Analysis of Algorithms

Basic Graph Algorithms

Andres Mendez-Vazquez

October 28, 2015

Outline

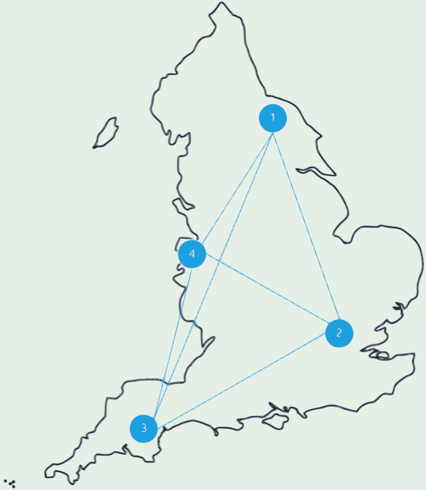
Cinvestav

Outline

Cinvestav

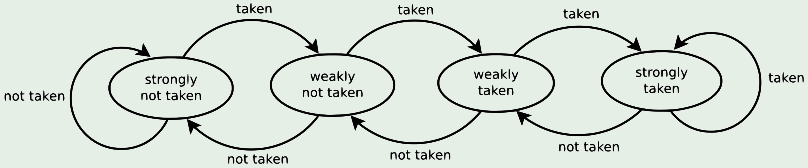
We are full of Graphs

Maps



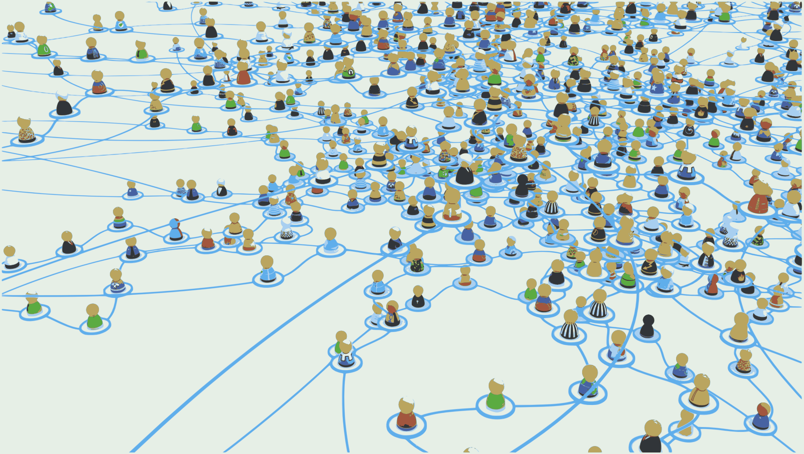
We are full of Graphs

Branch CPU estimators



We are full of Graphs

Social Networks



Outline

Cinvestav

History

Something Notable

Graph theory started with Euler who was asked to find a nice path across the seven Königsberg bridges

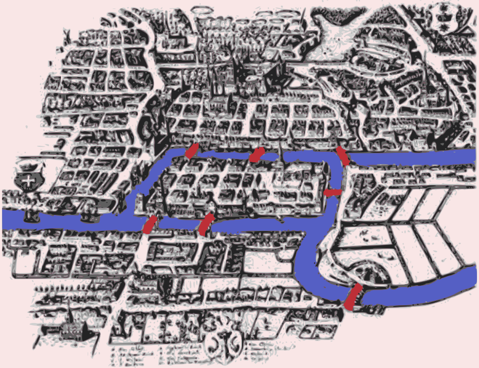
The Actual City

History

Something Notable

Graph theory started with Euler who was asked to find a nice path across the seven Königsberg bridges

The Actual City



No solution for a odd number of Bridges

What we want

The (Eulerian) path should cross over each of the seven bridges exactly once

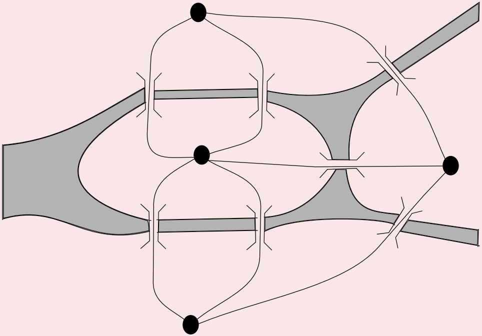
We cannot do this for the original problem

No solution for a odd number of Bridges

What we want

The (Eulerian) path should cross over each of the seven bridges exactly once

We cannot do this for the original problem



Necessary Condition

Euler discovered that

A necessary condition for the walk of the desired form is that the graph be connected and have exactly zero or two nodes of odd degree.

