Algoritma Traversal Graph

Graph Traversal Algorithms

- In general, graphs do not have a vertex, like a root, that initiates unique paths to each of the vertices. From any starting vertex in a graph, it might not be possible to search all of the vertices. In addition, a graph could have a cycle that results in multiple visits to a vertex.

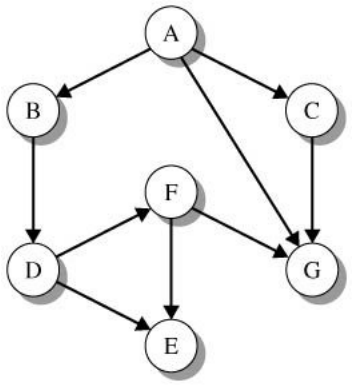
Graph Traversal Algorithms (continued)

- The breadth-first search visits vertices in the order of their path length from a starting vertex. It may not visit every vertex of the graph
- The depth-first search traverses all the vertices of a graph by making a series of recursive calls that follow paths through the graph.

Graph Traversal Algorithms (continued)

- Graph algorithms discern the state of a vertex during the algorithm by using the colors WHITE, GRAY, and BLACK.

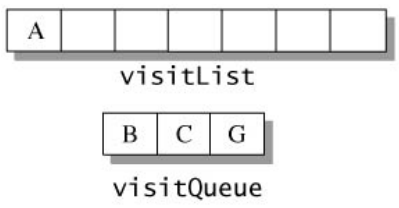
Breadth-First Search Algorithm



Demonstration graph
for the breadth-first search.

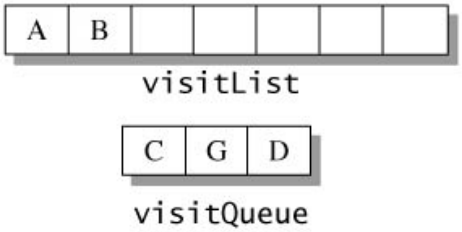
Breadth-First Search Algorithm (continued)

- Color all vertices of the sample graph WHITE and push the starting vertex (A) onto the queue visitQueue.
- Pop A from the queue, color it BLACK, and insert it into visitList, which is the list of visited vertices. Push all WHITE neighbors onto the queue.



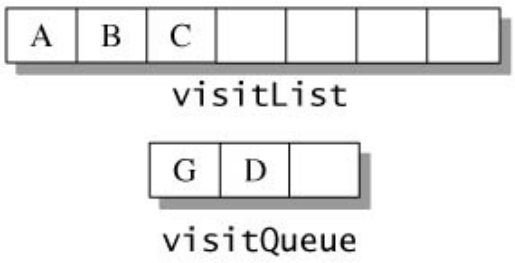
Breadth-First Search Algorithm (continued)

- Pop B from the queue and place it in visitList with color BLACK. The only adjacent vertex for B is D, which is still colored WHITE. Color D GRAY and add it to the queue



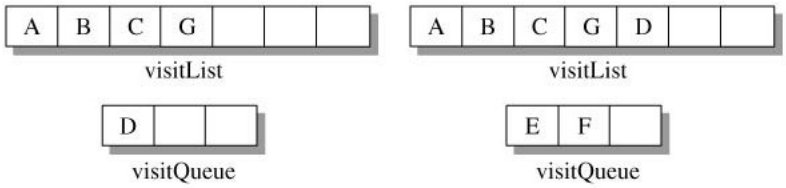
Breadth-First Search Algorithm (continued)

- Pop C and place it in visitList. The adjacent vertex G is GRAY. No new vertices enter the queue.



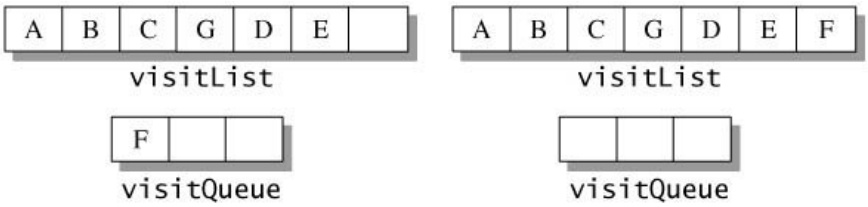
Breadth-First Search Algorithm (continued)

- Pop vertex G from the queue and place it in visitList. G has no adjacent vertices, so pop D from the queue. The neighbors, E and F, enter the queue.



