CSE 421: Introduction to Algorithms

Application of BFS

Yin Tat Lee
Bipartite Graphs

Definition: An undirected graph $G = (V, E)$ is bipartite if you can partition the vertex set into 2 parts (say, blue/red or left/right) so that all edges join vertices in different parts i.e., no edge has both ends in the same part.

Application:
• Scheduling: machine=red, jobs=blue
• Stable Matching: men=blue, woman=red

a bipartite graph
Testing Bipartiteness

**Problem:** Given a graph $G$, is it bipartite?

Many graph problems become:
- Easier/Tractable if the underlying graph is bipartite (matching)

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

![a bipartite graph $G$](image1)

![another drawing of $G$](image2)
An Obstruction to Bipartiteness

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

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

\[ b_{\text{bipartite}} \quad (2\text{-colorable}) \quad n_{\text{not bipartite}} \quad (\text{not 2-colorable}) \]
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 $L_0, \ldots, L_k$ 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).

Proof. (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 $\text{BFS}(s)$. 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).

Proof. (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**

**Corollary**: A graph $G$ is bipartite if and only if it contains no odd length cycles. Furthermore, one can test bipartiteness using BFS.

![Bipartite Graph](image1)

- **bipartite**
  - (2-colorable)

![Not Bipartite Graph](image2)

- **not bipartite**
  - (not 2-colorable)
Summary

• **BFS** \((s)\) implemented using queue.

• Edges into then-undiscovered vertices define a tree – the “Breadth First spanning tree” of \(G\)

• Level \(i\) in the tree are exactly all vertices \(v\) s.t., the shortest path (in \(G\)) from the root \(s\) to \(v\) is of length \(i\)

• All nontree edges join vertices on the same or adjacent layers of the tree

• Applications:
  • Shortest Path
  • Connected component
  • Test bipartiteness / 2-coloring
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

Initialization: mark all vertices undiscovered

DFS(ν)
Mark ν discovered

for each edge {ν, x}
if (x is undiscovered)
Mark x discovered
x → parent = u
DFS(x)

Mark ν fully-discovered
Non-Tree Edges in DFS

BFS tree $\neq$ DFS tree, but, as with BFS, DFS has found a tree in the graph s.t. non-tree edges are "simple" in some way.

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

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

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

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 (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)

Call Stack:
(Edge list)

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

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

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)
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 (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)

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)

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}

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)

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)

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

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

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)
- 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,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, J, I)
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)

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}

Color code:
- undiscovered
- discovered
- fully-explored
DFS(A)

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

st[] = 
{1, 2, 3}

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)

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)

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

st[] = 
{1}
DFS(A)

Color code:
- undiscovered
- discovered
- fully-explored

Call Stack:
(Edge list)

st[] = {1}

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

Call Stack: (Edge list)

TA-DA!!

st[] = {}

Color code:
- undiscovered
- discovered
- fully-explored
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

**Lemma:** For every **undirected** edge \( \{x, y\} \), then one of \( x \) or \( y \) is an ancestor of the other in the tree.

**Proof:**
Suppose that \( x \) is visited first.
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 \( x \).
Non-Tree Edges (Directed Graph)

**Lemma:** For every directed edge \((x, y)\), then either

- \(y\) is visited first or
- \(y\) is a descendant of \(x\)
CSE 421

Applications of DFS

Topological sort

Yin Tat Lee
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)

Def: A directed acyclic graph (DAG) is a graph that contains no directed cycles.

Def: A topological order of a directed graph $G = (V, E)$ is an ordering of its nodes as $v_1, v_2, ..., v_n$ so that for every edge $(v_i, v_j)$ we have $i < j$. 
Lemma: If $G$ has a topological order, then $G$ is a DAG.

Proof. (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 $1, 2, ..., n$ is a topological order, we must have $j < i$, a contradiction.
DAGs: A Sufficient Condition

$G$ has a topological order $\iff G$ 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).

**Proof.** (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.

The proof is similar to “tree has $n - 1$ edges”.

![Diagram of a DAG with a cycle](attachment:image.png)
Lemma: If $G$ is a DAG, then $G$ has a topological order.

Proof. (by induction on $n$)
Base case: true if $n = 1$.
Hypothesis: Every DAG with $n - 1$ vertices has a topological ordering.
Inductive Step: Given DAG with $n > 1$ nodes, find a source node $v$.
$G - \{v\}$ is a DAG, since deleting $v$ cannot create cycles.

By 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.

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

G has a topological order \iff G is a DAG
Quiz

How to find topological ordering in polynomial time?

Algorithm ($n^2$ time):
Function $\pi = Order(G)$
- Find a vertex $v$ in $G$ with no incoming edge (Time: $n$)
- Return $(v, Order(G - \{v\}))$. (Total Time: $m$)

How to improve the runtime?
- Maintain the set of vertices with no incoming edge.

Alternatively, you can solve this problem by DFS.
Example
Example

Topological order: 1, 2, 3, 4, 5, 6, 7
Summary for last few classes

- Terminology: vertices, edges, paths, connected component, tree, bipartite...
- Vertices vs Edges: $m = O(n^2)$ in general, $m = n - 1$ for trees
- 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
- Techniques: Induction on vertices/layers