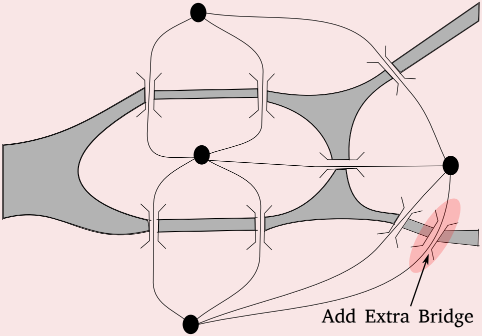
Add an extra bridge

Necessary Condition

Euler discovered that

A necessary condition for the walk of the desired form is that the graph be connected and have exactly zero or two nodes of odd degree.

Add an extra bridge



Studying Graphs

All the previous examples are telling us

Data Structures are required to design structures to hold the information coming from graphs!!!

Good Representations

They will allow to handle the data structures with ease!!!

Studying Graphs

All the previous examples are telling us

Data Structures are required to design structures to hold the information coming from graphs!!!

Good Representations

They will allow to handle the data structures with ease!!!

Cinvestav

Outline

Basic Theory

Definition

A Graph is composed of the following parts: **Nodes** and **Edges**

Nodes

They can represent multiple things:

- Distance between cities
- Friendships
- Matching Strings
- Etc

Basic Theory

Definition

A Graph is composed of the following parts: **Nodes** and **Edges**

Nodes

They can represent multiple things:

- People
- Cities
- States of Being
- etc

- Distance between cities
- Friendships
- Matching Strings
- Etc

Basic Theory

Definition

A Graph is composed of the following parts: **Nodes** and **Edges**

Nodes

They can represent multiple things:

- People
- Cities
- States of Being
- etc

Edges

They can represent multiple things too:

- Distance between cities
- Friendships
- Matching Strings
- Etc

Basic Theory

Definition

A Graph is composed of the following parts: **Nodes** and **Edges**

Nodes

They can represent multiple things:

- People
- Cities
- States of Being
- etc

Edges

They can represent multiple things too:

- Distance between cities
- Friendships
- Matching Strings
- Etc

Basic Theory

Definition

A Graph is composed of the following parts: **Nodes** and **Edges**

Nodes

They can represent multiple things:

- People
- Cities
- States of Being
- etc

Edges

They can represent multiple things too:

- Distance between cities
- Friendships
- Matching Strings
- Etc

Basic Theory

Definition

A Graph is composed of the following parts: **Nodes** and **Edges**

Nodes

They can represent multiple things:

- People
- Cities
- States of Being
- etc

Edges

They can represent multiple things too:

- Distance between cities
- Friendships
- Matching Strings
- Etc

Basic Theory

Definition

A Graph is composed of the following parts: **Nodes** and **Edges**

Nodes

They can represent multiple things:

- People
- Cities
- States of Being
- etc

Edges

They can represent multiple things too:

- Distance between cities
- Friendships
- Matching Strings
- Etc

Basic Theory

Definition

A Graph is composed of the following parts: **Nodes** and **Edges**

Nodes

They can represent multiple things:

- People
- Cities
- States of Being
- etc

Edges

They can represent multiple things too:

- Distance between cities
- Friendships
- Matching Strings
- Etc

Basic Theory

Definition

A Graph is composed of the following parts: **Nodes** and **Edges**

Nodes

They can represent multiple things:

- People
- Cities
- States of Being
- etc

Edges

They can represent multiple things too:

- Distance between cities
- Friendships
- Matching Strings
- Etc

Basic Theory

Definition

A Graph is composed of the following parts: **Nodes** and **Edges**

Nodes

They can represent multiple things:

- People
- Cities
- States of Being
- etc

Edges

They can represent multiple things too:

- Distance between cities
- Friendships
- Matching Strings
- Etc

Basic Theory

Definition

A graph $G = (V, E)$ is composed of a set of vertices (or nodes) V and a set of edges E, each assumed finite i.e. $|V| = n$ and $|E| = m$.

Example

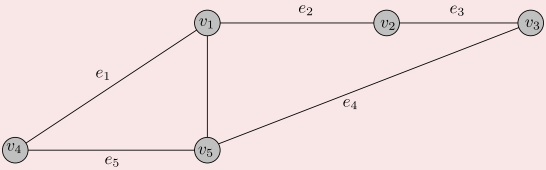
Cinvestav

Basic Theory

Definition

A graph $G = (V, E)$ is composed of a set of vertices (or nodes) V and a set of edges E, each assumed finite i.e. $|V| = n$ and $|E| = m$.

