Inf 2B: Graphs, BFS, DFS

Kyriakos Kalorkoti

School of Informatics
University of Edinburgh
A graph is a mathematical structure consisting of a set of vertices and a set of edges connecting the vertices.

Formally: $G = (V, E)$, where $V$ is a set and $E \subseteq V \times V$.

For edge $e = (u, v)$ we say that $e$ is directed from $u$ to $v$.

$G = (V, E)$ undirected if for all $v, w \in V$:

$$(v, w) \in E \iff (w, v) \in E.$$

Otherwise directed.

Directed $\sim$ arrows (one-way)
Undirected $\sim$ lines (two-way)

We assume $V$ is finite, hence $E$ is also finite.
A directed graph

\[ G = (V, E), \]
\[ V = \{0, 1, 2, 3, 4, 5, 6\}, \]
\[ E = \{(0, 2), (0, 4), (0, 5), (1, 0), (2, 1), (2, 5),
(3, 1), (3, 6), (4, 0), (4, 5), (6, 3), (6, 5)\}. \]
An undirected graph
Examples

- *Road Maps.*
  Edges represent streets and vertices represent crossings.

- *Computer Networks.*
  Vertices represent computers and edges represent network connections (cables) between them.

  Vertices represent webpages, and edges represent hyperlinks.

- ...
Adjacency matrices

Let $G = (V, E)$ be a graph with $n$ vertices. Vertices of $G$ are numbered $0, \ldots, n - 1$.

The adjacency matrix of $G$ is the $n \times n$ matrix

$$A = (a_{ij})_{0 \leq i, j \leq n - 1}$$

with

$$a_{ij} = \begin{cases} 
1, & \text{if there is an edge from vertex } i \text{ to vertex } j; \\
0, & \text{otherwise.}
\end{cases}$$
Adjacency matrix (Example)

\[
\begin{pmatrix}
0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 0 & 0
\end{pmatrix}
\]
Adjacency lists

Array with one entry for each vertex $v$, which is a list of all vertices adjacent to $v$.

Example
Quick Question

Given: graph \( G = (V, E) \), with \( n = |V|, m = |E| \).
For \( v \in V \), we write \( in(v) \) for in-degree, \( out(v) \) for out-degree.

Which data structure has faster (asymptotic) worst-case running-time, for checking if \( w \) is adjacent to \( v \), for a given pair of vertices?

1. Adjacency list is faster.
2. Adjacency matrix is faster.
3. Both have the same asymptotic worst-case running-time.
4. It depends.

**Answer:** 2. For an Adjacency Matrix we can check in \( \Theta(1) \) time. An adjacency list structure takes \( \Theta(1 + out(v)) \) time.
Quick Question

Given: graph $G = (V, E)$, with $n = |V|$, $m = |E|$. For $v \in V$, we write $in(v)$ for in-degree, $out(v)$ for out-degree.

Which data structure has faster (asymptotic) worst-case running-time, for visiting all vertices $w$ adjacent to $v$, for a given vertex $v$?

1. Adjacency list is faster.
2. Adjacency matrix is faster.
3. Both have the same asymptotic worst-case running-time.
4. It depends.

Answer: 3. Adjacency matrix requires $\Theta(n)$ time always. Adjacency list requires $\Theta(1 + out(v))$ time. In worst-case $out(v) = \Theta(n)$. 
## Adjacency Matrices vs Adjacency Lists

<table>
<thead>
<tr>
<th></th>
<th>adjacency matrix</th>
<th>adjacency list</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Space</strong></td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(n + m)$</td>
</tr>
<tr>
<td><strong>Time to check if $w$ adjacent to $v$</strong></td>
<td>$\Theta(1)$</td>
<td>$\Theta(1 + out(v))$</td>
</tr>
<tr>
<td><strong>Time to visit all $w$ adjacent to $v$.</strong></td>
<td>$\Theta(n)$</td>
<td>$\Theta(1 + out(v))$</td>
</tr>
<tr>
<td><strong>Time to visit all edges</strong></td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(n + m)$</td>
</tr>
</tbody>
</table>
Sparse and dense graphs

\( G = (V, E) \) graph with \( n \) vertices and \( m \) edges

Observation: \( m \leq n^2 \)

- \( G \) dense if \( m \) close to \( n^2 \)
- \( G \) sparse if \( m \) much smaller than \( n^2 \)
Graph traversals

A traversal is a strategy for visiting all vertices of a graph while respecting edges.

\[
\text{BFS} = \text{breadth-first search} \\
\text{DFS} = \text{depth-first search}
\]

General strategy:
1. Let \( v \) be an arbitrary vertex
2. Visit all vertices reachable from \( v \)
3. If there are vertices that have not been visited, let \( v \) be such a vertex and go back to (2)
Algorithm searchFromVertex\((G, v)\)

