Review of Priority Queues and Graph Searches

T. M. Murali

August 27, 29 2018

Motivation: Sort a List of Numbers

Sort

INSTANCE: Nonempty list $x_1, x_2, \ldots, x_n$ of integers.

SOLUTION: A permutation $y_1, y_2, \ldots, y_n$ of $x_1, x_2, \ldots, x_n$ such that $y_i \leq y_{i+1}$, for all $1 \leq i < n$.

Motivation: Sort a List of Numbers

Sort

INSTANCE: Nonempty list $x_1, x_2, \ldots, x_n$ of integers.

SOLUTION: A permutation $y_1, y_2, \ldots, y_n$ of $x_1, x_2, \ldots, x_n$ such that $y_i \leq y_{i+1}$, for all $1 \leq i < n$.

- Possible algorithm:
  - Store all the numbers in a data structure D.
  - Repeatedly find the smallest number in D, output it, and remove it.

Motivation: Sort a List of Numbers

Sort

INSTANCE: Nonempty list $x_1, x_2, \ldots, x_n$ of integers.

SOLUTION: A permutation $y_1, y_2, \ldots, y_n$ of $x_1, x_2, \ldots, x_n$ such that $y_i \leq y_{i+1}$, for all $1 \leq i < n$.

- Possible algorithm:
  - Store all the numbers in a data structure D.
  - Repeatedly find the smallest number in D, output it, and remove it.
- To get $O(n \log n)$ running time, each "find minimum" step and each "remove" step must take $O(\log n)$ time.

Candidate Data Structures for Sorting

- Possible algorithm:
  - Store all the numbers in a data structure D.
  - Repeatedly find the smallest number in D, output it, and remove it.
- Data structure must support three operations:

Candidate Data Structures for Sorting

- Possible algorithm:
  - Store all the numbers in a data structure D.
  - Repeatedly find the smallest number in D, output it, and remove it.
- Data structure must support three operations: insertion of a number, finding minimum, and deleting minimum in

Candidate Data Structures for Sorting

- Possible algorithm:
  - Store all the numbers in a data structure D.
  - Repeatedly find the smallest number in D, output it, and remove it.
- Data structure must support three operations: insertion of a number, finding minimum, and deleting minimum in $O(\log n)$ time.

Priority Queue

- Store a set S of elements, where each element v has a priority value $\texttt{key}(v)$.
- Smaller key values $\equiv$ higher priorities.
- Operations supported:
  - find the element with smallest key
  - remove the smallest element
  - insert an element
  - delete an element
  - update the key of an element
- Element deletion and key update require knowledge of the position of the element in the priority queue.

Heaps

- Combine benefits of both lists and sorted arrays.
- Conceptually, a heap is a balanced binary tree.
- *Heap order*: For every element v at a node i, the element w at i's parent satisfies $\texttt{key}(w) \leq \texttt{key}(v)$.

Heaps

- Combine benefits of both lists and sorted arrays.
- Conceptually, a heap is a balanced binary tree.
- *Heap order*: For every element v at a node i, the element w at i's parent satisfies $\texttt{key}(w) \leq \texttt{key}(v)$.
- We can implement a heap in a pointer-based data structure.

Heaps

- Combine benefits of both lists and sorted arrays.
- Conceptually, a heap is a balanced binary tree.
- *Heap order*: For every element v at a node i, the element w at i's parent satisfies $\texttt{key}(w) \leq \texttt{key}(v)$.
- We can implement a heap in a pointer-based data structure.
- Alternatively, assume maximum number N of elements is known in advance.
- Store nodes of the heap in an array.
  - Node at index i has children at indices $2i$ and $2i+1$ and parent at index $\lfloor i/2 \rfloor$.
  - Index 1 is the root.
  - How do you know that a node at index i is a leaf?

Heaps

- Combine benefits of both lists and sorted arrays.
- Conceptually, a heap is a balanced binary tree.
- *Heap order*: For every element v at a node i, the element w at i's parent satisfies $\texttt{key}(w) \leq \texttt{key}(v)$.
- We can implement a heap in a pointer-based data structure.
- Alternatively, assume maximum number N of elements is known in advance.
- Store nodes of the heap in an array.
  - Node at index i has children at indices $2i$ and $2i + 1$ and parent at index $\lfloor i/2 \rfloor$.
  - Index 1 is the root.
  - How do you know that a node at index i is a leaf? If $2i > n$, where n is the current number of elements in the heap.

Example of a Heap

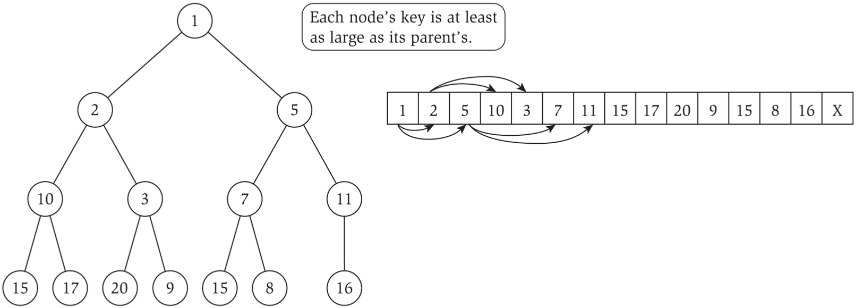


Figure 2.3 Values in a heap shown as a binary tree on the left, and represented as an array on the right. The arrows show the children for the top three nodes in the tree.

Inserting an Element: Heapify-up

1. Insert new element at index $n + 1$.
2. Fix heap order using Heapify-up$(H, n + 1)$.

```
Heapify-up(H,i):
  If i > 1 then
    let j = parent(i) = ⌊i/2⌋
    If key[H[i]]<key[H[j]] then
      swap the array entries H[i] and H[j]
      Heapify-up(H,j)
    Endif
  Endif
```

Inserting an Element: Heapify-up

1. Insert new element at index $n+1$.
2. Fix heap order using Heapify-up$(H, n+1)$.

```
Heapify-up(H,i):
  If i > 1 then
    let j = parent(i) = ⌊i/2⌋
    If key[H[i]]<key[H[j]] then
      swap the array entries H[i] and H[j]
      Heapify-up(H,j)
    Endif
  Endif
```

- Proof of correctness: read pages 61–62 of your textbook.

Example of `Heapify-up`

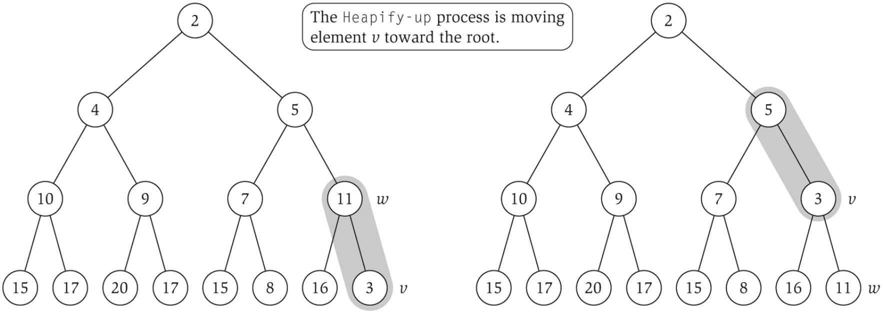


Figure 2.4 The `Heapify-up` process. Key 3 (at position 16) is too small (on the left). After swapping keys 3 and 11, the heap violation moves one step closer to the root of the tree (on the right).

Running time of `Heapify-up`

```
Heapify-up(H,i):
  If i > 1 then
    let j = parent(i) = ⌊i/2⌋
    If key[H[i]]<key[H[j]] then
      swap the array entries H[i] and H[j]
      Heapify-up(H,j)
    Endif
  Endif
```

- Running time of `Heapify-up`(i)

Running time of `Heapify-up`

```
Heapify-up(H,i):
  If i > 1 then
    let j = parent(i) = ⌊i/2⌋
    If key[H[i]]<key[H[j]] then
      swap the array entries H[i] and H[j]
      Heapify-up(H,j)
    Endif
  Endif
```

- Running time of `Heapify-up`(i) is $O(\log i)$.
  - Each invocation decreases the second argument by a factor of at least 2.
  - After k invocations, argument is at most $i/2^k$.
  - Therefore $i/2^k \geq 1$, which implies that $k \leq \log_2 i$.

Deleting an Element: `Heapify-down`

- Suppose H has $n+1$ elements.

1. Delete element at $H[i]$ by moving element at $H[n+1]$ to $H[i]$.
2. If element at $H[i]$ is too small, fix heap order using `Heapify-up`(H, i).
3. If element at $H[i]$ is too large, fix heap order using `Heapify-down`(H, i).

```
Heapify-down(H,i):
  Let n = length(H)
  If 2i > n then
    Terminate with H unchanged
  Else if 2i < n then
    Let left = 2i, and right = 2i + 1
    Let j be the index that minimizes key[H[left]] and key[H[right]]
  Else if 2i = n then
    Let j = 2i
  Endif
  If key[H[j]] < key[H[i]] then
     swap the array entries H[i] and H[j]
     Heapify-down(H,j)
  Endif
```

Deleting an Element: `Heapify-down`

- Suppose H has $n + 1$ elements.

1. Delete element at $H[i]$ by moving element at $H[n + 1]$ to $H[i]$.
2. If element at $H[i]$ is too small, fix heap order using `Heapify-up`(H, i).
3. If element at $H[i]$ is too large, fix heap order using `Heapify-down`(H, i).

```
Heapify-down(H,i):
  Let n = length(H)
  If 2i > n then
    Terminate with H unchanged
  Else if 2i < n then
    Let left = 2i, and right = 2i + 1
    Let j be the index that minimizes key[H[left]] and key[H[right]]
  Else if 2i = n then
    Let j = 2i
  Endif
  If key[H[j]] < key[H[i]] then
     swap the array entries H[i] and H[j]
     Heapify-down(H, j)
  Endif
```

Example of Heapify-down

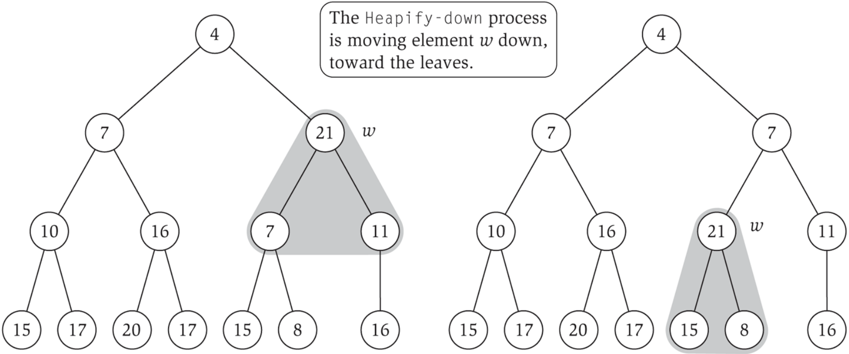


Figure 2.5 The Heapify-down process:. Key 21 (at position 3) is too big (on the left). After swapping keys 21 and 7, the heap violation moves one step closer to the bottom of the tree (on the right).

Running time of Heapify-down

```
Heapify-down(H,i):
  Let n = length(H)
  If 2i > n then
    Terminate with H unchanged
  Else if 2i < n then
    Let left = 2i, and right = 2i + 1
    Let j be the index that minimizes key[H[left]] and key[H[right]]
  Else if 2i = n then
    Let j = 2i
  Endif
  If key[H[j]] < key[H[i]] then
     swap the array entries H[i] and H[j]
     Heapify-down(H, j)
  Endif
```

- Every invocation of `Heapify-down` increases its second argument by a factor of at least two.

Running time of `Heapify-down`

```
Heapify-down(H,i):
  Let n = length(H)
  If 2i > n then
    Terminate with H unchanged
  Else if 2i < n then
    Let left = 2i, and right = 2i + 1
    Let j be the index that minimizes key[H[left]] and key[H[right]]
  Else if 2i = n then
    Let j = 2i
  Endif
  If key[H[j]] < key[H[i]] then
     swap the array entries H[i] and H[j]
     Heapify-down(H, j)
  Endif
```

- Every invocation of `Heapify-down` increases its second argument by a factor of at least two.
- After k invocations argument must be at least

Running time of `Heapify-down`

```
Heapify-down(H,i):
  Let n = length(H)
  If 2i > n then
    Terminate with H unchanged
  Else if 2i < n then
    Let left = 2i, and right = 2i + 1
    Let j be the index that minimizes key[H[left]] and key[H[right]]
  Else if 2i = n then
    Let j = 2i
  Endif
  If key[H[j]] < key[H[i]] then
     swap the array entries H[i] and H[j]
     Heapify-down(H, j)
  Endif
```

- Every invocation of `Heapify-down` increases its second argument by a factor of at least two.
- After k invocations argument must be at least $i2^k \leq n$, which implies that $k \leq \log_2 n/i$. Therefore running time is $O(\log_2 n/i)$.

Sorting Numbers with the Priority Queue

Sort

INSTANCE: Nonempty list $x_1, x_2, \ldots, x_n$ of integers.

SOLUTION: A permutation $y_1, y_2, \ldots, y_n$ of $x_1, x_2, \ldots, x_n$ such that $y_i \leq y_{i+1}$, for all $1 \leq i < n$.

Sorting Numbers with the Priority Queue

Sort

INSTANCE: Nonempty list $x_1, x_2, \ldots, x_n$ of integers.

SOLUTION: A permutation $y_1, y_2, \ldots, y_n$ of $x_1, x_2, \ldots, x_n$ such that $y_i \leq y_{i+1}$, for all $1 \leq i < n$.

- Final algorithm:
  - Insert each number in a priority queue H.
  - Repeatedly find the smallest number in H, output it, and delete it from H.

