CS 341: ALGORITHMS

Trevor Brown

trevor.brown@uwaterloo.ca

DC 2338, Office hour M3-4pm
THIS TIME

• Strong connectedness
  • Algorithm using DFS
• Strongly connected components
  • Sharir’s algorithm (using DFS)
STRONG CONNECTEDNESS
Testing existence of all-to-all paths
STRONG CONNECTEDNESS

- In a directed graph,
  - \( v \) is reachable from \( w \) if there is a path from \( w \) to \( v \)
  - we denote this path \( w \rightarrow v \)
- A graph \( G \) is strongly connected iff every node is reachable from every other node
  - More formally: \( \forall_{w,v} \ w \rightarrow v \)
**STRONG CONNECTEDNESS**

- Is this graph **strongly connected**?

  No path from c to other nodes.

- How about this one?

  Yes. One big cycle.
**STRONG CONNECTEDNESS**

- How about this graph?

- How about this one?

Yes. Multiple intersecting cycles.

No. Two cycles with only a one-directional path between them.
APPLICATIONS OF CHECKING STRONG CONNECTEDNESS

- You gain some symmetry from knowing a graph is strongly connected

- For example, you can start a graph traversal at any node, and know the traversal will reach every node

- Without strong connectedness, if you want to run a graph traversal that reaches every node in a single pass, you would have to do additional processing to determine an appropriate starting node
APPLICATIONS OF CHECKING STRONG CONNECTEDNESS

• Useful as a sanity check!
• Suppose you want to run an algorithm that requires strong connectedness, and you believe your input graph is strongly connected
• Validate your input by testing whether this is true!
• Subtle, difficult-to-detect bugs often result if such an algorithm is run only on one component of a graph
• [More concrete applications once we generalize and talk about strongly connected components…]
STRONG CONNECTEDNESS

- Lemma: a graph is strongly connected
- iff for every node $s$,
- all nodes are reachable from $s$;
- and $s$ is reachable from all nodes

Prove both directions:

$(\Rightarrow)$ Suppose for all $u, v$ we have $u \rightarrow \rightarrow v$. Fix any $s$. Node $s$ is reachable from all nodes, and vice versa.

$(\Leftarrow)$ Suppose $s$ is reachable from all nodes and vice versa. For any $u, v$, we have $u \rightarrow \rightarrow s \rightarrow \rightarrow v$, and $v \rightarrow \rightarrow s \rightarrow \rightarrow u$. 
**STRONG CONNECTEDNESS**

- How to use DFS to determine whether **every node is reachable** from a given node \( s \)?
- How to use DFS to determine whether **\( s \) is reachable** from every node?

DFS from \( s \) and see if every node turns black

What if we first **reverse** the direction of every edge?

Then \( s \rightarrow v \) in this new graph IFF \( v \rightarrow s \) in the original graph

DFS from \( s \)
• IsStronglyConnected($G = \{V, E\}$) where $V = v_1, v_2, ..., v_n$
  • $(colour, d, f) := DFSVisit(v_1, G)$
  • for $i := 1..n$
    • if $\text{colour}[v_i] \neq black$ then return $false$
  • Construct graph $H$ by \textbf{reversing} all edges in $G$
  • $(colour, d, f) := DFSVisit(v_1, H)$
  • for $i := 1..n$
    • if $\text{colour}[v_i] \neq black$ then return $false$
  • return $true$
REVERSING EDGES: ADJACENCY MATRIX

reverse all edges

target

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

source

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
REVERSING EDGES: ADJACENCY MATRIX

reverse all edges
REVERSING EDGES: ADJACENCY MATRIX

reverse all edges

source

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

target

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

reverse all edges
REVERSING EDGES: ADJACENCY MATRIX

reverse all edges

source

target
REVERSING EDGES: ADJACENCY MATRIX

reverse all edges
REVERSING EDGES: ADJACENCY MATRIX

reverse all edges
Reversing Edges: Adjacency Matrix

Reverse all edges
REVERSING EDGES: ADJACENCY MATRIX

reverse all edges

source

target

a | b | c | d | e | f | g
---|---|---|---|---|---|---
a | 1 |   |   |   |   |   |
b |   | 1 |   |   |   |   |   |
c | 1 | 1 | 1 |   |   |   |   |
d | 1 |   |   | 1 |   |   |   |
e |   |   |   |   | 1 |   |   |
f |   |   |   |   |   | 1 |   |
g |   |   |   |   |   |   | 1

reverse all edges
REVERSING EDGES: ADJACENCY MATRIX

Source

Target

reverse all edges
Can do matrix transpose, or can just swap variables for source & target in your code!

