Subgraphs

- A subgraph $S$ of a graph $G$ is a graph such that
  - The vertices of $S$ are a subset of the vertices of $G$
  - The edges of $S$ are a subset of the edges of $G$
- A spanning subgraph of $G$ is a subgraph that contains all the vertices of $G$
Connectivity

- A graph is connected if there is a path between every pair of vertices.
- A connected component of a graph $G$ is a maximal connected subgraph of $G$.

Trees and Forests

- A (free) tree is an undirected graph $T$ such that:
  - $T$ is connected
  - $T$ has no cycles
  - This definition of tree is different from the one of a rooted tree
- A forest is an undirected graph without cycles.
- The connected components of a forest are trees.
Spanning Trees and Forests

- A spanning tree of a connected graph is a spanning subgraph that is a tree.
- A spanning tree is not unique unless the graph is a tree.
- Spanning trees have applications to the design of communication networks.
- A spanning forest of a graph is a spanning subgraph that is a forest.

Depth-First Search

- Depth-first search (DFS) is a general technique for traversing a graph.
- 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.
- DFS on a graph with \( n \) vertices and \( m \) edges takes \( O(n + m) \) time.
- DFS can be further extended to solve other graph problems:
  - Find and report a path between two given vertices.
  - Find a cycle in the graph.
- Depth-first search is to graphs what Euler tour is to binary trees.
DFS Algorithm from a Vertex

**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).

Java Implementation

```java
/** 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
    if (!known.contains(e)) { // e is the tree edge that discovered v
      forest.put(e, u); // recursively explore from v
      DFS(g, v, known, forest);
    }
  }
}
```

© 2014 Goodrich, Tamassia, Goldwasser
Depth-First Search
Depth-First Search

Example

- **unexplored vertex**
- **visited vertex**
- **unexplored edge**
- **discovery edge**
- **back edge**

Example (cont.)
DFS and Maze Traversal

- 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)

Properties of DFS

Property 1

\(DFS(G, v)\) visits all the vertices and edges in the connected component of \(v\)

Property 2

The discovery edges labeled by \(DFS(G, v)\) form a spanning tree of the connected component of \(v\)
**Analysis of DFS**

- 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 incidentEdges is called once for each vertex.
- DFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure.
  - Recall that $\Sigma_v \deg(v) = 2m$.

**Path Finding**

- We can specialize the DFS algorithm to find a path between two given vertices $u$ and $z$ using the template method pattern.
- 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.

**Algorithm** pathDFS(G, v, z)

```plaintext
setLabel(v, VISITED)
S.push(v)
if v = z
    return S.elements()
for all e ∈ G.incidentEdges(v)
    if getLabel(e) = UNEXPLORED
        w ← opposite(v,e)
        if getLabel(w) = UNEXPLORED
           setLabel(e, DISCOVERY)
            S.push(e)
            pathDFS(G, w, z)
            S.pop(e)
        else
            setLabel(e, BACK)
    S.pop(v)
```
Path Finding in Java

```java
// © 2014 Goodrich, Tamassia, Goldwasser
// Depth-First Search

depthFirstSearch GRAPH g, VERTEX v, VERTEX w

// © 2014 Goodrich, Tamassia, Goldwasser
// Depth-First Search
```

Cycle Finding

- We can specialize the DFS algorithm to find a simple cycle using the template method pattern.
- 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 $w$.

```
Algorithm cycleDFS(G, v, w)
setLabel(v, VISITED)
S.push(v)

for all e ∈ G.incidentEdges(v)
if getLabel(e) = UNEXPLORED
w ← opposite(v, e)
S.push(e)
if getLabel(w) = UNEXPLORED
setLabel(e, DISCOVERY)
pathDFS(G, w, z)
S.pop(e)
else
T ← new empty stack
repeat
    o ← S.pop()
    T.push(o)
until o = w
return T.elements()
S.pop(v)
```

© 2014 Goodrich, Tamassia, Goldwasser
DFS for an Entire Graph

- The algorithm uses a mechanism for setting and getting “labels” of vertices and edges.

Algorithm $DFS(G)$

- **Input**: graph $G$
- **Output**: labeling of the edges of $G$ as discovery edges and back edges

```
for all $u \in G$.vertices()
    setLabel($u$, UNEXPLORED)

for all $e \in G$.edges()
    setLabel($e$, UNEXPLORED)

for all $v \in G$.vertices()
    if getLabel($v$) = UNEXPLORED
        $DFS(G, v)$
```

All Connected Components

- Loop over all vertices, doing a DFS from each unvisited one.

```java
/** Performs DFS for the entire graph and returns the DFS forest as a map. */
public static <V,E> Map<Vertex<V>, Edge<E>> DFSComplete(Graph<V,E> g) {
    Set<Vertex<V>> known = new HashSet<>();
    Map<Vertex<V>, Edge<E>> forest = new ProbHashMap<>();
    for (Vertex<V> u : g.vertices())
        if (!known.contains(u))
            DFS(g, u, known, forest); // (re)start the DFS process at u
    return forest;
}
```