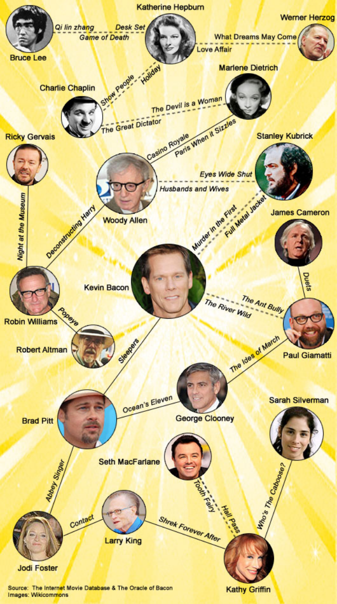
Sorting Numbers with the Priority Queue

Sort

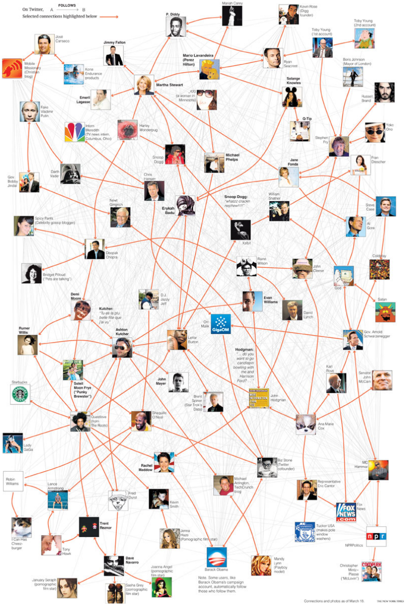
INSTANCE: Nonempty list $x_1, x_2, \ldots, x_n$ of integers.

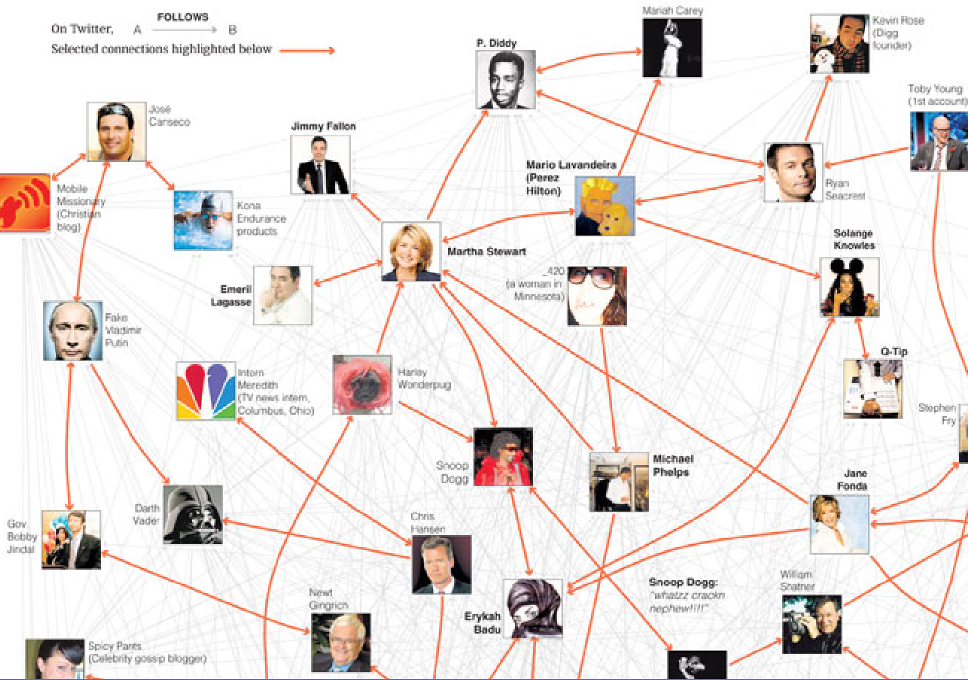
SOLUTION: A permutation $y_1, y_2, \ldots, y_n$ of $x_1, x_2, \ldots, x_n$ such that $y_i \leq y_{i+1}$, for all $1 \leq i < n$.

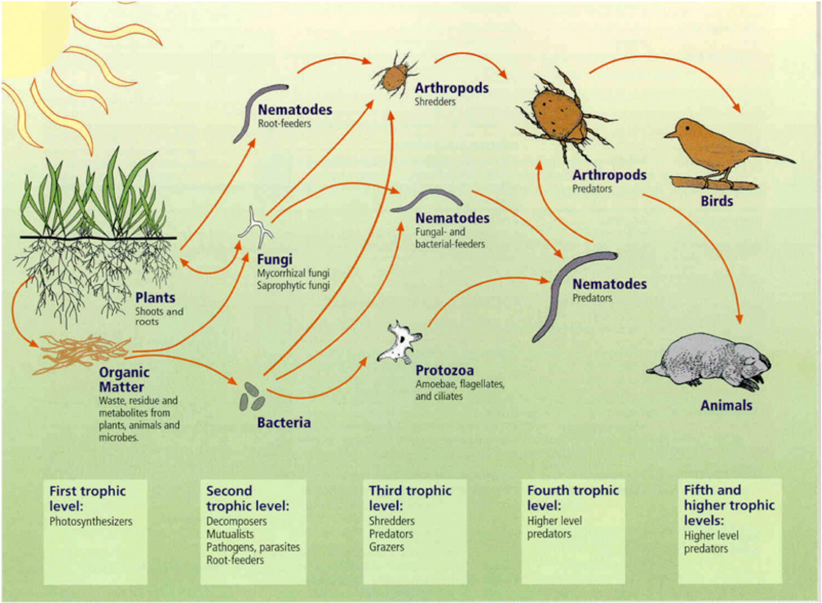
- Final algorithm:
  - Insert each number in a priority queue H.
  - Repeatedly find the smallest number in H, output it, and delete it from H.
- Each insertion and deletion takes $O(\log n)$ time for a total running time of $O(n \log n)$.

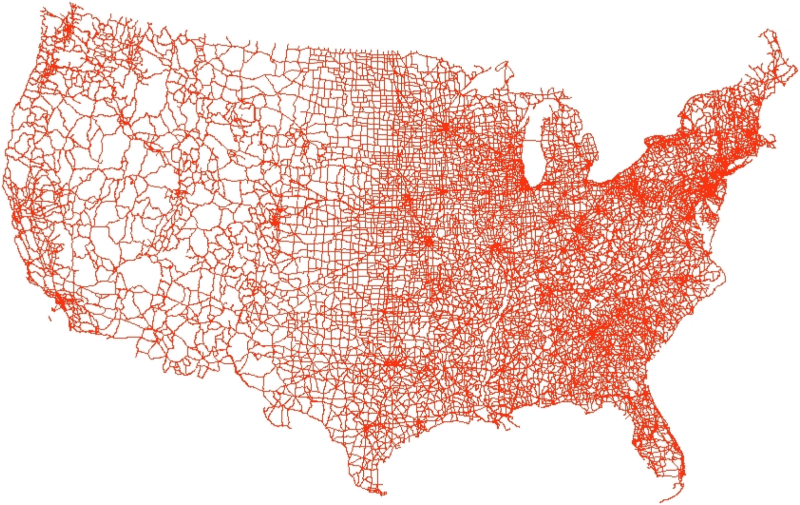


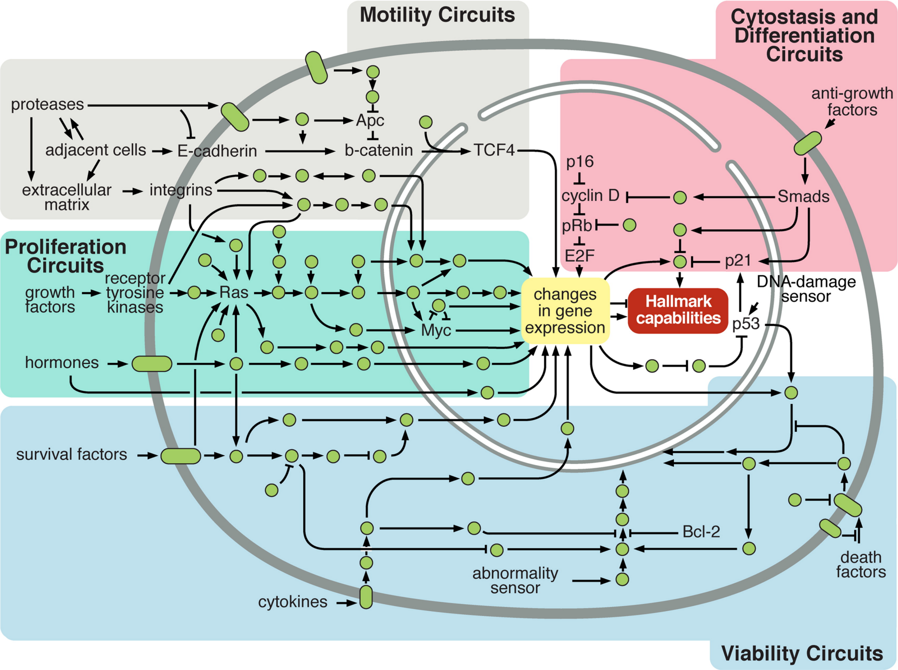
The Oracle of Bacon


On Twitter, A FOLLOWS B
Selected connections highlighted below
P. Diddy
Mariah Carey
Kevin Rose (Digg founder)
Toby Young (2nd account)
Toby Young (1st account)
José Canseco
Jimmy Fallon
Mario Lavandeira (Perez Hilton)
Ryan Seacrest
Boris Johnson (Mayor of London)
Mobile Missionary (Christian blog)
Kona Endurance products
Martha Stewart
Solange Knowles
Emeril Lagasse
Russell Brand
Fake Vladimir Putin
Intern Meredith (TV news intern, Columbus, Ohio)
Harley Wonderpug
Q-Tip
Yoko Ono
Stephen Fry
Snoop Dogg
Michael Phelps
Jane Fonda
Fran Drescher
Gov. Bobby Jindal
Darth Vader
Chris Hansen
Newt Gingrich
Erykah Badu
Snoop Dogg: "whatzz crackn nephew!!!!"
William Shatner
Steve Case
Al Gore
Spicy Pants (Celebrity gossip blogger)
Xzibit
Deepak Chopra
Rainn Wilson
John Cleese
Coldplay
Bridget Pilloud ("Pets are talking")
God
Satan
Demi Moore
D.J. Jazzy Jeff
Evan Williams
David Lynch
Om Malik
GigaOM
Ashton Kutcher
LeVar Burton
Gov. Arnold Schwarzenegger
Rumer Willis
Hodgman
Karl Rove
Senator John McCain
Starbucks
John Mayer
Brent Spiner (Star Trek's Data)
John Hodgman
Questlove (from The Roots)
Shaquille O'Neal
Ana Marie Cox
Lady GaGa
Rachel Maddow
Biz Stone (Twitter cofounder)
MC Hammer
Robin Williams
Lance Armstrong
Michael Arrington, TechCrunch blog
Fred Durst
Kevin Smith
Representative Eric Cantor
Fox News
Trent Reznor
Tucker USA (makes pole window washers)
NPRPolitics
I Can Has Cheezburger
Tony Hawk
Jenna Haze (Pornographic film star)
Dave Navarro
Barack Obama
Mandy Lynn (Playboy model)
Christopher Mintz-Plasse ("McLovin")
January Seraph (pornographic film star)
Sasha Grey (pornographic film star)
Joanna Angel (pornographic film star)
Note. Some users, like Barack Obama's campaign account, automatically follow those who follow them.
Connections and photos as of March 15.
THE NEW YORK TIMES


On Twitter, A FOLLOWS B
Selected connections highlighted below
P. Diddy
Mariah Carey
Kevin Rose (Digg founder)
Toby Young (1st account)
José Canseco
Jimmy Fallon
Mario Lavandeira (Perez Hilton)
Ryan Seacrest
Mobile Missionary (Christian blog)
Kona Endurance products
Martha Stewart
Solange Knowles
Emeril Lagasse
_420 (a woman in Minnesota)
Fake Vladimir Putin
Q-Tip
Intern Meredith (TV news intern, Columbus, Ohio)
Harley Wonderpug
Stephen Fry
Snoop Dogg
Michael Phelps
Jane Fonda
Darth Vader
Gov. Bobby Jindal
Chris Hansen
William Shatner
Snoop Dogg: "whatzz crackn nephew!!!!"
Newt Gingrich
Erykah Badu
Spicy Pants (Celebrity gossip blogger)


Arthropods
Shredders
Nematodes
Root-feeders
Arthropods
Predators
Birds
Nematodes
Fungal- and
bacterial-feeders
Fungi
Mycorrhizal fungi
Saprophytic fungi
Plants
Shoots and
roots
Nematodes
Predators
Protozoa
Amoebae, flagellates,
and ciliates
Organic
Matter
Waste, residue and
metabolites from
plants, animals and
microbes.
Bacteria
Animals
First trophic
level:
Photosynthesizers
Second
trophic level:
Decomposers
Mutualists
Pathogens, parasites
Root-feeders
Third trophic
level:
Shredders
Predators
Grazers
Fourth trophic
level:
Higher level
predators
Fifth and
higher trophic
levels:
Higher level
predators





Motility Circuits
Cytostasis and Differentiation Circuits
Proliferation Circuits
Viability Circuits
proteases
adjacent cells
E-cadherin
extracellular matrix
integrins
Apc
b-catenin
TCF4
anti-growth factors
p16
cyclin D
pRb
E2F
Smads
p21
DNA-damage sensor
p53
growth factors
receptor tyrosine kinases
Ras
Myc
changes in gene expression
Hallmark capabilities
hormones
survival factors
Bcl-2
abnormality sensor
cytokines
death factors

Graphs

- Model pairwise relationships (edges) between objects (nodes).

