CSE 421: Introduction to Algorithms

DFS - DAGs
Shayan Oveis Gharan
HW1 Grade

Q: I received low grade in HW1 what should I do?
• Understand what was your mistake. Did you understand the problem statement correctly?
• Show up to office hours and ask for hints or to explain your solution
• Review materials of 311 on proofs/induction
• Do exercises from the book/Problem Solving Sessions

Q: My HW1 grade is low, will I be able to receive 4.0?
• Yes! I look at your progress. Many students are behind at beginning but by practice they catch up and receive 4.0

Q: I have filled out a regrade request, but was not convinced, what should I do?
• Show up to my office hour and discuss your solution
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.

Color code:
- undiscovered
- discovered
- fully-explored

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

st[] = {1}
DFS(A)

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,B,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)

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

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

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 (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 (G,E,F)
E (D,F)
F (D,E,G)

Color code:
- undiscovered
- discovered
- fully-explored

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 (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)

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)

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)
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)

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)
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)

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)
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)

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)
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)

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[] = {1,2}
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)

st[] = {1}
DFS(A)

Call Stack:
(Edge list)
TA-DA!!

st[] = {}

Color code:
undiscovered
discovered
fully-explored

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)
DFS(A)

Edge code:
- Tree edge
- Back edge

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)

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
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 discovered first, suppose WLOG that x is discovered first and therefore DFS(x) was called before DFS(y)

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

Therefore y was discovered during the call to DFS(x) so y is a descendant of x by observation.
DAGs and Topological Ordering
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

a topological ordering of that DAG—all edges left-to-right
**Lemma**: If $G$ has a topological order, then $G$ is a DAG.

**Pf.** (by contradiction)

Suppose that $G$ has a topological order $1, 2, \ldots, n$ 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 $1, \ldots, n$ is a topological order, we must have $j < i$, a contradiction.

![Diagram of a directed cycle and topological order]

The directed cycle $C$ and the supposed topological order: $1, 2, \ldots, n$.
DAGs: A Sufficient Condition

G has a topological order \[ \Rightarrow \] 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 $\iff$ G is a DAG
Topological Order Algorithm: Example
Topological Order Algorithm: Example

Topological order: 1, 2, 3, 4, 5, 6, 7

Induction gives Algorithms!
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)
DFS on Directed Graphs

• Before DFS(s) returns, it visits all previously unvisited vertices reachable via directed paths from s

• Every cycle contains a back edge in the DFS tree
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
Greedy Algorithms
Greedy Strategy

**Goal:** Given currency denominations: 1, 5, 10, 25, 100, give change to customer using *fewest* number of coins.

**Ex:** 34¢.

**Cashier's algorithm:** At each iteration, give the *largest* coin valued \( \leq \) the amount to be paid.

**Ex:** $2.89.
Greedy is not always Optimal

**Observation:** Greedy algorithm is sub-optimal for US postal denominations: 1, 10, 21, 34, 70, 100, 350, 1225, 1500.

**Counterexample.** 140¢.
Greedy: 100, 34, 1, 1, 1, 1, 1, 1.
Optimal: 70, 70.

**Lesson:** Greedy is short-sighted. Always chooses the most attractive choice at the moment. But this may lead to a dead-end later.
Greedy Algorithms Outline

Pros
• Intuitive
• Often simple to design (and to implement)
• Often fast

Cons
• Often incorrect!

Proof techniques:
• Stay ahead
• Structural
• Exchange arguments