u,v) \end{Bmatrix} & \text{otherwise} \end{cases}$$

shortest distance from s to v

by paths with $\leq i$ edges

- Have to compute $\mathrm{dist}_{\leq i}(v)$ for $i \leq n - 1$
- In every iteration, relax all the edges

if $\mathrm{dist}(v) > \mathrm{dist}(u) + w(u,v)$ **then**
$\mathrm{dist}(v) = \mathrm{dist}(u) + w(u,v)$

The Bellman-Ford algorithm

```
BellmanFord(G, s)
foreach v ∈ G do
    dist(v) ← ∞
dist(s) ← 0
repeat |V| − 1 times
    foreach edge u → v do
        if dist(v) > dist(u) + wt(u, v) then
            dist(v) = dist(u) + wt(u, v)
```

The Bellman-Ford algorithm

BellmanFord(G, s)

foreach $v \in G$ **do**
    $\text{dist}(v) \leftarrow \infty$
$\text{dist}(s) \leftarrow 0$
repeat $|V| - 1$ **times**
    foreach edge $u \to v$ **do**
        if $\text{dist}(v) > \text{dist}(u) + \text{wt}(u, v)$ **then**
            $\text{dist}(v) = \text{dist}(u) + \text{wt}(u, v)$

Lemma: Let $\text{dist}_{\leq i}(v)$ be the distance of the shortest path from s to v using at most i edges. Then, after the i^{th} iteration of the **repeat** loop for every vertex v,

$$\text{dist}(v) \leq \text{dist}_{\leq i}(v)$$

The Bellman-Ford algorithm

```
BELLMANFORD(G, s)
foreach v ∈ G do
    dist(v) ← ∞
dist(s) ← 0
repeat |V| − 1 times
    foreach edge u → v do
        if dist(v) > dist(u) + wt(u, v) then
            dist(v) = dist(u) + wt(u, v)
```

Lemma: Let $\mathrm{dist}_{\leq i}(v)$ be the distance of the shortest path from s to v using at most i edges. Then, after the i^{th} iteration of the **repeat** loop for every vertex v,

$$\mathrm{dist}(v) \leq \mathrm{dist}_{\leq i}(v)$$

Proof: Induction on i

The Bellman-Ford algorithm

$\textsc{BellmanFord}(G, s)$

foreach $v \in G$ **do**
  $\text{dist}(v) \leftarrow \infty$
$\text{dist}(s) \leftarrow 0$
repeat $|V| - 1$ **times**
  foreach edge $u \rightarrow v$ **do**
    if $\text{dist}(v) > \text{dist}(u) + \text{wt}(u, v)$ **then**
      $\text{dist}(v) = \text{dist}(u) + \text{wt}(u, v)$

Lemma: Let $\text{dist}_{\leq i}(v)$ be the distance of the shortest path from s to v using at most i edges. Then, after the i^{th} iteration of the **repeat** loop for every vertex v,

$$\text{dist}(v) \leq \text{dist}_{\leq i}(v)$$

Proof: Induction on i
Base case $i = 0$: $\text{dist}(s)$ is set to 0, and the rest to ∞

The Bellman-Ford algorithm

$\textsc{BellmanFord}(G, s)$

foreach $v \in G$ **do**
    $\text{dist}(v) \leftarrow \infty$
$\text{dist}(s) \leftarrow 0$
repeat $|V| - 1$ **times**
    foreach edge $u \to v$ **do**
        if $\text{dist}(v) > \text{dist}(u) + \text{wt}(u, v)$ **then**
            $\text{dist}(v) = \text{dist}(u) + \text{wt}(u, v)$

Lemma: Let $\text{dist}_{\le i}(v)$ be the distance of the shortest path from s to v using at most i edges. Then, after the i^{th} iteration of the **repeat** loop for every vertex v,

$$\text{dist}(v) \le \text{dist}_{\le i}(v)$$

must be a simple path!

