Exercise 22.1-1

Given an adjacency-list representation of a directed graph, how long does it take to compute the out-degree of every vertex? How long does it take to compute the in-degrees?

Solution:

Given an adjacency-list representation $Adj$ of a directed graph, the out-degree of a vertex $u$ is equal to the length of $Adj[u]$, and the sum of the lengths of all the adjacency lists in $Adj$ is $|E|$. Thus the time to compute the out-degree of every vertex is $\Theta(|V| + |E|)$.

The in-degree of a vertex $u$ is equal to the number of times it appears in all the lists in $Adj$. If we search all the lists for each vertex, the time to compute the in-degree of every vertex is $\Theta(|V| |E|)$.

(Alternatively, we can allocate an array $T$ of size $|V|$ and initialize its entries to zero. Then we only need to scan the lists in $Adj$ once, incrementing $T[u]$ when we see $u$ in the lists. The values in $T$ will be the in-degrees of every vertex. This can be done in $\Theta(|V| + |E|)$ time with $\Theta(|V|)$ additional storage.)
Exercise 22.1-2

Give an adjacency-list representation for a complete binary tree on 7 vertices. Give an equivalent adjacency-matrix representation. Assume that vertices are numbered from 1 to 7 as in a binary heap.

Solution:

A complete binary tree looks like:

```
   1
  / \
 2   3
 /   /
4   5   6
  \  / \
   7
```

An adjacency-list representation of this tree is shown below:

```
1 -> 2, 3
2 -> 1, 4
3 -> 1, 7
4 -> 2, 6
5 -> 4, 6
6 -> 5, 7
7 -> 3
```

An equivalent adjacency-matrix representation of this tree is shown below:

```
   1 2 3 4 5 6 7
1 [0, 1, 1, 0, 0, 0, 0]
2 [1, 0, 0, 1, 1, 0, 0]
3 [1, 0, 0, 0, 0, 1, 1]
4 [0, 1, 0, 0, 0, 0, 0]
5 [0, 1, 0, 0, 0, 0, 0]
6 [0, 0, 1, 0, 0, 0, 0]
7 [0, 0, 1, 0, 0, 0, 0]
```
Exercise 22.2-1

Show the dist and pred values that result from running breadth-first search on the directed graph below, using vertex 3 as the source.

Solution:

The procedure of the breadth-first search is shown above. From the result, we can see that:

<table>
<thead>
<tr>
<th>Node</th>
<th>Pred</th>
<th>dist</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>NIL</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>NIL</td>
<td>∞</td>
</tr>
</tbody>
</table>
Exercise 22.2-2

Show the dist and pred values that result from running breadth-first search on the undirected graph below, using vertex u as the source.

Solution:

(a)  
(b)  
(c)  
(d)  
(e)  
(f)  
(g)  
(h)  
(i)  

Q u

Q r

Q s

Q v

Q w

Q ∅
The procedure of the breadth-first search is shown above. From the result, we can see that:

<table>
<thead>
<tr>
<th>Node</th>
<th>Pred</th>
<th>dist</th>
</tr>
</thead>
<tbody>
<tr>
<td>u</td>
<td>NIL</td>
<td>0</td>
</tr>
<tr>
<td>t</td>
<td>u</td>
<td>1</td>
</tr>
<tr>
<td>x</td>
<td>u</td>
<td>1</td>
</tr>
<tr>
<td>y</td>
<td>u</td>
<td>1</td>
</tr>
<tr>
<td>w</td>
<td>t</td>
<td>2</td>
</tr>
<tr>
<td>s</td>
<td>w</td>
<td>3</td>
</tr>
<tr>
<td>r</td>
<td>s</td>
<td>4</td>
</tr>
<tr>
<td>v</td>
<td>r</td>
<td>5</td>
</tr>
</tbody>
</table>

**Exercise 22.3-1**

Make a 3-by-3 chart with row and column labels WHITE, GRAY, and BLACK. In each cell \((i, j)\), indicate whether, at any point during a depth-first search of directed graph, there can be an edge from a vertex of color \(i\) to a vertex of color \(j\). For each possible edge, indicate what edge types it can be. Make a second such chart for depth-first search of an undirected graph.

