Introduction to Graph Theory & Traversals

Algorithm Design & Analysis

Fall 2018
Outline

Graph Data Structures

Graph Traversals

Breadth-First Search: BFS

BFS: Extensions and Optimizations

Depth-First Search

Directed Graphs
Aside: Graph Data Structure Review
(You should know this!)

Assume $G$ has vertices $\{1, \ldots, n\}$ and $m$ edges.
Aside: Graph Data Structure Review
(You should know this!)

Assume $G$ has vertices $\{1, \ldots, n\}$ and $m$ edges.

Two basic structures for graphs (directed or undirected)
Aside: Graph Data Structure Review (You should know this!)

Assume $G$ has vertices $\{1, \ldots, n\}$ and $m$ edges.

Two basic structures for graphs (directed or undirected)

- **The Adjacency Matrix:** An $n \times n$ array $E$, where $E[u, v] = 1$ if $u, v$ form an edge and $E[u, v] = 0$ otherwise.
Aside: Graph Data Structure Review
(You should know this!)

Assume $G$ has vertices $\{1, \ldots, n\}$ and $m$ edges.

Two basic structures for graphs (directed or undirected)

- **The Adjacency Matrix**: An $n \times n$ array $E$, where $E[u, v] = 1$ if $u, v$ form an edge and $E[u, v] = 0$ otherwise
- **The Adjacency List**: An array $E[1..n]$ of edge lists; elements in the edge list $E[u]$ identify neighbors of $u$
The Adjacency Matrix: An $n \times n$ array $E$, where $E[u, v] = 1$ if $u, v$ form an edge and $E[u, v] = 0$ otherwise.
**Adjacency Matrix**

**The Adjacency Matrix:** An $n \times n$ array $E$, where $E[u, v] = 1$ if $u, v$ form an edge and $E[u, v] = 0$ otherwise.

- If $G$ is undirected, $E[u, v] = E[v, u]$ (or $E$ is compacted by only having values when $u < v$).
**Adjacency Matrix**

**The Adjacency Matrix:** An $n \times n$ array $E$, where $E[u, v] = 1$ if $u, v$ form an edge and $E[u, v] = 0$ otherwise

- If $G$ is undirected, $E[u, v] = E[v, u]$ (or $E$ is compacted by only having values when $u < v$).
- Uses $O(n^2)$ space regardless of number of edges
**Adjacency Matrix**

**The Adjacency Matrix:** An $n \times n$ array $E$, where $E[u, v] = 1$ if $u, v$ form an edge and $E[u, v] = 0$ otherwise

- If $G$ is undirected, $E[u, v] = E[v, u]$ (or $E$ is compacted by only having values when $u < v$).
- Uses $O(n^2)$ space regardless of number of edges
- Edge queries can be answered in $O(1)$ time; "next neighbor of $v$" can take $O(n)$ time.
**Adjacency Lists**

**The Adjacency List:** An array $E[1..n]$ of edge lists; elements in the edge list $E[u]$ identify neighbors of $u$


**Adjacency Lists**

*The Adjacency List:* An array $E[1..n]$ of edge lists; elements in the edge list $E[u]$ identify neighbors of $u$

- If $G$ is undirected, $v \in E[u]$ iff $u \in E[v]$ (or $E$ is compacted by only having values when $u < v$).
The Adjacency List: An array $E[1..n]$ of edge lists; elements in the edge list $E[u]$ identify neighbors of $u$

- If $G$ is undirected, $v \in E[u]$ iff $u \in E[v]$ (or $E$ is compacted by only having values when $u < v$).
- Uses $O(n + m)$ space: better than Adjacency Array if $m \in o(n^2)$ (for example, when $G$ is planar)
Adjacency Lists

The Adjacency List: An array $E[1..n]$ of edge lists; elements in the edge list $E[u]$ identify neighbors of $u$

- If $G$ is undirected, $v \in E[u]$ iff $u \in E[v]$ (or $E$ is compacted by only having values when $u < v$).
- Uses $O(n + m)$ space: better than Adjacency Array if $m \in o(n^2)$ (for example, when $G$ is planar)

Note: $g(n) \in o(f(n))$ if $\lim_{n \to \infty} \frac{g(n)}{f(n)} = 0$
Adjacency Lists

The Adjacency List: An array $E[1..n]$ of edge lists; elements in the edge list $E[u]$ identify neighbors of $u$

