EECS 2011 M: Fundamentals of Data Structures

Suprakash Datta
Office: LAS 3043

Course page: `http://www.eecs.yorku.ca/course/2011M`
Also on Moodle

Graphs: Exploration and Searching

Method to explore many key properties of a graph

- Nodes that are reachable from a specific node v
- Detection of cycles
- Extraction of strongly connected components
- Topological sorts
- Find a path with the minimum number of edges between two given vertices

Note: Some slides in this presentation have been adapted from the author's and Prof Elder's slides.

Graph Search Algorithms

- Depth-first Search (DFS)
- Breadth-first Search (BFS)

Depth-first Search

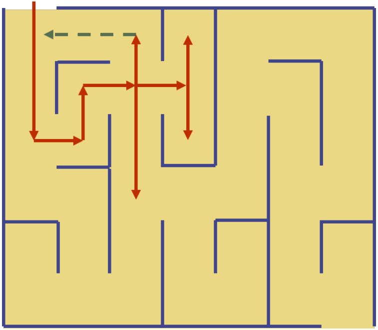
A DFS traversal of a graph G

- Visits all the vertices and edges of G
- Determines whether G is connected
- Computes the connected components of G
- Computes a spanning forest of G
- Find a cycle in the graph

Depth-first Search - 2

The DFS algorithm is similar to a classic strategy for exploring a maze

- We mark each intersection, corner and dead end (vertex) visited
- We mark each corridor (edge) traversed
- We keep track of the path back to the entrance (start vertex) by means of a rope (recursion stack)



Depth-first Search - Algorithm

Algorithm DFS(G, u):

Input: A graph G and a vertex u of G

Output: A collection of vertices reachable from u, with their discovery edges

Mark vertex u as visited.

for each of u's outgoing edges, $e = (u, v)$ **do**

  if vertex v has not been visited **then**

    Record edge e as the discovery edge for vertex v.

    Recursively call DFS(G, v).

Depth-first Search - Java Implementation

```
/** Performs depth-first search of Graph g starting at Vertex u. */
public static <V,E> void DFS(Graph<V,E> g, Vertex<V> u,
                Set<Vertex<V>> known, Map<Vertex<V>,Edge<E>> forest) {
  known.add(u);                              // u has been discovered
  for (Edge<E> e : g.outgoingEdges(u)) {     // for every outgoing edge from u
    Vertex<V> v = g.opposite(u, e);
    if (!known.contains(v)) {
      forest.put(v, e);                      // e is the tree edge that discovered v
      DFS(g, v, known, forest);              // recursively explore from v
    }
  }
}
```

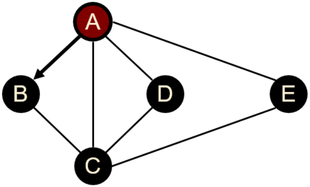
An Augmented DFS

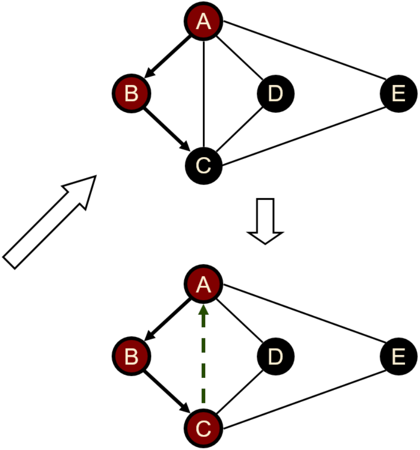
When a vertex discovered, explore every incident edge from it. Keep track of progress by colouring vertices:

- Black: undiscovered vertices
- Red: discovered, but not finished (still exploring from it)
- Gray: finished (Discovered everything reachable from it).

