Graph searches: BFS and DFS

Pierre Aboulker - pierreaboulker@gmail.com
1 - Graph Search
Generic Graph Search

- Searching a graph means: visit the vertices following the edges of the graph.
- It works the same for non-oriented and oriented graphs.
- Searching a graph often gives you some information on the structure of the graphs.
- A search also output an ordering on the vertices.
Graph search

Goal 1: Given a graph $G$ and a source vertex\footnote{or starting vertex} $s$, a graph search explore all vertices that are reachable from $s$, that is vertices $u$ such that there is a path linking $s$ and $u$.

\begin{algorithm}
\begin{algorithmic}
\State \bf{Initialization:} mark $s$ as explored, the rest as unexplored
\While{Possible}
\State Choose an edge $uv$ such that $u$ is explored and $v$ is unexplored
\State Mark $v$ as unexplored
\EndWhile
\end{algorithmic}
\end{algorithm}
Graph search

**Goal 1:** Given a graph $G$ and a source vertex $s$, a graph search explore all vertices that are reachable from $s$, that is vertices $u$ such that there is a path linking $s$ and $u$.

---

**Algorithm 2** $\text{GENERIC-SEARCH}(G = (V, E), s)$

1: Initialization: mark $s$ as explored, the rest as unexplored
2: while Possible do
3: Choose an edge $uv$ such that $u$ is explored and $v$ is unexplored
4: Mark $v$ as unexplored

---

Idea of correctness: at the end of the algorithm, no edge has one explored extremity and one unexplored extremity

Idea for running time: don’t explore an edge twice (in order to have linear time later)
Proof of correctness

We want to prove that

A vertex \( v \) is explored \( \iff \) \( v \) is reachable from \( s \)
Proof of correctness

We want to prove that

\[ \text{A vertex } v \text{ is explored } \iff v \text{ is reachable from } s \]

- **Explored } \implies \text{Reachable:}**
Proof of correctness

We want to prove that

A vertex $v$ is explored $\iff$ $v$ is reachable from $s$

- **Explored $\Rightarrow$ Reachable:**
  - Let $\{s = v_0, v_1, \ldots, v_{n_s}\}$ be the explored vertices, and assume $v_i$ has been explored before $v_j$ whenever $i < j$.
  - Assume for contradiction that some of them are not reachable.
  - Let $i$ be the minimum subscript such that $v_i$ is not reachable.
  - Hence, at the moment where $v_i$ is explored, it must have a neighbor $v_k \in \{v_0, \ldots, v_{i-1}\}$.
  - By minimality of $i$, $v_k$ is reachable from $v_0$, so $v_i$ too:

$$v_0 \leadsto v_k \lor v_k v_i$$
Proof of correctness

- **Reachable** ⇒ **explored**:
Proof of correctness

• **Reachable ⇒ Explored:**
  - Assume for contradiction that there exists \( x \) that is reachable and not explored.
  - Choose such an \( x \) as near from \( s \) as possible (minimize \( \text{dist}(s, x) \)).
  - Let \( P = s x_1 \ldots x_k x \) be a shortest \((s, x)\)-path.
  - By minimality of \( d(s, x) \), \( x_k \) has been explored.
  - Hence \( x_k x \) is an edge such that \( x_k \) is explored and \( x \) is not, contradiction.
What do you mean by "non-generic" search?

**Algorithm 3** \texttt{GENERIC-SEARCH}(G = (V, E), s)

1: Initially \(s\) is explored, the rest is unexplored
2: \textbf{while} Possible \textbf{do}
3: Choose an edge \(uv\) such that \(u\) is explored and \(v\) is unexplored
4: Mark \(v\) as unexplored

In this generic search, there is a lot of choices when it comes to decide the next edge to follow. There is several ways to break the ties, each giving different kind of informations on the graph.
Breadth First Search (BFS)

- The distance \(d(u, v)\) between two vertices \(u\) and \(v\) is the length of a shortest path linking \(u\) and \(v\). It is \(\infty\) if no such path exists\(^2\).
- BFS starts at a vertex \(s\), then explored all vertices at distance 1 from \(s\), then all vertices at distance 2, then at distance 3 etc etc.

\(^{2}\text{it is the case if and only if } u \text{ and } v \text{ are in distinct connected components}\)
A few words about BFS

- **INPUT:** a graph $G = (V, E)$ and a source vertex $s$.
- **GOAL:** visit every vertex reachable from $s$.

**Functionning:**
- BFS visits all vertices at distance 1 from $s$, then vertices at distance 2, then at distance 3 etc etc
- The algorithm works on both directed and undirected graphs.

**By-product:**
- It computes the distances from $s$ to all other vertices.
- It produces a **BFS tree** rooted at $s$ that contains all reachable vertices:
  - For every $u$ reachable from $s$, the unique $su$-path of the BFS tree corresponds to a shortest $su$-path of $G$.
  - The BFS-tree is a spanning tree of the connected component containing $u$ (often denoted by $C_u$).
Pseudo-code for BFS: use a queue

- An unexplored vertex is white,
- An explored vertex is gray or black.
- It is black if all its neighbors have been explored.

