Depth-First Search

Version of October 11, 2014

The Depth-First Search (DFS) Algorithm

What does Depth-First Search (DFS) do?

The Depth-First Search (DFS) Algorithm

What does Depth-First Search (DFS) do?

- Traverses all vertices in graph,

The Depth-First Search (DFS) Algorithm

What does Depth-First Search (DFS) do?

- Traverses all vertices in graph, and thereby
- Reveal properties of the graph.

The Depth-First Search (DFS) Algorithm

What does Depth-First Search (DFS) do?

- Traverses all vertices in graph, and thereby
- Reveal properties of the graph.

Four arrays are used to keep information gathered during traversal

The Depth-First Search (DFS) Algorithm

What does Depth-First Search (DFS) do?

- Traverses all vertices in graph, and thereby
- Reveal properties of the graph.

Four arrays are used to keep information gathered during traversal

1. *color*[*u*]: the color of each vertex visited
    - WHITE: undiscovered
    - GRAY: discovered but not finished processing
    - BLACK: finished processing

The Depth-First Search (DFS) Algorithm

What does Depth-First Search (DFS) do?

- Traverses all vertices in graph, and thereby
- Reveal properties of the graph.

Four arrays are used to keep information gathered during traversal

1. *color*[*u*]: the color of each vertex visited
   - WHITE: undiscovered
   - GRAY: discovered but not finished processing
   - BLACK: finished processing
2. *pred*[*u*]: predecessor pointer
   - pointing back to the vertex from which u was discovered

The Depth-First Search (DFS) Algorithm

What does Depth-First Search (DFS) do?

- Traverses all vertices in graph, and thereby
- Reveal properties of the graph.

Four arrays are used to keep information gathered during traversal

1. $color[u]$: the color of each vertex visited
   - WHITE: undiscovered
   - GRAY: discovered but not finished processing
   - BLACK: finished processing
2. $pred[u]$: predecessor pointer
   - pointing back to the vertex from which u was discovered
3. $d[u]$: discovery time
   - a counter indicating when vertex u is discovered

The Depth-First Search (DFS) Algorithm

What does Depth-First Search (DFS) do?

- Traverses all vertices in graph, and thereby
- Reveal properties of the graph.

Four arrays are used to keep information gathered during traversal

1. $color[u]$: the color of each vertex visited
   - WHITE: undiscovered
   - GRAY: discovered but not finished processing
   - BLACK: finished processing
2. $pred[u]$: predecessor pointer
   - pointing back to the vertex from which u was discovered
3. $d[u]$: discovery time
   - a counter indicating when vertex u is discovered
4. $f[u]$: finishing time
   - a counter indicating when the processing of vertex u (and all its descendants) is finished

The DFS Algorithm

How does DFS work?

The DFS Algorithm

How does DFS work?

- It starts from an initial vertex.

The DFS Algorithm

How does DFS work?

- It starts from an initial vertex.
- After visiting a vertex, it recursively visits *all* of its neighbors.
- The strategy is to search "deeper" in the graph whenever possible

DFS Algorithm

DFS(G)

```
// Initialize
foreach u in V do
    color[u] = WHITE; // undiscovered
    pred[u] = NULL; // no predecessor
end
time=
```

DFS Algorithm

DFS(G)

```
// Initialize
foreach u in V do
    color[u] = WHITE; // undiscovered
    pred[u] = NULL; // no predecessor
end
time= 0;
foreach u in V do
```

DFS Algorithm

DFS(G)

```
// Initialize
foreach u in V do
    color[u] = WHITE; // undiscovered
    pred[u] = NULL; // no predecessor
end
time= 0;
foreach u in V do
    // start a new tree
    if
```

DFS Algorithm

DFS(G)

```
// Initialize
foreach u in V do
    color[u] = WHITE; // undiscovered
    pred[u] = NULL; // no predecessor
end
time= 0;
foreach u in V do
    // start a new tree
    if color[u] =
```

DFS(G)

```
// Initialize
foreach u in V do
    color[u] = WHITE; // undiscovered
    pred[u] = NULL; // no predecessor
end
time= 0;
foreach u in V do
    // start a new tree
    if color[u] = WHITE then

    end
end
```

