Depth First Search (DFS)

Part I

Depth First Search (DFS)

Today

Two topics:
- Structure of directed graphs
- DFS and its properties
- One application of DFS to obtain fast algorithms

DFS special case of Basic Search.

DFS is useful in understanding graph structure.

DFS used to obtain linear time ($O(m + n)$) algorithms for
- Finding cut-edges and cut-vertices of undirected graphs
- Finding strong connected components of directed graphs
- Linear time algorithm for testing whether a graph is planar

...many other applications as well.
DFS in Undirected Graphs

Recursive version. Easier to understand some properties.

\[
\text{DFS}(G) \quad \text{DFS}(u)
\]

for all \( u \in V(G) \) do

Mark \( u \) as unvisited
Set pred\((u)\) to null

T is set to \( \emptyset \)

while \( \exists \) unvisited \( u \) do

DFS\((u)\)

Output \( T \)

Implementated using a global array \textit{Visited} for all recursive calls. \( T \) is the search tree/forest.

Example

Edges classified into two types: \( uv \in E \) is a

1. tree edge: belongs to \( T \)
2. non-tree edge: does not belong to \( T \)

Properties of tree

Proposition

- \( T \) is a forest
- connected components of \( T \) are same as those of \( G \).
- If \( uv \in E \) is a non-tree edge then, in \( T \), either:
  - \( u \) is an ancestor of \( v \), or
  - \( v \) is an ancestor of \( u \).

Question: Why are there no cross-edges?


**Example**

<table>
<thead>
<tr>
<th>vertex</th>
<th>[pre, post]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[1,</td>
</tr>
<tr>
<td>1</td>
<td>[1, 16]</td>
</tr>
<tr>
<td>2</td>
<td>[2, ]</td>
</tr>
<tr>
<td>2</td>
<td>[2, 15]</td>
</tr>
<tr>
<td>4</td>
<td>[3, 14]</td>
</tr>
<tr>
<td>5</td>
<td>[14, 17]</td>
</tr>
</tbody>
</table>

**pre and post numbers**

Node $u$ is **active** in time interval $[\text{pre}(u), \text{post}(u)]$

**Proposition**

For any two nodes $u$ and $v$, the two intervals $[\text{pre}(u), \text{post}(u)]$ and $[\text{pre}(v), \text{post}(v)]$ are disjoint or one is contained in the other.

**Proof.**

- Assume without loss of generality that $\text{pre}(u) < \text{pre}(v)$. Then $v$ visited after $u$.
- If $\text{DFS}(v)$ invoked before $\text{DFS}(u)$ finished, $\text{post}(v) < \text{post}(u)$.
- If $\text{DFS}(v)$ invoked after $\text{DFS}(u)$ finished, $\text{pre}(v) > \text{post}(u)$.

**pre and post numbers useful in several applications of DFS**

**in Directed Graphs**

$\text{DFS}(G)$

- Mark all nodes $u$ as unvisited
  - $T$ is set to $\emptyset$
  - $time = 0$
- while there is an unvisited node $u$ do
  - $\text{DFS}(u)$
- Output $T$

$\text{DFS}(u)$

- Mark $u$ as visited
  - $\text{pre}(u) = ++time$
  - for each edge $(u, v)$ in $\text{Out}(u)$ do
    - if $v$ is not visited
      - add edge $(u, v)$ to $T$
      - $\text{DFS}(v)$
  - $\text{post}(u) = ++time$

**Example**

- $A$  
  - $B$  
    - $C$  
      - $D$  
        - $E$  
          - $F$  
            - $G$  
              - $H$  
                - $I$
          - $[2, 11]$  
    - $[1, 16]$  
- $[2, 15]$  
- $[3, 10]$  
- $[6, 7]$  
- $[4, 5]$  
- $[8, 9]$  
- $[12, 15]$  
- $[13, 14]$  
- $[17, 20]$  

DFS Properties

Generalizing ideas from undirected graphs:

- **DFS(G)** takes \( O(m + n) \) time.
- Edges added form a *branching*: a forest of out-trees. Output of **DFS(G)** depends on the order in which vertices are considered.
- If \( u \) is the first vertex considered by **DFS(G)** then **DFS(u)** outputs a directed out-tree \( T \) rooted at \( u \) and a vertex \( v \) is in \( T \) if and only if \( v \in \text{rch}(u) \).
- For any two vertices \( x, y \) the intervals \([\text{pre}(x), \text{post}(x)]\) and \([\text{pre}(y), \text{post}(y)]\) are either disjoint or one is contained in the other.

