Breadth-First Search and Depth-First Search

Traversing two-dimensional grids

Dean Hendrix

Tools of the trade

- A goal of this course is to provide you with important tools of the trade, teach you how to use them, and help you learn when and how to apply them.
- Two very important tools are **breadth-first search** and **depth-first search**.
- In general, we can use BFS or DFS to systematically examine every possible location in a given search space.

  Sedgewick: "Breadth-first search amounts to an army of searchers fanning out to cover the territory; depth-first search corresponds to a single searcher probing unknown territory as deeply as possible, retreating only when hitting dead ends."

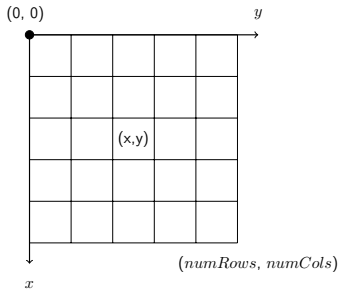
Context

What are we searching through?

Context

What are we searching through?

- BFS and DFS are typically discussed in terms of *graphs*, but the actual search space could be many different things.
- For this note set we will represent the search space with a two-dimensional array.
- We can think of the search space as a two-dimensional grid with a coordinate system that conforms to how Java arrays are indexed.



Context

What are we searching for?

Context

What are we searching for?

- The target of the search can vary widely, from some particular element that might be at a given (x,y) position in the grid to a sequence of moves through the grid that produces a certain outcome.
  - A treasure chest in a game
  - A path from entrance to exit in a maze
  - A shortest sequence of "friend" connections in a social network from one person to another.
- For this note set we will just use BFS and DFS as systematic ways to explore every position in the 2d array.
- So, we're treating BFS and DFS as *traversal* methods rather than search strategies.

Implementation

We will look at the following implementations, all in the class `GridWalker`.

- Breadth-first search: iterative, using a queue
- Depth-first search: iterative, using a stack
- Depth-first search with backtracking: recursive
- Depth-first search with backtracking: iterative
- Breadth-first with memory: iterative

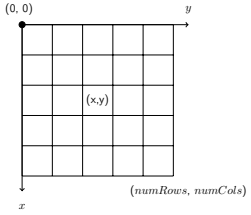
GridWalker Fields

```java
// 2d area to traverse
private int[][] grid;

// visited positions in the grid
private boolean[][] visited;

// dimensions of the grid
private int numRows;
private int numCols;

// number of neighbors, degrees of motion
private final int MAX_NEIGHBORS = 8;

// order in which positions are visited.
// used to enumerate positions in the grid.
private int order;
```

GridWalker Fields

```
// 2d area to traverse
private int[][] grid;
```

- This is the two-dimensional array that represents the search space.



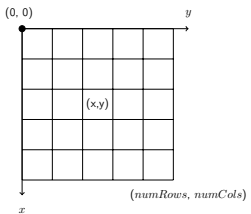
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

- The data type is `int` so that it will be easy to record the order in which a given search method explores each cell.

GridWalker Fields

```
// visited positions in the grid
private boolean[][] visited;
```

- This is a two-dimensional array that keeps track of which cells have been explored.



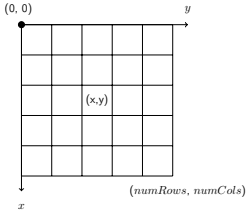
F	F	F	F	F
F	F	F	F	F
F	F	F	F	F
F	F	F	F	F
F	F	F	F	F

- False (F) means the cell has not been explored and true (T) means that the cell has been explored.

GridWalker Fields

```
// order in which positions are visited
private int order;
```

- This is a counter that keeps track of the order in which each cell is explored.



3	4	5	0	0
0	2	0	0	0
0	0	1	0	0
0	0	0	0	0
0	0	0	0	0

T	T	T	F	F
F	T	F	F	F
F	F	T	F	F
F	F	F	F	F
F	F	F	F	F

- Each time a new cell is explored, `order` is incremented and stored the corresponding `grid` position.

Output

Each search implementation marks `grid[x][y]` with an integer to record the order in which each grid position was examined.

