Graph Traversal: Depth-First Search

To traverse it means:

Produce a list of all the vertices that can be reached starting at some vertex
Come fly high with us!
High Planes Airline (HPAir)
Searching for a flight path

For each customer request $oC \rightarrow dC$, find a directed path of HPAir flights from the origin city $oC$ to the destination city $dC$.
- e.g. $P \rightarrow Z$ returns $[(P, W), (W, Y), (Y, Z)]$

The flight map is a digraph
- Arc $(a,b)$ between vertices means
  - There is a flight from city $a$ to city $b$
- Directed path means
  - There is a sequence of flight connections
Draw your Adjacency Matrix

<table>
<thead>
<tr>
<th></th>
<th>P</th>
<th>Q</th>
<th>R</th>
<th>S</th>
<th>T</th>
<th>W</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td></td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The diagram on the right shows the connections between the nodes, which correspond to the entries in the table.
Depth-First Search (Non-recursive solution)

- The solution performs an exhaustive search
  - Beginning at the origin city, tries every possible sequence of flights until either
    - Finds a sequence that gets to the destination city
    - Determines that no such sequence exists

- Keep going deep in the graph as much as you can selecting among adjacent vertices
- Backtrack (if you must)
  to recover from a choice that did not reach the city
- Repeat the above steps until successful in finding the sequence or failure to reach destination

- Which data structure is useful in backtracking?
  - It should help you remember how you got the the current point
DFS(origin, destination): Search the Map

```java
stk = new Stack<E>();

stk.push(origin);

while (stk is not empty) {
    if (you can find an unvisited neighbor, anotherCity, from the city on top of stack)
        stk.push(anotherCity);
    else // cannot find an unvisited neighbor
        stk.pop(); // backtrack
```
... and remember where you’ve been

stk = new Stack<E>();

Mark all nodes as not-visited (yet);

stk.push(origin);

Mark(origin) as visited;

while (stk is not empty) {

    if (you can find an unvisited neighbor, anotherCity, from the city on top of stack)

        stk.push(anotherCity);
        Mark(anotherCity) as visited;

    else // cannot find an unvisited neighbor

        stk.pop(); // backtrack

}
Depth-First-Search Example: From P->Z

```java
stk = new Stack<>();
Mark all nodes as not visited yet;
stk.push(origin);
Mark(origin) as visited;
while (stk is not empty) {
    if (you can find an unvisited neighbor, anotherCity, from the city on top of stack)
        stk.push(anotherCity);
        Mark(anotherCity) as visited;
    else //cannot find an unvisited neighbor
        stk.pop(); // backtrack
}
```

Marked (visited)
Would DFS work for undirected graphs?

```java
stk = new Stack<E>();
Mark all nodes as not visited yet;
stk.push(origin);
Mark(origin) as visited;
while (stk is not empty) {
    if (you can find an unvisited neighbor, anotherCity, from the city on top of stack)
        stk.push(anotherCity);
        Mark(anotherCity) as visited;
    else // cannot find an unvisited neighbor
        stk.pop(); // backtrack
}
```
DFS Searching a Maze

Try rooms in some order (e.g. left, down, right, up)
“To find the way out of a labyrinth there is only one means. At every new junction never seen before, the path we have taken will be marked with three signs. If you see that the junction has already been visited, you will make only one mark on the path you have taken. If all the apertures have already been marked, then you must retrace your steps. But if one or two apertures of the junction are still without signs, you will choose any one, making another sign on it. Proceeding through an aperture that bears only one sign, you will make two more, so that now the aperture bears three.”
Testing for Connectivity using DFS(oC)

Connected: An undirected graph for which there is a path from any node to any other node

Is this graph connected?

Connected component: A connected sub-graph

Can we use DFS to find all connected components?
An undirected graph is connected if and only if for each vertex $v$ in a graph containing $n$ vertices, the size of the result of a breadth-first traversal at $v$ is $n$. 
Assuming you began at node a, give the order of traversal if you visited every node.

**DFS:**
From Parking to Parking
WC Campus Directed Graph
WC Campus DAG
**Strong Connectivity**

- **Strongly Connected**: A graph for which there is a directed path from any node to any other node.

- Is this graph strongly connected?

- **Strongly connected component**: A strongly connected sub-graph.

- Can you find the strongly connected components of this graph?
private ArrayIterator<T> iteratorDFS(int startIndex) {

    LinkedStack<Integer> traversalStack = new LinkedStack<Integer>();
    ArrayIterator<T> dfsIter = new ArrayIterator<T>();
    boolean[] visited = new boolean[getNumVertices()];

    for (int vID = 0; vID < getNumVertices(); vID++)
        visited[vID] = false;

    traversalStack.push(startIndex);
    dfsIter.add(vertices.get(startIndex));
    visited[startIndex] = true;

    while (!traversalStack.isEmpty()){
        currentVertex = traversalStack.peek(); found = false;
        for (int vID = 0; vID < getNumVertices() && !found;
            vID++)
            if (isArc(currentVertex, vID) && !visited[vID]) {
                traversalStack.push(vID);
                dfsIter.add(vertices.get(vID));
                visited[vID] = true;
                found = true;
            }
    }

stk = new Stack<E>();
Mark all nodes as not visited yet;
stk.push(origin);
Mark(origin) as visited;
while(stk is not empty) {
    if (you can find an unvisited neighbor, anotherCity, from the city on top of stack)
        stk.push(anotherCity);
        Mark(anotherCity) as visited
else // cannot find an unvisited neighbor
    stk.pop(); // backtrack