Graph Traversals

Slides by Carl Kingsford

Jan. 31, 2014

Based on/Reading: Chapter 3 of Kleinberg & Tardos
Depth-First Search

DFS keeps walking down a path until it is forced to backtrack.

It backtracks until it finds a new path to go down.

Think: Solving a maze.

It results in a search tree, called the depth-first search tree.

In general, the DFS tree will be very different than the BFS tree.
Depth-First Search
Example DFS

0:0:1

1:-1:-1
2:-1:-1
3:-1:-1
4:-1:-1
5:-1:-1
6:-1:-1
7:-1:-1
8:-1:-1
9:-1:-1
10:-1:-1
11:-1:-1
12:-1:-1
13:-1:-1
14:-1:-1

6:1:1
13:1:1
10:1:1
11:1:1
14:1:1
5:1:1
8:1:1
12:1:1
1:1:1

3:1:1
4:1:1
9:1:1
2:1:1
Example DFS
Example DFS
Example DFS
Example DFS
Example DFS

0:0:-1
1:5:-1 2:4:-1
3:-1:-1 4:-1:-1
5:2:-1
6:7:-1
7:6:-1
8:-1:-1
9:-1:-1
10:-1:-1
11:9:-1
12:3:-1
13:8:-1
14:1:-1

Graph:

- Node 11 is highlighted.
- Edges connect nodes as follows:
  - 0 to 1
  - 1 to 2, 10
  - 2 to 3, 12
  - 3 to 4, 13
  - 4 to 9
  - 5 to 6
  - 6 to 7
  - 7 to 8
  - 8 to 14
  - 10 to 0
  - 11 to 6, 12
  - 12 to 13
  - 13 to 14
  - 14 to 0

- The graph is a directed acyclic graph (DAG).

Example DFS
Example DFS
Example DFS
Example DFS
Example DFS
Tree Terminology

The root of the tree is a distinguished node that we “hang” the rest of the tree off of. Here, the roots are the starting nodes for DFS or BFS.

A node $x$ is an ancestor of node $y$ in a tree if there’s a path from $y$ to $x$ to the root of the tree.

Note that $x$ does not need to have an edge to $y$ to be its ancestor.

Node $y$ is a descendant of $x$ if $x$ is an ancestor of $y$. 
Theorem. Let $x$ and $y$ be nodes in the DFS tree $T_G$ such that \{x, y\} is an edge in undirected graph $G$. Then one of $x$ or $y$ is an ancestor of the other in $T_G$.

Proof. Suppose, wlog, $x$ is reached first in the DFS. When we reach $x$, node $y$ must not yet have been explored.

All the nodes that are marked explored between first encountering $x$ and leaving $x$ for the last time are descendants of $x$ in $T_G$.

It must become explored before leaving $x$ for the last time (otherwise, we should add \{x, y\} to $T_G$). Hence, $y$ is a descendent of $x$ in $T_G$. \qed
Either $y$ is visited through some other child $w$ of $x$, or it is visited directly from $x$ just before we leave $x$ for good:
Example Breadth-First Search
Example Breadth-First Search
Example Breadth-First Search
Example Breadth-First Search
Example Breadth-First Search
Example Breadth-First Search
Example Breadth-First Search
Example Breadth-First Search
Example Breadth-First Search
Example Breadth-First Search
Example Breadth-First Search
Example Breadth-First Search
Example Breadth-First Search
Example Breadth-First Search
Example Breadth-First Search
Breadth-First Search

Breadth-first search explores the nodes of a graph in increasing distance away from some starting vertex \( s \).

It decomposes the component into layers \( L_i \) such that the shortest path from \( s \) to each of nodes in \( L_i \) is of length \( i \).

Breadth-First Search:

1. \( L_0 \) is the set \( \{s\} \).
2. Given layers \( L_0, L_1, \ldots, L_j \), then \( L_{j+1} \) is the set of nodes that are not in a previous layer and that have an edge to some node in layer \( L_j \).
A BFS traversal of a graph results in a breadth-first search tree:

Can we say anything about the non-tree edges?
BFS Tree Example

A BFS traversal of a graph results in a breadth-first search tree:

Can we say anything about the non-tree edges?
BFS Tree
Property of Non-BFS-Tree Edges

**Theorem.** Choose $x \in L_i$ and $y \in L_j$ such that $\{x, y\}$ is an edge in undirected graph $G$. Then $i$ and $j$ differ by at most 1.

