## Objectives

Data structures: Graphs



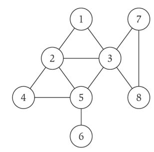
## Undirected Graphs G = (V, E)

V = nodes (vertices)

E = edges between pairs of nodes

Captures pairwise relationship between objects

Graph size parameters: $n = |V|$, $m = |E|$



V = { 1, 2, 3, 4, 5, 6, 7, 8 }
E = { 1-2, 1-3, 2-3, 2-4, 2-5, 3-5, 3-7, 3-8, 4-5, 5-6 }
n = 8
m = 11

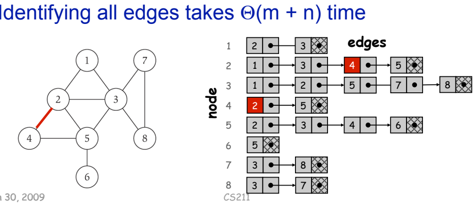


## Graph Representation: Adjacency List

Node indexed array of lists

- Two representations of each edge
- Space = 2m + n = O(m + n)
- Checking if (u, v) is an edge takes O(deg(u)) time
- Identifying all edges takes Θ(m + n) time

degree = number of neighbors of u





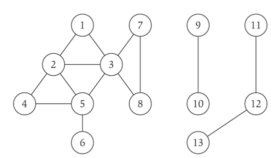

## Paths and Connectivity

Def. A *path* in an undirected graph G = (V, E) is a sequence P of nodes $v_1, v_2, \ldots, v_{k-1}, v_k$

- each consecutive pair $v_i$, $v_{i+1}$ is joined by an edge in E

Def. A path is *simple* if all nodes are distinct

Def. An undirected graph is *connected* if ∀ pair of nodes u and v, there is a path between u and v

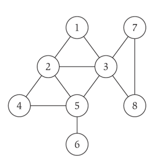


•*Short path*
•*Distance*



## Cycles

Def. A *cycle* is a path $v_1, v_2, \ldots, v_{k-1}, v_k$ in which $v_1 = v_k$, k > 2, and the first k-1 nodes are all distinct



cycle *C* = 1-2-4-5-3-1

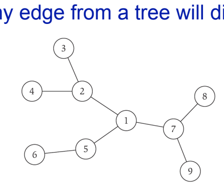


## Trees

Def. An undirected graph is a *tree* if it is connected and does not contain a cycle

Simplest connected graph

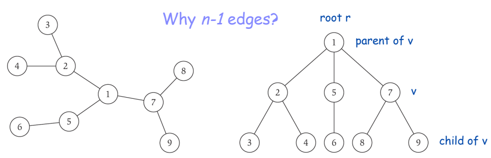
- Deleting any edge from a tree will disconnect it



## Rooted Trees

Given a tree T, choose a root node *r* and orient each edge away from *r*

Models hierarchical structure




a tree

the same tree, rooted at 1

Jan 30, 2009 CS211 7

## GRAPH TRAVERSAL

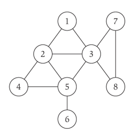
8

## Connectivity

s-t connectivity problem. Given two node *s* and *t*, is there a path between *s* and *t*?

s-t shortest path problem. Given two node *s* and *t*, what is the length of the shortest path between s and t?

Applications



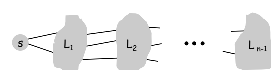
- Facebook
- Maze traversal
- Kevin Bacon number
- Fewest number of hops in a communication network

Jan 30, 2009 CS211 9

## Breadth First Search

Intuition. Explore outward from *s* in all possible directions, adding nodes one "layer" at a time

Algorithm



- $L_0 = \{ s \}$
- $L_1$ = all neighbors of $L_0$
- $L_2$ = all nodes that do not belong to $L_0$ or $L_1$, and that have an edge to a node in $L_1$
- $L_{i+1}$ = all nodes that do not belong to an earlier layer, and that have an edge to a node in $L_i$

Theorem. For each *i*, $L_i$ consists of all nodes at distance exactly *i* from *s*. There is a path from *s* to *t* iff *t* appears in some layer.

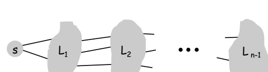
*What does this mean?*

Jan 30, 2009 CS211 10

## Breadth First Search

Theorem. For each *i*, $L_i$ consists of all nodes at distance exactly *i* from *s*. There is a path from *s* to *t* iff *t* appears in some layer.

- Shortest path to *t* from *s*, is the *i* from $L_i$
- All nodes *reachable* from *s* are in $L_1, L_2, \ldots, L_{n-1}$



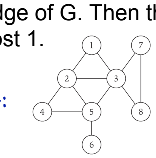
Jan 30, 2009 CS211 11

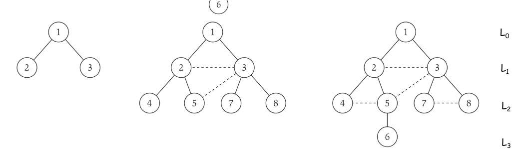
## Breadth First Search

Property. Let T be a BFS tree of G = (V, E), and let (x, y) be an edge of G. Then the level of x and y differ by at most 1.

