Data Structures and Algorithms
Graph Search Algorithms

Goodrich & Tamassia Sections 13.3 & 13.4
Sahni, Sections 17.8

- Breadth first and depth first search
- Search algorithms
- Returning path information

Graph Searching

We often need to find all vertices reachable from a given vertex, e.g. to find a path from one node to another, or to prove that no path exists.

```
    3             6
   ^ \           ^
  /   \         /
 /     v       /
1 ---> 4 ---> 5 ---> 9
 \     ^
  \   /
   v /
    2         7 ---> 8
```

Need methods to systematically explore all possible paths.

Exercise: Is there a path from vertices 3 to 9?
Exercise: Is there a path from vertices 2 to 7?
Exercise: How many paths are there from vertex 1 to vertex 6?
Exercise: List the vertices visited by
- a DFS starting from vertex 1
- a BFS starting from vertex 1

Breadth First vs Depth First Search

Two main search methods:

Depth First (DFS): Continue down current path until no more options. Then backup and try alternatives.

Children of current node explored before siblings.

A *backtracking* algorithm.

Breadth First (BFS): Explore paths of length M before paths of length M+1.

A *greedy* algorithm.

Easiest illustrated by considering how they apply to searching *trees*:

Searching Graphs

```
    1
  /   \
 /     \
v       v
2       3
 \     /
  \   /
   v v
    4 ----> 5
```

BFS Order:

DFS Order:

N.B: BFS and DFS find the same vertices, just in different orders.

Implementing Breadth First Search

For BFS we keep the vertices still to be searched in a *queue*.

```
bfs(vertex v)
{
  mark v as visited
  initialise Q to be a queue containing only v
  while ( Q isn't empty )
  {
    delete vertex w from Q
    for each u adjacent to w
      if (u not visited)
        { add u to Q;
          mark u as visited }
  }
}
```


Example:

```
                              bfs(1)
      1                       Queue :
    /   \
   /     \
  v       v
  2       3
   \     /
    \   /
     v v
      4 ----> 5
```


Java BFS of an Adjacency-Matrix Graph

Java 1.4 implementation from Sahni.
Sets `reach[i]` to `label` for all vertices reachable from vertex `v`.

```
public void bfs(int v, int [] reach, int label)
  {
    ArrayQueue q = new ArrayQueue(10);
    reach[v] = label;
    q.put(new Integer(v));
    while (!q.isEmpty())
    { // remove a labeled vertex from the queue
      int w = ((Integer) q.remove()).intValue();
      // mark unreached vertices adjacent from w
      for (int u = 1; u <= n; u++) {
        if (a[w][u] && reach[u] == 0)
        {// u is an unreached vertex
          q.put(new Integer(u));
          reach[u] = label;
        }
      }
    }
  }
```


Java BFS of an Adjacency-List Graph

```
public void bfs(int v, int [] reach, int label)
  {
    ArrayQueue q = new ArrayQueue(10);
    reach[v] = label;
    q.put(new Integer(v));
    while (!q.isEmpty())
    { // remove a labeled vertex from the queue
      int w = ((Integer) q.remove()).intValue();
      // mark unreached vertices adjacent from w
      for (ChainNode p = aList[w].firstNode;
           p != null; p = p.next)
      {
        int u = ((EdgeNode) p.element).vertex;
        if (reach[u] == 0)
        {// u is an unreached vertex
          q.put(new Integer(u));
          reach[u] = label;
        }
      }
    }
  }
```

A Generic BFS

Note that the code on the previous slide explicitly uses the list-implementation when traversing the adjacency list: `p = p.next`

Such implementation dependencies are not desirably, since any change in the representation requires a change of the code.

Make the `bfs` method implementation-independent by writing it as a member of the `Graph` class, and without reference to the representation.

Use an `iterator` to visit each adjacent vertex.

A Generic BFS

```
public void bfs(int v, int [] reach, int label)
{
   ArrayQueue q = new ArrayQueue(10);
   reach[v] = label;
   q.put(new Integer(v));
   while (!q.isEmpty())
   {  // remove a labeled vertex from the queue
      int w = ((Integer) q.remove()).intValue();
      // mark all unreached vertices adjacent from w
      Iterator it = aList[w].iterator();
      while (it.hasNext())
      {  // visit an adjacent vertex of w
         EdgeNode e = (EdgeNode) it.next();
         int u = e.vertex;
         if (reach[u] == 0)
         {  // u is an unreached vertex
            q.put(new Integer(u));
            reach[u] = label; // mark reached
         }
      }
   }
}
```


Costs and Benefits of Generic Code

Advantages of Generic Code:

- Reduces coding effort: write a single `bfs` method, rather than many, e.g. one for adjacency-list, one for adjacency-matrix, etc.
- If efficiency is important you can always override with a implementation-specific method.

Disadvantages of Generic Code:

- May reduce time or space performance, e.g. 100-vertex graph `Graph.bfs` 29ms, where `AdjacencyDigraph.bfs` took 0.9ms

Slogan: Try to write generic code, unless there's a very good reason.

Depth First Search

For DFS we hold the vertices to be searched in a *stack*, and can produce an elegant solution using Java's recursion stack.

```
dfs(Vertex v)
{
  mark v as visited
  for each w adjacent to v
    if (w not visited)
      dfs(w);
}
```

Example:

```
                            dfs(1)
       1                    dfs(2) dfs(3)
     /   \
    /     \
   v       v
   2       3
    \     /
     \   /
      v v
       4 ----> 5
```

Java Generic DFS

Assumes `reach` and `label` are data members of the Graph class. Sets `reach[i]` to `label` for all vertices reachable from vertex `v`.

```
public void dfs(int v, int [] reach, int label)
{
  Graph.reach = reach;
  Graph.label = label;
  rDfs(v);
}

/** recursive dfs method */
private void rDfs(int v)
{
  reach[v] = label;
  Iterator iv = iterator(v);
  while (iv.hasNext())
  {// visit an adjacent vertex of v
    int u = ((EdgeNode) iv.next()).vertex;
    if (reach[u] == 0)
      // u is an unreached vertex
      rDfs(u);
  }
}
```


Summary

- Many applications require you to find all nodes reachable from a node.
- Standard systematic methods are BFS and DFS.
- BFS and DFS are very similar but the former uses a queue, and the latter uses a stack.
- Generic programming reduces programming effort.