Graphs

- Model pairwise relationships (edges) between objects (nodes).
- Useful in a large number of applications:

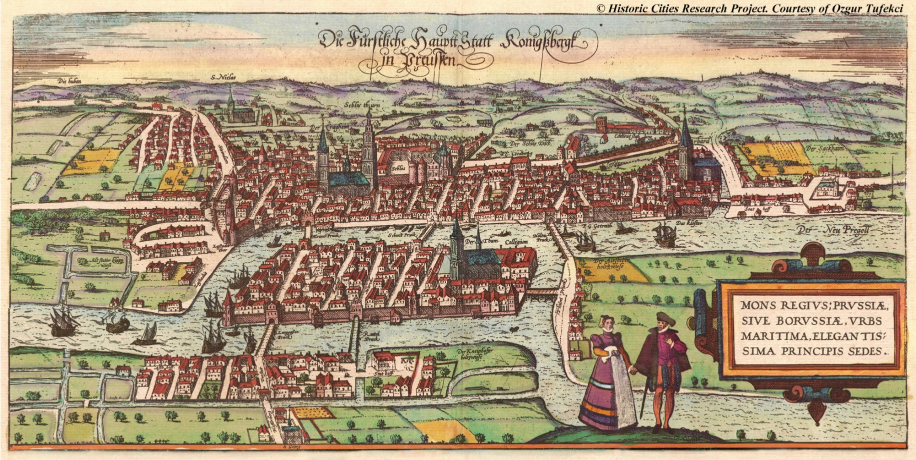
Graphs

- Model pairwise relationships (edges) between objects (nodes).
- Useful in a large number of applications: computer networks, the World Wide Web, ecology (food webs), social networks, software systems, job scheduling, VLSI circuits, cellular networks, ...
- Other examples: gene and protein networks, our bodies (nervous and circulatory systems, brains), buildings, transportation networks, ...

Graphs

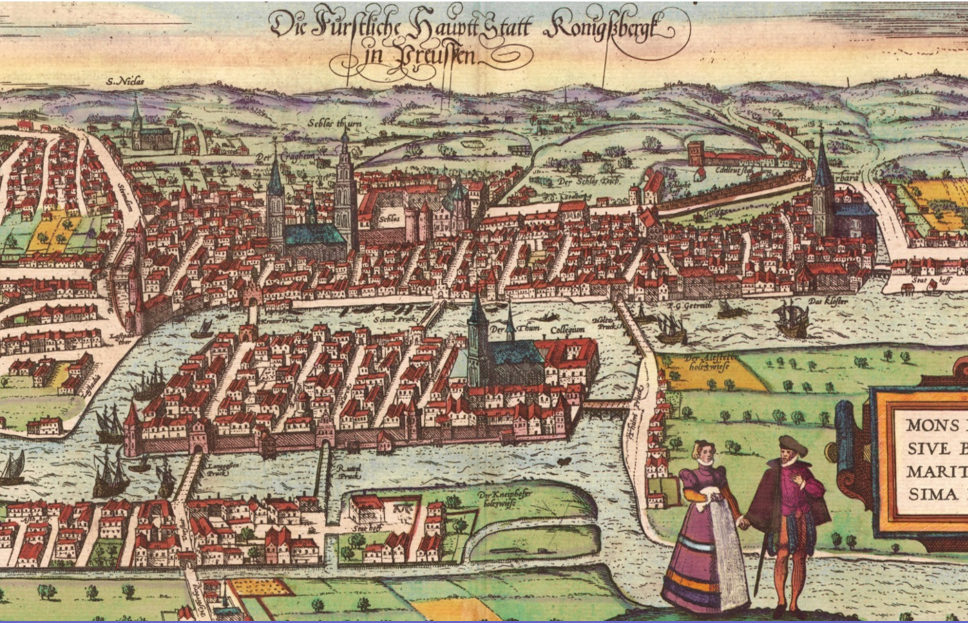
- Model pairwise relationships (edges) between objects (nodes).
- Useful in a large number of applications: computer networks, the World Wide Web, ecology (food webs), social networks, software systems, job scheduling, VLSI circuits, cellular networks, ...
- Other examples: gene and protein networks, our bodies (nervous and circulatory systems, brains), buildings, transportation networks, ...

Euler and Graphs

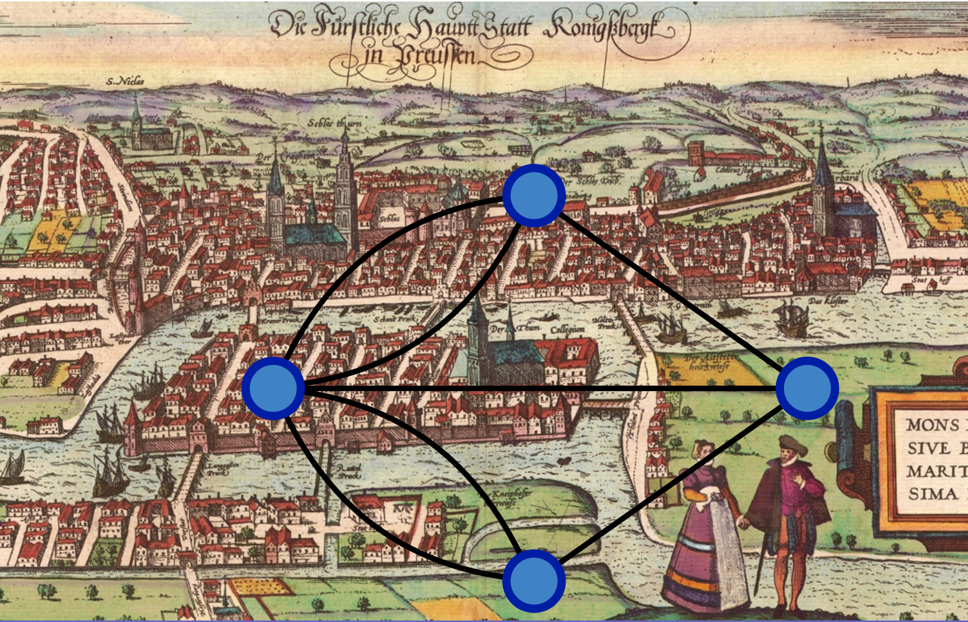


Devise a walk through the city that
crosses each of the seven bridges exactly once.

Euler and Graphs

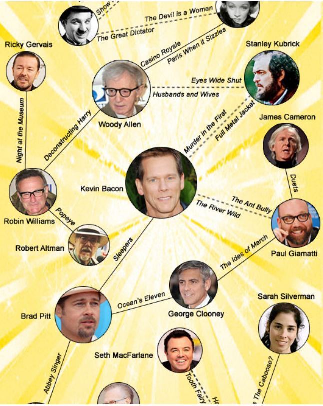


Euler and Graphs



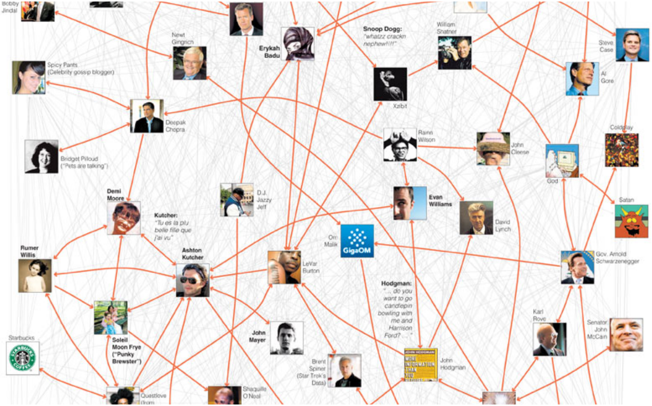
Definition of a Graph

- *Undirected graph* $G = (V, E)$: set V of nodes and set E of edges, where $E \subseteq V \times V$.
  - Elements of E are unordered pairs.
  - Edge (u, v) is *incident* on u, v; u and v are *neighbours* of each other.
  - Exactly one edge between any pair of nodes.
  - G contains no self loops, i.e., no edges of the form (u, u).

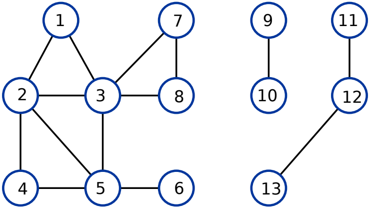


Definition of a Graph

- *Directed graph* $G = (V, E)$: set V of nodes and set E of edges, where $E \subseteq V \times V$.
  - Elements of E are ordered pairs.
  - $e = (u, v)$: u is the *tail* of the edge e, v is its *head*; e is *directed from u to v*.
  - A pair of nodes may be connected by two directed edges: (u, v) and (v, u).
  - G contains no self loops.

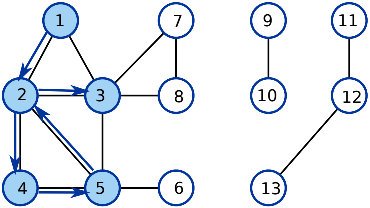


Paths and Connectivity



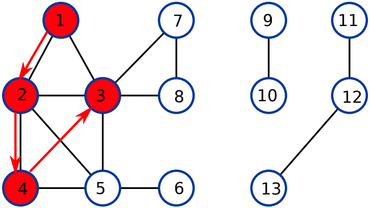
- A *v_1-v_k path* in an undirected graph $G = (V, E)$ is a sequence P of nodes $v_1, v_2, \ldots, v_{k-1}, v_k \in V$ such that every consecutive pair of nodes $v_i, v_{i+1}, 1 \leq i < k$ is connected by an edge in E.

Paths and Connectivity



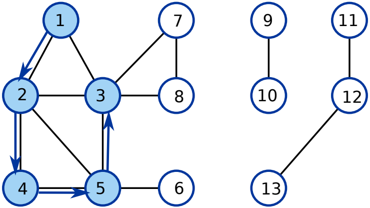
- A v_1-v_k *path* in an undirected graph $G = (V, E)$ is a sequence P of nodes $v_1, v_2, \ldots, v_{k-1}, v_k \in V$ such that every consecutive pair of nodes $v_i, v_{i+1}, 1 \le i < k$ is connected by an edge in E.

Paths and Connectivity



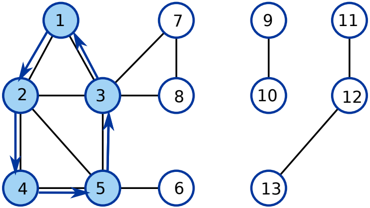
- A v_1-v_k *path* in an undirected graph $G = (V, E)$ is a sequence P of nodes $v_1, v_2, \ldots, v_{k-1}, v_k \in V$ such that every consecutive pair of nodes $v_i, v_{i+1}, 1 \leq i < k$ is connected by an edge in E.

Paths and Connectivity



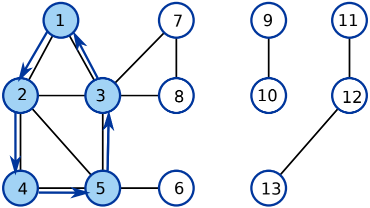
- A *v_1-v_k path* in an undirected graph $G = (V, E)$ is a sequence P of nodes $v_1, v_2, \ldots, v_{k-1}, v_k \in V$ such that every consecutive pair of nodes $v_i, v_{i+1}, 1 \leq i < k$ is connected by an edge in E.
- A path is *simple* if all its nodes are distinct.

Paths and Connectivity



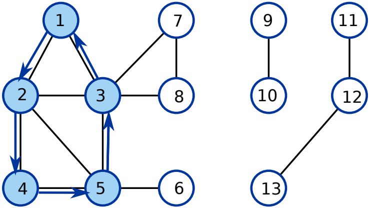
- A *v_1-v_k path* in an undirected graph $G = (V, E)$ is a sequence P of nodes $v_1, v_2, \ldots, v_{k-1}, v_k \in V$ such that every consecutive pair of nodes $v_i, v_{i+1}, 1 \leq i < k$ is connected by an edge in E.
- A path is *simple* if all its nodes are distinct.
- A *cycle* is a path where $k > 2$, the first $k - 1$ nodes are distinct, and $v_1 = v_k$.

Paths and Connectivity



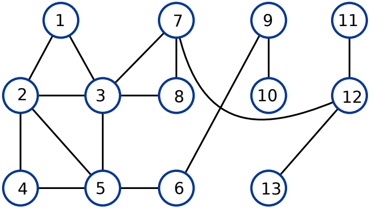
- A *v_1-v_k path* in an undirected graph $G = (V, E)$ is a sequence P of nodes $v_1, v_2, \ldots, v_{k-1}, v_k \in V$ such that every consecutive pair of nodes $v_i, v_{i+1}, 1 \leq i < k$ is connected by an edge in E.
- A path is *simple* if all its nodes are distinct.
- A *cycle* is a path where $k > 2$, the first $k - 1$ nodes are distinct, and $v_1 = v_k$.
- Similar definitions carry over to directed graphs as well.

Paths and Connectivity



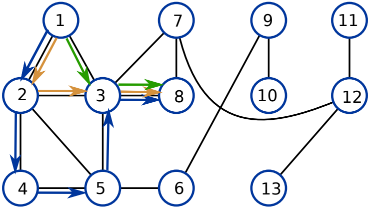
- A *v_1-v_k path* in an undirected graph $G = (V, E)$ is a sequence P of nodes $v_1, v_2, \ldots, v_{k-1}, v_k \in V$ such that every consecutive pair of nodes $v_i, v_{i+1}, 1 \leq i < k$ is connected by an edge in E.
- A path is *simple* if all its nodes are distinct.
- A *cycle* is a path where $k > 2$, the first $k - 1$ nodes are distinct, and $v_1 = v_k$.
- Similar definitions carry over to directed graphs as well.
- An undirected graph G is *connected* if for every pair of nodes $u, v \in V$, there is a path from u to v in G.

Paths and Connectivity



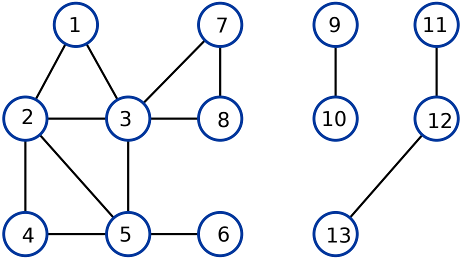
- A *v_1-v_k path* in an undirected graph $G = (V, E)$ is a sequence P of nodes $v_1, v_2, \ldots, v_{k-1}, v_k \in V$ such that every consecutive pair of nodes $v_i, v_{i+1}, 1 \leq i < k$ is connected by an edge in E.
- A path is *simple* if all its nodes are distinct.
- A *cycle* is a path where $k > 2$, the first $k-1$ nodes are distinct, and $v_1 = v_k$.
- Similar definitions carry over to directed graphs as well.
- An undirected graph G is *connected* if for every pair of nodes $u, v \in V$, there is a path from u to v in G.