Algorithm 4 BFS($G$, start vertex $s$)

1: for each $u \in V \setminus \{s\}$ do
2: \hspace{1em} $u$.color $\leftarrow$ White
3: s.color $\leftarrow$ Gray
4: Enqueue($Q$, s)   \Comment{Queue initialized with s}
5: while $Q \neq \emptyset$ do
6: \hspace{1em} $u = \text{Dequeue}(Q)$
7: for each $v \in Adj[u]$ do
8: \hspace{2em} if $v$.color == White then
9: \hspace{3em} $v$.color $\leftarrow$ Gray
10: \hspace{3em} Enqueue($Q$, $v$)
11: \hspace{1em} $u$.color $\leftarrow$ Black
BFS Execution (pics from CLRS)

Graph searches: BFS and DFS
Attributes and their implementations

Most algorithms that operate on graphs need to maintain attributes for vertices and/or edges. Like `.color` in BFS.

- There is no one best way to implement vertex and edge attributes.
Attributes and their implementations

Most algorithms that operate on graphs need to maintain attributes for vertices and/or edges. Like .\textit{color} in BFS.

- There is no one best way to implement vertex and edge attributes.
- For a given situation, your decision will likely depend on the programming language you are using, the algorithm you are implementing, and how the rest of your program uses the graph.
Attributes and their implementations

Most algorithms that operate on graphs need to maintain attributes for vertices and/or edges. Like `.color` in BFS.

- There is no one best way to implement vertex and edge attributes.
- For a given situation, your decision will likely depend on the programming language you are using, the algorithm you are implementing, and how the rest of your program uses the graph.

1. Represent vertex attributes in an additional arrays, like: `color[1, \ldots, n]`. In this case, "u.d" would actually be stored in the array entry `color[u]`. 
Attributes and their implementations

Most algorithms that operate on graphs need to maintain attributes for vertices and/or edges. Like `.color` in BFS.

- There is no one best way to implement vertex and edge attributes.
- For a given situation, your decision will likely depend on the programming language you are using, the algorithm you are implementing, and how the rest of your program uses the graph.

1. Represent vertex attributes in an additional arrays, like: `color[1, \ldots, n]`. In this case, "u.d" would actually be stored in the array entry `color[u]`.
2. In an object-oriented programming language, vertex attributes might be represented as instance variables within a subclass of a Vertex class.
First observations on BFS

**Algorithm 5** BFS($G$, start vertex $s$)

1: $s$.color $\leftarrow$ Gray
2: **for** each $u \in V \setminus \{s\}$ **do**
3: \quad $u$.color $\leftarrow$ White
4: \quad $Q \leftarrow s$
5: **while** $Q \neq \emptyset$ **do**
6: \quad $u = \text{DEQUEUE}(Q)$
7: \quad **for** each $v \in \text{Adj}[u]$ **do**
8: \quad \hspace{1em} **if** $v$.color == White **then**
9: \quad \hspace{2em} $v$.color $\leftarrow$ Gray
10: \quad \hspace{2em} $\text{ENQUEUE}(Q, v)$
11: \quad $u$.color $\leftarrow$ Black

Observations:

- Only unexplored vertex can enter the Queue
- All vertices in the Queue are gray.
- $u$ is blackened $\Rightarrow$ neighbors of $u$ are explored (gray or black)
Correctness: a vertex $v$ is explored if and only if $v$ is reachable from $s$.

Proof: it is a special case of the generic search.

Running time: $O(n_s + m_s)$ where $n_s$ and $m_s$ are respectively the number of vertices and the number of edges of the connected component containing $s$.

Reason: Each edge is looked twice and $\sum_{v \in V} d(v) = 2m$. 
Application of BFS: shortest paths

**Goal:** compute \( \text{dist}(s, v) \) for every vertex \( v \).

- \( u.\text{dist} \) is the distance from \( s \) to \( u \).

---

**Algorithm 6**\( \text{BFS}(G, \text{start vertex } s) \)

1. \( s.\text{color} \leftarrow \text{Gray}, \ s.\text{dist} \leftarrow 0 \)
2. \textbf{for} each \( u \in V \setminus \{s\} \) \textbf{do}
3. \( u.\text{color} \leftarrow \text{White}, \ u.\text{dist} = \infty \)
4. \( Q \leftarrow s \)
5. \textbf{while} \( Q \neq \emptyset \) \textbf{do}
6. \( u = \text{DEQUEUE}(Q) \)
7. \textbf{for} each \( v \in \text{Adj}[u] \) \textbf{do}
8. \( \textbf{if } v.\text{color} == \text{White} \textbf{ then} \)
9. \( v.\text{color} \leftarrow \text{Gray} \)
10. \( v.\text{dist} \leftarrow u.\text{dist} + 1 \)
11. \( \text{ENQUEUE}(Q, v) \)
12. \( u.\text{color} \leftarrow \text{Black} \)
Lemma on shortest path and distances

**Lemma:** Let $G$ be a graph and $s \in V$. Then for every edge $uv$, $\text{dist}(s, v) \leq \text{dist}(s, u) + 1$

**Proof:**

**Case 1:** $u$ is reachable from $s$,
- Let $P$ be a $(s, u)$-path of length $\text{dist}(s, u)$.
- Then $P$ followed by the edge $uv$ is a $sv$-path.
- Its length is at most $\text{dist}(s, u) + 1$.