- If $G$ is undirected, $v \in E[u]$ iff $u \in E[v]$ (or $E$ is compacted by only having values when $u < v$).
- Uses $O(n + m)$ space: better than Adjacency Array if $m \in o(n^2)$ (for example, when $G$ is planar)
  Note: $g(n) \in o(f(n))$ if $\lim_{n \to \infty} \frac{g(n)}{f(n)} = 0$
- "Next neighbor of $v"$ takes $O(1)$ time; edge queries can take $O(n)$ time.
Priority-Based Traversals

Two popular methods: Breadth-First Search (BFS) and Depth-First Search. Given a connected graph, they can both
• respect the local structure of the graph
• visit every vertex and every edge
• produce a spanning tree
• can be used to determine basic graph properties such as connectedness
• can be tweaked to work on directed graphs as well

Both are special cases of priority-based traversal.
Two popular methods: Breadth-First Search (BFS) and Depth-First Search. Given a connected graph, they can both
Two popular methods: Breadth-First Search (BFS) and Depth-First Search. Given a connected graph, they can both

- respect the local structure of the graph
Two popular methods: Breadth-First Search (BFS) and Depth-First Search. Given a connected graph, they can both

- respect the local structure of the graph
- visit every vertex and every edge
Two popular methods: Breadth-First Search (BFS) and Depth-First Search. Given a connected graph, they can both

- respect the local structure of the graph
- visit every vertex and every edge
- produce a spanning tree
Two popular methods: Breadth-First Search (BFS) and Depth-First Search. Given a connected graph, they can both

- respect the local structure of the graph
- visit every vertex and every edge
- produce a spanning tree
- can be used to determine basic graph properties such as connectedness
Two popular methods: Breadth-First Search (BFS) and Depth-First Search. Given a connected graph, they can both

- respect the local structure of the graph
- visit every vertex and every edge
- produce a spanning tree
- can be used to determine basic graph properties such as connectedness
- can be tweaked to work on directed graphs as well
Priority-Based Traversals

Two popular methods: Breadth-First Search (BFS) and Depth-First Search. Given a connected graph, they can both

- respect the local structure of the graph
- visit every vertex and every edge
- produce a spanning tree
- can be used to determine basic graph properties such as connectedness
- can be tweaked to work on directed graphs as well

Both are special cases of priority-based traversal.
BFS: An Example

Three types of vertex: unvisited, visited, and explored.
BFS: An Example

Three types of vertex: unvisited, visited, and explored.
**BFS: An Example**

Three types of vertex: **unvisited, visited, and explored.**
BFS: An Example

Three types of vertex: unvisited, visited, and explored.
BFS: An Example

Three types of vertex: unvisited, visited, and explored.
Three types of vertex: unvisited, visited, and explored.
BFS: An Example

Three types of vertex: unvisited, visited, and explored.
What do we need to describe a BFS algorithm?
What do we need to describe a BFS algorithm?

- Know which vertex to explore next
What do we need to describe a BFS algorithm?

- Know which vertex to explore next
- Know which vertices we should not look at again
What do we need to describe a BFS algorithm?

- Know which vertex to explore next
- Know which vertices we should not look at again
- Identify three vertex states: unvisited, visited but not explored, explored
What do we need to describe a BFS algorithm?

- Know which vertex to explore next
- Know which vertices we should not look at again
- Identify three vertex states: unvisited, visited but not explored, explored
- Some graph operations are needed: getting the neighbors of a vertex, for example
What do we need to describe a BFS algorithm?

- Know which vertex to explore next
- Know which vertices we should not look at again
- Identify three vertex states: unvisited, visited but not explored, explored
- Some graph operations are needed: getting the neighbors of a vertex, for example
- What if $G$ is not connected?
BFS Algorithm: Version 1

Algorithm 1 Build Breadth-First Search Tree of $G$ from vertex $r$

```
procedure BFST($G$, $r$) $\triangleright G = (V, E)$
Mark all $v \in V$ as unvisited $\triangleright$ Initialization steps
Let $T$ be an empty graph
Add $r$ to $T$; mark $r$ as visited; $r$.level $\leftarrow 0$
while There are visited vertices do
  $current \leftarrow$ some visited vertex having minimum level
  Mark $current$ as explored
  for all unvisited neighbors $v$ of $current$ do
    Add $\{current, v\}$ to $T$
    Mark $v$ as visited
    $v$.level $\leftarrow current$.level + 1
  end for
end while
end procedure
```
Properties of BFST

- After \( r \) is added to \( T \), \( T \) remains a tree throughout the run of the algorithm.
- The vertices with level \( i \) are those of distance \( i \) from \( r \).
- Thus, \( T \) consists of all vertices reachable from \( r \), that is, \( T \) is a spanning tree of a component of \( G \).
- All edges of \( G \) not in \( T \) connect vertices at consecutive levels (or at the same level) of \( T \).
- BFST \((G, r)\) can be used to find all connected components of \( G \).
Properties of BFST

