

Traveler Problem

How did I do it?

Ray Liu

November 2015

Problem Statement

For example, for a |4x4 one, here is a solution

```
* X *-*  
| | | |  
*-* * * |  
| | | |  
*-* * * |  
| | | |  
*-*-* * |
```

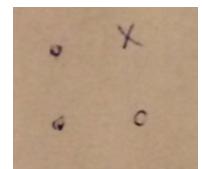
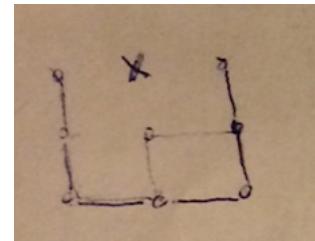
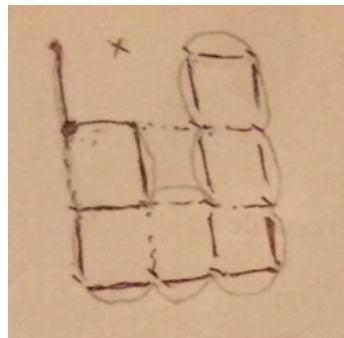
Figure out a travel plan to go through each and every city exactly once with only horizontal and vertical moves but no diagonal ones.
The X should be excluded.

```
* X * * *  
* * * * *  
* * * * *  
* * * * *  
* * * * *
```

```
* X *-*-*  
| | | | |  
*-*-*-*-*  
| | | | |  
*-*-*-*-*  
| | | | |  
*-*-*-*-*  
| | | | |  
*-*-*-*-*
```

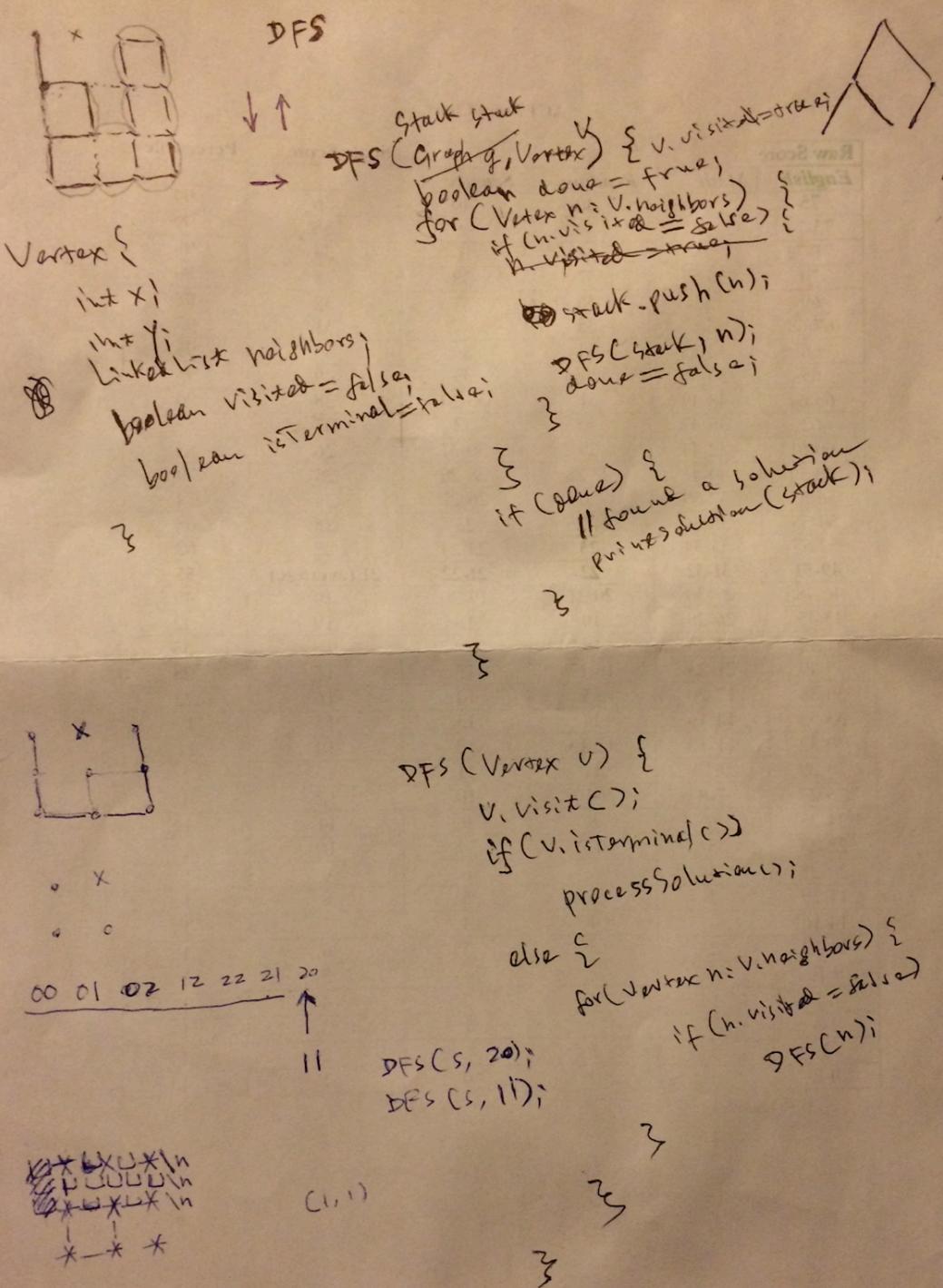
Use good examples to figure out a solution

