Outline

1. Introduction
2. Traversal Trees
3. DFS
4. Directed Acyclic Graphs
5. BFS
6. Strongly Connected Directed Graphs
7. Diameters of Graphs
8. Articulation Points
Graph Traversals

Input
- A simple, undirected, and connected graph $G = (V, E)$ with $|V| = n$ vertices and $|E| = m$ edges.

Objective
- Find a traversal path that
  - visits all the vertices of the graph and
  - traverses all the edges of a graph.

Remark
- Vertices can be visited more than once and edges can be traversed more than once.
Graph Traversals

A general scheme

- **Start** with one of the vertices and traverse one of its incident edges to reach another vertex.
- Using traversed edges, **go** to one of the visited vertices that has an incident untraversed edge.
- **Traverse** this untraversed edge.
- **Continue** until all the edges are traversed.

Correctness

- Connectivity implies that all edges are traversed and that all vertices are visited.
Graph Traversals

Efficiency objectives

- Apply simple rules to find an untraversed edge.
- Implement the traversal procedure with an efficient data structure.
- Finish fast in $\Theta(n + m)$ running time.

Example

- An Euler path is the **shortest** possible traversal path if exists – no edge is traversed more than once.
Models

Directed graphs
- An edge is traversed only from its origin to its destination.

Disconnected graphs
- When stuck, **jump** to an unvisited vertex.
- **Continue** until all vertices are visited and all edges are traversed.
Traversal path: A
Example

Traversal path: $AB$
Traversals path: $ABE$
Example

Traversal path: \textbf{ABED}
Example

Traversal path: $ABEDF$
Example

Traversal path: $ABEDFE$
Traversal path: $ABEDFEB$
Example

Traversal path: \textit{ABEDFEBBC}
Example

Traversal path: \textit{ABEDFEBCA}
Example

### Graph Traversal

Traversal path: **ABEDFEBCAD**

![Graph Diagram](image)
Traversal path: \textbf{ABEDFEBCADF}
Traversal path: $ABEDFEBCADFC$
Traversals Trees

The tree structure

- The traversal tree is a **rooted**, **ordered**, and **directed**.
  - The first visited vertex is the root of the tree.
  - Vertex $u$ is the parent of $v$ if the first visit to $v$ happened after traversing the edge $(u, v)$.
  - The children of a vertex $u$ are ordered according to the time they were first visited from $u$.

Edge classification

- **Tree edge**: an edge from a vertex to one of its children.
- **Back edge**: an edge from a vertex to one of its ancestors.
- **Forward edge**: an edge from a vertex to one of its descendants which is not its child.
- **Cross edge**: an edge from a vertex to another vertex that is neither one of its ancestors nor one of its descendants in the traversal tree.
Example

Traversal path: A
Example

Traversal path: \( AB \)
Example

Traversal path: $ABE$
Example

Traversal path: \textit{ABED}
Traversals Trees

Example

Graph Traversals

Traversals path: $ABEDF$
Traversal path: ABEDFE
Traversal path: **ABEDFEB**
Traversing Trees

Example

Traversal path: \( ABEDFEB \)
Traversal path: \textit{ABEDFEBCA}
Example

Traversal path: \textit{ABEDFEBCAD}
Example

Traversal path: $ABEDFEBCADF$
Traversals Trees

Example

Traversal path: \textit{ABEDFEBCADFC}
DFS – Depth First Search

**Input**
- An undirected graph $G = (V, E)$.
- A global order on the $n$ vertices.

**Output**
- A traversal forest that contains a traversal tree for each connected component of the graph.

**Traversal rule**
- For each connected component, visit a vertex, then recursively visit all of its neighbors in order.

**Directed graphs**
- In the traversal forest, each tree is a directed tree.
- Each tree has a directed path from its root to any other vertex in the tree.
Example – a DFS traversal Path