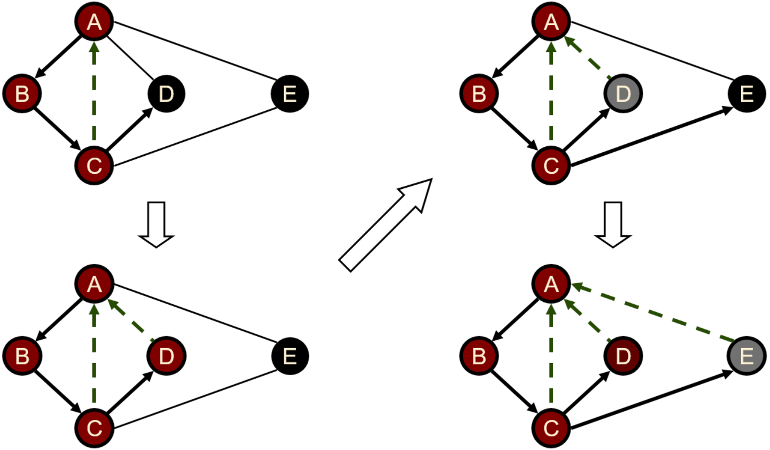
Depth-first Search - Example

- unexplored
- being explored
- finished
- unexplored edge
- discovery edge
- back edge





Depth-first Search - Example



Augmented DFS - Algorithm

DFS(G)

Precondition: G is a graph

Postcondition: all vertices in G have been visited

```
for each vertex u ∈ V[G]
    color[u] = BLACK //initialize vertex
for each vertex u ∈ V[G]
    if color[u] = BLACK //as yet unexplored
        DFS-Visit(u)
```

DFS-Visit - Algorithm

DFS-Visit (u)

Precondition: vertex u is undiscovered

Postcondition: all vertices reachable from u have been processed

```
colour[u] ← RED
for each v ∈ Adj[u] //explore edge (u,v)
    if color[v] = BLACK
        DFS-Visit(v)
colour[u] ← GRAY
```

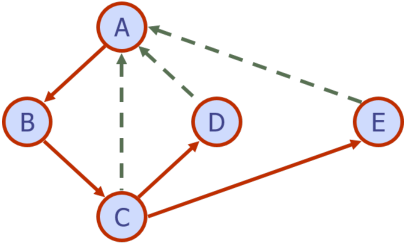
Q: How are the edges classified?

Q: What do back edges signify?

Notice the implicit stack in the code.

DFS: Properties

- Property 1:
  DFS-Visit(v) visits all the vertices and edges in the connected component of v
- Property 2:
  The discovery edges labeled by DFS(v) form a spanning tree of the connected component of v



DFS: Analysis

- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice
  - once as UNEXPLORED once as DISCOVERY or BACK
- Method DFS-Visit is called once for each vertex
- DFS runs in $\theta(n + m)$ time provided the graph is represented by the adjacency list structure:
  Recall that $\sum_v deg(v) = 2m$

DFS on Directed Graphs

- Tree edges are edges in the depth-first forest G_π. Edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v)
- Back edges are those edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree
- Forward edges are non-tree edges (u, v) connecting a vertex u to a descendant v in a depth-first tree
- Cross edges are all other edges. They can go between vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other.
- Classifying edges can help to identify properties of the graph, e.g., a graph is acyclic iff DFS yields no back edges

DFS on Undirected Graphs

- In a depth-first search of a connected undirected graph, every edge is either a tree edge or a back edge

DFS: Another Extension

- In addition to, or instead of labeling vertices with colours, they can be labeled with discovery and finishing times.
- Time is an integer that is incremented whenever a vertex changes state
  - from unexplored to discovered
  - from discovered to finished
- These discovery and finishing times can then be used to solve other graph problems (e.g., computing strongly-connected components)
- Two timestamps put on every vertex:
  - discovery time $d(v) \geq 1$
  - finish time $1 < f(v) \leq 2n$

Modified DFS-Visit - Algorithm

DFS-Visit (u)

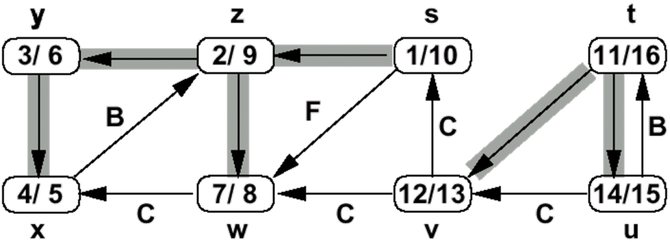
Precondition: vertex u is undiscovered

Postcondition: all vertices reachable from u have been processed

```
colour[u] ← RED
time ← time + 1
d[u] ← time
for each v ∈ Adj[u] //explore edge (u,v)
    if color[v] = BLACK
        DFS-Visit(v)
colour[u] ← GRAY
time ← time + 1
f[u] ← time
```