**Case 2:** $u$ is not reachable from $s$,
- Then $v$ is not reachable from $s$.
- Hence: $\text{dist}(s, u) = \text{dist}(s, v) = \infty$
Computing both distances and the BFS tree

- \( u.\pi \) is the parent of \( u \) in the BFS-tree.

**Algorithm 7** BFS(G, start vertex \( s \))

1: \( s.color \leftarrow \text{Gray} \), \( s.dist \leftarrow 0 \) and \( s.\pi \leftarrow \text{NIL} \)
2: for each \( u \in V \setminus \{s\} \) do
3: \( u.color \leftarrow \text{White} \), \( u.dist = \infty \) and \( s.\pi \leftarrow \text{NIL} \)
4: \( Q \leftarrow s \)
5: while \( Q \neq \emptyset \) do
6: \( u = \text{DEQUEUE}(Q) \)
7: for each \( v \in \text{Adj}[u] \) do
8: if \( v.color == \text{White} \) then
9: \( v.color \leftarrow \text{Gray} \)
10: \( v.\pi \leftarrow u \)
11: \( v.dist \leftarrow u.dist + 1 \)
12: \( \text{ENQUEUE}(Q, v) \)
13: \( u.color \leftarrow \text{Black} \)

BFS-tree: \( G_{\pi} = (V_{\pi}, E_{\pi}) : \)
- \( V_{\pi} = \{u : u.pi \neq \text{NIL}\} \)
- \( E_{\pi} = \{u.piu : u \in V_{\pi} \setminus \{s\}\} \)
Print shortest paths

The following procedure prints out the vertices on a shortest path from \( s \) to \( u \), assuming that a BFS-tree of \( G \) has already been computed:

<table>
<thead>
<tr>
<th>Algorithm 8</th>
<th>Print-Path ((G, s, u))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>if ( s == u ) then</td>
</tr>
<tr>
<td>2:</td>
<td>print( (s) )</td>
</tr>
<tr>
<td>3:</td>
<td>else if ( v.pi == NIL )</td>
</tr>
<tr>
<td>4:</td>
<td>print( (&quot;s and u are not connected.&quot;&quot;) )</td>
</tr>
<tr>
<td>5:</td>
<td>else</td>
</tr>
<tr>
<td>6:</td>
<td>Print-Path ((G, s, v.\pi))</td>
</tr>
<tr>
<td>7:</td>
<td>print( (v) )</td>
</tr>
</tbody>
</table>
BFS Application 1: connected components

**Def:** two vertices \( u \) and \( v \) are connected if there is a path linking them.

**Goal:** Preprocess graph to answer queries of the form: "is \( v \) connected to \( w \)" in constant time.

**Solution:** use BFS (or any search) as a subroutine.
BFS Application 1: connected components

The relation "is connected to" is an equivalence relation:

- Reflexive: $v$ is connected to $v$.
- Symmetric: if $v$ is connected to $w$, then $w$ is connected to $v$.
- Transitive: if $v$ connected to $w$ and $w$ connected to $x$, then $v$ connected to $x$.

Def. A connected component is a maximal set of connected vertices.
Paul Erdős (1913-1996)
Proofs from the book

“You don’t have to believe in God, but you should believe in The Book.”

Paul Erdős
BFS Application 2: Erdős Number

Paul Erdős wrote more than 1500 articles. He had many collaborators which drived people to define the Erdős number:

- The Erdős number of Erdős is 0.
- If you have a paper with Erdős you have Erdős number 1.
- If you don’t have a paper with Erdős but you have a paper with someone that has a paper with Erdős, then you have Erdős number 2.
- etc etc
BFS Application 2: Erdős Number

Paul Erdős wrote more than 1500 articles. He had many collaborators which drived people to define the Erdős number:

- The Erdős number of Erdős is 0.
- If you have a paper with Erdős you have Erdős number 1.
- If you don’t have a paper with Erdős but you have a paper with someone that has a paper with Erdős, then you have Erdős number 2.
- etc etc

Modelisation: A graph $G = (V, E)$ where:

- $V$ is the set of science researchers.
- Two researchers are adjacent if they wrote a paper together.
- Erdős number is the distance from Erdős.
BFS Application 2: Erdős Number

Paul Erdős wrote more than 1500 articles. He had many collaborators which drived people to define the Erdős number:

- The Erdős number of Erdős is 0.
- If you have a paper with Erdős you have Erdős number 1.
- If you don’t have a paper with Erdős but you have a paper with someone that has a paper with Erdős, then you have Erdős number 2.
- etc etc

Modelisation: A graph $G = (V, E)$ where:

- $V$ is the set of science researchers.
- Two researchers are adjacent if they wrote a paper together.
- Erdős number is the distance from Erdős.

