How to Generate Perfect Mazes?

V. Bellot, M. Cautrès, J-M. Favreau, M. Gonzalez-Thauvin, P. Lafourcade, K. Le Cornec, B. Mosnier, S. Rivière-Wekstein

LIMOS, University Clermont Auvergne, France

Abstract

A *perfect* maze is a maze where any two cells can be joined by a unique path. In the literature, there exist eleven maze generation algorithms as compiled by Buck in 2015 in his book "*Mazes for Programmers*". Each algorithm creates mazes differently. Our aim is to analyze how perfect mazes are generated. For this, we use the simple measures introduced by Buck, as well as the physical based measures introduced by McClendon in 2001. We introduce a new measure that helps us establish a ranking for perfect mazes. We also propose two new maze generation algorithms, called *Prim & Kill* and *Twist & Merge*. According to our measure, these two algorithms generate mazes differently than the existing algorithms do.

1. Introduction

Mazes were first introduced during the Antiquity as aesthetic or voluntarily complex structures. They have been studied from a mathematical point of view – e.g., by the graph theory. A maze is a multi-dimensional area divided by impassable walls that create corridors, dead-ends or crossroads. We study square mazes in which each cell is a square[1], as illustrated in Figure 5. We consider

*Corresponding author

[1]Our work can be generalized to different shapes but this simplifies the presentation of our results.

only *perfect* mazes, i.e., mazes that satisfy the two following constraints:

- all cells are part of a unique connected space,
- no cyclic path is allowed in its construction.

These two constraints are equivalent to the following constraint: for every couple of cells there is one and only one path that connects them. Through centuries, mazes became more and more popular and are nowadays considered as fun puzzles. Solving a perfect maze consists in finding the unique path between two given points. Solving perfect mazes is fun by definition, since each choice of the user is crucial to quickly find the solution in a maze without loop. Choosing a path other than the unique solution will result in following the wrong path and thus wasting time.

There exist eleven well described randomized algorithms [19, 5] to automatically generate perfect mazes. However, these algorithms do not set the entry and the exit. In order not to discriminate any algorithm, we decided to choose these two points as the points that construct the longest path in the maze. Since these pairs of points are not necessarily unique, we randomly take one candidate among all possible choices.

In [2], the author measured the difficulty of a maze by the number of steps to solve it. In [3], the authors defined a fitness function to design the evolution of maze-like levels for use in games. In [10], Kaplan introduced the notion of reconfigurable mazes, where some pieces of a maze can be switched in order to form a new maze. This notion and its applications to games design were more studied in [8]. In [12], the authors proposed the user to generate mazes for games where they satisfy some properties based on the game mechanics specificities.

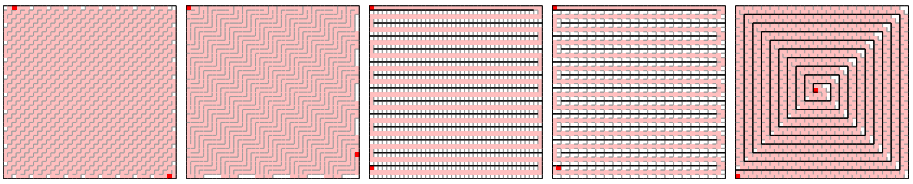
In this approach it is possible to control the topology of the solution path of a maze.

Surprisingly, we found only few works [5, 16, 7, 13] that propose metrics on mazes.

In [5], Buck explained the eleven maze algorithms presented in [19] and used them to teach programming. He also proposed the following metrics (in Appendix 2 of [5]): number of dead-ends, length of the longest path (the number of cells it contains), number of cells with horizontal or vertical passages (east-to-west, and north-to-south), number of "elbow" cells (passage entering from one side, then turning either right or left), number of three-way intersections and number of four-way intersections. He also wrote "*the statistics presented in this appendix are very superficial. There is a lot more that could be done*". Our goal in this article will therefore be to propose a metric to compare mazes generators.

In [16], McClendon proposed two measures to qualify a maze: a *complexity* and a *difficulty* measure. We show in section 3.3 that the complexity is almost equivalent to the length of the solution, which is not a satisfactory measure if one considers the pleasure of solving mazes: a perfect maze with no intersection has a huge solution length, but is boring and not fun to solve because the solution is its unique path. The player has no choice but to follow it. We give such examples in Figure 1a, 1b, 1c, 1d and 1e.

One of our aims is to design a measure that can determine if a maze generator algorithm produces boring mazes or fun mazes. Moreover, we have identified mazes where McClendon's difficulty is not relevant to evaluate the fun of perfect maze generation algorithms. In Figure 2, the maze on the right is clearly more



(a) Stairs (b) Big stairs (c) Parking (d) Parking 2 (e) Intestin

Figure 1: Examples of 40×40 manually generated mazes. The two marked cells are drawn in red, the solution path in light red. Non significant walls are drawn in gray.

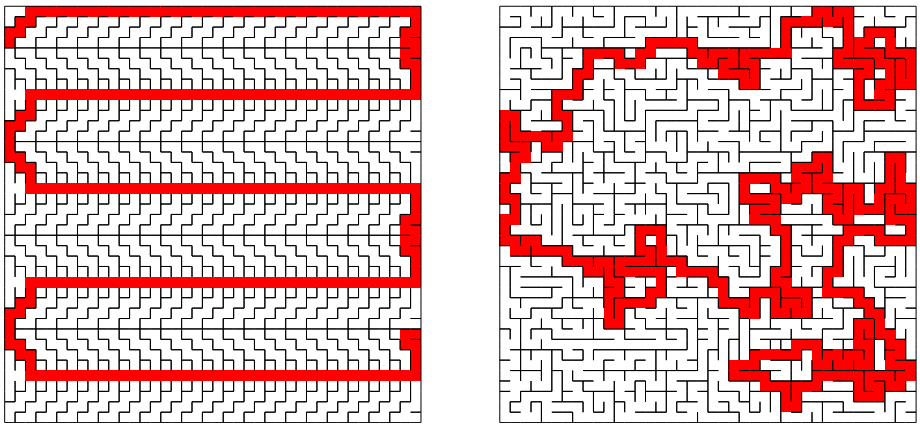


Figure 2: On the left side an ad-hoc maze of solution length 228, complexity: 3.9 difficulty: 222.6 and on the right side a maze obtained with the *Hunt & Kill* algorithm of solution length 324, complexity: 4.6 and difficulty: 48.5.

fun to solve than the one on the left. However, the difficulty of the right maze is 39.7 while the difficulty of the left maze is 222.6.

