Data Structures

Graphs

DataLab

November 19, 2016

Outline

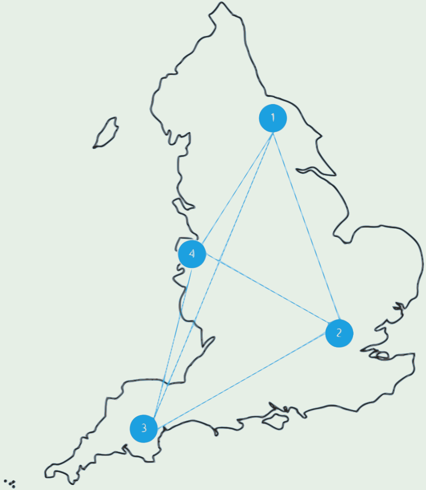
DataLab
Data Science Community

Outline

DataLab
Data Science Community

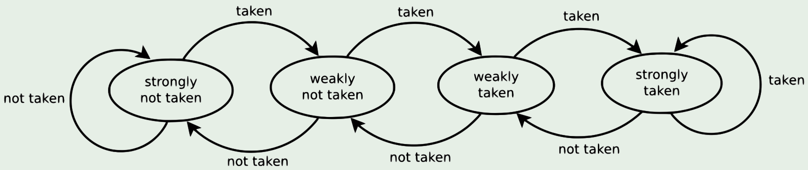
We are full of Graphs

Maps



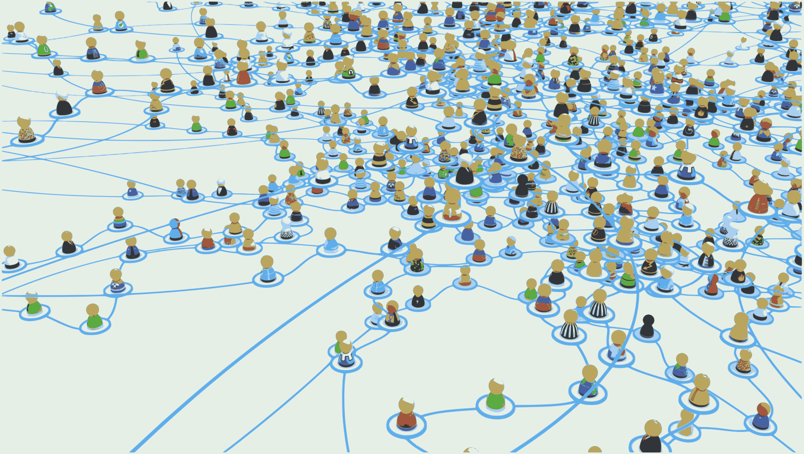
We are full of Graphs

State Machines for Branch Prediction in CPU's



We are full of Graphs

Social Networks



Outline

We need NICE representations

First One

Matrix Representation

Second One

Adjacency Representation

We need NICE representations

First One

Matrix Representation

Second One

Adjacency Representation

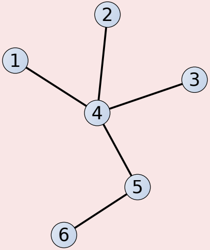
Outline

DataLab
Data Science Community

Adjacency Matrix Representation

This is the simplest one

If we number the nodes of an undirected graph:



	1	2	3	4	5	6
1	0	0	0	1	0	0
2	0	0	0	1	0	0
3	0	0	0	1	0	0
4	1	1	1	0	1	0
5	0	0	0	1	0	1
6	0	0	0	0	1	0

Adjacency Matrix Representation

In a natural way the edges can be identified by the nodes

For example, the edge between 1 and 4 nodes gets named as (1,4)

Then

How, we use this to represent the graph through a Matrix or and Array of Arrays??!!!

Adjacency Matrix Representation

In a natural way the edges can be identified by the nodes

For example, the edge between 1 and 4 nodes gets named as (1,4)

Then

How, we use this to represent the graph through a Matrix or and Array of Arrays??!!!

What about the following?

How do we indicate that an edge exist given the following matrix

	1	2	3	4	5	6
1	—	—	—	—	—	—
2	—	—	—	—	—	—
3	—	—	—	—	—	—
4	—	—	—	—	—	—
5	—	—	—	—	—	—
6	—	—	—	—	—	—

You say it!!

- Use a 0 for no-edge
- Use a 1 for edge

What about the following?

How do we indicate that an edge exist given the following matrix

$$
\begin{array}{c|cccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & - & - & - & - & - & - \\
2 & - & - & - & - & - & - \\
3 & - & - & - & - & - & - \\
4 & - & - & - & - & - & - \\
5 & - & - & - & - & - & - \\
6 & - & - & - & - & - & - \\
\end{array}
$$

You say it!!

- Use a 0 for no-edge
- Use a 1 for edge

We have then...

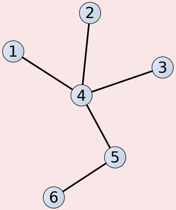
Definition

- $0/1$ $N \times N$ matrix with N =Number of nodes or vertices
- $A(i,j) = 1$ iff (i,j) is an edge

DataLab
Data Science Community

We have then...

For the previous example



	1	2	3	4	5	6
1	0	0	0	1	0	0
2	0	0	0	1	0	0
3	0	0	0	1	0	0
4	1	1	1	0	1	0
5	0	0	0	1	0	1
6	0	0	0	0	1	0

	1	2	3	4	5	6
1	0	0	0	1	0	0
2	0	0	0	1	0	0
3	0	0	0	1	0	0
4	1	1	1	0	1	0
5	0	0	0	1	0	1
6	0	0	0	0	1	0

Properties of the Matrix for Undirected Graphs

Property One

Diagonal entries are zero.

Property Two

Adjacency matrix of an undirected graph is symmetric:

$$A(i,j) = A(j,i) \text{ for all } i \text{ and } j$$

Properties of the Matrix for Undirected Graphs

Property One

Diagonal entries are zero.

Property Two

Adjacency matrix of an undirected graph is symmetric:

$$A(i,j) = A(j,i) \text{ for all } i \text{ and } j$$

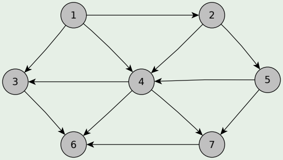
What about direct Graphs!!!

Similar idea

- Use a 0 for no-edge
- Use a 1 for directed edge

Example

We have that



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

Outline

What about the code?

Partial Code

We can try to write it!!!

What about Time Complexity?

Operations on a Graph

Most of the basic operations in a graph are:

- Adding an edge – $O(1)$
- Deleting an edge – $O(1)$
- Answering the question "is there an edge between i and j" – $O(1)$
- Finding the successors of a given vertex – $O(N)$
- Finding (if exists) a path between two vertices – $O(N^2)$

What about Time Complexity?

Operations on a Graph

Most of the basic operations in a graph are:

- Adding an edge – $O(1)$
- Deleting an edge – $O(1)$
- Answering the question "is there an edge between i and j" – $O(1)$
- Finding the successors of a given vertex – $O(N)$
- Finding (if exists) a path between two vertices – $O(N^2)$

What about Time Complexity?

Operations on a Graph

Most of the basic operations in a graph are:

- Adding an edge – $O(1)$
- Deleting an edge – $O(1)$
- Answering the question "is there an edge between i and j" – $O(1)$
- Finding the successors of a given vertex – $O(N)$
- Finding (if exists) a path between two vertices – $O(N^2)$

What about Time Complexity?

