```
    int [] new_nodes = connected_nodes(path,
                                       num_path);
```

If there are still connected nodes to search, add the next possible "node to visit" to the top of the stack (variable $path$ in the program) and recursively call the method $findPathHelper$ again:

```
    if (new_nodes != null) {
      for (int j=0; j<new_nodes.length; j++) {
        path[num_path] = new_nodes[j];
        int [] test = findPathHelper(new_path,
                                     num_path + 1,
                                     goal_node);
        if (test != null) {
          if (test[test.length-1] == goal_node) {
              return test;
          }
        }
      }
    }
```

If we have not found the goal node, return null, instead of an array of node indices:

```
    return null;
  }
```

Derived class $BreadthFirstSearch$ also must define abstract method $findPath$. This method is very similar to the breadth first search method used for finding a path in a maze: a queue is used to store possible moves. For a maze, we used a queue class that stored instances of the class Dimension, so for this problem, the queue only needs to store integer node indices. The return value of $findPath$ is an array of node indices that make up the path from the starting node to the goal.

```
  public int [] findPath(int start_node,
                         int goal_node) {
```

We start by setting up a flag array $alreadyVisited$ to prevent visiting the same node twice, and allocating a predecessors array that we will use to find the shortest path once the goal is reached:

```
    // data structures for depth first search:
```

```
boolean [] alreadyVisitedFlag =
                        new boolean[numNodes];
int [] predecessor = new int[numNodes];
```

The class $IntQueue$ is a private class defined in the file BreadthFirstSearch.java; it implements a standard queue:

```
IntQueue queue = new IntQueue(numNodes + 2);
```

Before the main loop, we need to initialize the already visited and predecessor arrays, set the visited flag for the starting node to true, and add the starting node index to the back of the queue:

```
for (int i=0; i<numNodes; i++) {
  alreadyVisitedFlag[i] = false;
  predecessor[i] = -1;
}
alreadyVisitedFlag[start_node] = true;
queue.addToBackOfQueue(start_node);
```

The main loop runs until we find the goal node or the search queue is empty:

```
outer: while (queue.isEmpty() == false) {
```

We will read (without removing) the node index at the front of the queue and calculate the nodes that are connected to the current node (but not already on the visited list) using the $connected\_nodes$ method (the interested reader can see the implementation in the source code for this class):

```
  int head = queue.peekAtFrontOfQueue();
  int [] connected = connected_nodes(head);
  if (connected != null) {
```

If each node connected by a link to the current node has not already been visited, set the predecessor array and add the new node index to the back of the search queue; we stop if the goal is found:

```
  for (int i=0; i<connected.length; i++) {
     if (alreadyVisitedFlag[connected[i]] == false) {
       predecessor[connected[i]] = head;
```

```
            queue.addToBackOfQueue(connected[i]);
            if (connected[i] == goal_node) break outer;
          }
        }
        alreadyVisitedFlag[head] = true;
        queue.removeFromQueue(); // ignore return value
      }
    }
```

Now that the goal node has been found, we can build a new array of returned node indices for the calculated path using the predecessor array:

```
    int [] ret = new int[numNodes + 1];
    int count = 0;
    ret[count++] = goal_node;
    for (int i=0; i<numNodes; i++) {
      ret[count] = predecessor[ret[count - 1]];
      count++;
      if (ret[count - 1] == start_node)  break;
    }
    int [] ret2 = new int[count];
    for (int i=0; i<count; i++) {
      ret2[i] = ret[count - 1 - i];
    }
    return ret2;
  }
```

In order to run both the depth first and breadth first graph search examples, change directory to src-search-maze and type the following commands:

```
javac *.java
java GraphDepthFirstSearch
java GraphBeadthFirstSearch
```

Figure 2.6 shows the results of finding a route from node 1 to node 9 in the small test graph. Like the depth first results seen in the maze search, this path is not optimal.

Figure 2.7 shows an optimal path found using a breadth first search. As we saw in the maze search example, we find optimal solutions using breadth first search at the cost of extra memory required for the breadth first search.