

Graph Traversal

Building blocks and observations –

- Graph ADT provides operations for getting edges incident on a vertex, and end vertices of an edge
 - from a vertex you can find edges, and from an edge you can find the vertex at the other end
- there may be more than one vertex adjacent to another, so you can't just trace through the graph using a single finger to point at where you are – need a container to hold *discovered* vertices

Using a ~~queue~~ stack for the container leads to ~~breadth-first~~ depth-first search.

- however, DFS is typically implemented recursively rather than using a separate stack

Depth-First Search

```
dfs(G, s)    G is the graph, s is the starting vertex
for each vertex u in V-{\s} do
    state[u] = "undiscovered"
    prev[u] = null
state[s] = "discovered"
prev[s] = null
dfshelper(G, s)
```

this is a generalized form of the algorithm which allows for both early (before visiting incident edges) and late (after visiting incident edges) operations

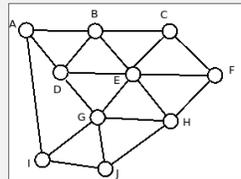
```
dfshelper(G, u)
process vertex u (early)
for each edge (u, v) in G.incidentEdges(u) do
    if state[v] = "undiscovered" then
        process edge (u, v)
        state[v] = "discovered" ← a vertex is discovered when an
                                incident (incoming) edge has
                                been visited – have found a path
                                from s to it
        prev[v] = u
        dfshelper(G, v)
state[u] = "processed" ← a vertex is processed when all of
                        its incident (outgoing) edges
                        have been visited – have found
                        everything reachable from it
process vertex u (late)
```

DFS

```
dfs(G, s)
for each vertex u in V-{\s} do
    state[u] = "undiscovered"
    prev[u] = null
state[s] = "discovered"
prev[s] = null
dfshelper(G, s)
```

```
dfshelper(G, u)
process vertex u (early)
for each edge (u, v) in G.incidentEdges(u) do
    if state[v] = "undiscovered" then
        process edge (u, v)
        state[v] = "discovered"
        prev[v] = u
        dfshelper(G, v)
state[u] = "processed"
process vertex u (late)
```

incidentEdges(u) determines what order the edges are visited in
the recursion keeps track of where the algorithm is in the sequence – execution continues when the call returns



Running Time of DFS

total $O(n+m)$ for adjacency list,
 $O(n^2)$ for adjacency matrix

```
dfs(G, s)
for each vertex u in V-{\s} do
    state[u] = "undiscovered"
    prev[u] = null
state[s] = "discovered"
prev[s] = null
dfshelper(G, s)
```

$O(n)$ with $O(n)$ traversal of vertices and $O(1)$ labeling

```
dfshelper(G, u)
process vertex u (early)
for each edge (u, v) in G.incidentEdges(u) do
    if state[v] = "undiscovered" then
        process edge (u, v)
        state[v] = "discovered"
        prev[v] = u
        dfshelper(G, v)
state[u] = "processed"
process vertex u (late)
```

incident edges is $O(\deg(u))$ for adjacency list, $O(n)$ for adjacency matrix
total is $O(m)$ for adjacency list (each edge is visited twice, once from each end) and $O(n^2)$ for adjacency matrix (get incident edges once per vertex)

Applications of DFS – Undirected Graphs

- **reachability**

- every vertex reachable from s will be discovered/processed during DFS

intuition – we follow every edge leaving each discovered vertex, and every vertex put in the stack is eventually removed and marked as processed

```

dfs(G,s)
for each vertex u in V-{s} do
    state[u] = "undiscovered"
    prev[u] = null
state[s] = "discovered"
prev[s] = null
dfshelper(G,s)

dfshelper(G,u)
process vertex u (early)
for each edge (u,v) in G.incidentEdges(u) do
    if state[v] = "undiscovered" then
        process edge (u,v)
        state[v] = "discovered"
        prev[v] = u
        dfshelper(G,v)
    state[u] = "processed"
process vertex u (late)
    
```

BFS/DFS Search Trees

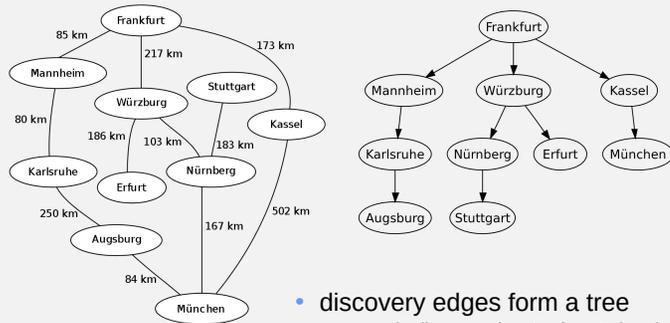
Classify each graph edge (u,v) as it is visited during traversal –