Breadth-First Search Algorithm (continued)

- Continue in this fashion until the queue is empty.



Implementing Breadth-First Search

- Define

```
public enum VertexColor
{
        WHITE, GRAY, BLACK
}
```


Implementing Breadth-First Search (continued)

- The DiGraph class declares three methods that access and update the color attribute of a vertex.

class DiGraph<T> (color methods)	ds.util
void	**colorWhite**() Set the color in each vertex to WHITE.
VertexColor	**getColor**(T v) Returns the color of vertex v. If v is not a graph vertex, throws IllegalArgumentException.
VertexColor	**setColor**(T v, VertexColor c) Sets the color of vertex v and returns the previous color. If v is not a graph vertex, throws IllegalArgumentException.

Implementing Breadth-First Search (continued)

- The method bfs() returns a list of vertices visited during the breadth-first search from a starting vertex.

bfs()

```
// perform the breadth-first traversal
// from sVertex and return the list
// of visited vertices
public static <T> LinkedList<T> bfs(
DiGraph<T> g, T sVertex)
{
   // queue stores adjacent vertices; list
   // stores visited vertices
   LinkedQueue<T> visitQueue = new LinkedQueue<T>();
   LinkedList<T> visitList = new LinkedList<T>();

   // set and iterator retrieve and scan
   // neighbors of a vertex
   Set<T> edgeSet;
   Iterator<T> edgeIter;

   T currVertex = null, neighborVertex = null;
```

bfs() (continued)

```
// check that starting vertex is valid
if (!g.containsVertex(sVertex))
   throw new IllegalArgumentException(
         "bfs(): starting vertex not in the graph");

// color all vertices WHITE
g.colorWhite();

// initialize queue with starting vertex
visitQueue.push(sVertex);

while (!visitQueue.isEmpty())
{
   // remove a vertex from the queue, color
   // it black, and add to the list of
   // visited vertices
   currVertex = visitQueue.pop();
   g.setColor(currVertex,VertexColor.BLACK);
   visitList.add(currVertex);
```


bfs() (continued)

```
   // obtain the set of neighbors for current vertex
   edgeSet = g.getNeighbors(currVertex);
   // sequence through the neighbors and look
   // for vertices that have not been visited
   edgeIter = edgeSet.iterator();
   while (edgeIter.hasNext())
   {
      neighborVertex = edgeIter.next();
```

bfs() (concluded)

```
            if (g.getColor(neighborVertex) ==
                  VertexColor.WHITE)
            {
               // color unvisited vertex GRAY and
               // push it onto queue
               g.setColor(neighborVertex,VertexColor.GRAY);
               visitQueue.push(neighborVertex);
            }
         }
      }

      return visitList;
   }
```

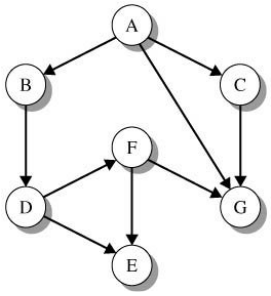

bfs() (concluded)

```
            if (g.getColor(neighborVertex) ==
                  VertexColor.WHITE)
            {
               // color unvisited vertex GRAY and
               // push it onto queue
               g.setColor(neighborVertex,VertexColor.GRAY);
               visitQueue.push(neighborVertex);
            }
         }
      }

      return visitList;
   }
```

Running Time of Breadth-First Search

- The running time for the breadth-first search is O(V + E), where V is the number of vertices and E is the number of edges.