Complexity?

REVERSING EDGES: ADJACENCY MATRIX

Source:  
```
<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Target:  
```
<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

$$M_E$$

$$\left(M_E\right)^T$$
REVERSING EDGES: ADJACENCY LISTS

reverse edges

source

Complexity?

target

1. TransposeLists(adj[1..n])
2. newAdj = new array of n lists
3. for u = 1 .. n
4.     for v in adj[u]
5.         newAdj[v].insert(u)
6. return newAdj
• \textit{IsStronglyConnected}(G = \{V, E\}) where V = v_1, v_2, \ldots, v_n
  • \((colour, d, f) := DFSVisit(v_1, G)\)
  • for \(i := 1..n\)
    • if \(colour[v_i] \neq black\) then return \text{false}
  • Construct graph \(H\) by \textbf{reversing} all edges in \(G\)
  • \((colour, d, f) := DFSVisit(v_1, H)\)
  • for \(i := 1..n\)
    • if \(colour[v_i] \neq black\) then return \text{false}
  • return \text{true}

Complexity for adjacency lists? \(O(n + m)\)
Every node is black. Next step!

DFSVisit(a) in G
(a is arbitrary)
Every node is black. Next step!

**EXAMPLE EXECUTION 1**

Construct graph $H$

$DFSVisit(a)$ in $G$
($a$ is arbitrary)

Every node is black. Next step!

$DFSVisit(a)$ in $H$

Every node is black. So $G$ is strongly connected!
Every node is black. Next step!

Could the result change if we started at a different node?

construct graph $H$

$DFSVisit(a)$ in $G$
($a$ is arbitrary)

Every node is black. Next step!

$DFSVisit(a)$ in $H$

Some nodes are not black

No path from those nodes to $a$

So $G$ is not strongly connected!
STRONGLY CONNECTED COMPONENTS (SCC)

Graphs that are not strongly connected can be divided into components that are
These are called **strongly connected components (SCCs)**.

- This graph could be divided into **two graphs** that are each strongly connected.
STRONGLY CONNECTED COMPONENTS

- It could also be divided into **three graphs**...

  ![Diagram](image)

  - **Maximal, so SCC**
  - **Not maximal, so not SCC**

- But we want our SCCs to be **maximal** (as large as possible)
STRONGLY CONNECTED COMPONENTS

- So, the goal is to find these (maximal) SCCs:
For two vertices \( x \) and \( y \) of \( G \), define \( x \sim y \) if \( x = y \); or if \( x \neq y \) and there exist directed paths from \( x \) to \( y \) and from \( y \) to \( x \).

The relation \( \sim \) is an equivalence relation.

The strongly connected components of \( G \) are the equivalence classes of vertices defined by the relation \( \sim \).

A strongly connected component of a digraph \( G \) is a maximal strongly connected subgraph of \( G \).

Note: a connected component can contain just a single node.

Example: a node with no out-edges.
Consider this graph. These are its SCCs:

- The following is its component graph:
  - a, b, c, d
  - f, e, g
  - h, i
  - i, j, k
  - l

It has one node for each SCC.

And an edge between two nodes IFF there is an edge between the corresponding SCCs.
APPLICATIONS OF SCCs AND COMPONENT GRAPHS

- Finding **all cyclic** dependencies in code
- Can find **single** cycle with an earlier DFS-based algorithm
- But it is nicer to find all cycles at once, so you don’t have to fix one to expose another
APPLICATIONS OF SCCs AND COMPONENT GRAPHS

• Finding SCCs in a social network graph
  • Yields information about communities that have formed
  • Social Networks can study the evolution of those communities
APPLICATIONS OF SCCs AND COMPONENT GRAPHS

- Explicit model checking in formal verification
- In model checking, we have a state machine, which represents the model of our soft-/hardware, and we try to prove that temporal logic formulas hold over it
- Idea: for a set of program states, prove bad things cannot happen in those states
- Example: CTL formula $\text{EG} (p)$ can be verified by finding SCCs & checking paths to them
APPLICATIONS OF SCCs AND COMPONENT GRAPHS