In [7], Gabrovšek proposed six of the maze algorithms presented into [19], and uses four solving algorithms defined as agents walking across the mazes: random walk, depth first search (with or without heuristics), and breadth first search. The average number of steps is then used to evaluate the difficulty of the mazes generated by each algorithm, for each agent. These measures are based on the length of the solution, as we will see later that our measure is complementary to such approaches.

In [13], the authors compared three techniques for maze generation Depth-first search, Prim's, and Recursive Definition.

Contributions: Here are our main contributions:

- Our first contribution is a measure that captures the fun experience when solving a maze. Our measure counts the number of *non-significant walls* in a maze, to estimate which walls of the maze are not considered by a human scan. Combining this measure with McClendon's difficulty makes a realistic evaluation of the fun.

- Our second contribution is the design of two original maze generation algorithms, called *Prim & Kill* (PK) and *Twist & Merge* (TM). The first algorithm (PK) is the merge of two existing algorithms: *Prim* and *Hunt & Kill*. The second algorithm (TM) uses random walks, favorizes turns rather than corridors, and merges all generated paths in order to build a perfect maze. These algorithms offer different approach than the eleven existing generators. Using our fun measure, we show that TM is the funniest algorithm and PK is tied for second with the *Hunt & Kill* algorithm.

Related works: The "*Think Labyrinth!*" [19] website by Pullen is one of the richest online sources of information about mazes where each algorithm is presented. In [5], Buck used eleven perfect mazes algorithms to teach programming.

In his thesis [6], Foltin considered maze generation and human interaction. He studied only 4 maze generation algorithms (*Hunt & Kill*, *Prim*'s, *Kruskal*'s and the *Recursive Backtracking* algorithms) and do not provided a way to classify the algorithms.

Kim and Crawfis's article [11] is one of the most relevant resources concerning maze evaluation. The authors constructed a large database made of about 22 million mazes generated by *Prim*'s and the *Recursive Backtracking* algorithms.

They proposed similar metrics as Buck [5] and a generic evaluation function to identify the maze that satisfies metrics selected by the user, e.g., the maximum number of dead-ends. They only study two maze generation algorithms and they let the user determines which metric is the most adapted. Our aim is to establish a ranking to evaluate the fun of solving a maze.

In [16], McClendon defined two measures to evaluate mazes, considering any kind of geometry for the mazes, even hand-drawn ones. These measures are computable regardless of the path traced to solve the maze, and the author indicated that the measurements associated with the maze itself must be calculated by keeping the minimum on all possible paths. In Section 6, we translate this approach to our maze framework, and we use one of these two measures to design our fun measure.

Other studies about maze have been made, but they clearly have different goals. In [17] Okamoto and Uehara explained how to construct a *picturesque maze* - i.e., a maze whose solution path draws a picture. They also considered perfect mazes and their algorithm is based on an adaptation of the *Spanning Tree* generation algorithm.

Turan and Aydin showed a dynamic terrain-spaced maze generation algorithm [20]. They generated mazes that have big empty areas and cycles. They are used for maps in video games, but they do not generate perfect mazes.

A perfect maze based steganographic method is presented by Lee, Lee and Chen [15]. They explain that information can be enciphered in a perfect maze using the direction of bifurcations in the solution path. The algorithm used to generate perfect mazes is an adaptation of *Prim*'s algorithm, one of the eleven algorithms that we consider.

Applications: Our main contributions is to design two new maze generators. They increase the collection of existing algorithms. We believe that it is important to have a diversity in the maze generator algorithms. Moreover, there are two main applications for having several maze generator algorithms.

- First, pen and pencil games editors constitue a possible target in order to create a "Mazes magazine". The idea is to produce each month several mazes with different difficulty levels for kids. For this promising project, our algorithms are good candidates to design original puzzles for kids or adults. We beleive that such magazine can have an audience as the recent succes story of coloring magazines for adults that are relaxing and aim at deacreasing the level of stress.

- The second main application domain is clearly the video games. Many applications already exist on smart phones to solve mazes. However all applications, that we have found and tested, have two main limitations:

    1. Some applications are using a limited number of mazes that are hard coded in the application. This clearly only offers a limited number of levels for the gamers.

    2. Many other applications always use the same maze generator, such as *Prim* or *Recursive Backtracking*. According to the difficulty level and the experience of the gamer, a greater variety of algorithms can offer a better player experience and increase the fun of such applications. Our algorithms clearly increase the diversity of fun maze generators for such applications.

Outline: In Section 2, we give some definitions concerning perfect mazes. In Section 3, we explain the measures introduced by Buck and McClendon. We also introduce our measure that consider the number of non-significant walls. In Section 4, we present the eleven existing maze generation algorithms. In Section 5, we present our two maze generation algorithms: *Prim & Kill* and *Twist & Merge*. In Section 6, we compare all the 13 algorithms according to our measure. Finally, we conclude in Section 7.

