Today

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

Strong Connected Components (SCCs)

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

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

- 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 $G$. 

Meta-graph of SCCs

Graph $G$

Graph of SCCs $G^{\text{SCC}}$
Reversal and SCCs

Proposition

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

Proof.

Exercise.

SCCs and DAGs

Proposition

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

Proof.

If $G^{\text{SCC}}$ 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 $G$. 
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.

- $G$ is a DAG if and only if $G^{rev}$ is a DAG.
- $G$ is a DAG if and only each node is in its own strong connected component.

Formal proofs: exercise.

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:
1. Pick a source $u$, output it.
2. Remove $u$ and all edges out of $u$.
3. 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

---

Part II

Depth First Search (DFS)

**Depth First Search**

DFS is a special case of Basic Search but is a versatile graph exploration strategy. John Hopcroft and Bob Tarjan (Turing Award winners) demonstrated the power of DFS to understand graph structure. DFS can be 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.

```latex
DFS(G) ----> DFS(u) 
for all \( u \in V(G) \) do 
  Mark \( u \) as unvisited 
  Set \( \text{pred}(u) \) to null 
  \( T \) is set to \( \emptyset \) 
while \( \exists \) unvisited \( u \) do 
  DFS(u) 
Output T
```

Implemented using a global array \( \text{Visited} \) for all recursive calls.

\( T \) is the search tree/forest.

Example

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

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

Properties of \( T \)

- \( T \) 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?

DFS with Visit Times

Keep track of when nodes are visited.

```latex
DFS(G) ----> DFS(u) 
for all \( u \in V(G) \) do 
  Mark \( u \) as unvisited 
  \( T \) is set to \( \emptyset \) 
  \( \text{pre}(u) = +\text{time} \) 
while \( \exists \) unvisited \( u \) do 
  DFS(u) 
Output T
```

```latex
DFS(G) ----> DFS(u) 
for all \( u \in V(G) \) do 
  Mark \( u \) as visited 
  \( \text{time} = 0 \) 
while \( \exists \) unvisited \( u \) do 
  DFS(u) 
Output T
```

```latex
DFS(G) ----> DFS(u) 
for all \( u \in V(G) \) do 
  Mark \( u \) as visited 
  \( \text{pre}(u) = +\text{time} \) 
while \( \exists \) unvisited \( u \) do 
  DFS(u) 
Output T
```

```latex
DFS(G) ----> DFS(u) 
for all \( u \in V(G) \) do 
  Mark \( u \) as visited 
  \( \text{time} = 0 \) 
while \( \exists \) unvisited \( u \) do 
  DFS(u) 
Output T
```

```latex
DFS(G) ----> DFS(u) 
for all \( u \in V(G) \) do 
  Mark \( u \) as visited 
  \( \text{pre}(u) = +\text{time} \) 
while \( \exists \) unvisited \( u \) do 
  DFS(u) 
Output T
```
**Example**

![Graph Example](image)

**in Directed Graphs**

**DFS**

Mark all nodes as unvisited

\[ T = \emptyset \]

while there is an unvisited node do

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

Output \( T \)

**DFS**

Mark as visited

\[ \text{pre}(u) = ++\text{time} \]

for each edge \((u, v)\) in \(\text{Out}(u)\) do

if \(v\) is not visited

add edge \((u, v)\) to \(T\)

\[ \text{DFS}(v) \]

\[ \text{post}(u) = ++\text{time} \]

---

**pre and post numbers**

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

**Proposition**

For any two nodes \(u, v\), 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**

---

**DFS Properties**

Generalizing ideas from undirected graphs:

- **DFS** takes \(O(m + n)\) time.
- Edges added form a branching: a forest of out-trees. Output of **DFS** depends on the order in which vertices are considered.
- If \(u\) is the first vertex considered by **DFS** then **DFS** 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** is useful in dir graphs but it is.
**Types of Edges**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cross</td>
<td>Forward</td>
</tr>
<tr>
<td>Backward</td>
<td></td>
</tr>
</tbody>
</table>

**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?

---

**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.

---

**Using**

... 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 $\text{DFS}(G)$
- 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, $\text{DFS}(G)$ 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 = v₁ → v₂ → ... → vₖ → v₁. Let vᵢ be first node in C visited in DFS. All other nodes in C are descendants of vᵢ since they are reachable from vᵢ. Therefore, (vᵢ₋₁, vᵢ) (or (vₖ, v₁) if i = 1) is a back edge.

