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

The flight map is a digraph:

- Arc \((a,b)\) between vertices \(a, b\) indicates that there is a direct flight from city \(a\) to city \(b\)

To search\((oC, dC)\) it means:

- find a directed path of HPAir flights from origin City \(oC\) to destination City \(dC\)
- \(\text{search } (P)\) returns all the Cities that the algorithm encountered in its search
  vs
- \(\text{search } (P, Z)\) returns directed path \([(P,W), (W,Y),(Y,Z)]\)
  where \(W, Y\) are cities of intermediate stopovers
Draw your Adjacency Lists
Depth-First Search Algorithm
(An exhaustive search)

- **Keep going deep** in the graph as far as you can by selecting one of adjacent vertices.

- **Backtrack** (if you must) to recover from a choice that did not reach the city so you can select another adjacent vertex.

- **Repeat** the above steps until
  - You were successful in finding the sequence or
  - You could not reach the destination.

- But how do you select an adjacent vertex?
  - Look at each and every vertex on the AdjLists.

- And how do you know where to backtrack to?
  - pop a stack! Like the undo command does!
backStack = new Stack<E>(); // to backtrack
remember origin: backStack.push(origin)
while (there is a City on top of the stack)
{
  look at each City on the map, one-by-one
  if (see a connecting flight to City aC)
    you found a City to fly to: aC
    stk.push(aC);
  else if you cannot find a flight
    backtrack:
    stk.pop(); // and try again
}
DFS(origin): Search the whole Map ... but remember what you visited

backStack = new Stack<E>(); // to backtrack

remember origin: backStack.push(origin)

mark that you have visited no city yet

while (there is a City on top of the stack)
{
    look at each unmarked City on the map, one-by-one
    if (see a connecting flight to City aC)
        you found a City to fly to: aC
        stk.push(aC); mark(aC) as visited
    else if you cannot find a flight
        backtrack:
        stk.pop(); // and try again
}
DFS(origin): Search the whole Map ... but remember what you visited

backStack = new Stack<E>(); // to backtrack
remember origin: backStack.push(origin)
mark that you have visited no city yet
while (there is a City on top of the stack)
{   look at each unmarked City on the map,
    if (see a connecting flight to City aC)
        you found a City to fly to: aC
        stk.push(aC); mark(aC) as visited
    else if you cannot find a flight
        backtrack:
            stk.pop(); // and try again
}
backStack
And the answer is…

- We reached Z, how did we get there?
  - the backStack contains the answer!

- But do we want to search the same graph every time someone asks for a flight?
  - No, better remember your DFS path all the way back to origin

- How?
  - Keep the whole DFS path onto some a data structure (array, LL, something else)
DFS(origin): Search the whole Map ... keep a list of what you visited

```java
private <T> LinkedList<T> DFS(int originIndex)
{
    Stack<Integer> backStack = new LinkedList<>();
    LinkedList<T> dfsLL = new LinkedList<>(); // to record dfs path
    boolean[] marked = new boolean[getNumVertices()];
    // created and initialized to false (not-yet visited)!

    backStack.push(originIndex); // Visit the origin city
    dfsLL.add(vertices.get(originIndex)); // and add it to the dfsLL
    marked[originIndex] = true; // mark that you visited the city

    // (cont)
```
Keep going, DFS!

... (cont.)

while (!backStack.isEmpty()) {// there are still cities on stk
    currentVertex = backStack.peek(); found = false; // any flights?
    for (int aC = 0; aC < getNumVertices() && !found; aC++) {
        // if you see a connecting flight to aC
        if (isArc(currentVertex,aC) && !marked[aC]) {
            backStack.push(aC); // visit aC
            dfsLL.add(vertices.get(aC)); // and record it to dfsLL
            marked[aC] = true; found = true; // mark aC as visited
        } //if
    }
    if (!found && !backStack.isEmpty()){
        // if can’t find another City, backtrack!
        backtrack.pop();
    } } // if, while
return dfsLL;
Let’s practice with the WC Campus Directed Graph

dfsLL:______________________________________________________________
Would DFS work on undirected graphs?

stk = new Stack<E>();
initialize boolean [] visited to not-yet!

Visit the current city: stk.push(origin)

while (there are still cities on the stk) {
    if (you see a connecting flight to aC)
        visit aC: stk.push(aC);
        mark visited[aC] as true
    else // cannot find another City
        backtrack: stk.pop();
}
DFS through the ages

It turns out it is an old idea...
Have you heard of the Labyrinth?
“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.”
DFS Searching a Maze

Try rooms in some order (e.g. East, South, West, North)
How about “connectivity” and “strong connectivity”

Just when you thought you were done...
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 you see them?

- Can we use DFS to find all connected components?
  - How?
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?