Traversal path: A
Example – a DFS traversal Path

Traversal path: $AB$
Example – a DFS traversal Path

Traversal path: $ABC$
Example – a DFS traversal Path

Traversal path: \(ABCA\)
Example – a DFS traversal Path

Traversal path: $ABCAC$
Example – a DFS traversal Path

Traversal path: ABCACF
Example – a DFS traversal Path

Traversal path: $ABCACFD$
Example – a DFS traversal Path

Traversal path: \textit{ABCACFDA}
Example – a DFS traversal Path

Traversal path: $ABCACFDAD$
Example – a DFS traversal Path

Traversal path: ABCACFDADE
Example – a DFS traversal Path

Traversal path: ABCACFDADEB
Example – a DFS traversal Path

Traversal path: **ABCACFDADEBE**
Example – a DFS traversal Path

Traversa}l path: ABCACFDADEBEF
Example – a DFS traversal Path

Traversal path: ABCACFDADEBEF – EDFCBA
Example – the DFS traversal Tree

Traversal path: A
Example – the DFS traversal Tree

Traversal path: \( AB \)
Example – the DFS traversal Tree

Traversal path: \textit{ABC}
Example – the DFS traversal Tree

Traversal path: $ABCA$
Example – the DFS traversal Tree

Traversal path: $ABCAC$
Example – the DFS traversal Tree

Traversal path: \textit{ABCACF}
Example – the DFS traversal Tree

Traversal path: **ABCACFD**
Example – the DFS traversal Tree

Traversal path: \textit{ABCACFDA}
Example – the DFS traversal Tree

Traversal path: $ABCACFDAD$
Example – the DFS traversal Tree

Traversal path: $ABCACFDEA$
Example – the DFS traversal Tree

Traversal path: $ABCACFDADEB$
Example – the DFS traversal Tree

Traversal path: $ABCACFDADEBE$
Example – the DFS traversal Tree

Traversal path: \textit{ABCACFDADEBEF}
Data Structure

Vertices

- Each vertex is colored by one of the following colors:
  - *White*: The recursive visit has not started.
  - *Gray*: The recursive visit started but not finished.
  - *Black*: The recursive visit finished.

Time

- A global discrete time variable *time* that is updated when a vertex becomes *Gray* and when a vertex becomes *Black*.
  - *d*(v): The time vertex v becomes *Gray*.
  - *f*(v): The time vertex v becomes *Black*.

Traversal trees

- A parenthood function \( \Pi(v) \) in the traversal forest:
  - \( \Pi(v) = nil \) if v is a root of a tree in the forest.
  - \( \Pi(v) \) is v’s parent in the traversal tree that contains v.
The DFS Procedure

Initial call

**DFS**\( (G) \)

for each vertex \( v \in V \) do

\( \text{Color}(v) = \text{White} \)

\( \Pi(v) = \text{nil} \)

\( \text{time} = 0 \)

for each vertex \( v \in V \) do

if \( \text{Color}(v) = \text{White} \) then

**DFS-Visit**\( (v) \)

Observation

- **DFS** handles all connected components.
The DFS Procedure

Recursive call

DFS-Visit\( (v) \)

\[
\text{Color}(v) = \text{Gray} \\
\text{time}++ \\
d(v) = \text{time}
\]

for each neighbor \( u \) of \( v \) do

\[
\text{if Color}(u) = \text{white then} \\
\quad \Pi(u) = v \\
\quad \text{DFS-Visit}(u)
\]

\[
\text{Color}(v) = \text{Black} \\
\text{time}++ \\
f(v) = \text{time}
\]

Observation

- **DFS-Visit** handles one connected component.
Example – Directed DFS

$time = 0$
Example – Directed DFS

$time = 1$
Example – Directed DFS

$time = 2$
Example – Directed DFS

\[ \text{time} = 3 \]
Example – Directed DFS

\[ time = 4 \]
Example – Directed DFS

(time = 5)
Example – Directed DFS

\[
time = 6
\]
Example – Directed DFS

time = 7
Example – Directed DFS

\[ \text{time} = 8 \]
Example – Directed DFS

$time = 9$
Example – Directed DFS

time = 10
Example – Directed DFS

\[ time = 11 \]
Example – Directed DFS

\[
\text{time} = 12
\]
DFS – Correctness

**Lemma**
- DFS visits all the vertices.

**Proof**
- Each vertex changes colors as follows:
  - **White** → **Gray** → **Black**

**Lemma**
- DFS traverses all the edges.

**Proof**
- For each vertex, DFS examines all of its incident edges in undirected graphs and all of its outgoing edges in directed graphs.
Visiting vertices

- When visiting an already visited vertex, just traverse the edge and return back.