In other words, edges of $G$ that do not appear in the tree connect nodes either in the same layer or adjacent layer.

**Proof.** Suppose not, and that $i < j - 1$.

All the neighbors of $x$ will be found by layer $i + 1$.

Therefore, the layer of $y$ is less than $i + 1$, so $j \leq i + 1$, which contradicts $i < j - 1$. \qed
General Tree Growing (following Gross & Yellen)

We can think of BFS and DFS (and several other algorithms) as special cases of tree growing:

- Let $T$ be the current tree $T$, and
- Maintain a list of frontier edges: the set of edges of $G$ that have one endpoint in $T$ and one endpoint not in $T$:

![Diagram showing tree growing process]

- Repeatedly choose a frontier edge (somehow) and add it to $T$. 
Tree Growing

TreeGrowing(graph G, vertex v, func nextEdge):
    T = (v, ∅)
    S = set of edges incident to v
    While S is not empty:
        e = nextEdge(G, S)
        T = T + e // add edge e to T
        S = updateFrontier(G, S, e)
    return T

- The function nextEdge(G, S) returns a frontier edge from S.
- updateFrontier(G, S, e) returns the new frontier after we add edge e to T.
Prim’s Algorithm

**Prim’s Algorithm**: Run TreeGrowing starting with any root node, adding the frontier edge with the smallest weight.

**Theorem.** *Prim’s algorithm produces a minimum spanning tree.*

$$S = \text{set of nodes already in the tree when } e \text{ is added}$$
These algorithms are all special cases / variants of Tree Growing, with different versions of nextEdge:

1. Depth-first search
2. Breadth-first search
3. Prim’s minimum spanning tree algorithm
4. Dijkstra’s shortest path
5. A*
BFS & DFS as Tree Growing

What’s nextEdge for DFS?

What’s nextEdge for BFS?
BFS & DFS as Tree Growing

What’s nextEdge for DFS?

Select a frontier edge whose tree endpoint was discovered most recently.

Why? We can use a stack to implement DFS.
Runtime for DFS: $O(|Edges|)$

What’s nextEdge for BFS?
BFS & DFS as Tree Growing

What’s nextEdge for DFS?

Select a frontier edge whose tree endpoint was discovered most recently.

Why? We can use a stack to implement DFS.
Runtime for DFS: \( O(|\text{Edges}|) \)

What’s nextEdge for BFS?

Select a frontier edge whose tree endpoint was discovered earliest.

Why? We can use a queue to implement BFS.
Runtime for BFS: \( O(|\text{Edges}|) \)
Implementations of BFS and DFS
nextEdge for BFS

nextEdge: frontier edge connecting to node with *earliest* discovery time.

A **queue** maintains a list of items in order of their discovery so that the item discovered farthest in the past can be accessed quickly:

![Queue Diagram]

```
G E D B A
enqueue(F)
∅
dequeue()
```

defqueue()

tail

Dequeue(F)

dequeue()

head

Queues are “first in, first out” (FIFO) data structures.
procedure bfs(G, s):
    Q := queue containing only s
    while Q not empty
        v := Q.front(); Q.remove_front()
        for w ∈ G.neighbors(v):
            if w not seen:
                mark w seen
                Q.enqueue(w)
Recursive implementation of DFS

procedure dfs(G, u):
    while u has an unvisited neighbor in G
        v := an unvisited neighbor of u
        mark v visited
        dfs(G, v)
**nextEdge for DFS**

nextEdge: frontier edge connecting to node with *latest* discovery time.

A **stack** maintains a list of items so they can be accessed in reverse order of their discovery times.

A stack is a “last in, first out” (LIFO) data structure.
Stack-based implementation of DFS

procedure dfs(G, s):
    S := stack containing only s
    while S not empty
        v := S.pop()
        if v not visited:
            mark v visited
            for w ∈ G.neighbors(v): S.push(w)