NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

PROGRAM OF UNDERGRADUATE STUDIES

BACHELOR THESIS

Procedural Maze Generation

Petros L. Ioannidis

SUPERVISOR: Panagiotis Stamatopoulos, Assistant Professor

ATHENS

JUNE 2016

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΠΡΟΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Διαδικαστική Παραγωγή Λαβύρινθων

Πέτρος Λ. Ιωαννίδης

ΕΠΙΒΛΕΠΩΝ: Παναγιώτης Σταματόπουλος, Επίκουρος Καθηγητής

ΑΘΗΝΑ

ΙΟΥΝΙΟΣ 2016

BACHELOR THESIS

Procedural Maze Generation

Petros L. Ioannidis
A.M.: 1115200900082

SUPERVISOR: Panagiotis Stamatopoulos, Assistant Professor

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Διαδικαστική Παραγωγή Λαβύρινθων

Πέτρος Λ. Ιωαννίδης
Α.Μ.: 1115200900082

ΕΠΙΒΛΕΠΩΝ: Παναγιώτης Σταματόπουλος, Επίκουρος Καθηγητής

ABSTRACT

The purpose of this thesis is to develop and benchmark a set of algorithms used for random procedural maze generation. The goal is to examine both a set of algorithms that produces perfect mazes and a set of algorithms that produces braid mazes. Three algorithms that produce perfect mazes are developed and benchmarked. Those three algorithms already exist in the bibliography and are reimplemented. A new algorithm is proposed regarding the production of braid labyrinths. The proposed algorithm is implemented using 4 different techniques, each one trying to improve the speed and the final form of the algorithm. Finally, to generate and visualize the solution of each maze, a breadth first search solver is implemented. The aforementioned algorithms are implemented using python 2.7 with the addition of Pygame that is utilized to implement the visualization of the maze and its solution.

SUBJECT AREA: Procedural Content Generation, Maze Generation, Braid Maze Generation

KEYWORDS: content generation, maze, braid maze, maze generation, generate and test, random restarts, backtraking

ΠΕΡΙΛΗΨΗ

Ο στόχος της πτυχιακής αυτής είναι η ανάπτυξη και η μέτρηση της απόδοσης ενός συνόλου αλγόριθμων τυχαίας διαδικαστικής παραγωγής λαβυρίνθων. Ο στόχος είναι να μελετηθούν τόσο οι αλγόριθμοι παραγωγής τέλειων λαβυρίνθων όσο και αλγόριθμοι παραγωγής λαβυρίνθων με κύκλους. Για το στόχο αυτό 3 αλγόριθμοι παραγωγής τέλειων λαβυρίνθων αναπτύχθηκαν και μετρήθηκαν. Οι 3 αυτοί αλγόριθμοι υπάρχουν ήδη στη βιβλιογραφία, με την πτυχιακή αυτή να τους υλοποιεί για μέτρηση. Ένας νέος αλγόριθμος αναπτύσσεται και προτείνεται μέσα στην πτυχιακή αυτή σχετικά με την παραγωγή λαβυρίνθων που περιέχουν κύκλους, με τον αλγόριθμο αυτό να υλοποιείται με 4 διαφορετικές τεχνικές με στόχο την βελτίωση της ταχύτητας αλλά και της μορφής του τελικού λαβυρίνθου. Τέλος, για την εύρεση και οπτικοποίηση της λύσης των λαβυρίνθων, ένας αλγόριθμος λύσης με την χρήση αναζήτησης κατά πλάτος υλοποιήθηκε. Οι προαναφερθέντες αλγόριθμοι υλοποιήθηκαν με την χρήση της Python 2.7 και της βιβλιοθήκης Pygame που χρησιμοποιήθηκε για την οπτικοποίηση του κάθε λαβυρίνθου και της λύσης του.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Διαδικαστική Παραγωγή Περιεχομένου, Παραγωγή Λαβυρίνθου, Παραγωγή Λαβυρίνθου με κύκλους

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: παραγωγή περιεχομένου, λαβύριθνος, λαβύριθνος με κύκλους, παραγωγή λαβυρίνθου, παραγωγή και έλεγχος, τυχαία επανεκκίνηση, οπισθοδρόμηση

ACKNOWLEDGEMENTS

For his assistance regarding this thesis I would like to thank my supervisor professor Panagiotis Stamatopoulos. His help and advice were influencial and important during the development of this thesis.

CONTENTS

LIST OF FIGURES

PREFACE

The current thesis was developed within the undergraduate program of the Department of Informatics at the University of Athens. The object was to develop a set of algorithms that is able to generate random mazes of both perfect and braid types and visualize those mazes.

1. INTRODUCTION

A maze is a structure with hallways and walls, and to design one - especially in case of a 2D maze - is not very challenging. The problem of maze generation becomes difficult when the desired maze is going to have large size n. In that case it will be difficult and time consuming for a human designer to create such a maze, while an automated algorithmic designer could use the speed of the computer to generate mazes of large sizes fast and efficient. In addition to speed, another important aspect of such a designer would be the variety of mazes that are produced. For example, an algorithm that created only one maze would be useless after the first run, while an algorithm that is able to produce all possible mazes would be useful on generating mazes since not only there is variety on the mazes created but also there could be no valid technique that would be utilized in order to guess the solution or part of it, the only way for a player to discover a solution would be to solve the maze.

Another type of variety is to be able to generate mazes of different types(subsection 2.1.1), e.g. perfect mazes and braid mazes, since the player could require a different type of challenge. In the case between perfect mazes and braid mazes the structural difference prohibit us from using the algorithms that generate one of the two to be utilized in order to generate the other. In other words, one would have to construct a different designer for those two types of algorithms. The present thesis is one attempt to do so.

Regarding the generation of a perfect maze a variety of algorithms exist on research that solve the problem and do so in various ways. Three basic and important algorithms are presented on this thesis. First of all, the Recursive Backtracker algorithm, which is the most basic algorithm to construct a perfect maze. Secondly, the Kruskal's algorithm, which is a modified version of the original Kruskal's algorithm[2]. Finally, the Wilson's algorithm[5], which is able to uniformly generate every possible maze. The aforementioned algorithms are analyzed in depth on chapter 3.

Contrary to perfect mazes the bibliography regarding the generation of braid mazes is little to non-existent. Through the research that was done in order to complete this thesis, no braid maze generation algorithm was discovered on completed research. In order to compare the two types of mazes and attempt to solve the problem of generating braid mazes, a simple algorithm - the distribution algorithm - was developed. The distribution algorithm, although it generates braid mazes, those mazes contain a significant number of squares, i.e. 4 blocks that form a path which returns to the original block. Having squares reduces the difficulty of the algorithm and when the only path after entering that square is to exit again, it acts as a dead-end. The existence of squares was the reason of the creation of the Generate and Test algorithm, which is the first algorithm on this thesis that generates braid algorithms with no squares. This algorithm basically runs the previous algorithm and filters out all mazes with squares, which makes the algorithm very slow as the size of the maze increases, since the probability of that maze not containing a single square is getting reduced as the size of the maze increases. In order to increase the speed of the generation the Eliminate Squares algorithm was created, on which the

squares where eliminated before the main generation process initiate the construction of the maze - which is still the same process that is used on the distribution algorithm. In case that elimination process was to create an illegal maze, the algorithm would backtrack to the previous square that was eliminated and try to find another way to eliminate it. Contrary to the goal of its creation, this algorithm is even slower than the Generate and Test algorithm, until the Random Restart method is applied. By using the Random Restart method the previous algorithm generates mazes fast, almost as fast as the Distribution algorithm, even as the size increases. All 4 algorithms are described in chapter 4.

Finally, one important aspect is to compare the 2 problems, the perfect maze generation and the braid maze generation, in terms of time complexity. By doing that we could observe the difference in the difficulty of producing such mazes. However, first we need to define what is a maze and its characteristics in order to understand how the algorithms use those characteristics in order to generate those mazes(see chapter 2).

2. BACKGROUND AND RELATED WORK

2.1 Mazes

2.1.1 Definition

A maze is defined as a collection of paths with the following 2 properties[3]:

- The collection of paths must be continuous. That means that for every pair of 2 random tiles t_i, t_j, at least one path exists between them.
- An entrance and an exit must exist.

This definition ensures that the player is able to reach every tile in the maze regardless of his/hers position. In the context of this thesis, we view mazes as puzzles, therefore the previous condition is important, because if we allowed locations that were unreachable to the player, it would be the same as removing them from the maze, thus lowering the difficulty of the maze for the player.

2.1.2 Dimension

The dimension of a maze is the number of dimensions in space that the maze covers. The most common type of mazes are 2-dimensional(see figure 2.7), those are the mazes the algorithms on this thesis generate. In addition, mazes could be 3-dimensional(see figure 2.1), where in this case there are ladders that connect the multiple levels that a 3D maze contains.

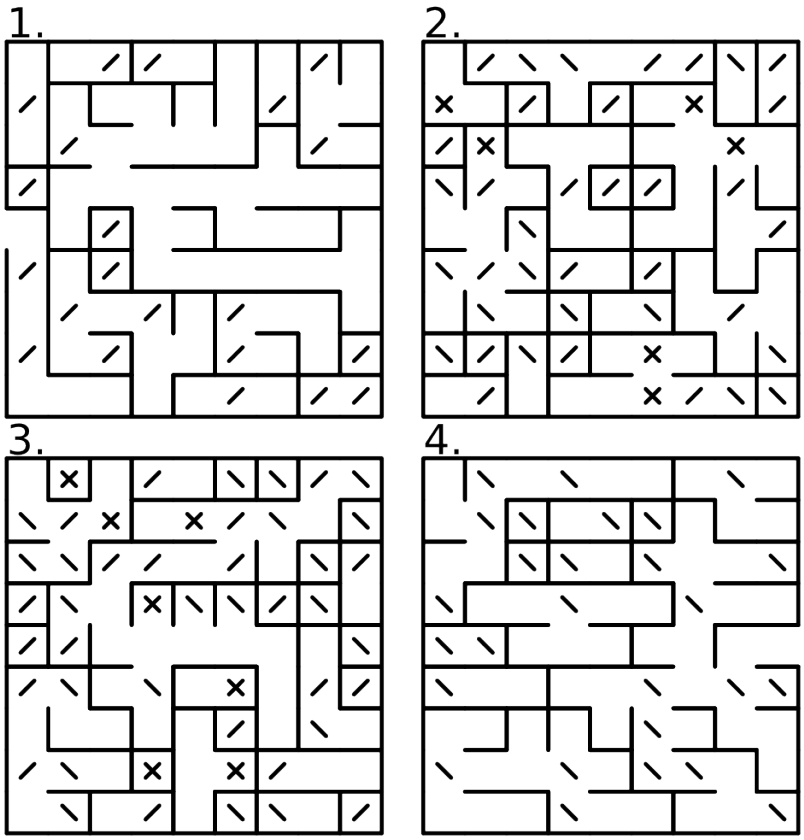
It is possible for a maze to cover a higher dimensional space than the 3D space. For example, a 4-dimensional maze; Such a maze is rendered as a 3D maze, with the 4th dimension accessible through portals(see figure 2.2).

One final class based on the dimension of the maze is the Weave maze. A weave maze covers a 2.5D space, meaning that essentially this class of mazes are 2-dimensional and simultaneously their passages are allowed to overlap each other (see figure 2.3). The difference between a weave maze and a 3D maze is that a 3D maze covers the 3rd dimension with a set of levels that contain ladders, passages and walls, while a weave maze merely uses the 3rd dimension when 2 paths overlap, with the whole maze spawning on a single level.

2.1.3 Hyperdimension

The Hyperdimension class refers to the dimension of the object that is to move through the maze. Based on the dimension of the object a maze is classified as:

Figure 2.1: 3D maze



Vertical layers are numbered starting from the bottom layer to the top. Stairs up are indicated with '/'; stairs down with '\'; and stairs up-and-down with 'x'.

- Non-hypermaze. The object is 1-dimensional, i.e. a point.
- Hypermaze. The object is 2-dimensional, i.e. a line, or higher.

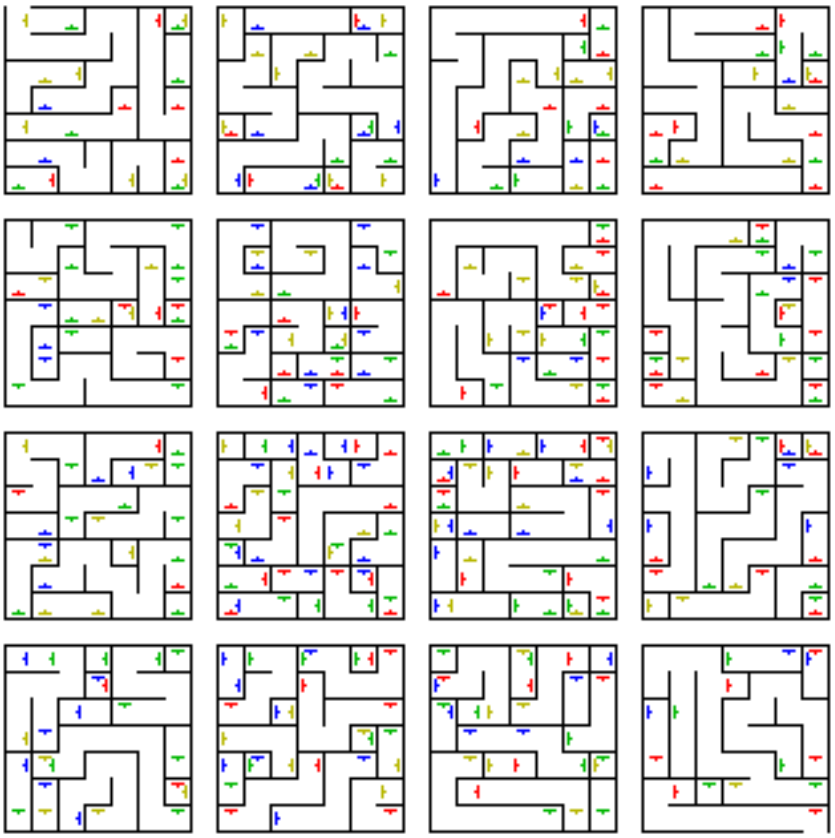
In the case of a non-hypermaze, the solution would be 2-dimensional, i.e. a line, since we move a single point in space. Regarding a hypermaze where the object covers n dimensions, its solution would cover $n + 1$ dimensions(a hypermaze where the object is a line would have as a solution a plane, a hypermaze where the object is a plane would have as a solution a solid etc.).

