3.4 Testing Bipartiteness
Bipartite Graphs

Def. An undirected graph $G = (V, E)$ is **bipartite (2-colorable)** if the nodes can be colored red or blue such that no edge has both ends the same color.

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

"bi-partite" means "two parts." An equivalent definition: $G$ is bipartite if you can partition the node set into 2 parts (say, blue/red or left/right) so that all edges join nodes in different parts/no edge has both ends in the same part.

*a bipartite graph*
Testing Bipartiteness

Testing bipartiteness. Given a graph $G$, is it bipartite?

Many graph problems become:
- easier if the underlying graph is bipartite (matching)
- tractable if the underlying graph is bipartite (independent set)

Before attempting to design an algorithm, we need to understand structure of bipartite graphs.

*a bipartite graph $G$*

*another drawing of $G$*
Lemma. If a graph $G$ is bipartite, it cannot contain an odd length cycle.

Pf. Impossible to 2-color the odd cycle, let alone $G$. 
Lemma. Let $G$ be a connected graph, and let $L_0$, …, $L_k$ be the layers produced by BFS starting at node $s$. Exactly one of the following holds.

(i) No edge of $G$ joins two nodes of the same layer, and $G$ is bipartite.
(ii) An edge of $G$ joins two nodes of the same layer, and $G$ contains an odd-length cycle (and hence is not bipartite).
Lemma. Let $G$ be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node $s$. Exactly one of the following holds.

(i) No edge of $G$ joins two nodes of the same layer, and $G$ is bipartite.
(ii) An edge of $G$ joins two nodes of the same layer, and $G$ contains an odd-length cycle (and hence is not bipartite).

Pf. (i) Suppose no edge joins two nodes in the same layer. By previous lemma, all edges join nodes on adjacent levels.

Bipartition:
- red = nodes on odd levels,
- blue = nodes on even levels.

Case (i)
Bipartite Graphs

Lemma. Let $G$ be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node $s$. Exactly one of the following holds.

(i) No edge of $G$ joins two nodes of the same layer, and $G$ is bipartite.
(ii) An edge of $G$ joins two nodes of the same layer, and $G$ contains an odd-length cycle (and hence is not bipartite).

Pf. (ii)
Suppose $(x, y)$ is an edge & $x, y$ in same level $L_j$.
Let $z =$ their lowest common ancestor in BFS tree.
Let $L_i$ be level containing $z$.
Consider cycle that takes edge from $x$ to $y$, then tree from $y$ to $z$, then tree from $z$ to $x$.
Its length is $1 + (j-i) + (j-i)$, which is odd.
Obstruction to Bipartiteness

Cor: A graph $G$ is bipartite iff it contains no odd length cycle.

NB: the proof is algorithmic—it finds a coloring or odd cycle.

\begin{itemize}
  \item[bipartite] (2-colorable)
  \item[not bipartite] (not 2-colorable)
\end{itemize}
BFS(s) Implementation

Global initialization: mark all vertices "undiscovered"

BFS(s)
  mark s "discovered"
  queue = { s }
  while queue not empty
    u = remove_first(queue)
    for each edge {u,x}
      if (x is undiscovered)
        mark x discovered
        append x on queue
    mark u fully explored

Exercise: modify code to determine if the graph is bipartite
3.6 DAGs and Topological Ordering
Precedence Constraints

Precedence constraints. Edge \((v_i, v_j)\) means task \(v_i\) must occur before \(v_j\).

Applications

Course prerequisites: course \(v_i\) must be taken before \(v_j\)

Compilation: must compile module \(v_i\) before \(v_j\)

Computing workflow: output of job \(v_i\) is input to job \(v_j\)

Manufacturing or assembly: sand it before you paint it…

Spreadsheet evaluation order: if A7 is "=A6+A5+A4", evaluate them first
Directed Acyclic Graphs

Def. A **DAG** is a directed acyclic graph, i.e., one that contains no directed cycles.

Ex. Precedence constraints: edge \((v_i, v_j)\) means \(v_i\) must precede \(v_j\).

Def. A **topological order** of a directed graph \(G = (V, E)\) is an ordering of its nodes as \(v_1, v_2, \ldots, v_n\) so that for every edge \((v_i, v_j)\) we have \(i < j\).