Operations on a Graph

Most of the basic operations in a graph are:

- Adding an edge – $O(1)$
- Deleting an edge – $O(1)$
- Answering the question “is there an edge between i and j” – $O(1)$
- Finding the successors of a given vertex – $O(N)$
- Finding (if exists) a path between two vertices – $O(N^2)$

What about Time Complexity?

Operations on a Graph

Most of the basic operations in a graph are:

- Adding an edge – $O(1)$
- Deleting an edge – $O(1)$
- Answering the question "is there an edge between i and j" – $O(1)$
- Finding the successors of a given vertex – $O(N)$
- Finding (if exists) a path between two vertices – $O(N^2)$

What about Time Complexity?

Operations on a Graph

Most of the basic operations in a graph are:

- Adding an edge – $O(1)$
- Deleting an edge – $O(1)$
- Answering the question "is there an edge between i and j" – $O(1)$
- Finding the successors of a given vertex – $O(N)$
- Finding (if exists) a path between two vertices – $O(N^2)$

DataLab
Data Science Community

What about Time Complexity?

Operations on a Graph

Most of the basic operations in a graph are:

- Adding an edge – $O(1)$
- Deleting an edge – $O(1)$
- Answering the question "is there an edge between i and j" – $O(1)$
- Finding the successors of a given vertex – $O(N)$
- Finding (if exists) a path between two vertices – $O(N^2)$

DataLab
Data Science Community

Space Drawbacks of This Representation

We need the following amount of space

If you have N integers of 4 bytes each, we requiere

$$4 \times N \times N = 4N^2$$

space

Which is a killer!!!

If your graph does not have an edge between any two pair of nodes

So

What to do?

Space Drawbacks of This Representation

We need the following amount of space

If you have N integers of 4 bytes each, we requiere

$$4 \times N \times N = 4N^2$$

space

Which is a killer!!!

If your graph does not have an edge between any two pair of nodes

So

What to do?

Space Drawbacks of This Representation

We need the following amount of space

If you have N integers of 4 bytes each, we requiere

$$4 \times N \times N = 4N^2$$

space

Which is a killer!!!

If your graph does not have an edge between any two pair of nodes

So

What to do?

Possible Solutions

If you have an undirected graph

- For an undirected graph, may store only lower or upper triangle (exclude diagonal).
- Space used (Assume Integers): $4 \times \frac{N(N-1)}{2} = 2N(N-1)$

Possible Solutions

If you have an undirected graph

- For an undirected graph, may store only lower or upper triangle (exclude diagonal).
- Space used (Assume Integers): $4 \times \frac{N(N-1)}{2} = 2N\left(N-1\right)$

Better

Use Sparse Matrix Representations

Possible Solutions

If you have an undirected graph

- For an undirected graph, may store only lower or upper triangle (exclude diagonal).
- Space used (Assume Integers): $4 \times \frac{N(N-1)}{2} = 2N\left(N-1\right)$

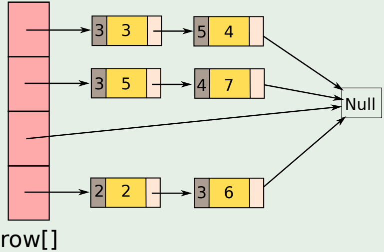
Better

Use Sparse Matrix Representations

We use the sparse Representation of Matrices!!!

Example: Array of Row Chains

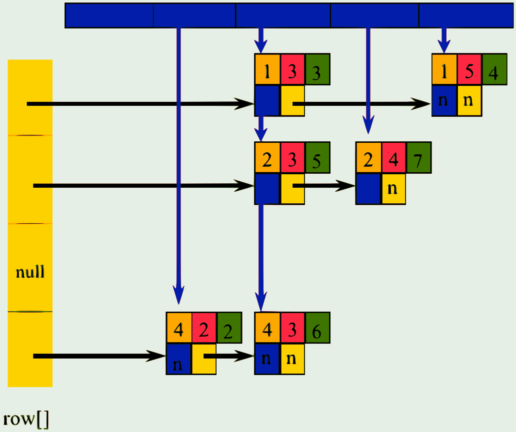
$$
\begin{matrix}
0 & 0 & 3 & 0 & 4 \\
0 & 0 & 5 & 7 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 2 & 6 & 0 & 0
\end{matrix}
$$



We use the sparse Representation of Matrices!!!

Example: Orthogonal Lists

$$\begin{matrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{matrix}$$



Outline

DataLab
Data Science Community

Adjacency List Representation

Definition

Adjacency list for vertex i is a linear list of vertices adjacent from vertex i.

Basically

An array of N adjacency lists.

Thus

Each adjacency list is a chain.

Adjacency List Representation

Definition

Adjacency list for vertex i is a linear list of vertices adjacent from vertex i.

Basically

An array of N adjacency lists.

Thus

Each adjacency list is a chain.

Adjacency List Representation

Definition

Adjacency list for vertex i is a linear list of vertices adjacent from vertex i.

Basically

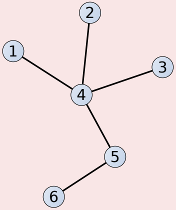
An array of N adjacency lists.

Thus

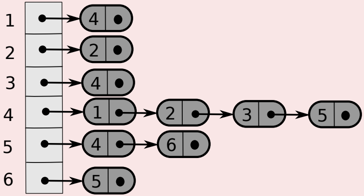
Each adjacency list is a chain.

Adjacency List Representation

For the previous example



	1	2	3	4	5	6
1	0	0	0	1	0	0
2	0	0	0	1	0	0
3	0	0	0	1	0	0
4	1	1	1	0	1	0
5	0	0	0	1	0	1
6	0	0	0	0	1	0



Properties

Space for storage

For undirected or directed graphs $O(V+E)$

Search: Successful or Unsuccessful

$O(1+degree(v))$

- Weight function $w : E \to \mathbb{R}$
- The weight $w(u,v)$ of the edge $(u,v) \in E$ is simply stored with vertex v in u's adjacency list.

Properties

Space for storage

For undirected or directed graphs $O\left(V+E\right)$

Search: Successful or Unsuccessful

$O(1+degree(v))$

In addition

Adjacency lists can readily be adapted to represent weighted graphs

- Weight function $w : E \to \mathbb{R}$
- The weight $w(u,v)$ of the edge $(u,v) \in E$ is simply stored with vertex v in u's adjacency list

DataLab
Data Science Community

Properties

Space for storage

For undirected or directed graphs $O\left(V+E\right)$

Search: Successful or Unsuccessful

$O(1+degree(v))$

In addition

Adjacency lists can readily be adapted to represent weighted graphs

- Weight function $w : E \rightarrow \mathbb{R}$
- The weight $w(u,v)$ of the edge $(u,v) \in E$ is simply stored with vertex v in u's adjacency list

Properties

Space for storage

For undirected or directed graphs $O\left(V+E\right)$

Search: Successful or Unsuccessful

$O(1+degree(v))$

In addition

Adjacency lists can readily be adapted to represent weighted graphs

- Weight function $w:E\rightarrow\mathbb{R}$
- The weight $w(u,v)$ of the edge $(u,v)\in E$ is simply stored with vertex v in u's adjacency list

Possible Disadvantage

When looking to see if an edge exist

There is no quicker way to determine if a given edge (u,v)

Outline

DataLab
Data Science Community

What about the code?

Partial Code

We can try to write it!!!

Traversing the Graph

Why?

Do you have any examples?

Traversing the Graph

Why?

Do you have any examples?

Yes

- Search for paths satisfying various constraints
  - Shortest Path
- Visit some sets of vertices
  - Tours
- Search for subgraphs
  - Isomorphisms

Traversing the Graph

Why?