2.1.4 Topology

A maze is classified based on the geometry of the space that maze exists in. That class is called the Topology class and it contains 2 types of mazes:

- Normal. The geometry of the space is Euclidian(see figure 2.7).

Figure 2.2: 4D maze



- Planair. A maze has an abnormal(non-Euclidian) topology(see figure 2.4).

2.1.5 Tessellation

The geometry of the cells of the maze form th Tessellation class. A cell has one of the following types(see figure 2.5):

(a) Orthogonal or Gamma. The cells form a rectangular grid.

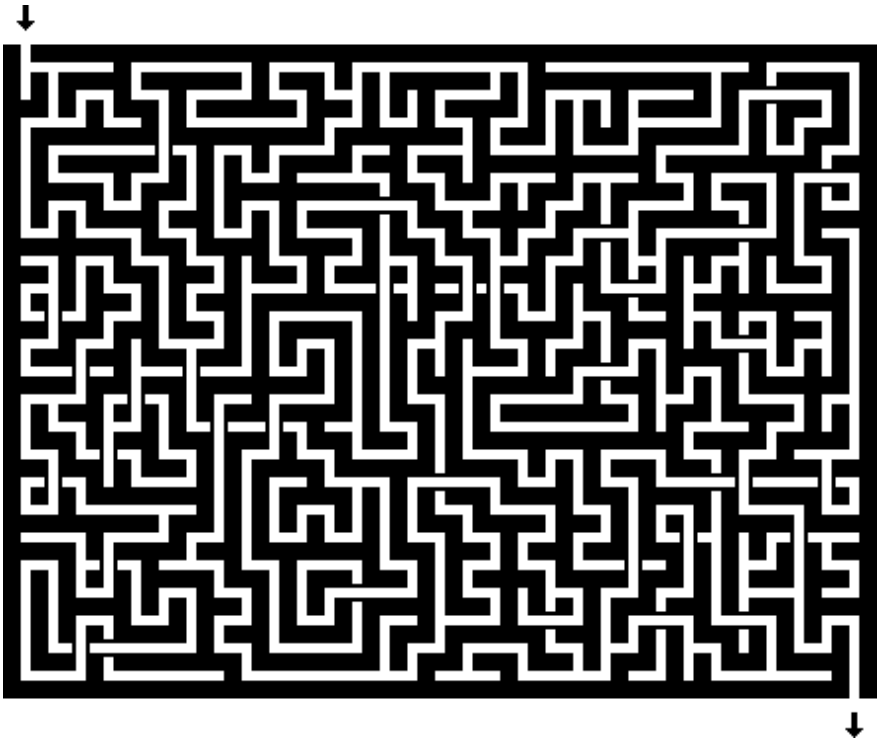
(b) Delta. A Delta maze are composed of interlocking triangles.

(c) Sigma. A Sigma maze are composed of interlocking hexagons.

(d) Theta. Theta Mazes are composed of concentric circles of passages, where the start or finish is in the center, and the other on the outer edge. Cells usually have four possible passage connections, but may have more due to the greater number of cells in outer passage rings.

(e) Upsilon. An Upsilon maze are composed of interlocking octagons and squares.

Figure 2.3: Weave maze



(f) Zeta. A Zeta Maze is on a rectangular grid, except 45 degree angle diagonal passages between cells are allowed in addition to horizontal and vertical ones.

(g) Omega. The term ”omega” refers to most any Maze with a consistent non-orthogonal tessellation.

(h) Crack. A crack Maze is an amorphous Maze without any consistent tessellation, but rather has walls or passages at random angles.

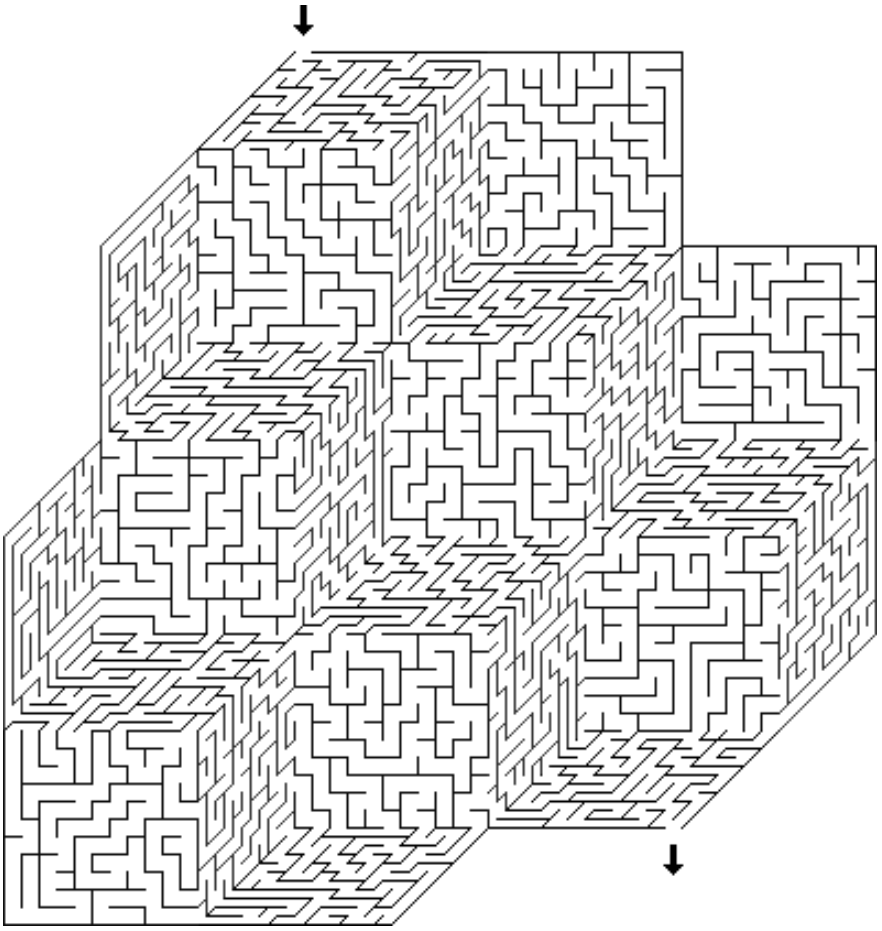
(i) Fractal. A fractal Maze is a Maze composed of smaller Mazes. A nested cell fractal Maze is a Maze with other Mazes tessellated within each cell, where the process may be repeated multiple times. An infinite recursive fractal Maze is a true fractal, where the Maze contains copies of itself, and is in effect an infinitely large Maze.

2.1.6 Routing

Based on the aforementioned definition we could further classify the maze structure based on the properties of the paths that construct the maze. One basic property is whether or not the paths have branches. In this case the mazes are classified as follows[3]:

- A unicursal maze, where no branches exists. This type of maze contains only a single path from the entrance to the exit, therefore no loops or dead-ends exist in the maze(see figure 2.6).

Figure 2.4: Planair maze



- A multicursal maze, where branches are allowed. This type of maze contains multiple paths that lead to dead-ends or the exit. In addition, loops may exist in this type of maze(see figure 2.7).

A unicursal maze poses no challenge for a player, since the player only has to follow the path and he/she is taken directly to the exit of the maze. In contrast, a multicursal maze requires from the player to choose which path to follow and therefore requires from the player to solve the maze in order to escape. In this thesis we are going to work with multicursal mazes.

From figure 2.7 we are able to observe another classification that applies only to multicursal mazes, and that is the manner of their branching, which is dividing the multicursal mazes into 2 categories[3]:

- A perfect maze. A maze with branches and dead-ends.(chapter 3)
- A braid maze. A maze with branches and loops.(chapter 4)

2.2 Maze generation algorithms

The purpose of a maze generation algorithm is to generate a maze without the assistance of a user, thus allowing the creation of infinitely many mazes. Some of those algorithms, like the Wilson's algorithm[5], are able to uniformly spawn every possible maze, which means that we are able to store every maze using just the seed the random number generator uses to produce that maze. This leads to storing the maze more efficiently, along with automatically generating different maze puzzles without having to design them.

Maze generation algorithms have 8 basic characteristics that define the way that algorithm generates the maze. Consequently, this also defines the characteristics of the maze that is to be created. Depending on the algorithm used, the mazes that are produced tend to have similar characteristics. The 8 basic characteristics maze generation algorithmes have are as follows:

- Dead-end percentage. The percentage of cells in the maze that are dead-ends.
- Type. This could either be a tree or a set depending on the way the algorithm constructs the maze. A tree type algorithm would build the maze by expanding already built areas, while a set type algorithm could expand any part of the maze in any order.
- Focus. An algorithm focuses on walls or passages in order to build the maze.
- Bias free. A bias free algorithm treats all directions and sides of a maze equally.
- Uniform. This is whether the algorithm generates all possible mazes with equal probability. The algorithm could either be uniform, non-uniform or not able to generate all possible mazes at all.
- Memory. The memory complexity of the algorithm.
- Time. The time complexity of the algorithm.
- Solution percentage. The percentage of cells of the maze that the solution contains.

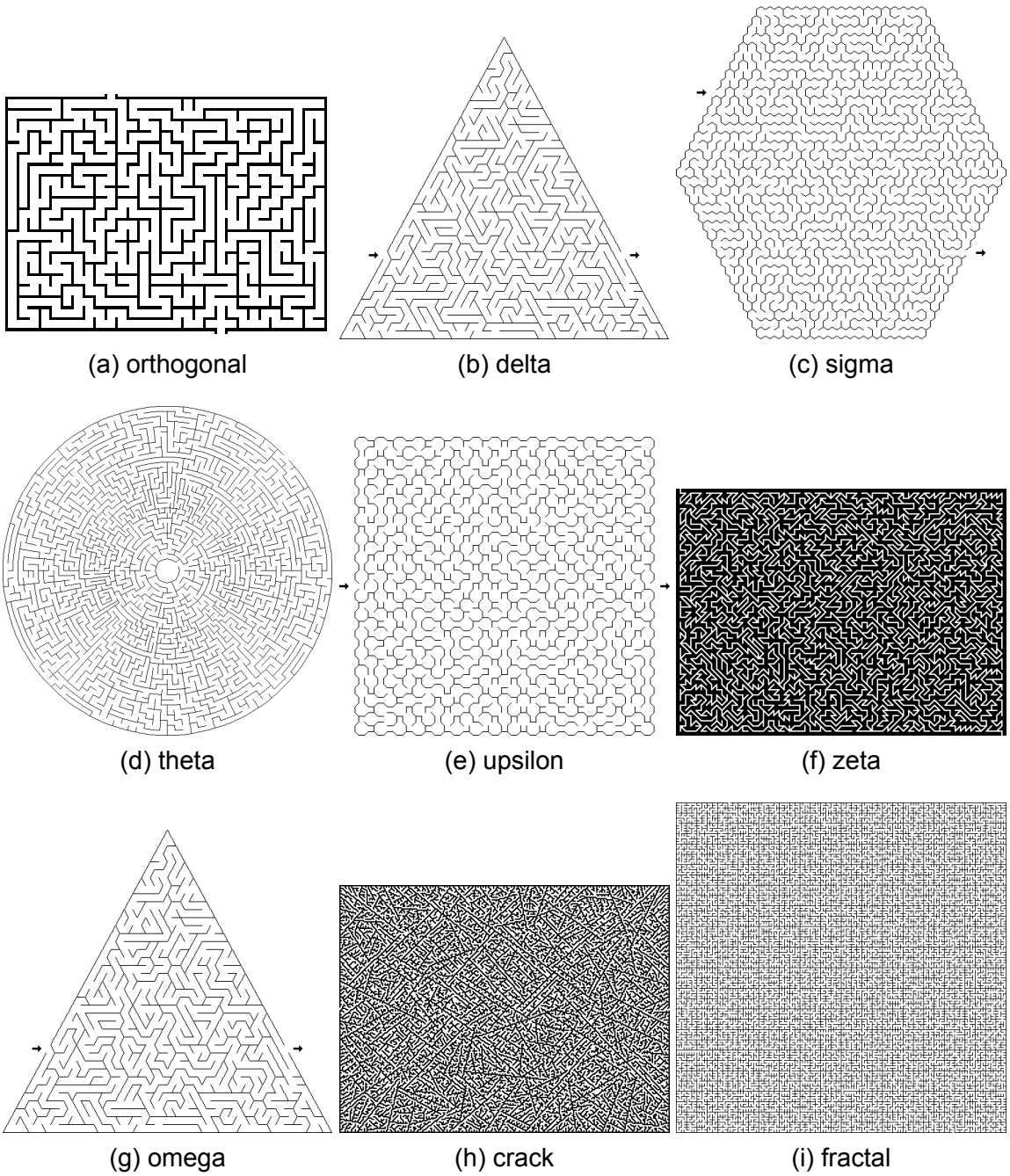
2.3 Maze solving algorithms

Many of the algorithms that exist with the purpose of solving a maze are based on path-finding algorithms that operate on graphs. The reason for this is that many of the properties a maze possesses are similar to properties of a graph.