E.g., \(\forall\) edge \((v_i, v_j)\), finish \(v_i\) before starting \(v_j\)

\[\text{a DAG} \quad \text{a topological ordering of that DAG—} \] \(\text{all edges left-to-right}\)
Directed Acyclic Graphs

Lemma. If $G$ has a topological order, then $G$ is a DAG.
Directed Acyclic Graphs

Lemma. If $G$ has a topological order, then $G$ is a DAG.

Pf. (by contradiction)

Suppose that $G$ has a topological order $v_1, \ldots, v_n$ and that $G$ also has a directed cycle $C$.

Let $v_i$ be the lowest-indexed node in $C$, and let $v_j$ be the node just before $v_i$; thus $(v_j, v_i)$ is an edge.

By our choice of $i$, we have $i < j$.

On the other hand, since $(v_j, v_i)$ is an edge and $v_1, \ldots, v_n$ is a topological order, we must have $j < i$, a contradiction.
Directed Acyclic Graphs

Lemma.
If G has a topological order, then G is a DAG.

Q. Does every DAG have a topological ordering?

Q. If so, how do we compute one?
Directed Acyclic Graphs

Lemma. If G is a DAG, then G has a node with no incoming edges.

Pf. (by contradiction)
Suppose that G is a DAG and every node has at least one incoming edge. Let's see what happens.
Pick any node v, and begin following edges backward from v. Since v has at least one incoming edge (u, v) we can walk backward to u. Then, since u has at least one incoming edge (x, u), we can walk backward to x.
Repeat until we visit a node, say w, twice. Let C be the sequence of nodes encountered between successive visits to w. C is a cycle.

Why must this happen?
Directed Acyclic Graphs

Lemma. If $G$ is a DAG, then $G$ has a topological ordering.

Pf Algorithm?
Directed Acyclic Graphs

Lemma. If $G$ is a DAG, then $G$ has a topological ordering.

Pf. (by induction on $n$)
Base case: true if $n = 1$.
Given DAG on $n > 1$ nodes, find a node $v$ with no incoming edges.
$G - \{v\}$ is a DAG, since deleting $v$ cannot create cycles.
By inductive hypothesis, $G - \{v\}$ has a topological ordering.
Place $v$ first in topological ordering; then append nodes of $G - \{v\}$
in topological order. This is valid since $v$ has no incoming edges. $
\Box$

To compute a topological ordering of $G$:
Find a node $v$ with no incoming edges and order it first
Delete $v$ from $G$
Recursively compute a topological ordering of $G-\{v\}$
and append this order after $v$
Topological Ordering Algorithm: Example

Topological order:

\[ \text{V}_7, \text{V}_6, \text{V}_5, \text{V}_4, \text{V}_3, \text{V}_2, \text{V}_1 \]
Topological Ordering Algorithm: Example

Topological order: $v_1$
Topological Ordering Algorithm: Example

Topological order: $v_1, v_2$
Topological Ordering Algorithm: Example

Topological order: $v_1, v_2, v_3$
Topological Ordering Algorithm: Example

Topological order: $v_1, v_2, v_3, v_4$
Topological Ordering Algorithm: Example

Topological order: $v_1, v_2, v_3, v_4, v_5$
Topological Ordering Algorithm: Example

Topological order: $v_1, v_2, v_3, v_4, v_5, v_6$
Topological Order: \(v_1, v_2, v_3, v_4, v_5, v_6, v_7\).
Topological Sorting Algorithm

Linear time implementation?
**Topological Sorting Algorithm**

Maintain the following:
- \(\text{count}[w] = \) (remaining) number of incoming edges to node \(w\)
- \(S = \) set of (remaining) nodes with no incoming edges

**Initialization:**
- \(\text{count}[w] = 0\) for all \(w\)
- \(\text{count}[w]++\) for all edges \((v,w)\)
- \(S = S \cup \{w\}\) for all \(w\) with \(\text{count}[w] == 0\)

**Main loop:**
- while \(S\) not empty
  - remove some \(v\) from \(S\)
  - make \(v\) next in topo order
  - for all edges from \(v\) to some \(w\)
    - decrement \(\text{count}[w]\)
    - add \(w\) to \(S\) if \(\text{count}[w]\) hits 0

**Correctness:** clear, I hope

**Time:** \(O(m + n)\) (assuming edge-list representation of graph)
Depth-First Search

Follow the first path you find as far as you can go
Back up to last unexplored edge when you reach a dead end, then go as far you can

Naturally implemented using recursive calls or a stack
DFS(v) – Recursive version

Global Initialization:

for all nodes v, v.dfs# = -1  // mark v "undiscovered"
dfscounter = 0  // (global variable)
DFS(s);  // start DFS at node s;

DFS(v)

v.dfs# = dfscounter++  // v "discovered", number it
for each edge (v,x)

if (x.dfs# = -1)  // tree edge (x previously undiscovered)
    DFS(x)
Why fuss about trees (again)?