- *discovery* or *tree edges* – v is not already discovered
 - prev labels identify these edges
- *back edges* – v is an ancestor (other than the parent) of u
- *forward edges* – v is a descendant of u
- *cross edges* – v is not an ancestor or a descendant of u

Properties (undirected graphs) –

- discovery (tree) edges form a tree
- non-tree edges in BFS tree are cross edges connecting to the same level or one level higher in another branch
- non-tree edges in DFS tree are back edges

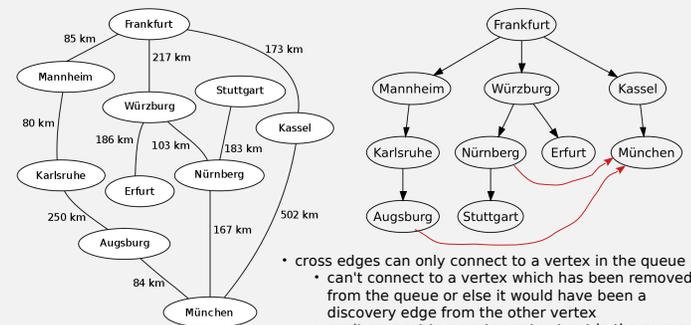
BFS/DFS Search Trees



- discovery edges form a tree
 - a newly-discovered vertex is not already part of the structure so it can't be involved in a cycle

BFS/DFS Search Trees

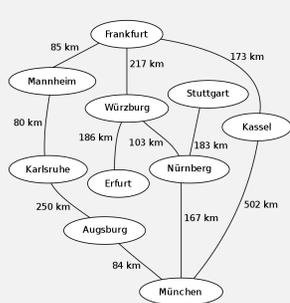
- non-tree edges in BFS tree are cross edges connecting to the same level or one lower in another branch



- cross edges can only connect to a vertex in the queue
 - can't connect to a vertex which has been removed from the queue or else it would have been a discovery edge from the other vertex
 - can't connect to a vertex not yet put in the queue or else would be a discovery edge from this vertex
- vertices in the queue at the same time can only be from adjacent levels

BFS/DFS Search Trees

- non-tree edges in DFS tree are back edges

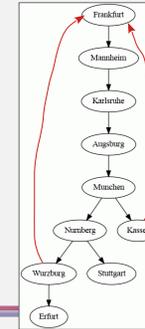
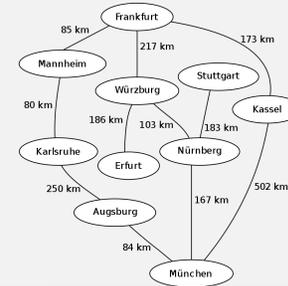


if (u,v) is a non-tree edge, v must be discovered but not processed...
 – if v has been processed, all of its incident edges have been visited and thus $(v,u) = (u,v)$ is a discovery edge
 ...meaning that v is an ancestor of $u \rightarrow (u,v)$ is a back edge
 – if v is the parent of u , $(v,u) = (u,v)$ is a discovery edge

Applications of DFS – Undirected Graphs

finding cycles

- back edge (u,v) forms a cycle consisting of the tree edges from v to u plus back edge (u,v)
- a graph is a tree if and only if there are no back edges



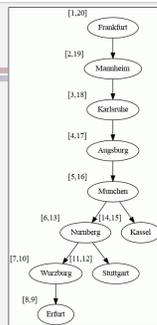
Entry and Exit Times

Recording entry and exit times –

- early process (before incident edges)
 - $time = time + 1$
 - $entry[v] = time$
- late process (after incident edges)
 - $time = time + 1$
 - $exit[v] = time$

Properties –

- the $[entry, exit]$ interval for v is properly nested within interval for ancestor u
 - entry times for ancestors of v are smaller than for v , while exit times are larger
- the number of descendants of v is $(exit[v] - entry[v]) / 2$
 - the $[entry, exit]$ interval for all of the descendants is properly nested within the interval for v – so there is both an entry and an exit for each
 - time is incremented once for each entry and once for each exit

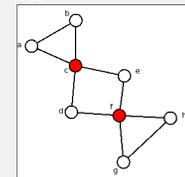


Applications of DFS – Undirected Graphs

articulation (cut) vertices

- a *cut vertex* is a vertex whose removal disconnects the graph (single point of failure)
- a *biconnected graph* has no cut vertices (at least two vertices must be removed to disconnect)
- observation – if a back edge connects a descendant of v with an ancestor of v , v is not a cut vertex
 - because the back edge forms a cycle

