CSE 421

DFS / Topological Ordering

Shayan Oveis Gharan
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**
Non-Tree Edges in DFS

All non-tree edges join a vertex and one of its descendents/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.
Suppose edge lists at each vertex are sorted alphabetically.

DFS(A)

Color code:
- undiscovered
- discovered
- fully-explored

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

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

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

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

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)

Nodes:
- A,1
- B,2
- C,3
- D,4
- E
- G
- H
- I
- K
- L
- M

Edges:
- A to B
- A to C
- A to J
- B to A
- B to C
- B to J
- C to B
- C to D
- C to G
- C to H
- D to C
- D to E
- D to F
- E
- G
- H
- I
- K
- L
- M
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)

Diagram:
- A (1)
- B (2)
- C (3)
- D (4)
- E (5)
- F
- G
- H
- I
- J
- K
- L
- M
DFS(A)

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

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

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

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)
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)

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

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

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)

Diagram:
- A
- B
- C
- D
- E
- F
- G
- H
- I
- J
- K
- L
- M
DFS(A)

Color code:
undiscovered
discovered
fully-explored

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

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

Graph:
- A
- B
- C
- D
- E
- F
- G
- H
- I
- J
- K
- L
- M

Nodes:
- A,1
- B,2
- C,3
- D,4
- E,5
- F,6
- G,7
- H,8
- I,9
- J
- K
- L
- M

Edges:
- A to B
- A to J
- B to A
- B to C
- B to J
- C to B
- C to D
- C to G
- C to H
- H to C
- H to I
- I to H
- J
- K
- L
- M

Colors:
- Red: undiscovered
- Green: discovered
- Black: 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)
I (H)

A,1
B,2
C,3
D,4
E,5
F,6
G,7
H,8
J
K
L
M
DFS(A)

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

Color code:
undiscovered
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)
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)

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)
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)
J (A,B,H,K,L)
K (J,L)
L (J,K,M)
M(L)
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)
DFS(A)
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)
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)
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)

Graph:
- 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)
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)
- B (2)
- C (3)
- D (4)
- E (5)
- F (6)
- G (7)
- H (8)
- I (9)
- J
- K (11)
- L (12)
- M (13)
DFS(A)

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

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)

Graph:
- A
  - B
    - C
      - D
      - E
    - G
    - H
    - F
    - I
  - J
    - K
    - L
    - M
DFS(A)

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

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

Color code:
- undiscovered
- discovered
- fully-explored

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

A,1

B,2

C,3

G,7

H,8

K,11

L,12

M,13

D,4

F,6

I,9

E,5
DFS(A)

Call Stack:
(Edge list)

TA-DA!!
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 descendents/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

**Lemma:** 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 $i < j$. 

![Diagram of a DAG and a topological ordering](diagram.png)

- A DAG
- A topological ordering of that DAG—all edges left-to-right
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](image-url)
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.
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 $\iff$ 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 \((v,w)\) \(\text{O}(m + n)\)
- \( S = S \cup \{w\} \) 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 \(\text{O}(1) \) per node
  - for all edges from \( v \) to some \( w \) \(\text{O}(1) \) 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