Do you have any examples?

Yes

- Search for paths satisfying various constraints
  - Shortest Path
- Visit some sets of vertices
  - Tours
- Search for subgraphs
  - Isomorphisms

Traversing the Graph

Why?

Do you have any examples?

Yes

- Search for paths satisfying various constraints
  - Shortest Path
- Visit some sets of vertices
  - Tours
- Search for subgraphs
  - Isomorphisms

Traversing the Graph

Why?

Do you have any examples?

Yes

- Search for paths satisfying various constraints
  - Shortest Path
- Visit some sets of vertices
  - Tours
- Search for subgraphs
  - Isomorphisms

Traversing the Graph

Why?

Do you have any examples?

Yes

- Search for paths satisfying various constraints
  - Shortest Path
- Visit some sets of vertices
  - Tours
- Search for subgraphs
  - Isomorphisms

Traversing the Graph

Why?

Do you have any examples?

Yes

- Search for paths satisfying various constraints
  - Shortest Path
- Visit some sets of vertices
  - Tours
- Search for subgraphs
  - Isomorphisms

Outline

Breadth-first search

Definition

Given a graph $G = (V, E)$ and a source vertex s, breadth-first search systematically explores the edges of $G$to "discover" every vertex that is reachable from the vertex s

Something Notable

A vertex is discovered the first time it is encountered during the search

Breadth-first search

Definition

Given a graph $G = (V, E)$ and a source vertex s, breadth-first search systematically explores the edges of $G$to "discover" every vertex that is reachable from the vertex s

Something Notable

A vertex is discovered the first time it is **encountered** during the search

Defining Some Basic Terms

Color

- Given a node u, we have that
  - $color$ is a field indicating
    - WHITE Never visited node
    - GRAY Node pointer is at the Queue Q
    - BLACK Node has been visited and processed

Defining Some Basic Terms

Color

- Given a node u, we have that
  - $color$ is a field indicating
    - WHITE Never visited node
    - GRAY Node pointer is at the Queue Q
    - BLACK Node has been visited and processed

Defining Some Basic Terms

Color

- Given a node u, we have that
  - $color$ is a field indicating
    - ⋆ WHITE Never visited node
    - ⋆ GRAY Node pointer is at the Queue Q
    - ⋆ BLACK Node has been visited and processed

Defining Some Basic Terms

Color

- Given a node u, we have that
  - $color$ is a field indicating
    - ★ WHITE Never visited node
    - ★ GRAY Node pointer is at the Queue Q
    - ★ BLACK Node has been visited and processed

Defining Some Basic Terms

Color

- Given a node u, we have that
  - $color$ is a field indicating
    - WHITE Never visited node
    - GRAY Node pointer is at the Queue Q
    - BLACK Node has been visited and processed

Defining Some Basic Terms

Distance d

- Given a node u, we have that
  - d is a field indicating the distance from the node source s so far

Defining Some Basic Terms

Predecesor π

- Given a node u, we have that
  - π is a field indicating who is the immediate predecessor of u in the path from s to u

Breadth-First Search Algorithm

Algorithm

BFS(G, s)

for each vertex $u \in G.V - \{s\}$
  $u.color =$ WHITE
  $u.d = \infty$
  $u.\pi = NIL$
$s.color =$GRAY
$s.d = 0$
$s.\pi =$ NIL
$Q = \emptyset$
Enqueue(Q, s)
while $Q \neq \emptyset$
  $u =$Dequeue(Q)
  for each $v \in G.Adj[u]$
    if $v.color ==$WHITE
      $v.color =$GRAY
      $v.d = u.d + 1$
      $v.\pi = u$
      Enqueue(Q, v)
  $u.color =$ BLACK

Breadth-First Search Algorithm

Algorithm

BFS(G, s)

for each vertex $u \in G.V - \{s\}$
    $u.color =$ WHITE
    $u.d = \infty$
    $u.\pi = NIL$
$s.color =$GRAY
$s.d = 0$
$s.\pi =$ NIL
$Q = \emptyset$
Enqueue(Q, s)
while $Q \neq \emptyset$
    $u =$Dequeue(Q)
    for each $v \in G.Adj[u]$
        if $v.color ==$WHITE
            $v.color =$GRAY
            $v.d = u.d + 1$
            $v.\pi = u$
            Enqueue(Q, v)
    $u.color =$ BLACK

Breadth-First Search Algorithm

Algorithm

BFS(G, s)

for each vertex $u \in G.V - \{s\}$
    $u.color =$ WHITE
    $u.d = \infty$
    $u.\pi = NIL$
$s.color =$GRAY
$s.d = 0$
$s.\pi =$ NIL
$Q = \emptyset$
Enqueue(Q, s)
while $Q \neq \emptyset$
    $u =$Dequeue(Q)
    for each $v \in G.Adj[u]$
        if $v.color ==$WHITE
            $v.color =$GRAY
            $v.d = u.d + 1$
            $v.\pi = u$
            Enqueue(Q, v)
    $u.color =$ BLACK

Breadth-First Search Algorithm

Algorithm

```
BFS(G, s)
    for each vertex u ∈ G.V − {s}
        u.color = WHITE
        u.d = ∞
        u.π = NIL
    s.color = GRAY
    s.d = 0
    s.π = NIL
    Q = ∅
    Enqueue(Q, s)
    while Q ≠ ∅
        u = Dequeue(Q)
        for each v ∈ G.Adj[u]
            if v.color == WHITE
                v.color = GRAY
                v.d = u.d + 1
                v.π = u
                Enqueue(Q, v)
        u.color = BLACK
```

Breadth-First Search Algorithm

Algorithm

BFS(G, s)

1. **for** each vertex $u \in G.V - \{s\}$
2. $\quad u.color$ = WHITE
3. $\quad u.d = \infty$
4. $\quad u.\pi = NIL$
5. $s.color$ =GRAY
6. $s.d = 0$
7. $s.\pi$ = NIL
8. $Q = \emptyset$
9. Enqueue(Q, s)
10. **while** $Q \neq \emptyset$
11. $\quad u$ =Dequeue(Q)
12. $\quad$ **for** each $v \in G.Adj[u]$
13. $\quad\quad$ **if** $v.color$ ==WHITE
14. $\quad\quad\quad v.color$ =GRAY
15. $\quad\quad\quad v.d = u.d + 1$
16. $\quad\quad\quad v.\pi = u$
17. $\quad\quad\quad$ Enqueue(Q, v)
18. $\quad u.color$ = BLACK

Breadth-First Search Algorithm

Algorithm

```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
Enqueue(Q, s)
while Q ≠ ∅
    u = Dequeue(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            Enqueue(Q, v)
    u.color = BLACK
```

Breadth-First Search Algorithm

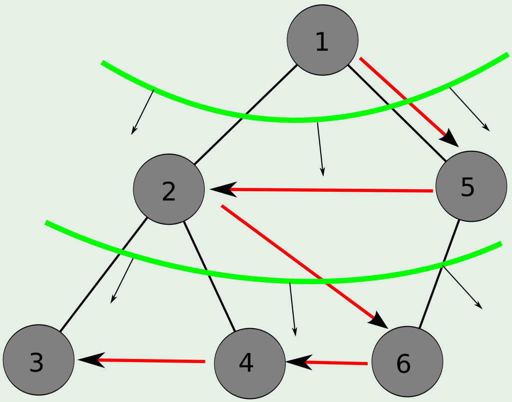
Algorithm

BFS(G, s)

for each vertex $u \in G.V - \{s\}$
    $u.color =$ WHITE
    $u.d = \infty$
    $u.\pi = NIL$
$s.color =$GRAY
$s.d = 0$
$s.\pi =$ NIL
$Q = \emptyset$
Enqueue(Q, s)
while $Q \neq \emptyset$
    $u =$Dequeue(Q)
    for each $v \in G.Adj\,[u]$
        if $v.color ==$WHITE
            $v.color =$GRAY
            $v.d = u.d + 1$
            $v.\pi = u$
            Enqueue(Q, v)
    $u.color =$ BLACK