DFS(G)

```
// Initialize
foreach u in V do
    color[u] = WHITE; // undiscovered
    pred[u] = NULL; // no predecessor
end
time= 0;
foreach u in V do
    // start a new tree
    if color[u] = WHITE then
        DFSVisit(u);
    end
end
```

DFSVisit(u)

color[u] = ; `// u is discovered`

DFSVisit(u)

color[u] $= \mathrm{GRAY}$; `// u is discovered`

DFSVisit(u)

```
color[u] = GRAY; // u is discovered
d[u] = time=time+1; // u's discovery time
```

DFSVisit(u)

color[u] = GRAY; `// u is discovered`
d[u] = time=time+1; `// u's discovery time`
foreach *v in Adj(u)* **do**

DFSVisit(u)

```
color[u] = GRAY; // u is discovered
d[u] = time=time+1; // u's discovery time
foreach v in Adj(u) do
    // Visit undiscovered vertex
    if color[v] = WHITE then
        pred[v] = u;
```

DFSVisit(u)

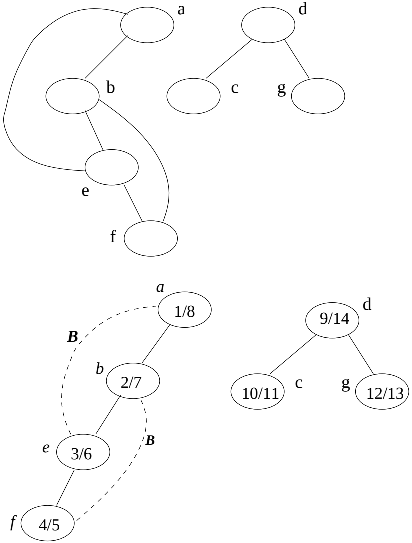
```
color[u] = GRAY; // u is discovered
d[u] = time=time+1; // u's discovery time
foreach v in Adj(u) do
    // Visit undiscovered vertex
    if color[v] = WHITE then
        pred[v] = u;
        DFSVisit(v);
    end
end
```

DFSVisit(u)

```
color[u] = GRAY; // u is discovered
d[u] = time=time+1; // u's discovery time
foreach v in Adj(u) do
    // Visit undiscovered vertex
    if color[v] = WHITE then
        pred[v] = u;
        DFSVisit(v);
    end
end
color[u] = BLACK; // u has finished
```

DFSVisit(u)

```
color[u] = GRAY; // u is discovered
d[u] = time=time+1; // u's discovery time
foreach v in Adj(u) do
    // Visit undiscovered vertex
    if color[v] = WHITE then
        pred[v] = u;
        DFSVisit(v);
    end
end
color[u] = BLACK; // u has finished
f[u] =time=time+1; // u's finish time
```


a
d
b
c
g
e
f
a
1/8
d
9/14
B
b
2/7
10/11
c
g
12/13
B
e
3/6
f
4/5

The DFS Algorithm

The outputs of DFS:

The DFS Algorithm

The outputs of DFS:

1. The time stamp arrays: $d[v], f[v]$

The DFS Algorithm

The outputs of DFS:

1. The time stamp arrays: $d[v], f[v]$
2. The predecessor array $pred[v]$

The outputs of DFS:

1. The time stamp arrays: $d[v], f[v]$
2. The predecessor array $pred[v]$

The DFS Forest:

The DFS Algorithm

The outputs of DFS:

1. The time stamp arrays: $d[v], f[v]$
2. The predecessor array $pred[v]$

The DFS Forest:

- Use $pred[v]$ to define a graph $F = (V, E_f)$ as follows:

$$E_f = \{(pred[v], v) | v \in V, pred[v] \neq \mathrm{NULL}\}$$

The DFS Algorithm

The outputs of DFS:

1. The time stamp arrays: $d[v], f[v]$
2. The predecessor array $pred[v]$

The DFS Forest:

- Use $pred[v]$ to define a graph $F = (V, E_f)$ as follows:

$$E_f = \{(pred[v], v) | v \in V, pred[v] \neq \mathrm{NULL}\}$$

- This is a graph with no cycles, and hence a forest, i.e. a collection of trees.
- Called a DFS Forest.
- Vertices in the subtree rooted at u are those discovered while u is gray.