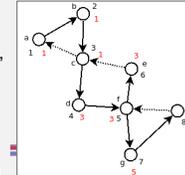
cut vertices marked in red



idea – for each vertex, determine its *earliest reachable ancestor* in the DFS search tree

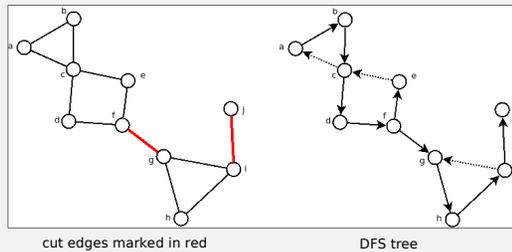
- number vertices in the order first encountered by DFS (entry time)
- earliest reachable ancestor = lowest-numbered of v , the vertices adjacent to v via back edges, and the earliest reachable ancestors of children of v
- v is a cut vertex if
 - the earliest reachable ancestor of at least one of v 's children is the child itself or v
 - if v is the root, it must also have two or more children

DFS tree – DFS entry order in black, earliest reachable ancestor in red



Applications of DFS – Undirected Graphs

- **bridges (cut edges)** – edges whose removal disconnects the graph
 - edge (u,v) is a cut edge if it is a tree edge and there's no back edge from v or a descendant of v to u or an ancestor of u

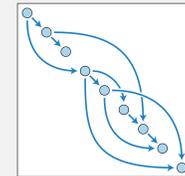


Applications of DFS – Directed Graphs

- **topological sort** – order the vertices of G so that all edges are oriented from an earlier vertex to a later one
 - possible if and only if G is a DAG (directed acyclic graph)
 - algorithm – the ordering is the reverse of the order in which vertex processing is completed (exit time) when dfs is started from a vertex s where $\text{indeg}(s) = 0$ (i.e. s has no incoming edges)

intuition –

- exit timestamp for u is after all of the outgoing incident edges (u,v) have been processed, which means u 's exit timestamp is after the exit timestamps of its adjacent vertices v and u occurs before v in the topological ordering
- edges are oriented (u,v) – u appears before v in the ordering so the edges are correctly oriented



Topological Sort

```
time = 0
for each vertex v in V do
  if G.inDegree(v) = 0
    dfs(G,v)
```

```
dfs(G,s)
for each vertex u in V-{s} do
  state[u] = "undiscovered"
  prev[u] = null
state[s] = "discovered"
prev[s] = null
dfsHelper(G,s)
```

```
dfsHelper(G,u)
```

```
time = time+1
```

```
entry[u] = time
```

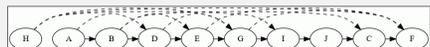
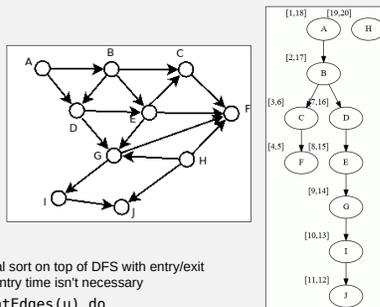
```
for each edge (u,v) in G.incidentEdges(u) do
  if state[v] = "undiscovered" && exit[v] is undefined then
    state[v] = "discovered"
```

```
prev[v] = u
dfsHelper(G,v)
state[u] = "processed"
```

```
time = time+1
exit[u] = time
```

can layer topological sort on top of DFS with entry/exit times but tracking entry time isn't necessary

an undefined exit time means that this vertex was also not reached on a previous dfs iteration (no need to continue with it if it has previously been reached)



Applications of DFS – Directed Graphs

- **is G strongly connected?** – *strongly connected* means a directed path exists between every pair of vertices
 - algorithm
 - $\text{dfs}(s)$, then reverse all of the edges of G and repeat $\text{dfs}(s)$ – G is strongly connected if the same set of vertices are discovered/processed each time

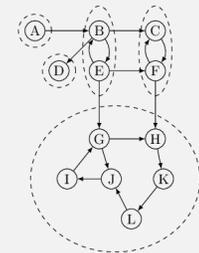
- **strongly connected components**

– an algorithm

- repeatedly compute the intersection of vertices reachable by $\text{dfs}(s)$ and by $\text{dfs}(s)$ with the graph's edges reversed, removing each set as a strongly connected component

– another algorithm

- repeatedly find a cycle and contract those vertices into a single vertex
- when there are no more cycles, each remaining vertex represents a different strongly connected component



Takeaways

- DFS algorithm
- DFS-based algorithms / applications
 - graph traversal
 - reachability
 - finding cycles (undirected graphs)
 - cut vertices (undirected graphs)
 - cut edges (undirected graphs)
 - topological sort (directed graphs)
 - strongly connected / strongly connected components (directed graphs)