Example



Properties

Incident

An edge $e_k = (v_i, v_j)$ is incident with the vertices v_i and v_j.

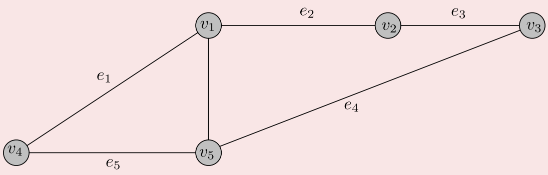
A simple graph has no self-loops or multiple edges like below

Properties

Incident

An edge $e_k = (v_i, v_j)$ is incident with the vertices v_i and v_j.

A simple graph has no self-loops or multiple edges like below



Some properties

Degree

The degree $d(v)$ of a vertex V is its number of incident edges

A self loop

A self-loop counts for 2 in the degree function.

Proposition

The sum of the degrees of a graph $G = (V, E)$ equals $2|E| = 2m$ (trivial).

Cinvestav

Some properties

Degree

The degree $d(v)$ of a vertex V is its number of incident edges

A self loop

A self-loop counts for 2 in the degree function.

Proposition

The sum of the degrees of a graph $G = (V, E)$ equals $2|E| = 2m$ (trivial).

Some properties

Degree

The degree $d(v)$ of a vertex V is its number of incident edges

A self loop

A self-loop counts for 2 in the degree function.

Proposition

The sum of the degrees of a graph $G = (V, E)$ equals $2|E| = 2m$ (trivial).

Some properties

Complete

A complete graph K_n is a simple graph with all $n(n-1)/2$ possible edges, like the graph below for $n = 2, 3, 4, 5$.

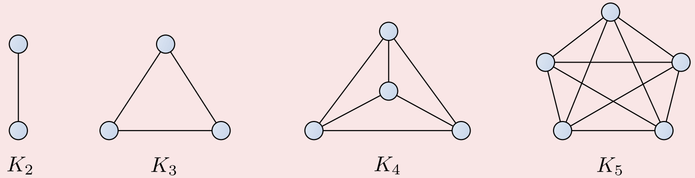
Example

Some properties

Complete

A complete graph K_n is a simple graph with all $^{n(n-1)}/_{2}$ possible edges, like the graph below for $n = 2, 3, 4, 5$.

Example



Outline

Cinvestav

Clearly

We need to represent

- Nodes
- Vertices

We need to represent

- Directed edges
- Undirected edges

Clearly

We need to represent

- Nodes
- Vertices

We need to represent

- Directed edges
- Undirected edges

We need NICE representations of this definition

First One

Adjacency Representation

Second One

Matrix Representation

We need NICE representations of this definition

First One

Adjacency Representation

Second One

Matrix Representation

Adjacency-list representation

Basic Definition

It is an array of size $|V|$ with

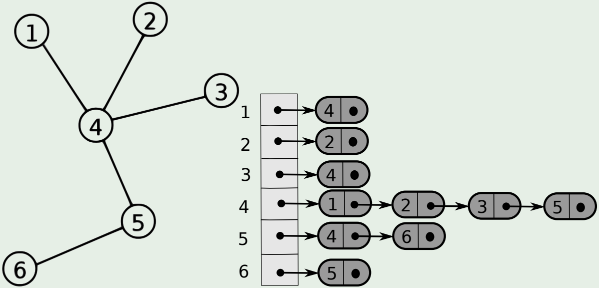
- A list for each bucket representing a node telling us which nodes are connected to it by one edge

Adjacency-list representation

Basic Definition

It is an array of size $|V|$ with

- A list for each bucket representing a node telling us which nodes are connected to it by one edge



Properties

Space for storage

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

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

Cinvestav

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

Cinvestav

Possible Disadvantage

When looking to see if an edge exist

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

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

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

You say it!!

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

Cinvestav

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

Cinvestav

We have then...

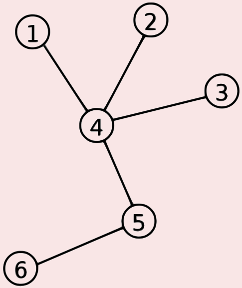
Definition

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

Cinvestav

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

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:

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

Cinvestav

Complexity

Memory

$$\Theta\left(V^2\right) \tag{1}$$

Looking for an edge

$O(1)$

Complexity

Memory

$$\Theta\left(V^2\right) \tag{1}$$

Looking for an edge

$O(1)$

Traversing the Graph

Why do we need to traverse the graph?

Do you have any examples?

