Overview

- Undirected Graph Traversals
  - Depth-First Search
  - Breadth-First Search

**Undirected Graph Traversal - DFS:**

- **Definition:**
  - A *graph traversal* is a systematic procedure for visiting all vertices and edges of a graph.
  - Efficient if it visits all vertices and edges in linear time.
  - Two efficient methods:
    - Depth-First Search
    - Breadth-First Search

- **Depth-First Search (DFS):**
  - “Search” deeper in the graph whenever possible.
  - Edges are “explored” out of the the most recently visited vertex \( v \) that still has unexplored edges leaving it.
  - When all of \( v \)’s edges have been explored, search “backtracks” to explore edges leaving the vertex from which \( v \) was discovered.
  - This process continues until all vertices reachable from the original source vertex have been discovered.
  - If any undiscovered vertices remain, one of them is selected as a new source and search repeats.
Undirected Graph Traversal - DFS:

- Visualize DFS by orienting edges along the directions they are explored during the traversal.
  - Discovery or tree Edges:
    - Edges used to discover new vertices.
  - Back Edges:
    - Edges leading to already visited vertices.

Exploring a Labyrinth Without Getting Lost

- A depth-first search (DFS) in an undirected graph G is like wandering in a labyrinth, with a string and a can of red paint without getting lost.
- We start at vertex $s$, tying the end of our string to the point and painting it “visited”. Next we label $s$ as our current vertex called $u$.
- Now we travel along an arbitrary edge $(u,v)$.
- If edge $(u,v)$ leads us to an already visited vertex $v$ we return to $u$.
- If vertex $v$ is unvisited, we unroll our string and move to $v$, paint $v$ “visited”, set $v$ as our current vertex, and repeat the previous steps.
- Eventually, we will get to a point where all incident edges on $v$ lead to visited vertices. We then backtrack by unrolling our string to a previously visited vertex $u$. Then $u$ becomes our current vertex and we repeat the previous steps.

Exploring a Labyrinth Without Getting Lost (cont.)

- Then, if all incident edges on $v$ lead to visited vertices, we backtrack as we did before. We continue to backtrack along the path we have traveled, finding and exploring unexplored edges, and repeating the procedure.
- When we backtrack to vertex $s$ and there are no more unexplored edges incident on $s$, we have finished our DFS search.

- Discovery edges form a spanning tree of the connected component starting at start vertex $s$. 
Undirected Graph Traversal - DFS:

- Algorithm:

  Algorithm DFS(v);
  Input: A vertex v in a graph
  Output: A labeling of the edges as "discovery" edges and "backedges"
  for each edge e incident on v do
    if edge e is unexplored then
      let w be the other endpoint of e
      if vertex w is unexplored then
        label e as a discovery edge
        recursively call DFS(w)
      else
        label e as a backedge

- Algorithm Assumptions:
  - Have a “way” to determine whether a vertex or edge has been explored or not.
  - Have a “way” to label edges as discovery or back edges.
  - This may require additional storage space and may affect running time!

Undirected Graph Traversal - DFS:

- Running Time:

  - Remember:
    - DFS is called on each vertex exactly once.
    - Every edge is examined exactly twice, once from each of its vertices.
  - For n vertices and m edges in the connected component of the vertex s, a DFS starting at s runs in O(E + m) time if:
    - The graph is represented in a data structure, like the adjacency list, where vertex and edge methods take constant time
    - Marking a vertex as explored and testing to see if a vertex has been explored takes O(degree)
    - By marking visited nodes, we can systematically consider the edges incident on the current vertex so we do not examine the same edge more than once.
Undirected Graph Traversal - DFS:

Marking Vertices

• Let’s look at ways to mark vertices in a way that satisfies the above condition.
• Extend vertex positions to store a variable for marking

Before Position

After Position

Element

isMarked

• Use a hash table mechanism which satisfies the above condition is the probabilistic sense, because it supports the mark and test operations in $O(1)$ expected time.

Let $G$ be a graph with $n$ vertices and $m$ edges represented with an adjacency list structure. There exists $O(n+m)$ algorithms based on DFS to compute:

◆ Test whether $G$ is connected.
◆ Compute spanning tree of $G$ if $G$ is connected.
◆ Compute connected components of $G$.
◆ Compute path between two vertices of $G$ or report no path such path exists.
◆ Compute cycle in $G$ or report no cycle exists.

DFS Example: (1)

Determining Incident Edges:

• DFS depends on how you obtain the incident edges.
• If we start at $A$ and we examine the edge to $F$, then to $B$, then $E$, $C$, and finally $G$.

The resulting graph is:

• discoveryEdge
• backEdge
• return from dead end

If we instead examine the tree starting at $A$ and looking at $F$, the $C$, then $E$, $B$, and finally $F$.

the resulting set of backEdges, discoveryEdges and recursion points is different.

• Now an example of a DFS.

DFS Example: (2)
DFS Example: (11)

And we're done!
Breadth-First Search

- Like DFS, a Breadth-First Search (BFS) traverses a connected component of a graph, and in doing so defines a spanning tree with several useful properties.

  - The starting vertex $s$ has level 0, and, as in DFS, defines that point as an “anchor.”
  - In the first round, the string is unrolled the length of one edge, and all of the edges that are only one edge away from the anchor are visited.
  - These edges are placed into level 1.
  - In the second round, all the new edges that can be reached by unrolling the string 2 edges are visited and placed in level 2.
  - This continues until every vertex has been assigned a level.
  - The label of any vertex $v$ corresponds to the length of the shortest path from $s$ to $v$. 
BFS - A Graphical Representation

a)

b)

c)

d)
More BFS

e)

f)
Algorithm BFS(s):

**Input:** A vertex \( s \) in a graph

**Output:** A labeling of the edges as “discovery” edges and “cross edges”

initialize container \( L_0 \) to contain vertex \( s \)

\( i \leftarrow 0 \)

while \( L_i \) is not empty do

create container \( L_{i+1} \) to initially be empty

for each vertex \( v \) in \( L_i \) do

for each edge \( e \) incident on \( v \) do

if edge \( e \) is unexplored then

let \( w \) be the other endpoint of \( e \)

if vertex \( w \) is unexplored then

label \( e \) as a discovery edge

insert \( w \) into \( L_{i+1} \)

else

label \( e \) as a cross edge

\( i \leftarrow i + 1 \)
Properties of BFS

• **Proposition:** Let $G$ be an undirected graph on which a BFS traversal starting at vertex $s$ has been performed. Then
  - The traversal visits all vertices in the connected component of $s$.
  - The discovery-edges form a spanning tree $T$, which we call the **BFS tree**, of the connected component of $s$
  - For each vertex $v$ at level $i$, the path of the BFS tree $T$ between $s$ and $v$ has $i$ edges, and any other path of $G$ between $s$ and $v$ has at least $i$ edges.
  - If $(u, v)$ is an edge that is not in the BFS tree, then the level numbers of $u$ and $v$ differ by at most one.

• **Proposition:** Let $G$ be a graph with $n$ vertices and $m$ edges. A BFS traversal of $G$ takes time $O(n + m)$. Also, there exist $O(n + m)$ time algorithms based on BFS for the following problems:
  - Testing whether $G$ is connected.
  - Computing a spanning tree of $G$
  - Computing the connected components of $G$
  - Computing, for every vertex $v$ of $G$, the minimum number of edges of any path between $s$ and $v$. 