**Note:** Not obvious whether **DFS(G)** is useful in directed graphs but it is.

Tree

Edges of \( G \) can be classified with respect to the **DFS** tree \( T \) as:

- **Tree edges** that belong to \( T \)
- A *forward edge* is a non-tree edges \((x, y)\) such that \( \text{pre}(x) < \text{pre}(y) < \text{post}(y) < \text{post}(x) \).
- A *backward edge* is a non-tree edge \((y, x)\) such that \( \text{pre}(x) < \text{pre}(y) < \text{post}(y) < \text{post}(x) \).
- A *cross edge* is a non-tree edges \((x, y)\) such that the intervals \([\text{pre}(x), \text{post}(x)]\) and \([\text{pre}(y), \text{post}(y)]\) are disjoint.

Types of Edges

![Types of Edges Diagram]

Cycles in graphs

**Question:** Given an *undirected* graph how do we check whether it has a cycle and output one if it has one?

**Question:** Given an *directed* graph how do we check whether it has a cycle and output one if it has one?
Using DFS... to check for Acyclicity and compute Topological Ordering

**Question**
Given G, is it a **DAG**? If it is, generate a topological sort. Else output a cycle C.

**DFS based algorithm:**
- Compute **DFS(G)**
- If there is a back edge \(e = (v, u)\) then G is not a **DAG**. Output cycle C formed by path from u to v in T plus edge \((v, u)\).
- Otherwise output nodes in decreasing post-visit order. **Note:** no need to sort, **DFS(G)** can output nodes in this order.

Algorithm runs in \(O(n + m)\) time.
Correctness is not so obvious. See next two propositions.

**Back edge and Cycles**

**Proposition**
G has a cycle iff there is a back-edge in **DFS(G)**.

**Proof.**
If: \((u, v)\) is a back edge implies there is a cycle C consisting of the path from v to u in **DFS** search tree and the edge \((u, v)\).

Only if: Suppose there is a cycle C = \(v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k \rightarrow v_1\). Let \(v_i\) be first node in C visited in **DFS**.
All other nodes in C are descendants of \(v_i\) since they are reachable from \(v_i\).
Therefore, \((v_{i-1}, v_i)\) (or \((v_k, v_1)\) if \(i = 1\)) is a back edge.

**Proof**
**Proposition**
If G is a **DAG** and \(\text{post}(v) > \text{post}(u)\), then \((u, v)\) is not in G.

**Proof.**
Assume \(\text{post}(v) > \text{post}(u)\) and \((u, v)\) is an edge in G. We derive a contradiction. One of two cases holds from DFS property.
- Case 1: \([\text{pre}(u), \text{post}(u)]\) is contained in \([\text{pre}(v), \text{post}(v)]\).
  - Implies that u is explored during **DFS(v)** and hence is a descendent of v. Edge \((u, v)\) implies a cycle in G but G is assumed to be DAG!
- Case 2: \([\text{pre}(u), \text{post}(u)]\) is disjoint from \([\text{pre}(v), \text{post}(v)]\).
  - This cannot happen since v would be explored from u.

**Example**

![Graph Example](https://example.com/graph.png)
Part II

Strong connected components

Algorithmic Problem
Find all SCCs of a given directed graph.

Previous lecture:
Saw an $O(n \cdot (n + m))$ time algorithm.
This lecture: sketch of a $O(n + m)$ time algorithm.

Graph of SCCs $G$: B, E, F, A, C, D

Meta-graph of SCCs
Let $S_1, S_2, \ldots, S_k$ be the strong connected components (i.e., SCCs) of $G$. The graph of SCCs is $G^{SCC}$

- Vertices are $S_1, S_2, \ldots, S_k$
- There is an edge $(S_i, S_j)$ if there is some $u \in S_i$ and $v \in S_j$ such that $(u, v)$ is an edge in $G$.

Reversal and SCCs

Proposition
For any graph $G$, the graph of SCCs of $G^{rev}$ is the same as the reversal of $G^{SCC}$.

Proof. Exercise.
**SCCs and DAGs**

**Proposition**

For any graph $G$, the graph $G^{SCC}$ has no directed cycle.

**Proof.**

If $G^{SCC}$ has a cycle $S_1, S_2, \ldots, S_k$ then $S_1 \cup S_2 \cup \cdots \cup S_k$ should be in the same SCC in $G$. Formal details: exercise.