Yes

- Search for paths satisfying various constraints
  - Shortest Path

Traversing the Graph

Why do we need to traverse the graph?

Do you have any examples?

Yes

- Search for paths satisfying various constraints
  - Shortest Path
- Visit some sets of vertices
  - Tours
- Search if two graphs are equivalent
  - Isomorphisms

Traversing the Graph

Why do we need to traverse the graph?

Do you have any examples?

Yes

- Search for paths satisfying various constraints
  - Shortest Path
- Visit some sets of vertices
  - Tours
- Search if two graphs are equivalent
  - Isomorphisms

Cinvestav

Traversing the Graph

Why do we need to traverse the graph?

Do you have any examples?

Yes

- Search for paths satisfying various constraints
  - Shortest Path
- Visit some sets of vertices
  - Tours
- Search if two graphs are equivalent
  - Isomorphisms

Cinvestav

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

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)

```
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

Breadth-First Search Algorithm

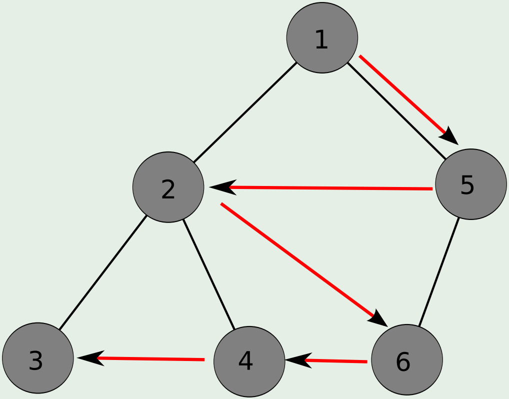
Algorithm

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

BFS allows to change the order of recursion

Remember



Loop Invariance

The While loop

This while loop maintains the following invariant :

- At the test in line 10, the queue Q consists of the set of gray vertices

First iteration

$Q = s$ and $s.color$ = GRAY

Maintenance

The inner loop only pushes gray nodes into the queue.

Loop Invariance

The While loop

This while loop maintains the following invariant :

- At the test in line 10, the queue Q consists of the set of gray vertices

First iteration

$Q = s$ and $s.color =$ GRAY

Maintenance

The inner loop only pushes gray nodes into the queue.

Loop Invariance

The While loop

This while loop maintains the following invariant :

- At the test in line 10, the queue Q consists of the set of gray vertices

First iteration

$Q = s$and $s.color =$ GRAY

Maintenance

The inner loop only pushes gray nodes into the queue.

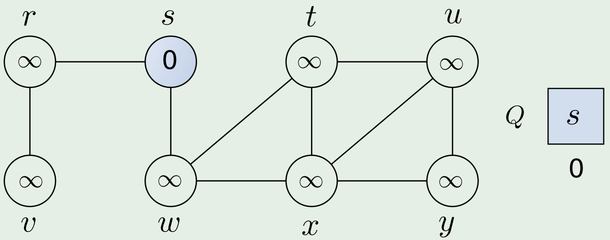
Loop Invariance

Termination

When every node that can be visited is painted black

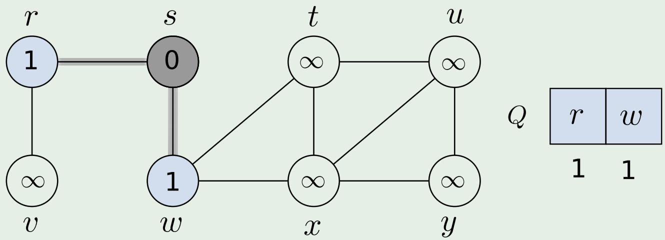
Example

What do you see?



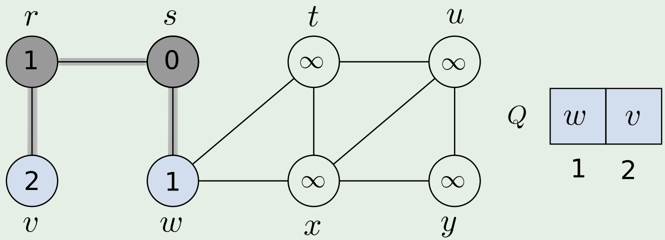
Example

What do you see?



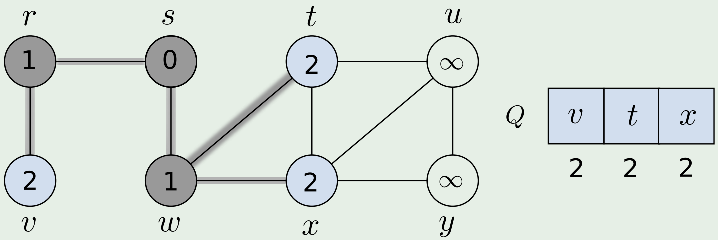
Example

What do you see?



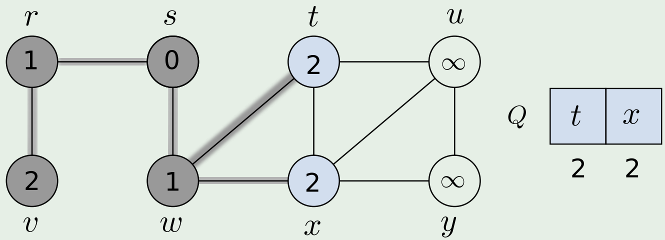
Example

What do you see?



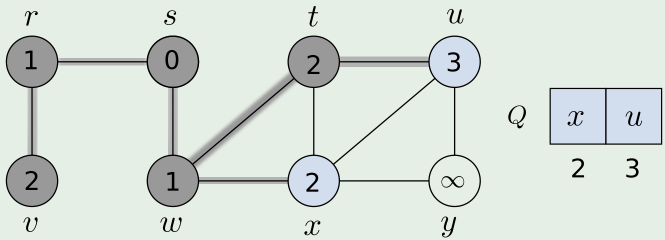
Example

What do you see?



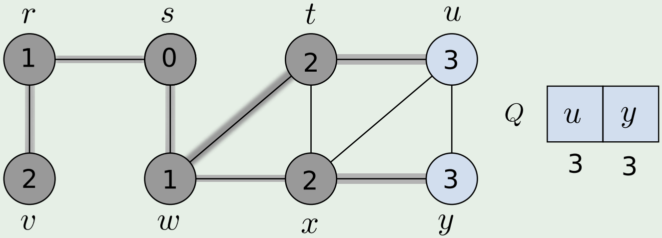
Example

What do you see?



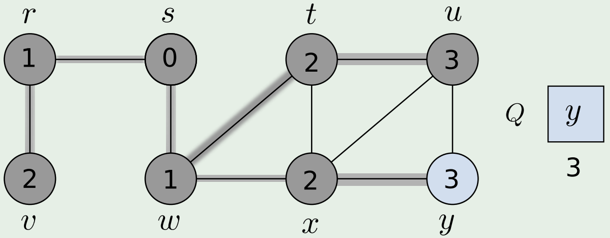
Example

What do you see?



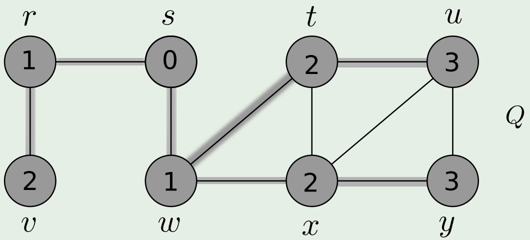
Example

What do you see?



Example

What do you see?



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 $O(V+E)$

Cinvestav

Complexity

Overhead of Creation

$O(V)$

Then

Total complexity $O(V+E)$

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

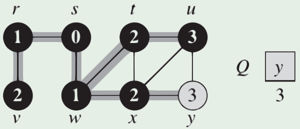