Undirected graphs

- The traversal path is not explicit. Traversing back the edges is done implicitly due to the recursive calls.

Directed graphs

- The traversal path sometimes uses the wrong direction due to the recursion.
DFS – Time Complexity

**Adjacency lists**
- $\Theta(m)$: Each edge is examined once in directed graphs and twice in undirected graphs.
- $\Theta(n)$: Each vertex is colored three times.
- $\Theta(n + m)$: Overall complexity.

**Adjacency matrix**
- $\Theta(n)$: For each vertex, for examining all of its incident edges in undirected graphs and all of its outgoing edges in directed graphs.
- $\Theta(n^2)$: Overall complexity.
DFS – Time Intervals

**Definition**
- The DFS interval of vertex \( v \) is \([d(v), f(v)]\).

**Lemma**
- Assume \( d(u) < d(v) \) for an edge \((u, v)\):
  - Either the intervals of \( u \) and \( v \) are disjoint
    \[ d(u) < f(u) < d(v) < f(v). \]
  - Or \( u \)'s interval contains \( v \)'s interval
    \[ d(u) < d(v) < f(v) < f(u). \]
Example – Time Intervals

\[
\begin{align*}
\text{time} = 1 & \quad A = \text{Gray} & \quad d(A) = 1 \\
\text{time} = 2 & \quad B = \text{Gray} & \quad d(B) = 2 \\
\text{time} = 3 & \quad C = \text{Gray} & \quad d(C) = 3 \\
\text{time} = 4 & \quad C = \text{Black} & \quad f(C) = 4 \\
\text{time} = 5 & \quad E = \text{Gray} & \quad d(E) = 5 \\
\text{time} = 6 & \quad D = \text{Gray} & \quad d(D) = 6 \\
\text{time} = 7 & \quad D = \text{Black} & \quad f(D) = 7 \\
\text{time} = 8 & \quad E = \text{Black} & \quad f(E) = 8 \\
\text{time} = 9 & \quad B = \text{Black} & \quad f(B) = 9 \\
\text{time} = 10 & \quad A = \text{Black} & \quad f(A) = 10 \\
\text{time} = 11 & \quad F = \text{Gray} & \quad d(F) = 11 \\
\text{time} = 12 & \quad F = \text{Black} & \quad f(F) = 12
\end{align*}
\]
Example – Time Intervals

DFS
DFS – Traversal Tree Edge Classification

Classification

- **A tree edge (T):** An edge from a *Gray* vertex to a *White* vertex.
- **A back edge (B):** An edge from a *Gray* vertex to a *Gray* vertex.
- **A forward edge (F):** An edge from a *Gray* vertex to a *Black* vertex whose interval is nested.
- **A cross edge (C):** An edge from a *Gray* vertex to a *Black* vertex whose interval finished.

Observation

- The above classification covers all the edges.
- Each edge belongs to one class.
Example – Edge Classification

*(A,B)*  \(\text{Gray} \rightarrow \text{White} \)  \(\text{T}\)

*(B,C)*  \(\text{Gray} \rightarrow \text{White} \)  \(\text{T}\)

*(B,E)*  \(\text{Gray} \rightarrow \text{White} \)  \(\text{T}\)

*(E,D)*  \(\text{Gray} \rightarrow \text{White} \)  \(\text{T}\)

*(D,A)*  \(\text{Gray} \rightarrow \text{Gray} \)  \(\text{B}\)

*(A,C)*  \(\text{Gray} \rightarrow \text{Black}\) (nested)  \(\text{F}\)

*(F,C)*  \(\text{Gray} \rightarrow \text{Black}\) (finished)  \(\text{C}\)

*(F,D)*  \(\text{Gray} \rightarrow \text{Black}\) (finished)  \(\text{C}\)

*(F,E)*  \(\text{Gray} \rightarrow \text{Black}\) (finished)  \(\text{C}\)
Lemma

DFS has no **forward** and no **cross** edges.

Proof

There is no edge from a *Gray* vertex to a *Black* vertex.