Solution: run BFS with source vertex Erdős
hand-drawing of part of the Erdős graph by Ron Graham
Depth First Search (DFS)

- DFS is an aggressive search, it backtracks only when no other choice is available.
- It is a special kind of search, so it takes a graph $G$ and a source vertex $s$ as input, and visits all vertices reachable from $s$.
- It has several applications on directed graphs:
  - Computes topological ordering of Directed Acyclic Graph (DAG)
  - Computes strong connected components of directed graphs
- It does all that in linear time.
Preparing for a date:

What situations might I prepare for?
1) Medical emergency
2) Dancing
3) Food too expensive

Okay, what kinds of emergencies can happen?
1) A) Snakebite
   B) Lightning strike
   C) Fall from chair
2) Marine stinger

Hm. Which snakes are dangerous? Let's see...
1) A) Corn snake
    B) Garter snake
    C) Copperhead

The research comparing snake venoms is scattered and inconsistent. I'll make a spreadsheet to organize it.

I'm here to pick you up. You're not dressed?
By loc, the inland taipan has the deadliest venom of any snake!

I really need to stop using depth-first searches.
Definitions for directed graphs

- **Digraph**: like a graph but edges are directed.

- A vertex \( v \) has two kinds of neighbors:
  - **Out-neighborhood**: \( N^+(v) \) out-degree: \( d^+(v) \),
  - **In-neighborhood**: \( N^-(v) \) in-degree: \( d^-(v) \).

- **Directed path**: path with all edges in the same direction.
- **Directed cycle**: cycle with all edges in the same direction.
- \( u \) is **reachable** from \( v \) if there is a directed path from \( u \) to \( v \).

**Property**: \[
\sum_{u \in V} d^+(u) = \sum_{u \in V} d^-(u) = |E|
\]
Example of modelisation: road network

Vertices are the intersections, Edges are the one-way street.
# Digraphs applications

<table>
<thead>
<tr>
<th>digraph</th>
<th>vertex</th>
<th>directed edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>transportation</td>
<td>street intersection</td>
<td>one-way street</td>
</tr>
<tr>
<td>web</td>
<td>web page</td>
<td>hyperlink</td>
</tr>
<tr>
<td>food web</td>
<td>species</td>
<td>predator-prey relationship</td>
</tr>
<tr>
<td>WordNet</td>
<td>synset</td>
<td>hypernym</td>
</tr>
<tr>
<td>scheduling</td>
<td>task</td>
<td>precedence constraint</td>
</tr>
<tr>
<td>financial</td>
<td>bank</td>
<td>transaction</td>
</tr>
<tr>
<td>cell phone</td>
<td>person</td>
<td>placed call</td>
</tr>
<tr>
<td>infectious disease</td>
<td>person</td>
<td>infection</td>
</tr>
<tr>
<td>game</td>
<td>board position</td>
<td>legal move</td>
</tr>
<tr>
<td>citation</td>
<td>journal article</td>
<td>citation</td>
</tr>
<tr>
<td>object graph</td>
<td>object</td>
<td>pointer</td>
</tr>
<tr>
<td>inheritance hierarchy</td>
<td>class</td>
<td>inherits from</td>
</tr>
<tr>
<td>control flow</td>
<td>code block</td>
<td>jump</td>
</tr>
</tbody>
</table>

Picture from Kevin Wayne
Some digraphs problems

- **Path.** Is there a directed path from $s$ to $t$?
- **Induced path.** Is there a directed induced path between $s$ and $t$?
- **Shortest path.** What is the shortest directed path from $s$ to $t$?
- **Topological sort.** Can you draw a digraph so that all edges point upwards?
- **Strong connectivity.** Is there a directed path between all pairs of vertices?
- **Transitive closure.** For every pair of vertices $u$, $v$, and the edge from $u$ to $v$ if there is a directed path from $u$ to $v$.
- **PageRank.** What is the importance of a web page?
Digraph representation: adjacency list

Maintain vertex-indexed array of lists containing the out-neighbors of each vertex.
## Compare digraph representations

<table>
<thead>
<tr>
<th>representation</th>
<th>space</th>
<th>insert edge from v to w</th>
<th>edge from v to w?</th>
<th>iterate over vertices pointing from v?</th>
</tr>
</thead>
<tbody>
<tr>
<td>list of edges</td>
<td>E</td>
<td>1</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>adjacency matrix</td>
<td>$V^2$</td>
<td>1†</td>
<td>1</td>
<td>V</td>
</tr>
<tr>
<td>adjacency lists</td>
<td>$E + V$</td>
<td>1</td>
<td>outdegree(v)</td>
<td>outdegree(v)</td>
</tr>
</tbody>
</table>

† disallows parallel edges

Picture from Kevin Wayne
A first Pseudo-code for DFS

Do exactly the same as BFS, just replace the queue by a stack.

Algorithm 9 DFS \((G, \text{start vertex } s)\)

1: Initialize all vertices as unexplored
2: \(S \leftarrow s\) ▶ a stack data structure initialized with \(s\)
3: while \(S \neq \emptyset\) do
4: \(u = \text{POP}(S)\)
5: for each \(v \in \text{Adj}[u]\) do ▶ For each edge \(uv\)
6: if \(v\) is unexplored then
7: \(\text{Push}(S, v)\)
8: mark \(v\) as explored

Observation:

- Only unexplored vertex can enter the Stack, and are marked explored as soon as they enter.
We want more

As for BFS, we color the vertices:
- *White* if it is unexplored
- *Gray* if it is discovered, that is some of its (out)-neighbors are still unexplored
- *Black* if it is finished, that is all its (out)-neighbors has been discovered.

