Bjarne Poulsen

Maze

Theory

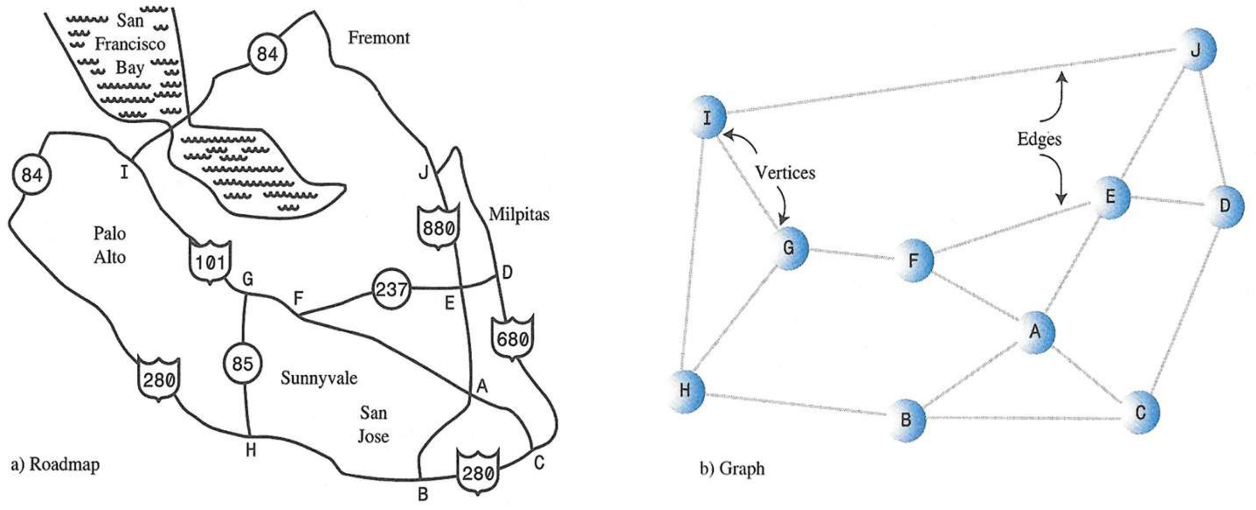
An introduction to the concepts of graph theory. This is a subject which is necessary to explain, due to the use of graph algorithms, which are best explained using graph theory.

The representation of mazes as 2-dimensional integer arrays will explained. This information is necessary, to understand the implementations of the algorithm in the prototype, and to allow the addition of new algorithms.

The fundamental algorithms are introduced, explaining their decision making. This includes algorithms for both generating.

Graph Theory

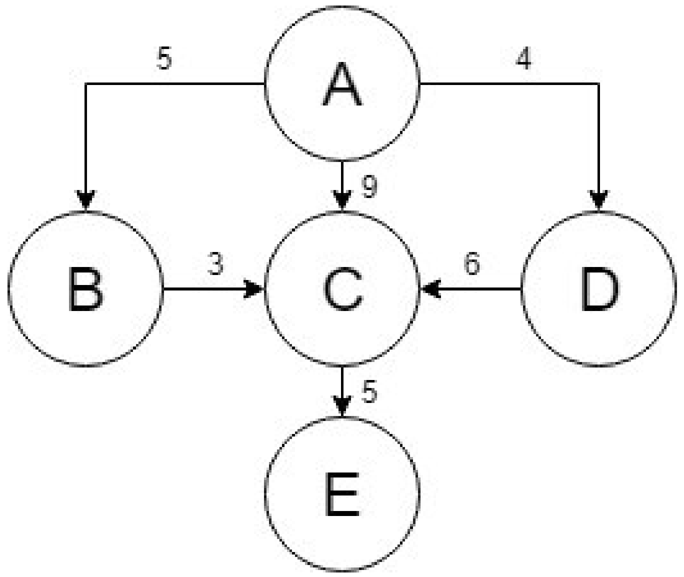
A graph is a data structure similar to a spanning tree. Unlike a tree however, a graph is not structured for easy navigation between data nodes, but rather its structure is dictated by a physical problem. A graph can for example be used to describe a map, with the nodes representing cities on the map, and the edges between the nodes representing railway connections between the cities (Waite and Lafore, 1998).



As seen on graphs are ideal for describing maps. The map from the figure, has been translated into a graph, with the nodes called ”vertices”, representing the points where the freeways intersect, and the lines connecting them, called ”edges”, representing segments of the freeways.