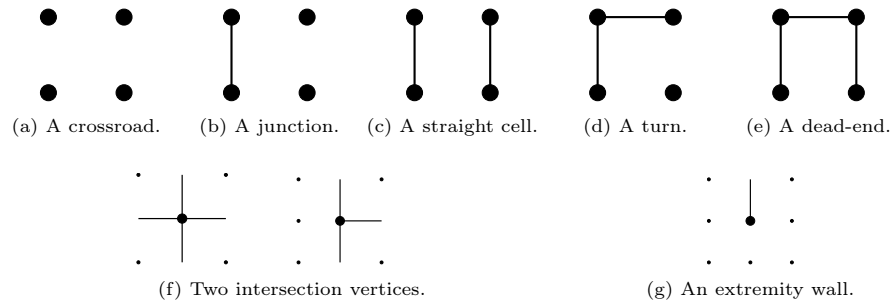
2. Notations and Definitions about Perfect Mazes

For simplicity's sake, we only consider square perfect mazes where all border walls are indestructible. Hence, a maze $\mathcal{M}$ is defined on a subset of $\mathbb{N} \times \mathbb{N}$ called a *board*, typically a rectangular region composed of cells. A cell $c[i, j] \in \mathcal{M}$ represents the cell at the row i and at the column j of the board. When the context is clear, we just denote a cell by c.

We define $\mathcal{N}$ as a subset of $\mathcal{M} \times \mathcal{M}$ corresponding to the 4-neighbor adjacency as follows: $(c[i, j], c'[i', j']) \in \mathcal{N} \Leftrightarrow (i = i' \wedge |j - j'| = 1) \vee (j = j' \wedge |i - i'| = 1)$. A maze is defined by a function $w : \mathcal{N} \to \{0, 1\}$ that describes the wall positions. In the following, a wall is said *active* (resp. *inactive*) between c and c' if $w(c, c') = 1$ (resp. $w(c, c') = 0$). To define a maze with this function, we need the following properties:

- $\forall (c, c') \in \mathcal{N}, w(c, c') = w(c', c)$ (symmetry)
- $\forall (c, c') \notin \mathcal{N}, w(c, c') = 0$ (no wall outside the maze)

The graph defined by the cells of a perfect maze $\mathcal{M}$ and the set of pairs such that $w(c, c') = 0$ is a tree: an acyclic undirected graph with a single connected component. Hence we have $\forall c \in \mathcal{M}, 0 \leq \sum_{c_i \in N(c)} w(c, c_i) \leq 3$, where $N(c)$ is



(a) A crossroad. (b) A junction. (c) A straight cell. (d) A turn. (e) A dead-end.

(f) Two intersection vertices. (g) An extremity wall.

Figure 3: Examples of the cell classification modulo symmetries or rotations and of the wall classification.

the set of adjacent cells to $c \in \mathcal{M}$. We can categorize a cell of a maze according its number of walls:

- A *crossroad cell* is a cell surrounded by zero wall (Figure 3(a)).
- A *junction cell* is a cell surrounded by only one wall (Figure 3(b)).
- A *decision cell* corresponds to a crossroad or a junction (Figure 3(a) and 3(b)).
- A *dead-end* is a cell with three surrounding walls (Figure 3(e)).
- A *straight cell* is a cell surrounded by two opposite walls (Figure 3(c)).
- A *turn* is a cell with two consecutive surrounding walls (Figure 3(d)).
- An *interection wall* is a wall that is connected to other walls (Figure 3(f)).
- An *extremity wall* is a wall that is not connected to another one on one side (Figure 3(g)).

Furthermore, we define other elements that exist in a maze:

- A *path* is a sequence $c_0, ..., c_i \in \mathcal{M}$ of distinct adjacent cells such that $\forall j \in [\![0, i-1]\!] : w(c_j, c_{j+1}) = 0$. We call *length* of a path the number of cells of the path.

- A *longest path* in a maze is a path that has the maximal length[2].
- A *corridor* is a path in which each cell is surrounded by two walls (turn or straight cell). Both cells at the extremities of the corridor must be decision cells.
- A *dead-end path* is a variation of a corridor. The only difference is that at least one extremity is a dead-end instead of a decision cell.

In Section 4, we present the eleven maze generation algorithms described in detail in [19]. All these algorithms are perfect randomized maze generation algorithms. In our implementations, all the random choices have been computed using an equi-probable pseudo-random generator. In Section 4 gives the detailed descriptions of each of these algorithms and simple examples. In the rest of the article, we use the following notations for the name of the algorithms *Aldous Broder*: AB, *Binary Tree*: BT, *Eller*: E, *Growing Tree*: GT, *Hunt & Kill*: HK, *Kruskal*: K, *Prim*: P, *Prim & Kill*: PK, *Recursive Backtracking*: RB, *Recursive Division*: RD, *Sidewinder*: S, *Twist & Merge*: TM, *Wilson*: W. Each of them produce on a board $\mathcal{M}$ a perfect maze in a randomized processing, using two elementary operations: *destroy a wall* $(c, c') \in \mathcal{N}$ (set $w(c, c') = 0$) and *activate a wall* $(c, c') \in \mathcal{N}$ (set $w(c, c') = 1$).

Most of these algorithms start their process with a maze where all the walls are active. Only one algorithm (*Recursive Division*) starts with the empty configuration, where $w(c, c') = 0, \forall (c, c') \in \mathcal{N}$. In any cases, the initial maze is surrounded by walls that will not be destroyed: $\forall c \notin \mathcal{M}, \forall c' \in \mathcal{M}, (c, c') \in \mathcal{N} \Rightarrow w(c, c') = 1$.

[2]Note here that we cannot ensure its uniqueness.

3. Ranking

Before presenting our measure of fun, we recall the definitions given by Buck and McClendon.

3.1. Simple Buck's Measures

The easiest way to compare mazes is to use simple statistics. In [5], Buck proposes six intrinsic attributes of mazes: number of turns, number of straight cells, number of junctions, number of crossroads, number of dead-ends, and length of the solution path. This quick but useful overview helps to understand what is at stake in maze generation.

3.2. McClendon's Complexity and Difficulty Measures

In his work [16], McClendon proposes a measure based on the trajectory equivalent to the one followed by the eyes of a player trying to solve the maze. Long and straight corridors are easier and faster to analyze than twisted paths. This velocity of vision is fully taken into account by McClendon's measures.

The definitions given by McClendon[3] can be adapted to rectangular mazes as following.