Otherwise, the *Black* vertex, before becoming black, would examine its incident edge to the *Gray* vertex in the other direction.
Theorem

- An undirected graph has a cycle iff the DFS creates at least one back edge.

Proof $\leftarrow$

- Suppose $(u, v)$ is a back edge examined at $u$.
- Then $v$ is an ancestor of $u$.
- Therefore, there exists a path from $v$ to $u$.
- Adding $(u, v)$ to the path creates a cycle.
Cycles in Undirected Graphs

Theorem

An undirected graph has a cycle iff the DFS creates at least one back edge.

Proof ⇒

- Suppose $C$ is a cycle in $G$.
- Let $v$ be the first Gray vertex in $C$.
- Let $(u, v)$ be the preceding edge in $C$.
- After time $d(v)$, DFS explores the $v$ to $u$ path.
- When $(u, v)$ is traversed, both $v$ and $u$ are Gray.
- Hence, $(u, v)$ is a back edge.
Forest or not a Forest?

**Problem**
- Is an undirected graph a forest?
- Does an undirected graph have a cycle?

**Algorithm**
- Run the DFS algorithm.
- Terminate if a **back** edge is found.

**Time complexity:** $\Theta(n)$
- If there are no cycles there are $O(n)$ edges. Therefore, the DFS time complexity $\Theta(m + n)$ becomes $\Theta(n)$.
- If there exists a cycle, the algorithm terminates the first time a *Gray* vertex is visited for a second time. This happens after at most $\Theta(n)$ edge traversals.
Theorem

A directed graph $G$ is DAG iff running DFS on $G$ does not produce a back edge.

Proof $\Rightarrow$

Suppose that $(u \rightarrow v)$ is a back edge.
Then $v$ is an ancestor of $u$.
Therefore, there exists a directed path from $v$ to $u$.
Adding $(u \rightarrow v)$ to the path creates a directed cycle.
Theorem

A directed graph $G$ is DAG iff running DFS on $G$ does not produce a back edge.

Proof $\Leftarrow$

Suppose that $\mathcal{C}$ is a directed cycle in $G$.
Let $v$ be the first Gray vertex in $\mathcal{C}$.
Let $(u \rightarrow v)$ be the preceding edge in $\mathcal{C}$.
After time $d(v)$, DFS explores the $v$ to $u$ directed path.
When $(u \rightarrow v)$ is traversed, both $v$ and $u$ are Gray.
Hence, $(u \rightarrow v)$ is a back edge.
**Definition**
- A topological sort of a DAG is a linear ordering of its vertices.
- For any directed edge \((u \rightarrow v)\), vertex \(u\) must appear before vertex \(v\) in this ordering.

**Problem**
- Find one of the topological sorts of a DAG.

**Algorithm**
- Create an empty linked list.
- Run DFS on the DAG.
- A vertex becomes *Black*: add it to the front of the list.
- Return the linked list as a topological sort.
Correctness and Complexity

Correctness

- Consider examining the edge \((u \rightarrow v)\).
- At this time \(u\) is \textit{Gray}.
- \(v\) is not \textit{Gray} since then \((u \rightarrow v)\) is a \textit{back} edge.
- If \(v\) is \textit{White} then \(f(v) < f(u)\) since \(v\) becomes \textit{Black} before \(u\).
- If \(v\) is \textit{Black} then \(f(v) < f(u)\) since \(u\) is still \textit{Gray}.
- \(f(v) < f(u)\) implies that \(u\) appears before \(v\) in the list.

Complexity

- \(\Theta(n + m)\) running time.
- The same complexity as DFS.
BFS – Breadth First Search

**Input**
- An undirected graph \( G = (V, E) \).
- A global order on the \( n \) vertices.

**Output**
- A traversal forest that contains a traversal tree for each connected component of the graph.

**Traversal rule**
- Visit a vertex, then visit all of its neighbors, then visit all of the neighbors of its neighbors, \( \ldots \)  

**Directed graphs**
- In the traversal forest, each tree is a directed tree.
- Each tree has a directed path from its root to any other vertex in the tree.
Example – a BFS traversal Path