For example, a perfect maze(chapter 3) is equivalent to a tree in graph theory, since a perfect maze contains no loops, and as a result we are able to use path-finding algorithms that work on trees in order to solve a perfect maze, e.g. Recursive Backtracker.

Similarly, a braid maze is an undirected graph that contain loops with all edges having a cost of 1, which is the reason why a uniform-cost search[4] is able to do a complete search in a braid maze(chapter 5), while a Depth-first search does not guarantee the discovery of the shortest path in this case.

Figure 2.5: Tessellation



(a) orthogonal (b) delta (c) sigma

(d) theta (e) upsilon (f) zeta

(g) omega (h) crack (i) fractal

Figure 2.6: Unicursal maze

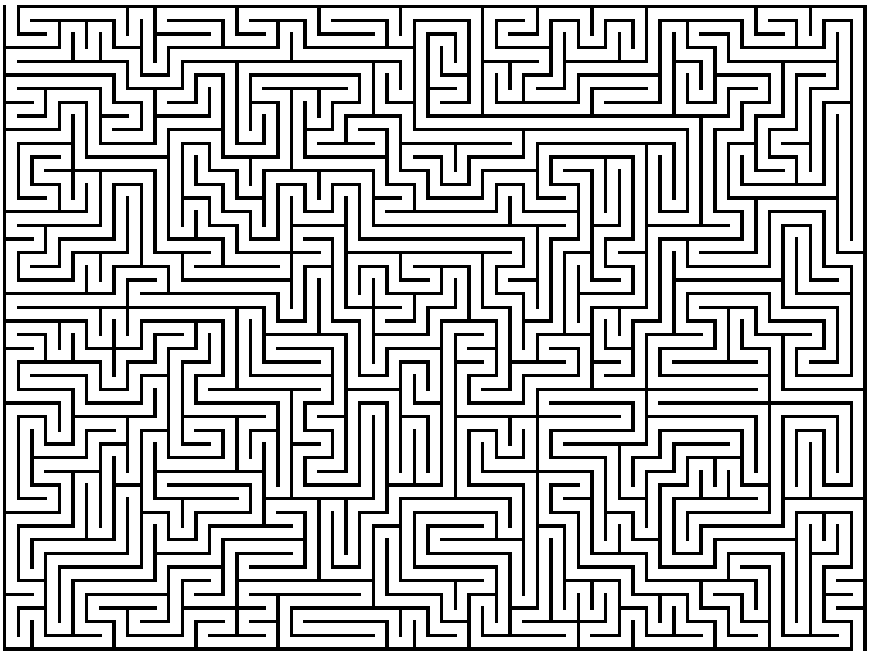
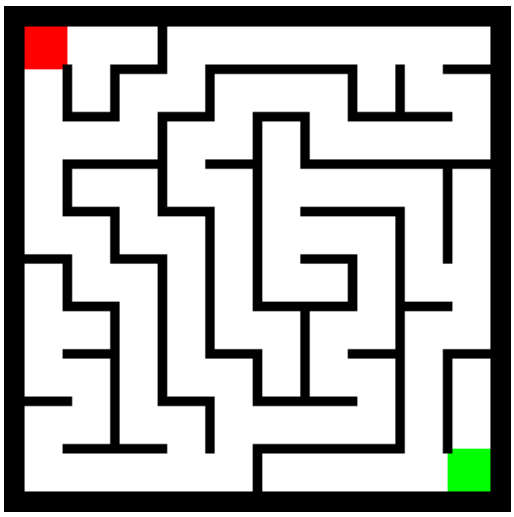
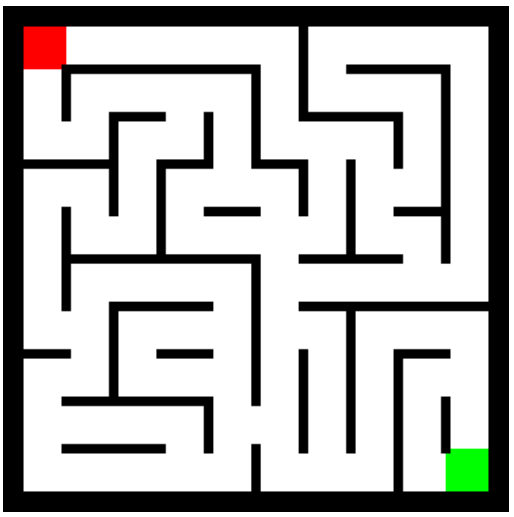


Figure 2.7: Multicursal maze



(a) A multicursal maze with dead-ends



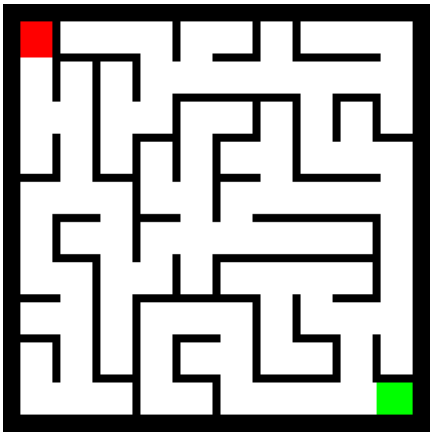
(b) A multicursal maze with loops

3. PERFECT MAZE

A perfect maze is defined as a maze with no loops inside(see figure 3.1). Formally, this is defined as:

> A perfect maze is a maze where for every pair of 2 random tiles t_i, t_j exactly one path exists between them.

Figure 3.1: A perfect maze



In the following algorithms we choose the entrance of the maze to be the top left-hand corner(red cell) and the exit to be the lower right-hand corner(green cell). Since the definition of a perfect maze states that for every pair of tiles a path between them exists, the construction of the maze is not altered by the entrance and exit tiles, thus we are allowed to choose those 2 tiles arbitrarily.

3.1 Recursive Backtracker Algorithm

One simple way to procedurally generate random mazes is to use the Recursive Backtracker algorithm which is based on the Depth-first search algorithm[4]. The idea behind this method is that we start with a maze without any carved cells, we choose a cell at random, we start carving a passage starting from that cell and mark it as visited. Then, to continue the passage, we choose a random neighbor of that cell, we carve that neighbor, mark it as visited and connect the 2 cells. This procedure is continued until the random neighbor we chose has already been visited and carved. When that happens we stop and continue with the next neighbor of that cell. When all the neighbors of the cell have already been visited and carved, we backtrack to the previous cell until all its neighbors are visited and carved. In the end of this procedure we are left with a perfect maze(see figure 3.2).

Due to the nature of backtracking there are two possible implementations. Firstly, backtracking is possible to be implemented using recursion. Secondly, it could be implemented using an iterative implementation with the assistance of a stack data structure.

3.1.1 Recursive implementation

A simple way to implement backtracking in an algorithm is with the use of recursion, and that is the first implementation that is described in this thesis. The reason that backtracking is easy to implement using recursion is that there is no need to keep track of which cell we will need to backtrack to, it is already stored on the Python stack.

Each time we need to move to the next cell we execute a new function that stores all the data needed for the execution of the part of the algorithm regarding the new cell. In addition, when we finish the procedure regarding that cell, the return call returns the execution of the program to the previous cell, i.e. the previous procedure that is stored in the Python stack and has not finished executing yet.

Algorithm 1 recursive traverse

```
1:  function recursive_traverse(start)
2:      if start is visited OR start is out of bounds then
3:          return False
4:      end if
5:      start ← visited
6:      neighbor_list ← neighbors(start)
7:      neighbor_list ← shuffle(neighbor_list)
8:      for neighbor in neighbor_list do
9:          connection ← recursive_traverse(neighbor)
10:         if connection = True then
11:             connect(start, neighbor)
12:         end if
13:     end for
14:     return True
15: end function
```

3.1.2 Iterative implementation

Using the Python stack, although easy to implement, is not memory efficient. Generating a function stack for each cell is resource draining and when the number of cells becomes large the number of function stacks increases.

To solve this problem we implement an iterative implementation of the same algorithm using a Stack data structure. Instead of relying on the stack to keep track of the cell we need to backtrack to, we keep track of that information inside the stack. Every time we

complete processing a cell we pop the next cell from the stack. In addition, everytime we process a new cell, all its neighbors are pushed in the stack. This greatly reduces the memory utilized, especially for large mazes.

Algorithm 2 iterative traverse

```
1:  function iterative_traverse(start)
2:      st ← stack()
3:      push(st, (start, start))
4:      while st is not empty do
5:          (start, new_start) ← pop(st)
6:          if start is visited OR start is out of bounds then
7:              continue
8:          end if
9:          start ← visited
10:         if start ≠ new_start then
11:             connect(start, new_start)
12:         end if
13:         neighbor_list ← neighbors(new_start)
14:         neighbor_list ← shuffle(neighbor_list)
15:         for neighbor in neighbor_list do
16:             push(st, (new_start, neighbor))
17:         end for
18:     end while
19: end function
```

3.2 Kruskal's Algorithm

One of the algorithms that are borrowed from graph theory is the Kruskal's Algorithm[1] which is used in graph theory in order to compute the minimum-cost path between 2 vertices of a connected weighted graph. In order for this algorithm to generate random mazes a slightly different randomized version is used(see Algorithm 3).

In the beginning, we create one set for each cell t_i that exists in the maze and we add to that set the t_i cell. In addition, we create a list of all possible walls that could exist in the maze, with that list having the form $[\dots,(t_i,t_j),\dots]$ where (t_i,t_j) means that a wall exists between the cell t_i and the cell t_j. Then, the list of walls is shuffled, and on that shuffled list we start iterating on its members. For every wall that we iterate on, we check if the cells that are divided by this wall belong to distinct sets, and if that is true we connect the two cells and join the sets of the the cells that were divided before.

The end result is a perfect maze(see figure 3.3).

Algorithm 3 Kruskal's Algorithm

```
1: cell_set[t_i] ← set(t_i)∀t_i where t_i cells in maze
2: walls ← (t_i, t_j), ∀t_i, t_j where t_i, t_j neighboring cells
3: for wall in walls do
4:     if t_i not in cell_set[t_j] and t_j not in cell_set[t_i] then
5:         connect(t_i, t_j)
6:         joined_set ← cell_set[t_i] ∪ cell_set[t_j]
7:         for cell in joined_set do
8:             cell_set[cell] ← joined_set
9:         end for
10:    end if
11: end for
```

3.3 Wilson's Algorithm

Wilson's Algorithm[5] was originally created in order to generate random spanning trees. A spanning tree T of an undirected graph G is a subgraph that is a tree which includes all of the vertices of G. Consider a maze with no walls, such a maze is an undirected graph G. By using the Wilson's algorithm on that maze we generate a random spanning tree, which is a perfect maze, since tree graphs are equivalent to perfect mazes. In addition, Wilson's Algorithm generates a uniform spanning tree, since it chooses which spanning tree to create uniformly in terms of probabilities, therefore this algorithm has the uniform maze generation property that was described above, thus generating all possible mazes with equal probability.

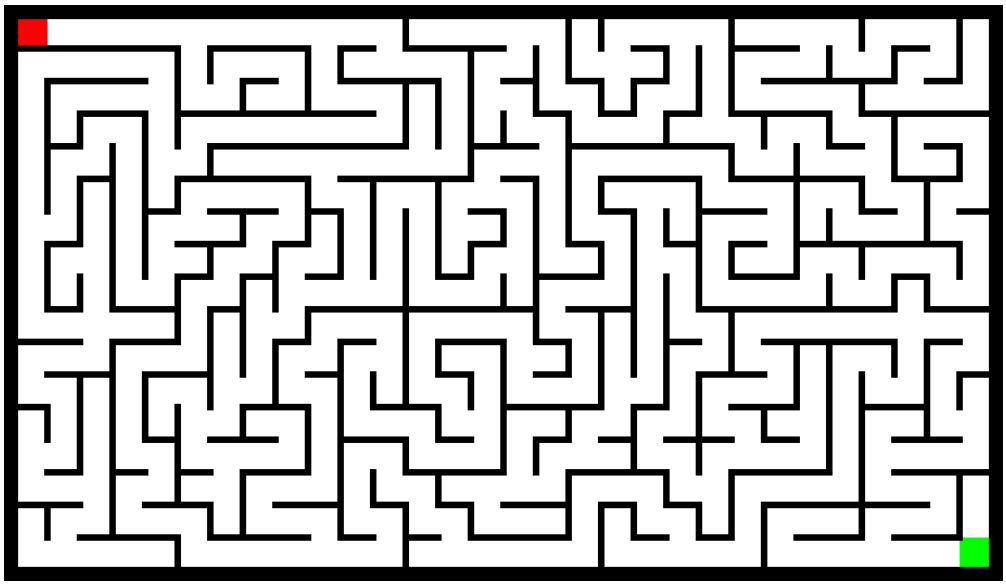
First, 2 sets are created, a set $visited\_cell$ set which is empty, and a $non\_visited\_cell$ set which contains all t_i cells of the maze in random order. The cell that it is going to be used as the starting point is removed from the $non\_visited\_cell$ set and is added to the $visited\_cell$ set. While non visited cells still exists, a new path is being carved by first taking a non visited cell and removing it from the $non\_visited\_cell$ set, then one of its neighbors is chosen and added to the path. If that neighbor has not been visited yet, repeat the previous process. Otherwise, every cell in the path is removed from the $non\_visited\_cell$ set and is added to the $visited\_cell$ set. Next, that path is carved in the maze. This process is repeated until the $non\_visited\_cell$ set is empty.