• Studying **connectome graphs**
• These are graphs used for modelling **nervous systems**
• In such graphs vertices correspond to cells and edges correspond to physical cell contacts or synapses
• SCCs seem to have an important role in brain connectome graphs
• For example, a 2016 paper found that a fly brain connectome had **one** large SCC w/785 nodes (neurons), and other SCCs had only 1-2 nodes
• Similar structure in rat and cat brain samples. Motivates studying why!
APPLICATIONS OF SCCs AND COMPONENT GRAPHS

- **Data filtering** before running other algorithms
- Consider Google maps; Nodes = intersections, edges = roads
- Don’t want to run path finding alg. on entire **global** graph!
- First restrict execution to a rectangle
- Then throw away everything except the strongly connected component that contains the source & target
APPLICATIONS OF SCCs AND COMPONENT GRAPHS

• Solving **2-satisfiability**
  • Conjunctive normal form Boolean formulae with constraints on pairs of variables, e.g., \( f(x_1, x_2, x_3) = (x_1 \lor x_2) \land (\neg x_2 \lor x_3) \land (x_3 \lor \neg x_1) \)
  • Problem: is there an assignment of \((x_1, x_2, x_3)\) that makes this formula **true**?

• 2-satisfiability can solve many problems!
  • Arranging text labels (or other objects) in a diagram to avoid overlap, if each has two possible positions
  • Routing wires that can only bend once in a VLSI integrated circuit design
SO THE PROBLEM IS IMPORTANT...

• How do we solve it?
• What if we run DFS, then reverse all edges, then run DFS (like checking whether an entire graph is strongly connected?)
What if we run DFS, then **reverse all edges, then run DFS**?

- **DFSVisit(a)**
- **DFSVisit(h)**
- **DFSVisit(j)**

Reverse edges

We **did** visit everything we saw in our **DFSVisit(h)** above! But we also saw **more** nodes: j, k, l

Not clear how to identify remaining SCCs...

What if we perform our **DFSVisit** calls in a different order?
• What if we run DFS, then reverse all edges, then run DFS?

\[\text{DFSVisit}(a) \quad \text{DFSVisit}(h) \quad \text{DFSVisit}(j)\]

\[\text{DFSVisit}(j) \quad \text{DFSVisit}(h) \quad \text{DFSVisit}(a)\]

Problem: from e, we can reach other SCCs
Idea: perform same DFSVisit calls, but in reverse order

Identified three SCCs correctly, but clearly need more DFSVisit calls!
What if we run DFS, then reverse all edges, then run DFS?

\[ \text{DFSVisit}(a) \quad \text{DFSVisit}(h) \quad \text{DFSVisit}(j) \]

\[ \text{DFSVisit}(j) \quad \text{DFSVisit}(h) \quad \text{DFSVisit}(a) \quad \text{DFSVisit}(e) \]

Idea: call DFSVisit on all nodes in order of largest to smallest finishing time

Identified all SCCs correctly! Why?
By the DFS-based topological sort algorithm we saw earlier, we only visit nodes in the current SCC! So, we can identify precisely what is in this SCC.

**REASONING ABOUT WHY THIS WORKS**

- We DFS visit nodes by largest to smallest finishing time.
  - Recall this would be a topological order in a DAG.
  - But there may be cycles in each component!
- We prove the component graph is a DAG.
- And we prove our DFS visits the component graph in a topological order.
- So, the idea is: we perform DFS on SCCs one-by-one, in a topological order, which guarantees that any other SCCs reachable from the current DFS must already be finished/black (so won’t be visited).
FACT: COMPONENT GRAPH IS A **DAG**

- Proof by contradiction
  - Suppose there is a cycle $v_1, v_2, \ldots, v_k$
  - Consider any pair of nodes $u, v$ in any of the corresponding components
  - Suppose $u$ and $v$ are located in different components $C_u \neq C_v$
  - Then one can navigate from $u$ to $v$ by moving from $C_u$ to $C_v$ in the component graph (moving freely within each component)
  - But then $C_u$ and $C_v$ are part of the **same SCC**---a contradiction!
**PROVING DECREASING FINISH TIMES INDUCE A TOPOLOGICAL ORDER ON THE COMPONENT GRAPH**

- **Definition:** For a strongly connected component $C$, let $d[C] = \min\{d[v] : v \in C\}$ and $f[C] = \max\{f[v] : v \in C\}$

- **Lemma:** if $C_i, C_j$ are SCCs and there is an edge $C_i \rightarrow C_j$, then $f[C_i] > f[C_j]$