- After $r$ is added to $T$, $T$ remains a tree throughout run of algorithm
Properties of BFST

- After $r$ is added to $T$, $T$ remains a tree throughout run of algorithm
- The vertices with $level = i$ are those of distance $i$ from $r$
Properties of BFST

- After $r$ is added to $T$, $T$ remains a tree throughout run of algorithm
- The vertices with $level = i$ are those of distance $i$ from $r$
- Thus $T$ consists of all vertices reachable from $r$: that is, $T$ is a spanning tree of a component of $G$
Properties of BFST

- After $r$ is added to $T$, $T$ remains a tree throughout run of algorithm.
- The vertices with $\text{level} = i$ are those of distance $i$ from $r$.
- Thus $T$ consists of all vertices reachable from $r$: that is, $T$ is a spanning tree of a component of $G$.
- All edges of $G$ not in $T$ connect vertices at consecutive levels (or at the same level) of $T$. 
Properties of BFST

- After $r$ is added to $T$, $T$ remains a tree throughout run of algorithm.
- The vertices with level $i$ are those of distance $i$ from $r$.
- Thus $T$ consists of all vertices reachable from $r$: that is, $T$ is a spanning tree of a component of $G$.
- All edges of $G$ not in $T$ connect vertices at consecutive levels (or at the same level) of $T$.
- $BFST(G, r)$ can be used to find all connected components of $G$.
BFS Algorithm: Version 2

Algorithm 2 Breadth-First Search of $G$ from vertex $r$

procedure BFS($G, r$)  \> $G = (V, E)$

Mark all $v \in V$ as unvisited  \> Initialization steps
Mark $r$ as visited; $r$.level $\leftarrow 0$

while There are visited vertices do

    current $\leftarrow$ some visited vertex having minimum level
    Mark current as explored

    for all unvisited neighbors $v$ of current do

        Mark $v$ as visited; $v$.level $\leftarrow$ current.level + 1

    end for

end while

end procedure
BFS: Implementation and Complexity

Data Structure Requirements
BFS: Implementation and Complexity

Data Structure Requirements

- Assume \( V = \{1, \ldots, n\} \) indexing an array of vertex adjacency lists.
BFS: Implementation and Complexity

Data Structure Requirements

- Assume $V = \{1, \ldots, n\}$ indexing an array of vertex adjacency lists.
- So getting the "next neighbor" of a vertex can be done in constant time.
Data Structure Requirements

- Assume $V = \{1, \ldots, n\}$ indexing an array of vertex adjacency lists.
- So getting the "next neighbor" of a vertex can be done in constant time.
- $T$ can be stored as an edge list.
**BFS: Implementation and Complexity**

Data Structure Requirements

- Assume $V = \{1, \ldots, n\}$ indexing an array of vertex adjacency lists.
- So getting the "next neighbor" of a vertex can be done in constant time.
- $T$ can be stored as an edge list
- Each vertex/edge stores a label.
BFS: Implementation and Complexity

Data Structure Requirements

• Assume $V = \{1, \ldots, n\}$ indexing an array of vertex adjacency lists.

• So getting the "next neighbor" of a vertex can be done in constant time.

• $T$ can be stored as an edge list

• Each vertex/edge stores a label.

• Store a copy of each 'visited' vertex in a priority queue.
**BFS: Implementation and Complexity**

Data Structure Requirements

- Assume $V = \{1, \ldots, n\}$ indexing an array of vertex adjacency lists.
- So getting the "next neighbor" of a vertex can be done in constant time.
- $T$ can be stored as an edge list
- Each vertex/edge stores a label.
- Store a copy of each 'visited' vertex in a priority queue.
- For connected $G$ gives an $O(m + n)$ space and $O(m + n\log n)$ time algorithm, where $|V| = n$ and $|E| = m$
BFS: Implementation and Complexity

Data Structure Requirements

• Assume $V = \{1, \ldots, n\}$ indexing an array of vertex adjacency lists.
• So getting the "next neighbor" of a vertex can be done in constant time.
• $T$ can be stored as an edge list
• Each vertex/edge stores a label.
• Store a copy of each 'visited' vertex in a priority queue.
• For connected $G$ gives an $O(m + n)$ space and $O(m + n \log n)$ time algorithm, where $|V| = n$ and $|E| = m$
• Better: Use a queue instead of a priority queue. This reduces the run time to $O(m + n)$. 
Algorithm 3 Better Breadth-First Search of $G$ from vertex $r$