Proof

Proposition

If G is a DAG and post(v) > post(u), then (u, v) is not in G.

Proof.

Assume post(v) > post(u) and (u, v) is an edge in G. We derive a contradiction. One of two cases holds from DFS property.

- Case 1: [pre(u), post(u)] is contained in [pre(v), 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: [pre(u), post(u)] is disjoint from [pre(v), post(v)]. This cannot happen since v would be explored from u.

Example

![Graph Diagram]

Part III

Linear time algorithm for finding all strong connected components of a directed graph
Finding all SCCs 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 $G^{SCC}$

Reminder
$G^{SCC}$ is created by collapsing every strong connected component to a single vertex.

Proposition
For a directed graph $G$, its meta-graph $G^{SCC}$ is a DAG.

Big Challenge(s)
How do we find a vertex in a sink SCC of $G^{SCC}$?
Can we obtain an implicit topological sort of $G^{SCC}$ without computing $G^{SCC}$?

Answer: $DFS(G)$ gives some information!
Linear Time Algorithm

...for computing the strong connected components in $G$

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

Theorem
Algorithm runs in time $O(m + n)$ and correctly outputs all the SCCs of $G$.

Linear Time Algorithm: An Example
Removing connected components: 1

Original graph $G$ with rev post numbers:

```
G 16
FE 11
B C 6
D 12
H 10
A 5
```

Do DFS from vertex $G$ remove it.

```
G 16
FE 11
B C 6
D 12
H 10
A 5
```

SCC computed: \{G\}

Linear Time Algorithm: An Example
Removing connected components: 2

```
G 16
FE 11
B C 6
D 12
H 10
A 5
```

Do DFS from vertex $G$ remove it.

```
G 16
FE 11
B C 6
D 12
H 10
A 5
```

SCC computed: \{G\}

```
G 16
FE 11
B C 6
D 12
H 10
A 5
```

Do DFS from vertex $H$, remove it.

```
G 16
FE 11
B C 6
D 12
H 10
A 5
```

SCC computed: \{G\}, \{H\}
Linear Time Algorithm: An Example

Removing connected components: 3

Do **DFS** from vertex \( H \), remove it.

Removing connected components: 4

Do **DFS** from vertex \( B \)
Remove visited vertices:
\[ \{ F, B, E \} \]

SCC computed:
\[ \{ G \}, \{ H \} \]

SCC computed:
\[ \{ G \}, \{ H \}, \{ F, B, E \} \]

Linear Time Algorithm: An Example

Final result

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 IV

An Application to make

Make/Makefile

(A) I know what make/makefile is.
(B) I do NOT know what make/makefile is.

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

```
project: main.o utils.o command.o
    cc -o project main.o utils.o command.o

main.o: main.c defs.h
    cc -c main.c

utils.o: utils.c defs.h command.h
    cc -c utils.c

command.o: command.c defs.h command.h
    cc -c command.c
```

### Computational Problems for make

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

### Algorithms for make

1. Is the makefile reasonable? Is G a DAG?
2. If it is reasonable, in what order should the object files be created? Find a topological sort of a DAG.
3. If it is not reasonable, provide helpful debugging information. Output a cycle. More generally, output all strong connected components.
4. If some file is modified, find the fewest compilations needed to make the 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

1. Given a directed graph $G$, its SCCs and the associated acyclic meta-graph $G^{\text{SCC}}$ give a structural decomposition of $G$ that should be kept in mind.

2. There is a DFS based linear time algorithm to compute all the SCCs and the meta-graph. Properties of DFS crucial for the algorithm.

3. 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).