Change the Order of Recursion

It is like a wave going from a node

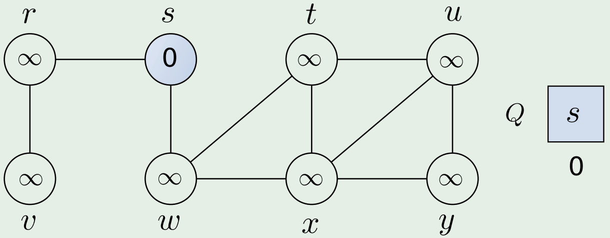


Outline

DataLab
Data Science Community

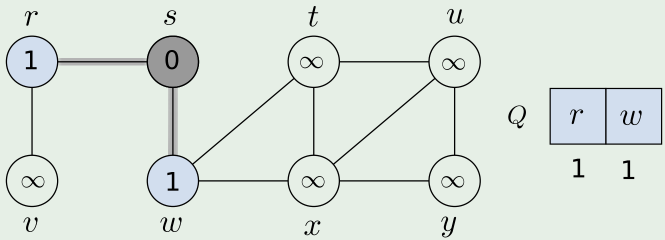
Example

What do you see?



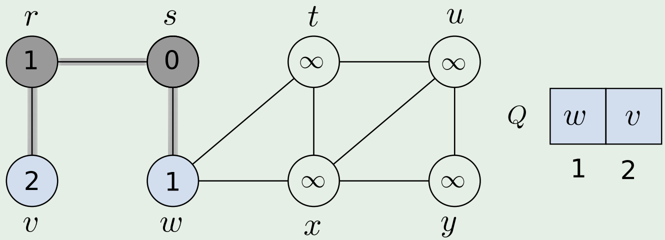
Example

What do you see?



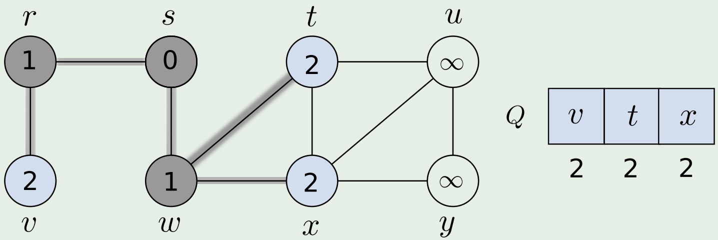
Example

What do you see?



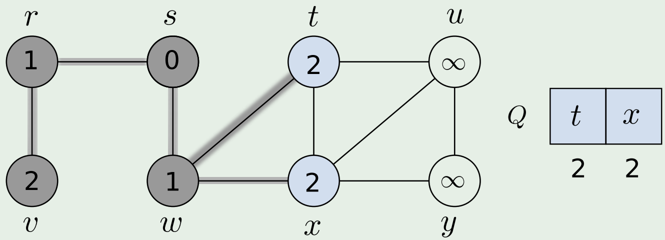
Example

What do you see?



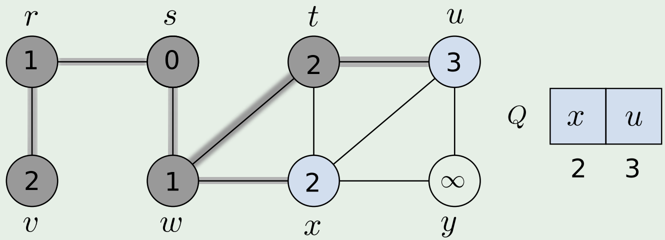
Example

What do you see?



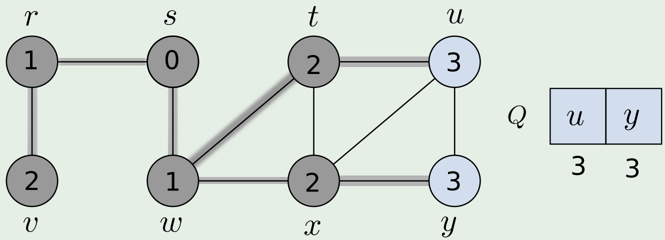
Example

What do you see?



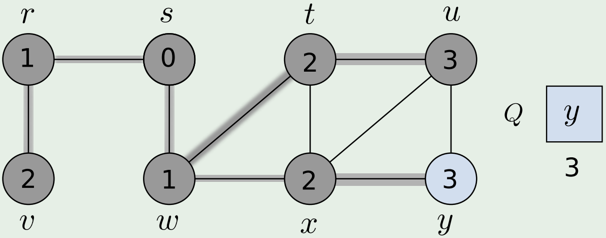
Example

What do you see?



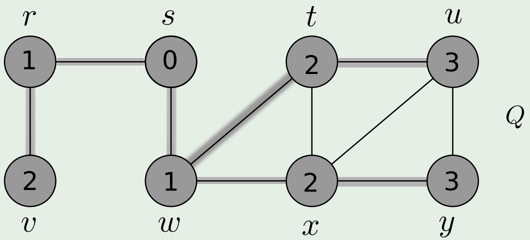
Example

What do you see?



Example

What do you see?



Outline

Complexity

What about the outer loop?

$O(V)$ Enqueue / Dequeue operations – Each adjacency list is processed only once.

What about the inner loop?

The sum of the lengths of f all the adjacency lists is $\Theta(E)$ so the scanning takes $O(E)$

Complexity

What about the outer loop?

$O(V)$ Enqueue / Dequeue operations – Each adjacency list is processed only once.

What about the inner loop?

The sum of the lengths of f all the adjacency lists is $\Theta(E)$ so the scanning takes $O(E)$

Complexity

Overhead of Creation

$O(V)$

Then

Total complexity $O(V + E)$

Complexity

Overhead of Creation

$O\left(V\right)$

Then

Total complexity $O\left(V+E\right)$

Properties: Predecessor Graph

Something Notable

Breadth-First Search constructs a **Breadth-First Tree**, initially containing only its root, which is the source vertex s

Thus

We say that u is the predecessor or parent of v in the breadth-first tree.

Properties: Predecessor Graph

Something Notable

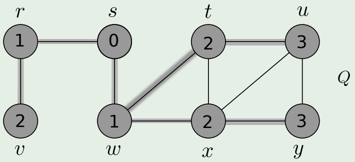
Breadth-First Search constructs a **Breadth-First Tree**, initially containing only its root, which is the source vertex s

Thus

We say that u is the predecessor or parent of v in the breadth-first tree.

For example

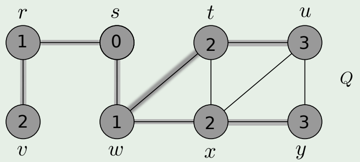
From the previous example



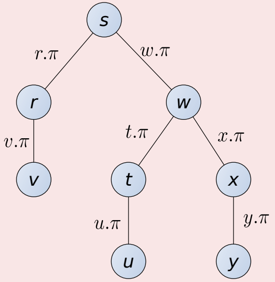
Predecessor Graph

For example

From the previous example



Predecessor Graph



This allow to use the Algorithm for finding The Shortest Path

Clearly

This is the unweighted version or all weights are equal!!!

