CHAPTER 14
GRAPH ALGORITHMS

ACKNOWLEDGEMENT: THESE SLIDES ARE ADAPTED FROM SLIDES PROVIDED WITH DATA STRUCTURES AND ALGORITHMS IN JAVA, GOODRICH, TAMASSIA AND GOLDWASSER (WILEY 2016)
DEPTH-FIRST SEARCH
DEPTH-FIRST SEARCH

- **Depth-first search (DFS)** is a general technique for traversing a graph
- A DFS traversal of a graph $G$
  - Visits all the vertices and edges of $G$
  - Determines whether $G$ is connected
  - Computes the connected components of $G$
  - Computes a spanning forest of $G$
- DFS on a graph with $n$ vertices and $m$ edges takes $O(n + m)$ time
- DFS can be further extended to solve other graph problems
  - Find and report a path between two given vertices
  - Find a cycle in the graph
- Depth-first search is to graphs as what Euler tour is to binary trees
**DFS ALGORITHM FROM A VERTEX**

**Algorithm DFS(G, u)**

**Input:** A graph G and a vertex u of G

**Output:** A collection of vertices reachable from u, with their discovery edges

1. Mark u as visited
2. for each edge \( e = (u, v) \in G.\text{outgoingEdges}(u) \) do
3. if v has not been visited then
4. Record e as a discovery edge for v
5. DFS(G, v)
EXAMPLE

unexplored vertex
visited vertex
unexplored edge
discovery edge
back edge

\[ I(A) = \{B, C, D, E\} \]

\[ I(B) = \{A, C, F\} \]
\[ I(B) = \{A, C, F\} \]

\[ I(C) = \{A, B, D, E\} \]

\[ I(C) = \{A, B, D, E\} \]
\[ I(C) = \{A, B, D, E\} \]
\[ I(C) = \{A, B, D, E\} \]

\[ I(D) = \{A, C\} \]

\[ I(E) = \{A, C\} \]
Example

$I(C) = \{A, B, D, E\}$

$I(B) = \{A, C, F\}$

$I(G) = \emptyset$

$I(F) = \{B\}$

$I(B) = \{A, C, F\}$

$I(A) = \{A, B, C, D\}
EXERCISE
DFS ALGORITHM

• Perform DFS of the following graph, start from vertex A
  • Assume adjacent edges are processed in alphabetical order
  • Number vertices in the order they are visited
  • Label edges as discovery or back edges
DFS AND MAZE TRAVERSAL

• The DFS algorithm is similar to a classic strategy for exploring a maze
  • We mark each intersection, corner and dead end (vertex) visited
  • We mark each corridor (edge) traversed
  • We keep track of the path back to the entrance (start vertex) by means of a rope (recursion stack)
**DFS ALGORITHM**

- The algorithm uses a mechanism for setting and getting “labels” of vertices and edges

**Algorithm DFS(G)**

**Input:** Graph G

**Output:** Labeling of the edges of G as discovery edges and back edges

1. for each \( v \in G \) \.vertices() do
2. setLabel(\( v \), UNEXPLORED)
3. for each \( e \in G \).edges() do
4. setLabel(\( e \), UNEXPLORED)
5. for each \( v \in G \).vertices() do
6. if getLabel(\( v \)) = UNEXPLORED then
7. DFS(G, v)

**Algorithm DFS(G, v)**

**Input:** Graph G and a start vertex v

**Output:** Labeling of the edges of G in the connected component of v as discovery edges and back edges

1. setLabel(\( v \), VISITED)
2. for each \( e \in G \).outgoingEdges(\( v \)) do
3. if getLabel(\( e \)) = UNEXPLORED
4. \( w \leftarrow G \).opposite(\( v \), \( e \))
5. if getLabel(\( w \)) = UNEXPLORED then
6. setLabel(\( e \), DISCOVERY)
7. DFS(G, w)
8. else
9. setLabel(\( e \), BACK)
PROPERTIES OF DFS

• Property 1
  • DFS\((G, v)\) visits all the vertices and edges in the connected component of \(v\)

• Property 2
  • The discovery edges labeled by DFS\((G, v)\) form a spanning tree of the connected component of \(v\)
ANALYSIS OF DFS

- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or BACK
- Function $\text{DFS}(G, v)$ and the method outgoingEdges() are called once for each vertex
- DFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\sum_v \text{deg}(v) = 2m$
APPLICATION
PATH FINDING

• We can specialize the DFS algorithm to find a path between two given vertices $u$ and $z$ using the template method pattern

• We call $\text{DFS}(G, u)$ with $u$ as the start vertex

• We use a stack $S$ to keep track of the path between the start vertex and the current vertex

• As soon as destination vertex $z$ is encountered, we return the path as the contents of the stack