- 5x5 grid is too complex, and it does not seem to have solutions. Not a good example for figuring out solutions.
- Use 4x4 grid instead, which has obvious solutions.
- Even 3x3.
- Even 2x2.



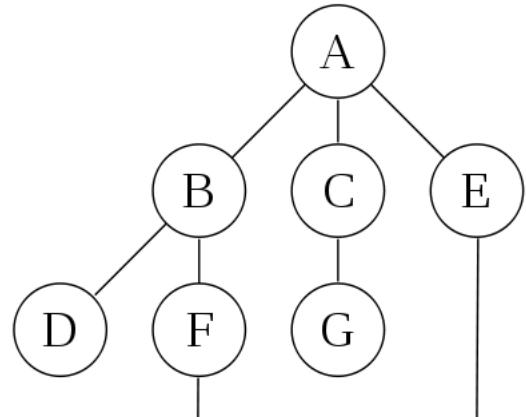
Use paper and pencil to figure out a solution.

- Do NOT use a computer until you have found the solution.
- Always use specific examples (good examples) to figure out the algorithm.
Abstraction does not work.
- After finding a solution, try it with other examples. Think abstraction.



Reference known algorithms

- DFS seems relevant.
- https://en.wikipedia.org/wiki/Depth-first_search



Pseudocode [edit]

Input: A graph G and a vertex v of G

Output: All vertices reachable from v labeled as discovered

A recursive implementation of DFS:^[5]

```
1  procedure DFS( $G, v$ ):
2      label  $v$  as discovered
3      for all edges from  $v$  to  $w$  in  $G.\text{adjacentEdges}(v)$  do
4          if vertex  $w$  is not labeled as discovered then
5              recursively call DFS( $G, w$ )
```

My DFS method

- Recursive
- Use a stack so I can record my path

```
Traveler_Ray traveler = new Traveler_Ray();
Grid grid = new Grid(n);
traveler.grid = grid;
Stack<Vertex> stack = new Stack<Vertex>();
traveler.DFS(stack, grid.getStartingVertex());
```

```
private void DFS(Stack<Vertex> stack, Vertex v) {
    stack.push(v);
    List<Vertex> unvisitedNeighbors = v.getUnvisitedNeighbors(stack);
    if (unvisitedNeighbors.isEmpty())
        grid.processSolution(stack);
    else {
        for (Vertex n : unvisitedNeighbors){
            DFS(stack, n);
        }
    }
    stack.pop();      // shouldn't have to do this
}
```

Vertex class

```
public class Vertex {  
    private int x;  
    private int y;  
    private LinkedList<Vertex> neighbors;  
  
    public Vertex() {  
        this.neighbors = new LinkedList<Vertex>();  
    }  
  
    public String toString() {  
        return "" + getX() + getY() + " ";  
    }  
  
    public void setNeighbors(LinkedList<Vertex> neighbors){  
        this.neighbors = neighbors;  
    }  
  
    public LinkedList<Vertex> getUnvisitedNeighbors(Stack<Vertex> s){  
        LinkedList<Vertex> list = new LinkedList<Vertex>();  
        for(Vertex v : neighbors) {  
            if(s.search(v) < 0) {  
                list.add(v);  
            }  
        }  
        return list;  
    }  
}
```

Vertex {
 int x;
 int y;
 LinkedList neighbors;
 boolean visited = false;
 boolean isTerminal = false;
}
}

Grid class

- 2-D array of Vertex
- Hole Vertex
- Starting Vertex
- solutions
- buildGraph()
- processSolution(stack)
- getSolutions()
- The main() method for testing the class

```
public class Grid {  
    private Vertex[][] vertices;  
    private int size;  
    private int hole_x = 1;  
    private int hole_y = 0;  
    private int starting_x = 0;  
    private int starting_y = 0;  
    private SolutionSet solutions;  
    private int numAttempts = 0;  
  
    public Grid(int n) {  
        size = n;  
        vertices = new Vertex[n][n];  
        for(int i=0; i<n; i++) {  
            for(int j=0; j<n; j++) {  
                Vertex v = new Vertex();  
                v.setX(i);  
                v.setY(j);  
                vertices[i][j] = v;  
            }  
        }  
        buildGraph();  
        solutions = new SolutionSet(this);  
    }  
}
```

```
Grid 4 created: starting: 00 , hole: 10  
+ X + +
```

```
+ + + +  
+ + + +  
+ + + +
```

```
public static void main(String[] args) {  
    Grid grid = new Grid(4);  
    System.out.println("Grid " + grid.size() + " created: starting: "  
                      + grid.getStartingVertex() + ", hole: " + grid.getHole());  
    System.out.println(grid);  
}
```