If x is in $L_i$, then y must be in $L_{i+1}$ or earlier

G:






(a) (b) (c)

Jan 30, 2009 12

## Implementation: Maintaining Sets

Either a queue or a stack



## Implementation: Maintaining Sets

Either a queue or a stack

Queue: FIFO

- First in, first out

Stack: LIFO

- Last in, last out

Both as a doubly linked list

- Always take first on list
- Difference in where inserted
  - Have first and last pointers
  - Done in constant time



## Implementing BFS

Graph: Adjacency list

Discovered array

Maintain layers in separate lists, L[i]



## Implementing BFS

Graph: Adjacency list

Discovered array

Maintain layers in separate lists, L[i]

```
BFS(s):
    Discovered[v] = false, for all v
    Discovered[s] = true
    L[0] = {s}
    layer counter i = 0
    BFS tree T = {}
    while L[i] != {}
       L[i+1] = {}
       For each node u ∈ L[i]
           Consider each edge (u,v) incident to u
           if Discovered[v] = false then
                Set Discovered[v] = true
                Add edge (u, v) to tree T
                Add v to the list L[i + 1]
       i+=1
```

L[i] as a queue or stack?



## Analysis

```
BFS(s):
    Discovered[v] = false, for all v
    Discovered[s] = true
    L[0] = {s}
    layer counter i = 0
    BFS tree T = {}
    while L[i] != {}
       L[i+1] = {}
       For each node u ∈ L[i]
           Consider each edge (u,v) incident to u
           if Discovered[v] = false then
                Set Discovered[v] = true
                Add edge (u, v) to tree T
                Add v to the list L[i + 1]
       i+=1
```



## Analysis

```
BFS(s):
    Discovered[v] = false, for all v
    Discovered[s] = true
    L[0] = {s}
    layer counter i = 0
    BFS tree T = {}
    while L[i] != {}
       L[i+1] = {}
       For each node u ∈ L[i]
           Consider each edge (u,v) incident to u
           if Discovered[v] = false then
                Set Discovered[v] = true
                Add edge (u, v) to tree T
                Add v to the list L[i + 1]
       i+=1
```

n

$O(n^2)$

At most n

At most n-1

## Analysis

n

```
BFS(s):
   Discovered[v] = false, for all v
   Discovered[s] = true
   L[0] = {s}
   layer counter i = 0
   BFS tree T = {}
   while L[i] != {}
      L[i+1] = {}
      For each node u ∈ L[i]
          Consider each edge (u,v) incident to u
          if Discovered[v] = false then
              Set Discovered[v] = true
              Add edge (u, v) to tree T
              Add v to the list L[i + 1]
      i+=1
```

At most n

O(deg(u))

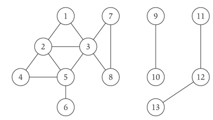
$\Sigma_{u \in V} \deg(u) = 2m$



## Connected Component

Find all nodes *reachable* from s

- BFS is one approach




Connected component containing node 1 = { 1, 2, 3, 4, 5, 6, 7, 8 }

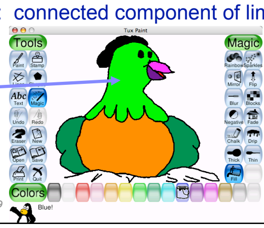


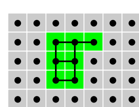
## Application: Flood Fill

Given lime green pixel in an image, change color of entire blob of neighboring lime pixels to blue

- Node: pixel
- Edge: two neighboring lime pixels
- Blob: connected component of lime pixels

recolor lime green blob to blue



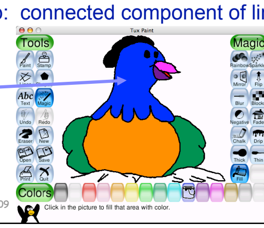


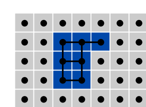
## Application: Flood Fill

Given lime green pixel in an image, change color of entire blob of neighboring lime pixels to blue

- Node: pixel
- Edge: two neighboring lime pixels
- Blob: connected component of lime pixels

recolor lime green blob to blue



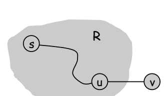


## Connected Component

Find all nodes *reachable* from s

In general….

$R$ 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




it's safe to add v

Theorem. Upon termination, R is the connected component containing s

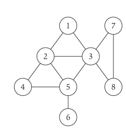
- BFS = explore in order of distance from s
- DFS = explore in a different way



## Depth First Search

How does DFS work on this graph?

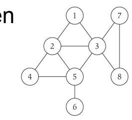
- Starting from node 1



## Depth First Search

Need to keep track of where you've been

When reach a "dead-end" (already explored all neighbors), backtrack to node with unexplored neighbor



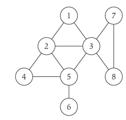

Algorithm:

```
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
        DFS(v)
```



## DFS vs BFS

Resulting trees?