For applications, we also compute:
- $v.d$: time when $v$ has been discovered (grayed), and
- $v.f$: time when $v$ is finished (blackened).
- For every vertex $v$, we have $1 \leq v.d \leq v.f \leq 2n$

As well as a DFS forest:
- As in BFS, whenever DFS discovers a vertex $v$ during a scan of the adjacency list of an already discovered vertex $u$, it records this event by setting $v$’s predecessor attribute $v.\pi \leftarrow u$
- $G_\pi = (V, E_\pi)$ where $E_\pi = \{ uu.\pi : u \in V, u.\pi \neq Nil \}$
### Algorithm 10 DFS($G$)

1. **for** each $u \in V$ **do**
2.   \[ u.color \leftarrow \text{White}, \text{ and } u.pi \leftarrow \text{NIL} \]
3. \[ \text{time} \leftarrow 0 \] \hspace{1cm} ▷ \text{time} \text{ is a global variable used for timestamping}
4. **for** Each vertex $v \in V$ **do**
5.   \[ \text{if } v.color == \text{White} \text{ then} \]
6.   \[ \text{DFS-Visit}(G, v) \]

### Algorithm 11 DFS-Visit($G, v$)

1. \[ \text{time} \leftarrow \text{time} + 1 \]
2. \[ u.d \leftarrow \text{time} \text{ and } u.color \leftarrow \text{Gray} \]
3. **for** Each $v \in \text{Adj}[u]$ **do** \hspace{1cm} ▷ For each edge $uv$
4.   \[ \text{if } v.color == \text{White} \text{ then} \]
5.   \[ v.pi \leftarrow u \]
6.   \[ \text{DFS-Visit}(G, v) \]
7. \[ \text{time} \leftarrow \text{time} + 1 \]
8. \[ u.color \leftarrow \text{Black} \text{ and } u.f \leftarrow \text{time} \]
DFS Execution

Introduction to algorithms, 3rd edition, Cormen, Leiserson, Rivest, Stein
DFS Execution

Introduction to algorithms, 3rd edition, Cormen, Leiserson, Rivest, Stein
DFS Execution

Introduction to algorithms, 3rd edition, Cormen, Leiserson, Rivest, Stein
DFS Execution

Graph searchs: BFS and DFS

Introduction to algorithms, 3rd edition, Cormen, Leiserson, Rivest, Stein
DFS Execution

Introduction to algorithms, 3rd edition, Cormen, Leiserson, Rivest, Stein
DFS Execution

Introduction to algorithms, 3rd edition, Cormen, Leiserson, Rivest, Stein
DFS Execution

Introduction to algorithms, 3rd edition, Cormen, Leiserson, Rivest, Stein
DFS Execution

Graph searchs: BFS and DFS

Introduction to algorithms, 3rd edition, Cormen, Leiserson, Rivest, Stein
DFS Execution

Introduction to algorithms, 3rd edition, Cormen, Leiserson, Rivest, Stein
DFS Execution

Introduction to algorithms, 3rd edition, Cormen, Leiserson, Rivest, Stein
DFS Execution

Introduction to algorithms, 3rd edition, Cormen, Leiserson, Rivest, Stein
DFS Execution

Introduction to algorithms, 3rd edition, Cormen, Leiserson, Rivest, Stein
DFS Execution

Introduction to algorithms, 3rd edition, Cormen, Leiserson, Rivest, Stein
DFS Execution

Introduction to algorithms, 3rd edition, Cormen, Leiserson, Rivest, Stein
DFS Execution

Introduction to algorithms, 3rd edition, Cormen, Leiserson, Rivest, Stein
DFS Execution

Introduction to algorithms, 3rd edition, Cormen, Leiserson, Rivest, Stein

Graph searches: BFS and DFS
Running time

Algorithm 12 DFS(G)

1: for each \( u \in V \) do
2: \( u.color \leftarrow White \), and \( u.pi \leftarrow NIL \)
3: \( time \leftarrow 0 \) \hspace{1cm} \triangleright time is a global variable used for timestamping
4: for Each vertex \( v \in V \) do
5: \hspace{0.5cm} if \( v.color == White \) then
6: \hspace{1cm} DFS-Visit(\( G, v \))

Algorithm 13 DFS-Visit(\( G, v \))

1: \( time \leftarrow time + 1 \)
2: \( u.d \leftarrow time \) and \( u.color \leftarrow Gray \)
3: for Each \( v \in Adj[u] \) do \hspace{1cm} \triangleright For each edge \( uv \)
4: \hspace{1cm} if \( v.color == White \) then \hspace{1cm} \triangleright if \( v \) has not been discovered yet
5: \hspace{1.5cm} \( v.pi \leftarrow u \)
6: \hspace{1cm} DFS-Visit(\( G, v \))
7: \( time \leftarrow time + 1 \)
8: \( u.color \leftarrow Black \) and \( u.f \leftarrow time \)

Running time \( O(n + m) \)
First properties of DFS

- The attributes $u.d$ and $u.f$ range between 1 and $2n$.

- For all vertices $u$, $u.d < u.f$.

- At time $t$, the color of a vertex $u$ is:
  - white if $t < u.d$,
  - gray if $u.d \leq t \leq u.f$,
  - black if $u.f < t$

- $G_\pi$ is a forest ($u = v.\pi$ if and only if DFS-VISIT($G, v$) was called during a search of $u$'s adjacency list).