**procedure** BBFS($G, r$)

Mark all $v \in V$ and all $e \in E$ as unvisited
Initialize an empty queue $Q$
Mark $r$ as visited; $Q$.enqueue($r$)

while There are visited vertices do
  
  $current \leftarrow Q$.dequeue()

  for all neighbors $v$ of $current$ do
    
    if $v$ is unvisited then
      
      Mark $v$ as visited; $Q$.enqueue($v$)
    
    end if
   
    if $\{current, v\}$ is unvisited then
      
      Mark $\{current, v\}$ as visited
    
    end if
  
  end for

end while

end procedure
Properties of BBFS

For a connected graph $G$
Properties of BBFS

For a connected graph $G$

- $BBFS(G, r)$ visits every vertex and edge of $G$
Properties of BBFS

For a connected graph $G$

- $BBFS(G, r)$ visits every vertex and edge of $G$
- The queue $Q$ fulfills the role of the arrays $L[i]$ of levels in text.
Properties of BBFS

For a connected graph $G$

- $BBFS(G, r)$ visits every vertex and edge of $G$
- The queue $Q$ fulfills the role of the arrays $L[i]$ of levels in text.
- Runs in optimal $O(n + m)$ time and space
Properties of BBFS

For a connected graph $G$

- $BBFS(G, r)$ visits every vertex and edge of $G$
- The queue $Q$ fulfills the role of the arrays $L[i]$ of levels in text.
- Runs in optimal $O(n + m)$ time and space
- We can tweak $BBFS$ so that it
Properties of BBFS

For a connected graph $G$

- $BBFS(G, r)$ visits every vertex and edge of $G$
- The queue $Q$ fulfills the role of the arrays $L[i]$ of levels in text.
- Runs in optimal $O(n + m)$ time and space
- We can tweak $BBFS$ so that it
  - Assigns each vertex a label (level) equal to its distance from $r$
Properties of BBFS

For a connected graph $G$

- $BBFS(G, r)$ visits every vertex and edge of $G$
- The queue $Q$ fulfills the role of the arrays $L[i]$ of levels in text.
- Runs in optimal $O(n + m)$ time and space
- We can tweak $BBFS$ so that it
  - Assigns each vertex a label (level) equal to its distance from $r$
  - Labels each edge as a tree-edge or a non-tree-edge
Properties of BBFS

For a connected graph $G$

- $BBFS(G, r)$ visits every vertex and edge of $G$
- The queue $Q$ fulfills the role of the arrays $L[i]$ of levels in text.
- Runs in optimal $O(n + m)$ time and space
- We can tweak $BBFS$ so that it
  - Assigns each vertex a label (level) equal to its distance from $r$
  - Labels each edge as a tree-edge or a non-tree-edge
  - Constructs all of the connected components of a non-connected graph
Properties of BBFS

For a connected graph $G$

- $BBFS(G, r)$ visits every vertex and edge of $G$
- The queue $Q$ fulfills the role of the arrays $L[i]$ of levels in text.
- Runs in optimal $O(n + m)$ time and space
- We can tweak $BBFS$ so that it
  - Assigns each vertex a label (level) equal to its distance from $r$
  - Labels each edge as a tree-edge or a non-tree-edge
  - Constructs all of the connected components of a non-connected graph
  - Provides shortest paths from every vertex back to $r$
Application: Deciding Bipartiteness
Application: Deciding Bipartiteness

Definition

A bipartition of a set $X$ is a pair of subsets $X_1, X_2$ of $X$ such that

1. $X_1 \cup X_2 = X$, and
2. $X_1 \cap X_2 = \emptyset$
Application: Deciding Bipartiteness

Definition

A bipartition of a set $X$ is a pair of subsets $X_1, X_2$ of $X$ such that

1. $X_1 \cup X_2 = X$, and
2. $X_1 \cap X_2 = \emptyset$

A bipartition of $X$ is also called a partition of $X$ (into 2 parts) or a 2-coloring of $X$
Application: Deciding Bipartiteness

Definition
A bipartition of a set $X$ is a pair of subsets $X_1$, $X_2$ of $X$ such that

1. $X_1 \cup X_2 = X$, and
2. $X_1 \cap X_2 = \emptyset$

A bipartition of $X$ is also called a partition of $X$ (into 2 parts) or a 2-coloring of $X$

Definition
A graph $G = (V, E)$ is bipartite if $V$ can be partitioned into two sets $V_1$ and $V_2$ so that every edge $e \in E$ has a vertex in each of $V_1$ and $V_2$. 
Application: Deciding Bipartiteness

Definition
A bipartition of a set $X$ is a pair of subsets $X_1$, $X_2$ of $X$ such that

1. $X_1 \cup X_2 = X$, and
2. $X_1 \cap X_2 = \emptyset$

A bipartition of $X$ is also called a partition of $X$ (into 2 parts) or a 2-coloring of $X$

Definition
A graph $G = (V, E)$ is bipartite if $V$ can be partitioned into two sets $V_1$ and $V_2$ so that every edge $e \in E$ has a vertex in each of $V_1$ and $V_2$.