Traversal path: A
Example – a BFS traversal Path

Traversal path: $AB$
Example – a BFS traversal Path

Traversal path: \textit{ABA}
Example – a BFS traversal Path

Traversal path: **ABAC**
Example – a BFS traversal Path

Traversal path: ABACA
Example – a BFS traversal Path

Traversal path: \textit{ABACAD}
Example – a BFS traversal Path

Traversal path: **ABACADA**
Example – a BFS traversal Path

Traversal path: **ABACADAB**
Example – a BFS traversal Path

Traversal path: \textit{ABACADABC}
Example – a BFS traversal Path

Traversal path: \text{ABACADABCB}
Example – a BFS traversal Path

Traversal path: ABACADABCBCE
Example – a BFS traversal Path

Traversal path: \textbf{ABACADABCBEB}
Example – a BFS traversal Path

Traversal path: $ABACADABCBEBBA$
Example – a BFS traversal Path

Traversal path: ABACADABCBEBAC
Example – a BFS traversal Path

Traversal path: \textit{ABACADABCBEBACF}
Example – a BFS traversal Path

Traversal path : \textit{ABACADABCBEBAFC}

Amotz Bar-Noy (CUNY)
Example – a BFS traversal Path

Traversal path: \textit{ABACADABCBEBACFCA}
Example – a BFS traversal Path

Traversal path:  ABACADABCBEBACFCAD
Example – a BFS traversal Path

Traversal path: \textit{ABACADABCBEBACFCADE}
Example – a BFS traversal Path

Traversal path: \textit{ABACADABCBEBAACFCADED}
Example – a BFS traversal Path

Traversal path: \textit{ABACADABCBEBAFCDEDF}
Example – a BFS traversal Path

Traversal path: \textit{ABACADABCBEBACFCADEDFD}
Example – a BFS traversal Path

Traversal path: \textit{ABACADABCBEBACFCADEDFA}
Example – a BFS traversal Path

Traversal path: \textit{ABACADABCBEBAFCACEDFDAB}
Example – a BFS traversal Path

Traversal path: \textit{ABACADABCBEBACFCADEDFDABE}
Example – a BFS traversal Path

Traversal path: ABACADABCBEBACFCADEDFDABEF
Example – a BFS traversal Path

Traversal path: \textit{ABACADABCBEBACFCADEDFDABEF \textendash EBA}
Example – the BFS Traversal Tree

ABACADABCBEBACFCADEDFAEBEF
Data Structure

Vertices

- Maintain a level function $\text{Level}(v)$:
  - $\text{Level}(v) = 0$ if $v$ is a root of a tree in the forest.
  - $\text{Level}(v) = \ell$ if $v$ is at distance $\ell$ from the root.

A FIFO queue $Q$

- $x = \text{First}(Q)$: Delete $x$ the first element in $Q$.
- $\text{Last}(Q) = x$: Add $x$ as the last element in $Q$.
- $\text{Create}(Q)$: Create an empty queue $Q$.
- $\text{Empty}(Q)$: Check if the queue $Q$ is empty.

Traversal trees

- A parenthood function $\Pi(v)$ in the traversal forest:
  - $\Pi(v) = \text{nil}$ if $v$ is a root of a tree in the forest.
  - $\Pi(v)$ is $v$’s parent in the traversal tree that contains $v$. 
The BFS Procedure

Initial call

\[ \text{BFS}(G) \]
\[
\text{for each vertex } v \in V \text{ do} \\
\quad \text{Level}(v) = \infty \\
\quad \Pi(v) = \text{Nil} \\
\text{for each vertex } r \in V \text{ do} \\
\quad \text{if Level}(r) = \infty \text{ then} \\
\quad \quad \text{BFS-Visit}(r)
\]