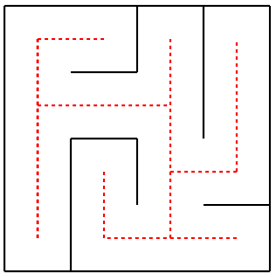
As McClendon says "*Let the maze M have a solution T. If T has small complexity, but also has many branches of large complexity then the complexity would be large, but the maze would be easy to solve*". As we show latter the complexity is related to the lenght of the of maze. Moreover he continues saying that "*If instead of adding the terms in (3)*[4] *above, we were to multiply them we would arrive at a measure that seems to better describe the difficulty of a*

[3] A pedagogical exemple is proposed by McClendon on his website `http://www.math.uco.edu/mcclendon/complexityrecmazes.pdf`

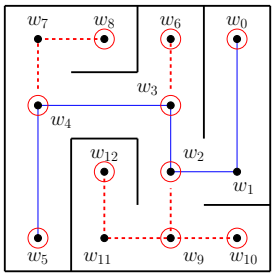
[4] (3) is the formula that define the complexity.

maze than the complexity measure". In our paper, we choose to use the same definition to measure the difficulty of solving a maze, even if it is not strongly motivated by a scientific argument.

The solution path is first extracted, and considered as a dead-end path. Each branch in *hallways* is then split (which is either a corridor or a dead-end path). In each of these parts (including the solution path) turns are identified, and arcs are defined as straight lines between two consecutive turns or between an extremity and its adjacent turn. The graph of a maze is the union of these arcs. This process guarantees uniqueness of the graph for rectangular mazes. This McClendon's measure represents the vision trajectory. This generic definition is a simplified version of McClendon's one for all kind of mazes. Like McClendon, we assimilate a maze and its graph.



(a) A maze and its graph (dotted).



(b) Solution path (plain) and branches (dotted). The red circles are hallway delimiters and the black points are arc delimiters.

Figure 4: McClendon's notations and definitions.

The first part of the measure consists in extracting from a maze M the solution path T (plain line on Figure 4(b)) and batching connected components called *branches* (dots lines on Figure 4(b)) in the set $\mathcal{B} = (B_i)_{i \in [\![0,b]\!]}$ where b is the number of branches in the maze.

We can also define $\mathcal{W} = \{w_j\}_{j \in [\![0,k-1]\!]}$ as the k turns and extremities of this graph, represented by black dots in Figure 4(b)). One can notice that a

turn can belongs to several branches or corridors, such as w_2, w_3, w_4 and w_9 (Figure 4(b)). Each branch can be split into *hallways*, corresponding to a set of connected corridors whose ends are deadends, or turns that belongs to two or more corridors. Let $\mathcal{W}_h$ be the ordered list of turns contained in the hallway h.

We also define W_h as the number of turns in a hallway h. The exact and original definition of a turn does not really matter since we are studying rectangular mazes, in which a turn is self-explanatory. Following the same approach, the length $D(h)$ of a hallway h is defined as the sum of the length of the contained arcs: $\sum_{k=2}^{W_h} d_{h,k}$, where $d_{h,k}$ is the length of the k-st arc of the hallway h.

Let $\theta_{h,k}$ be the angle of the k-st turn in the hallway h. We can then translate into our context McCLendon's complexity $\gamma(M)$ and difficulty $\delta(M)$ of a maze M as following:

- The complexity of a hallway h is defined as $\gamma(h) = D(h) \sum_{k=2}^{W_h-1} \frac{\theta_{h,k}}{d_{c,k} \cdot \pi}$. As all our graphs are orthogonal, the angles $\theta_{h,k}$ are exactly equal to $\frac{\pi}{2}$ so we can simplify McClendon's formula for the complexity of a corridor as follows: $\gamma(h) = D(h) \sum_{k=2}^{W_h-1} \frac{1}{2d_{c,k}}$. The complexity of a branch is defined as the sum of the complexity of each contained hallway.

- The complexity and the difficulty of a maze are defined by: $\gamma(M) = \log\left(\sum_{i=0}^{b} \gamma(B_i)\right)$ and $\delta(M) = \log\left(\gamma(B_0) \prod_{i=1}^{b} (\gamma(B_i) + 1)\right)$.

Hence the complexity is equal to the difficulty for all mazes without branches (cf. *Stairs* algorithm in Figure 1a) or with only straight branches (cf. *Parking* algorithm in Figure 1c).

3.3. McClendon's Complexity is almost Solution Length and is not Fun

The time to draw the solution path can be assimilated to the length of the solution path. In order to compare this measure to the complexity of McClendon, we generated 20 mazes with each algorithm identified in the literature (see Section 4) and studied the measures calculated on each of them. We have thus noticed that the length of the solution path and the complexity introduced by McClendon are strongly correlated, as illustrated in Figure 6b.

Finally, this measure cannot be considered as relevant to evaluate the fun, since very long solution mazes have only short dead-end paths, which greatly reduces hesitation when solving the mazes. The player then follows an almost obvious path, which is not fun, *e.g.* with mazes generated by *Recursive Backtracking* (see Section 4.6).

To overcome these limitations, we introduce in the next section an alternative measure to evaluate the fun of a maze.

3.4. Our Measure: Number of Non-significant Walls

We studied a large number of criteria to design a ranking of fun, including counting the number of decision cells, the number of turns, etc. None of them was conclusive. We finally looked at the time needed to scan the no-solution branches, which corresponds to the fun of a maze. We evaluate this time by counting the number of non-significant walls (NSW), i.e., the walls one can ignore while scanning the board. The more non-significant walls a maze contains, the faster the maze is scanned.

In order to define non-significant walls, we define a *vertex* as the middle point v between four adjacent cells. Let $N(v)$ be the adjacent cells of a given vertex v. The *arity* of a vertex is defined as the number of adjacent walls: $a(v) =$

Table 1: Mean number of non-significant walls (NSW), Difficulty (D) and ratio (NSW/D) for each algorithm, computed from 1.000 randomly generated 40×40 mazes per algorithm.