**Solution:**

**Directed graph:**

<table>
<thead>
<tr>
<th></th>
<th>WHITE</th>
<th>GRAY</th>
<th>BLACK</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHITE</td>
<td>tree, back, forward, and cross</td>
<td>back and cross</td>
<td>cross</td>
</tr>
<tr>
<td>GRAY</td>
<td>tree and forward</td>
<td>tree, forward, back</td>
<td>tree, forward, and cross</td>
</tr>
<tr>
<td>BLACK</td>
<td>×</td>
<td>back and cross</td>
<td>tree, forward, back and cross</td>
</tr>
</tbody>
</table>

**Undirected graph:**

<table>
<thead>
<tr>
<th></th>
<th>WHITE</th>
<th>GRAY</th>
<th>BLACK</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHITE</td>
<td>tree and back</td>
<td>tree and back</td>
<td>×</td>
</tr>
<tr>
<td>GRAY</td>
<td>tree and back</td>
<td>tree and back</td>
<td>tree and back</td>
</tr>
<tr>
<td>BLACK</td>
<td>×</td>
<td>tree and back</td>
<td>tree and back</td>
</tr>
</tbody>
</table>

**Exercise 22.3-2**

Show how depth-first search works on the graph of Figure 22.6. Assume that the for loop of lines 5–7 of the DFS procedure considers the vertices in alphabetical order, and assume that each adjacency list is ordered alphabetically. Show the discovery and finishing times for each vertex, and show the classification of each edge.

![Figure 22.6](image)

*Figure 22.6*  A directed graph for use in Exercises 22.3-2 and 22.5-2.
Solution:

The discovery and finishing times for each vertex are shown in the figure below:

Tree edges: (q, s), (s, v), (v, w), (q, t), (t, x), (x, z), (t, y), (r, u)
Back edges: (w, s), (z, x), (y, q)
Forward edges: (q, w)
Cross edges: (r, y), (u, y)

Exercise 22.3-3

Show the parenthesis structure of the depth-first search of Figure 22.4.

Solution:

Exercise 22.3-7

Rewrite the procedure DFS, using a stack to eliminate recursion.

Solution:

Assume that the stack has following operations:

- PUSH(S, v) – pushes v into the stack;
- POP(S) – returns the top of the stack and removes it;
- TOP(S) – returns the top of the stack without removing it.
Denote an empty stack by \texttt{EMPTY}.

Then the pseudocode of this algorithm becomes as follows:

\begin{verbatim}
DFS_STACK (G, s)
    for each vertex \( u \in V(G) - \{s\} \)
    \{
        u.color = \texttt{WHITE}
        u.pred = \texttt{NIL}
    \}
    time = 0
    \( S = \texttt{EMPTY} \)

    time = time + 1
    s.td = time
    s.color = \texttt{GRAY}
    PUSH(\( S, s \))

    while \( S \neq \texttt{EMPTY} \)
    \{
        t = \texttt{Top}(S)
        if \( \exists v \in V(G).\text{Adj}[t] \text{ s.t. } v.color == \texttt{WHITE} \) // v's adjacency list hasn't been fully examined
        \{
            v.pred = t
            time = time + 1
            v.td = time
            v.color = \texttt{GRAY}
            PUSH(\( S, v \))
        \}
        else \hspace{1cm} // v's adjacency list has been fully examined
        \{
            t = Pop(\( S \))
            time = time + 1
            v.tf = time
            v.color = \texttt{BLACK}
        \}
    }
\end{verbatim}

Exercise 22.4-1

Show the ordering of vertices produced by \texttt{TOPOLOGICAL-SORT} when it is run on the dag of Figure 22.8, under the assumption of Exercise 22.3-2.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{topo_sort_dag.png}
\caption{A dag for topological sorting.}
\end{figure}

Solution:

According to the assumption of Exercise 22.3-2, the \texttt{for} loop of lines 5–7 of the DFS procedure...
considers the vertices in alphabetical order, and that each adjacency list is ordered alphabetically.

DFS on Figure 22.8:

Ordering of vertices:

Exercise 22.4-2

Give a linear-time algorithm that takes as input a directed acyclic graph $G = (V, E)$ and two vertices $s$ and $t$, and returns the number of simple paths from $s$ to $t$ in $G$. For example, the directed acyclic graph of Figure 22.8 contains exactly four simple paths from vertex $p$ to vertex $v$: $pov$, $poryv$, $porysv$, and $psryv$. (Your algorithm needs only to count the simple paths, not list them.)

Figure 22.8  A dag for topological sorting.

Solution:

Add a field to the vertex representation to hold an integer count. Initially, set vertex $t$’s count to 1 and other vertices’ count to 0. Start running DFS with $s$ as the start vertex. When $t$ is discovered, it should be immediately marked as finished (BLACK), without further processing starting from it. Subsequently, each time DFS finishes a vertex $v$, set $v$’s count to the sum of the counts of all vertices adjacent to $v$. When DFS finishes vertex $s$, stop and return the count computed for $s$. 