Observation

- BFS handles all connected components.
The BFS Procedure

Queue handling

**BFS-Visit**($r$)

- $Level(r) = 0$
- $Create(Q)$
- $Last(Q) = r$

while (not $Empty(Q)$) do
  
  $v = First(Q)$
  
  for each neighbor $u$ of $v$ do

  if $Level(u) = \infty$ then

    $\Pi(u) = v$

    $Level(u) = Level(v) + 1$

    $Last(Q) = u$

Observation

- **BFS-Visit** handles one connected component.
BFS – Example

```
<table>
<thead>
<tr>
<th>Initial Queue</th>
<th>A(0)</th>
<th>B(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>D(1)</th>
<th>E(2)</th>
<th>F(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Empty Queue</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

A(0) B(1)
C(1)
D(1)
C(1)
E(2)
D(1)
D(1)
E(2)
F(2)

Amotz Bar-Noy (CUNY)
BFS – Correctness

Lemma
- BFS visits all the vertices.

Proof
- Each vertex changes its level \( \ell \) as follows
  \[
  \ell = \infty \rightarrow 0 \leq \ell < \infty
  \]

Lemma
- BFS traverses all the edges.

Proof
- For each vertex, BFS examines all of its incident edges in undirected graphs and all of its outgoing edges in directed graphs.
BFS – Traversal Path

**Undirected graphs**
- The traversal path is not explicit. Traversing back the edges is done implicitly due to the queue handling.

**Directed graphs**
- The traversal path sometimes uses the wrong direction due to the queue handling.
BFS – Time Complexity

Adjacency lists
- $\Theta(m)$: Each edge is examined once in directed graphs and twice in undirected graphs.
- $\Theta(n)$: Each vertex gets a level $\ell < \infty$ exactly once.
- $\Theta(n + m)$: Overall complexity.

Adjacency matrix
- $\Theta(n)$: For each vertex, for examining all of its incident edges in undirected graphs and all of its outgoing edges in directed graphs. of each vertex.
- $\Theta(n^2)$: Overall complexity.
The BFS Traversal Tree

Lemma

- BFS has no **forward** edges in directed graphs and in undirected graphs.

Proof

- A **forward** edge would be a **tree** edge.

Lemma

- BFS has no **back** edges in undirected graphs.

Proof

- A **back** edge would be a **tree** edge while examined in the other direction.
The BFS Traversal Tree

**Lemma**

- In undirected graphs, \( \text{Level}(u) = \text{Level}(v) \) or \( \text{Level}(u) = \text{Level}(v) - 1 \) for a BFS cross edge \((u, v)\).

**Proof**

- \( u \) is added to the queue before \( v \).
- Otherwise BFS would examine \((v, u)\) before \((u, v)\).
- Therefore, \( \text{Level}(u) \leq \text{Level}(v) \).
- \( \text{Level}(u) < \text{Level}(v) - 1 \Rightarrow (u, v) \) would be a tree edge.
Shortest Paths

Problem
- Let $u$ and $v$ be 2 vertices in a connected undirected graph $G$.
- Find one of the shortest paths from $u$ to $v$ and its length.

Algorithm
- Run the BFS algorithm starting with the vertex $v$.
- Terminate when the vertex $u$ is visited the first time.
- The length of the shortest path from $u$ to $v$ is the level of vertex $u$ in the BFS traversal tree.
- The path $(u, \Pi(u), \Pi(\Pi(u)), \ldots, v)$ is one of the shortest paths from $u$ to $v$.

Time complexity
- $\Theta(n + m)$: the same as BFS.
**Strongly Connected Directed Graphs**

**Definition**
- A directed graph $G = (V, E)$ is **strongly connected** if for any pair of vertices $u, v \in V$ there exists a directed path from $u$ to $v$.

**Problem**
- Given a directed graph $G = (V, E)$, determine if $G$ is strongly connected.
A Straightforward Algorithm

**Algorithm**
- For all vertices $v \in V$, run DFS or BFS starting with the vertex $v$.
- If there exists a vertex $u$ that is not reachable from $v$, then return **NO** else return **YES**.

**Correctness**
- By definition of DFS and BFS.

**Complexity**
- $\Theta(nm)$ for $n$ times DFS or BFS.