The purpose of the graph is only to illustrate which vertices are connected to each other, and thus all other information is considered irrelevant. For example, the length of an edge in a graph does not represent the actual length between the two vertices the edge connects to, but merely represents that the two vertices are indeed connected.

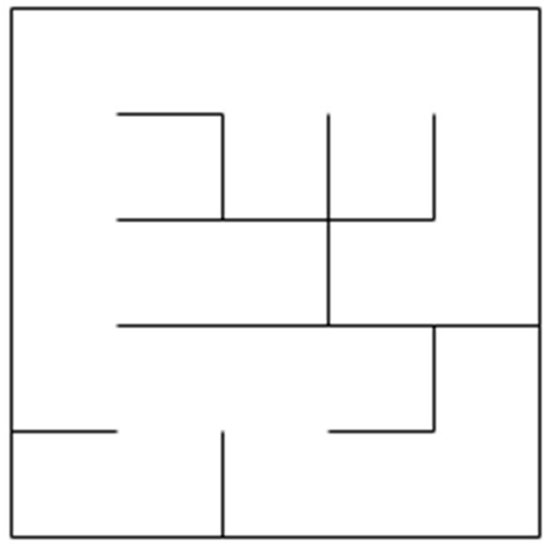
Directed and Weighted Graphs


$$\begin{tikzcd}
 & A \arrow[dl, "5"'] \arrow[d, "9"] \arrow[dr, "4"] & \\
B \arrow[r, "3"] & C \arrow[d, "5"] & D \arrow[l, "6"'] \\
 & E &
\end{tikzcd}$$

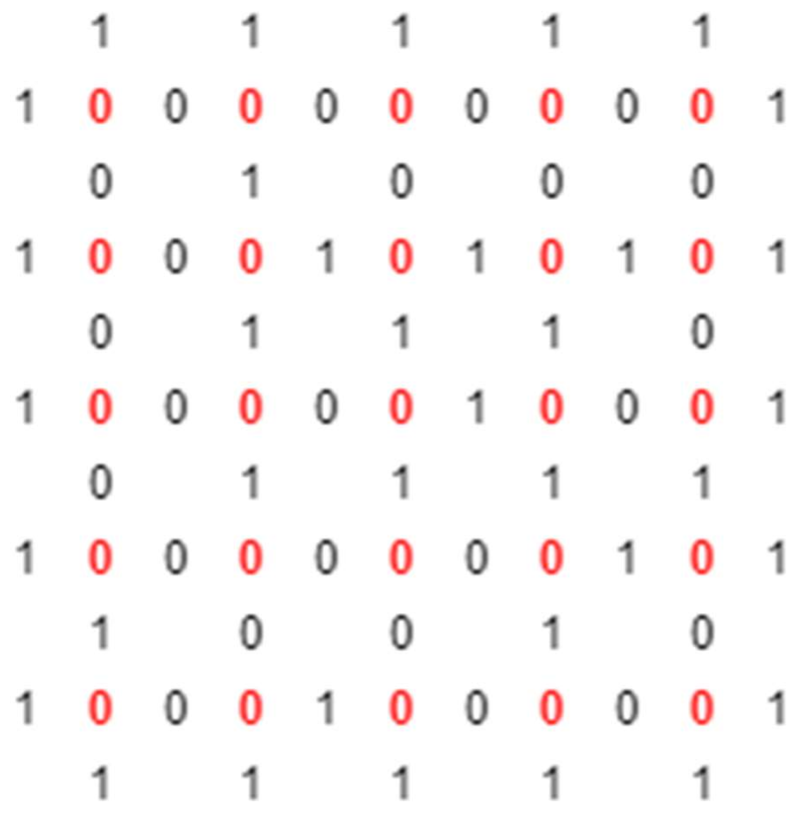
Representation of Mazes

The mazes are generated by the maze-generating algorithms, and should always be solvable by any of the corresponding maze-solving algorithms. All maze-generating algorithms have to be implemented to generate mazes, stored as a two-dimensional array of integers.

```
1  1  1  1  1  1  1  1  1  1  1
1  0  0  0  0  0  0  0  0  0  1
1  0  1  1  1  0  1  0  1  0  1
1  0  0  0  1  0  1  0  1  0  1
1  0  1  1  1  1  1  1  1  0  1
1  0  0  0  0  0  1  0  0  0  1
1  0  1  1  1  1  1  1  1  1  1
1  0  0  0  0  0  0  0  1  0  1
1  1  1  0  1  0  1  1  1  0  1
1  0  0  0  1  0  0  0  0  0  1
1  1  1  1  1  1  1  1  1  1  1
```



A two dimensional array of integers, generated by one of the maze-generating algorithms, and the maze it represents.