Algorithm $\text{pathDFS}(G, v, z)$

1. $\text{setLabel}(v, \text{VISITED})$
2. $S.\text{push}(v)$
3. if $v = z$
4. return $S.\text{elements}()$
5. for each $e \in G.\text{outgoingEdges}(v)$ do
6. if $\text{getLabel}(e) = \text{UNEXPLORED}$ then
7. $w \leftarrow G.\text{opposite}(v, e)$
8. if $\text{getLabel}(w) = \text{UNEXPLORED}$ then
9. $\text{setLabel}(e, \text{DISCOVERY})$
10. $S.\text{push}(e)$
11. $\text{pathDFS}(G, w)$
12. $S.\text{pop}()$
13. else
14. $\text{setLabel}(e, \text{BACK})$
15. $S.\text{pop}()$
APPLICATION
CYCLE FINDING

- We can specialize the DFS algorithm to find a simple cycle using the template method pattern
- We use a stack $S$ to keep track of the path between the start vertex and the current vertex
- As soon as a back edge $(v, w)$ is encountered, we return the cycle as the portion of the stack from the top to vertex $w$

```
Algorithm cycleDFS(G, v)
1. setLabel(v, VISITED)
2. S.push(v)
3. for each e ∈ G.outgoingEdges(v) do
4.   if getLabel(e) = UNEXPLORED then
5.     w ← G.opposite(v, e)
6.     S.push(e)
7.     if getLabel(w) = UNEXPLORED then
8.       setLabel(e, DISCOVERY)
9.       cycleDFS(G, w)
10.      S.pop()
11.     else
12.       T ← empty stack
13.       repeat
14.         T.push(S.pop())
15.       until T.top() = w
16.       return T.elements()
17.     S.pop()
```
We can specialize the traversal algorithms (DFS and BFS) to digraphs by traversing edges only along their direction.

In the directed DFS algorithm, we have four types of edges:
- discovery edges
- back edges
- forward edges
- cross edges

A directed DFS starting at a vertex \( s \) determines the vertices reachable from \( s \).
REACHABILITY

• DFS tree rooted at $v$: vertices reachable from $v$ via directed paths
STRONG CONNECTIVITY

- Each vertex can reach all other vertices
STRONG CONNECTIVITY ALGORITHM

- Pick a vertex $v$ in $G$
- Perform a DFS from $v$ in $G$
  - If there’s a $w$ not visited, print “no”
- Let $G'$ be $G$ with edges reversed
- Perform a DFS from $v$ in $G'$
  - If there’s a $w$ not visited, print “no”
  - Else, print “yes”
- Running time: $O(n + m)$
STRONGLY CONNECTED COMPONENTS

• Maximal subgraphs such that each vertex can reach all other vertices in the subgraph
• Can also be done in $O(n + m)$ time using DFS, but is more complicated (similar to biconnectivity).

\[
\begin{align*}
\{a, c, g\} \\
\{f, d, e, b\}
\end{align*}
\]
BREADTH-FIRST SEARCH
BREADTH-FIRST SEARCH

- **Breadth-first search (BFS)** is a general technique for traversing a graph
- A BFS traversal of a graph $G$
  - Visits all the vertices and edges of $G$
  - Determines whether $G$ is connected
  - Computes the connected components of $G$
  - Computes a spanning forest of $G$
- BFS on a graph with $n$ vertices and $m$ edges takes $O(n + m)$ time
- BFS can be further extended to solve other graph problems
  - Find and report a path with the minimum number of edges between two given vertices
  - Find a simple cycle, if there is one
BFS ALGORITHM

• The algorithm uses a mechanism for setting and getting "labels" of vertices and edges

Algorithm BFS(G)
Input: Graph G
Output: Labeling of the edges and partition of the vertices of G
1. for each v ∈ G.vertices() do
2. setLabel(v, UNEXPLORED)
3. for each e ∈ G.edges() do
4. setLabel(e, UNEXPLORED)
5. for each v ∈ G.vertices() do
6. if getLabel(v) = UNEXPLORED then
7. BFS(G,v)

Algorithm BFS(G,s)
1. L₀ ← {s}
2. setLabel(s, VISITED)
3. i ← 0
4. while ¬Lᵢ.isEmpty() do
5. Lᵢ₊₁ ← Ø
6. for each v ∈ Lᵢ do
7. for each e ∈ G.outgoingEdges(v) do
8. if getLabel(e) = UNEXPLORED then
9. w ← G.opposite(v,e)
10. if getLabel(w) = UNEXPLORED then
11. setLabel(e, DISCOVERY)
12. setLabel(w, VISITED)
13. Lᵢ₊₁ ← Lᵢ₊₁ ∪ {w}
14. else
15. setLabel(e, CROSS)
16. i ← i + 1
EXAMPLE

unexplored vertex
visited vertex
unexplored edge
discovery edge
cross edge
EXAMPLE

discovery edge
cross edge
visited vertex
unexplored vertex
unexplored edge
discovery edge
cross edge
EXERCISE
BFS ALGORITHM

• Perform BFS of the following graph, start from vertex A
  • Assume adjacent edges are processed in alphabetical order
  • Number vertices in the order they are visited and note the level they are in
  • Label edges as discovery or cross edges
PROPERTIES

• Notation
  • $G_s$: connected component of $s$

• Property 1
  • $\text{BFS}(G, s)$ visits all the vertices and edges of $G_s$

• Property 2
  • The discovery edges labeled by $\text{BFS}(G, s)$ form a spanning tree $T_s$ of $G_s$