Bipartite graphs are also called 2-colorable graphs.
Application: Deciding Bipartiteness

Theorem

The following statements are equivalent for a connected graph $G$:

1. $G$ is bipartite
2. Every circuit in $G$ has even length
3. No BFS tree has edges between vertices at the same level
4. Some BFS tree has no edges between two vertices at the same level

Note: Conditions (a) and (b) seem hard to check directly; but conditions (c) and (d) allow an easy check!
Application: Deciding Bipartiteness

Theorem

The following statements are equivalent for a connected graph $G$
Application: Deciding Bipartiteness

Theorem

The following statements are equivalent for a connected graph $G$

(a) $G$ is bipartite
Application: Deciding Bipartiteness

Theorem
The following statements are equivalent for a connected graph $G$

(a) $G$ is bipartite
(b) Every circuit in $G$ has even length
**Application: Deciding Bipartiteness**

*Theorem*

The following statements are equivalent for a connected graph $G$

(a) $G$ is bipartite

(b) Every circuit in $G$ has even length

(c) No BFS tree has edges between vertices at same level
Application: Deciding Bipartiteness

Theorem
The following statements are equivalent for a connected graph $G$

(a) $G$ is bipartite
(b) Every circuit in $G$ has even length
(c) No BFS tree has edges between vertices at same level
(d) Some BFS tree has no edges between two vertices at the same level
**Application: Deciding Bipartiteness**

**Theorem**

The following statements are equivalent for a connected graph $G$

(a) $G$ is bipartite

(b) Every circuit in $G$ has even length

(c) No BFS tree has edges between vertices at same level

(d) Some BFS tree has no edges between two vertices at the same level

Note: Conditions (a) and (b) seem hard to check directly; but conditions (c) and (d) allow an easy check!
Application: Deciding Bipartiteness

Theorem
The following statements are equivalent for a connected graph $G$

(a) $G$ is bipartite
(b) Every circuit in $G$ has even length
(c) No BFS tree has edges between vertices at same level
(d) Some BFS tree has no edges between two vertices at the same level
Application: Deciding Bipartiteness

Theorem
The following statements are equivalent for a connected graph $G$

(a) $G$ is bipartite
(b) Every circuit in $G$ has even length
(c) No BFS tree has edges between vertices at same level
(d) Some BFS tree has no edges between two vertices at the same level

Proof.
Application: Deciding Bipartiteness

Theorem
The following statements are equivalent for a connected graph $G$

(a) $G$ is bipartite
(b) Every circuit in $G$ has even length
(c) No BFS tree has edges between vertices at same level
(d) Some BFS tree has no edges between two vertices at the same level

Proof.

(a) $\implies$ (b) Vertices in circuit must alternate between $V_1$ and $V_2$. 
Application: Deciding Bipartiteness

Theorem
The following statements are equivalent for a connected graph $G$

(a) $G$ is bipartite
(b) Every circuit in $G$ has even length
(c) No BFS tree has edges between vertices at same level
(d) Some BFS tree has no edges between two vertices at the same level

Proof.

(a) $\implies$ (b) Vertices in circuit must alternate between $V_1$ and $V_2$.
(b) $\implies$ (c) Contradiction: Such an edge implies an odd circuit.
Application: Deciding Bipartiteness

Theorem
The following statements are equivalent for a connected graph $G$

(a) $G$ is bipartite
(b) Every circuit in $G$ has even length
(c) No BFS tree has edges between vertices at same level
(d) Some BFS tree has no edges between two vertices at the same level

Proof.

(a) $\implies$ (b) Vertices in circuit must alternate between $V_1$ and $V_2$.
(b) $\implies$ (c) Contradiction: Such an edge implies an odd circuit.
(c) $\implies$ (d) A rare, justified use of the term “obvious".
Application: Deciding Bipartiteness

Theorem
The following statements are equivalent for a connected graph $G$

(a) $G$ is bipartite
(b) Every circuit in $G$ has even length
(c) No BFS tree has edges between vertices at same level
(d) Some BFS tree has no edges between two vertices at the same level

Proof.