BFS tree $\neq$ DFS tree, but, as with BFS, DFS has found a tree in the graph s.t. non-tree edges are "simple" – only descendant/ancestor
Suppose edge lists at each vertex are sorted alphabetically.

### DFS(A)

Call Stack (Edge list):

A (B,J)

Color code:
- **undiscovered**
- **discovered**
- **fully-explored**
Suppose edge lists at each vertex are sorted alphabetically.

DFS(A)

Color code:
- undiscovered
- discovered
- fully-explored

Call Stack: (Edge list)
- A (B,J)
- B (A,C,J)

D
E

A,1

B,2

C

G

F

H

I

J

K

L

M
Suppose edge lists at each vertex are sorted alphabetically.

**DFS(A)**

Color code:
- undiscovered
- discovered
- fully-explored

Call Stack:
- (Edge list)
- A (B,J)
- B (A,C,J)
- C (B,D,G,H)

Diagram:
- A
- B
- C
- D
- E
- F
- G
- H
- I
- J
- K
- L
- M
Suppose edge lists at each vertex are sorted alphabetically.

**DFS(A)**

Call Stack:
- (Edge list)
  - A (B, J)
  - B (A, C, J)
  - C (B, D, G, H)
  - D (C, E, F)

Color code:
- **undiscovered**
- **discovered**
- **fully-explored**
Suppose edge lists at each vertex are sorted alphabetically.

Call Stack:
(Edge list)
A (B, J)
B (A, C, J)
C (B, D, G, H)
D (C, E, F)
E (D, F)

Color code:
 undiscovered
 discovered
 fully-explored

DFS(A)
Suppose edge lists at each vertex are sorted alphabetically.

**DFS(A)**

Color code:
- **undiscovered**
- **discovered**
- **fully-explored**

Call Stack:
- (Edge list)
  - A (B,J)
  - B (A,C,J)
  - C (B,D,G,H)
  - D (C,E,F)
  - E (D,F)
  - F (D,E,G)
Suppose edge lists at each vertex are sorted alphabetically.

**DFS(A)**

- Edge lists are sorted alphabetically.

- **Call Stack:**
  - (Edge list)
  - A (B, J)
  - B (A, C, J)
  - C (B, D, G, H)
  - D (C, F, F)
  - E (D, F)
  - F (B, E, G)
  - G (C, F)

- **Color code:**
  - undiscovered
  - discovered
  - fully-explored
Suppose edge lists at each vertex are sorted alphabetically.

**DFS(A)**

- **A**, 1
- **B**, 2
- **C**, 3
- **D**, 4
- **E**, 5
- **F**, 6
- **G**, 7

Color code:
- **undiscovered**
- **discovered**
- **fully-explored**

Call Stack:
- (Edge list)
- A (B, J)
- B (A, C, J)
- C (B, D, G, H)
- D (C, E, F)
- E (D, F)
- F (D, E, G)
- G (C, F)

Diagram:
- A connected to B, J
- B connected to A, C, J
- C connected to B, D, G, H
- D connected to C, E, F
- E connected to D, F
- F connected to D, E, G
- G connected to C, F

M

K

L

J

H

I

E, 5

D, 4

G, 7

C, 3

B, 2

A, 1

95
Suppose edge lists at each vertex are sorted alphabetically.

Color code:
- undiscovered
- discovered
- fully-explored

Call Stack:
- (Edge list)
  - A (B,J)
  - B (A,C,J)
  - C (B,D,G,H)
  - D (C,F,F)
  - E (D,F)
  - F (D,E,G)
Suppose edge lists at each vertex are sorted alphabetically.

**DFS(A)**

Call Stack: (Edge list)
- A (B, J)
- B (A, C, J)
- C (B, D, G, H)
- D (C, E, F)
- E (D, F)

Color code:
- undiscovered
- discovered
- fully-explored
Suppose edge lists at each vertex are sorted alphabetically.