Running Time of DFS

- The procedure DFSVisit is called exactly once for each vertex $u \in V$

Running Time of DFS

- The procedure DFSVisit is called exactly once for each vertex $u \in V$
  — since DFSVisit is invoked only on white vertices and the first thing it does is paint the vertex gray

Running Time of DFS

- The procedure DFSVisit is called exactly once for each vertex $u \in V$
  — since DFSVisit is invoked only on white vertices and the first thing it does is paint the vertex gray
- During an execution of DFSVisit(u),
  the for loop is executed $|\text{Adj}(u)| = \text{degree}(u)$ times

Running Time of DFS

- The procedure DFSVisit is called exactly once for each vertex $u \in V$
  — since DFSVisit is invoked only on white vertices and the first thing it does is paint the vertex gray
- During an execution of DFSVisit(u),
  the for loop is executed $|\text{Adj}(u)| = \text{degree}(u)$ times

On each vertex u, we spend time $T_u = O(1 + \text{degree}(u))$

- The procedure DFSVisit is called exactly once for each vertex $u \in V$
  — since DFSVisit is invoked only on white vertices and the first thing it does is paint the vertex gray
- During an execution of DFSVisit(u),
  the for loop is executed $|\text{Adj}(u)| = \text{degree}(u)$ times

On each vertex u, we spend time $T_u = O(1 + \text{degree}(u))$

The total running time is

$$\sum_{u \in V} T_u \leq \sum_{u \in V} (O(1 + \text{degree}(u))) = O(V + E)$$

Running Time of DFS

- The procedure DFSVisit is called exactly once for each vertex $u \in V$
  — since DFSVisit is invoked only on white vertices and the first thing it does is paint the vertex gray
- During an execution of DFSVisit(u),
  the for loop is executed $|\text{Adj}(u)| = \text{degree}(u)$ times

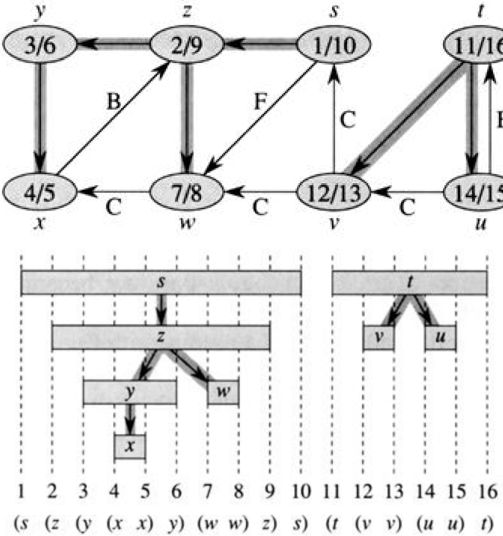
On each vertex u, we spend time $T_u = O(1 + \text{degree}(u))$

The total running time is

$$\sum_{u \in V} T_u \leq \sum_{u \in V} (O(1 + \text{degree}(u))) = O(V + E)$$

Hence, the running of DFS on a graph with V vertices and E edges is $O(V + E)$

Time-Stamp Structure



Return

Time-Stamp Structure...

- u is a descendant (in DFS trees) of v, if and only if $[d[u], f[u]]$ is a subinterval of $[d[v], f[v]]$ (Example)

Time-Stamp Structure...

- u is a descendant (in DFS trees) of v, if and only if $[d[u], f[u]]$ is a subinterval of $[d[v], f[v]]$ (Example)
- u is an ancestor of v, if and only if $[d[u], f[u]]$ contains $[d[v], f[v]]$ (Example)

Time-Stamp Structure...