Proof: Induction on i
Induction step $s \to u_1 \to \dots u_k \to v$: shortest path to v of length $\le i$

The Bellman-Ford algorithm

BELLMANFORD(G, s)

foreach $v \in G$ **do**
  dist(v) $\leftarrow \infty$
dist(s) $\leftarrow 0$
repeat $|V| - 1$ **times**
  foreach edge $u \rightarrow v$ **do**
    if dist(v) > dist(u) + wt(u, v) **then**
      dist(v) = dist(u) + wt(u, v)

Lemma: Let $\text{dist}_{\leq i}(v)$ be the distance of the shortest path from s to v using at most i edges. Then, after the i^{th} iteration of the **repeat** loop for every vertex v,

$$\text{dist}(v) \leq \text{dist}_{\leq i}(v)$$

must be a simple path!

Proof: Induction on i

Induction step $s \rightarrow u_1 \rightarrow \ldots u_k \rightarrow v$: shortest path to v of length $\leq i$

After the $(i-1)^{st}$ iteration, $\text{dist}(u_k) \leq \text{dist}_{i-1}(u_k)$

The Bellman-Ford algorithm

BellmanFord(G, s)

foreach $v \in G$ **do**
    dist(v) $\leftarrow \infty$
dist(s) $\leftarrow 0$
repeat $|V| - 1$ **times**
    foreach edge $u \to v$ **do**
        if dist(v) > dist(u) + wt(u, v) **then**
            dist(v) = dist(u) + wt(u, v)

Lemma: Let $\text{dist}_{\le i}(v)$ be the distance of the shortest path from s to v using at most i edges. Then, after the i^{th} iteration of the **repeat** loop for every vertex v,

$$\text{dist}(v) \le \text{dist}_{\le i}(v)$$

must be a simple path!

Proof: Induction on i
Induction step $s \to u_1 \to \dots u_k \to v$: shortest path to v of length $\le i$
After the $(i-1)^{st}$ iteration, $\text{dist}(u_k) \le \text{dist}_{i-1}(u_k)$
After the i^{th} iteration, $\text{dist}(v) \le \text{dist}(u_k) + \text{wt}(u, v) \le \text{dist}_{\le i-1}(u_k) + \text{wt}(u, v) \le \text{dist}_{\le i}(v)$

The Bellman-Ford algorithm

BELLMANFORD(G, s)

foreach $v \in G$ **do**
    dist(v) $\leftarrow \infty$
dist(s) $\leftarrow 0$
repeat $|V| - 1$ **times**
    foreach edge $u \rightarrow v$ **do**
        if dist(v) > dist(u) + wt(u, v) **then**
            dist(v) = dist(u) + wt(u, v)

Lemma: Let $\text{dist}_{\leq i}(v)$ be the distance of the shortest path from s to v using at most i edges. Then, after the i^{th} iteration of the **repeat** loop for every vertex v,

$$\text{dist}(v) \leq \text{dist}_{\leq i}(v)$$

Running time: $O(|V||E|)$

The Bellman-Ford algorithm

$\textsc{BellmanFord}(G, s)$

foreach $v \in G$ **do**
    $\text{dist}(v) \leftarrow \infty$
$\text{dist}(s) \leftarrow 0$
repeat $|V| - 1$ **times**
    foreach edge $u \to v$ **do**
        if $\text{dist}(v) > \text{dist}(u) + \text{wt}(u, v)$ **then**
            $\text{dist}(v) = \text{dist}(u) + \text{wt}(u, v)$

Lemma: Let $\text{dist}_{\leq i}(v)$ be the distance of the shortest path from s to v using at most i edges. Then, after the i^{th} iteration of the **repeat** loop for every vertex v,

$$\text{dist}(v) \leq \text{dist}_{\leq i}(v)$$

Running time: $O(|V||E|)$

Question: How do we detect if G has a negative weight cycle?