Breadth-First Search Example




```
// create a graph g and declare startVertex and visitList
DiGraph<String> g = DiGraph.readGraph("bfsgraph.dat");
String startVertex;
List<String> visitList;
                ...
// call bfs() with arguments g and startVertx
visitList = DiGraphs.bfs(g, startVertex);
// output the visitList
System.out.println("BFS visitList from " + startVertex +
                   ":    " + visitList);
```

Breadth-First Search Example (concluded)

Output:

```
    Run 1: (startVertex = "A")
      BFS visitList from A:   [A, G, B, C, D, E, F]
    Run 2: (startVertex = "D")
      BFS visitList from D:   [D, E, F, G]
    Run 3: (startVertex = "E")
      BFS visitList from E:   [E]
```


Depth-First Visit

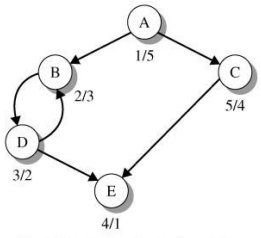
- The depth-first visit algorithm is modeled after the recursive postorder scan of a binary tree. In the tree, a node is visited only after visits are made to all of the nodes in its subtree. In the graph, a node is visited only after visiting all of the nodes in paths that emanate from the node.

Depth-First Visit (continued)

- Backtrack to the previous recursive step and look for another adjacent vertex and launch a scan down its paths. There is no ordering among vertices in an adjacency list, so the paths and hence the order of visits to vertices can vary.

Depth-First Visit (continued)

- Discover A (color GRAY)
- Discover B (color GRAY)
- Discover D (color GRAY)
- Discover E (color GRAY, then BLACK)
- Backtrack D (color BLACK)
- Backtrack B (color BLACK)
- Backtrack A (A remains GRAY)
- Discover C (color BLACK)
- Backtrack A (color BLACK). Visit com



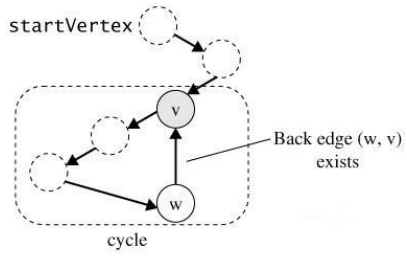
Graph illustrating depth-first visit.

Depth-First Visit (continued)

- The depth-first visit is a recursive algorithm that distinguishes the discovery and finishing time (when a vertex becomes BLACK) of a vertex.
- The depth-first visit returns a list of the vertices found in the reverse order of their finishing times.

Depth-First Visit (continued)

- An edge that connects a vertex to a neighbor that has color GRAY is called a *back edge*.
  - A depth-first visit has a cycle if and only if it has a back edge.



Discovering a cycle assuming v is a GRAY neighbor of w (Edge (w, v) is a back edge).

dfsVisit()

```
// depth-first visit assuming a WHITE starting
// vertex; dfsList contains the visited vertices in
// reverse order of finishing time; when checkForCycle
// is true, throws IllegalPathStateException if it
// detects a cycle
public static <T> void dfsVisit(DiGraph<T> g, T sVertex,
LinkedList<T> dfsList, boolean checkForCycle)
{
   T neighborVertex;
   Set<T> edgeSet;
   // iterator to scan the adjacency set of a vertex
   Iterator<T> edgeIter;
   VertexColor color;

   if (!g.containsVertex(sVertex))
      throw new IllegalArgumentException(
            "dfsVisit(): vertex not in the graph");
```

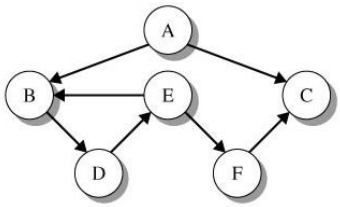

dfsVisit() (continued)

```
   // color vertex GRAY to note its discovery
   g.setColor(sVertex, VertexColor.GRAY);

   edgeSet = g.getNeighbors(sVertex);

   // sequence through the adjacency set and look
   // for vertices that are not yet discovered
   // (colored WHITE); recursively call dfsVisit()
   // for each such vertex; if a vertex in the adjacency
   // list is GRAY, the vertex was discovered during a
   // previous call and there is a cycle that begins and
   // ends at the vertex; if checkForCycle is true,
   // throw an exception
   edgeIter = edgeSet.iterator();
```

dfsVisit() (concluded)

```
   while (edgeIter.hasNext())
   {
      neighborVertex = edgeIter.next();
      color = g.getColor(neighborVertex);
      if (color == VertexColor.WHITE)
         dfsVisit(g, neighborVertex, dfsList,
               checkForCycle);
      else if (color == VertexColor.GRAY && checkForCycle)
         throw new IllegalPathStateException(
               "dfsVisit(): graph has a cycle");
   }

   // finished with vertex sVertex; make it BLACK
   // and add it to the front of dfsList
   g.setColor(sVertex, VertexColor.BLACK);
   dfsList.addFirst(sVertex);
}
```