Breadth-first search constructs a breadth-first tree, initially containing only its root, which is the source vertex s

Thus

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

For example

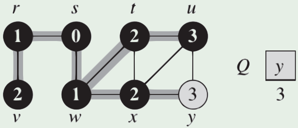
From the previous example



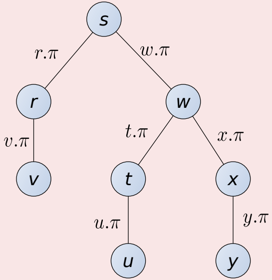
Predecessor Graph

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

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

We claim that

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

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

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

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

Cinvestav

Intuitive Idea of Claim

Correctness of breadth-first search

- Let $G = (V, E)$ be a directed or undirected graph.
- Suppose that BFS is run on G from a given source vertex $s \in V$.

Intuitive Idea of Claim

Correctness of breadth-first search

- Let $G = (V, E)$ be a directed or undirected graph.
- Suppose that BFS is run on G from a given source vertex $s \in V$.

Then

Then, during its execution, BFS discovers every vertex $v \in V$ that is reachable from the source.

Cinvestav

Intuitive Idea of Claim

Correctness of breadth-first search

- Let $G = (V, E)$ be a directed or undirected graph.
- Suppose that BFS is run on G from a given source vertex $s \in V$.

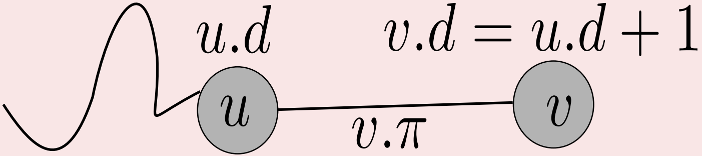
Then

Then, during its execution, BFS discovers every vertex $v \in V$ that is reachable from the source.

Intuitive Idea of Claim

Distance Idea

- First, once a node u is reached from v, we use the previous shortest distance from s to update the node distance.



Intuitive Idea of Claim

You can use the idea of strong induction

- Given a branch of the breadth-first search $u\ s, u_1, u_2, ..., u_n$

$$u_n.\pi = u_{n-1}.\pi + 1 = u_{n-1}.\pi + 2 = n + s.\pi = n \tag{2}$$

Thus, we have that

- For any vertex $v \neq s$ that is reachable from s, one of the shortest path from s to v is
  - **A shortest path from s to $v.\pi$ followed by thew edge $(v.\pi, v)$.**

Intuitive Idea of Claim

You can use the idea of strong induction

- Given a branch of the breadth-first search u $s, u_1, u_2, ..., u_n$

$$u_n.\pi = u_{n-1}.\pi + 1 = u_{n-1}.\pi + 2 = n + s.\pi = n \tag{2}$$

Thus, we have that

- For any vertex $v \neq s$ that is reachable from s, one of the shortest path from s to v is
  - **A shortest path from s to $v.\pi$ followed by thew edge $(v.\pi, v)$.**

Cinvestav

Outline

1. Introduction
   - Graphs Everywhere
   - History
   - Basic Theory
   - Representing Graphs in a Computer
2. Traversing the Graph
   - Breadth-first search
   - Depth-First Search
3. Applications
   - Finding a path between nodes
   - Connected Components
   - Spanning Trees
   - Topological Sorting

Cinvestav

Depth-first search

Given G

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

The Pseudo-code

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

$time = time + 1$

$u.f = time$

The Pseudo-code

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 Pseudo-code

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
```