(a) $\implies$ (b) Vertices in circuit must alternate between $V_1$ and $V_2$.
(b) $\implies$ (c) Contradiction: Such an edge implies an odd circuit.
(c) $\implies$ (d) A rare, justified use of the term “obvious”.
(d) $\implies$ (a) Edges must span consecutive levels: levels provide bipartition of $G$. 
Implications of the Theorem

\( G \) is bipartite iff no BFS tree for \( G \) has two vertices at the same level that form an edge in \( G \).

- When the BBFS algorithm visits an edge, we know the level of both of its endpoints.
- So when that edge is visited, if both ends have the same level, then STOP! \( G \) is not bipartite.
- If the algorithm never discovers such an edge, \( G \) is bipartite.
- This modified BFS still runs in \( O(n + m) \) time.
- \( G \) not connected? Run on each component: \( O(|V| + |E|) \) time.
- Moreover, if \( G \) is not bipartite, we can produce an odd circuit in \( G \) as proof [Admire the awesomeness!]

Principle: Prefer algorithms that provide certificate of correctness!
Implications of the Theorem

So $G$ is bipartite iff no BFS tree for $G$ has two vertices at the same level that form an edge in $G$. 
Implications of the Theorem

So $G$ is bipartite iff no BFS tree for $G$ has two vertices at the same level that form an edge in $G$.

- When the BBFS algorithm visits an edge, we know the level of both of its endpoints.
Implications of the Theorem

So $G$ is bipartite iff no BFS tree for $G$ has two vertices at the same level that form an edge in $G$.

- When the BBFS algorithm visits an edge, we know the level of both of its endpoints.
- So when that edge is visited, if both ends have the same level, then STOP! $G$ is not bipartite.
Implications of the Theorem

So $G$ is bipartite iff no BFS tree for $G$ has two vertices at the same level that form an edge in $G$.

- When the BBFS algorithm visits an edge, we know the level of both of its endpoints.
- So when that edge is visited, if both ends have the same level, then STOP! $G$ is not bipartite.
- If the algorithm never discovers such an edge, $G$ is bipartite.
Implications of the Theorem

So $G$ is bipartite iff no BFS tree for $G$ has two vertices at the same level that form an edge in $G$.

- When the BBFS algorithm visits an edge, we know the level of both of its endpoints.
- So when that edge is visited, if both ends have the same level, then STOP! $G$ is not bipartite.
- If the algorithm never discovers such an edge, $G$ is bipartite.
- This modified $BFS$ still runs in $O(n + m)$ time.
Implications of the Theorem

So $G$ is bipartite iff no BFS tree for $G$ has two vertices at the same level that form an edge in $G$.

- When the BBFS algorithm visits an edge, we know the level of both of its endpoints.
- So when that edge is visited, if both ends have the same level, then STOP! $G$ is not bipartite.
- If the algorithm never discovers such an edge, $G$ is bipartite.
- This modified BFS still runs in $O(n + m)$ time.
- $G$ not connected? Run on each component: $O(|V| + E)$ time.
Implications of the Theorem

So $G$ is bipartite iff no BFS tree for $G$ has two vertices at the same level that form an edge in $G$.

- When the BBFS algorithm visits an edge, we know the level of both of its endpoints.
- So when that edge is visited, if both ends have the same level, then STOP! $G$ is not bipartite.
- If the algorithm never discovers such an edge, $G$ is bipartite.
- This modified BFS still runs in $O(n + m)$ time.
- $G$ not connected? Run on each component: $O(|V| + E|$) time.
- Moreover, if $G$ is not bipartite, we can produce an odd circuit in $G$ as proof [Admire the awesomeness!]
Implications of the Theorem

So $G$ is bipartite iff no BFS tree for $G$ has two vertices at the same level that form an edge in $G$.

- When the BBFS algorithm visits an edge, we know the level of both of its endpoints.
- So when that edge is visited, if both ends have the same level, then STOP! $G$ is not bipartite.
- If the algorithm never discovers such an edge, $G$ is bipartite.
- This modified BFS still runs in $O(n + m)$ time.
- $G$ not connected? Run on each component: $O(|V| + E)$ time
- Moreover, if $G$ is not bipartite, we can produce an odd circuit in $G$ as proof [Admire the awesomeness!]