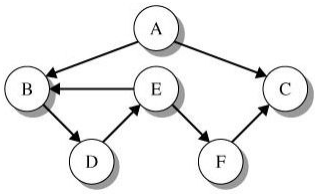

Depth-First Visit Example




```
LinkedList<String> finishOrder = new LinkedList<String>();
g.colorWhite();
DiGraphs.dfsVisit(g, "B", finishOrder, false);
```

```
Output:  finishOrder: [B, D, E, F, C]
```

Depth-First Visit Example (concluded)

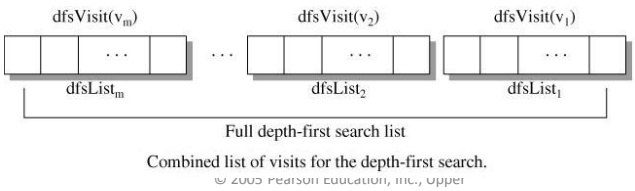



```
finishOrder = new LinkedList<String>();
g.colorWhite();
try
{
    DiGraphs.dfsVisit(g, "E", finishOrder, true);
    System.out.println("finishOrder: " + finishOrder);
}
catch (IllegalPathStateException ipse)
{
    System.out.println(ipse.getMessage());
}
```

```
Output: dfsVisit(): cycle involving vertices D and E
```


Depth-First Search Algorithm

- Depth-first search begins with all WHITE vertices and performs depth-first visits until all vertices of the graph are BLACK.
- The algorithm returns a list of all vertices in the graph in the reverse order of their finishing times.



Combined list of visits for the depth-first search.

dfs()

```
// depth-first search; dfsList contains all
// the graph vertices in the reverse order
// of their finishing times
public static <T> void dfs(DiGraph<T> g,
LinkedList<T> dfsList)
{
   Iterator<T> graphIter;
   T vertex = null;

   // clear dfsList
   dfsList.clear();

   // initialize all vertices to WHITE
   g.colorWhite();
```

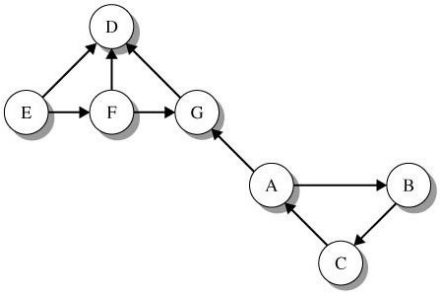

dfs() (concluded)

```
   // call dfsVisit() for each WHITE vertex
   graphIter = g.vertexSet().iterator();
   while (graphIter.hasNext())
   {
      vertex = graphIter.next();
      if (g.getColor(vertex) == VertexColor.WHITE)
         dfsVisit(g, vertex, dfsList, false);
   }
}
```

Running Time for Depth-First Search

- An argument similar to that for the breadth-first search shows that that the running time for dfs() is O(V+E), where V is the number of vertices in the graph and E is the number of edges.

Depth-First Search Example




```
dfsVisit() starting at E followed by dfsVisit() starting at A
dfsList: [A, B, C, E, F, G, D]
```

Program 24.2

```
import java.io.FileNotFoundException;

import ds.util.DiGraph;
import ds.util.DiGraphs;

public class Program24_2
{
   public static void main(String[] args)
   throws FileNotFoundException
   {
      DiGraph<String> g = DiGraph.readGraph("cycle.dat");

      // determine if the graph is acyclic
      if (DiGraphs.acyclic(g))
         System.out.println("Graph is acyclic");
      else
         System.out.println("Graph is not acyclic");
```


Program 24.2 (concluded)

```
      // add edge (E,B) to create a cycle
      System.out.print("   Adding edge (E,B): ");
      g.addEdge("E", "B", 1);

      // retest the graph to see if it is acyclic
      if (DiGraphs.acyclic(g))
         System.out.println("New graph is acyclic");
      else
         System.out.println("New graph is not acyclic");
   }
}
```

```
Run:

Graph is acyclic
   Adding edge (E,B): New graph is not acyclic
```