## Implementing DFS



## Implementing DFS

Keep nodes to be processed in a *stack*

```
DFS(s):
   Initialize S to be a stack with one element s
   Explored[v] = false, for all v
   Parent[v] = 0, for all v
   DFS tree T = {}
   while S != {}
     Take a node u from S
     If Explored[u] = false
        Explored[u] = true
        Add edge (u, parent[u]) to T (if u ≠ s)
        For each edge (u, v) incident to u
           Add v to the stack S
           Parent[v] = u
```



## Analyzing DFS

```
DFS(s):
   Initialize S to be a stack with one element s
   Explored[v] = false, for all v
   Parent[v] = 0, for all v
   DFS tree T = {}
   while S != {}
     Take a node u from S
     If Explored[u] = false
        Explored[u] = true
        Add edge (u, parent[u]) to T (if u ≠ s)
        For each edge (u, v) incident to u
           Add v to the stack S
           Parent[v] = u
```



## Analyzing DFS

O(n+m)

```
DFS(s):
   Initialize S to be a stack with one element s
   Explored[v] = false, for all v
   Parent[v] = 0, for all v
   DFS tree T = {}
   while S != {}
     Take a node u from S
     If Explored[u] = false
        Explored[u] = true
        Add edge (u, parent[u]) to T (if u ≠ s)
  deg(u)For each edge (u, v) incident to u
           Add v to the stack S (if not explored?)
           Parent[v] = u
```

## Set of All Connected Components

For any two nodes *s* and *t* in a graph, their connected components are either identical or disjoint

Proof?



## Set of All Connected Components

For any two nodes *s* and *t* in a graph, their connected components are either identical or disjoint

Proof sketch:

(i) There is a path between *s* and *t* → same set of connected components

(ii) There is no path between *s* and *t* → disjoint set of connected components



## Set of All Connected Components

How can we find all connected components of graph?



## Set of All Connected Components

How can we find set of all connected components of graph?

```
R* = set of connected components
While there is a node that does not belong to R*
    select s not in R*
```

```
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
```

```
Add R to R*
```

Running time?



## Set of All Connected Components

How can we find set of all connected components of graph?

```
R* = set of connected components
While there is a node that does not belong to R*
    select s not in R*
```

```
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
```

```
Add R to R*
```

Running time: $O(m+n)$



## TESTING BIPARTITENESS

## Bipartite Graphs

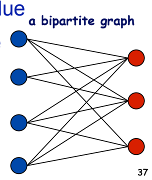
Def. An undirected graph G = (V, E) is bipartite if the nodes can be colored red or blue such that every edge has one red and one blue end

- Generally: vertices divided into sets X and Y

Applications:

- Stable marriage: men = red, women = blue
- Scheduling: machines = red, jobs = blue

a bipartite graph



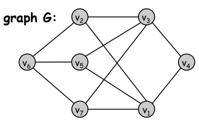


## Testing Bipartiteness

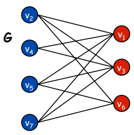
Given a graph G, is it bipartite?

- Many graph problems become:
  - easier if underlying graph is bipartite (matching)
  - tractable if underlying graph is bipartite (independent set)
- Before designing an algorithm, need to understand structure of bipartite graphs

a bipartite graph G:



another drawing of G





## An Obstruction to Bipartiteness

Lemma. If a graph G is bipartite, it cannot contain an odd length cycle.

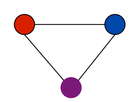
Proof Intuition. Consider a cycle of 3, then a larger odd cycle



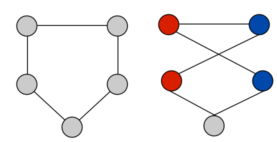
## An Obstruction to Bipartiteness

Lemma. If a graph G is bipartite, it cannot contain an odd length cycle.

Proof Intuition. Consider a cycle of 3, then a larger odd cycle



Not bipartite
(2-colorable)



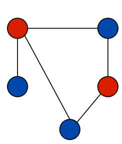
not bipartite
(not 2-colorable)



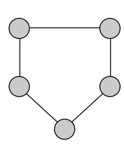
## An Obstruction to Bipartiteness

Lemma. If a graph G is bipartite, it cannot contain an odd length cycle.

Pf. Not possible to 2-color the odd cycle, let alone G.



bipartite
(2-colorable)



not bipartite
(not 2-colorable)

If find an odd cycle, graph is NOT bipartite



## How Can We Determine Bipartite Graphs?

Given a connected graph Why connected?

Color one node red

- Doesn't matter which color (Why?)

What should we do next?

How will we know that we're finished?

What does this process sound like?

## How Can We Determine Bipartite Graphs?

Given a connected graph

- Color one node red
  - Doesn't matter which color (Why?)
- What should we do next?

How will we know that we're finished?

What does this process sound like?

- BFS: alternating colors, layers