- Three sample output grids:

26	27	28	29	30
10	11	12	15	16
13	2	3	4	17
14	5	1	6	18
19	7	8	9	20
21	22	23	24	25

20	19	18	17	16
21	11	12	13	15
10	30	29	28	14
9	27	1	26	23
8	25	24	2	22
7	6	5	4	3

4	5	6	7	8
3	11	10	9	17
12	2	15	16	18
13	14	1	19	20
24	23	22	21	29
25	26	27	28	30

Modeling positions in the grid

The positions in the grid will be modeled by the inner class `Position`.

```
// model an (x,y) position in the grid
class Position {
    int x;
    int y;

    public Position(int x, int y) {  }

    @Override
    public String toString() {  }

    public Position[] neighbors() {  }
}
```

Modeling positions in the grid

```java
// model an (x,y) position in the grid
class Position {
   int x;
   int y;

   /** Constructs a Position with coordinates (x,y). */
   public Position(int x, int y) {
      this.x = x;
      this.y = y;
   }

   /** Returns a string representation of this Position. */
   @Override
   public String toString() {
      return "(" + x + ", " + y + ")";
   }

    public Position[] neighbors() {  }
}
```

Methods on Position objects

```
/**
 * Is this position valid in the search area?
 */
private boolean isValid(Position p) {
    return (p.x >= 0) && (p.x < numRows) &&
           (p.y >= 0) && (p.y < numCols);
}
```

Methods on `Position` objects

```
/**
 * Has this valid position been visited?
 */
private boolean isVisited(Position p) {
    return visited[p.x][p.y];
}
```

Methods on `Position` objects

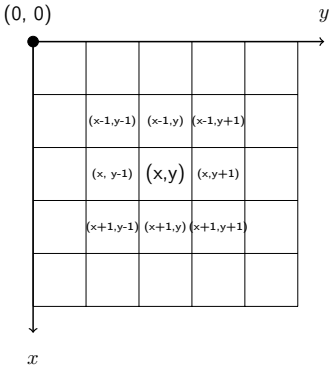
```
/**
 * Mark this valid position as having been visited.
 */
private void visit(Position p) {
    visited[p.x][p.y] = true;
}
```

Methods on `Position` objects

```
/**
 * Process this valid position.
 */
private void process(Position p) {
    grid[p.x][p.y] = order++;
}
```

Modeling positions in the grid

The eight neighbors of a position in the grid:

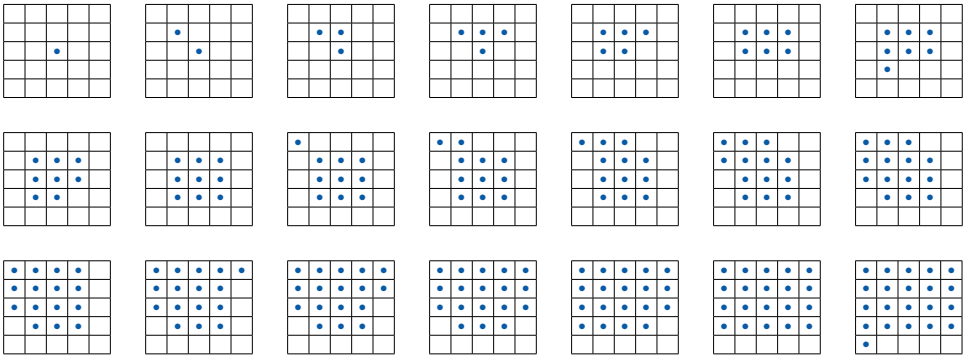


Methods on Position objects

```
/** Returns all the neighbors of this Position. */
public Position[] neighbors() {
   Position[] nbrs = new Position[MAX_NEIGHBORS];
   int count = 0;
   Position p;
   // generate all eight neighbor positions
   // add to return value if valid
   for (int i = -1; i <= 1; i++) {
      for (int j = -1; j <= 1; j++) {
         if (!((i == 0) && (j == 0))) {
            p = new Position(x + i, y + j);
            if (isValid(p)) {
               nbrs[count++] = p;
            }
         }
      }
   }
   return Arrays.copyOf(nbrs, count);
}
```

Breadth-first search

- Breadth-first search: Explore the immediate neighborhood before going deeper.