• Property 3
  • For each vertex $v \in L_i$
    • The path of $T_s$ from $s$ to $v$ has $i$ edges
    • Every path from $s$ to $v$ in $G_s$ has at least $i$ edges
ANALYSIS

• Setting/getting a vertex/edge label takes $O(1)$ time

• Each vertex is labeled twice
  • once as UNEXPLORED
  • once as VISITED

• Each edge is labeled twice
  • once as UNEXPLORED
  • once as DISCOVERY or CROSS

• Each vertex is inserted once into a sequence $L_i$

• Method outgoingEdges() is called once for each vertex

• BFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure
  • Recall that $\Sigma_v \deg(v) = 2m$
APPLICATIONS

• Using the template method pattern, we can specialize the BFS traversal of a graph $G$ to solve the following problems in $O(n + m)$ time
  • Compute the connected components of $G$
  • Compute a spanning forest of $G$
  • Find a simple cycle in $G$, or report that $G$ is a forest
  • Given two vertices of $G$, find a path in $G$ between them with the minimum number of edges, or report that no such path exists
<table>
<thead>
<tr>
<th>Applications</th>
<th>DFS</th>
<th>BFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spanning forest, connected components, paths, cycles</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Shortest paths</td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>Biconnected components</td>
<td>√</td>
<td></td>
</tr>
</tbody>
</table>

**Applications**

- DFS
  - Spanning forest, connected components, paths, cycles
  - Shortest paths
  - Biconnected components

- BFS
  - Spanning forest, connected components, paths, cycles
  - Shortest paths
  - Biconnected components
**DFS VS. BFS**

**Back edge** \((v, w)\)
- \(w\) is an ancestor of \(v\) in the tree of discovery edges

**Cross edge** \((v, w)\)
- \(w\) is in the same level as \(v\) or in the next level in the tree of discovery edges
TOPOLOGICAL ORDERING
DAGS AND TOPOLOGICAL ORDERING

• A directed acyclic graph (DAG) is a digraph that has no directed cycles

• A topological ordering of a digraph is a numbering
  • $v_1, ..., v_n$
  • Of the vertices such that for every edge $(v_i, v_j)$, we have $i < j$

• Example: in a task scheduling digraph, a topological ordering a task sequence that satisfies the precedence constraints

• Theorem - A digraph admits a topological ordering if and only if it is a DAG
• Scheduling: edge \((a, b)\) means task \(a\) must be completed before \(b\) can be started
EXERCISE
TOPOLOGICAL SORTING

• Number vertices, so that \((u, v)\) in \(E\) implies \(u < v\)

A typical student day

wake up → study computer sci. → eat → nap → more c.s. → play → write c.s. program → bake cookies → sleep → dream about graphs → work out
EXERCISE
TOPOLOGICAL SORTING

• Number vertices, so that \((u, v)\) in \(E\) implies \(u < v\)
ALGORITHM FOR TOPOLOGICAL SORTING

Algorithm TopologicalSort(G)
1. \( H \leftarrow G \)
2. \( n \leftarrow G.\text{numVertices}() \)
3. while \( \neg H.\text{isEmpty}() \) do
4. Let \( v \) be a vertex with no outgoing edges
5. Label \( v \leftarrow n \)
6. \( n \leftarrow n - 1 \)
7. \( H.\text{removeVertex}(v) \)
IMPLEMENTATION WITH DFS

• Simulate the algorithm by using depth-first search
• $O(n + m)$ time.

**Algorithm topologicalDFS($G$)**

Input: DAG $G$
Output: Topological ordering of $G$

1. $n \leftarrow G$.numVertices()
2. Initialize all vertices as $UNEXPLOR ED$
3. for each vertex $v \in G$.vertices() do
   4. if getLabel($v$) = $UNEXPLOR ED$ then
      5. topologicalDFS($G,v$)

**Algorithm topologicalDFS($G,v$)**

Input: DAG $G$, start vertex $v$
Output: Labeling of the vertices of $G$ in the connected component of $v$

1. setLabel($v, VISITED$)
2. for each $e \in G$.outgoingEdges($v$) do
   3. $w \leftarrow G$.opposite($v,e$)
   4. if getLabel($w$) = $UNEXPLOR ED$ then
      5. // $e$ is a discovery edge
   6. topologicalDFS($G,w$)
   7. else
      8. // $e$ is a forward, cross, or back edge
      9. Label $v$ with topological number $n$
10. $n \leftarrow n - 1$
TOPOLOGICAL SORTING EXAMPLE
TOPOLOGICAL SORTING EXAMPLE
TOPOLOGICAL SORTING EXAMPLE
TOPOLOGICAL SORTING EXAMPLE
TOPOLOGICAL SORTING EXAMPLE
TOPOLOGICAL SORTING EXAMPLE
TOPOLOGICAL SORTING EXAMPLE
TOPOLOGICAL SORTING EXAMPLE
TOPOLOGICAL SORTING EXAMPLE

Diagram showing a directed graph with nodes numbered 1 to 9, illustrating a topological sort example.