In the end we are left with a perfect maze(see figure 3.4).

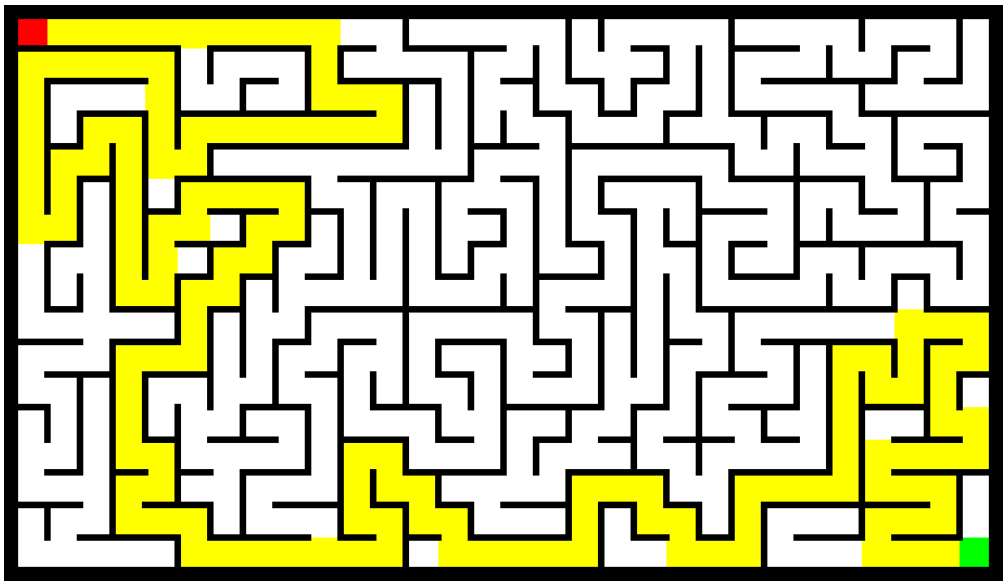
Algorithm 4 Wilson's Algorithm

```
1: visited_cells ← empty_set()
2: non_visited_cells ← t_i ∀t_i where t_i cells in maze
3: non_visited_cells ← shuffle(non_visited_cells)
4: start ← pop(non_visited_cells)
5: visited_cells ← visited_cells ∪ start
6: while non_visited_cells is not empty do
7:     path ← {}
8:     tile ← pop(non_visited_cells)
9:     while tile not in visited_cells do
10:        neighboring_cells ← neighbors(tile)
11:        neighboring_cells ← shuffle(neighboring_cells)
12:        path[tile] ← pop(neighboring_cells)
13:        tile ← path[tile]
14:    end while
15:    for tile in path do
16:        non_visited_cells ← non_visited_cells \ tile
17:        visited_cells ← visited_cells ∪ tile
18:        connect(tile, path[tile])
19:    end for
20: end while
```

Figure 3.2: DFS maze

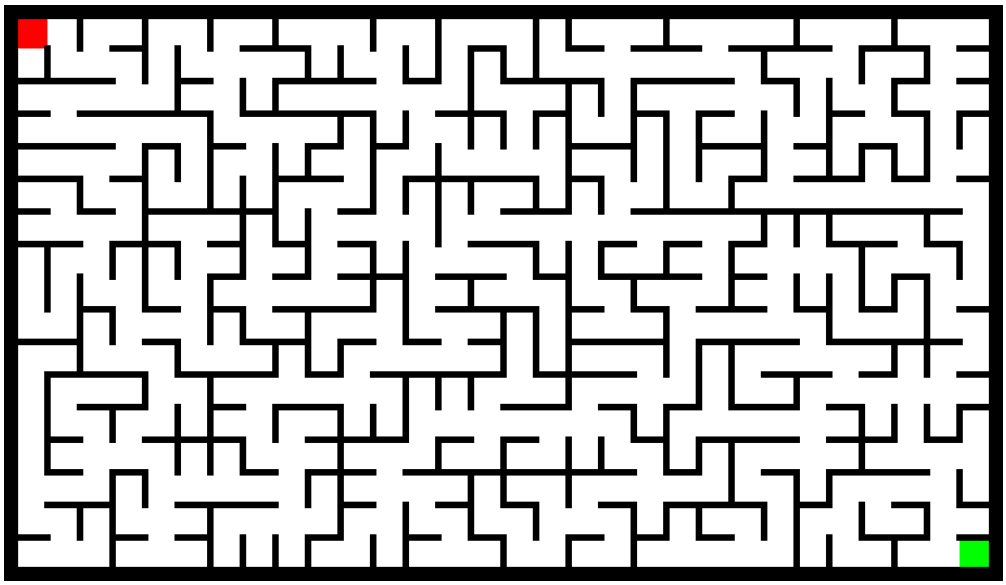


(a) DFS maze hidden

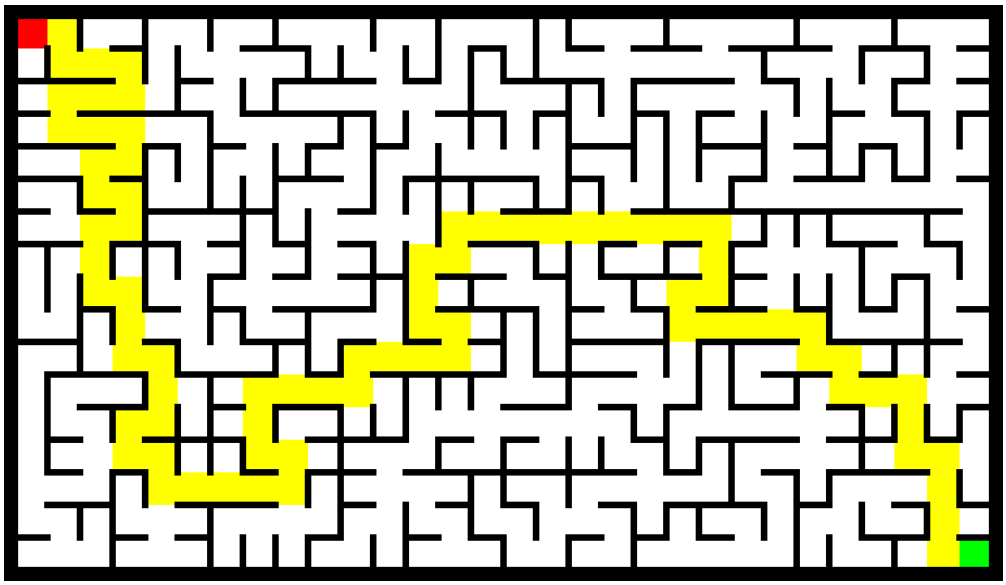


(b) DFS maze solved

Figure 3.3: Kruskal maze

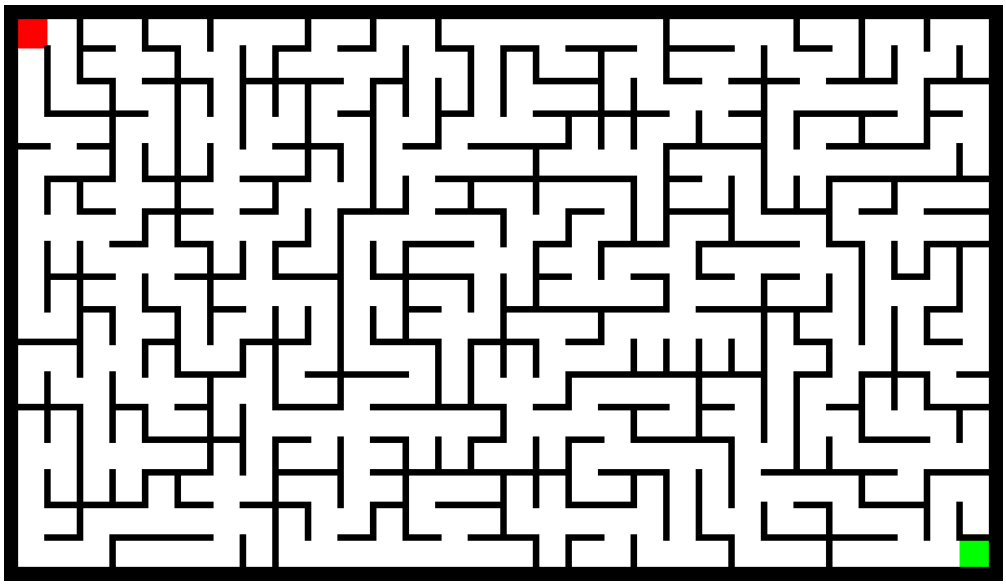


(a) Kruskal maze hidden

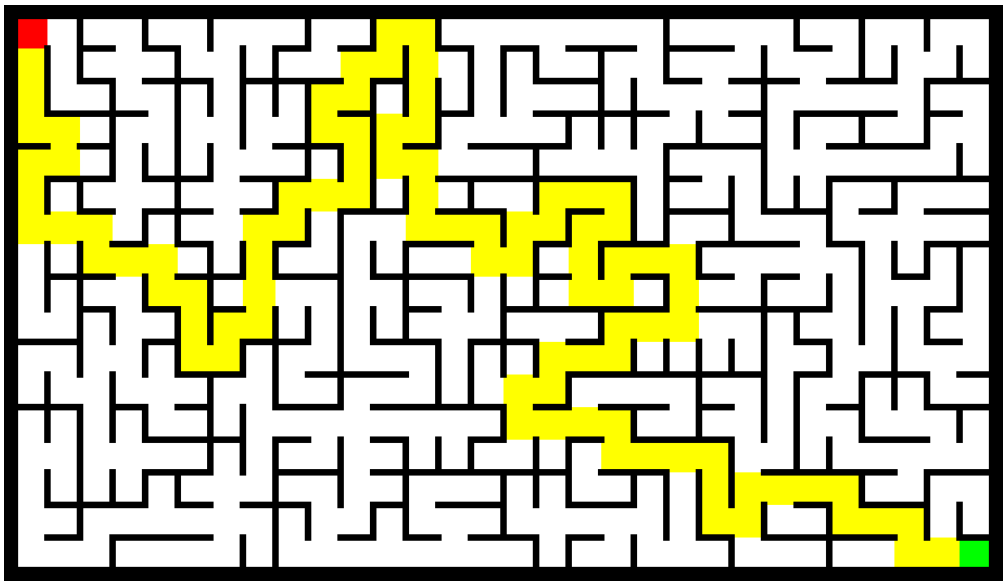


(b) Kruskal maze solved

Figure 3.4: Wilson maze



(a) Wilson maze hidden



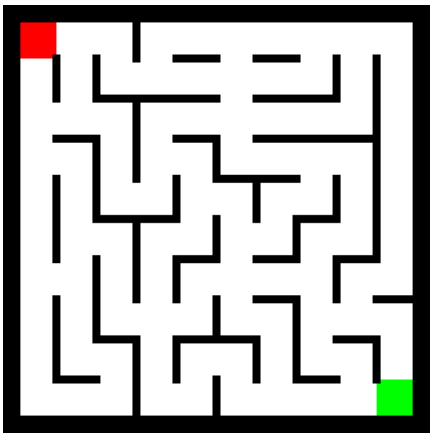
(b) Wilson maze solved

4. BRAID MAZE

A braid maze is defined as a maze in which no dead-end exists(see figure 4.1). In context of the algorithms discussed later we will give the following formal definition.

> A braid maze is a maze where for every tile inside that maze the number of neighbors for that tile must be greater than 1.

Figure 4.1: A braid maze



Contrary to perfect mazes, there is no bibliography regarding the procedural random generation of braid mazes. In this chapter we are going to construct such an algorithm(see algorithm 5).

