Breadth-First Search and Depth-First Search

COMP 2210 — Dr. Hendrix

March 13, 2015

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**.

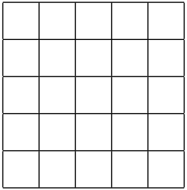
In general, we can use BFS or DFS to systematically examine every possible location in a given search space.

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?

- BFS and DFS are typically discussed in terms of *graphs*, but the actual search space could be many different things.
- For this note set the search space won't have any particular meaning, and we will represent the search space with a two-dimensional array.



Context

What are we searching for?

- The target of the search will, of course, vary.
- If the search space is a maze we might be searching for a path from the entrance to the exit.
- If the search space is a social network we might be searching for the shortest “friend” connection between a group of students and a group of people on a security watchlist.
- For this note set we won’t be searching for anything. Instead, we will just use BFS and DFS to examine every position in the 2d array.
- So, for now at least, we’re only using BFS and DFS as *traversal* methods.

Implementation

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

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

BfsDfs Fields

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

// visited positions in the search area
private boolean[][] visited;

// dimensions of the search area
private int width;
private int height;

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

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

Output

All three search implementations mark `grid[x][y]` with an integer to record the order in which each grid position was examined by the search.

initial state

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
0	0	0	0	0

final state

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

Modeling positions in the grid

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

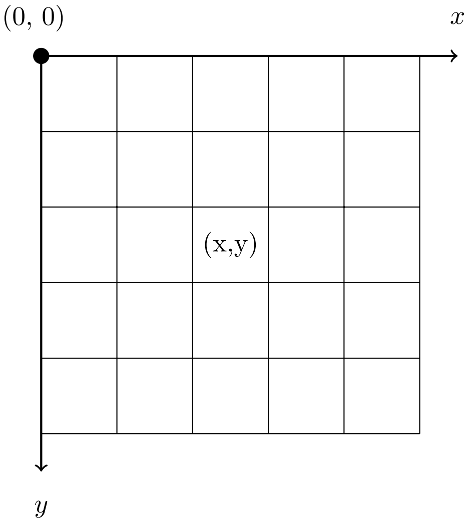
```
// model an (x,y) position in the grid
class Position {
    int x;
    int y;

    public Position(int _x, int _y) {  }

    @Override
    public String toString() {  }

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

Modeling positions in the grid

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

    public Position(int _x, int _y) {
        x = _x;
        y = _y;
    }

    @Override
    public String toString() {
        return "(" + x + ", " + y + ")";
    }

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

Modeling positions in the grid

A position needs to find its neighbors.



Modeling positions in the grid

```
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);
}
```

Methods on Position objects

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

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

```java
/**
 * 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++;
  }
```

Breadth-first search

```
public void breadth_first(int x, int y) {
   markAllUnvisited();
   Position start = new Position(x, y);
   if (isValid(start)) {
      order = 1;
      bfs(start);
   }
}
```

Breadth-first search

```
private void bfs(Position start) {
   Deque<Position> q = new LinkedList<Position>();
   visit(start);
   process(start);
   q.add(start);
   while (!q.isEmpty()) {
      Position p = q.remove();
      for (Position n : p.neighbors()) {
         if (!isVisited(n)) {
            visit(n);
            process(n);
            q.add(n);
         }
      }
   }
}
```

Breadth-first search

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

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

Breadth-first search

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

Iterative Depth-first search

```
public void depth_first_stack(int x, int y) {
   markAllUnvisited();
   Position start = new Position(x, y);
   if (isValid(start)) {
      order = 1;
      dfs_stack(start);
   }
}
```

Iterative Depth-first search

```
private void dfs_stack(Position start) {
   Deque<Position> s = new LinkedList<Position>();
   s.addFirst(start);
   visit(start);
   while (!s.isEmpty()) {
      Position p = s.removeFirst();
      process(p);
      for (Position n : p.neighbors()) {
         if (!isVisited(n)) {
            visit(n);
            s.addFirst(n);
         }
      }
   }
}
```

Iterative Depth-first search

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

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

Iterative Depth-first search

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

Recursive Depth-first search

```
public void depth_first_recursive(int x, int y) {
    markAllUnvisited();
    Position start = new Position(x, y);
    if (isValid(start)) {
        order = 1;
        dfs_recursive(start);
    }
}
```

Recursive Depth-first search

```
private void dfs_recursive(Position p) {
    visit(p);
    process(p);
    for (Position n : p.neighbors()) {
        if (!isVisited(n)) {
            dfs_recursive(n);
        }
    }
}
```

Recursive Depth-first search

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

Recursive Depth-first search

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

Recursive Depth-first search

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