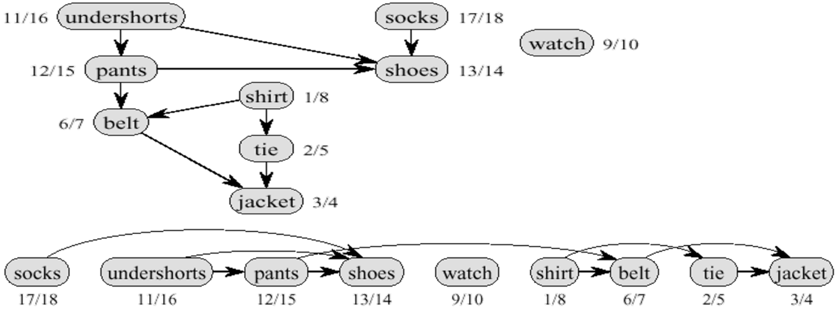
DFS does not change except the global variable time is initialized to 0 in it

Modified DFS - Advantages



- Time stamps are useful for many purposes
- E.g., Topological Sort – sorting vertices of a directed acyclic graph

DFS Application: Topological Sort



- call DFS(G) to compute finishing times $f[v]$ for each vertex v
- return the list of vertices sorted in decreasing order of $f[v]$

DFS Application: Path Finding

- We can adapt the DFS algorithm to find a path between vertices u and z
- We call DFS(G, u) with u as the start vertex
- We use a stack S to keep track of the path between the start vertex and the current vertex
- As soon as destination vertex z is encountered, we return the path as the contents of the stack
- Q: What is the color of the nodes on the path?

DFS Application: Cycle Finding

- We can adapt the DFS algorithm to find a simple cycle
- We use a stack S to keep track of the path between the start vertex and the current vertex
- As soon as a back edge (v, w) is encountered, we return the cycle as the portion of the stack from the top to vertex w

Breadth First Search

Another general technique for traversing a graph

- A BFS traversal of a graph G
  - Visits all the vertices and edges of G
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning forest of G
- BFS on a graph with $|V|$ vertices and $|E|$ edges takes $\theta(|V| + |E|)$ time
- BFS can be further extended to solve other graph problems
  - Find and report a path with the minimum number of edges between two given vertices
  - Cycle detection

Breadth First Search - 2

- Notice that in BFS exploration takes place on a level or wavefront consisting of nodes that are all the same distance from the source s
- We can label these successive wavefronts by their distance: $L_0, L_1, \ldots$
- Notice that a queue is used instead of a stack

Breadth First Search - 3

- Input: directed or undirected graph $G = (V, E)$, source vertex $s \in V$
- Output: for all $v \in V$
  - $d[v]$, the shortest distance from s to v
  - $\pi[v] = u$, such that (u, v) is the last edge on the shortest distance from s to v
- Idea: send out search 'wave' from s
- Keep track of progress by colouring vertices:
  - Undiscovered vertices are coloured black
  - Just discovered vertices (on the wavefront) are coloured red
  - Previously discovered vertices (behind wavefront) are coloured grey

Breadth First Search - Algorithm

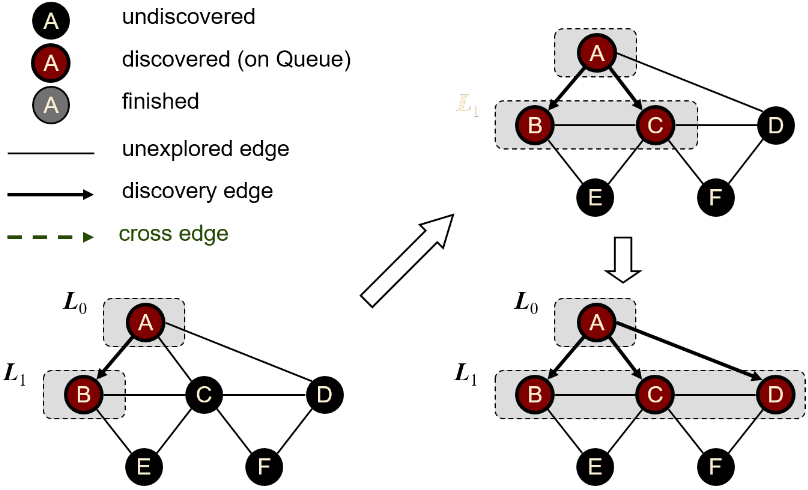
BFS(G,s)