- **Proof.** Case 1 ($d[C_i] < d[C_j]$):
  - Let $u$ be the earliest discovered node in $C_i$
  - All nodes in $C_i \cup C_j$ are white-reachable from $u$, so they are descendants in the DFS forest and finish before $u$
  - So $f[C_i] > f[C_j]$
**PROVING DECREASING FINISH TIMES INDUCE A TOPOLOGICAL ORDER ON THE COMPONENT GRAPH**

- **Definition:** For a strongly connected component $C$, let $d[C] = \min\{d[v] : v \in C\}$ and $f[C] = \max\{f[v] : v \in C\}$

- **Lemma:** if $C_i, C_j$ are SCCs and there is an edge $C_i \rightarrow C_j$, then $f[C_i] > f[C_j]$

- **Proof.** Case 2 ($d[C_i] > d[C_j]$):
  - Since component graph is a DAG, there is **no edge** $C_j \rightarrow C_i$
  - Thus, **no nodes** in $C_i$ are reachable from $C_j$
  - So we discover $C_j$ and finish $C_j$ **without** discovering $C_i$
  - Therefore $d[C_j] < f[C_j] < d[C_i] < f[C_i]$. QED
Using the Lemma to Build an Algorithm

- **Lemma**: if $C_i, C_j$ are SCCs and there is an edge $C_i \rightarrow C_j$, then $f[C_i] > f[C_j]$

- **Algorithm**:
  - $(v_{i_1}, v_{i_2}, ..., v_{i_n}) = DFS\_topsort(G)$
  - $H :=$ construct by reversing each edge in $G$
  - return := $DFS\_SCC(H, (v_{i_1}, v_{i_2}, ..., v_{i_n}))$

Calls DFSVisit on nodes in topological order, and gives each node an SCC number in a component[] array, which is then returned.

This is called Sharir’s algorithm (sometimes Kosaraju’s algorithm).

**This paper** first introduced it.
**PSEUDOCODE**

Assume that $f[v_i_1] > f[v_i_2] > \cdots > f[v_i_n]$.

**Algorithm: DFS** ($H, (v_i_1, v_i_2, \ldots, v_i_n)$)

\[
\text{for } j \leftarrow 1 \text{ to } n \\
\begin{align*}
\text{do } colour[v_{i_j}] & \leftarrow \text{white} \\
\text{sec} & \leftarrow 0 \\
\text{for } j \leftarrow 1 \text{ to } n \\
\begin{align*}
\text{if } colour[v_{i_j}] = \text{white} & \\
\text{then } \\
\text{do } & \\
\text{sec} & \leftarrow \text{sec} + 1 \\
\text{DFSvisit}(H, v_{i_j}, \text{sec}) & \\
\text{return } (\text{comp})
\end{align*}
\end{align*}
\]

**Algorithm:** $\text{DFSvisit}(H, v, \text{sec})$

\[
\begin{align*}
\text{colour}[v] & \leftarrow \text{gray} \\
\text{comp}[v] & \leftarrow \text{sec} \\
\text{for each } w \in \text{Adj}[v] \\
\begin{align*}
\text{do } & \\
\text{if } \text{colour}[w] = \text{white} & \\
\text{then } \text{DFSvisit}(H, w, \text{sec}) & \\
\text{colour}[v] & \leftarrow \text{black}
\end{align*}
\end{align*}
\]

Complexity? $O(n + m)$
Proof of Correctness of Sharir’s Algorithm

First, note that $G$ and $H$ have the same strongly connected components. Let $u = v_{i_1}$ be the first vertex visited in step 3. Let $C$ be the s.c.c. containing $u$ and let $C'$ be any other s.c.c. 

$f(C) > f(C')$, so there is no edge from $C'$ to $C$ in $G$ (by the Lemma). Therefore there is no edge from $C$ to $C'$ in $H$.

Hence no vertex in $C'$ is reachable from $u$ in $H$.

Therefore, $DFSvisit(u)$ explores the vertices in $C$ (and only those vertices); this forms one DFS tree in $H$.

Next, $DFSvisit(v_{i_2})$ explores the vertices in the s.c.c. containing $v_{i_2}$, etc.

Every time we make an initial call to $DFSvisit$, we are exploring a new s.c.c.

We increment $scc$, which is used to label the various s.c.c. $comp[v]$ denotes the label of the s.c.c. containing $v$. 