The Pseudo-code

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 Pseudo-code

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 Pseudo-code

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 Pseudo-code

Code for DFS

DFS(G)

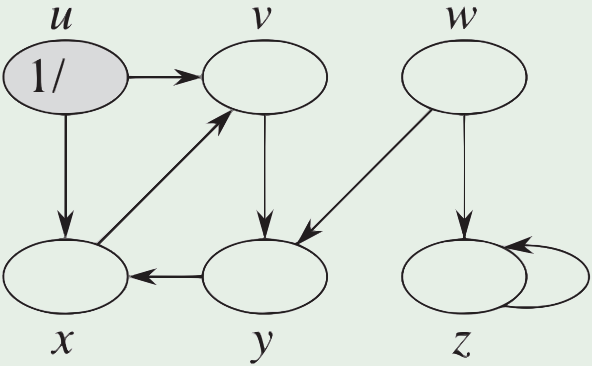
for each vertex $u \in G.V$
    $u.color =$ WHITE
    $u.\pi = NIL$
$time = 0$
for each vertex $u \in G.V$
    if $u.color =$ WHITE
        DFS-VISIT(G, u)

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
$time = time + 1$
$u.f = time$

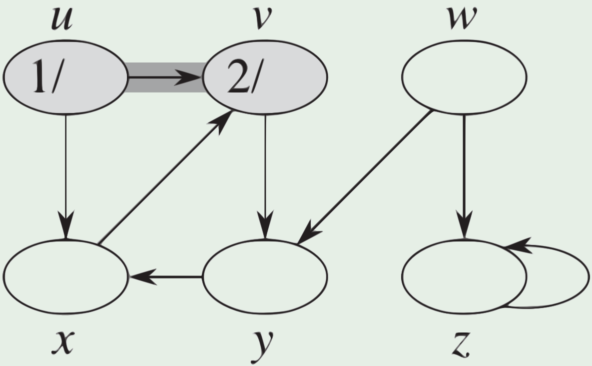
Example

What do we do?



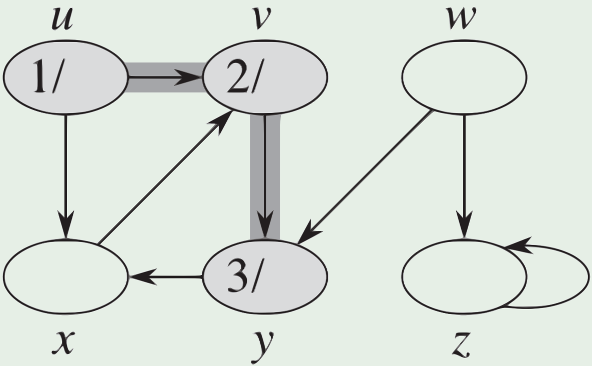
Example

What do we do?