- A vertex $v$ is a descendant of a vertex $u$ in the DFS forest if and only if $v$ is discovered during the time $u$ is gray (i.e. $u.d \leq v.d \leq u.f$).
Theorem (Parenthesis theorem)

In any DFS of a (directed or undirected) graph $G = (V, E)$, for any two vertices $u$ and $v$, exactly one of the following three conditions holds:

1. the intervals $[u.d, u.f]$ and $[v.d, v.f]$ are disjoint,
2. the interval $[u.d, u.f]$ is contained in $[v.d, v.f]$ (i.e. $v.d < u.d < u.f < v.f$).
3. the interval $[v.d, v.f]$ is contained in $[u.d, u.f]$ (i.e. $u.d < v.d < v.f < u.f$).
Theorem (Parenthesis theorem)
In any DFS of a (directed or undirected) graph $G = (V, E)$, for any two vertices $u$ and $v$, exactly one of the following three conditions holds:

1. the intervals $[u.d, u.f]$ and $[v.d, v.f]$ are disjoint, AND $u$ and $v$ are in two distinct components of the DFS forest.

2. the interval $[u.d, u.f]$ is contained in $[v.d, v.f]$ (i.e. $v.d < u.d < u.f < v.f$). AND $u$ is a descendant of $v$.

3. the interval $[v.d, v.f]$ is contained in $[u.d, u.f]$ (i.e. $u.d < v.d < v.f < u.f$). AND $u$ is a descendant of $v$. 
Proof We begin with the case in which $u.d < v.d$. We consider two subcases, according to whether $v.d < u.f$ or not. The first subcase occurs when $v.d < u.f$, so $v$ was discovered while $u$ was still gray, which implies that $v$ is a descendant of $u$. Moreover, since $v$ was discovered more recently than $u$, all of its outgoing edges are explored, and $v$ is finished, before the search returns to and finishes $u$. In this case, therefore, the interval $[v.d, v.f]$ is entirely contained within the interval $[u.d, u.f]$. In the other subcase, $u.f < v.d$, and by inequality (22.2), $u.d < u.f < v.d < v.f$; thus the intervals $[u.d, u.f]$ and $[v.d, v.f]$ are disjoint. Because the intervals are disjoint, neither vertex was discovered while the other was gray, and so neither vertex is a descendant of the other.

The case in which $v.d < u.d$ is similar, with the roles of $u$ and $v$ reversed in the above argument.
Corollary: A vertex $v$ is a descendant of a vertex $u$ in the DFS forest if and only if $u.d < v.d < v.f < u.f$

Theorem (White-path theorem)
In a DFS forest of a digraph $G = (V, E)$, a vertex $v$ is a descendant of vertex $u$ if and only if at the time $v.d$, there $(v, u)$-path made of white vertices.
Classification of the edges

We can define four edge types in terms of the DFS forest $G_{\pi}$ produced by a DFS:

1. **Tree edges** are edges of $G_{\pi}$.
2. **Back edges** are edges $uv$ such that $u$ is an ancestor of $v$.
3. **Forward edges** are edges $uv$ such that $v$ is an ancestor of $u$.
4. **Cross edge** are edges $uv$ such that $u$ is not an ancestor of $v$ and $v$ is not an ancestor of $u$ (i.e., $u$ and $v$ are in two distinct connected components of the DFS forest).

A lot of informations are contained in this, for example, a digraph has a directed cycle if and only (any) DFS produces a **back edge**.
Recursive pseudo-code for DFS

**Algorithm 14 DFS(G)**

1: **for** each $u \in V$ **do**
2: $u.color \leftarrow \text{White}$, and $u.pi \leftarrow \text{NIL}$
3: $time \leftarrow 0$  \hspace{1cm} $\triangleright$ \textit{time} is a global variable used for timestamping
4: **for** Each vertex $v \in V$ **do**
5: \hspace{.5cm} **if** $v.color == \text{White}$ **then**
6: \hspace{1cm} \hspace{.5cm} \text{DFS-Visit}(G, v)

**Algorithm 15 DFS-Visit(G, v)**

1: $time \leftarrow time + 1$
2: $u.d \leftarrow time$ and $u.color \leftarrow \text{Gray}$
3: **for** Each $v \in \text{Adj}[u]$ **do**  \hspace{1cm} $\triangleright$ For each edge $uv$
4: \hspace{1cm} **if** $v.color == \text{White}$ **then**  \hspace{1cm} $\triangleright$ if $v$ has not been discovered yet
5: \hspace{1.5cm} $v.pi \leftarrow u$
6: \hspace{1.5cm} \text{DFS-Visit}(G, v)
7: $time \leftarrow time + 1$
8: $u.color \leftarrow \text{Black}$ and $u.f \leftarrow time$
Compute the type of the edges

We can compute the type of each edge when running DFS, with no extra cost.

Assume DFS is running and the edge $uv$ is being scanned. Then the color of $v$ gives us some indication on the type of $uv$:

- If $v$ is White, then $uv$ is a tree edge (easy to see).
- If $v$ is gray, then $uv$ is a back edge.
- If $v$ is black then
  - $uv$ is a forward edge if $u.d < v.d$
  - $uv$ is a cross edge if $u.d > v.d$.