Algos	GT	RD	E	P	K	AB	W	BT	S	HK	PK	TM	RB
NSW	1521.0	1090.0	**719.0**	868.6	844.5	823.5	822.2	817.4	808.2	723.1	742.9	725.6	635.3
D	3.13	12.72	49.29	25.13	33.87	34.88	35.18	25.57	27.86	43.90	47.82	**57.18**	31.74
NSW/D	485.0	89.0	14.91	35.22	25.41	24.07	23.87	32.42	29.41	16.78	15.90	**12.92**	20.33

$\sum_{(c,c')\in\mathcal{N};c,c'\in N(v)} w(c,c')$. A vertex is said to be an *intersection* if $a(v) \geq 3$ (Figure 3(f)). A wall $w(c,c')$ is said to be an *extremity wall* if $\exists v : c,c' \in N(v)$ and $a(v) = 1$ (Figure 3(g)).

A wall $w(c,c')$ is said to be *non-significant* if it is removed during the non-significant wall deletion. The non-significant wall deletion starts by marking all the intersection vertices. An iterative process is then applied, by deleting at each step all the extremity walls, except if their extremity vertex has been marked as an initial intersection vertex.

In Table 1, we give the mean number of non-significant walls and difficulty for 1.000 generated 40×40 mazes with different algorithms.

4. Existing Generation Algorithms

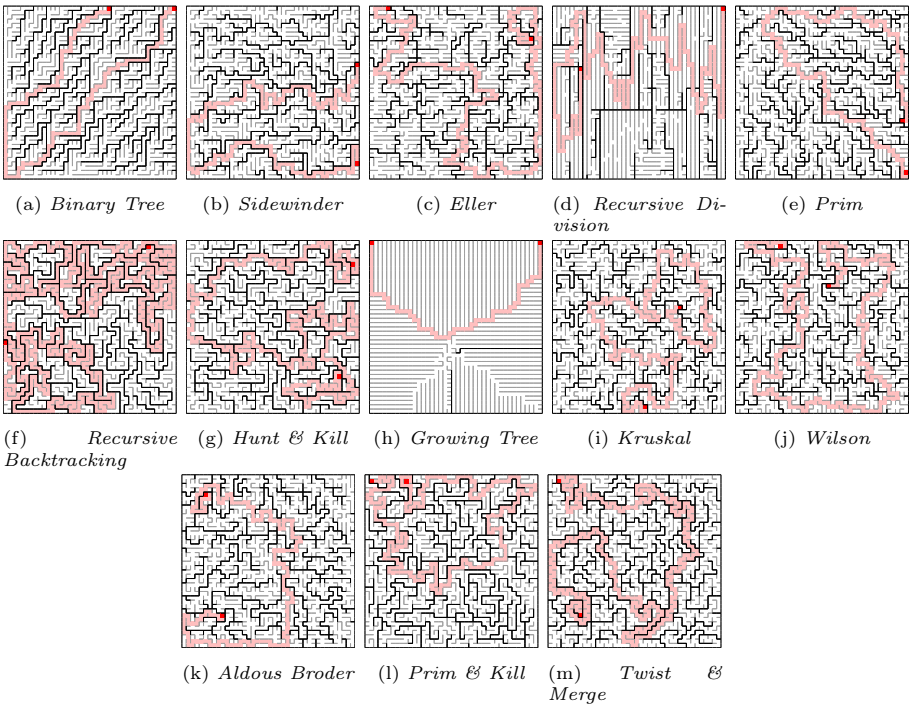
In this section, we present using our notations and in a uniform way eleven randomized maze generation algorithms identified in the literature[5]. For each algorithm, we give its description and a rendering (Figure 5).

4.1. Binary Tree*'s Algorithm*

It is by far the simplest perfect maze generation algorithm. It exists in four versions, according to the chosen orientation. We describe the bottom-right version.

For each cell in the grid, the algorithm destroys either the bottom or the right wall, using an equiprobable random selection. If the wall selected cannot

[5]More details can be found in [5] in order to see how it is possible to implement them.



(a) *Binary Tree* (b) *Sidewinder* (c) *Eller* (d) *Recursive Division* (e) *Prim*

(f) *Recursive Backtracking* (g) *Hunt & Kill* (h) *Growing Tree* (i) *Kruskal* (j) *Wilson*

(k) *Aldous Broder* (l) *Prim & Kill* (m) *Twist & Merge*

Figure 5: Examples of 40×40 mazes generated by each of the algorithms. We give the rendering of one generated maze. The two marked cells are drawn in red, the solution path in light red. Non significant walls are drawn in gray.

be destroyed (borders of the maze), it destroys the other wall. If both walls cannot be destroyed (borders of the maze), the algorithm does nothing to the cell.

The maze generated by this algorithm has a particular shape: there is a general orientation of the corridors, diagonally in function of the selected version. For instance, with the bottom-right version, the top line and the left column of the grid are two long corridors, due to the "other wall" condition.

Figure 5(a) shows a rendering maze, its solution, and the contained non-significant walls.

4.2. Sidewinder*'s Algorithm*

At the initialization, all the walls on the grid are active except those in the vertical direction, on the top line.

For each line, beginning at the second one, the algorithm processes from the left cell to the right cell following those instructions:

- Mark the current cell.
- If the right wall is not a border, randomly decide if it is destroyed or not, using an equiprobable random choice.
- If the wall is destroyed, step one cell right, otherwise destroy the top wall of a random cell among marked ones, and then unmark all cells and step one cell right.

With such an algorithm, it is impossible to have a maze with top-oriented dead-ends. The maze generated has also a general vertical orientation.

Figure 5(b) shows a rendering maze, its solution, and the contained non-significant walls.

4.3. Eller*'s Algorithm*

Like the *Sidewinder* or the *Binary Tree* algorithms, mazes are processed line by line. In this algorithm, a flag $f(c) \in \mathbb{N}$ is associated to each cell c, as a connected component descriptor: if (c, c') two cells are flagged with the same label, it exists a path between these two cells. At the initialization, all the walls are active, thus each cell has its own flag. After each wall deletion, flags are updated according to the connected components in the maze, by updating all the flags of one side.

