Graph Traversals

- A graph-traversal algorithm
  - Visits all the vertices that it can reach starting at some vertex
  - Visits all vertices of the graph
  - Must not loop forever, if a graph contains a cycle
  - Must never visit a vertex more than once

  - connected component (for undirected graphs) =
  - The subset of vertices visited during a traversal that begins at a given vertex

  - strongly connected component (for directed graphs) =
  - The subset of vertices visited during a traversal that begins at any of its members

A Non-Recursive Solution That Uses a Stack

- 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

- The ADT Stack is useful in organizing an exhaustive search
  - It helps you remember how you got to the current point

- Backtracking can be used to recover from a wrong choice of a city

A Search Problem

- High Planes Airline Company (HPAir) Problem
  - For each customer request, indicate whether a sequence of HPAir flights exists from the origin city to the destination city

- The flight map for HPAir is a graph
  - Arc between vertices
    - There is a flight between cities
  - Directed path
    - There is a sequence of flight connections

DFS(originCity): Searching the Flight Map

```java
stk = new Stack<>()
stk.push(originCity);

while (a sequence of flights from originCity to destinCity has not been found) {
    if (you cannot go anywhere from the city on top of stack)
        stk.pop(); // backtrack
    else
        select a neighbor, anotherCity,
        from the city on top of stack;
        stk.push(anotherCity);
}
```
... and remember where you’ve been

```java
stk = new Stack<E>(); Clear Marks;
stk.push(originCity);
Mark(originCity) as visited;
while (a sequence of flights from originCity to destinCity has not been found)
{
  if (you cannot find an unvisited city from the city on top of stack)
    stk.pop(); // backtrack
  else select an unvisited neighbor, anotherCity,
    from the city on top of stack;
    stk.push(anotherCity);
    Mark(anotherCity) as visited;
}
```

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

```java
stk = new Stack<E>(); Clear Marks;
stk.push(originCity);
Mark(originCity) as visited;
while (a sequence of flights from originCity to destinCity has not been found)
{
  if (you cannot find an unvisited city from the city on top of stack)
    stk.pop(); // backtrack
  else select an unvisited neighbor, anotherCity,
    from the city on top of stack;
    stk.push(anotherCity);
    Mark(anotherCity) as visited;
}
```

Would DFS(oC) work for undirected graphs?

Labyrinth: Umberto Eco advises Theseus

"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 two signs on it. Proceeding through an aperture that bears only one sign, you will make two more, so that now the aperture bears three. All the parts of the labyrinth must have been visited if, arriving at a junction, you never take a passage with three signs, unless none of the other passages is now without signs."
Mazes as Graphs

Testing for Connectivity using DFS(oC)

Strong Connectivity

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?

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?
The Web is a Graph

- Directed Graph of Nodes and Arcs
  - Nodes = web pages
  - Arcs = hyperlinks from one page to another
- A graph can be explored
- A graph can be indexed

URL
http://cs.wellesley.edu/~cs230/PPTs/Hash.html

Access method
Server and domain
Path
Document

The Web Graph - Starting at CS230 Home

CS230
Assignments
Instructor
CS Dept.
Syllabus
Documentation
PLTC
Textbook
...
...
...
...
...
...

Searching the Web

- The web can be considered a graph "the web graph"
- Web pages are the graph nodes
- Hyperlinks on pages are graph edges
- The web graph is huge (way over one million billions nodes) — maybe infinite (pages are created on the fly)
- For the web graph, DFS is not a good strategy. (Why?)
- You need to search your neighborhood before going deeper

The shape of the Web is ... a "bow-tie" (!)

Breadth First Example: BFS(9)

Queue: 9 6 7 8 3 4 5 1 2
Iterator:
**BFS(v) pseudocode**

// BFS traversal starting at v
Initialization:
Mark all vertices as unvisited
enQueue v onto a new queue Q
Mark v as visited
While (Q is not empty)
  deQueue a vertex w from Q
  For each unvisited vertex u adjacent to w:
    enQueue u onto Q
    Mark u as visited

BFS from S to G:
The BFS tree shows the visits
How do you remember the path?

**Dependency Graph on a DAG**

- Defined on a Directed Acyclic Graph (a "DAG")
- Usually reflect dependencies or requirements
  - I.e., Assembly lines, Supply lines, Organizational charts, ...
  - BTW: You cannot take 231 after 230 unless...
- Understanding dependencies requires "topological sorting"

**Resolving DAG Dependencies**

- **Topological order**
  - A list of vertices in a DAG such that
    - vertex x precedes vertex y iff there is a directed edge from x to y in the graph
  - There may be several topological orders in a given graph
- **Topological sorting**
  - Arranging the vertices into a topological order

**How do you remember the path?**

Initialization: enqueue path [S] in Q
While you have not reached G
  dequeue a path from BFS queue and check the last node x in the path
  extend the path to unvisited neighbors of x
  and enqueue extended paths to back of Q

... in the next step it will reach the goal G

...
A(nother) Topological Sorting Algorithm

- Select a vertex \( v \) that has no predecessor
- Remove \( v \) from the graph (along with all associated arcs),
- Add \( v \) to the end of a list of vertices \( L \)
- Repeat previous steps
- When the graph is empty, \( L \)'s vertices will be in topological order

A Topological Sorting Algorithm

- Select a vertex \( v \) that has no successor
- Remove \( v \) from the graph (along with all associated arcs),
- Add \( v \) to the beginning of a list of vertices \( L \)
- Repeat previous steps
- When the graph is empty, \( L \)'s vertices will be in topological order