Breadth-first search

```
private void bfs(Position start) {
    Deque<Position> queue = new ArrayDeque<>();
    visit(start);
    process(start);
    queue.addLast(start);
    while (!queue.isEmpty()) {
        Position position = queue.removeFirst();
        for (Position neighbor : position.neighbors()) {
            if (!isVisited(neighbor)) {
                visit(neighbor);
                process(neighbor);
                queue.addLast(neighbor);
            }
        }
    }
}
```

Breadth-first search

- Breadth-first search starting at (3,2):

26	27	28	29	30
10	11	12	15	16
13	2	3	4	17
14	5	1	6	18
19	7	8	9	20
21	22	23	24	25

Breadth-first search

- Breadth-first search starting at (0,0):

1	2	5	10	17
3	4	6	11	18
7	8	9	12	19
13	14	15	16	20
21	22	23	24	25
26	27	28	29	30

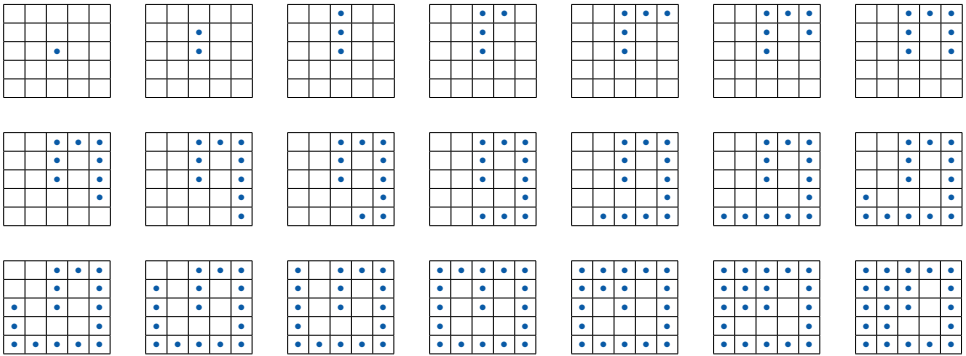
Breadth-first search

- Breadth-first search starting at (5,4):

26	27	28	29	30
17	18	19	22	23
20	10	11	12	15
21	13	5	6	7
24	14	8	2	3
25	16	9	4	1

Depth-first search

- Depth-first search: Explore as deeply as possible before backing up.



Iterative Depth-first search

```
private void dfsIterativeA(Position start) {
   Deque<Position> stack = new ArrayDeque<>();
   visit(start);
   stack.addFirst(start);
   while (!stack.isEmpty()) {
      Position position = stack.removeFirst();
      process(position);
      for (Position neighbor : position.neighbors()) {
         if (!isVisited(neighbor)) {
            visit(neighbor);
            stack.addFirst(neighbor);
         }
      }
   }
}
```

Iterative Depth-first search

- Depth-first search starting at (3,2):

20	19	18	17	16
21	11	12	13	15
10	30	29	28	14
9	27	1	26	23
8	25	24	2	22
7	6	5	4	3

Iterative Depth-first search

- Depth-first search starting at (0,0):

1	30	28	20	19
29	2	27	24	18
26	25	3	23	17
12	22	21	4	16
11	13	15	14	5
10	9	8	7	6

Iterative Depth-first search

- Depth-first search starting at (5,4):

18	19	20	14	13
17	16	15	21	12
25	24	23	22	11
26	7	8	9	10
6	27	28	30	29
5	4	3	2	1

Implementation

- See class `GridWalker` for implementations of the following algorithms that simply traverse an open grid.
- See class `MazeSearcher` for implementations of the following algorithms that find a given location in a maze.
  - Breadth-first search: iterative, using a queue
  - Depth-first search: iterative, using a stack
  - Depth-first search with backtracking: recursive
  - Depth-first search with backtracking: iterative
  - Breadth-first with memory: iterative