effect of a breadth first search is that it "fans out" uniformly from the starting node until the goal node is found.

The class constructor for $BreadthFirstSearch$ calls the super class constructor to initialize the maze, and then uses the auxiliary method $doSearchOn2Dgrid$ for performing a breadth first search for the goal. We will look at the class $BreadthFirstSearch$ in some detail. Breadth first search uses a queue instead of a stack (depth first search) to store possible moves. The utility class $DimensionQueue$ implements a standard queue data structure that handles instances of the class $Dimension$.

The method $doSearchOn2Dgrid$ is not recursive, it uses a loop to add new search positions to the end of an instance of class $DimensionQueue$ and to remove and test new locations from the front of the queue. The two-dimensional array $allReadyVisited$ keeps us from searching the same location twice. To calculate the shortest path after the goal is found, we use the predecessor array:

```
private void doSearchOn2DGrid() {
  int width = maze.getWidth();
  int height = maze.getHeight();
  boolean alReadyVisitedFlag[][] =
          new boolean[width][height];
  Dimension predecessor[][] =
        new Dimension[width][height];
  DimensionQueue queue =
                new DimensionQueue();
  for (int i=0; i<width; i++) {
    for (int j=0; j<height; j++) {
      alReadyVisitedFlag[i][j] = false;
      predecessor[i][j] = null;
    }
  }
```

We start the search by setting the already visited flag for the starting location to true value and adding the starting location to the back of the queue:

```
  alReadyVisitedFlag[startLoc.width][startLoc.height]
     = true;
  queue.addToBackOfQueue(startLoc);
  boolean success = false;
```

This outer loop runs until either the queue is empty or the goal is found:

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

We peek at the $Dimension$ object at the front of the queue (but do not remove it) and get the adjacent locations to the current position in the maze:

```
Dimension head = queue.peekAtFrontOfQueue();
Dimension [] connected =
                getPossibleMoves(head);
```

We loop over each possible move; if the possible move is valid (i.e., not null) and if we have not already visited the possible move location, then we add the possible move to the back of the queue and set the predecessor array for the new location to the last square visited (head is the value from the front of the queue). If we find the goal, break out of the loop:

```
for (int i=0; i<4; i++) {
  if (connected[i] == null) break;
  int w = connected[i].width;
  int h = connected[i].height;
  if (alReadyVisitedFlag[w][h] == false) {
    alReadyVisitedFlag[w][h] = true;
    predecessor[w][h] = head;
    queue.addToBackOfQueue(connected[i]);
    if (equals(connected[i], goalLoc)) {
      success = true;
      break outer; // we are done
    }
  }
}
```

We have processed the location at the front of the queue (in the variable head), so remove it:

```
  queue.removeFromFrontOfQueue();
}
```

Now that we are out of the main loop, we need to use the predecessor array to get the shortest path. Note that we fill in the $searchPath$ array in reverse order, starting with the goal location:

```
maxDepth = 0;
if (success) {
  searchPath[maxDepth++] = goalLoc;
```

```
      for (int i=0; i<100; i++) {
        searchPath[maxDepth] =
           predecessor[searchPath[maxDepth - 1].
             width][searchPath[maxDepth - 1].
             height];
        maxDepth++;
        if (equals(searchPath[maxDepth - 1],
                   startLoc))
          break;  // back to starting node
      }
    }
  }
```

Figure 2.4 shows a good path solution between starting and goal nodes. Starting from the initial position, the breadth first search engine adds all possible moves to the back of a queue data structure. For each possible move added to this queue in one search cycle, all possible moves are added to the queue for each new move recorded. Visually, think of possible moves added to the queue as "fanning out" like a wave from the starting location. The breadth first search engine stops when this "wave" reaches the goal location. In general, I prefer breadth first search techniques to depth first search techniques when memory storage for the queue used in the search process is not an issue. In general, the memory requirements for performing depth first search is much less than breadth first search.

To run the two example programs from this section, change directory to src/search/-maze and type:

```
javac *.java
java MazeDepthFirstSearch
java MazeBreadthFirstSearch
```

Note that the classes $MazeDepthFirstSearch$ and $MazeBreadthFirstSearch$ are simple Java JFC applications that produced Figures 2.3 and 2.4. The interested reader can read through the source code for the GUI test programs, but we will only cover the core AI code in this book. If you are interested in the GUI test programs and you are not familiar with the Java JFC (or Swing) classes, there are several good tutorials on JFC programming at java.sun.com.

2.3 Finding Paths in Graphs

In the last section, we used both depth first and breadth first search techniques to find a path between a starting location and a goal location in a maze. Another common