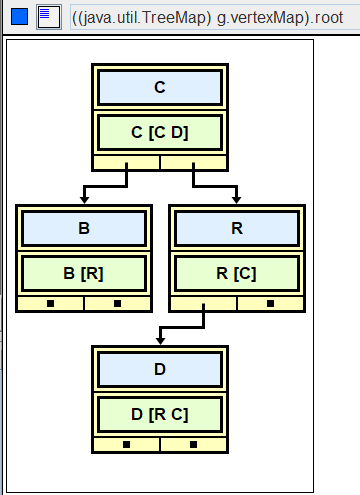
**Graphs 4:**

**Edge-List: DFS and BFS**

As we have mentioned before, a common task for graphs is to find the reachable vertices. This time we exhaustively traverse the edge-list representation, following the edges of each vertex until we reach an end or an already visited vertex.



|  |  |
| --- | --- |
| R | [C] |
| B | [R] |
| C | [C D] |
| D | [C R] |

If you start at R, can you get to D? \_\_

If you start at R, can you get to B? \_\_

Traversals come in two varieties, a *depth-first* search and a *breadth-first* search.

**Depth-first search**

A *depth-first* search traverses a graph and returns a list of reachable vertices. The order in which the vertices are processed is determined by a *stack*.

In what order are the vertices visited, if Vertex D is the source?

|  |  |
| --- | --- |
| B | [R] |
| C | [C D] |
| D | [R C] |
| R | [C] |

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |

|  |
| --- |
| First, create both a **list** of reachable vertices and a **stack.** |
| Push D’s adjacent vertices on the stack. |
| While the stack is not empty |
| Pop a vertex. If the vertex isn't in the list, put it in the list. |
| Process the vertex’s edges: push each vertex on the stack |
| Return the list of reachable vertices. |

Write the code:

public List<Vertex> depthFirstSearch(String name)

**Breadth-first search**

A *breadth-first* search also traverses a graph and returns a list of reachable vertices. The order in which the vertices are processed is determined by a *queue*. The algorithm is the same, but use a queue instead of a stack, changing the appropriate method calls. You should be able to visualize why a stack produces a depth-first search and a queue produces a breadth-first search. You should know that the breadth-first search produces the shortest path, shortest in the sense of visiting the fewest vertices (or edges.)

In what order are the vertices visited, if Vertex D is the source?

|  |  |
| --- | --- |
| B | [R] |
| C | [C D] |
| D | [R C] |
| R | [C] |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |

|  |
| --- |
| First, create both a **list** of reachable vertices and a **queue.** |
| Enqueue D’s adjacent vertices. |
| While the queue is not empty |
| Dequeue a vertex. If the vertex isn't in the list, put it in the list. |
| Process the vertex’s edges: enqueue each adjacent vertex. |
| Return the list of reachable vertices. |

**Two Multiple Choice Questions** Consider this graph:

v0 <------- v2

/ \

/ \

-> v1 <-/ \-> v4

/ \

/ \

/ \->v3 -------> v5

/ /

/ /

v6 <---------/

1. In what order are the vertices visited for a depth-first search (DFS) that starts at v0?
2. In what order are the vertices visited for a breadth-first search (BFS) that starts at v0?

**Assignment**

In AdjList, comment in the DFS\_BFS interface and implement it.

public class AdjList implements AdjListInterface, DFS\_BFS//, EdgeListWithCities

Here is the DFS\_BFS interface:

interface DFS\_BFS  
{  
 public List<Vertex> depthFirstSearch(String name);  
 public List<Vertex> breadthFirstSearch(String name);  
 /\* extra credit methods \*/  
 public List<Vertex> depthFirstRecur(String name);  
 public List<Vertex> depthFirstRecurHelper(Vertex v, List<Vertex> reachable);  
}

**Sample Run** (DFS\_BFS\_Driver.java using AdjList )

|  |
| --- |
| Edge List Representation!  B [R]  C [C D]  D [R C]  R [C]  Depth First Search  Enter source: B  [R [C], C [C D], D [R C]]  Enter source: C  [D [R C], C [C D], R [C]]  Enter source: D  [C [C D], D [R C], R [C]]  Enter source: R  [C [C D], D [R C], R [C]]  Enter source: -1  Breadth First Search  Enter source: B  [R [C], C [C D], D [R C]]  Enter source: C  [C [C D], D [R C], R [C]]  Enter source: D  [R [C], C [C D], D [R C]]  Enter source: R  [C [C D], D [R C], R [C]]  Enter source: -1 |

**Extension**

DFS can be implemented recursively. You have used this kind of recursion in previous labs: for every vertex you visit, process it, then recur on each of its edges.