1. mark \(v\)
2. put \(v\) onto schedule \(S\)
3. while schedule \(S\) is not empty do
4. remove a vertex \(v\) from \(S\)
5. for all \(w\) adjacent to \(v\) do
6. if \(w\) is not marked then
7. mark \(w\)
8. put \(w\) onto schedule \(S\)

Algorithm search\((G)\)

1. initialise schedule \(S\)
2. for all \(v \in V\) do
3. if \(v\) is not marked then
4. searchFromVertex\((G, v)\)
BFS

Visit all vertices reachable from \( v \) in the following order:

- \( v \)
- all neighbours of \( v \)
- all neighbours of neighbours of \( v \) that have not been visited yet
- all neighbours of neighbours of neighbours of \( v \) that have not been visited yet
- etc.
Algorithm \texttt{bfs}(G)

1. Initialise Boolean array \textit{visited}, setting all entries to \texttt{FALSE}.
2. Initialise \textit{Queue} \texttt{Q}
3. \texttt{for all} \( v \in V \) \texttt{do}
4. \hspace{1em} \texttt{if} \ \textit{visited}[v] = \texttt{FALSE} \ \texttt{then}
5. \hspace{2em} \texttt{bfsFromVertex}(G, v)
Algorithm bfsFromVertex($G, v$)

1. $visited[v] = \text{TRUE}$
2. $Q$.enqueue($v$)
3. while not $Q$.isEmpty() do
4. \hspace{1em} $v \leftarrow Q$.dequeue()
5. \hspace{1em} for all $w$ adjacent to $v$ do
6. \hspace{2em} if $visited[w] = \text{FALSE}$ then
7. \hspace{3em} \hspace{1em} $visited[w] = \text{TRUE}$
8. \hspace{2em} $Q$.enqueue($w$)
Quick Question

Given a graph $G = (V, E)$ with $n = |V|$, $m = |E|$, what is the worst-case running time of BFS, in terms of $m, n$?

1. $\Theta(m + n)$
2. $\Theta(n^2)$
3. $\Theta(mn)$
4. Depends on the number of components.

Answer: 1. To see this need to be careful about bounding running time for the loop at lines 5–8. Must use the Adjacency List structure.

Answer: 2. if we use adjacency matrix representation.
DFS

Visit all vertices reachable from \( v \) in the following order:

- \( v \)
- some neighbour \( w \) of \( v \) that has not been visited yet
- some neighbour \( x \) of \( w \) that has not been visited yet
- etc., until the current vertex has no neighbour that has not been visited yet
- Backtrack to the first vertex that has a yet unvisited neighbour \( v' \).
- Continue with \( v' \), a neighbour, a neighbour of the neighbour, etc., backtrack, etc.
**Algorithm** dfs($G$)

1. Initialise Boolean array $visited$, setting all to FALSE
2. Initialise $Stack$ $S$
3. **for all** $v \in V$ **do**
4. \hspace{1cm} **if** $visited[v] = \text{FALSE}$ **then**
5. \hspace{2cm} dfsFromVertex($G, v$)
Algorithm dfsFromVertex(G, v)

1. S.push(v)
2. while not S.isEmpty() do
3.     v ← S.pop()
4.     if visited[v] = FALSE then
5.         visited[v] = TRUE
6.     for all w adjacent to v do
7.         S.push(w)
Recursive DFS

**Algorithm dfs(G)**

1. Initialise Boolean array *visited* by setting all entries to FALSE
2. for all \( v \in V \) do
3. if \( visited[v] = FALSE \) then
4. dfsFromVertex\((G, v)\)

**Algorithm dfsFromVertex(G, v)**

1. \( visited[v] \leftarrow TRUE \)
2. for all \( w \) adjacent to \( v \) do
3. if \( visited[w] = FALSE \) then
4. dfsFromVertex\((G, w)\)
Analysis of DFS

\( G = (V, E) \) graph with \( n \) vertices and \( m \) edges

Without recursive calls:

- \( \text{dfs}(G) \): time \( \Theta(n) \)
- \( \text{dfsFromVertex}(G, v) \): time \( \Theta(1 + \text{out-degree}(v)) \)

Overall time:

\[
T(n, m) = \Theta(n) + \sum_{v \in V} \Theta(1 + \text{out-degree}(v)) = \Theta\left(n + \sum_{v \in V} (1 + \text{out-degree}(v))\right) = \Theta\left(n + n + \sum_{v \in V} \text{out-degree}(v)\right) = \Theta\left(n + \sum_{v \in V} \text{out-degree}(v)\right) = \Theta(n + m)
\]