Paths and Connectivity



- A v_1-v_k *path* in an undirected graph $G = (V, E)$ is a sequence P of nodes $v_1, v_2, \ldots, v_{k-1}, v_k \in V$ such that every consecutive pair of nodes $v_i, v_{i+1}, 1 \leq i < k$ is connected by an edge in E.
- A path is *simple* if all its nodes are distinct.
- A *cycle* is a path where $k > 2$, the first $k - 1$ nodes are distinct, and $v_1 = v_k$.
- Similar definitions carry over to directed graphs as well.
- An undirected graph G is *connected* if for every pair of nodes $u, v \in V$, there is a path from u to v in G.
- *Distance* $d(u, v)$ between two nodes u and v is the minimum number of edges in any u-v path.

s-t Connectivity

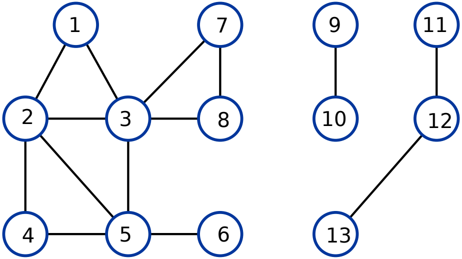


s-t Connectivity

INSTANCE: An undirected graph $G = (V, E)$ and two nodes $s, t \in V$.

QUESTION: Is there an s-t path in G?

s-t Connectivity



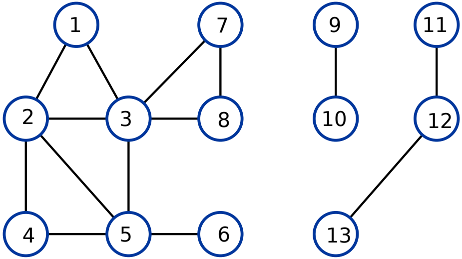
s-t Connectivity

INSTANCE: An undirected graph $G = (V, E)$ and two nodes $s, t \in V$.

QUESTION: Is there an s-t path in G?

- The *connected component of G containing s* is the set of all nodes u such that there is an s-u path in G.

s-t Connectivity



s-t Connectivity

INSTANCE: An undirected graph $G = (V, E)$ and two nodes $s, t \in V$.

QUESTION: Is there an s-t path in G?

- The *connected component of G containing s* is the set of all nodes u such that there is an s-u path in G.
- Algorithm for the s-t Connectivity problem: compute the connected component of G that contains s and check if t is in that component.

Computing Connected Components

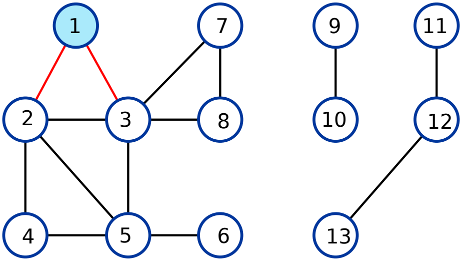
- "Explore" G starting from s and maintain set R of visited nodes.

R will consist of nodes to which s has a path
Initially $R = \{s\}$
While there is an edge (u, v) where $u \in R$ and $v \notin R$
  Add v to R
Endwhile

Computing Connected Components

- "Explore" G starting from s and maintain set R of visited nodes.

```
R will consist of nodes to which s has a path
Initially R = {s}
While there is an edge (u, v) where u ∈ R and v ∉ R
  Add v to R
Endwhile
```



Computing Connected Components

- "Explore" G starting from s and maintain set R of visited nodes.

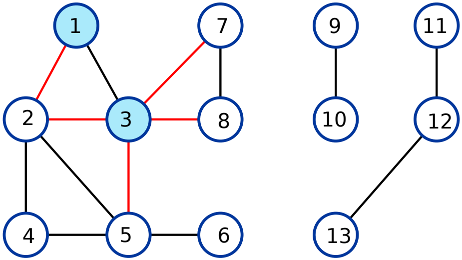
R will consist of nodes to which s has a path

Initially $R = \{s\}$

While there is an edge (u, v) where $u \in R$ and $v \notin R$

  Add v to R

Endwhile



Computing Connected Components

- "Explore" G starting from s and maintain set R of visited nodes.

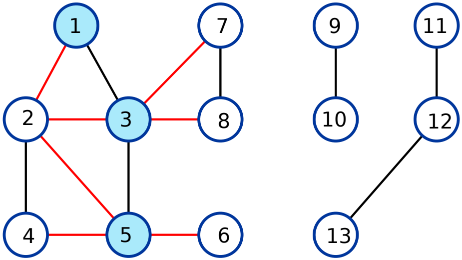
R will consist of nodes to which s has a path

Initially $R = \{s\}$

While there is an edge (u, v) where $u \in R$ and $v \notin R$

  Add v to R

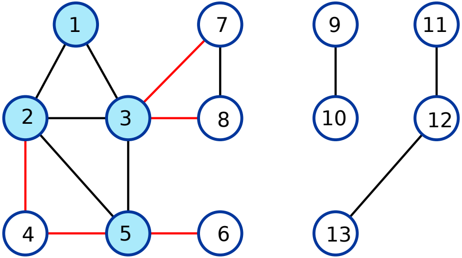
Endwhile



Computing Connected Components

- "Explore" G starting from s and maintain set R of visited nodes.

R will consist of nodes to which s has a path
Initially $R = \{s\}$
While there is an edge (u, v) where $u \in R$ and $v \notin R$
  Add v to R
Endwhile



Computing Connected Components

- "Explore" G starting from s and maintain set R of visited nodes.

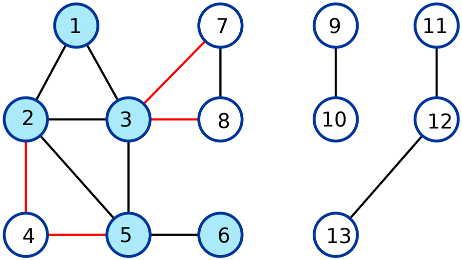
R will consist of nodes to which s has a path

Initially $R = \{s\}$

While there is an edge (u, v) where $u \in R$ and $v \notin R$

  Add v to R

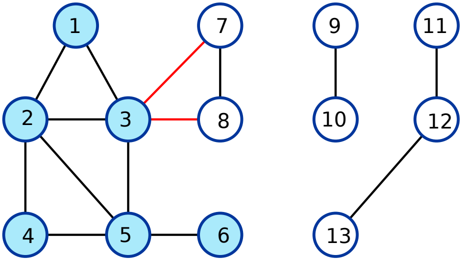
Endwhile



Computing Connected Components

- "Explore" G starting from s and maintain set R of visited nodes.

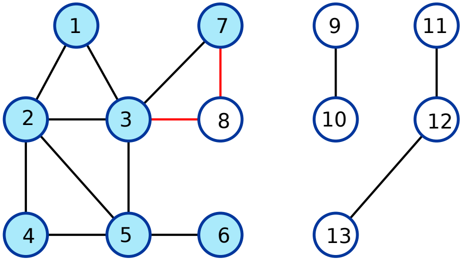
R will consist of nodes to which s has a path
Initially $R = \{s\}$
While there is an edge (u, v) where $u \in R$ and $v \notin R$
  Add v to R
Endwhile



Computing Connected Components

- "Explore" G starting from s and maintain set R of visited nodes.

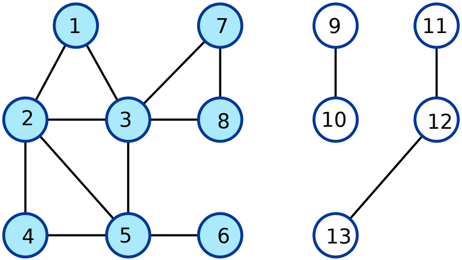
```
R will consist of nodes to which s has a path
Initially R = {s}
While there is an edge (u, v) where u ∈ R and v ∉ R
  Add v to R
Endwhile
```



Computing Connected Components

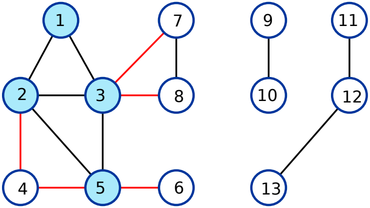
- "Explore" G starting from s and maintain set R of visited nodes.

```
R will consist of nodes to which s has a path
Initially R = {s}
While there is an edge (u, v) where u ∈ R and v ∉ R
  Add v to R
Endwhile
```



Issues in Computing Connected Components

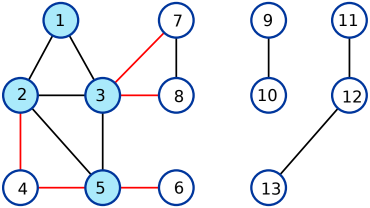
```
R will consist of nodes to which s has a path
Initially R = {s}
While there is an edge (u, v) where u ∈ R and v ∉ R
  Add v to R
Endwhile
```



- Why does the algorithm terminate?
- Does the algorithm truly compute connected component of G containing s?
- What is the running time of the algorithm?

Issues in Computing Connected Components

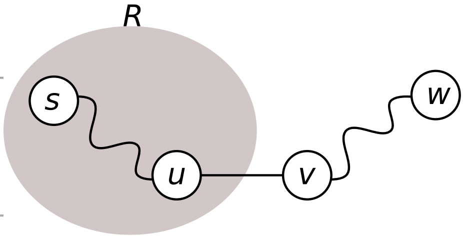
```
R will consist of nodes to which s has a path
Initially R = {s}
While there is an edge (u, v) where u ∈ R and v ∉ R
  Add v to R
Endwhile
```



- Why does the algorithm terminate? Each iteration adds a new node to R.
- Does the algorithm truly compute connected component of G containing s?
- What is the running time of the algorithm?

Correctness of the Algorithm

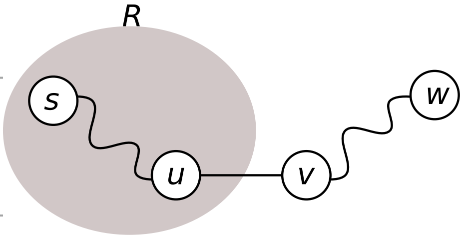
```
R will consist of nodes to which s has a path
Initially R = {s}
While there is an edge (u, v) where u ∈ R and v ∉ R
  Add v to R
Endwhile
```



- Claim: at the end of the algorithm, the set R is exactly the connected component of G containing s.

Correctness of the Algorithm

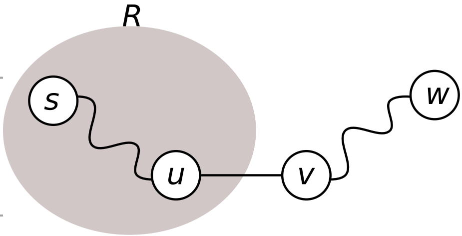
R will consist of nodes to which s has a path
Initially $R = \{s\}$
While there is an edge (u, v) where $u \in R$ and $v \notin R$
  Add v to R
Endwhile



- Claim: at the end of the algorithm, the set R is exactly the connected component of G containing s.
- Proof: At termination, suppose $w \notin R$ but there is an s-w path P in G.
  - Consider first node v in P not in R ($v \neq s$).
  - Let u be the predecessor of v in P:

Correctness of the Algorithm

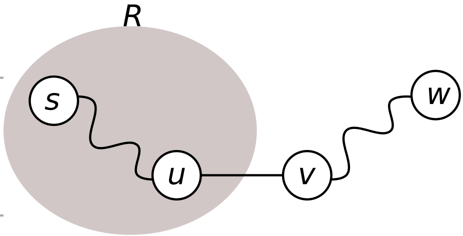
```
R will consist of nodes to which s has a path
Initially R = {s}
While there is an edge (u, v) where u ∈ R and v ∉ R
  Add v to R
Endwhile
```



- Claim: at the end of the algorithm, the set R is exactly the connected component of G containing s.
- Proof: At termination, suppose $w \notin R$ but there is an s-w path P in G.
  - Consider first node v in P not in R ($v \neq s$).
  - Let u be the predecessor of v in P: u is in R.
  - (u, v) is an edge with $u \in R$ but $v \notin R$, contradicting the stopping rule.

Correctness of the Algorithm

R will consist of nodes to which s has a path
Initially $R = \{s\}$
While there is an edge (u, v) where $u \in R$ and $v \notin R$
  Add v to R
Endwhile



- Claim: at the end of the algorithm, the set R is exactly the connected component of G containing s.
- Proof: At termination, suppose $w \notin R$ but there is an s-w path P in G.
  - Consider first node v in P not in R ($v \neq s$).
  - Let u be the predecessor of v in P: u is in R.
  - (u, v) is an edge with $u \in R$ but $v \notin R$, contradicting the stopping rule.
  - Note: wrong to assume that predecessor of w in P is not in R.

Running Time of the Algorithm

```
R will consist of nodes to which s has a path
Initially R = {s}
While there is an edge (u, v) where u ∈ R and v ∉ R
  Add v to R
Endwhile
```

Running Time of the Algorithm

R will consist of nodes to which s has a path

Initially $R = \{s\}$

While there is an edge (u, v) where $u \in R$ and $v \notin R$

  Add v to R

Endwhile

- Analyse algorithm in terms of two parameters: the number of nodes n and the number of edges m.
- Implement the `while` loop by examining each edge in E. Running time of each loop is

Running Time of the Algorithm

```
R will consist of nodes to which s has a path
Initially R = {s}
While there is an edge (u, v) where u ∈ R and v ∉ R
  Add v to R
Endwhile
```

- Analyse algorithm in terms of two parameters: the number of nodes n and the number of edges m.
- Implement the `while` loop by examining each edge in E. Running time of each loop is $O(m)$.
- How many `while` loops does the algorithm execute?