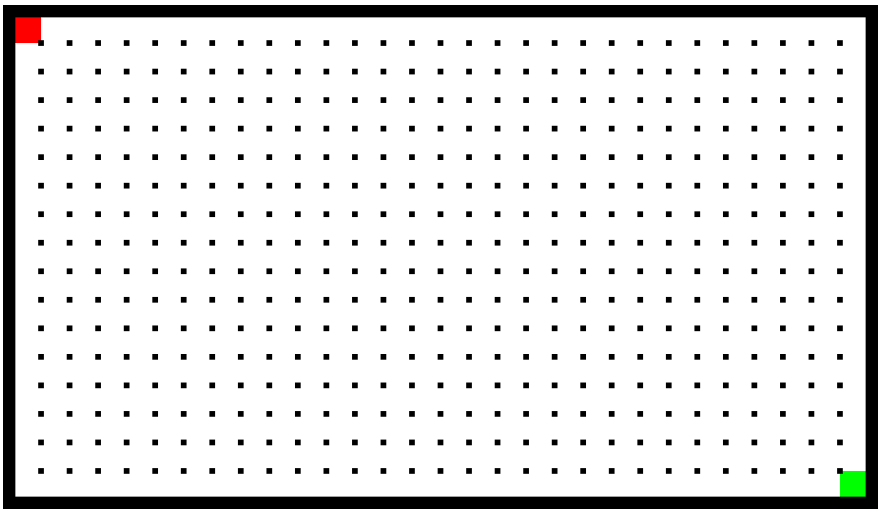
The algorithm focuses on walls to generate the maze, with the initial state of the algorithm to be an empty maze, i.e. a maze free of walls with only an entrance and an exit(see figure 4.2).

The next step is to calculate the number of neighbors each cell will have - inside our algorithm this number is used as a soft-constraint for the creation of the braid maze. The result is a $i \times j$ matrix, where i and j the sizes of the maze. In that matrix, for each cell, a number - either a 2 or a 3 - is stored(see figure 4.3) The matrix is the foundation of the maze and one of the parts of the algorithm that could be altered in order to create different braid mazes. Depending on the way these values are chosen the final maze changes.

After the matrix is created, the algorithm starts building the maze by choosing a tile, then choosing one of its neighbors and disconnecting them by adding a wall between them. As we can observe, in the initial state of the maze(4.2) there are no walls, therefore a solution exists. Each time a wall is added to the maze there is a function `flood` which floods the maze with water and checking if the water reaches every tile of the maze, thus ensuring the property that there is a path between every pair of tiles(see section 2.1.1).

When the requested amount of walls is carved for a tile, that tile is removed from the list and the algorithm continues the process for a second tile, until all tiles are processed.

Figure 4.2: An empty maze



4.1 Braid maze generation with probability distribution

One simple method to generate the neighbor matrix that is mentioned in the previous section, is to treat each cell on that matrix as a random variable. That is the concept of the first algorithm that we used to generate braid mazes. In more details, a probability is assigned for the value of 2 and the value of 3 is treated as the complementary event. Next, the algorithm iterates on each cell and randomly assignes a value to that cell based on the probabilities that are used. In the example given in the previous section this algorithm is used with the probability n that a cell will contain a value of 2 being 0.7.(see algorithm 6)

4.2 Braid Algorithm Generate and Test

As we can observe on the maze 4.3 generated by the previous algorithm, a significant number of squares exist. A square is defined as:

> Suppose there are 4 cells c_0, c_1, c_2, c_3 in a maze with the following coordinates respectively: $(i, j), (i + 1, j), (i + 1, j + 1), (i, j + 1)$. Then those 4 cells form a square if and only if c_0 is connected with c_1, c_1 with c_2, c_2 with c_3 and c_3 with c_0.

Those squares make the maze more organic, making it more similar to a realistic cluster of structures(e.g. those squares could represent building blocks). However, those squares reduce the level of difficulty of a maze since they pose no challenge neither for a computer nor a human player. This results into a simpler easier maze.

In order to eliminate those squares a test has been devised(see algorithm 7).

The first algorithm that we developed to eliminate squares uses a generate and test technique. It generates a maze using algorithm 5 and then for each square of the algorithm

Algorithm 5 Braid Algorithm

```
1: cells ← t_i ∀t_i where t_i cells in maze
2: cells ← shuffle(cells)
3: while cells do
4:     cell ← pop(cells)
5:     neighbor_list ← neighbors(cell)
6:     for neighbor in neighbor_list do
7:         if matrix(cell) ≥ neighbors(cell) then
8:             break
9:         end if
10:        if matrix(neighbor) ≥ neighbors(neighbor) then
11:            continue
12:        end if
13:        connect(cell, neighbor)
14:        if flood(maze) == False then
15:            disconnect(cell, neighbor)
16:        end if
17:    end for
18: end while
```

Algorithm 6 Distribution Algorithm

```
1: probability ← n
2: for cell in cells do
3:     random ← uniform_distribution(0, 1)
4:     if random ≤ n then
5:         cell ← 2
6:     else
7:         cell ← 3
8:     end if
9: end for
```

executed the test presented on algorithm 7 7. If that test succeed for each cell of the maze(minus the final row and column since its evaluation is done indirectly from evaluating all the other cells), then the maze is presented as a result. Otherwise, the maze is discarded and a new maze is generated by algorithm 5(see algorithm 8).

The result is a maze with no squares inside its paths.(see figure 4.4)

4.3 Pre-Eliminate Squares

Although the method of the generate and test algorithm(algorithm 4.4) provides a solution regarding eliminating the squares of the maze, that method is highly inefficient and only generates mazes of relatively small sizes. The goal is to produce challenging mazes,

Algorithm 7 Test for square algorithm

```
1: function test_for_square(cell_x, cell_y)
2:     test ← False
3:     coordinates ← ((cell_x, cell_y), (cell_x, cell_y + 1), (cell_x + 1, cell_y + 1), (cell_x + 1, cell_y), (cell_x, cell_y))
4:     coordinates ← coordinates \ out of maze bounds cells
5:     if |coordinates| < 5 then
6:         test ← True
7:     end if
8:     count ← 0
9:     while count < 4 do
10:        count ← count + 1
11:        if coordinates[count] not connected with coordinates[count + 1] then
12:            test ← True
13:            break
14:        end if
15:    end while
16:    return test
17: end function
```

and one important aspect of a challenging maze is its size. Therefore, while the generate and test technique provides an example of the type of algorithm we wish to generate, it does not solve the problem. The problem with that method is that as sizes increase, the probability of randomly producing a maze with no squares decreases.

For that reason a second algorithm was devised with the main goal of minimizing the number of squares that are created while building the maze, and in the end trying to eliminate the squares that were created.

First, the algorithm calculates all squares that exist in the maze, and since in the beginning the maze is empty 4.2 the calculation is done using the definition of the square and calculating all possible groups of cells that fulfill the definition. Each of those squares only needs one of those pairs of cells disconnected(a wall between those cells). Since the set of those squares is calculated the algorithm chooses one square at random and then one pair at random and raises a wall between those cells. Then, it continues with the next square until there are no squares left or there is no valid pair for some square. Whenever the algorithm is required to raise a wall between 2 cells that have 2 neighbors it considers that pair invalid, since by raising a wall between them will create a dead end. In case this happens the algorithm backtracks to the previous square that eliminated and tries another pair until a valid solution for all squares is constructed.

On each square the algorithm has 4 choices on which wall to raise, and the number of squares is $(n-1)*(m-1)$ given that the size of the maze is $n \times m$. Since the backtracking algorithm is a Depth-first search[4] with a branching factor of 4 and depth $(n-1)*(m-1)$ the worst case time complexity is $O(4^{(n-1)\times(m-1)})$.

Algorithm 8 Generate and Test Algorithm

```
1: no_squares ← False
2: while no_squares == False do
3:     cells ← t_i ∀t_i where t_i cells in maze
4:     cells ← shuffle(cells)
5:     while cells do
6:         cell ← pop(cells)
7:         neighbor_list ← neighbors(cell)
8:         for neighbor in neighbor_list do
9:             if matrix(cell) ≥ neighbors(cell) then
10:                break
11:            end if
12:            if matrix(neighbor) ≥ neighbors(neighbor) then
13:                continue
14:            end if
15:            connect(cell, neighbor)
16:            if flood(maze) == False then
17:                disconnect(cell, neighbor)
18:            end if
19:        end for
20:    end while
21:    no_squares ← True
22:    cells ← t_i ∀t_i where t_i cells in maze
23:    for cell in cells do
24:        if test_for_square(cell) = False then
25:            no_squares ← False
26:            break
27:        end if
28:    end for
29: end while
```

That process leaves the maze in a state without squares. When this process is finished the distribution algorithm(algorithm 6) takes place to add some final walls to the maze. In the end a braid maze with no squares is generated(see figure 4.5).

4.4 Random restarts algorithm

The pre-eliminate squares algorithm solves the problem faster and for larger mazes than the generate and test algorithm, since the algorithm is allowed to fix the existing squares, instead of waiting to randomly disappear from the maze. However, one problem that arises in the pre-eliminate squares algorithm is that there are instances on which the algorithm makes a choice close to the root of the backtracking tree, that leads to an unsolv-

Algorithm 9 Pre-eliminate squares

```
1: squares ← ∪_{∀c_i}(c_ix, c_iy), (c_ix, c_iy + 1), (c_ix + 1, c_iy + 1), (c_ix + 1, c_iy), (c_ix, c_iy), where c_i square in maze
2: function preeliminate_squares(squares)
3:     if squares = ∅ then
4:         return True
5:     end if
6:     square ← random(squares)
7:     choices ← [0, 1, 2, 3]
8:     choices ← shuffle(choices)
9:     if test_for_square(square) = False then
10:        for choice in choices do
11:            if |connection(square[choice])| > 2 then
12:                if |connection(square[choice + 1])| > 2 then
13:                    disconnect(square[choice], square[choice + 1])
14:                    solution ← preeliminate_squares(squares \ square)
15:                    if solution = True then
16:                        return True
17:                    else
18:                        connect(square[choice], square[choice + 1])
19:                    end if
20:                end if
21:            end if
22:        end for
23:        return False
24:    else
25:        return preeliminate_squares(squares \ square)
26:    end if
27: end function
28: preeliminate_squares(squares)
```

able maze, and as the size grows linearly the complexity grows exponentially, therefore a wrong choice in the beginning of the algorithm could lead to a potentially long execution time, while a wrong choice close to the end of the backtracking tree would be corrected quickly, and since the algorithm is random and the maze empty in the beginning, the first choices regarding which walls to raise are decided random. Due to all the aforementioned properties, the final algorithm we decided to implement is a random restarts algorithm. The idea is that since the first choices are decided randomly, stopping the execution, droping the maze, and starting to generate a new one is cheaper than waiting for a wrong choice close to the root of the backtracking tree to be fixed. The algorithm is given a time interval, that gradually increases, in order to attempt and correct the wrong choices that are made. By giving the algorithm this time interval we ensure that the errors which are deep in the branches of the tree are fixed and at the same time execution time is not wasted on errors

that are time consuming to be found and fixed. To sum up, in the beginning we define a starting time interval t and a number of tries n, then the pre-eliminate algorithm starts executing while the execution time is being measured. If the execution time surpasses the time t the execution is restarted with an empty maze. After n restarts the time t becomes $2t$ for another n tries. This continues until a solution is found(see algorithm 10).

The end result again is a braid maze without any squares(see figure 4.6).

Algorithm 10 Random restarts algorithm

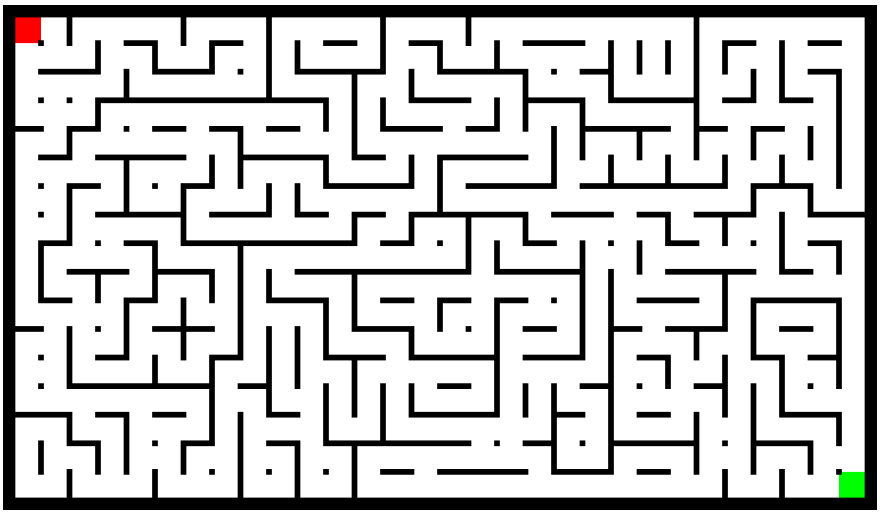
```
1: squares ← ∪_{∀c_i}(c_ix, c_iy), (c_ix, c_iy + 1), (c_ix + 1, c_iy + 1), (c_ix + 1, c_iy), (c_ix, c_iy), where c_i square in maze
2: function preeliminate_squares(squares, t_0, t)
3:     if current_time() − t_0 > t then
4:         return False
5:     end if
6:     if squares = ∅ then
7:         return True
8:     end if
9:     square ← random(squares)
10:    choices ← [0, 1, 2, 3]
11:    choices ← shuffle(choices)
12:    if test_for_square(square) = False then
13:        for choice in choices do
14:            if |connection(square[choice])| > 2 then
15:                if |connection(square[choice + 1])| > 2 then
16:                    disconnect(square[choice], square[choice + 1])
17:                    solution ← preeliminate_squares(squares \ square, t_0, t)
18:                    if solution = True then
19:                        return True
20:                    else
21:                        connect(square[choice], square[choice + 1])
22:                    end if
23:                end if
24:            end if
25:        end for
26:        return False
27:    else
28:        return preeliminate_squares(squares \ square, t_0, t)
29:    end if
30: end function
```

```
31: function random_restarts(squares)
32:     t ← 0.1
33:     n ← 15
34:     while True do
35:         c ← 0
36:         while c < n do
37:             c ← c + 1
38:             t_0 ← current_time()
39:             solution ← preeliminate_squares(squares \ square, t_0, t)
40:             if solution = True then
41:                 return True
42:             end if
43:         end while
44:         t ← 2t
45:     end while
46: end function
47: random_restarts(squares)
```