The algorithm starts with the first line. On the current line, a random selection of walls $w(c, c')$ are destroyed where $f(c) \neq f(c')$.

The next step consists in destroying a random selection of bottom walls of the current line (at least one for each group of cells with the same flag). The following line becomes the current one.

These instructions are repeated until the last line is reached. For the last line, if two adjacent cells (c, c') have a different flag value, the wall between them is destroyed.

Figure 5(c) shows a rendering maze, its solution, and the contained non-significant walls.

4.4. Recursive Division*'s Algorithm*

It is a recursive algorithm and the only *wall-adder* algorithm. At the initialization, all the walls are inactive.

The grid is cut in two parts with a straight wall (either horizontally or vertically) from one border to the opposite. We destroy a random wall on this line in order to make a passage between the two areas.

This operation is recursively repeated on both areas delimited by the wall, until we reach an area with a height or length of one cell.

Figure 5(d) shows a rendering maze, its solution, and the contained non-significant walls.

4.5. Prim*'s Algorithm*

At the initialization, all the walls are active, and one cell is randomly selected and marked.

The algorithm randomly selects an unmarked cell among the neighbors of the marked cells. The wall between these two cells is destroyed and we mark the current unmarked cell. The operation is repeated until all the cells of the grid are marked.

Figure 5(e) shows a rendering maze, its solution, and the contained non-significant walls.

4.6. Recursive Backtracking*'s Algorithm*

It is a recursive algorithm. At the initialization, all the walls are active. One cell of the maze is randomly selected and marked, considered as the current cell.

While the current cell is not surrounded by marked cells, we randomly select and mark an unmarked neighbor. The wall between the current cell and its neighbor is destroyed and the neighbor becomes the new current cell.

If there is no unmarked cell among the neighbors of the current cell, the algorithm goes back one cell and the instructions are repeated. The algorithm stops when all the cells are marked.

Figure 5(f) shows a rendering maze, its solution, and the contained non-significant walls.

4.7. Hunt & Kill*'s Algorithm*

At the initialization, all the walls of the grid are active. One cell of the maze is randomly selected and marked.

Starting in kill mode from this cell, a random walk is performed using the 4-neighborhood connectivity, only selecting unmarked cell. At each step, we destroy the walls in this path and mark the corresponding cells. The walk is stopped when the current cell is surrounded by marked neighbors.

The algorithm then switches to hunt mode: the maze is scanned line per line, from left to right and from top to bottom, until an unmarked cell neighboring a marked cell is found. The wall between these two cells is destroyed, and the unmarked cell is marked. It becomes the starting point of the next random walk of the kill mode.

The algorithm repeats these instructions until all the cells are marked.

Figure 5(g) shows a rendering maze, its solution, and the contained non-significant walls.

4.8. Growing Tree*'s Algorithm*

At the initialization, all the walls on the grid are active. An empty list is created, and a cell is randomly selected, marked and added to the list. A cell among those which are in the list is selected. If it has unmarked neighbors, one of them is randomly selected and the corresponding wall is destroyed. The algorithm marks and adds the selected neighbor to the end of the list. If it has no unmarked neighbors, the cell is removed from the list. The instructions are

repeated until the list is empty. The way cells are selected at the beginning of each iteration determines the behavior of the algorithm.

- Selecting the last cell of the list corresponds to the behavior of the *Recursive Backtracking* algorithm.
- Selecting the first cell of the list divides the maze in four parts. Each part is filled with parallel corridors in one given direction, long enough to reach the border of the maze.
- Selecting the cell in the middle of the list is like selecting the first cell, but a small winding area appears around the starting cell.
- Selecting the cell randomly corresponds to the behavior of *Prim*'s algorithm.

It is also possible to mix up the different behaviors. In our implementation, we selected the cell in the middle of the list.

Figure 5(h) shows a rendering maze, its solution, and the contained non-significant walls.

4.9. Kruskal*'s Algorithm*

In this algorithm, a flag $f(c) \in \mathbb{N}$ is associated to each cell c, as a connected component descriptor: if two cells (c, c') are flagged with the same label, it exists a path between these two cells. At the initialization, all the walls are active, thus each cell has its own flag. We create a list initialized with all the walls of the maze. It is used as the list of remaining walls. After each wall deletion, the flags are updated according to the connected components in the maze, by updating the flags of one side.

We randomly chose an existing wall inside the remaining walls and remove it from the list. If the two cells (c, c') have the same flag (i.e., are already

connected by a path), the wall is maintained in the board, otherwise the wall is destroyed. The operation is repeated until the list of the remaining walls is empty.

A classical way to implement *Kruskal*'s algorithm [14] is to use a disjoint set data structure, which is why some implementations refer to this algorithm as the *disjoint set algorithm.*

The difference between *Kruskal*'s and *Eller*'s algorithm is the way walls are chosen: *Eller*'s algorithm browses the maze line by line while *Kruskal*'s algorithm maintains a list of open walls and randomly select the next one.

Figure 5(i) shows a rendering maze, its solution, and the contained non-significant walls.

4.10. Wilson*'s Algorithm [22]*

It is a random-walk based algorithm. At the initialization, all the walls of the grid are active, and we randomly mark a cell.

A cell among the unmarked cells is randomly selected. A random walk is performed from this cell and is stopped when it reaches a marked cell. All the cells of this walk are stored in a stack. During the walk, if an already stored cell is reached, all the cells between the current cell and the reached cell from the stack (current cell included) are removed and the reached cell becomes the new current cell.

After each random walk, all the cells in the stack are marked, and all the walls in between are destroyed. The stack is then cleaned, and a new random walk is performed. This process stops when all the cells of the grid are marked.

Figure 5(j) shows a rendering maze, its solution, and the contained non-significant walls.