To see the second point, observe that at any moment, a set of gray vertices induces a path of the DFS forest (a path of tree edges).

To see the third points, observe that when you’re blacken, all your descendants and ancestors have been discovered already.

---

\(^3\)Note that $u$ is gray at this moment
Compute the type of the edges

We can compute the type of each edge when running DFS, with no extra cost.

Assume DFS is running and the edge $uv$ is being scanned. Then the color of $v$ gives us some indication on the type of $uv$:

- If $v$ is White, then $uv$ is a tree edge (easy to see).
- If $v$ is gray, then $uv$ is a back edge.
- If $v$ is black then
  - $uv$ is a forward edge if $u.d < v.d$.
  - $uv$ is a cross edge if $u.d > v.d$.

\[^3\text{Note that } u \text{ is gray at this moment}\]
First Application of DFS: topological sort
Precedence scheduling

Goal: Given a set of tasks to be completed with precedence constraints, in which order should we schedule the tasks?

Digraph model: vertex = tasks, edge = precedence constraint.
Definition of a topological ordering

**Definition** A topological ordering of the vertices of a directed graph $G$ is a labeling $f : V(G) \to \{1, \ldots, n\}$ such that:

- The $f(v)$'s are the set $\{1, 2, \ldots, n\}$ and
- $uv \in E$ if and only if $f(u) < f(v)$.

**Observations:**

- It is an ordering of the vertices along a horizontal line so that all directed edges go from left to right.
- It is not unique (see example on the board)

**Theorem:** $G$ admits a topological ordering if and only if it is a directed acyclic graph (DAG), that is $G$ has no directed cycle.

**Motivation:** schedule a bunch of tasks when there is precedence constraints among the tasks.
Straightforward solution to compute a topological order

**Sink** = vertex of outdegree 0.

**Observations:**
- Every DAG has a sink
- The last vertex of a topological ordering is a sink.

**Algorithm 16** \textsc{straightforward-topological-sort}(G)

\begin{verbatim}
Find a sink vertex \( v \)
\( f(v) \leftarrow n \)
\textsc{straightforward-topological-ordering}(G \setminus \{v\})
\end{verbatim}

Running time: \( O(n^2) \)

But DFS can do the same thing more efficiently and in a beautiful slick way.
Topological ordering using DFS

**Algorithm 17** \textsc{Topological-sort}(G)

Call \textsc{DFS}(G)

As each vertex is blackens, insert it onto the front of a linked list

\textbf{return} the list
Running time and correctness

Running time: $O(n + m)$.

Sketch of proof of correctness:

- Assume you are executing the algo and you are at the time where a vertex, say $u$, is blackened and put in the linked list.
- It is enough to prove that all the outneighbors of $u$ are already in the linked list.
- It reduces to prove that all the outneighbors of $u$ are black at this time.
- Let $v$ be an outneighbor of $u$,
  - it cannot be white (since $u$ is blackened),
  - and if it is gray then $G$ has a cycle (so we can output: "cycle").
Rigorous proof of correctness

First a Lemma of independant interest:

Lemma 22.11
A directed graph $G$ is acyclic if and only if a depth-first search of $G$ yields no back edges.

Proof $\Rightarrow$: Suppose that a depth-first search produces a back edge $(u, v)$. Then vertex $v$ is an ancestor of vertex $u$ in the depth-first forest. Thus, $G$ contains a path from $v$ to $u$, and the back edge $(u, v)$ completes a cycle.

$\Leftarrow$: Suppose that $G$ contains a cycle $c$. We show that a depth-first search of $G$ yields a back edge. Let $v$ be the first vertex to be discovered in $c$, and let $(u, v)$ be the preceding edge in $c$. At time $v.d$, the vertices of $c$ form a path of white vertices from $v$ to $u$. By the white-path theorem, vertex $u$ becomes a descendant of $v$ in the depth-first forest. Therefore, $(u, v)$ is a back edge. ■
Rigorous proof of correctness

**Theorem 22.12**

TOPOLOGICAL-SORT produces a topological sort of the directed acyclic graph provided as its input.

**Proof** Suppose that DFS is run on a given dag $G = (V, E)$ to determine finishing times for its vertices. It suffices to show that for any pair of distinct vertices $u, v \in V$, if $G$ contains an edge from $u$ to $v$, then $v.f < u.f$. Consider any edge $(u, v)$ explored by DFS($G$). When this edge is explored, $v$ cannot be gray, since then $v$ would be an ancestor of $u$ and $(u, v)$ would be a back edge, contradicting Lemma 22.11. Therefore, $v$ must be either white or black. If $v$ is white, it becomes a descendant of $u$, and so $v.f < u.f$. If $v$ is black, it has already been finished, so that $v.f$ has already been set. Because we are still exploring from $u$, we have yet to assign a timestamp to $u.f$, and so once we do, we will have $v.f < u.f$ as well. Thus, for any edge $(u, v)$ in the dag, we have $v.f < u.f$, proving the theorem.