Principle: Prefer algorithms that provide certificate of correctness!
DFS: An Example

Two types of vertex: unvisited, visited. Green is just for emphasis!
**DFS: An Example**

Two types of vertex: *unvisited, visited*. *Green* is just for emphasis!
DFS: An Example

Two types of vertex: unvisited, visited. Green is just for emphasis!
DFS: An Example

Two types of vertex: unvisited, visited. Green is just for emphasis!
DFS: An Example

Two types of vertex: unvisited, visited. Green is just for emphasis!
DFS: An Example

Two types of vertex: unvisited, visited. Green is just for emphasis!
DFS: An Example

Two types of vertex: unvisited, visited. Green is just for emphasis!
DFS: An Example

Two types of vertex: unvisited, visited. Green is just for emphasis!
Recursive Depth-First Search

Algorithm 4 Depth-First Search of $G$ from vertex $r$

**Require:** all vertices are *unvisited*; $T = \{r\}$ is a 1-vertex tree

procedure DFS($G$, $r$, $T$) \quad $\triangledown G = (V, E)$

Mark $r$ as *visited*

for all neighbors $v$ of $r$ do

if $v$ is unvisited then

Add $\{r, v\}$ to $T$

DFS($G$, $v$, $T$)

end if

end for

end procedure

**Ensure:** $T$ is a spanning tree for the component of $G$ containing $r$
Properties of DFS
Properties of DFS

- When algorithm terminates, $T$ forms a spanning tree of the component of $G$ containing $r$. 
**Properties of DFS**

- When algorithm terminates, $T$ forms a spanning tree of the component of $G$ containing $r$.
- $T$ is a tree because (i) it is connected and (ii) it has one more vertex than edge (see theorem from text)
Properties of DFS

- When algorithm terminates, $T$ forms a spanning tree of the component of $G$ containing $r$.
  - $T$ is a tree because (i) it is connected and (ii) it has one more vertex than edge (see theorem from text)
  - $T$ contains every vertex reachable from $r$
Properties of DFS

- When algorithm terminates, $T$ forms a spanning tree of the component of $G$ containing $r$.
  - $T$ is a tree because (i) it is connected and (ii) it has one more vertex than edge (see theorem from text)
  - $T$ contains every vertex reachable from $r$
    - Induction on distance of reachable vertex from $r$
Properties of DFS

- When algorithm terminates, $T$ forms a spanning tree of the component of $G$ containing $r$.
  - $T$ is a tree because (i) it is connected and (ii) it has one more vertex than edge (see theorem from text)
  - $T$ contains every vertex reachable from $r$
    - Induction on distance of reachable vertex from $r$
    - If $v$ is visited, so are its neighbors
Properties of DFS

• When algorithm terminates, $T$ forms a spanning tree of the component of $G$ containing $r$.
  • $T$ is a tree because (i) it is connected and (ii) it has one more vertex than edge (see theorem from text)
  • $T$ contains every vertex reachable from $r$
    • Induction on distance of reachable vertex from $r$
    • If $v$ is visited, so are its neighbors
• Now consider $T$ as a rooted tree with root $r$
Properties of DFS

- When algorithm terminates, $T$ forms a spanning tree of the component of $G$ containing $r$.
  - $T$ is a tree because (i) it is connected and (ii) it has one more vertex than edge (see theorem from text)
  - $T$ contains every vertex reachable from $r$
    - Induction on distance of reachable vertex from $r$
    - If $v$ is visited, so are its neighbors
- Now consider $T$ as a rooted tree with root $r$
  - Every vertex visited during a call to $DFS(G, v)$ is a descendent of $v$ in $T$
Properties of DFS

- When algorithm terminates, $T$ forms a spanning tree of the component of $G$ containing $r$.
  - $T$ is a tree because (i) it is connected and (ii) it has one more vertex than edge (see theorem from text)
  - $T$ contains every vertex reachable from $r$
    - Induction on distance of reachable vertex from $r$
    - If $v$ is visited, so are its neighbors
- Now consider $T$ as a rooted tree with root $r$
  - Every vertex visited during a call to $DFS(G, v)$ is a descendent of $v$ in $T$
    - We consider any vertex to be (trivially) a descendent of itself
Properties of DFS

- When algorithm terminates, \( T \) forms a spanning tree of the component of \( G \) containing \( r \).
  - \( T \) is a tree because (i) it is connected and (ii) it has one more vertex than edge (see theorem from text)
  - \( T \) contains every vertex reachable from \( r \)
    - Induction on distance of reachable vertex from \( r \)
    - If \( v \) is visited, so are its neighbors
- Now consider \( T \) as a rooted tree with root \( r \)
  - Every vertex visited during a call to \( DFS(G, v) \) is a descendent of \( v \) in \( T \)
    - We consider any vertex to be (trivially) a descendent of itself
  - For every edge \( e = \{u, v\} \) in \( G \), one of \( u \) or \( v \) is an ancestor of the other in \( T \).
The proof

For every edge $e = \{u, v\}$ in $G$, one of $u$ or $v$ is an ancestor of the other in $T$.