	1		1		1		1		1	
1	0	0	0	0	0	0	0	0	0	1
	0		1		0		0		0	
1	0	0	0	1	0	1	0	1	0	1
	0		1		1		1		0	
1	0	0	0	0	0	1	0	0	0	1
	0		1		1		1		1	
1	0	0	0	0	0	0	0	1	0	1
	1		0		0		1		0	
1	0	0	0	1	0	0	0	0	0	1
	1		1		1		1		1	

Functional representation of the array . The red numbers indicate spaces that can be occupied, while the black numbers show movement options, or the lack thereof.

Starting on the uppermost left 0, it can be seen that a wall is present on top of, and to the left of that square, but no wall is present to the right or at the bottom. Moving to the red 0 to the right of that starting 0, it is seen that a wall is present at the top and bottom of that square, but not to the left or right. Every one movement is taken along the squares of the actual maze, this is signified by passing a black 0 in the array representing that maze. As a result, the array will have a width and height of $(2 \cdot n) + 1$, where n is the width and height in squares. For example, the maze in the figure has a width and height of five squares, giving the corresponding array a width and height of $(2 \cdot 5) + 1 = 11$.

Evaluating Algorithms

Time complexity

Time complexity will be measured as the running time of the algorithm, from given input till a solution has been found, including potentially converting the input maze, to something the algorithm will be able to process.

Algorithmic decision-making

Algorithmic decision-making will be counted as the amount of times a given algorithm updates its ”position” in the maze, which will be referred to as the ”iterations”.

Depth-First Generation Algorithm

The DFG algorithm works through very simple principles. It starts by making a maze filled only with walls, similar to how the Prim’s algorithm starts. It then selects a starting point with the restriction, that it must have an odd numbered x- and y-coordinate, and turns this cell into a path (Kano, 2011).

After this setup, it follows the following instructions (Kano, 2011):

1. Check all the directions (up, down, left, right), to see if the cell which is two cells ahead in the direction, contains a wall. If there is a wall two cells in the direction, it is added to a list which is then shuffled.

2. A random direction from this list is then picked, and the algorithm then ”moves” two cells in that direction, and turns these two cells into paths.

3. Steps 1 and 2 are repeated, until a cell is reached, which is surrounded by paths in all direction. Once such a cell is reached, the algorithm backtracks until it finds a cell which is not surrounded by paths, and then repeats step 1 and 2 again.

4. When the algorithm backtracks to the starting point and it is surrounded by paths, the maze is finished. https://www.youtube.com/watch?v=Xthh4SEMA2o

```
from random import randint, shuffle, choice
import sys
#needed for DFS...
sys.setrecursionlimit(10000)
#Each maze cell contains a tuple of directions of cells to which it is connected
#Takes a maze and converts it to an array of 1's and blanks to represent walls, etc
def convert(maze):
    pretty_maze = [[“1"]*(2*len(maze[0])+1) for a in range(2*len(maze)+1)]
    for y,row in enumerate(maze):
        for x,col in enumerate(row):
            pretty_maze[2*y+1][2*x+1] = “0"
            for direction in col:
                pretty_maze[2*y+1+direction[0]][2*x+1+direction[1]] = “0"
    return pretty_maze
```

```
#Takes a converted maze and pretty prints it
def pretty_print(maze):
    for a in convert(maze):
        string = ""
        for b in a:
            string += b
        print (string)
    print("")

#Returns an empty maze of given size
def make_empty_maze(width, height):
    maze = [[[] for b in range(width)] for a in range(height)]
    return maze
```

```
#Recursive backtracker.
#Looks at its neighbors randomly, if unvisitied, visit and recurse
def DFS(maze, coords=(0,0)):
    directions = [(0,1),(1,0),(0,-1),(-1,0)]
    shuffle(directions)
    for direction in directions:
        new_coords = (coords[0] + direction[0], coords[1] + direction[1])
        if (0 <= new_coords[0] < len(maze)) and \
           (0 <= new_coords[1] < len(maze[0])) and \
           not maze[new_coords[0]][new_coords[1]]:
            maze[coords[0]][coords[1]].append(direction)
            maze[new_coords[0]][new_coords[1]].append((-direction[0], -direction[1]))
            DFS(maze, new_coords)
    return maze
```

```
size = 16
pretty_print(DFS(make_empty_maze(size,size)))
```