We have the following function

$\delta(s, v)$= shortest path from s to v

We claim that

Upon termination of BFS, every vertex $v \in V$ reachable from s has

$$v.d = \delta(s, v)$$

This allow to use the Algorithm for finding The Shortest Path

Clearly

This is the unweighted version or all weights are equal!!!

We have the following function

$\delta(s, v)$= shortest path from s to v

We claim that

Upon termination of BFS, every vertex $v \in V$ reachable from s has

$$v.d = \delta(s, v)$$

This allow to use the Algorithm for finding The Shortest Path

Clearly

This is the unweighted version or all weights are equal!!!

We have the following function

$\delta\left(s,v\right)=$ shortest path from s to v

We claim that

Upon termination of BFS, every vertex $v \in V$ reachable from s has

$$v.d = \delta(s,v)$$

Outline

Depth-first search

Given G

- Pick an unvisited vertex v, remember the rest.
  - Recurse on vertices adjacent to v

Outline

The Algorithm

Code for DFS

DFS(G)

for each vertex $u \in G.V$
    $u.color =$ WHITE
    $u.\pi = NIL$
$time = 0$
for each vertex $u \in G.V$
    if $u.color =$ WHITE
        DFS-VISIT(G, u)

DFS-VISIT(G, u)

$time = time + 1$
$u.d = time$
$u.color =$ GRAY
for each vertex $v \in G.Adj[u]$
    if $v.color ==$ WHITE
        $v.\pi = u$
        DFS-VISIT(G, v)
$u.color =$ BLACK
$time = time + 1$
$u.f = time$

The Algorithm

Code for DFS

DFS(G)

for each vertex $u \in G.V$
$\quad u.color =$ WHITE
$\quad u.\pi = NIL$
$time = 0$
for each vertex $u \in G.V$
$\quad$ **if** $u.color =$ WHITE
$\quad\quad$ **DFS-VISIT**(G, u)

DFS-VISIT(G, u)

$time = time + 1$
$u.d = time$
$u.color =$ GRAY
for each vertex $v \in G.Adj[u]$
$\quad$ **if** $v.color ==$ WHITE
$\quad\quad v.\pi = u$
$\quad\quad$ **DFS-VISIT**(G, v)
$u.color =$ BLACK
$time = time + 1$
$u.f = time$

Data Science Community

The Algorithm

Code for DFS

DFS(G)

1. **for** each vertex $u \in G.V$
2. $\quad u.color =$ WHITE
3. $\quad u.\pi = NIL$
4. $time = 0$
5. **for** each vertex $u \in G.V$
6. $\quad$ **if** $u.color =$ WHITE
7. $\quad\quad$ **DFS-VISIT**(G, u)

DFS-VISIT(G, u)

1. $time = time + 1$
2. $u.d = time$
3. $u.color =$ GRAY
4. **for** each vertex $v \in G.Adj[u]$
5. $\quad$ **if** $v.color ==$ WHITE
6. $\quad\quad v.\pi = u$
7. $\quad\quad$ **DFS-VISIT**(G, v)
8. $u.color =$ BLACK
9. $time = time + 1$
10. $u.f = time$

The Algorithm

Code for DFS

DFS(G)

1. **for** each vertex $u \in G.V$
2. $\quad u.color =$ WHITE
3. $\quad u.\pi = NIL$
4. $time = 0$
5. **for** each vertex $u \in G.V$
6. $\quad$ **if** $u.color =$ WHITE
7. $\quad\quad$ **DFS-VISIT**(G, u)

DFS-VISIT(G, u)

1. $time = time + 1$
2. $u.d = time$
3. $u.color =$ GRAY
4. **for** each vertex $v \in G.Adj[u]$
5. $\quad$ **if** $v.color ==$ WHITE
6. $\quad\quad v.\pi = u$
7. $\quad\quad$ **DFS-VISIT**(G, v)
8. $u.color =$ BLACK
9. $time = time + 1$
10. $u.f = time$

The Algorithm

Code for DFS

DFS(G)

for each vertex $u \in G.V$
 $u.color =$ WHITE
 $u.\pi = NIL$
$time = 0$
for each vertex $u \in G.V$
 if $u.color =$ WHITE
  DFS-VISIT(G, u)

DFS-VISIT(G, u)

$time = time + 1$
$u.d = time$
$u.color =$ GRAY
for each vertex $v \in G.Adj[u]$
 if $v.color ==$ WHITE
  $v.\pi = u$
  DFS-VISIT(G, v)
$u.color =$ BLACK
$time = time + 1$
$u.f = time$

The Algorithm

Code for DFS

DFS(G)

for each vertex $u \in G.V$
  $u.color =$ WHITE
  $u.\pi = NIL$
$time = 0$
for each vertex $u \in G.V$
  if $u.color =$ WHITE
    DFS-VISIT(G, u)

DFS-VISIT(G, u)

$time = time + 1$
$u.d = time$
$u.color =$ GRAY
for each vertex $v \in G.Adj[u]$
  if $v.color ==$ WHITE
    $v.\pi = u$
    DFS-VISIT(G, v)
$u.color =$ BLACK
$time = time + 1$
$u.f = time$

The Algorithm

Code for DFS

```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color = WHITE
        DFS-VISIT(G, u)
```

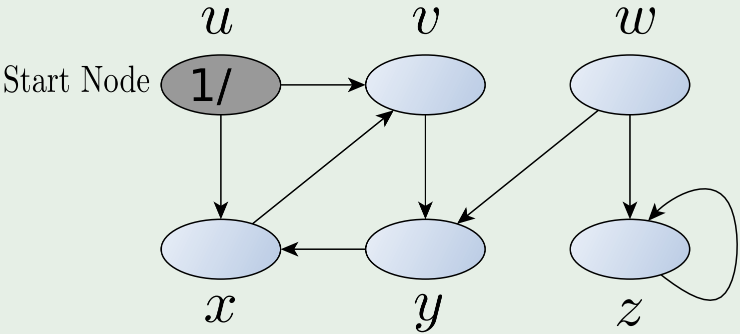
```
DFS-VISIT(G, u)
time = time + 1
u.d = time
u.color = GRAY
for each vertex v ∈ G.Adj[u]
    if v.color == WHITE
        v.π = u
        DFS-VISIT(G, v)
u.color = BLACK
time = time + 1
u.f = time
```

Outline

DataLab
Data Science Community

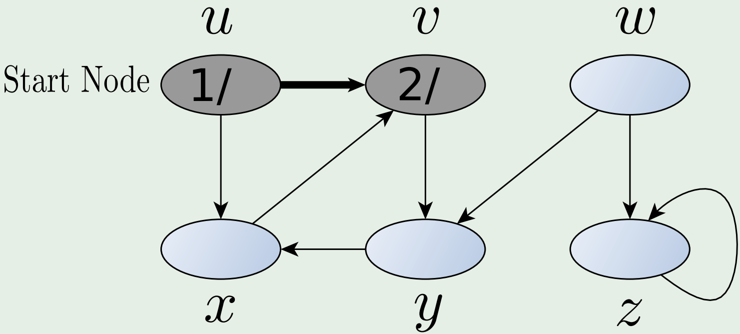
Example

What do we do?