Running Time of the Algorithm

R will consist of nodes to which s has a path
Initially $R = \{s\}$
While there is an edge (u, v) where $u \in R$ and $v \notin R$
  Add v to R
Endwhile

- Analyse algorithm in terms of two parameters: the number of nodes n and the number of edges m.
- Implement the `while` loop by examining each edge in E. Running time of each loop is $O(m)$.
- How many `while` loops does the algorithm execute? At most n.
- The running time is

Running Time of the Algorithm

R will consist of nodes to which s has a path
Initially $R = \{s\}$
While there is an edge (u, v) where $u \in R$ and $v \notin R$
  Add v to R
Endwhile

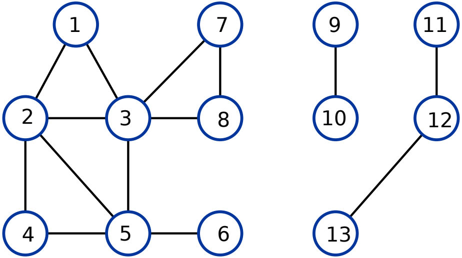
- Analyse algorithm in terms of two parameters: the number of nodes n and the number of edges m.
- Implement the `while` loop by examining each edge in E. Running time of each loop is $O(m)$.
- How many `while` loops does the algorithm execute? At most n.
- The running time is $O(mn)$.

Running Time of the Algorithm

```
R will consist of nodes to which s has a path
Initially R = {s}
While there is an edge (u, v) where u ∈ R and v ∉ R
  Add v to R
Endwhile
```

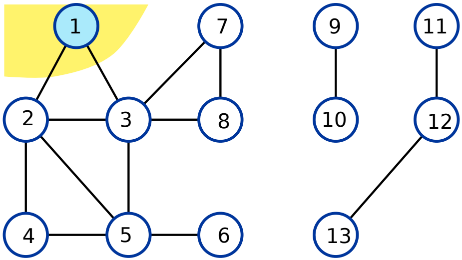
- Analyse algorithm in terms of two parameters: the number of nodes n and the number of edges m.
- Implement the `while` loop by examining each edge in E. Running time of each loop is $O(m)$.
- How many `while` loops does the algorithm execute? At most n.
- The running time is $O(mn)$.
- Can we improve the running time by processing edges more carefully?

Breadth-First Search (BFS)



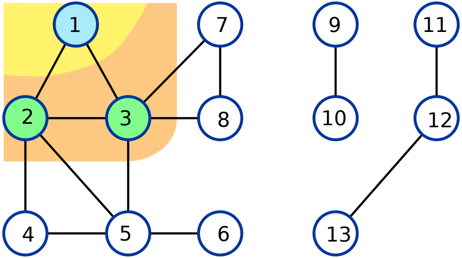
- Idea: explore G starting at s and going "outward" in all directions, adding nodes one layer at a time.

Breadth-First Search (BFS)



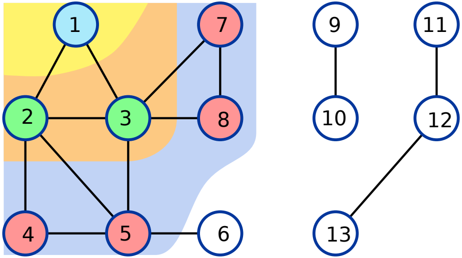
- Idea: explore G starting at s and going "outward" in all directions, adding nodes one layer at a time.
- Layer L_0 contains only s.

Breadth-First Search (BFS)



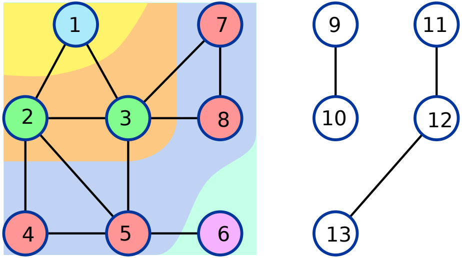
- Idea: explore G starting at s and going "outward" in all directions, adding nodes one layer at a time.
- Layer L_0 contains only s.
- Layer L_1 contains all neighbours of s.

Breadth-First Search (BFS)



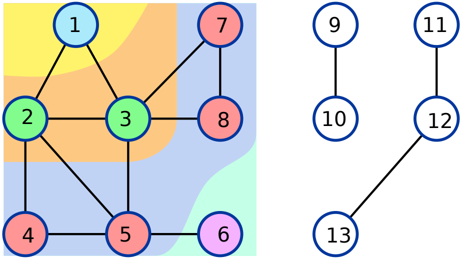
- Idea: explore G starting at s and going “outward” in all directions, adding nodes one layer at a time.
- Layer L_0 contains only s.
- Layer L_1 contains all neighbours of s.
- Given layers $L_0, L_1, \ldots, L_j$, layer L_{j+1} contains all nodes that
  1. do not belong to an earlier layer and
  2. are connected by an edge to a node in layer L_j.

Breadth-First Search (BFS)



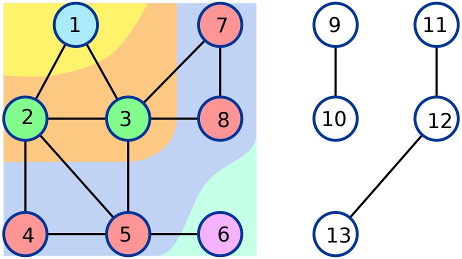
- Idea: explore G starting at s and going "outward" in all directions, adding nodes one layer at a time.
- Layer L_0 contains only s.
- Layer L_1 contains all neighbours of s.
- Given layers $L_0, L_1, \ldots, L_j$, layer L_{j+1} contains all nodes that
  1. do not belong to an earlier layer and
  2. are connected by an edge to a node in layer L_j.

Properties of BFS



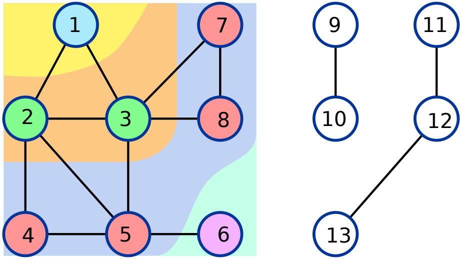
- We have not yet described how to compute these layers.
- Claim: For each $j \geq 1$, layer L_j consists of all nodes

Properties of BFS



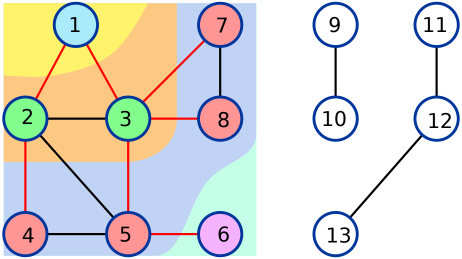
- We have not yet described how to compute these layers.
- Claim: For each $j \geq 1$, layer L_j consists of all nodes exactly at distance j from S. Proof

Properties of BFS



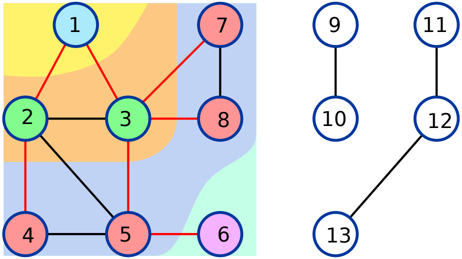
- We have not yet described how to compute these layers.
- Claim: For each $j \geq 1$, layer L_j consists of all nodes exactly at distance j from S. Proof by induction on j.
- Claim: There is a path from s to t if and only if t is a member of some layer.

Properties of BFS



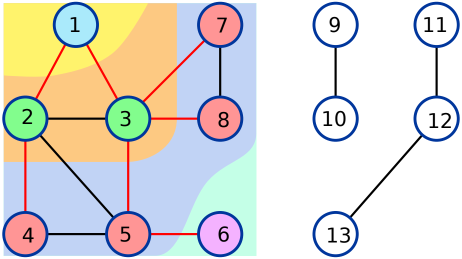
- We have not yet described how to compute these layers.
- Claim: For each $j \geq 1$, layer L_j consists of all nodes exactly at distance j from S. Proof by induction on j.
- Claim: There is a path from s to t if and only if t is a member of some layer.
- Let v be a node in layer L_{j+1} and u be the "first" node in L_j such that (u, v) is an edge in G. Consider the graph T formed by all such edges, directed from u to v.

Properties of BFS



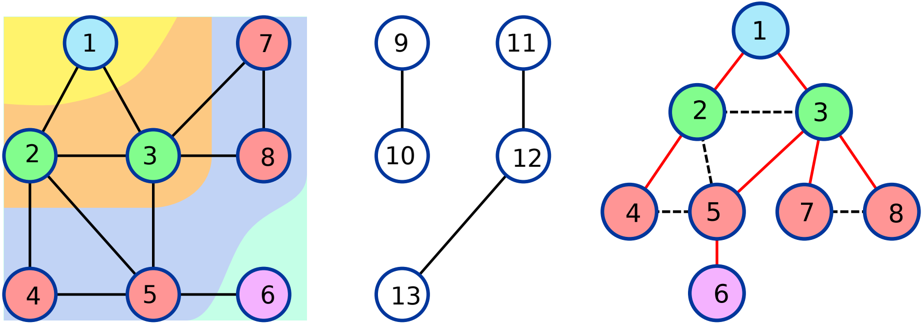
- We have not yet described how to compute these layers.
- Claim: For each $j \geq 1$, layer L_j consists of all nodes exactly at distance j from S. Proof by induction on j.
- Claim: There is a path from s to t if and only if t is a member of some layer.
- Let v be a node in layer L_{j+1} and u be the "first" node in L_j such that (u, v) is an edge in G. Consider the graph T formed by all such edges, directed from u to v.
  - Why is T a tree?

Properties of BFS



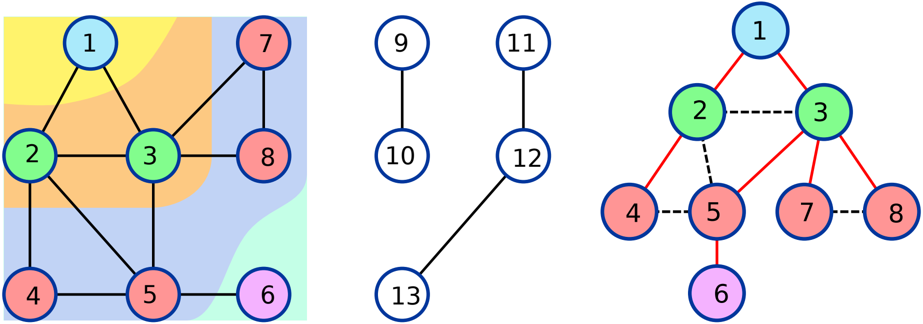
- We have not yet described how to compute these layers.
- Claim: For each $j \geq 1$, layer L_j consists of all nodes exactly at distance j from S. Proof by induction on j.
- Claim: There is a path from s to t if and only if t is a member of some layer.
- Let v be a node in layer L_{j+1} and u be the "first" node in L_j such that (u, v) is an edge in G. Consider the graph T formed by all such edges, directed from u to v.
  - Why is T a tree? It is connected. The number of edges in T is the number of nodes in all the layers minus 1.
  - T is called the *breadth-first search tree*.

BFS Trees



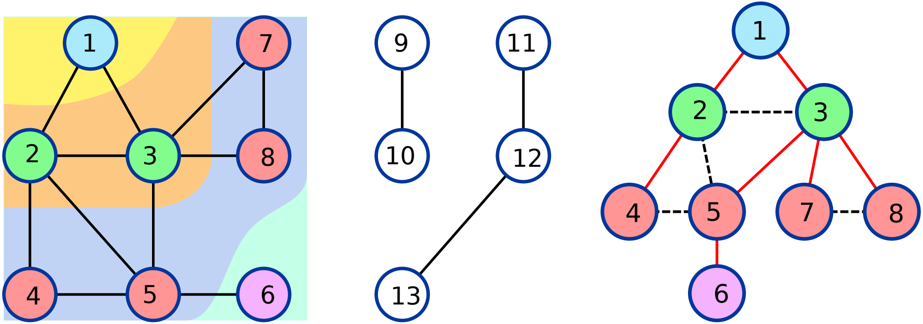
- *Non-tree edge*: an edge of G that does not belong to the BFS tree T.
- Claim: Let T be a BFS tree, let x and y be nodes in T belonging to layers L_i and L_j, respectively, and let (x, y) be an edge of G. Then $|i - j| \leq 1$.

BFS Trees



- *Non-tree edge*: an edge of G that does not belong to the BFS tree T.
- Claim: Let T be a BFS tree, let x and y be nodes in T belonging to layers L_i and L_j, respectively, and let (x, y) be an edge of G. Then $|i - j| \leq 1$.
- Proof by contradiction: Suppose $i < j - 1$. Node $x \in L_i \Rightarrow$ all nodes adjacent to x are in layers $L_1, L_2, \ldots L_{i+1}$. Hence y must be in layer L_{i+1} or earlier.

BFS Trees



- *Non-tree edge*: an edge of G that does not belong to the BFS tree T.
- Claim: Let T be a BFS tree, let x and y be nodes in T belonging to layers L_i and L_j, respectively, and let (x, y) be an edge of G. Then $|i - j| \leq 1$.
- Proof by contradiction: Suppose $i < j - 1$. Node $x \in L_i \Rightarrow$ all nodes adjacent to x are in layers $L_1, L_2, \ldots L_{i+1}$. Hence y must be in layer L_{i+1} or earlier.
- Still unresolved: an efficient implementation of BFS.

Depth-First Search (DFS)

- Explore G as if it were a maze: start from s, traverse first edge out (to node v), traverse first edge out of v, . . . , reach a dead-end, backtrack, ...

Depth-First Search (DFS)

- Explore G as if it were a maze: start from s, traverse first edge out (to node v), traverse first edge out of v, ..., reach a dead-end, backtrack, ...

1. Mark all nodes as "`Unexplored`".
2. Invoke `DFS(`s`)`.

```
DFS(u):
  Mark u as "Explored" and add u to R
  For each edge (u,v) incident to u
    If v is not marked "Explored" then
      Recursively invoke DFS(v)
    Endif
  Endfor
```

Depth-First Search (DFS)

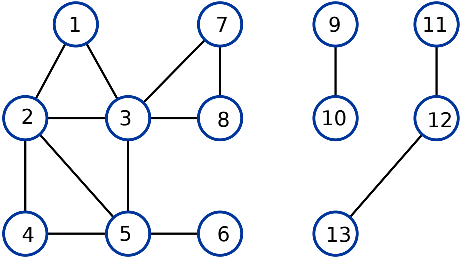
- Explore G as if it were a maze: start from s, traverse first edge out (to node v), traverse first edge out of v, ..., reach a dead-end, backtrack, ...