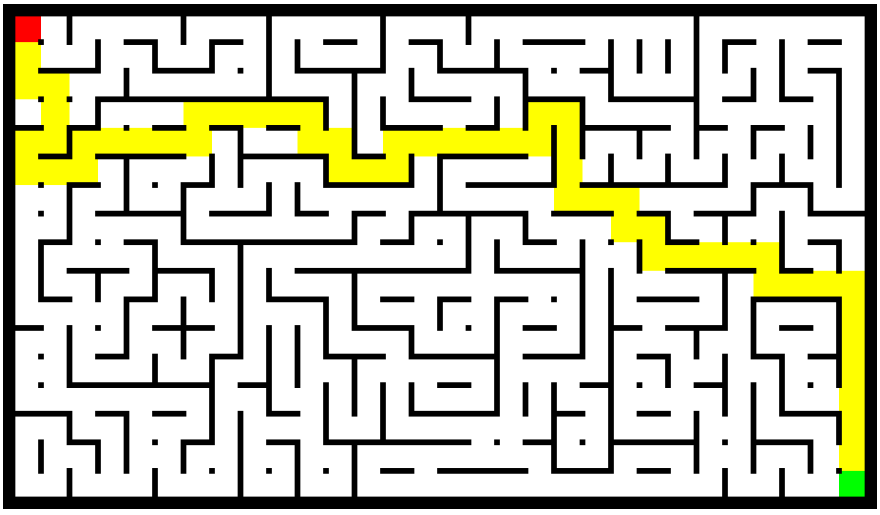
Figure 4.3: Generated braid maze

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
0	3	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	3	2	3	2	2	2	3	3	2	2
1	2	3	2	3	2	2	2	2	3	2	2	2	2	3	2	2	2	2	2	3	2	2	2	2	2	2	3	2	2	3
2	3	3	2	2	2	2	2	3	2	2	2	2	2	3	2	2	3	2	2	2	2	2	2	2	3	2	2	2	2	2
3	2	2	2	2	2	2	3	2	2	2	2	2	2	2	2	3	2	2	2	2	2	2	2	2	2	2	3	2	3	2
4	2	2	2	2	3	2	2	2	2	2	2	3	2	3	3	2	2	2	3	2	2	2	2	3	2	3	2	3	2	2
5	3	3	3	2	2	3	2	2	2	3	2	2	2	2	2	2	2	2	2	3	2	2	2	2	2	2	2	2	2	2
6	3	2	2	2	2	2	2	2	2	2	2	3	2	3	2	2	2	2	3	3	2	3	2	3	2	2	2	2	2	2
7	2	2	3	2	2	2	2	2	2	3	2	2	2	2	2	2	2	2	2	2	3	3	2	2	2	2	3	3	2	2
8	2	2	3	3	2	2	2	2	2	3	2	2	2	2	3	2	2	2	2	2	3	3	2	2	3	2	3	2	2	2
9	2	2	2	2	2	2	2	2	2	2	2	2	2	2	3	2	3	2	2	2	2	2	3	2	2	2	2	2	3	2
10	2	3	2	3	2	2	2	3	3	2	2	2	2	2	3	2	3	2	3	3	2	2	2	2	2	2	2	2	2	2
11	2	2	3	2	3	2	2	2	2	2	2	2	2	2	2	3	2	3	2	2	2	2	3	2	2	3	2	3	2	2
12	2	3	2	2	2	2	2	2	2	2	3	2	2	2	2	2	2	3	3	2	2	3	2	3	2	2	2	3	2	2
13	2	3	3	2	3	2	2	2	2	2	3	2	2	2	2	2	2	2	2	2	2	3	3	3	2	2	2	2	3	3
14	2	2	2	2	3	2	2	2	3	2	3	2	2	2	2	2	3	2	2	2	2	2	2	2	2	3	2	2	2	2
15	2	2	2	2	3	3	2	2	2	2	2	2	2	2	3	2	3	3	2	2	2	2	2	2	3	2	2	2	2	3
16	2	2	2	3	2	2	3	2	2	2	2	2	2	2	3	2	2	2	2	2	3	2	2	2	2	2	2	3	2	

(a) Neighbor matrix

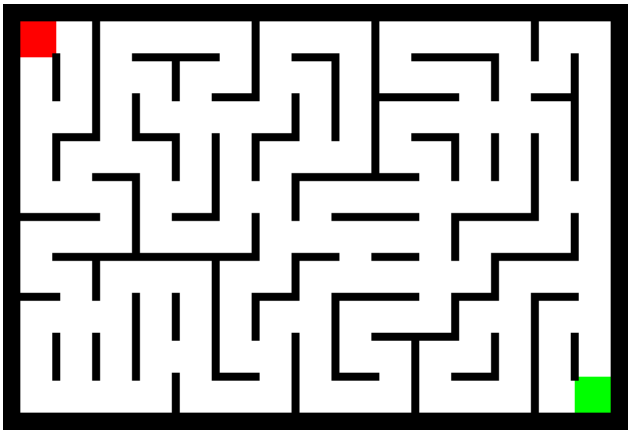


(b) Maze hidden

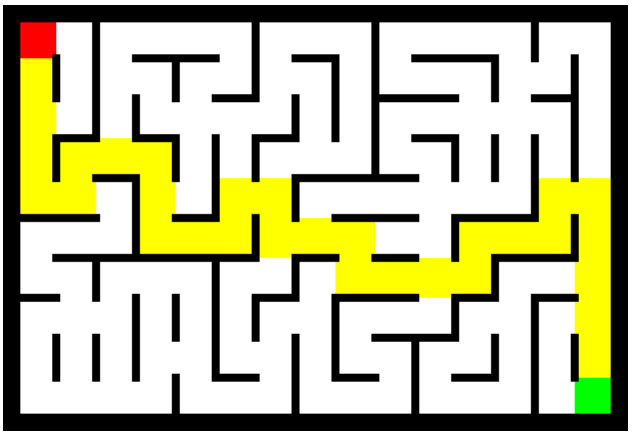


(c) Maze solved

Figure 4.4: Generate and Test maze

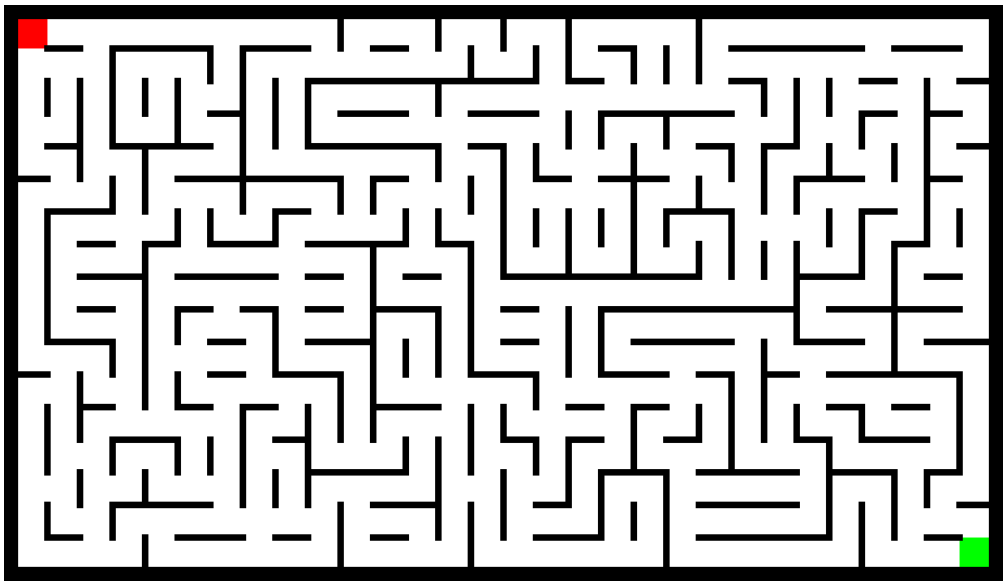


(a) Generate and Test maze hidden

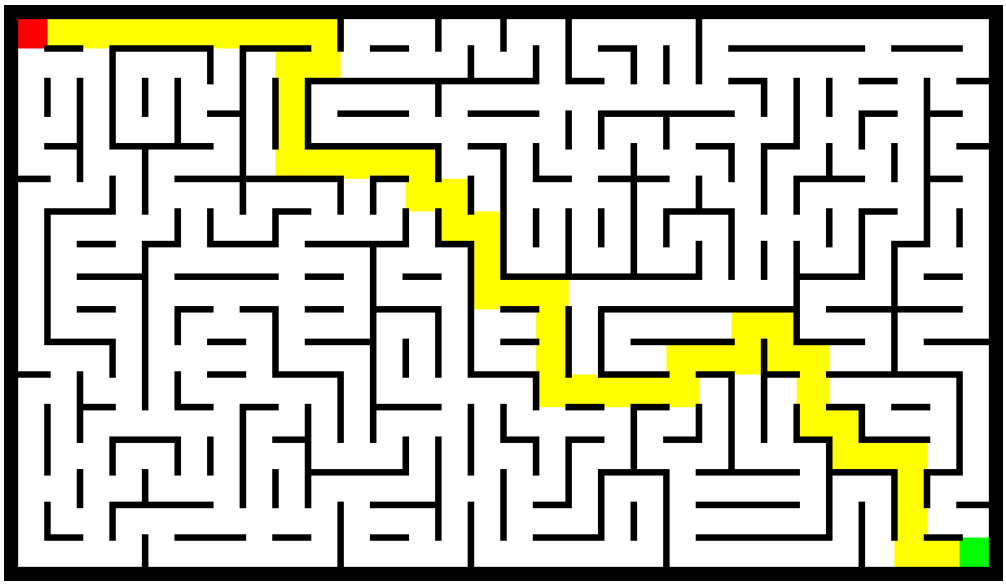


(b) Generate and Test maze solved

Figure 4.5: Pre-Eliminate Squares maze

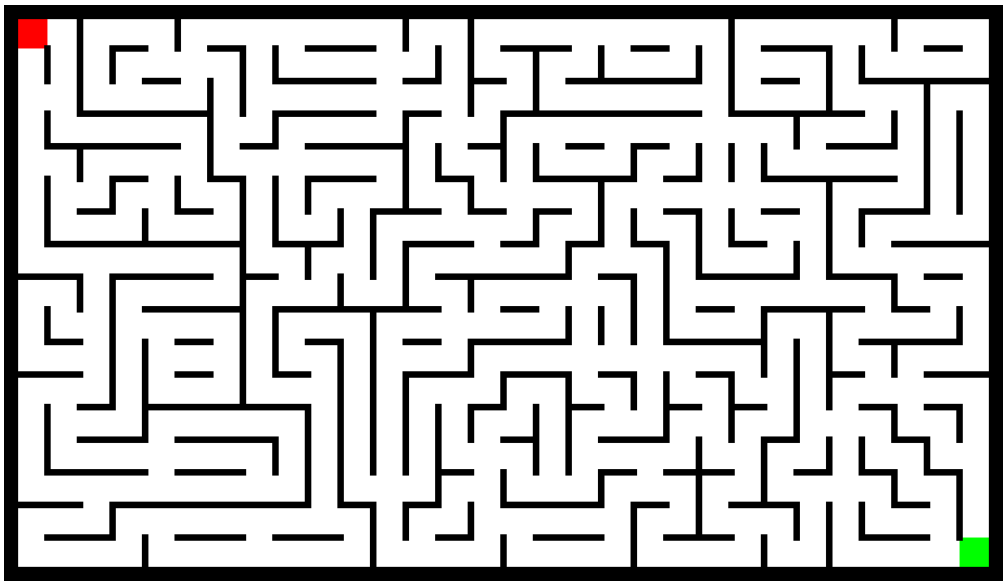


(a) Pre-Eliminate Squares maze hidden

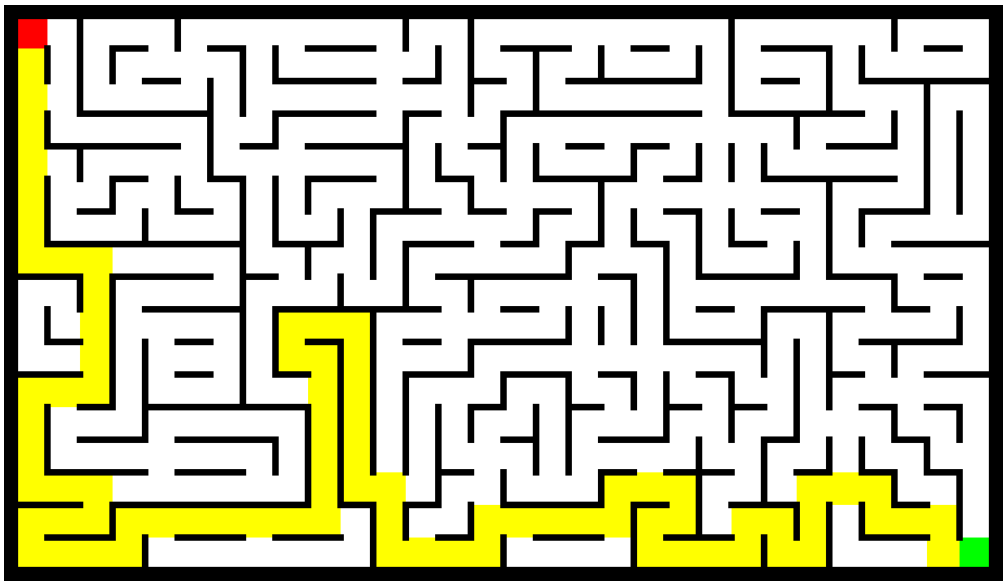


(b) Pre-Eliminate Squares maze solved

Figure 4.6: Random restarts maze



(a) Random restarts maze hidden



(b) Random restarts maze solved

5. SOLVER

In order to solve the maze and visualize the solution 2 classes where implemented:

- the generic parent class `Solver`
- the child class `BFSSolver`

5.1 Class Solver

The `Solver` class stores the maze information along with the solution path which is represented with a `list` that stores tuples the following way:

$$[(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)]$$

where $(x_i, y_i), i \in \{1, 2, \ldots, n\}$ are the coordinates of the ith tile of the solution path.