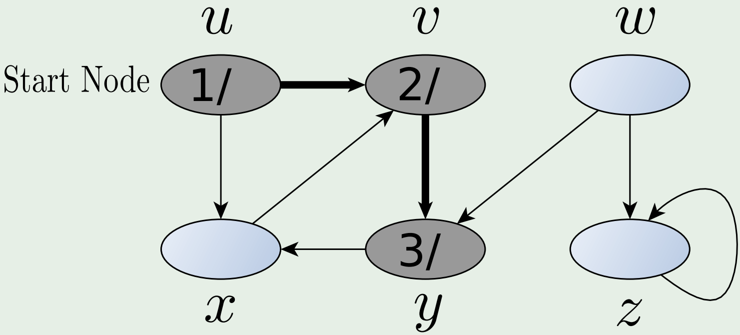
Example

What do we do?



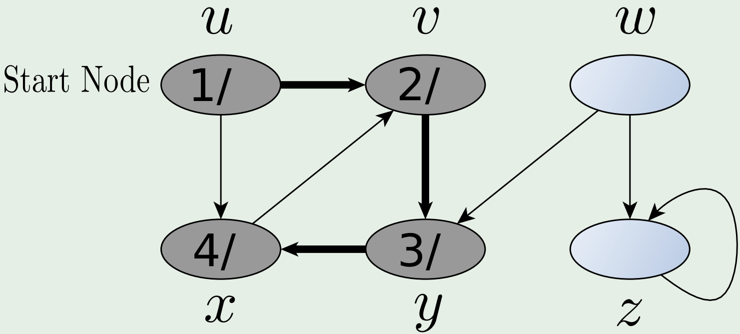
Example

What do we do?



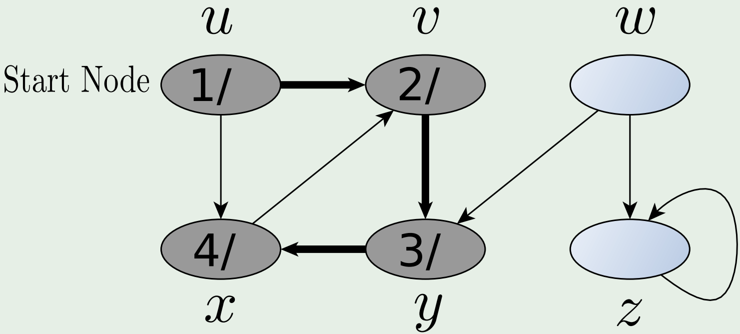
Example

What do we do?



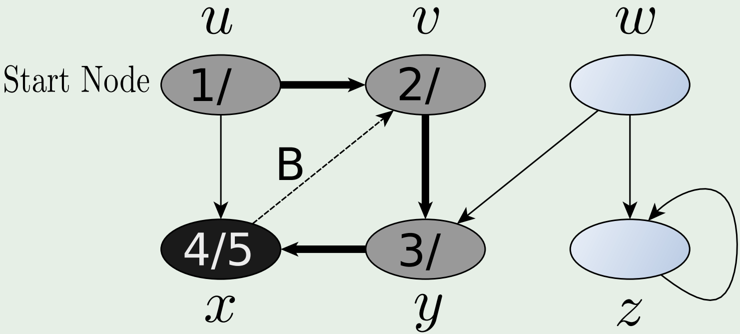
Example

What do we do?



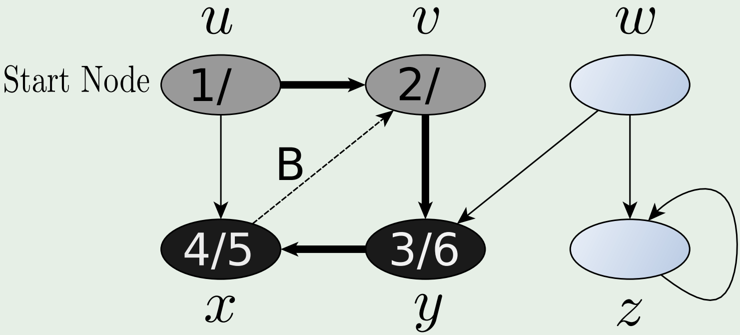
Example

What do we do?



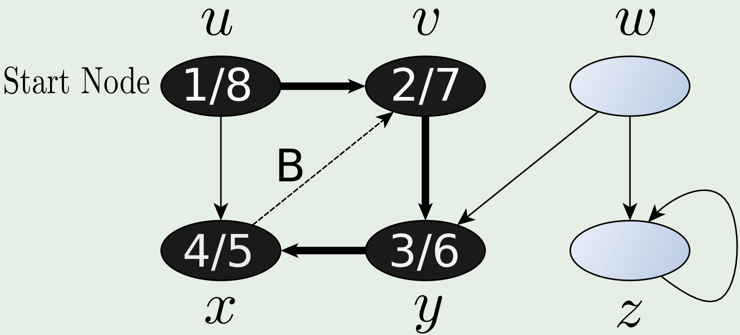
Example

What do we do?



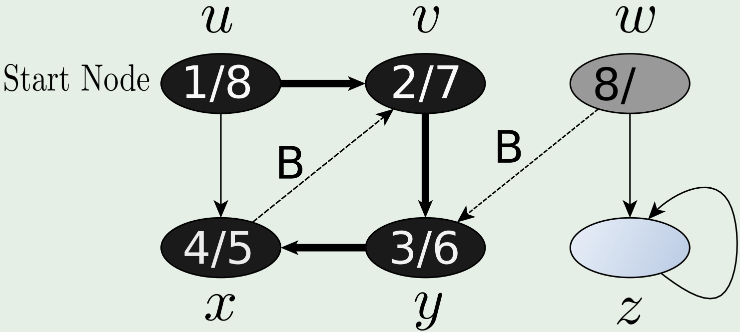
Example

Then, we keep going...



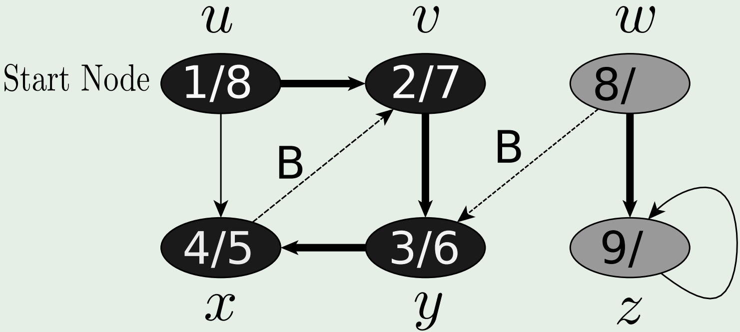
Example

Then, we keep going...



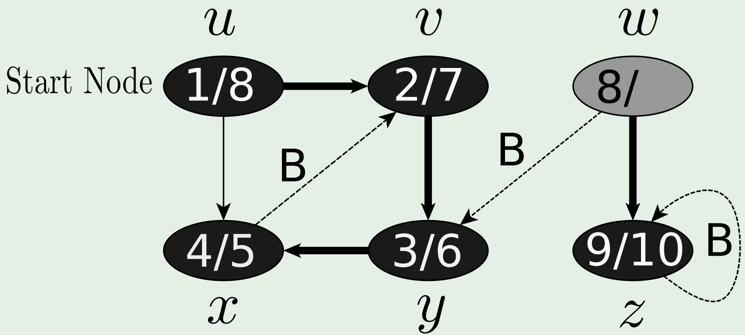
Example

Then, we keep going...



