CSE 421: Introduction to Algorithms

Bipartiteness - DFS
Shayan Oveis Gharan
Bipartite Graphs

Definition: An undirected graph $G=(V,E)$ 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 i.e., no edge has both ends in the same part.

Application:
- Scheduling: machine=red, jobs=blue
- Stable Matching: men=blue, wom=red

*a bipartite graph*
Testing Bipartiteness

Problem: Given a graph G, is it bipartite?

A bipartite graph $G$
Testing Bipartiteness

**Problem:** 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 $G$ is bipartite, then it does not contain an odd length cycle.

Pf: We cannot 2-color an odd cycle, let alone $G$. 

- **bipartite (2-colorable)**
- **not bipartite (not 2-colorable)**
A Characterization of Bipartite Graphs

Lemma: Let $G$ be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS(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).

Case (i)

Case (ii)
A Characterization of Bipartite Graphs

Lemma: Let $G$ be a connected graph, and let $L_0, ..., L_k$ be the layers produced by BFS(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:
\begin{align*}
\text{blue} &= \text{nodes on odd levels,} \\
\text{red} &= \text{nodes on even levels.}
\end{align*}
A Characterization of Bipartite Graphs

**Lemma:** Let $G$ be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS(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 $z = \text{lca}(x, y)$

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 cycles.

---

*bipartite* (2-colorable)  

*not bipartite* (not 2-colorable)
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(s) – Recursive version

Global Initialization: mark all vertices undiscovered

DFS(v)
  Mark v discovered

  for each edge {v,x}
    if (x is undiscovered)
      Mark x discovered
      DFS(x)

  Mark v full-discovered
Suppose edge lists at each vertex are sorted alphabetically.

**DFS(A)**

Color code:
- undiscovered
- discovered
- fully-explored

Call Stack (Edge list):

A (B, J)

st[] = {1}
DFS(A)

Color code:
- undiscovered
- discovered
- fully-explored

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

\[ st[] = \{1,2\} \]
DFS(A)

Color code:
- undiscovered
- discovered
- fully-explored

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

st[] = {1, 2, 3}
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)

st[] = {1,2,3,4}
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 (G,E,F)
E (D,F)

st[] = {1,2,3,4,5}
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)

st[] = {1,2,3,4,5,6}
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)
G (C, F)

$st[] = \{1, 2, 3, 4, 5, 6, 7\}$
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 (G,E,F)
- E (D,F)
- F (D,E,G)
- G (C,F)

\[ st[] = \{1,2,3,4,5,6,7\} \]
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)

\[ \text{st[]} = \{1,2,3,4,5,6\} \]
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)

st[] = {1, 2, 3, 4, 5}
DFS(A)

Call Stack:
(Edge list)

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

st[] = {1,2,3,4}
DFS(A)

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

st[] =
{1, 2, 3}
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)

st[] = \{1,2,3,8\}
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)
  - I (H)

st[] = \{1,2,3,8,9\}
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)

\[ st[] = \{1, 2, 3, 8\} \]
DFS(A)

Diagram:

- A (1)
- B (2)
- C (3)
- D (4)
- E (5)
- F (6)
- G (7)
- H (8)
- I (9)
- J (10)
- K
- L
- M

Call Stack:

- A (B,J)
- B (A,C,J)
- C (B,D,G,H)
- H (C,I,J)
- J (A,B,H,K,L)

Color code:
- undiscovered
- discovered
- fully-explored

st[] = {1, 2, 3, 8, 10}
DFS(A)

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)

st[] = {1,2,3,8,10,11}

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

st[] = {1,2,3,8,10,11,12}
DFS(A)

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)
M (L)

st[] = 
{1,2,3,8,10,11,12,13}
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)

st[] = 
{1,2,3,8,10,11,12}
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)

st[] = 
{1, 2, 3, 8, 10, 11}
DFS(A)

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)

st[] = {1,2,3,8,10}
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)

st[] =
{1,2,3,8,10}
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)

st[] = {1, 2, 3, 8}
DFS(A)

Color code:
- undiscovered
- discovered
- fully-explored

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

st[] = {1, 2, 3}
DFS(A)

Color code:
- undiscovered
- discovered
- fully-explored

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

st[] = \{1,2\}
DFS(A)

Color code:
- undiscovered
- discovered
- fully-explored

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

st[] = {1,2}
DFS(A)