In addition, the `Solver` class contains the routine `paint_solution(self)`. The routine iterates on the list that contains the solution path and marks and for each tuple (x_i, y_i) in that list the `maze_tile` connected with those coordinates is marked as part of the solution. This is utilized by the visualization algorithm in order to make the path yellow(see figure 5.2).

Finally, the routine `hide_solution(self)` is implemented in order to hide the solution from the user if necessary, by reversing the work of the `paint_solution(self)` routine and for all the tuples (x_i, y_i) in the solution path it marks the corresponding `maze_tile` as not part of the solution(see figure 5.1).

5.2 Class BFSSolver

The `Solver` class does not contain a routine for finding the solution path of a maze, thus it is a generic class used to provide the base for a child class with a solving algorithm to be implemented. The `BFSSolver` class is a child class that inherits from `Solver` the necessary utilities and also contains the `solve` routine which is the implementation of a Breadth-first Search algorithm[4] that works on the mazes that are created. This algorithm was chosen due to its property to calculate the shortest path even on paths that contain loops, i.e. braid mazes(ch. 4). An example can be seen on figure 5.1 where the solution of the maze is not computed yet and on figure 5.2 the solution is discovered.

Figure 5.1: A maze with its solution path hidden to the user

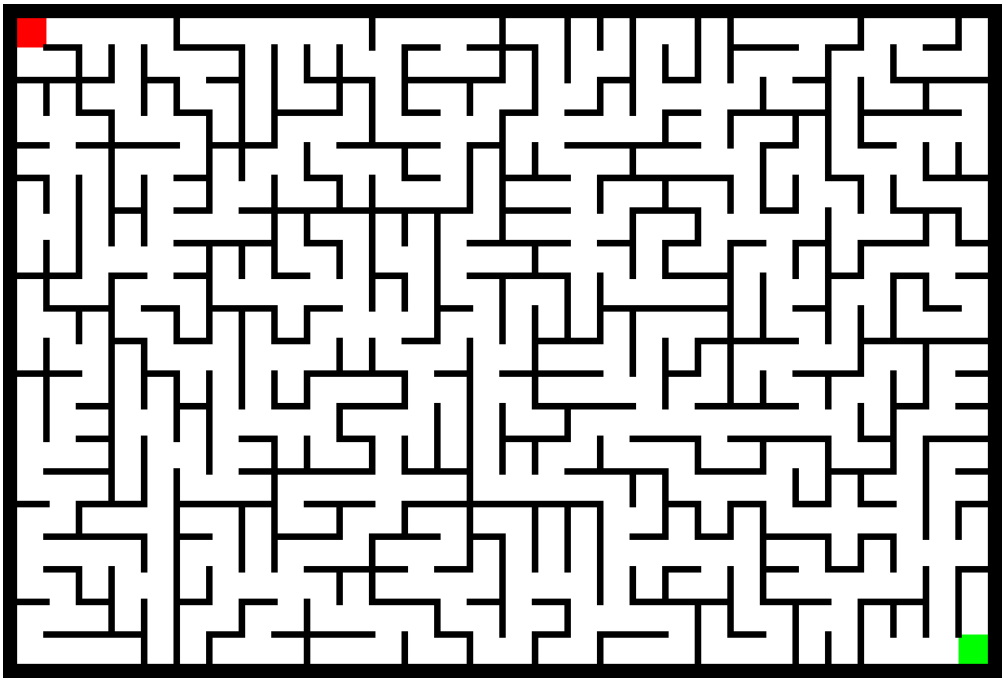
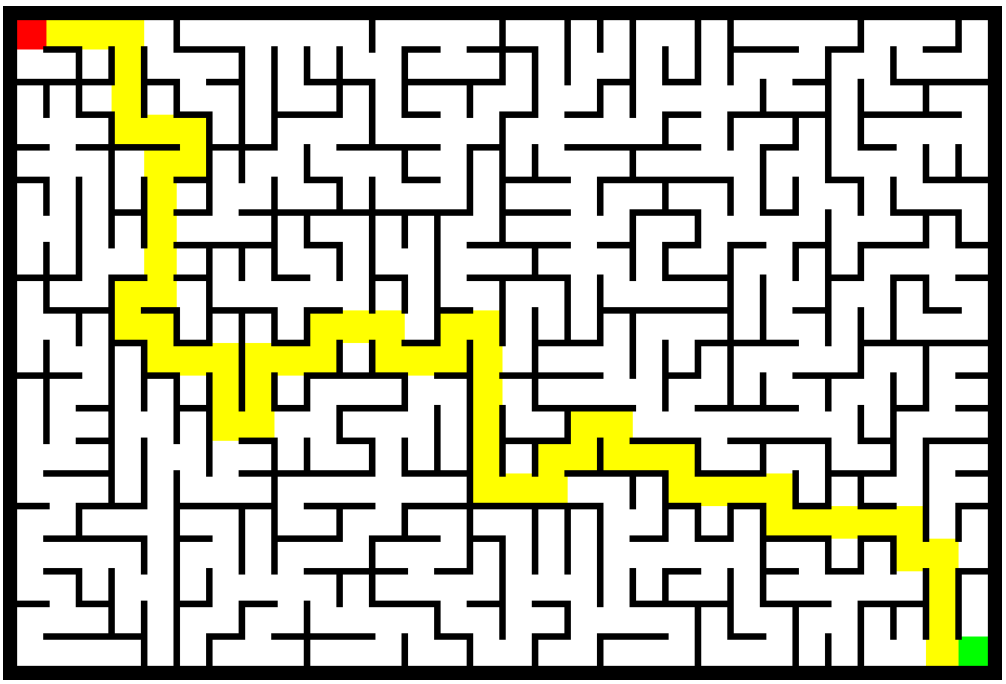


Figure 5.2: Solver has marked the solution on the maze

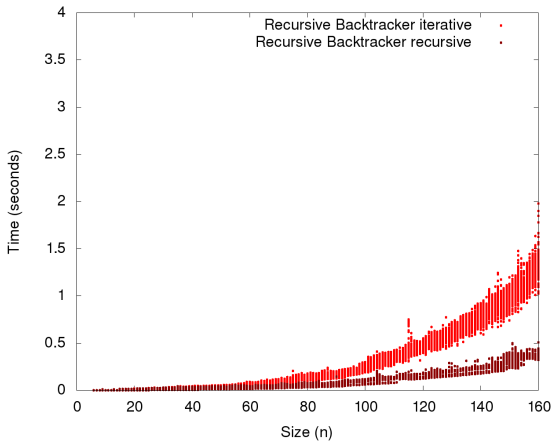


6. BENCHMARKS

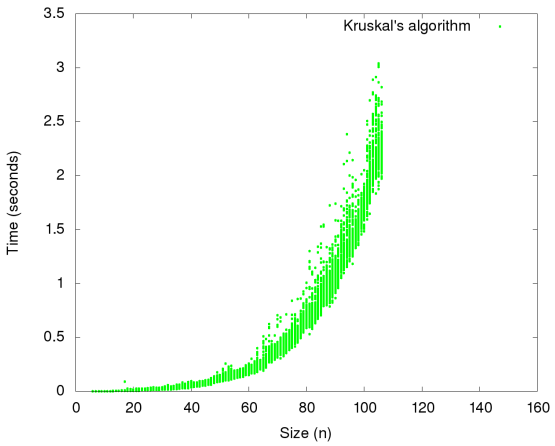
In the following section the benchmarks that were performed on the algorithms are presented. The benchmarks were executed on a machine with the following specifications:

- CPU 8-core Intel Core i7-4710MQ @ 2.50GHz 64-bit
- RAM 8GB
- GPU Nvidia GeForce GT 730M
- OS Linux kernal 3.13.0-45-generic x86_64

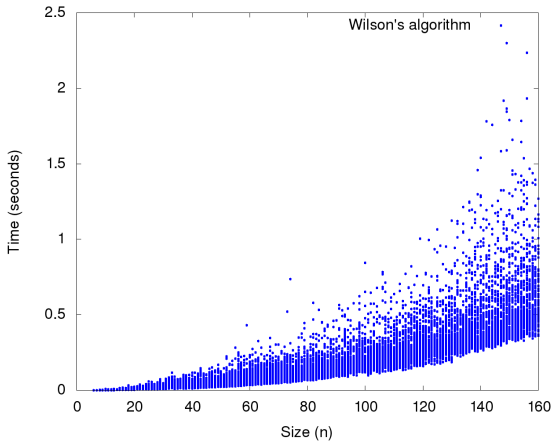
Figure 6.1: Perfect maze benchmarks



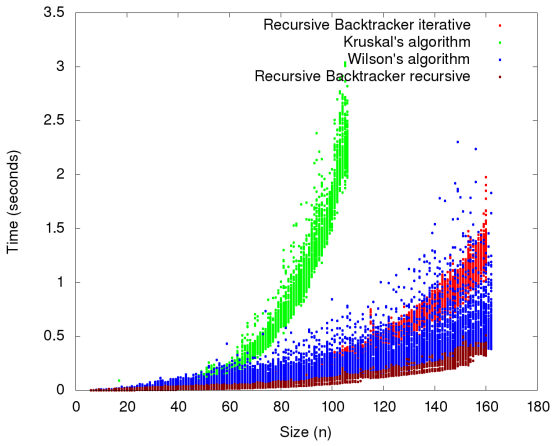
(a) Recursive Backtracker algorithm benchmark



(b) Kruskal's algorithm benchmark



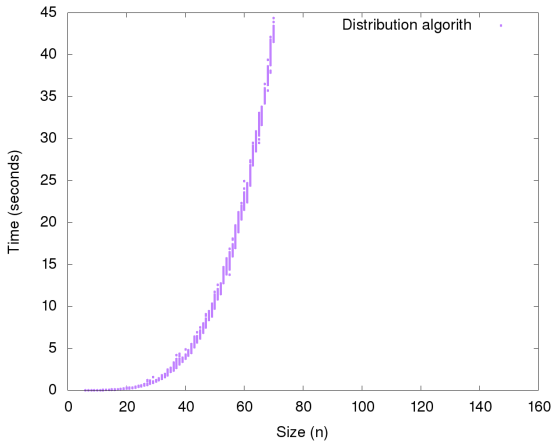
(c) Wilson's algorithm benchmark



(d) Perfect maze algorithms benchmark

On the benchmarks it is clear that the Recursive Backtracker recursive algorithm[algorithm 1] is the most time efficient. Analysing the benchmarks we observe that the Wilson's algorithm[algorithm 4] is statistically the second most efficient algorithm of the 4 presented regarding a generation of a perfect maze(see figure 6.1d) Occasionally, though, Wilson's algorithm performs worse than the Recursive Backtracker iterative algorithm[algorithm 2]. Finally, Kruskal's algorithm[algorithm 3] has the worse performance of the 4.