- If $m < n$, then the underlying undirected graph is not connected, and DFS or BFS would return **NO** after the first run.
- The worst case of $n$ runs of DFS or BFS happens only when $m > n - 2$. Therefore, the complexity is $\Theta(nm)$ and not $\Theta(n(n + m))$. 

An Efficient Algorithm

**Algorithm**

- For some vertex $w$, run DFS or BFS on $G$ starting with $w$.
- If there exists $u$ that is not reachable from $w$, then return **NO**.
- Reverse the direction of all edges to get a new directed graph $G'$.
- Run DFS or BFS on $G'$ starting with $w$.
- If there exists $v$ that is not reachable from $w$, then return **NO**.
- Else, return **YES**.

**Complexity**

- $\Theta(n + m)$ for running twice DFS or BFS.
- $\Theta(n + m)$ for reversing the direction of all edges to generate the graph $G'$ by scanning all the outgoing lists to generate new outgoing lists.
- $\Theta(n + m)$ overall complexity.
Correctness

The algorithm returns **NO**

- The algorithm produces an evidence of two vertices with no directed path between them.
- Either there exists $u$ s.t. there is no directed path from $w$ to $u$ in $G$.
- Or there exists $v$ s.t there is no directed path from $w$ to $v$ in $G'$ implying having no directed path from $v$ to $w$ in $G$.

The algorithm returns **YES**

- Let $v, u \in V$ be any pair of vertices in $G$.
- **YES** in $G \Rightarrow$ there is a directed path from $w$ to $u$ in $G$.
- **YES** in $G' \Rightarrow$ there is a directed path from $w$ to $v$ in $G'$ implying a directed path from $v$ to $w$ in $G$.
- Combining the path from $v$ to $w$ with the path from $w$ to $u$ generates the path from $v$ to $u$. 
Diameter of a Graph

Assumption
- Let $G$ be a connected undirected graph.

Notation
- For two vertices $u$ and $v$, denote by $\text{dist}(u, v)$ the length of the shortest path from $u$ to $v$ in $G$.

Definition
- The **diameter** of $G$, denoted by $D(G)$, is the longest shortest path in $G$:

  $$D(G) = \max_{u,v \in G} \{ \text{dist}(u, v) \} .$$

Problem
- Find the diameter of $G$. 
Algorithm for any Graph $G$

**Algorithm**

- For each vertex $v$, run BFS starting with $v$.
- Let $u$ be one of the **farthest** vertices from $v$ and let $r(v) = \text{dist}(v, u)$.
- $D(G) = \max_v \{r(v)\}$.

**Correctness**

- By definition of BFS.

**Complexity**

- In connected graphs $m \geq n - 1$. Therefore, the complexity of BFS is $\Theta(m)$.
- Overall complexity is $\Theta(nm)$ for running BFS $n$ times.
Diameter of a Tree

Problem
- Find the diameter of a tree $T$.

An efficient algorithm
- Run BFS starting with an arbitrary vertex $v$.
- Let $u$ be one of the farthest vertices from $v$.
- Run BFS starting with the vertex $u$.
- Let $w$ be one of the farthest vertices from $u$.
- $D(G) = \text{dist}(u, w)$.

Complexity
- $\Theta(n)$ for running BFS twice on a tree for which $m = n - 1$.
- The complexity of the general algorithm is $\Theta(n^2)$ for running BFS $n$ times on a tree.
Diameters of Graphs

The Algorithm Fails for Non-Trees

Counterexample

- $u$ is the only farthest vertex from $v$.
- $v = w$ is the only farthest vertex from $u$.
- $\text{dist}(x, y) > \text{dist}(u, w)$. 
Correctness

Setting

- Let \( x \) and \( y \) be two vertices such that \( D(G) = \text{dist}(x, y) \).
- If \( v \) is either \( x \) or \( y \), then \( \text{dist}(v, w) = D(G) \) and therefore \( \text{dist}(u, w) = D(G) \).
- Assume that \( \text{dist}(v, y) \geq \text{dist}(v, x) \).
- Let \( z \) be the vertex that connects \( v \) to the path \( x \) to \( y \) (\( z \) can be \( v \) or \( y \)).
**Correctness**

**Case I: $z$ is on the path from $v$ to $u$**

- By definition, $\text{dist}(v, u) \geq \text{dist}(v, y)$.
- Therefore, $\text{dist}(z, u) \geq \text{dist}(z, y)$.
- Therefore, $D(G) = \text{dist}(x, u)$.