---

**Directed Acyclic Graphs**

**Definition**

A directed graph $G$ is a directed acyclic graph (DAG) if there is no directed cycle in $G$. 

---

**Is this a DAG?**
Sources and Sinks

**Definition**
- A vertex $u$ is a **source** if it has no in-coming edges.
- A vertex $u$ is a **sink** if it has no out-going edges.

Simple Properties

**Proposition**
Every DAG $G$ has at least one source and at least one sink.

**Proof.**
Let $P = v_1, v_2, \ldots, v_k$ be a longest path in $G$. Claim that $v_1$ is a source and $v_k$ is a sink. Suppose not. Then $v_1$ has an incoming edge which either creates a cycle or a longer path both of which are contradictions. Similarly if $v_k$ has an outgoing edge. 

---

Topological Ordering/Sorting

**Definition**
A **topological ordering** of $G = (V, E)$ is an ordering $\prec$ on $V$ such that if $(u, v) \in E$ then $u \prec v$.

Informal equivalent definition:
One can order the vertices of the graph along a line (say the $x$-axis) such that all edges are from left to right.

---

DAGs and Topological Sort

**Lemma**
A directed graph $G$ can be topologically ordered iff it is a DAG.

Need to show both directions.
**Lemma**

A directed graph $G$ can be topologically ordered if it is a DAG.

**Proof.**
Consider the following algorithm:
- Pick a source $u$, output it.
- Remove $u$ and all edges out of $u$.
- Repeat until graph is empty.

Exercise: prove this gives topological sort.

Exercise: show algorithm can be implemented in $O(m + n)$ time.

**Note:** A DAG $G$ may have many different topological sorts.

**Question:** What is a DAG with the most number of distinct topological sorts for a given number $n$ of vertices?

**Question:** What is a DAG with the least number of distinct topological sorts for a given number $n$ of vertices?
Cycles in graphs

**Question:** Given an *undirected* graph how do we check whether it has a cycle and output one if it has one?

**Question:** Given an *directed* graph how do we check whether it has a cycle and output one if it has one?

---

**To Remember: Structure of Graphs**

**Undirected graph:** connected components of $G = (V, E)$ partition $V$ and can be computed in $O(m + n)$ time.

**Directed graph:** the meta-graph $G^{SCC}$ of $G$ can be computed in $O(m + n)$ time. $G^{SCC}$ gives information on the partition of $V$ into strong connected components and how they form a DAG structure.

Above structural decomposition will be useful in several algorithms

---

**Finding all s of a Directed Graph**

**Problem**

Given a directed graph $G = (V, E)$, output all its strong connected components.

**Straightforward algorithm:**
Mark all vertices in $V$ as not visited.
for each vertex $u \in V$ not visited yet do
find $SCC(G, u)$ the strong component of $u$:
Compute $rch(G, u)$ using $DFS(G, u)$
Compute $rch(G^{rev}, u)$ using $DFS(G^{rev}, u)$
$SCC(G, u) \leftarrow rch(G, u) \cap rch(G^{rev}, u)$
$\forall u \in SCC(G, u)$: Mark $u$ as visited.

Running time: $O(n(n + m))$
Is there an $O(n + m)$ time algorithm?
Structure of a Directed Graph

Graph G

Graph of SCCs \( G^{\text{SCC}} \)

Reminder

\( G^{\text{SCC}} \) is created by collapsing every strong connected component to a single vertex.

Proposition

For a directed graph G, its meta-graph \( G^{\text{SCC}} \) is a DAG.

Big Challenge(s)

How do we find a vertex in a sink SCC of \( G^{\text{SCC}} \)?

Can we obtain an implicit topological sort of \( G^{\text{SCC}} \) without computing \( G^{\text{SCC}} \)?

Answer: \( \text{DFS}(G) \) gives some information!

Linear-time Algorithm for \( \text{SCC}s: \) Ideas

Exploit structure of meta-graph...

Wishful Thinking Algorithm

- Let \( u \) be a vertex in a sink SCC of \( G^{\text{SCC}} \)
- Do \( \text{DFS}(u) \) to compute \( \text{SCC}(u) \)
- Remove \( \text{SCC}(u) \) and repeat

Justification

- \( \text{DFS}(u) \) only visits vertices (and edges) in \( \text{SCC}(u) \)
- ... since there are no edges coming out a sink!
- \( \text{DFS}(u) \) takes time proportional to size of \( \text{SCC}(u) \)
- Therefore, total time \( O(n + m) \)!