- u is a descendant (in DFS trees) of v, if and only if $[d[u], f[u]]$ is a subinterval of $[d[v], f[v]]$ (Example)
- u is an ancestor of v, if and only if $[d[u], f[u]]$ contains $[d[v], f[v]]$ (Example)
- u is unrelated to v, if and only if $[d[u], f[u]]$ and $[d[v], f[v]]$ are disjoint intervals (Example)

Proof

The idea is to consider every case

Proof

The idea is to consider every case
We first consider $d[v] < d[u]$

Proof

The idea is to consider every case

We first consider $d[v] < d[u]$

1. If $f[v] > d[u]$,

Proof

The idea is to consider every case

We first consider $d[v] < d[u]$

1. If $f[v] > d[u]$, then
   - u is discovered when v is still not finished yet (marked gray)

Proof

The idea is to consider every case

We first consider $d[v] < d[u]$

1. If $f[v] > d[u]$, then
   - u is discovered when v is still not finished yet (marked gray)
     $\Rightarrow$ u is a descendant of v

Proof

The idea is to consider every case
We first consider $d[v] < d[u]$

1. If $f[v] > d[u]$, then
   - u is discovered when v is still not finished yet (marked gray) $\Rightarrow$ u is a descendant of v
   - u is discovered later than $v \Rightarrow u$ should finish before v

Proof

The idea is to consider every case

We first consider $d[v] < d[u]$

1. If $f[v] > d[u]$, then
    - u is discovered when v is still not finished yet (marked gray) $\Rightarrow$ u is a descendant of v
    - u is discovered later than $v \Rightarrow u$ should finish before v
    - Hence we have $[d[u], f[u]]$ is a subinterval of $[d[v], f[v]]$

Proof

The idea is to consider every case

We first consider $d[v] < d[u]$

1. If $f[v] > d[u]$, then
   - u is discovered when v is still not finished yet (marked gray) $\Rightarrow$ u is a descendant of v
   - u is discovered later than $v \Rightarrow u$ should finish before v
   - Hence we have $[d[u], f[u]]$ is a subinterval of $[d[v], f[v]]$
2. If $f[v] < d[u]$, then
   - obviously $[d[v], f[v]]$ and $[d[u], f[u]]$ are

Proof

The idea is to consider every case

We first consider $d[v] < d[u]$

1. If $f[v] > d[u]$, then
   - u is discovered when v is still not finished yet (marked gray) $\Rightarrow$ u is a descendant of v
   - u is discovered later than $v \Rightarrow u$ should finish before v
   - Hence we have $[d[u], f[u]]$ is a subinterval of $[d[v], f[v]]$
2. If $f[v] < d[u]$, then
   - obviously $[d[v], f[v]]$ and $[d[u], f[u]]$ are disjoint
   - It means that when u or v is discovered, the others are not marked gray

Proof

The idea is to consider every case

We first consider $d[v] < d[u]$

1. If $f[v] > d[u]$, then
   - u is discovered when v is still not finished yet (marked gray) $\Rightarrow$ u is a descendant of v
   - u is discovered later than $v \Rightarrow u$ should finish before v
   - Hence we have $[d[u], f[u]]$ is a subinterval of $[d[v], f[v]]$
2. If $f[v] < d[u]$, then
   - obviously $[d[v], f[v]]$ and $[d[u], f[u]]$ are disjoint
   - It means that when u or v is discovered, the others are not marked gray
   - Hence neither vertex is a descendant of the other

Proof

The idea is to consider every case

We first consider $d[v] < d[u]$

1. If $f[v] > d[u]$, then
   - u is discovered when v is still not finished yet (marked gray) $\Rightarrow$ u is a descendant of v
   - u is discovered later than $v \Rightarrow u$ should finish before v
   - Hence we have $[d[u], f[u]]$ is a subinterval of $[d[v], f[v]]$
2. If $f[v] < d[u]$, then
   - obviously $[d[v], f[v]]$ and $[d[u], f[u]]$ are disjoint
   - It means that when u or v is discovered, the others are not marked gray
   - Hence neither vertex is a descendant of the other

The argument for other case, where $d[v] > d[u]$, is similar.