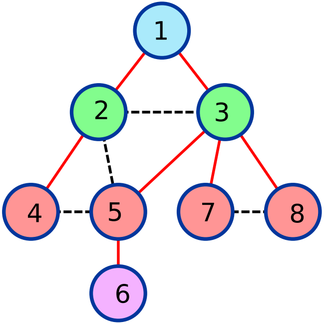
1. Mark all nodes as "`Unexplored`".
2. Invoke `DFS(`s`)`.

```
DFS(u):
  Mark u as "Explored" and add u to R
  For each edge (u,v) incident to u
    If v is not marked "Explored" then
      Recursively invoke DFS(v)
    Endif
  Endfor
```

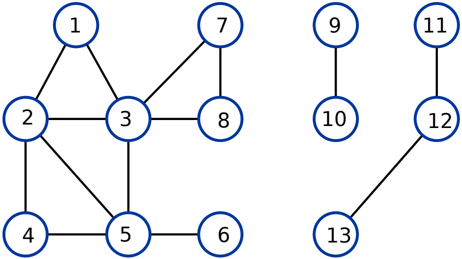
- *Depth-first search tree* is a tree T: when `DFS(`v`)` is invoked directly during the call to `DFS(`v`)`, add edge (u, v) to T.

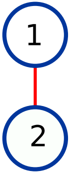
Example of DFS

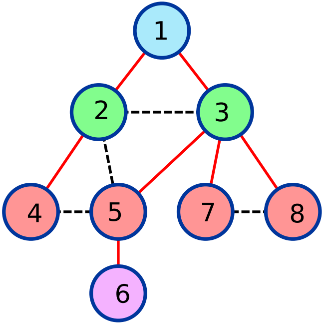




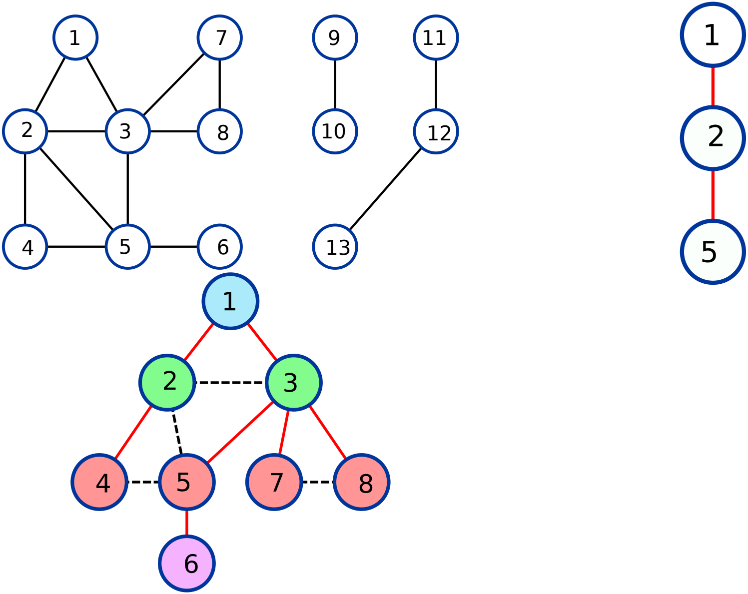
Example of DFS



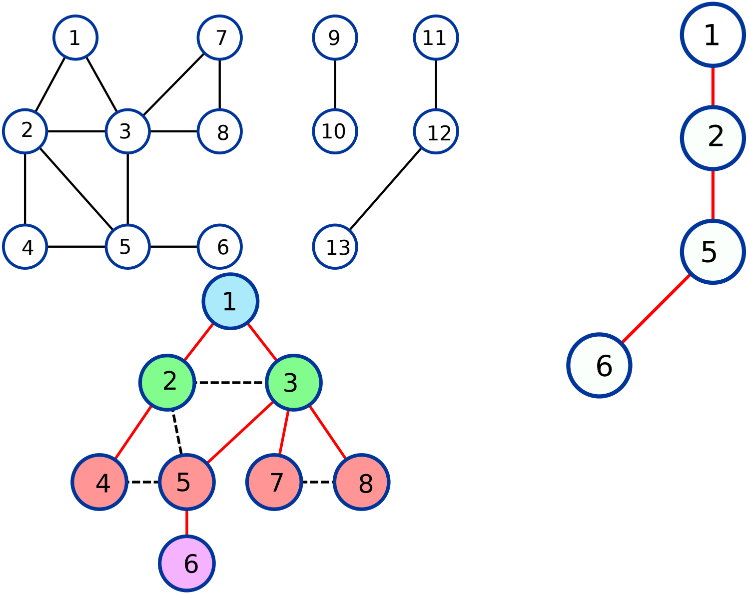




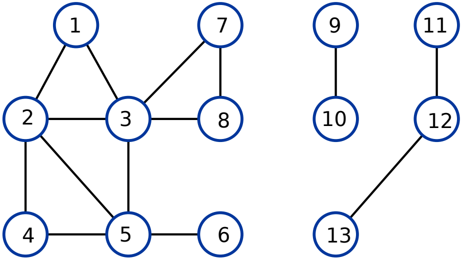
Example of DFS

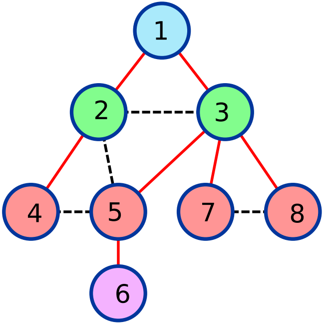


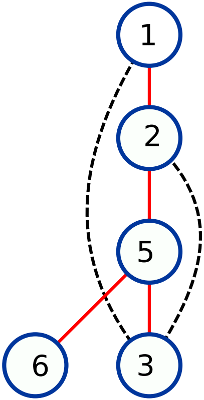
Example of DFS



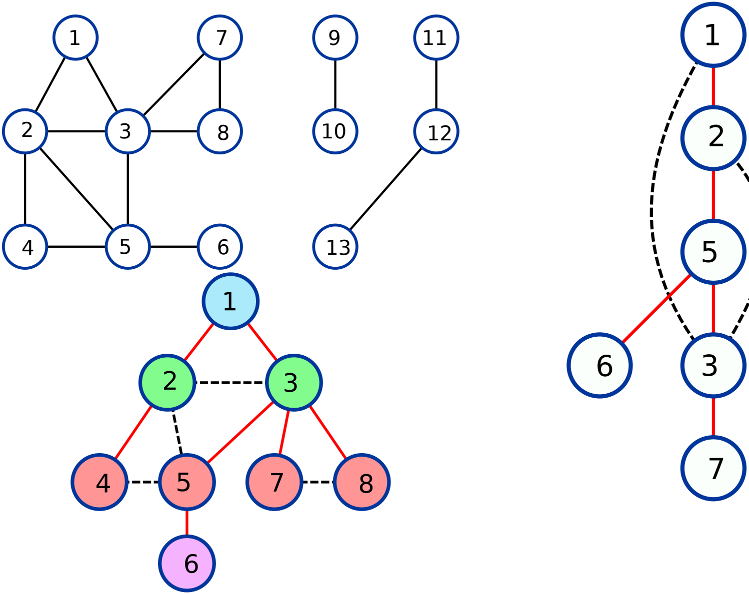
Example of DFS



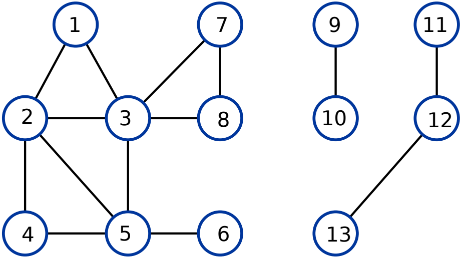


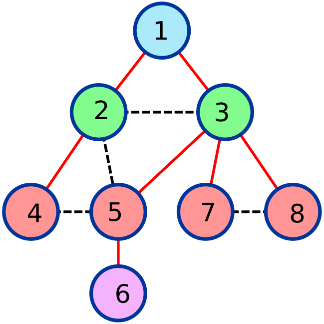


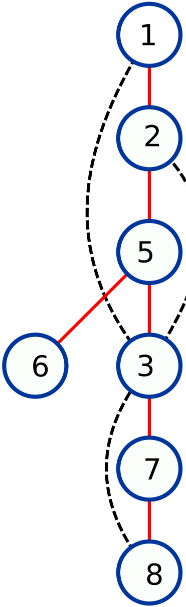
Example of DFS



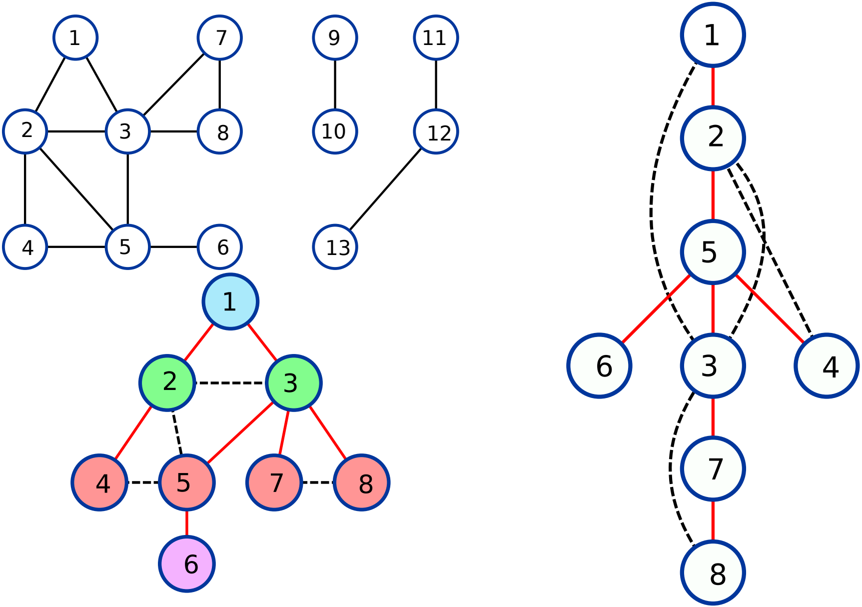
Example of DFS



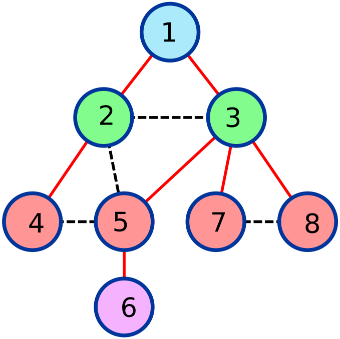


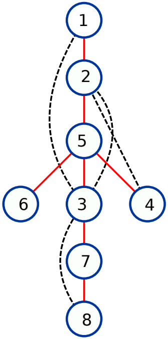


Example of DFS



BFS vs. DFS

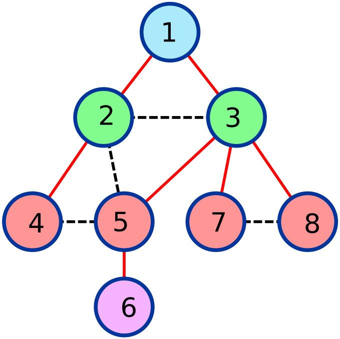


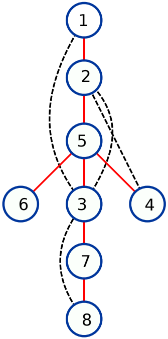


- Both visit the same set of nodes but in a different order.
- Both traverse all the edges in the connected component but in a different order.
- BFS trees have root-to-leaf paths that look as short as possible while paths in DFS trees tend to be long and deep.
- Non-tree edges

  BFS within the same level or between adjacent levels.

BFS vs. DFS

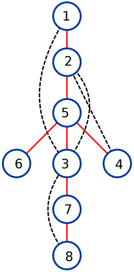




- Both visit the same set of nodes but in a different order.
- Both traverse all the edges in the connected component but in a different order.
- BFS trees have root-to-leaf paths that look as short as possible while paths in DFS trees tend to be long and deep.
- Non-tree edges
  - BFS within the same level or between adjacent levels.
  - DFS connect ancestors to descendants.

Properties of DFS Trees

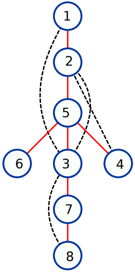
```
DFS(u):
  Mark u as "Explored" and add u to R
  For each edge (u,v) incident to u
    If v is not marked "Explored" then
      Recursively invoke DFS(v)
    Endif
  Endfor
```



- Observation: All nodes marked as "`Explored`" between the start of `DFS(u)` and its end are descendants of u in the DFS tree T.

Properties of DFS Trees

```
DFS(u):
  Mark u as "Explored" and add u to R
  For each edge (u,v) incident to u
    If v is not marked "Explored" then
      Recursively invoke DFS(v)
    Endif
  Endfor
```



- Observation: All nodes marked as "Explored" between the start of DFS(u) and its end are descendants of u in the DFS tree T.
- Claim: Let x and y be nodes in a DFS tree T such that (x, y) is an edge of G but not of T. Then one of x or y is an ancestor of the other in T. Read proof on page 86 of your textbook.

Representing Graphs

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - Size of the graph is defined to be $m + n$.
  - Strive for algorithms whose running time is linear in graph size, i.e., $O(m + n)$.