4.11. Aldous Broder*'s Algorithm [1, 4]*

It is a random-walk based algorithm. At the initialization, all the walls of the grid are active, and a random cell is marked, and set as the current cell.

A neighbor of the current cell is selected. If it is unmarked, the wall between it and the current cell is destroyed, and the neighbor is marked. Otherwise the algorithm does nothing. This neighbor becomes the new current cell.

We repeat these instructions until all the cells of the grid are marked.

Figure 5(k) shows a rendering maze, its solution, and the contained non-significant walls.

5. Our Two Maze Generation Algorithms

In this section, we present two original algorithms to go beyond the fun of mazes generated by existing algorithms (see section 6 for their comparison).

5.1. Prim & Kill *Algorithm (PK)*

We analyzed *Hunt & Kill* Algorithm (see Section 4.7) and find out that using a predefined set of marked cells, the Hunt mode is deterministic: it always picks cells in the same order. To counter this effect, we replaced the Hunt mode with one step of *Prim* (see Section 4.5) to bring more randomness to the process.

The main idea of *Prim & Kill* is to perform a random walk on a grid where all the walls are active, until we reach a point where no more wall can be destroyed. A similar random walk is then performed starting from a randomly selected wall of the visited region. We repeat this process until all cells of the grid are visited.

At the begining of this algorithm, we initialize two sets in order to know which cells have been visited. The first one contains all cells of the grid and is called $Unmarked$. The second one is empty and is called $Marked$. At the end of our algorithms, $Marked$ set contains all cells and $Unmarked$ is empty.

We first define a Random Walk function that starts from a given cell (*current*) and performs a random walk. This fonction also changes all visited cells during the random walk from the set $Unmarked$ to the set $Marked$.

Procedure Random Walk($Unmarked$, $Marked$, *current*)

```
while Neighbours(current) ∩ Unmarked ≠ ∅ do
    next = Pick randomly in Neighbours(current) ∩Unmarked ;
    Destroy wall between next and current ;
    Add element current to Marked ;
    Remove current from Unmarked ;
    previous = current ;
    current = next ;
return Unmarked, Marked
```

We now are able to give the algorithm for our *Hunt & Kill* algorithm. More precisely, at the beginning, all the cells are unmarked, except one selected randomly. The algorithm starts with a random walk from this cell (the Kill mode of *Hunt & Kill*). It marks all the visited cells and removes every wall in this random walk. The walk stops when all neighbors of the current cell are marked. A wall is then selected randomly among all the walls between a marked cell and an unmarked cell. After destroying this wall, a new random walk is performed. This process ends when all the cells are marked.

In Figure 5(l), we present an example of a maze generated with PK.

We notice that the random walks used in *Prim & Kill* generates a significant number of turns and a large number of non-significant walls (table 1).

5.2. Twist & Merge *Algorithm (TM)*

We introduce another algorithm based on a random walk with supplementary constraints. The main motivation of this algorithm is to generate mazes that

Algorithm 1: *Hunt & Kill* algorithm

Result: A 40×40 perfect maze.

Create a 40×40 maze full of walls;
$Unmarked$ = All cells;
$Marked = \emptyset$;
`/* HUNT MODE */`
$current$ = Pick randomly in $Unmarked$;
$Unmarked$, $Marked$ = Random Walk($Unmarked$, $Marked$, $current$)
while $Unmarked \neq \emptyset$ **do**
    $current$ = Pick randomly in $\{x \in Marked \mid \text{Neighbours}(x) \cap Unmarked \neq \emptyset\}$;
    $Unmarked$, $Marked$ = Random Walk($Unmarked$, $Marked$, $current$);

disadvantage the creation of corridors, since long corridors are paths of minor interest. We want to increase the fun using this approach.

The idea of *Twist & Merge* is to perform multiple *biaised* random walks to create paths where straight walks are forbidden. The produced regions are then merged into a single connected component. This approach favorizes the turns and increases the number of non-significant walks.

At the begining of this algorithm, we initialize a set called $Unmarked$ that contains all unvisited cells of the grid. This set is fill with all the cells of the maze. We also define an array called *Label* to store the label of the connected component in which each cell will be contained. At the beginning of the algorithm, this structured is initialized with a default label corresponding to the unmarked status.

We first define a Biaised Random Walk function that starts from a given cell ($current$) and performs a random walk. This fonction also remove all visited cells from the $Unmarked$ set, and set its label in the dedicated array *Label*. The neighbor selection is achieved using $aligned(.,.)$, where $aligned(x, y)$ is defined as all x neighbors if $x = y$, and as the opposite neighbor of x from y if $x \neq y$.

Procedure Biaised Random Walk($Unmarked$, $Label$, $current$, i)

```
previous = current;
while Neighbours(current) ∩ Unmarked ≠ ∅ do
    next = Pick randomly in { Neighbours(current) \ aligned(current,
      previous) } ∩ Unmarked ;
    Destroy wall between next and current ;
    Label[current] = i;
    Remove current from Unmarked ;
    previous = current ;
    current = next ;
return Unmarked, Label
```

We now are able to give the algorithm for our *Twist & Merge* algorithm. At the initialization, all the walls of the grid are active. The algorithm first starts in Twist mode. One cell of the maze is randomly selected and marked. Starting from this cell, a random walk is performed forbidding a straight walk of three cells. More formally if c_i and c_{i+1} are successive cells in the path, thus c_{i+2} cannot be $c_{i+1} + (c_{i+1} - c_i)$, where addition and substraction are opérations on coordinates of the cells. At each step, we destroy the walls in this path and mark corresponding cells. The walk is stopped when the current cell is surrounded by marked neighbors. This process is repeated until all cells are marked in the maze. Finally, the Merge mode is applied. It is a merging procedure of the connected components, similar to the one of *Kruskal*'s algorithm, in order to have a perfect maze.