https://gist.github.com/grantslatton/6906668

https://rosettacode.org/wiki/Maze_generation

Program Mazetest.py

Maze-Solving Algorithms

a simple recursive algorithm:

https://www.laurentluce.com/posts/solving-mazes-using-python-simple-recursivity-and-a-search/

```
grid = [[0, 0, 0, 0, 0, 1],
        [1, 1, 0, 0, 0, 1],
        [0, 0, 0, 1, 0, 0],
        [0, 1, 1, 0, 0, 1],
        [0, 1, 0, 0, 1, 0],
        [0, 1, 0, 0, 0, 2]]
```

```
def search(x, y):
    if grid[x][y] == 2:
        print("found at %d,%d" % (x, y))
        return True
    elif grid[x][y] == 1:
        print ('wall at %d,%d' % (x, y))
        return (False)
    elif grid[x][y] == 3:
        print ('visited at %d,%d' % (x, y))
        return (False)
```

```
print ('visiting %d,%d' % (x, y))
    # mark as visited
    grid[x][y] = 3
    # explore neighbors clockwise starting by the one on the right
    if ((x < (len(grid)-1) and search(x+1, y))
        or (y > 0 and search(x, y-1))
        or (x > 0 and search(x-1, y))
        or (y < len(grid)-1 and search(x, y+1))):
        return True
    return False

search(0, 0)
```

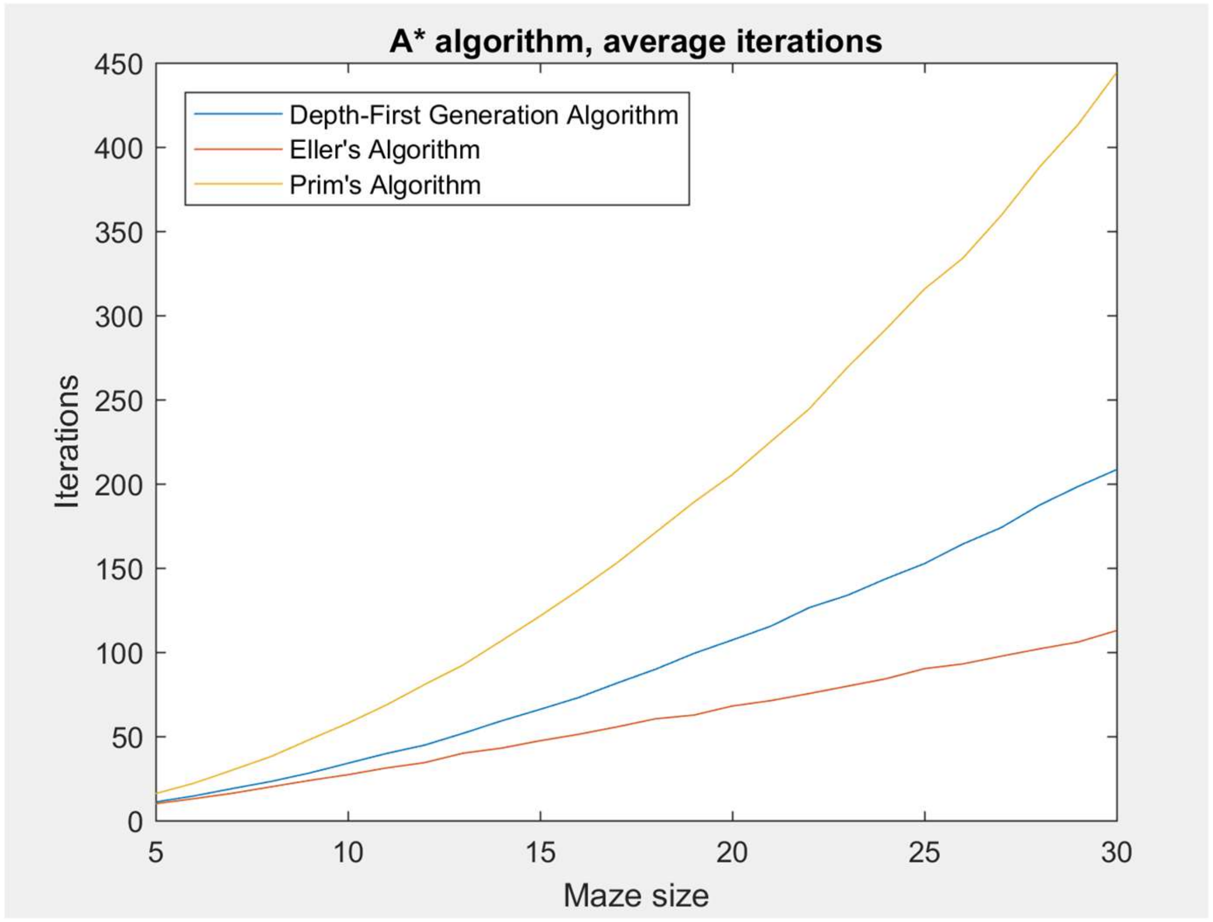
Assignment.