Representing Graphs

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - Size of the graph is defined to be $m + n$.
  - Strive for algorithms whose running time is linear in graph size, i.e., $O(m + n)$.
- Assume $V = \{1, 2, \ldots, n-1, n\}$.
- *Adjacency matrix* representation: $n \times n$ Boolean matrix, where the entry in row i and column j is 1 iff the graph contains the edge (i, j).
  - Space used is

Representing Graphs

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - Size of the graph is defined to be $m + n$.
  - Strive for algorithms whose running time is linear in graph size, i.e., $O(m + n)$.
- Assume $V = \{1, 2, \ldots, n-1, n\}$.
- *Adjacency matrix* representation: $n \times n$ Boolean matrix, where the entry in row i and column j is 1 iff the graph contains the edge (i, j).
  - Space used is $\Theta(n^2)$, which is optimal in the worst case.
  - Check if there is an edge between node i and node j in

Representing Graphs

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - Size of the graph is defined to be $m + n$.
  - Strive for algorithms whose running time is linear in graph size, i.e., $O(m + n)$.
- Assume $V = \{1, 2, \ldots, n-1, n\}$.
- *Adjacency matrix* representation: $n \times n$ Boolean matrix, where the entry in row i and column j is 1 iff the graph contains the edge (i, j).
  - Space used is $\Theta(n^2)$, which is optimal in the worst case.
  - Check if there is an edge between node i and node j in $O(1)$ time.
  - Iterate over all the edges incident on node i in

Representing Graphs

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - Size of the graph is defined to be $m + n$.
  - Strive for algorithms whose running time is linear in graph size, i.e., $O(m + n)$.
- Assume $V = \{1, 2, \ldots, n - 1, n\}$.
- *Adjacency matrix* representation: $n \times n$ Boolean matrix, where the entry in row i and column j is 1 iff the graph contains the edge (i, j).
  - Space used is $\Theta(n^2)$, which is optimal in the worst case.
  - Check if there is an edge between node i and node j in $O(1)$ time.
  - Iterate over all the edges incident on node i in $\Theta(n)$ time.

Representing Graphs

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - Size of the graph is defined to be $m + n$.
  - Strive for algorithms whose running time is linear in graph size, i.e., $O(m + n)$.
- Assume $V = \{1, 2, \ldots, n-1, n\}$.
- *Adjacency matrix* representation: $n \times n$ Boolean matrix, where the entry in row i and column j is 1 iff the graph contains the edge (i, j).
  - Space used is $\Theta(n^2)$, which is optimal in the worst case.
  - Check if there is an edge between node i and node j in $O(1)$ time.
  - Iterate over all the edges incident on node i in $\Theta(n)$ time.
- *Adjacency list* representation: array `Adj`, where `Adj`$[v]$ stores the list of all nodes adjacent to v.
  - An edge $e = (u, v)$ appears twice: in `Adj`$[u]$ and `Adj`$[v]$.

Representing Graphs

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - Size of the graph is defined to be $m + n$.
  - Strive for algorithms whose running time is linear in graph size, i.e., $O(m + n)$.
- Assume $V = \{1, 2, \ldots, n-1, n\}$.
- *Adjacency matrix* representation: $n \times n$ Boolean matrix, where the entry in row i and column j is 1 iff the graph contains the edge (i, j).
  - Space used is $\Theta(n^2)$, which is optimal in the worst case.
  - Check if there is an edge between node i and node j in $O(1)$ time.
  - Iterate over all the edges incident on node i in $\Theta(n)$ time.
- *Adjacency list* representation: array `Adj`, where `Adj`$[v]$ stores the list of all nodes adjacent to v.
  - An edge $e = (u, v)$ appears twice: in `Adj`$[u]$ and `Adj`$[v]$.
  - $n_v =$ the number of neighbours of node v.
  - Space used is

Representing Graphs

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - Size of the graph is defined to be $m + n$.
  - Strive for algorithms whose running time is linear in graph size, i.e., $O(m + n)$.
- Assume $V = \{1, 2, \ldots, n-1, n\}$.
- *Adjacency matrix* representation: $n \times n$ Boolean matrix, where the entry in row i and column j is 1 iff the graph contains the edge (i, j).
  - Space used is $\Theta(n^2)$, which is optimal in the worst case.
  - Check if there is an edge between node i and node j in $O(1)$ time.
  - Iterate over all the edges incident on node i in $\Theta(n)$ time.
- *Adjacency list* representation: array `Adj`, where `Adj`$[v]$ stores the list of all nodes adjacent to v.
  - An edge $e = (u, v)$ appears twice: in `Adj`$[u]$ and `Adj`$[v]$.
  - $n_v =$ the number of neighbours of node v.
  - Space used is $O(n + \sum_{v \in G} n_v) =$

Representing Graphs

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - Size of the graph is defined to be $m + n$.
  - Strive for algorithms whose running time is linear in graph size, i.e., $O(m + n)$.
- Assume $V = \{1, 2, \ldots, n-1, n\}$.
- *Adjacency matrix* representation: $n \times n$ Boolean matrix, where the entry in row i and column j is 1 iff the graph contains the edge (i, j).
  - Space used is $\Theta(n^2)$, which is optimal in the worst case.
  - Check if there is an edge between node i and node j in $O(1)$ time.
  - Iterate over all the edges incident on node i in $\Theta(n)$ time.
- *Adjacency list* representation: array `Adj`, where `Adj`$[v]$ stores the list of all nodes adjacent to v.
  - An edge $e = (u, v)$ appears twice: in `Adj`$[u]$ and `Adj`$[v]$.
  - $n_v =$ the number of neighbours of node v.
  - Space used is $O(n + \sum_{v \in G} n_v) = O(n + m)$, which is optimal for every graph.
  - Check if there is an edge between node u and node v in

Representing Graphs

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - Size of the graph is defined to be $m + n$.
  - Strive for algorithms whose running time is linear in graph size, i.e., $O(m + n)$.
- Assume $V = \{1, 2, \ldots, n-1, n\}$.
- *Adjacency matrix* representation: $n \times n$ Boolean matrix, where the entry in row i and column j is 1 iff the graph contains the edge (i, j).
  - Space used is $\Theta(n^2)$, which is optimal in the worst case.
  - Check if there is an edge between node i and node j in $O(1)$ time.
  - Iterate over all the edges incident on node i in $\Theta(n)$ time.
- *Adjacency list* representation: array `Adj`, where `Adj`$[v]$ stores the list of all nodes adjacent to v.
  - An edge $e = (u, v)$ appears twice: in `Adj`$[u]$ and `Adj`$[v]$.
  - $n_v =$ the number of neighbours of node v.
  - Space used is $O(n + \sum_{v \in G} n_v) = O(n + m)$, which is optimal for every graph.
  - Check if there is an edge between node u and node v in $O(n_u)$ time.
  - Iterate over all the edges incident on node u in

Representing Graphs

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - Size of the graph is defined to be $m + n$.
  - Strive for algorithms whose running time is linear in graph size, i.e., $O(m + n)$.
- Assume $V = \{1, 2, \ldots, n-1, n\}$.
- *Adjacency matrix* representation: $n \times n$ Boolean matrix, where the entry in row i and column j is 1 iff the graph contains the edge (i, j).
  - Space used is $\Theta(n^2)$, which is optimal in the worst case.
  - Check if there is an edge between node i and node j in $O(1)$ time.
  - Iterate over all the edges incident on node i in $\Theta(n)$ time.
- *Adjacency list* representation: array `Adj`, where `Adj`$[v]$ stores the list of all nodes adjacent to v.
  - An edge $e = (u, v)$ appears twice: in `Adj`$[u]$ and `Adj`$[v]$.
  - $n_v =$ the number of neighbours of node v.
  - Space used is $O(n + \sum_{v \in G} n_v) = O(n + m)$, which is optimal for every graph.
  - Check if there is an edge between node u and node v in $O(n_u)$ time.
  - Iterate over all the edges incident on node u in $\Theta(n_u)$ time.

Data Structures for Implementation

- "Implementation" of BFS and DFS: fully specify the algorithms and data structures so that we can obtain provably efficient times.
- Inner loop of both BFS and DFS: process the set of edges incident on a given node and the set of visited nodes.
- How do we store the set of visited nodes? Order in which we process the nodes is crucial.

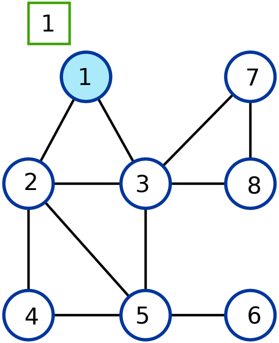
Data Structures for Implementation

- "Implementation" of BFS and DFS: fully specify the algorithms and data structures so that we can obtain provably efficient times.
- Inner loop of both BFS and DFS: process the set of edges incident on a given node and the set of visited nodes.
- How do we store the set of visited nodes? Order in which we process the nodes is crucial.
  - BFS: store visited nodes in a queue (first-in, first-out).
  - DFS: store visited nodes in a stack (last-in, first-out)

Using a Queue in BFS

- Maintain an array Discovered and set Discovered[v] = *true* as soon as the algorithm sees v.
- Maintain all the layers in a single queue L.

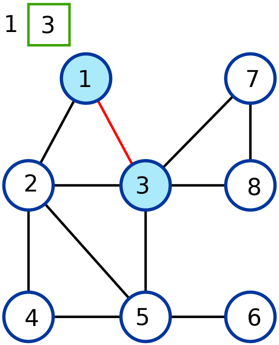
```
BFS(s):
   Set Discovered[s] = true
   Set Discovered[v] = false, for all other nodes v
   Initialize L to consist of the single element s
   While L is not empty
      Pop the node u at the head of L
      Consider each edge (u, v) incident on u
      If Discovered[v] = false then
         Set Discovered[v] = true
         Add edge (u, v) to the tree T
         Push v to the back of L
      Endif
   Endwhile
```



Using a Queue in BFS

- Maintain an array Discovered and set Discovered[v] = *true* as soon as the algorithm sees v.
- Maintain all the layers in a single queue L.

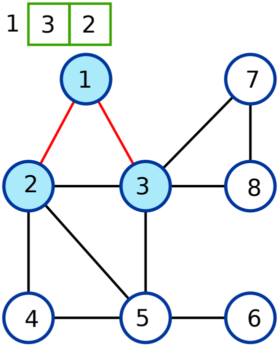
```
BFS(s):
   Set Discovered[s] = true
   Set Discovered[v] = false, for all other nodes v
   Initialize L to consist of the single element s
   While L is not empty
       Pop the node u at the head of L
       Consider each edge (u, v) incident on u
       If Discovered[v] = false then
          Set Discovered[v] = true
          Add edge (u, v) to the tree T
          Push v to the back of L
       Endif
   Endwhile
```



Using a Queue in BFS

- Maintain an array Discovered and set Discovered[v] $=$ *true* as soon as the algorithm sees v.
- Maintain all the layers in a single queue L.

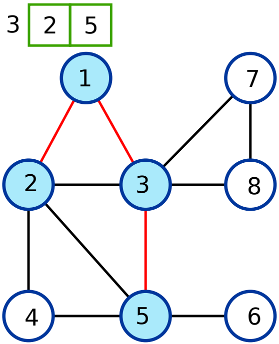
```
BFS(s):
   Set Discovered[s] = true
   Set Discovered[v] = false, for all other nodes v
   Initialize L to consist of the single element s
   While L is not empty
       Pop the node u at the head of L
       Consider each edge (u, v) incident on u
       If Discovered[v] = false then
          Set Discovered[v] = true
          Add edge (u, v) to the tree T
          Push v to the back of L
       Endif
   Endwhile
```



Using a Queue in BFS

- Maintain an array Discovered and set Discovered[v] = *true* as soon as the algorithm sees v.
- Maintain all the layers in a single queue L.

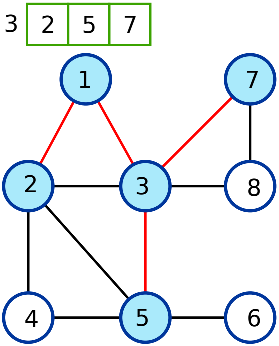
```
BFS(s):
   Set Discovered[s] = true
   Set Discovered[v] = false, for all other nodes v
   Initialize L to consist of the single element s
   While L is not empty
       Pop the node u at the head of L
       Consider each edge (u, v) incident on u
       If Discovered[v] = false then
          Set Discovered[v] = true
          Add edge (u, v) to the tree T
          Push v to the back of L
       Endif
   Endwhile
```



Using a Queue in BFS

- Maintain an array Discovered and set $\texttt{Discovered}[v] = \textit{true}$ as soon as the algorithm sees v.
- Maintain all the layers in a single queue L.

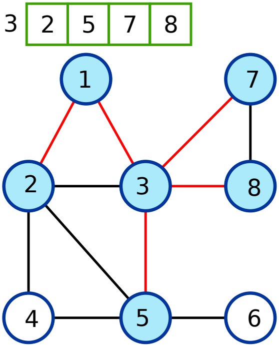
```
BFS(s):
   Set Discovered[s] = true
   Set Discovered[v] = false, for all other nodes v
   Initialize L to consist of the single element s
   While L is not empty
       Pop the node u at the head of L
       Consider each edge (u, v) incident on u
       If Discovered[v] = false then
          Set Discovered[v] = true
          Add edge (u, v) to the tree T
          Push v to the back of L
       Endif
   Endwhile
```



Using a Queue in BFS

- Maintain an array `Discovered` and set `Discovered`$[v] = true$ as soon as the algorithm sees v.
- Maintain all the layers in a single queue L.

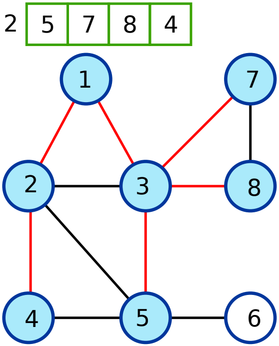
```
BFS(s):
  Set Discovered[s] = true
  Set Discovered[v] = false, for all other nodes v
  Initialize L to consist of the single element s
  While L is not empty
     Pop the node u at the head of L
     Consider each edge (u, v) incident on u
     If Discovered[v] = false then
        Set Discovered[v] = true
        Add edge (u, v) to the tree T
        Push v to the back of L
     Endif
  Endwhile
```



Using a Queue in BFS

- Maintain an array Discovered and set Discovered[v] $=$ *true* as soon as the algorithm sees v.
- Maintain all the layers in a single queue L.

