CSE 421: Introduction to Algorithms

Connected Components
Bipartiteness

Shayan Oveis Gharan
We want to answer the following type questions (fast):
Given vertices u,v is there a path from u to v in G?

**Idea:** Create an array A such that
For all u, A[u] is the label of the connected component that contains u

Therefore, question reduces to
If A[u] = A[v]?
**Connected Components Implementation**

Initial State: All vertices undiscovered, \( c \leftarrow 0 \)

for \( v = 1 \) to \( n \) do
  If \( \text{state}(v) \neq \text{fully-explored} \) then
    BFS\((v)\): setting \( A[u] \leftarrow c \) for each \( u \) found
    (and marking \( u \) discovered/fully-explored)
    \( c \leftarrow c + 1 \)

**Note:** We no longer initialize to undiscovered in the BFS subroutine

**Total Cost:** \( O(m+n) \)

In every connected component with \( n_i \) vertices and \( m_i \) edges BFS takes time \( O(m_i + n_i) \).
Lesson: We can execute any algorithm on disconnected graphs by running it on each connected component.

We can use the previous algorithm to detect connected components. There is no overhead, because the algorithm runs in time $O(m+n)$.

So, from now on, we can (almost) always assume the input graph is connected.
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
An Obstruction to Bipartiteness

**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.
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)](image1)

![Case (ii)](image2)
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. (i)
Suppose no edge joins two nodes in the same layer.
By previous lemma, all edges join nodes on adjacent levels.

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

Case (i)
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 $z$. 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)
- 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 (C,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 (G,E,F)
- E (D,F)
- F (D,E,G)

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

st[] = {1,2,3,4,5,6,7}
DFS(A)

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

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

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)
  - D (C,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 (G, 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)

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

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

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)
  - 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)
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}

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)

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}

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)

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)

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)

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

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

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

st[] = {1}
DFS(A)

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

st[] = 
{1}
DFS(A)

Color code:
- undiscovered
- discovered
- fully-explored

Call Stack:
(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 DFS 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