Pics from Introduction to algorithms, 3\textsuperscript{rd} edition, Cormen, Leiserson, Rivest, Stein
Second Application of DFS: decompose a directed graph into strong connected components
Definition of strongly connected component

Let $G$ be a directed graph. A **strong connected component** (scc) of $G$ is a maximum set of vertices $C$ such that for every pair of vertices $u, v \in C$, there is a directed path from $u$ to $v$ and from $v$ to $u$.

---

4 we have both $u \rightsquigarrow v$ and $v \rightsquigarrow u$, we say $u$ is strongly connected to $v$. 

Pics from Introduction to algorithms, 3rd edition, Cormen, Leiserson, Rivest, Stein
The reverse of a directed graph

Let $G = (V, E)$ be a directed graph. We define the reverse of $G$ as $G^R = (V, E^R)$ where $E^R = \{uv : vu \in E\}$ (reverse all edges).

**Observation:** $G$ and $G^R$ have exactly the same strongly connected components.

**Exercise:** How can you compute $G^R$ efficiently?
Pseudo-code for SCC-decomposition

The following linear-time algorithm mysteriously computes the scc using two depth-first searches, one on $G$ and one on $G^R$.

Algorithm 18 $SCC(G)$

1: Call DFS$(G)$ to compute the finished time $u.f$ of each vertex 
2: Compute $G^R$ 
3: Call DFS$(G^R)$, but in the main loop of DFS, consider the vertices in order of decreasing $u.f$ 
4: Output the vertices of each tree in the DFS forest computed by DFS$(G^R)$ as the set of strongly connected components

The main loop of DFS corresponds to line 4:

1: for each $u \in V$ do 
2: \hspace{1em} $u.color \leftarrow White$, and $u.pi \leftarrow NIL$ 
3: \hspace{1em} $time \leftarrow 0$ 
4: for Each vertex $v \in V$ do 
5: \hspace{1em} if $v.color == White$ then 
6: \hspace{2em} DFS-Visit$(G, v)$
Execution of SCC

Figure from Introduction to algorithms, 3rd edition, Cormen, Leiserson, Rivest, Stein
The idea behind this algorithm comes from a key property of the graph \( G^{SCC} = (V^{SCC}, E^{SCC}) \) obtained from \( G \) by shrinking strong connected components:

Let \( C_1, \ldots, C_k \) be the connected components of \( G \).

Set \( V^{SCC} = \{v_1, \ldots, v_k\} \), one vertex for each scc.

And \( v_i v_j \in E^{SCC} \) if and only if there is an edge from \( C_i \) to \( C_j \) in \( G \).
Key property of $G^{SCC}$

**Key property:** $G^{SCC}$ is a DAG.

It is an easy corollary of the following Lemma:

**Lemma:** Let $C$ and $C'$ be two scc of a directed graph $G$. Let $u, v \in C$ and $u', v' \in C'$. Assume moreover that there is a path from $u$ to $u'$. Then there cannot be a path from $v'$ to $v$.

**Proof:** If $G$ contains a path $v' \leadsto v$, then it contains paths and $v' \leadsto v \leadsto u$ and a path $u \leadsto u' \leadsto v'$. Thus, $u$ and $v'$ are reachable from each other, thereby contradicting the assumption that $C$ and $C'$ are distinct strongly connected components.
Definitions

We extend the discovery and finishing times to sets of vertices: if $S$ is a set of vertices, we write

- $d(S) = \min\{u.d : u \in S\}$ (first time a vertex of $S$ is discovered)
- $f(S) = \max\{u.f : u \in S\}$ (time when all vertices of $S$ are blacks).
Lemma: Let $C$ and $C'$ be two scc and let $u \in C$ and $v \in C'$ such that $uv \in E$. Then $f(C) > f(C')$.

Proof: We consider two cases, depending on which strongly connected component, $C$ or $C'$, had the first discovered vertex during the depth-first search.

If $d(C) < d(C')$, let $x$ be the first vertex discovered in $C$. At time $x.d$, all vertices in $C$ and $C'$ are white. At that time, $G$ contains a path from $x$ to each vertex in $C$ consisting only of white vertices. Because $(u, v) \in E$, for any vertex $w \in C'$, there is also a path in $G$ at time $x.d$ from $x$ to $w$ consisting only of white vertices: $x \sim u \rightarrow v \sim w$. By the white-path theorem, all vertices in $C$ and $C'$ become descendants of $x$ in the depth-first tree. By Corollary 22.8, $x$ has the latest finishing time of any of its descendants, and so $x.f = f(C) > f(C')$.

If instead we have $d(C) > d(C')$, let $y$ be the first vertex discovered in $C'$. At time $y.d$, all vertices in $C'$ are white and $G$ contains a path from $y$ to each vertex in $C'$ consisting only of white vertices. By the white-path theorem, all vertices in $C'$ become descendants of $y$ in the depth-first tree, and by Corollary 22.8, $y.f = f(C')$. At time $y.d$, all vertices in $C$ are white. Since there is an edge $(u, v)$ from $C$ to $C'$, Lemma 22.13 implies that there cannot be a path from $C'$ to $C$. Hence, no vertex in $C$ is reachable from $y$. At time $y.f$, therefore, all vertices in $C$ are still white. Thus, for any vertex $w \in C$, we have $w.f > y.f$, which implies that $f(C) > f(C')$. ■
By the previous Lemma, SCC output the strongly connected components of the input graph.