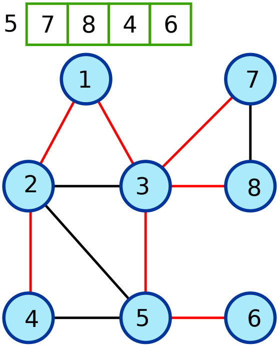
```
BFS(s):
   Set Discovered[s] = true
   Set Discovered[v] = false, for all other nodes v
   Initialize L to consist of the single element s
   While L is not empty
       Pop the node u at the head of L
       Consider each edge (u, v) incident on u
       If Discovered[v] = false then
          Set Discovered[v] = true
          Add edge (u, v) to the tree T
          Push v to the back of L
       Endif
   Endwhile
```



Using a Queue in BFS

- Maintain an array Discovered and set Discovered[v] = *true* as soon as the algorithm sees v.
- Maintain all the layers in a single queue L.

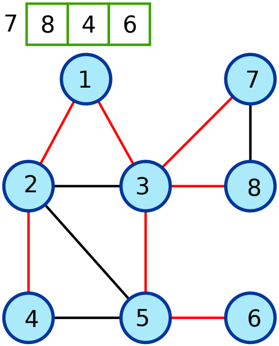
```
BFS(s):
   Set Discovered[s] = true
   Set Discovered[v] = false, for all other nodes v
   Initialize L to consist of the single element s
   While L is not empty
      Pop the node u at the head of L
      Consider each edge (u,v) incident on u
      If Discovered[v] = false then
         Set Discovered[v] = true
         Add edge (u,v) to the tree T
         Push v to the back of L
      Endif
   Endwhile
```



Using a Queue in BFS

- Maintain an array Discovered and set Discovered$[v] = true$ as soon as the algorithm sees v.
- Maintain all the layers in a single queue L.

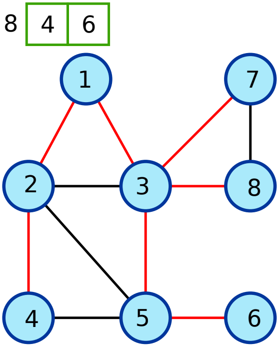
```
BFS(s):
   Set Discovered[s] = true
   Set Discovered[v] = false, for all other nodes v
   Initialize L to consist of the single element s
   While L is not empty
      Pop the node u at the head of L
      Consider each edge (u, v) incident on u
      If Discovered[v] = false then
         Set Discovered[v] = true
         Add edge (u, v) to the tree T
         Push v to the back of L
      Endif
   Endwhile
```



Using a Queue in BFS

- Maintain an array Discovered and set Discovered$[v]$ = *true* as soon as the algorithm sees v.
- Maintain all the layers in a single queue L.

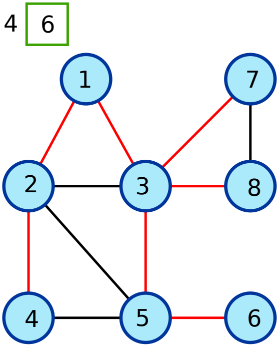
```
BFS(s):
  Set Discovered[s] = true
  Set Discovered[v] = false, for all other nodes v
  Initialize L to consist of the single element s
  While L is not empty
     Pop the node u at the head of L
     Consider each edge (u, v) incident on u
     If Discovered[v] = false then
        Set Discovered[v] = true
        Add edge (u, v) to the tree T
        Push v to the back of L
     Endif
  Endwhile
```



Using a Queue in BFS

- Maintain an array Discovered and set Discovered[v] $=$ *true* as soon as the algorithm sees v.
- Maintain all the layers in a single queue L.

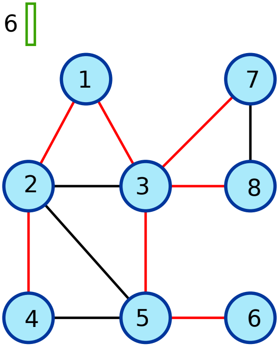
```
BFS(s):
   Set Discovered[s] = true
   Set Discovered[v] = false, for all other nodes v
   Initialize L to consist of the single element s
   While L is not empty
       Pop the node u at the head of L
       Consider each edge (u, v) incident on u
       If Discovered[v] = false then
          Set Discovered[v] = true
          Add edge (u, v) to the tree T
          Push v to the back of L
       Endif
   Endwhile
```



Using a Queue in BFS

- Maintain an array Discovered and set Discovered$[v]$ = *true* as soon as the algorithm sees v.
- Maintain all the layers in a single queue L.

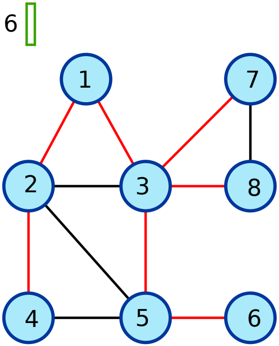
```
BFS(s):
   Set Discovered[s] = true
   Set Discovered[v] = false, for all other nodes v
   Initialize L to consist of the single element s
   While L is not empty
      Pop the node u at the head of L
      Consider each edge (u, v) incident on u
      If Discovered[v] = false then
         Set Discovered[v] = true
         Add edge (u, v) to the tree T
         Push v to the back of L
      Endif
   Endwhile
```



Using a Queue in BFS

- Maintain an array Discovered and set Discovered[v] $=$ *true* as soon as the algorithm sees v.
- Maintain all the layers in a single queue L.

```
BFS(s):
  Set Discovered[s] = true
  Set Discovered[v] = false, for all other nodes v
  Initialize L to consist of the single element s
  While L is not empty
     Pop the node u at the head of L
     Consider each edge (u, v) incident on u
     If Discovered[v] = false then
        Set Discovered[v] = true
        Add edge (u, v) to the tree T
        Push v to the back of L
     Endif
  Endwhile
```

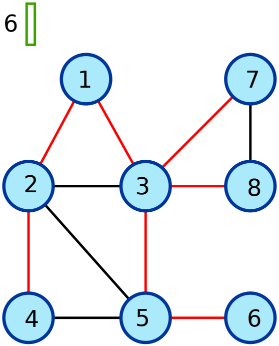


- Simple to modify this procedure to keep track of layer numbers as well.

Using a Queue in BFS

- Maintain an array Discovered and set Discovered[v] = *true* as soon as the algorithm sees v.
- Maintain all the layers in a single queue L.

```
BFS(s):
   Set Discovered[s] = true
   Set Discovered[v] = false, for all other nodes v
   Initialize L to consist of the single element s
   While L is not empty
       Pop the node u at the head of L
       Consider each edge (u, v) incident on u
       If Discovered[v] = false then
          Set Discovered[v] = true
          Add edge (u, v) to the tree T
          Push v to the back of L
       Endif
   Endwhile
```

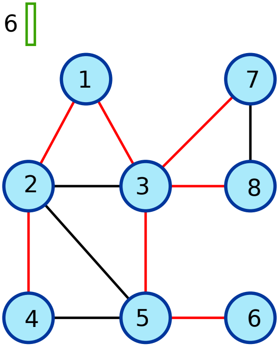


- Simple to modify this procedure to keep track of layer numbers as well. Store the pair (u, l_u), where l_u is the index of the layer containing u.

Using a Queue in BFS

- Maintain an array Discovered and set Discovered$[v]$ = *true* as soon as the algorithm sees v.
- Maintain all the layers in a single queue L.

```
BFS(s):
   Set Discovered[s] = true
   Set Discovered[v] = false, for all other nodes v
   Initialize L to consist of the single element s
   While L is not empty
      Pop the node u at the head of L
      Consider each edge (u, v) incident on u
      If Discovered[v] = false then
         Set Discovered[v] = true
         Add edge (u, v) to the tree T
         Push v to the back of L
      Endif
   Endwhile
```



- Simple to modify this procedure to keep track of layer numbers as well. Store the pair (u, l_u), where l_u is the index of the layer containing u.
- Claim: More formally: If BFS(s) pops (v, l_v) from L immediately after it pops (u, l_u), then either $l_v = l_u$ or $l_v = l_u + 1$.

Analysis of BFS Implementation

```
BFS(s):
   Set Discovered[s] = true
   Set Discovered[v] = false, for all other nodes v
   Initialize L to consist of the single element s
   While L is not empty
       Pop the node u at the head of L
       Consider each edge (u, v) incident on u
       If Discovered[v] = false then
          Set Discovered[v] = true
          Add edge (u, v) to the tree T
          Push v to the back of L
       Endif
   Endwhile
```

- How many times is a node popped from L?

Analysis of BFS Implementation

BFS(s):
    Set Discovered[s] = true
    Set Discovered[v] = false, for all other nodes v
    Initialize L to consist of the single element s
    While L is not empty
        Pop the node u at the head of L
        Consider each edge (u, v) incident on u
        If Discovered[v] = false then
            Set Discovered[v] = true
            Add edge (u, v) to the tree T
            Push v to the back of L
        Endif
    Endwhile

- How many times is a node popped from L? Exactly once.

Analysis of BFS Implementation

```
BFS(s):
   Set Discovered[s] = true
   Set Discovered[v] = false, for all other nodes v
   Initialize L to consist of the single element s
   While L is not empty
       Pop the node u at the head of L
       Consider each edge (u, v) incident on u
       If Discovered[v] = false then
          Set Discovered[v] = true
          Add edge (u, v) to the tree T
          Push v to the back of L
       Endif
   Endwhile
```

- How many times is a node popped from L? Exactly once.
- Time used by `for` loop for a node u:

Analysis of BFS Implementation

```
BFS(s):
   Set Discovered[s] = true
   Set Discovered[v] = false, for all other nodes v
   Initialize L to consist of the single element s
   While L is not empty
       Pop the node u at the head of L
       Consider each edge (u, v) incident on u
       If Discovered[v] = false then
          Set Discovered[v] = true
          Add edge (u, v) to the tree T
          Push v to the back of L
       Endif
   Endwhile
```

- How many times is a node popped from L? Exactly once.
- Time used by `for` loop for a node u: $O(n_u)$ time.

Analysis of BFS Implementation

```
BFS(s):
   Set Discovered[s] = true
   Set Discovered[v] = false, for all other nodes v
   Initialize L to consist of the single element s
   While L is not empty
       Pop the node u at the head of L
       Consider each edge (u, v) incident on u
       If Discovered[v] = false then
          Set Discovered[v] = true
          Add edge (u, v) to the tree T
          Push v to the back of L
       Endif
   Endwhile
```

- How many times is a node popped from L? Exactly once.
- Time used by `for` loop for a node u: $O(n_u)$ time.
- Total time for all `for` loops: $\sum_{u \in G} O(n_u) = O(m)$ time.
- Maintaining layer information:

Analysis of BFS Implementation

```
BFS(s):
   Set Discovered[s] = true
   Set Discovered[v] = false, for all other nodes v
   Initialize L to consist of the single element s
   While L is not empty
       Pop the node u at the head of L
       Consider each edge (u, v) incident on u
       If Discovered[v] = false then
          Set Discovered[v] = true
          Add edge (u, v) to the tree T
          Push v to the back of L
       Endif
   Endwhile
```

- How many times is a node popped from L? Exactly once.
- Time used by `for` loop for a node u: $O(n_u)$ time.
- Total time for all `for` loops: $\sum_{u \in G} O(n_u) = O(m)$ time.
- Maintaining layer information: $O(1)$ time per node.
- Total time is $O(n + m)$.

Recursive DFS to Stack-Based DFS

```
DFS(u):
  Mark u as "Explored" and add u to R
  For each edge (u, v) incident to u
    If v is not marked "Explored" then
      Recursively invoke DFS(v)
    Endif
  Endfor
```

- Procedure has "tail recursion": recursive call is the last step.

Recursive DFS to Stack-Based DFS

```
DFS(u):
  Mark u as "Explored" and add u to R
  For each edge (u, v) incident to u
    If v is not marked "Explored" then
      Recursively invoke DFS(v)
    Endif
  Endfor
```

- Procedure has "tail recursion": recursive call is the last step.
- Can replace the recursion by an iteration: use a stack to explicitly implement the recursion.

Analysing DFS

```
DFS(s):
  Initialize S to be a stack with one element s
  While S is not empty
    Take a node u from S
    If Explored[u] = false then
      Set Explored[u] = true
      For each edge (u, v) incident to u
        Add v to the stack S
      Endfor
    Endif
  Endwhile
```

- How many times is a node's adjacency list scanned?

Analysing DFS

```
DFS(s):
  Initialize S to be a stack with one element s
  While S is not empty
    Take a node u from S
    If Explored[u] = false then
       Set Explored[u] = true
       For each edge (u, v) incident to u
         Add v to the stack S
       Endfor
    Endif
  Endwhile
```

- How many times is a node's adjacency list scanned? Exactly once.

Analysing DFS

```
DFS(s):
  Initialize S to be a stack with one element s
  While S is not empty
    Take a node u from S
    If Explored[u] = false then
      Set Explored[u] = true
      For each edge (u, v) incident to u
        Add v to the stack S
      Endfor
    Endif
  Endwhile
```

- How many times is a node's adjacency list scanned? Exactly once.
- The total amount of time to process edges incident on node u's is

Analysing DFS

```
DFS(s):
  Initialize S to be a stack with one element s
  While S is not empty
    Take a node u from S
    If Explored[u] = false then
      Set Explored[u] = true
      For each edge (u, v) incident to u
        Add v to the stack S
      Endfor
    Endif
  Endwhile
```

- How many times is a node's adjacency list scanned? Exactly once.
- The total amount of time to process edges incident on node u's is $O(n_u)$.
- The total running time of the algorithm is

Analysing DFS

```
DFS(s):
  Initialize S to be a stack with one element s
  While S is not empty
    Take a node u from S
    If Explored[u] = false then
      Set Explored[u] = true
      For each edge (u, v) incident to u
        Add v to the stack S
      Endfor
    Endif
  Endwhile
```

- How many times is a node's adjacency list scanned? Exactly once.
- The total amount of time to process edges incident on node u's is $O(n_u)$.
- The total running time of the algorithm is $O(n + m)$.