Linear Time Algorithm

...for computing the strong connected components in G

\[ \text{do } \text{DFS}(G^{\text{rev}}) \text{ and output vertices in decreasing post order.} \]
\[ \text{Mark all nodes as unvisited} \]
\[ \text{for each } u \text{ in the computed order do} \]
\[ \text{if } u \text{ is not visited then} \]
\[ \text{DFS}(u) \]
\[ \text{Let } S_u \text{ be the nodes reached by } u \]
\[ \text{Output } S_u \text{ as a strong connected component} \]
\[ \text{Remove } S_u \text{ from } G \]

Theorem

Algorithm runs in time \( O(m + n) \) and correctly outputs all the SCCs of \( G \).
Linear Time Algorithm: An Example - Initial steps

Graph $G$:

Reverse graph $G^{\text{rev}}$:

$DFS$ of reverse graph:

Pre/Post $DFS$ numbering of reverse graph:

Linear Time Algorithm: An Example

Removing connected components: 1

Original graph $G$ with rev post numbers:

$DFS$ from vertex $G$

$SCC$ computed: $\{G\}$

Removing connected components: 2

$DFS$ from vertex $H$, remove it.

$SCC$ computed: $\{G\}, \{H\}$

Removing connected components: 3

$DFS$ from vertex $B$, remove it.

$SCC$ computed: $\{G\}, \{H\}, \{F, B, E\}$
Linear Time Algorithm: An Example

Removing connected components: 4
Do DFS from vertex \( F \)
Remove visited vertices: \( \{ F, B, E \} \).

SCC computed: 
\( \{ G \}, \{ H \}, \{ F, B, E \} \)

Exercise:
Given all the strong connected components of a directed graph \( G = (V, E) \) show that the meta-graph \( G^{SCC} \) can be obtained in \( O(m + n) \) time.

Obtaining the meta-graph...
Once the strong connected components are computed.

Exercise:
Given all the strong connected components of a directed graph \( G = (V, E) \) show that the meta-graph \( G^{SCC} \) can be obtained in \( O(m + n) \) time.

SCC computed:
\( \{ G \}, \{ H \}, \{ F, B, E \}, \{ A, C, D \} \)
Which is the correct answer!

Solving Problems on Directed Graphs

A template for a class of problems on directed graphs:
- Is the problem solvable when \( G \) is strongly connected?
- Is the problem solvable when \( G \) is a DAG?
- If the above two are feasible then is the problem solvable in a general directed graph \( G \) by considering the meta graph \( G^{SCC} \)?
Part V

An Application to make

make Utility [Feldman]

- Unix utility for automatically building large software applications
- A makefile specifies
  - Object files to be created,
  - Source/object files to be used in creation, and
  - How to create them

An Example makefile

```bash
project:  main.o utils.o command.o
c -o project main.o utils.o command.o

main.o:  main.c defs.h
c -c main.c
utils.o:  utils.c defs.h command.h
c -c utils.c
command.o:  command.c defs.h command.h
c -c command.c
```
makefile as a Digraph

- main.c → main.o
- utils.c → utils.o
- defs.h → project
- command.h → command.o
- command.c

Computational Problems for make

- Is the makefile reasonable?
- If it is reasonable, in what order should the object files be created?
- If it is not reasonable, provide helpful debugging information.
- If some file is modified, find the fewest compilations needed to make application consistent.

Algorithms for make

- Is the makefile reasonable? Is G a DAG?
- If it is reasonable, in what order should the object files be created? Find a topological sort of a DAG.
- If it is not reasonable, provide helpful debugging information. Output a cycle. More generally, output all strong connected components.
- If some file is modified, find the fewest compilations needed to make application consistent.
  - Find all vertices reachable (using DFS/BFS) from modified files in directed graph, and recompile them in proper order. Verify that one can find the files to recompile and the ordering in linear time.

Take away Points

- Given a directed graph G, its SCCs and the associated acyclic meta-graph $G^{SCC}$ give a structural decomposition of G that should be kept in mind.
- There is a DFS based linear time algorithm to compute all the SCCs and the meta-graph. Properties of DFS crucial for the algorithm.
- DAGs arise in many application and topological sort is a key property in algorithm design. Linear time algorithms to compute a topological sort (there can be many possible orderings so not unique).