The abstract base class defines an abstract method $findPath$ that must be overridden. We will start with the derived class $DepthFirstSearch$, looking at its implementation of findPath. The $findPath$ method returns an array of node indices indicating the calculated path:

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

The class variable path is an array that is used for temporary storage; we set the first element to the starting node index, and call the utility method $findPathHelper$:

```
    path[0] = start_node; // the starting node
    return findPathHelper(path, 1, goal_node);
  }
```

The method findPathHelper is the interesting method in this class that actually performs the depth first search; we will look at it in some detail:

The path array is used as a stack to keep track of which nodes are being visited during the search. The argument num_path is the number of locations in the path, which is also the search depth:

```
public int [] findPathHelper(int [] path,
                             int num_path,
                             int goal_node) {
```

First, re-check to see if we have reached the goal node; if we have, make a new array of the current size and copy the path into it. This new array is returned as the value of the method:

```
    if (goal_node == path[num_path - 1]) {
      int [] ret = new int[num_path];
      for (int i=0; i<num_path; i++) {
         ret[i] = path[i];
      }
      return ret;  // we are done!
    }
```

We have not found the goal node, so call the method $connected\_nodes$ to find all nodes connected to the current node that are not already on the search path (see the source code for the implementation of $connected\_nodes$):

```
    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:
```