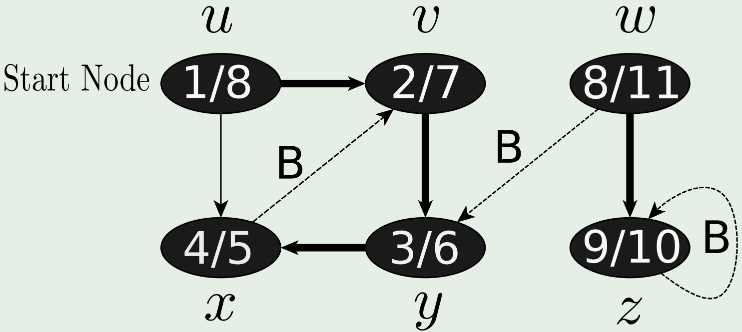
Example

Then, we keep going...



Example

Then, we keep going...



Outline

Complexity

Analysis

1. The loops on lines 1–3 and lines 5–7 of DFS take $\Theta(V)$.
2. The procedure DFS-VISIT is called exactly once for each vertex $v \in V$.
3. During an execution of DFS-VISIT(G, v) the loop on lines 4–7 executes $|Adj(v)|$ times.
4. But $\sum_{v \in V} |Adj(v)| = \Theta(E)$ we have that the cost of executing g lines 4–7 of DFS-VISIT is $\Theta(E)$.

DataLab
Data Science Community

Complexity

Analysis

1. The loops on lines 1–3 and lines 5–7 of DFS take $\Theta(V)$.
2. The procedure DFS-VISIT is called exactly once for each vertex $v \in V$.
3. During an execution of DFS-VISIT(G, v) the loop on lines 4–7 executes $|Adj(v)|$ times.
4. But $\sum_{v \in V} |Adj(v)| = \Theta(E)$ we have that the cost of executing g lines 4–7 of DFS-VISIT is $\Theta(E)$.

Then

DFS complexity is $\Theta(V + E)$

Complexity

Analysis

1. The loops on lines 1–3 and lines 5–7 of DFS take $\Theta(V)$.
2. The procedure DFS-VISIT is called exactly once for each vertex $v \in V$.
3. During an execution of DFS-VISIT(G, v) the loop on lines 4–7 executes $|Adj(v)|$ times.
4. But $\sum_{v \in V} |Adj(v)| = \Theta(E)$ we have that the cost of executing g lines 4–7 of DFS-VISIT is $\Theta(E)$.

Then

DFS complexity is $\Theta(V + E)$

Complexity

Analysis

1. The loops on lines 1–3 and lines 5–7 of DFS take $\Theta(V)$.
2. The procedure DFS-VISIT is called exactly once for each vertex $v \in V$.
3. During an execution of DFS-VISIT(G, v) the loop on lines 4–7 executes $|Adj(v)|$ times.
4. But $\sum_{v \in V} |Adj(v)| = \Theta(E)$ we have that the cost of executing g lines 4–7 of DFS-VISIT is $\Theta(E)$.

Then

DFS complexity is $\Theta(V + E)$

DataLab
Data Science Community

Complexity

Analysis

1. The loops on lines 1–3 and lines 5–7 of DFS take $\Theta(V)$.
2. The procedure DFS-VISIT is called exactly once for each vertex $v \in V$.
3. During an execution of DFS-VISIT(G, v) the loop on lines 4–7 executes $|Adj(v)|$ times.
4. But $\sum_{v \in V} |Adj(v)| = \Theta(E)$ we have that the cost of executing g lines 4–7 of DFS-VISIT is $\Theta(E)$.

Then

DFS complexity is $\Theta(V + E)$

Applications

We have several

- Finding a path between nodes
- Strongly Connected Components
- Spanning Trees
- Topological Sort - The Program (or Project) Evaluation and Review (PERT)
- Computer Vision Algorithms
- Artificial Intelligence Algorithms
- Importance in Social Network
- Rank Algorithms for Google
- Etc.

Applications

We have several

- Finding a path between nodes
- Strongly Connected Components
- Spanning Trees
- Topological Sort - The Program (or Project) Evaluation and Review (PERT)
- Computer Vision Algorithms
- Artificial Intelligence Algorithms
- Importance in Social Network
- Rank Algorithms for Google
- Etc.

Applications

We have several

- Finding a path between nodes
- Strongly Connected Components
- Spanning Trees
- Topological Sort - The Program (or Project) Evaluation and Review (PERT)
- Computer Vision Algorithms
- Artificial Intelligence Algorithms
- Importance in Social Network
- Rank Algorithms for Google
- Etc.

DataLab
Data Science Community

Applications

We have several

- Finding a path between nodes
- Strongly Connected Components
- Spanning Trees
- Topological Sort - The Program (or Project) Evaluation and Review (PERT)
- Computer Vision Algorithms
- Artificial Intelligence Algorithms
- Importance in Social Network
- Rank Algorithms for Google
- Etc.

Applications

We have several

- Finding a path between nodes
- Strongly Connected Components
- Spanning Trees
- Topological Sort - The Program (or Project) Evaluation and Review (PERT)
- Computer Vision Algorithms
- Artificial Intelligence Algorithms
- Importance in Social Network
- Rank Algorithms for Google
- Etc.

Applications

We have several

- Finding a path between nodes
- Strongly Connected Components
- Spanning Trees
- Topological Sort - The Program (or Project) Evaluation and Review (PERT)
- Computer Vision Algorithms
- Artificial Intelligence Algorithms
- Importance in Social Network
- Rank Algorithms for Google
- Etc.

Applications

We have several

- Finding a path between nodes
- Strongly Connected Components
- Spanning Trees
- Topological Sort - The Program (or Project) Evaluation and Review (PERT)
- Computer Vision Algorithms
- Artificial Intelligence Algorithms
- Importance in Social Network
- Rank Algorithms for Google
- Etc.

Applications

We have several

- Finding a path between nodes
- Strongly Connected Components
- Spanning Trees
- Topological Sort - The Program (or Project) Evaluation and Review (PERT)
- Computer Vision Algorithms
- Artificial Intelligence Algorithms
- Importance in Social Network
- Rank Algorithms for Google
- Etc.

Applications

We have several

- Finding a path between nodes
- Strongly Connected Components
- Spanning Trees
- Topological Sort - The Program (or Project) Evaluation and Review (PERT)
- Computer Vision Algorithms
- Artificial Intelligence Algorithms
- Importance in Social Network
- Rank Algorithms for Google
- Etc.

Outline

Finding a path between nodes

We do the following

- Start a breadth-first search at vertex v.
- Terminate when vertex u is visited or when Q becomes empty (whichever occurs first).

Finding a path between nodes

We do the following

- Start a breadth-first search at vertex v.
- Terminate when vertex u is visited or when Q becomes empty (whichever occurs first).

Time Complexity

- $O\left(V^2\right)$when adjacency matrix used.
- $O(V+E)$ when adjacency lists used.

Finding a path between nodes

We do the following

- Start a breadth-first search at vertex v.
- Terminate when vertex u is visited or when Q becomes empty (whichever occurs first).

Time Complexity

- $O\left(V^2\right)$when adjacency matrix used.
- $O(V+E)$ when adjacency lists used.

Finding a path between nodes

We do the following

- Start a breadth-first search at vertex v.
- Terminate when vertex u is visited or when Q becomes empty (whichever occurs first).

Time Complexity

- $O\left(V^2\right)$when adjacency matrix used.
- $O(V+E)$ when adjacency lists used.

This allow to use the Algorithm for finding The Shortest Path

Clearly

This is the unweighted version or all weights are equal!!!

We have the following function

$\delta(s, v)$= shortest path from s to v

We claim that

Upon termination of BFS, every vertex $v \in V$ reachable from s has distance(v) $= \delta(s, v)$

This allow to use the Algorithm for finding The Shortest Path

Clearly

This is the unweighted version or all weights are equal!!!