Precondition: G is a graph, s is a vertex in G

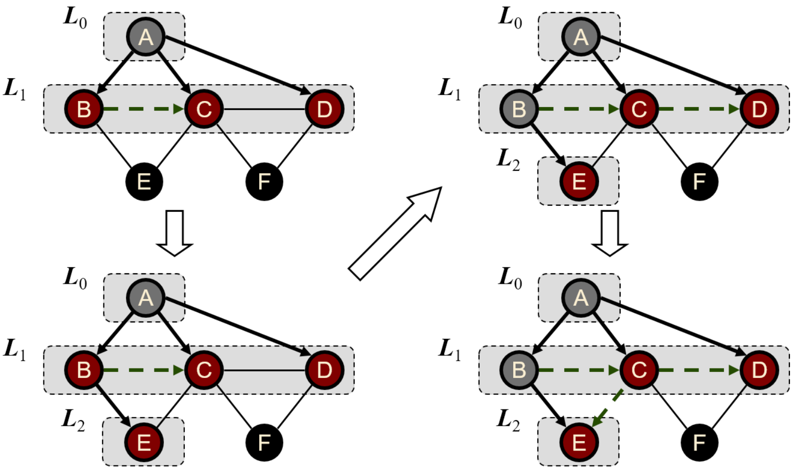
Postcondition: all vertices in G reachable from s have been visited

```
for each vertex u ∈ V[G]
    color[u] ← BLACK //initialize vertex
colour[s] ← RED
Q.enqueue(s)
while Q ≠ ∅
    u ← Q.dequeue()
    for each v ∈ Adj[u] //explore edge (u,v)
        if color[v] = BLACK
            colour[v] ← RED
            Q.enqueue(v)
    colour[u] ← GRAY
```

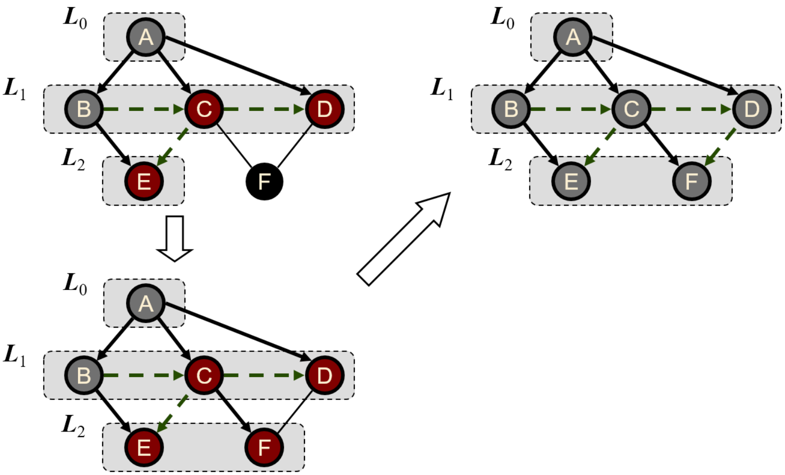
Breadth First Search - Example



Breadth First Search - Example



Breadth First Search - Example



BFS: Properties

Notation: G_s: connected component containing s

- Property 1: BFS(G, s) visits all the vertices and edges of G_s
- Property 2: The discovery edges labeled by BFS(G, s) form a spanning tree T_s of G_s
- Property 3: For each vertex v in L_i
  - The unique path from s to v on T_s has i edges
  - Every path from s to v in G_s has at least i edges

BFS: Analysis

- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled three times
  - once as BLACK (undiscovered)
  - once as RED (discovered, on queue)
  - once as GRAY (finished)
- Each edge is considered twice (for an undirected graph)
- Each vertex is placed on the queue once
- Thus BFS runs in $\theta(|V| + |E|)$ time provided the graph is represented by an adjacency list structure

BFS Application: Shortest Unweighted Paths

- Goal: To recover the shortest paths from a source node s to all other reachable nodes v in a graph
  - The length of each path and the paths themselves are returned
- Notes:
  - There are an exponential number of possible paths
  - Analogous to level order traversal for trees
  - This problem is harder for general graphs than trees because of cycles!