Tree Structure

- Undirected graph $G = (V, E)$, DFS forest $F = (V, E_f)$

Tree Structure

- Undirected graph $G = (V, E)$, DFS forest $F = (V, E_f)$
- Consider $(u, v) \in E$
  - tree edge: if $(u, v) \in E_f$

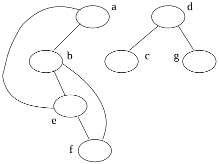
Tree Structure

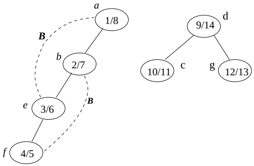
- Undirected graph $G = (V, E)$, DFS forest $F = (V, E_f)$
- Consider $(u, v) \in E$
  - tree edge: if $(u, v) \in E_f$ or equivalently $u = pred[v]$, i.e. u is the predecessor of v in the DFS tree

Tree Structure

- Undirected graph $G = (V, E)$, DFS forest $F = (V, E_f)$
- Consider $(u, v) \in E$
  - tree edge: if $(u, v) \in E_f$ or equivalently $u = pred[v]$, i.e. u is the predecessor of v in the DFS tree
  - back edge:

- Undirected graph $G = (V, E)$, DFS forest $F = (V, E_f)$
- Consider $(u, v) \in E$
  - tree edge: if $(u, v) \in E_f$ or equivalently $u = pred[v]$, i.e. u is the predecessor of v in the DFS tree
  - back edge: if v is an ancestor (excluding predecessor) of u in the DFS tree





Tree Structure

Theorem

An edge in an undirected graph is either a tree edge or a back edge.

Tree Structure

Theorem

An edge in an undirected graph is either a tree edge or a back edge.

Proof:

- Let (u, v) be an arbitrary edge in an undirected graph G.
- Without loss of generality, assume $d(u) < d(v)$.

Theorem

An edge in an undirected graph is either a tree edge or a back edge.

Proof:

- Let (u, v) be an arbitrary edge in an undirected graph G.
- Without loss of generality, assume $d(u) < d(v)$.
- Then v is discovered while u is gray (why?).

Tree Structure

Theorem

An edge in an undirected graph is either a tree edge or a back edge.

Proof:

- Let (u, v) be an arbitrary edge in an undirected graph G.
- Without loss of generality, assume $d(u) < d(v)$.
- Then v is discovered while u is gray (why?).
- Hence v is in the DFS subtree rooted at u.

Tree Structure

Theorem

An edge in an undirected graph is either a tree edge or a back edge.

Proof:

- Let (u, v) be an arbitrary edge in an undirected graph G.
- Without loss of generality, assume $d(u) < d(v)$.
- Then v is discovered while u is gray (why?).
- Hence v is in the DFS subtree rooted at u.
    - If $pred[v] = u$, then (u, v) is a tree edge.

Tree Structure

Theorem

An edge in an undirected graph is either a tree edge or a back edge.

Proof:

- Let (u, v) be an arbitrary edge in an undirected graph G.
- Without loss of generality, assume $d(u) < d(v)$.
- Then v is discovered while u is gray (why?).
- Hence v is in the DFS subtree rooted at u.
  - If $pred[v] = u$, then (u, v) is a tree edge.
  - if $prev[v] \neq u$, then (u, v) is a back edge.

An Application of DFS: Cycle Finding

Question

Given an undirected graph G, how to determine whether or not G contains a cycle?

Lemma

G is acyclic if and only if a DFS of G yields no back edges.

Proof.

$\Rightarrow$: Suppose that there is a back edge (u, v). Then, vertex v is an ancestor (excluding predecessor) of u in the DFS trees. There is thus a path from v to u in G, and the back edge (u, v) completes a cycle.

$\Leftarrow$: If there is no back edge, then since an edge in an undirected graph is either a tree edge or a back edge, there are only tree edges, implying that the graph is a forest, and hence is acyclic. $\square$

Cycle(G)

```
foreach u in V do
    color[u] = WHITE;
    pred[u] = NULL;
end
foreach u in V do
    if color[u] = WHITE then
        Visit(u);
    end
end
output "No Cycle";
```

Visit(u)

```
color[u] = GRAY;
foreach v in Adj(u) do
    // consider (u, v)
    if color[v] = WHITE then
        // v unvisited
        pred[v] = u;
        Visit(v); // visit v
    else if v != pred[u] then
        // back edge detected
        output "Cycle found!";
        exit; // terminate
    end
end
color[u] = BLACK;
```

Running time: $O(V)$

- only traverse tree edges, until the first back edge is found
- at most $V - 1$ tree edges