Call Stack: (Edge list)
- A (B, J)
- B (A, C, J)
- C (B, D, G, H)
- D (C, F, F)

DFS(A)

Color code:
- undiscovered
- discovered
- fully-explored
Suppose edge lists at each vertex are sorted alphabetically.

**DFS(A)**

Call Stack: (Edge list)
- A (B,J)
- B (A,C,J)
- C (B,D,G,H)

Color code:
- undiscovered
- discovered
- fully-explored
Suppose edge lists at each vertex are sorted alphabetically.

DFS(A)

Color code:
- undiscovered
- discovered
- fully-explored

Call Stack:
- (Edge list)
- A (B, J)
- B (A, C, J)
- C (B, D, G, H)

Diagram:
- A (1) connected to B (2) and J
- B (2) connected to A (1), C (3), and J
- C (3) connected to B (2), D (4), G (7), and H
- D (4) connected to E (5) and F (6)
- E (5) connected to D (4)
- F (6) connected to G (7)
- G (7) connected to F (6) and H
- H connected to G (7) and I
- K connected to J
- L connected to J
- M connected to L
Suppose edge lists at each vertex are sorted alphabetically.

Color code:
- undiscovered
- discovered
- fully-explored

Call Stack:
- (Edge list)
  - A (B,J)
  - B (A,C,J)
  - C (B,D,G,H)
  - H (C,I,J)

DFS(A)
Suppose edge lists at each vertex are sorted alphabetically.
Suppose edge lists at each vertex are sorted alphabetically.
**DFS(A)**

Suppose edge lists at each vertex are sorted alphabetically.

Color code:
- **undiscovered**
- **discovered**
- **fully-explored**

Call Stack:
(Edge list)

A \((B, J)\)
B \((A, C, J)\)
C \((B, D, G, H)\)
H \((C, I, J)\)

Suppose edge lists at each vertex are sorted alphabetically.
Suppose edge lists at each vertex are sorted alphabetically.

**DFS(A)**

Color code:
- **undiscovered**
- **discovered**
- **fully-explored**

Call Stack:
- (Edge list)
  - A (B, J)
  - B (A, C, J)
  - C (B, D, G, H)
  - H (C, I, J)
  - J (A, B, H, K, L)

105
**DFS(A)**

Suppose edge lists at each vertex are sorted alphabetically.

Color code:
- **undiscovered**
- **discovered**
- **fully-explored**

Call Stack:
- (Edge list)
  - A (B,J)
  - B (A,C,J)
  - C (B,D,G,H)
  - H (C,I,J)
  - J (A,B,H,K,L)
  - K (J,L)
  - L
  - M
Suppose edge lists at each vertex are sorted alphabetically.

DFS(A)

Call Stack: (Edge list)
- A (B,J)
- B (A,C,J)
- C (B,D,G,H)
- H (C,J,I)
- J (A,B,H,K,L)
- K (J,L)
- L (J,K,M)
DFS(A)

Suppose edge lists at each vertex are sorted alphabetically.

Color code:
- undiscovered
- discovered
- fully-explored

Call Stack:
(Edge list)
- A (B, J)
- B (A, C, J)
- C (B, D, G, H)
- H (C, J, I)
- J (A, B, H, K, L)
- K (J, L)
- L (J, K, M)
- M (L)
Suppose edge lists at each vertex are sorted alphabetically.

**DFS(A)**

Color code:
- **undiscovered**
- **discovered**
- **fully-explored**

Call Stack:
- (Edge list)
  - A (B,J)
  - B (A,C,J)
  - C (B,D,G,H)
  - H (C,I,J)
  - J (A,B,H,K,L)
  - K (J,L)
  - L (J,K,M)

Diagram:
- A,1
- B,2
- C,3
- D,4
- E,5
- F,6
- G,7
- H,8
- I,9
- J,10
- K,11
- L,12
- M,13
Suppose edge lists at each vertex are sorted alphabetically.
Suppose edge lists at each vertex are sorted alphabetically.

Color code:
- undiscovered
- discovered
- fully-explored

Call Stack:
- (Edge list)
  - A (B, J)
  - B (A, C, J)
  - C (B, D, G, H)
  - H (C, I, J)
  - J (A, B, H, K, L)
Suppose edge lists at each vertex are sorted alphabetically.