Grid class – continued

- buildGraph()
method

```
* X *-*-*  
| | | | |  
*-*-*-*-*  
| | | | |  
*-*-*-*-*  
| | | | |  
*-*-*-*-*
```

```
/*  
 * returns a string visualizing the empty grid  
 */  
public String toString() {  
    StringBuilder buf = new StringBuilder();  
  
    for(int row=0; row<size; row++) {  
        // first line: "+ X + +\n"  
        for(int col=0; col<size; col++) {  
            // + or X  
            if(row == hole_y && col == hole_x)  
                buf.append('X');  
            else  
                buf.append('+');  
            // ' ' or '\n'  
            if(col<size-1)  
                buf.append(' ');  
            else // last one  
                buf.append('\n');  
        }  
        // 2nd line: "      \n"  
        for(int col=0; col<size; col++) {  
            buf.append(' ');  
            // ' ' or '\n'  
            if(col<size-1)  
                buf.append(' ');  
            else // last one  
                buf.append('\n');  
        }  
    }  
    return buf.toString();  
}
```

Drawing a solution

- Solution.toStringShow(grid)
- Start with grid.toString()
- Replace certain ‘ ‘ with ‘-’ or ‘|’
- Math!

```
public String toString() {  
    StringBuilder buf = new StringBuilder();  
  
    for(int row=0; row<size; row++) {  
        // first line: "+ X +\n"  
        for(int col=0; col<size; col++) {  
            // + or X  
            if(row == hole_y && col == hole_x)  
                buf.append('X');  
            else  
                buf.append('+');  
            // ' ' or '\n'  
            if(col<size-1)  
                buf.append(' ');  
            else // last one  
                buf.append('\n');  
        }  
        // 2nd line: "      \n"  
        for(int col=0; col<size; col++) {  
            buf.append(' ');  
            // ' ' or '\n'  
            if(col<size-1)  
                buf.append(' ');  
            else // last one  
                buf.append('\n');  
        }  
    }  
  
    return buf.toString();  
}
```

```
// draw the solution represented by stack given the grid  
public String toStringShow(Grid grid) {  
    StringBuilder g = new StringBuilder(grid.toString());  
    int size = grid.size();  
  
    for(int i=1; i<stack.size(); i++){  
        Vertex v1 = stack.get(i-1);  
        Vertex v2 = stack.get(i);  
        int x1 = v1.getX(), y1 = v1.getY();  
        int x2 = v2.getX(), y2 = v2.getY();  
        // either x1==x2 or y1==y2  
        if(x1 == x2) { // x1==x2: 1 step down or up from (x1,y1)  
            // locate (x1,y1)  
            int index = 4 * size * y1 + 2 * x1;  
            // locate where to set '|'  
            if(y1 < y2) index += 2*size;  
            else index -= 2*size;  
            g.setCharAt(index, '|');  
        }  
        else { // y1==y2: 1 step right or left from (x1,y1)  
            // locate (x1,y1)  
            int index = 4 * size * y1 + 2 * x1;  
            // locate where to set '-'  
            if(x1 < x2) index++;  
            else index--;  
            g.setCharAt(index, '-');  
        }  
    }  
  
    return g.toString();  
}
```



Issues I ran into

- Traveler:DFS(): why stack.pop()?
- Grid:Grid(): v = new Vertex() after vertices = new Vertex[n][n].

```
public class Grid {  
    private Vertex[][] vertices;  
    private int size;  
    private int hole_x = 1;  
    private int hole_y = 0;  
    private int starting_x = 0;  
    private int starting_y = 0;  
    private SolutionSet solutions;  
    private int numAttempts = 0;  
  
    public Grid(int n)  {  
        size = n;  
        vertices = new Vertex[n][n];  
        for(int i=0; i<n; i++)  {  
            for(int j=0; j<n; j++)  {  
                Vertex v = new Vertex();  
                v.setX(i);  
                v.setY(j);  
                vertices[i][j] = v;  
            }  
        }  
        buildGraph();  
        solutions = new SolutionSet(this);  
    }
```

```
public class Traveler_Ray {  
  
    private Grid grid;  
  
    private void DFS(Stack<Vertex> stack, Vertex v) {  
        stack.push(v);  
        List<Vertex> unvisitedNeighbors = v.getUnvisitedNeighbors(stack);  
        if (unvisitedNeighbors.isEmpty())  
            grid.processSolution(stack);  
        else  {  
            for (Vertex n : unvisitedNeighbors){  
                DFS(stack, n);  
            }  
        }  
        stack.pop(); // shouldn't have to do this  
    }  
}
```