Proof.
The proof

For every edge $e = \{u, v\}$ in $G$, one of $u$ or $v$ is an ancestor of the other in $T$.

Proof.

• Clear if $e$ is in $T$, so assume not.
The proof

For every edge $e = \{u, v\}$ in $G$, one of $u$ or $v$ is an ancestor of the other in $T$.

Proof.

- Clear if $e$ is in $T$, so assume not.
- Assume DFS is called on $u$ before $v$. When the For loop inspected $v$, $v$ must have been already visited.
The proof

For every edge $e = \{u, v\}$ in $G$, one of $u$ or $v$ is an ancestor of the other in $T$.

Proof.

- Clear if $e$ is in $T$, so assume not.
- Assume DFS is called on $u$ before $v$. When the For loop inspected $v$, $v$ must have been already visited.
  - Or else $v$ becomes a descendent of $u$
The proof

For every edge $e = \{u, v\}$ in $G$, one of $u$ or $v$ is an ancestor of the other in $T$.

Proof.

• Clear if $e$ is in $T$, so assume not.
• Assume DFS is called on $u$ before $v$. When the For loop inspected $v$, $v$ must have been already visited.
  • Or else $v$ becomes a descendent of $u$
• But $v$ wasn’t visited when DFS was called on $u$. 
The proof

For every edge \( e = \{u, v\} \) in \( G \), one of \( u \) or \( v \) is an ancestor of the other in \( T \).

Proof.

- Clear if \( e \) is in \( T \), so assume not.
- Assume DFS is called on \( u \) before \( v \). When the For loop inspected \( v \), \( v \) must have been already visited.
  - Or else \( v \) becomes a descendent of \( u \)
- But \( v \) wasn’t visited when DFS was called on \( u \).
- Thus \( v \) was visited during the call \( DFS(G, u) \) and so it’s a descendent of \( u \).
Definition

An directed graph $G = (V, E)$ consists of two sets
Directed Graphs

**Definition**

An *directed graph* $G = (V, E)$ consists of two sets

- A set $V$ called the *vertices* of $G$
Directed Graphs

Definition

An *directed graph* $G = (V, E)$ consists of two sets

- A set $V$ called the *vertices* of $G$
- A set $E$ of ordered pairs of *distinct* vertices of $V$ called the *edges* of $G$
Directed Graphs

Definition

An directed graph $G = (V, E)$ consists of two sets

- A set $V$ called the vertices of $G$
- A set $E$ of ordered pairs of distinct vertices of $V$ called the edges of $G$

Note: No loops or multiple edges. Why?
Directed Graphs

Definition

An directed graph $G = (V, E)$ consists of two sets

- A set $V$ called the vertices of $G$
- A set $E$ of ordered pairs of distinct vertices of $V$ called the edges of $G$

Note: No loops or multiple edges. Why?

Properties of undirected graphs have counterparts in directed graphs, with some differences.
Directed Graphs

Definition
An directed graph $G = (V, E)$ consists of two sets
- A set $V$ called the vertices of $G$
- A set $E$ of ordered pairs of distinct vertices of $V$ called the edges of $G$

Note: No loops or multiple edges. Why?

Properties of undirected graphs have counterparts in directed graphs, with some differences.
Example: A directed walk in $G$ is a sequence $P = u = v_0, e_0, v_1, \ldots, e_n, v_n = v$ in which each $e_i = (v_{i-1}, v_i)$
Directed Graphs

Definition

An directed graph $G = (V, E)$ consists of two sets

- A set $V$ called the vertices of $G$
- A set $E$ of ordered pairs of distinct vertices of $V$ called the edges of $G$

Note: No loops or multiple edges. Why?

Properties of undirected graphs have counterparts in directed graphs, with some differences.

Example: A directed walk in $G$ is a sequence $P = u = v_0, e_0, v_1, \ldots, e_n, v_n = v$ in which each $e_i = (v_{i-1}, v_i)$

Now $v$ is reachable from $u$ if there is a directed path from $u$ to $v$
Reachability in Directed Graphs: An Example
Reachability in Directed Graphs: An Example

BFS and DFS both work on directed graphs
Reachability in Directed Graphs: An Example

BFS and DFS both work on directed graphs
Both visit exactly the nodes reachable from the start vertex
Reachability in Directed Graphs: An Example

BFS and DFS both work on directed graphs
Both visit exactly the nodes reachable from the start vertex
Reachability in Directed Graphs: An Example

BFS and DFS both work on directed graphs
Both visit exactly the nodes reachable from the start vertex