- By definition, $\text{dist}(u, w) \geq \text{dist}(u, x)$.
- Therefore, $D(G) = \text{dist}(u, w)$. 
Correctness

Case II: \( z \) is not on the path from \( v \) to \( u \).

\[
\begin{align*}
  &X \\
  &Z \\
  &Y \\
  &P \\
  &V \\
  &U \\
\end{align*}
\]

Let \( p \) be the vertex on the path from \( v \) to \( u \) and the path from \( v \) to \( z \) (\( p \) can be \( v \)).

By definition, \( \text{dist}(v, u) \geq \text{dist}(v, y) \).

Therefore, \( \text{dist}(p, u) \geq \text{dist}(p, y) \).

\( \text{dist}(z, u) > \text{dist}(p, u) \) and \( \text{dist}(z, y) < \text{dist}(p, y) \) by definition.

Therefore, \( \text{dist}(z, u) > \text{dist}(z, y) \).

Therefore, \( \text{dist}(x, u) > \text{dist}(x, y) \).

A contradiction since \( D(G) = \text{dist}(x, y) \).
Articulation Points

Assumption
- Let $G$ be a connected undirected graph.

Notation
- For a vertex $u \in G$, denote by $G_u$ the graph $G$ without the vertex $u$ and all of its incident edges.

Definition
- $v \in G$ is an **articulation point** if $G_v$ is a disconnected graph.

Problem
- Find all the articulation points of $G$. 
A Straightforward Algorithm

Algorithm
- For each vertex $v$, run BFS or DFS on the graph $G_v$.
- Output all the vertices $v$ whose traversal forest contains more than one tree.

Correctness
- By definitions of BFS and DFS.

Complexity
- In connected graphs $m \geq n - 1$. Therefore, the complexity of BFS or DFS is $\Theta(m)$.
- Overall complexity is $\Theta(nm)$ for running BFS or DFS $n$ times.
An Efficient Algorithm

Algorithm
- Run DFS on $G$ starting with an arbitrary vertex $v$.
- If $v$ has more than one child in the traversal tree then $v$ is an articulation point.
- If $u \neq v$ is not a leaf in the DFS traversal tree and $u$ has a child $w$ such that there is no back edge from one of the vertices in the subtree rooted at $w$ to one of its ancestors of $u$, then $u$ is an articulation point.

Correctness
- Recall that the DFS tree in an undirected graph does not have cross and forward edges.
- The correctness follows by carefully examining the DFS procedure and by the definition of articulation points.
An Efficient Algorithm

Complexity

- In the DFS tree, a back edge \((x, y)\) **bypasses** a non-leaf non-root \(v\), if \(x\) is in the subtree rooted at \(v\) and \(y\) is an ancestor of \(v\).

- With the same complexity \(\Theta(n + m)\), it is possible to modify DFS to maintain the following for each vertex \(u\):
  - Count the number of children of \(u\) in the DFS tree.
  - Detect if \(u\) has a child \(w\) whose rooted subtree has no back edges **bypassing** \(v\).

- Overall complexity is \(\Theta(n + m)\) for running the modified DFS algorithm.
Resources: Books

Text book
  - Depth-first search: Section 22.3 pp. 603–612.
  - Topological sort: Section 22.4 pp. 612–615.
  - Strongly connected components: Section 22.5 pp. 615–621.

Another book
- “Algorithm Design,” Goodrich and Tamassia, Wiley.
  - Traversing a Digraph: Section 13.4.1 pp. 374–377.
  - Directed Acyclic Graphs: Section 13.4.4 pp. 382–385.
Online Resources Theory and Implementation

**Directed Graphs**

- **Topological sorting**
  - https://www.geeksforgeeks.org/topological-sorting/
  - https://www.interviewcake.com/concept/java/topological-sort

- **Testing if a directed graph is strongly connected**
  - https://www.geeksforgeeks.org/connectivity-in-a-directed-graph/?ref=lbp

- **Finding all the strongly connecting components of a directed graph**
  - https://www.geeksforgeeks.org/strongly-connected-components/?ref=lbp

**Undirected graphs**

- **Diameter of a tree**
  - https://www.geeksforgeeks.org/diameter-tree-using-dfs/?ref=lbp

- **Articulation points**
  - https://www.geeksforgeeks.org/articulation-points-or-cut-vertices-in-a-graph/