**DFS(A)**

Color code:
- undiscovered
- discovered
- fully-explored

Call Stack:
- (Edge list)
  - A (B, J)
  - B (A, C, J)
  - C (B, D, G, H)
  - H (C, I, J)
  - J (A, B, H, K, L)

![Graph showing DFS traversal with edge lists sorted alphabetically.](image-url)
DFS(A)

Suppose edge lists at each vertex are sorted alphabetically.
DFS(A)

Suppose edge lists at each vertex are sorted alphabetically.

Color code:
- undiscovered
- discovered
- fully-explored

Call Stack:
(Edge list)

A (B, J)
B (A, C, J)
C (B, D, G, H)

E, 5
D, 4
G, 7
F, 6
H, 8
I, 9
K, 11
L, 12
M, 13
J, 10
B, 2
C, 3
A, 1

114
Suppose edge lists at each vertex are sorted alphabetically.

**DFS(A)**

- **Color code:**
  - undiscovered
  - discovered
  - fully-explored

**Call Stack:**
- (Edge list)
- A (B,J)
- B (A,C,J)

The diagram shows a directed graph with vertices labeled with letters and numbers, and edges connecting them. The edges are sorted alphabetically, and the vertices are colored to indicate their exploration status.
Suppose edge lists at each vertex are sorted alphabetically.

DFS(A)

Color code:
- undiscovered
- discovered
- fully-explored

Call Stack:
- (Edge list)
  - A (B,J)
  - B (A,C,J)

A,1
B,2
J,10
C,3
G,7
H,8
K,11
L,12
D,4
F,6
I,9
M,13
E,5
**DFS(A)**

Suppose edge lists at each vertex are sorted alphabetically.

Color code:
- **undiscovered**
- **discovered**
- **fully-explored**

**Call Stack:**
(Edge list)

A (B, J)

Diagram:
- A, 1
- B, 2
- J, 10
- C, 3
- G, 7
- H, 8
- K, 11
- L, 12
- M, 13
- D, 4
- F, 6
- I, 9
- E, 5
Suppose edge lists at each vertex are sorted alphabetically.

**DFS(A)**

Color code:
- undiscovered
- discovered
- fully-explored

Call Stack:
(Edge list)
A (B, J)

Graph:
- A,1 (Root)
- B,2
  - C,3
  - G,7
  - D,4
  - E,5
- J,10
  - K,11
  - L,12
  - M,13
- H,8
- I,9
Suppose edge lists at each vertex are sorted alphabetically.
DFS(A)

Edge code:
Tree edge
Back edge
DFS(A)
DFS(A)

Edge code:
- Tree edge
- Back edge
DFS(A)

Edge code:
Tree edge
Back edge
DFS(A)

Edge code:
- Tree edge
- Back edge
- No Cross Edges!
Properties of (Undirected) DFS(v)

Like BFS(v):
- DFS(v) visits x if and only if there is a path in G from v to x (through previously unvisited vertices)
- Edges into then-undiscovered vertices define a tree – the "depth first spanning tree" of G

Unlike the BFS tree:
- the DF spanning tree isn't minimum depth
- its levels don't reflect min distance from the root
- non-tree edges never join vertices on the same or adjacent levels

BUT…
Non-tree edges

All non-tree edges join a vertex and one of its descendents/ancestors in the DFS tree (undirected graphs)

No cross edges!
Why fuss about trees (again)?

As with BFS, DFS has found a tree in the graph s.t. non-tree edges are "simple"--only descendant/ancestor
DFS(v) – Recursive version

Global Initialization:
for all nodes v, v.dfs# = -1  // mark v "undiscovered"
dfscounter = 0
for v = 1 to n do
  if state(v) != fully-explored then
    DFS(v):
    Mark v "discovered".
    for each edge (v,x)
      if (x.dfs# = -1)  // (x previously undiscovered)
        DFS(x)
      else ...
    Mark v “fully-explored”

DFS(v):
  v.dfs# = dfscounter++  // v "discovered", number it
Mark v "discovered".
for each edge (v,x)
  if (x.dfs# = -1)  // (x previously undiscovered)
    DFS(x)
  else …
Mark v “fully-explored”
Kinds of edges – DFS on directed graphs

Edge \((u,v)\)

- **Tree**
  \[ [u \ [v \ v] \ u] \]

- **Forward**
  \[ [u \ [v \ v] \ u] \]

- **Cross**
  \[ [v \ v] \ [u \ u] \]

- **Back**
  \[ [v \ [u \ u] \ v] \]