Color code:
- undiscovered
- discovered
- fully-explored

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

\[ \text{st[]} = \{1\} \]
DFS(A)

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

st[] = 
{1}
DFS(A)

Color code:
- undiscovered
- discovered
- fully-explored

Call Stack:
(E) (Edge list)

TA-DA!!

st[] = {}
DFS(A)

Edge code:
Tree edge
Back edge
DFS(A)

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

Like BFS(s):
• DFS(s) visits x iff there is a path in G from s to x
  So, we can use DFS to find connected components
• 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
Non-Tree Edges in DFS

All non-tree edges join a vertex and one of its descendants/ancestors in the DFS tree

BFS tree ≠ DFS tree, but, as with BFS, DFS has found a tree in the graph s.t. non-tree edges are "simple" – only descendant/ancestor
Non-Tree Edges in DFS

**Obs:** During DFS(x) every vertex marked visited is a descendant of x in the DFS tree.

**Lemma:** For every edge \( \{x, y\} \), if \( \{x, y\} \) is not in DFS tree, then one of x or y is an ancestor of the other in the tree.

**Proof:**
One of x or y is visited first, suppose WLOG that x is visited first and therefore DFS(x) was called before DFS(y).

Since \( \{x, y\} \) is not in DFS tree, y was visited when the edge \( \{x, y\} \) was examined during DFS(x).

Therefore y was visited during the call to DFS(x) so y is a descendant of x.
DAGs and Topological Ordering
Precedence Constraints

In a directed graph, an edge \((i, j)\) means task \(i\) must occur before task \(j\).

Applications

- **Course prerequisite:**
  course \(i\) must be taken before \(j\)
- **Compilation:**
  must compile module \(i\) before \(j\)
- **Computing overflow:**
  output of job \(i\) is part of input to job \(j\)
- **Manufacturing or assembly:**
  sand it before paint it
Directed Acyclic Graphs (DAG)

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

**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 $i < j$.

![A DAG and a topological ordering](image)
DAGs: A Sufficient Condition

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

Pf. (by contradiction)
Suppose that G has a topological order 1,2,..., n and that G also has a directed cycle C.
Let i be the lowest-indexed node in C, and let j be the node just before i; thus (j, i) is an (directed) edge.
By our choice of i, we have i < j.
On the other hand, since (j, i) is an edge and 1, ..., n is a topological order, we must have j < i, a contradiction.

![Diagram of directed cycle C and supposed topological order 1, 2, ..., n]
DAGs: A Sufficient Condition

G has a topological order

? 

G is a DAG
Every DAG has a source node

**Lemma:** If $G$ is a DAG, then $G$ has a node with no incoming edges (i.e., a source).

**Pf.** (by contradiction)
Suppose that $G$ is a DAG and it has no source
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.

Is this similar to a previous proof?
Lemma: If $G$ is a DAG, then $G$ has a topological order.

Pf. (by induction on $n$)
Base case: true if $n = 1$.

IH: Every DAG with $n-1$ vertices has a topological ordering.

IS: Given DAG with $n > 1$ nodes, find a source node $v$.
$G - \{ v \}$ is a DAG, since deleting $v$ cannot create cycles.

By IH, $G - \{ v \}$ 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.

Reminder: Always remove vertices/edges to use IH
A Characterization of DAGs

- G has a topological order
- G is a DAG
Topological Order Algorithm: Example
Topological Order Algorithm: Example

Topological order: 1, 2, 3, 4, 5, 6, 7
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 \((v,w)\) \( \quad \text{O}(m + n) \)
- \( S = S \cup \{w\} \) 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 \( \quad \text{O}(1) \) per node
  - for all edges from \( v \) to some \( w \) \( \quad \text{O}(1) \) per edge
    - decrement \( \text{count}[w] \)
    - add \( w \) to \( S \) if \( \text{count}[w] \) hits 0

Correctness: clear, I hope

Time: \( \text{O}(m + n) \) (assuming edge-list representation of graph)
Summary

• Graphs: abstract relationships among pairs of objects

• Terminology: node/vertex/vertices, edges, paths, multi-edges, self-loops, connected

• Representation: Adjacency list, adjacency matrix

• Nodes vs Edges: \( m = O(n^2) \), often less

• BFS: Layers, queue, shortest paths, all edges go to same or adjacent layer

• DFS: recursion/stack; all edges ancestor/descendant

• Algorithms: Connected Comp, bipartiteness, topological sort