We have the following function

$\delta(s, v)$= shortest path from s to v

We claim that

Upon termination of BFS, every vertex $v \in V$ reachable from s has distance(v) $= \delta(s, v)$

This allow to use the Algorithm for finding The Shortest Path

Clearly

This is the unweighted version or all weights are equal!!!

We have the following function

$\delta(s,v)$= shortest path from s to v

We claim that

Upon termination of BFS, every vertex $v \in V$ reachable from s has distance$(v) = \delta(s,v)$

Outline

DataLab
Data Science Community

Connected Components

Definition

A connected component (or just component) of an undirected graph is a subgraph in which any two vertices are connected to each other by paths.

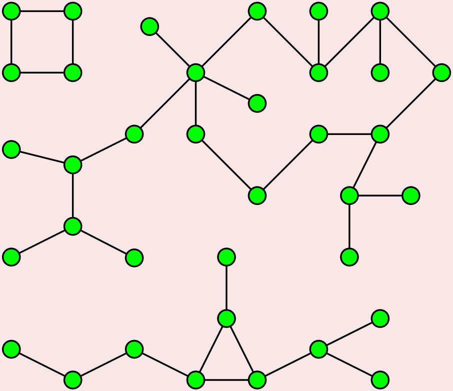
Example

Connected Components

Definition

A connected component (or just component) of an undirected graph is a subgraph in which any two vertices are connected to each other by paths.

Example



Procedure

First

Start a breadth-first search at any as yet unvisited vertex of the graph.

Thus

Newly visited vertices (plus edges between them) define a component.

Repeat

Repeat until all vertices are visited.

Procedure

First

Start a breadth-first search at any as yet unvisited vertex of the graph.

Thus

Newly visited vertices (plus edges between them) define a component.

Repeat

Repeat until all vertices are visited.

Procedure

First

Start a breadth-first search at any as yet unvisited vertex of the graph.

Thus

Newly visited vertices (plus edges between them) define a component.

Repeat

Repeat until all vertices are visited.

Time

$O\left(V^2\right)$

When adjacency matrix used

$O(V + E)$

When adjacency lists used (E is number of edges)

Time

$O\left(V^2\right)$

When adjacency matrix used

$O(V + E)$

When adjacency lists used (E is number of edges)

Outline

Spanning Tree with edges with same weight of no weight

Definition

A spanning tree of a graph $G = (V, E)$ is a acyclic graph where for $u, v \in V$, there is a path between them

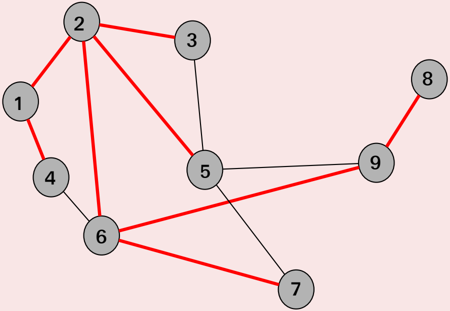
Example

Spanning Tree with edges with same weight of no weight

Definition

A spanning tree of a graph $G = (V, E)$ is a acyclic graph where for $u, v \in V$, there is a path between them

Example



Procedure

First

Start a **Breadth-First Search** at any vertex of the graph.

Thus

If graph is connected, the $n-1$ edges used to get to **unvisited vertices** define a spanning tree (**Breadth-First Spanning Tree**).

Procedure

First

Start a **Breadth-First Search** at any vertex of the graph.

Thus

If graph is connected, the $n-1$ edges used to get to **unvisited vertices** define a spanning tree (**Breadth-First Spanning Tree**).

Time

$O\left(V^2\right)$

When adjacency matrix used

$O(V+E)$

When adjacency lists used (E is number of edges)

Time

$O\left(V^2\right)$

When adjacency matrix used

$O(V+E)$

When adjacency lists used (E is number of edges)

Outline

DataLab
Data Science Community

Topological Sorting

Definitions

A topological sort (sometimes abbreviated topsort or toposort) or topological ordering of a directed graph is a linear ordering of its vertices such that for every directed edge (u, v) from vertex u to vertex y, u comes before v in the ordering.

DataLab
Data Science Community

Topological Sorting

Definitions

A topological sort (sometimes abbreviated topsort or toposort) or topological ordering of a directed graph is a linear ordering of its vertices such that for every directed edge (u, v) from vertex u to vertex y, u comes before v in the ordering.

From Industrial Engineering

- The canonical application of topological sorting (topological order) is in scheduling a sequence of jobs or tasks based on their dependencies.
- Topological sorting algorithms were first studied in the early 1960s in the context of the PERT technique for scheduling in project management (Jarnagin 1960).

Topological Sorting

Definitions

A topological sort (sometimes abbreviated topsort or toposort) or topological ordering of a directed graph is a linear ordering of its vertices such that for every directed edge (u, v) from vertex u to vertex y, u comes before v in the ordering.

From Industrial Engineering

- The canonical application of topological sorting (topological order) is in scheduling a sequence of jobs or tasks based on their dependencies.
- Topological sorting algorithms were first studied in the early 1960s in the context of the PERT technique for scheduling in project management (Jarnagin 1960).

DataLab
Data Science Community

Then

We have that

The jobs are represented by vertices, and there is an edge from x to y if job x must be completed before job y can be started.

Example

When washing clothes, the washing machine must finish before we put the clothes to dry.

Then

A topological sort gives an order in which to perform the jobs.

Then

We have that

The jobs are represented by vertices, and there is an edge from x to y if job x must be completed before job y can be started.

Example

When washing clothes, the washing machine must finish before we put the clothes to dry.

Then

A topological sort gives an order in which to perform the jobs.

Then

We have that

The jobs are represented by vertices, and there is an edge from x to y if job x must be completed before job y can be started.

Example

When washing clothes, the washing machine must finish before we put the clothes to dry.

Then

A topological sort gives an order in which to perform the jobs.

Algorithm

TOPOLOGICAL-SORT

1. Call DFS(G) to compute finishing times $v.f$ for each vertex v.
2. As each vertex is finished, insert it onto the front of a linked list
3. Return the linked list of vertices

Algorithm

TOPOLOGICAL-SORT

1. Call DFS(G) to compute finishing times $v.f$ for each vertex v.
2. As each vertex is finished, insert it onto the front of a linked list
3. Return the linked list of vertices

DataLab
Data Science Community

Algorithm

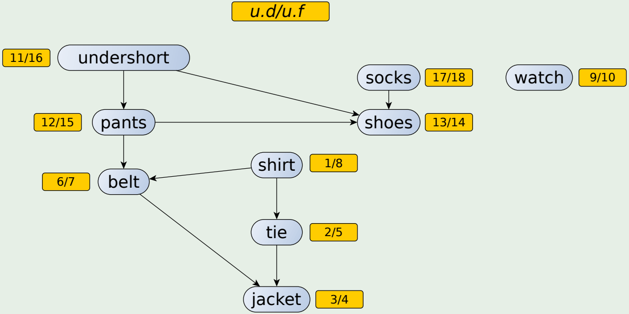
TOPOLOGICAL-SORT

1. Call DFS(G) to compute finishing times $v.f$ for each vertex v.
2. As each vertex is finished, insert it onto the front of a linked list
3. Return the linked list of vertices

DataLab
Data Science Community

Example

Dressing



Thus

Using the $u.f$

As each vertex is finished, insert it onto the front of a linked list

Example

After Sorting