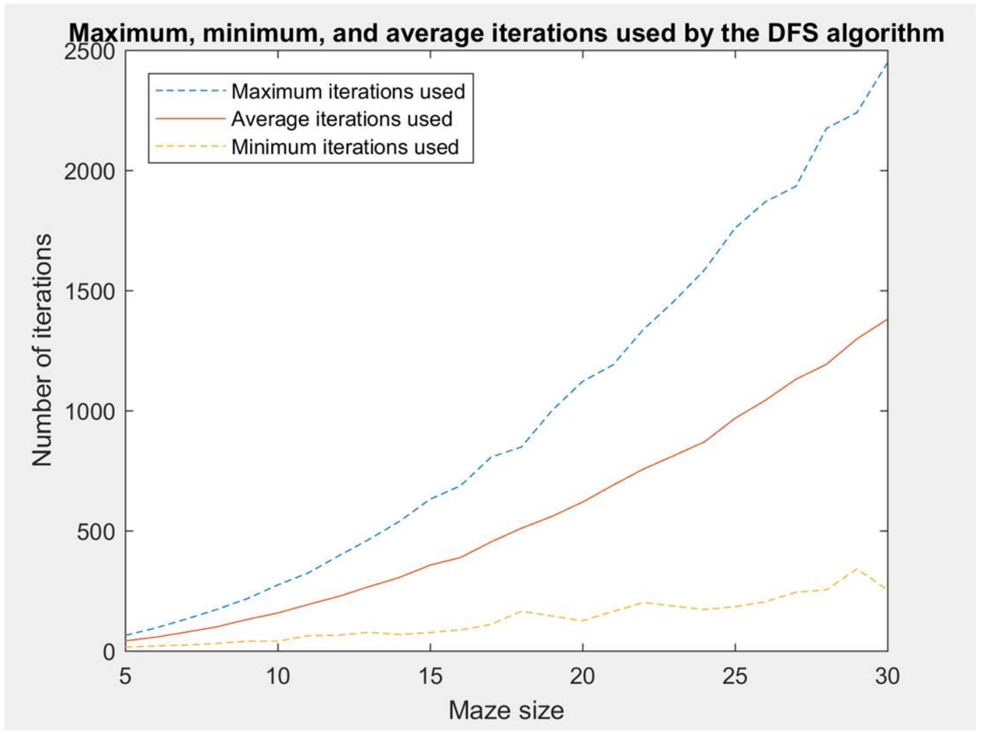
1. An analysis of the maze generation code should be done.
2. The code must be changed so that the output becomes:

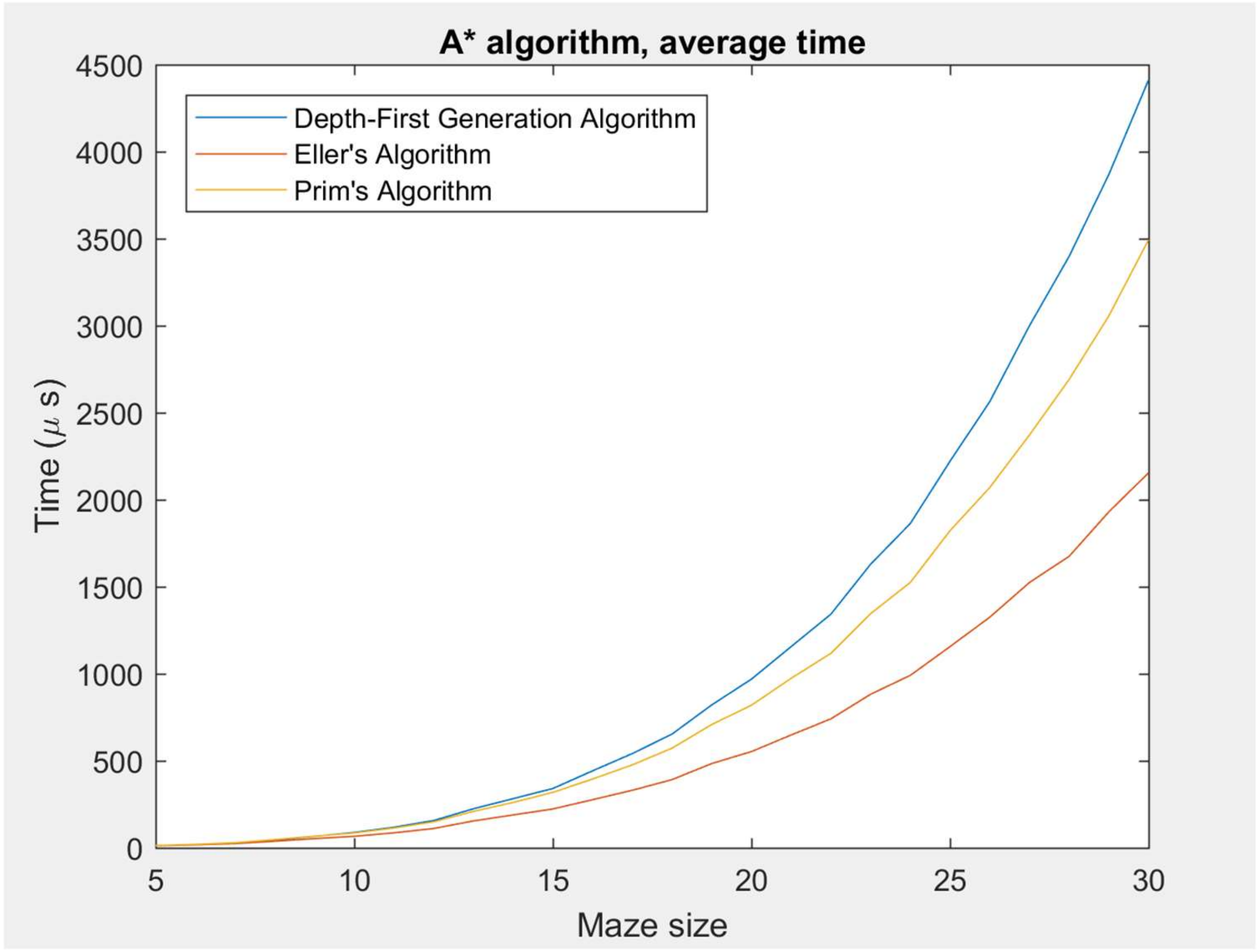
```
grid = [[0, 0, 0, 0, 0, 1],
         [1, 1, 0, 0, 0, 1],
         [0, 0, 0, 1, 0, 0],
         [0, 1, 1, 0, 0, 1],
        [0, 1, 0, 0, 1, 0],
        [0, 1, 0, 0, 0, 2]]
```

3. 6 mazes shall be made, the sizes of which shall be 5, 10, 15, 20, 25 and 30.
4. An analysis of the maze solution code should be done.
5. A counter must be entered into the program indicating where
   many points that have been visited.
6. Put in a timer that measures the time it takes to solve a maze.
7. Solve the 6 maze sizes 10 times and calculate the average value.

The above should be used to make the following plot when we come to lesson 5.


A* algorithm, average iterations
Depth-First Generation Algorithm
Eller's Algorithm
Prim's Algorithm
Iterations
Maze size
0
50
100
150
200
250
300
350
400
450
5
10
15
20
25
30


Maximum, minimum, and average iterations used by the DFS algorithm
Maximum iterations used
Average iterations used
Minimum iterations used
Number of iterations
Maze size
0
500
1000
1500
2000
2500
5
10
15
20
25
30


A* algorithm, average time
Depth-First Generation Algorithm
Eller's Algorithm
Prim's Algorithm
Time (μ s)
Maze size
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5
10
15
20
25
30

8. Both the maze generation and the solution must be inserted into your model in your MVC and view must be able to take the maze size and ? as input and return the time and number of points visited and ?
   The insertion must take into account that more generation and solution algorithms can be used.

Optional.

You can make an A * algorithm and test it against the recursive solution.

The assignment must be completed within 2 weeks and must be completed with your review group. Of course, it must be approved by me after.