Figure 6.2: Distribution algorithm benchmark



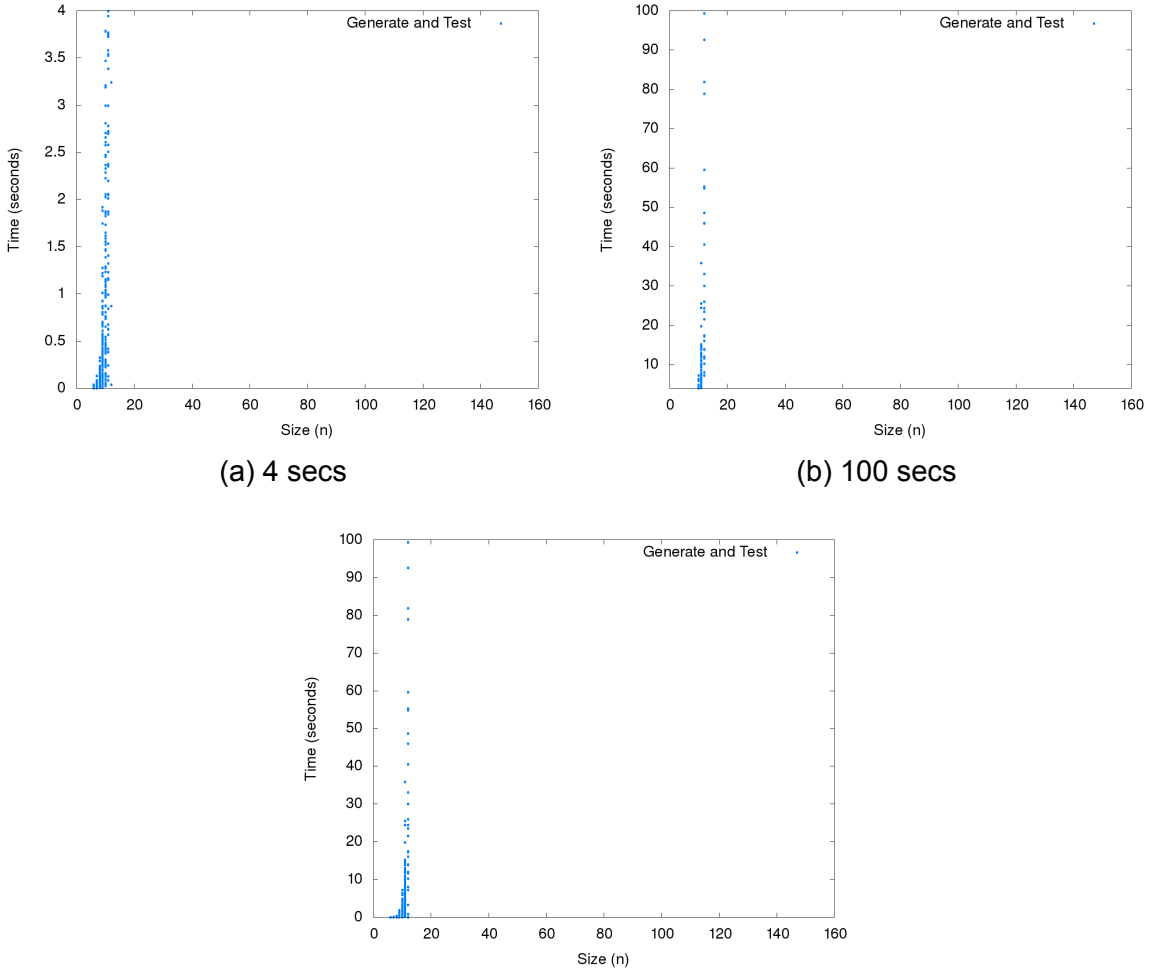
Regarding the braid maze generation we can observe that the distribution algorithm[algorithm 5] becomes a lower bound for the other algorithms since all of them will run the distribution algorithm during their execution(see figure 6.6).

We should mention that for low values of the size n the Random Restarts algorithm[algorithm 10] had a faster execution time than the distribution algorithm. This is expected due to the fact that by eliminating the squares on a maze we reduce the number of walls the rest of the procedure has to create, and for small sizes this affect a large percentage of the maze, while as the size increases the percentage of the maze that the elimination process fills is getting smaller, and the elimination itself becomes more time consuming than the rest of the algorithm.

We also observe the inefficiency of the Generate and Test algorithm[algorithm 8] and the inefficiency and the problematic behavior of the Eliminate Square algorithm[algorithm 9] that we described regarding the effects of a wrong decision. We observe the effects when on graph 6.4 the algorithm makes a jump from 48 seconds to 352 seconds while only on $n = 8$ in both cases, making the algorithm even worse than the Generate and Test algorithm. The effect on adding the Random Restarts method on the algorithm has obvious results, verifying the fact that on the search space of the mazes that are generated, finding a solution has high probability, thus making faster to restart the algorithm than trying to fix all the squares of a problematic maze.

One final observation regarding the nature of the problem we are trying to solve appears on graph 6.7. By analyzing this graph one could derive that generating a perfect maze has a smaller time complexity than a braid maze. All the algorithms presented regarding

Figure 6.3: Generate and Test algorithm benchmark



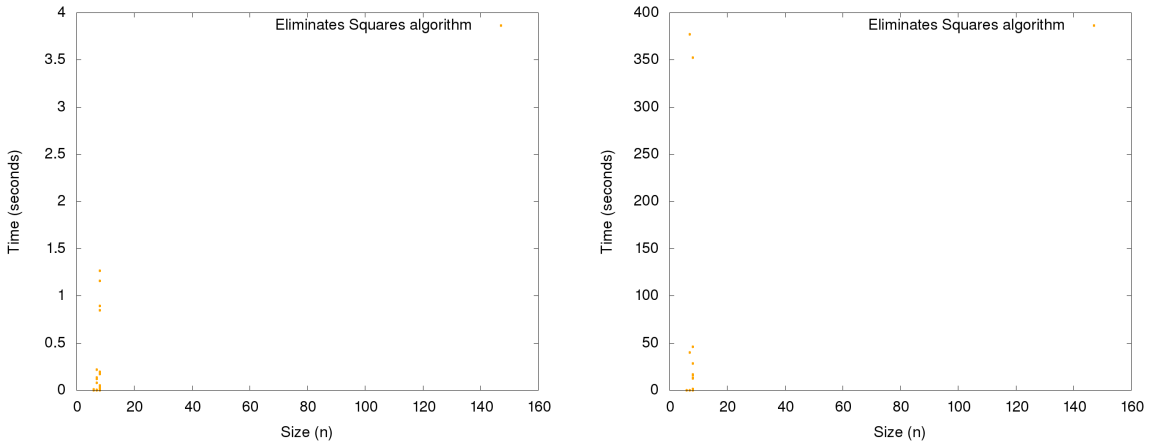
(a) 4 secs

(b) 100 secs

(c) full dataset

perfect mazes have a polynomial complexity, while on braid mazes only the distribution algorithm has a polynomial complexity, with the rest having an exponetial complexity, due to the elimination of squares.

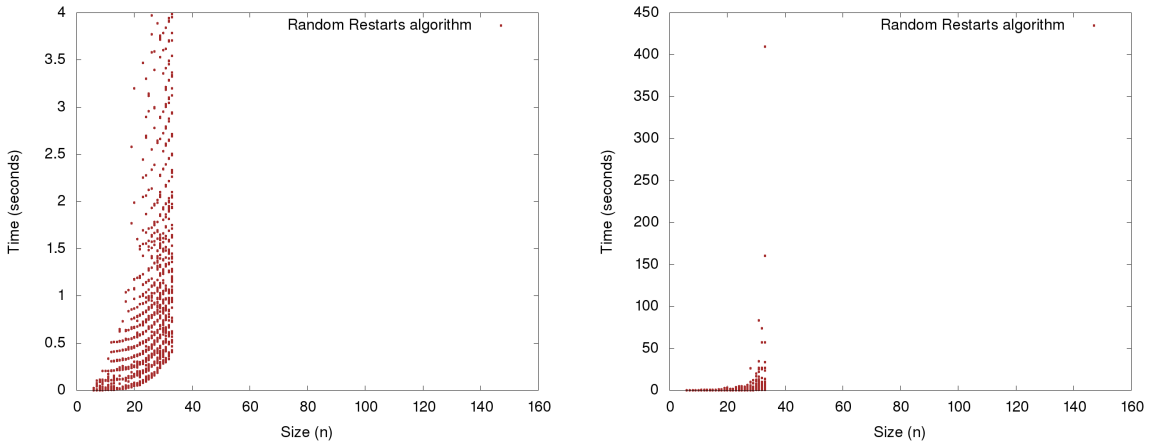
Figure 6.4: Eliminate Squares algorithm benchmark



(a) 4 secs

(b) full dataset

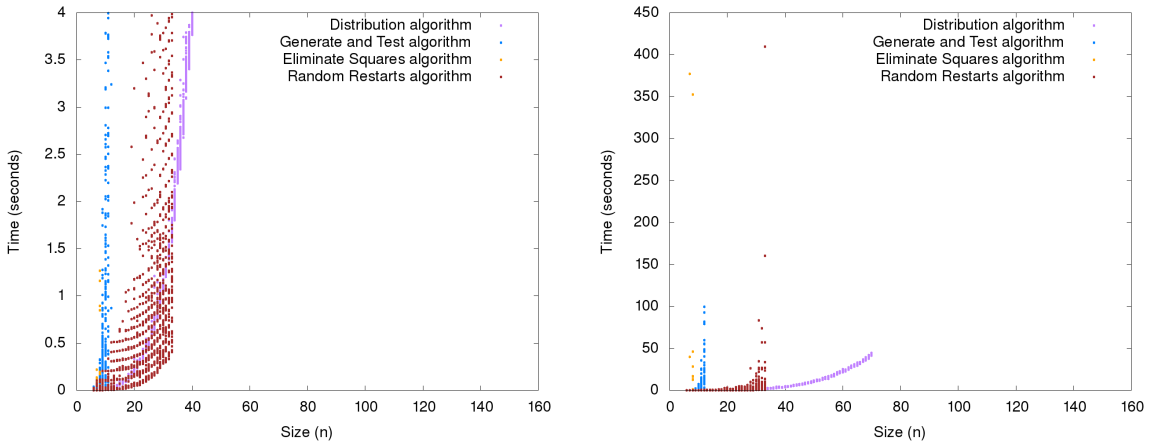
Figure 6.5: Random Restarts algorithm benchmark



(a) 4 secs

(b) full dataset

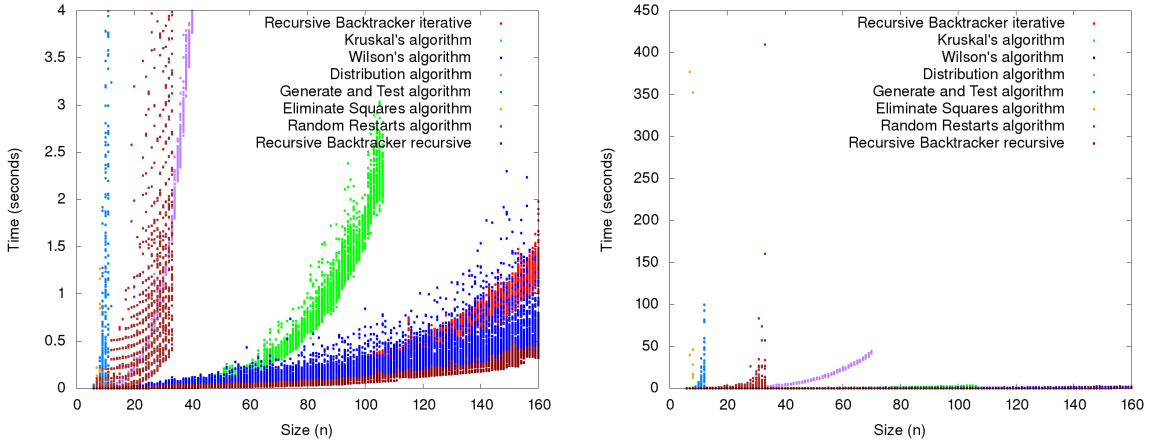
Figure 6.6: Braid generation algorithms benchmark



(a) 4 secs

(b) full dataset

Figure 6.7: Complete algorithms benchmark



(a) 4 secs

(b) full dataset

7. CONCLUSIONS AND FUTURE WORK

The generation of braid mazes has an increased difficulty and the algorithms that were presented on this thesis require a number of improvements in order to approach the speed and efficiency of the algorithms that exist in the bibliography regarding perfect mazes. However, the results of the Random Restarts algorithm show some promise, since using that algorithm we were able to generate mazes of size $n = 33$. Further research could be done regarding the implementation of search heuristic on the Eliminate Squares algorithm, which is also utilizes inside the Random Restarts algorithm, with the goal of reducing the wrong choices made on the backtracking process. In addition, that heuristic could be constructed in a way to affect the bias of the maze where in some cases that would be desirable, since depending on the reason the maze is generated it may be necessary to have a specific bias.

Another aspect of the Random Restarts algorithm is its parameters. For the purpose of this thesis the starting time interval of the restarts was 0.1 seconds and the number of tries before the interval was doubled was 15. Those parameters could be research in order to discover the optimal way of initiating them and raise them to achieve a better running time.

Regarding the way the neighbor matrix[figure 4.3] is created for a braid maze, right now it uses a specific distribution. Experimenting with other types of distribution could alter the final structure of the maze. A possible way would be to make each cell that is calculated to have 3 neighbors to reduce the probability of its neighboring cells to have 3 neighbors also in order to create a bias that would increase the size of a loop in which the user cannot escape. This will possibly increase the difficulty of a maze.

Finally, all those algorithms presented could be expanded on a 3-dimensional space or even on a higher dimensional space. However, a generation of such a maze is more complex, and it is possibly that those algorithms are not efficient enough to handle it, since the check for possible squares would be more time consuming and the choices of which walls to raise are increasing with the increase of the dimension.

REFERENCES

[1] Thomas H Cormen. *Introduction to algorithms*. MIT press, 2009.

[2] Joseph B Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50, 1956.

[3] W. H. Matthews. Mazes and labyrinths: A general account of their history and development. Jul 9, 2014.

[4] Stuart Russell and Peter Norvig. Artificial intelligence: A modern approach. 2003.

[5] David Bruce Wilson. Generating random spanning trees more quickly than the cover time. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 296–303. ACM, 1996.