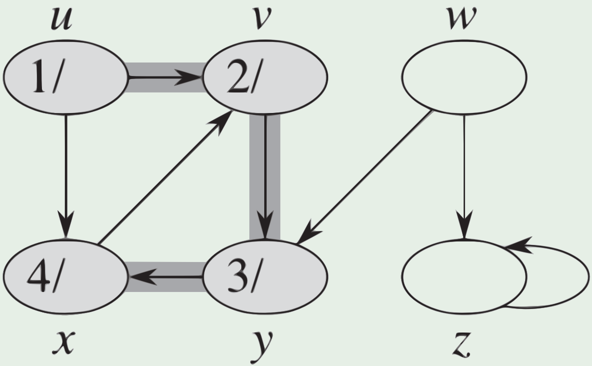
Example

What do we do?



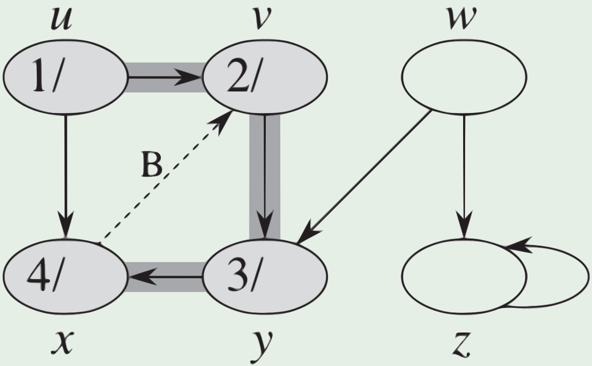
Example

What do we do?



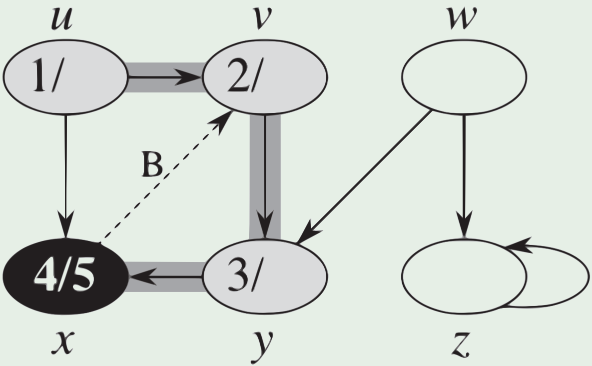
Example

What do we do?



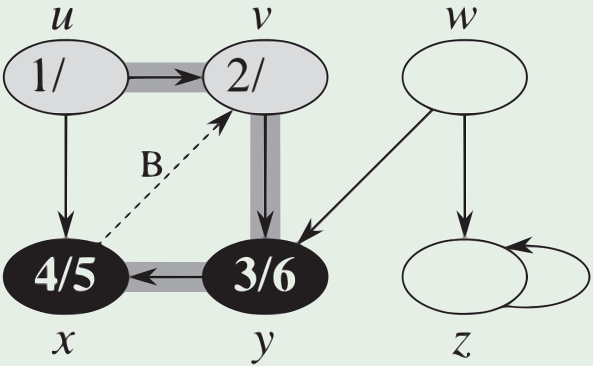
Example

What do we do?



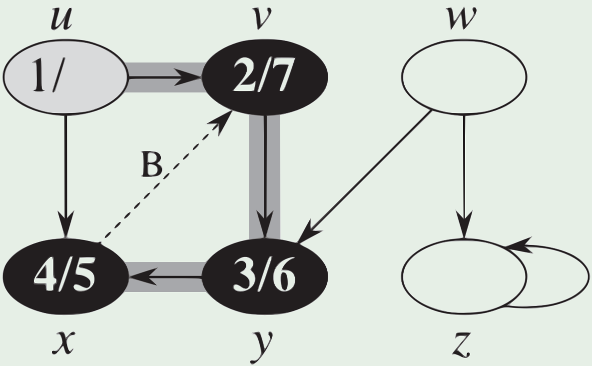
Example

What do we do?



Example

What do we do?



Complexity

Analysis

1. The loops on lines 1–3 and lines 5–7 of DFS take $\Theta\left(V\right)$.
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 $\left|Adj\left(v\right)\right|$ times.
4. But $\sum_{v \in V} \left|Adj\left(v\right)\right| = \Theta\left(E\right)$ we have that the cost of executing g lines 4–7 of DFS-VISIT is $\Theta\left(E\right)$.

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 $\Theta\left(V\right)$.
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 $\left|Adj\left(v\right)\right|$ times.
4. But $\sum_{v \in V} \left|Adj\left(v\right)\right| = \Theta\left(E\right)$ we have that the cost of executing g lines 4–7 of DFS-VISIT is $\Theta\left(E\right)$.

Then

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

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 $\Theta(V + E)$

Cinvestav

Applications

We have several

- Topological Sort
- Strongly Connected Components
- Computer Vision Algorithms
- Artificial Intelligence Algorithms
- Importance in Social Network
- Rank Algorithms for Google
- Etc.

Cinvestav

Applications

We have several

- Topological Sort
- Strongly Connected Components
- Computer Vision Algorithms
- Artificial Intelligence Algorithms
- Importance in Social Network
- Rank Algorithms for Google
- Etc.

Applications

We have several

- Topological Sort
- Strongly Connected Components
- Computer Vision Algorithms
- Artificial Intelligence Algorithms
- Importance in Social Network
- Rank Algorithms for Google
- Etc.

Applications

We have several

- Topological Sort
- Strongly Connected Components
- Computer Vision Algorithms
- Artificial Intelligence Algorithms
- Importance in Social Network
- Rank Algorithms for Google
- Etc.

Cinvestav

Applications

We have several

- Topological Sort
- Strongly Connected Components
- Computer Vision Algorithms
- Artificial Intelligence Algorithms
- Importance in Social Network
- Rank Algorithms for Google
- Etc.

Applications

We have several

- Topological Sort
- Strongly Connected Components
- Computer Vision Algorithms
- Artificial Intelligence Algorithms
- Importance in Social Network
- Rank Algorithms for Google
- Etc.

Applications

We have several

- Topological Sort
- Strongly Connected Components
- Computer Vision Algorithms
- Artificial Intelligence Algorithms
- Importance in Social Network
- Rank Algorithms for Google
- Etc.

Outline

Cinvestav

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).

Cinvestav

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

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

Cinvestav

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.

Cinvestav

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

Actually

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

Cinvestav

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

Actually

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

Actually

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

Cinvestav

Outline

Cinvestav

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.

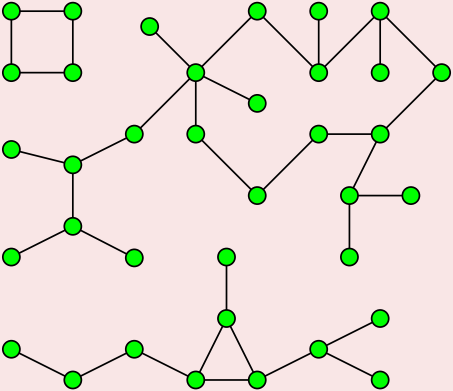
Example

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

Cinvestav

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

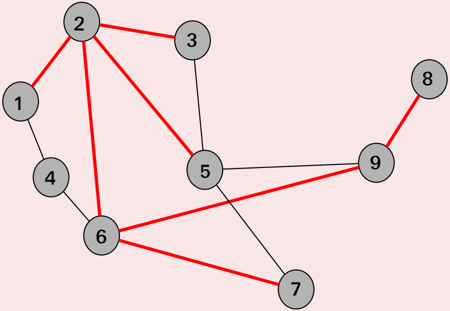
Example

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).

Cinvestav

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

Cinvestav

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

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).

Cinvestav

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

Pseudo Code

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

Pseudo Code

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

Pseudo Code

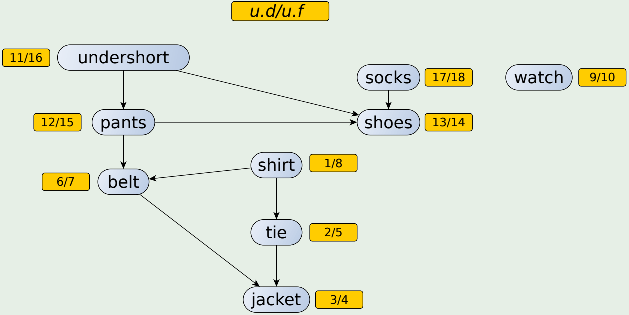
TOPOLOGICAL-SORT

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

Cinvestav

Example

Dressing



Thus

Using the $u.f$

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

Example

After Sorting