Algorithm 2: *Twist & Merge* algorithm

Result: A 40×40 perfect maze.

```
1  Create a 40 × 40 maze full of walls;
2  Unmarked = All cells;
3  forall x ∈ Unmarked do
4  |  Label[x] = 0 ;
5  end
   /* TWIST MODE                                                    */
6  i = 1;
7  while Unmarked ≠ ∅ do
8  |  current = Pick randomly in Unmarked;
9  |  Unmarked, Label = Biaised Random Walk(Unmarked, Label,
   |   current, i);
10 |  i = i + 1;
11 end
   /* MERGE MODE                                                    */
12 while size(Marked) ≠ 1 do
13 |  a, b = Pick randomly in {x, y | y ∈ Neighbours(x) ;
   |   Label[x] ≠ Label[y] };
14 |  Destroy wall between a and b ;
15 |  forall x ∈ cells do
16 |  |  if Label[x] = Label[a] then
17 |  |  |  Label[x] = Label[b] ;
18 |  |  end
19 |  end
20 end
```

In Figure 5(m), we present an example of a maze generated with TM. We notice that it generates mazes that have many small turns and few corridors.

6. Ranking and Comparisons

Our aim is to construct a measure that can rank perfect mazes according their fun. After discussion with maze players, we choose to consider that a fun maze is a maze that is not trivial or boring to solve, but where a player have several "*crucial*" choices to perform in order to find the unique path to the solution. To consolidate this definition, we identified that mazes with the long paths are not the funniest ones, as illustrated by Figure 3.

McClendon's difficulty (D) is based on the trajectory equivalent to the one followed by the eyes of a player trying to solve the maze (section 3.2), thus seems to be a good ingredient to evaluate the fun. In the other hand, the number of non-significant walls (NSW) seems to be also an interested measure, since the more it will have non-significant walls, the more it will be easy and not fun to solve.

Figure 6a and Table 1 shows that these two measures are not correlated, each of them cannot be considered by itself as a good evaluation of the fun. One can identify that *Recursive Backtracking* and *Prim* algorithms are almost equivalent from the difficulty point of view, but significantly distinct from the NSW point of view. On the other hand, *Hunt & Kill* and *Twist & Merge* algorithms are almost equivalent from the NSW point of view, but significantly distinct from the Difficulty point of view. We note that this last observation confirms the objective we had set ourselves in defining this algorithm.

To continue this analysis, we can note that ***Growing Tree* algorithm** produces mazes with a very small difficulty, and a large number of non-significant

walls. We confirm that it is the algorithm that produces the least fun mazes to solve. The ***Recursive Division*** **algorithm** is also one of the algorithms producing the least fun mazes, following the same reasoning.

At another side of the diagram, the ***Recursive Backtracking*** algorithm has the least number of non-significant walls, together with the longest solution path which is not fun to solve.

After all this observation, we can conclude that the funniest algorithm should have the highest difficulty and the lowest number of significative walls. Hence, we propose to mesure fun F as the ratio between the number of non-significant walls ν and the difficulty (table 1):

$$F(M) = \frac{\nu(M)}{\delta(M)}. \tag{1}$$

Our two algorithms original algorithms *Twist & Merge* and *Prim & Kill* are clearly improving the difficulty of generated mazes comparing to previous existing algorithms, with a very low number of non-significant walls, thus seems to be good candidates to generate fun algorithms.

To consolidate the relevance of our proposal, we used Buck measures to evaluate all the algorithms. Figure 7 illustrates that *Twist & Merge* generates mazes with the highest number of turns and the lowest number of straight cells, which was its goal.

7. Conclusion and Future Work

We proposed an original approach to classify mazes according to their fun, and we designed two original maze generators: *Prim & Kill* (PK) and *Twist &*

Merge (TM). We have shown that TM algorithm is the funniest among all the other algorithms.

One future work will be to analyze each pair of algorithms, identifying the probability to generate similar mazes.

In [9], Járai shows that the spaces of possible mazes generated by *Wilson* and *Aldous Broder* are equivalent, since they can be reduced to a spanning tree approach. A possible continuity of this work could be to use equivalent approaches to show the similarity of the generated maze spaces, thus build a statistical similarity classification of the algorithms.

Another future work will be to use our algorithms to generate other kind of mazes. One idea is naturally to see how they can be adapted to generate non perfect maze, that can be interesting for humans. To achieve this objective, we would therefore have to adapt our evaluation measures.

Other constraints could be added, for instance the shape of the solution as in [17] or the general shape of the maze like in [12, 21, 18] where cells are not squares. In this case, we should also have to adapt our evaluations measures.

References

[1] Aldous, D.J., 1990. The random walk construction of uniform spanning trees and uniform labelled trees. SIAM Journal on Discrete Mathematics 3, pp. 450–465.

[2] Ashlock, D., 2010. Automatic generation of game elements via evolution, in: Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games, pp. 289–296.

[3] Ashlock, D., Lee, C., McGuinness, C., 2011. Search-based procedural generation of maze-like levels. IEEE Transactions on Computational Intelligence and AI in Games 3, pp. 260–273.

[4] Broder, A., 1989. Generating random spanning trees, in: 30th Annual Symposium on Foundations of Computer Science, pp. 442–447.

[5] Buck, J., 2015. Mazes for Programmers: Code Your Own Twisty Little Passages. Pragmatic Bookshelf.

[6] Foltin, M., 2011. Automated Maze Generation and Human Interaction. Ph.D. thesis. Masaryk University Faculty of Informatics.

[7] Gabrovšek, P., 2019. Analysis of maze generating algorithms. IPSI Transactions on Internet Research 15, pp. 23–30.

[8] Hoshino, S., Takahashi, R., Mieno, K., Tamatsu, Y., Azechi, H., Ide, K., Takahashi, S., 2020. The reconfigurable maze provides flexible, scalable, reproducible, and repeatable tests. iScience 23, 100787.

[9] Járai, A.A., 2009. The uniform spanning tree and related models. Available online at `http://www.maths.bath.ac.uk/~aj276/teaching/USF/USFnotes.pdf`.

[10] Kaplan, C.S., 2014. The design of a reconfigurable maze, in: Greenfield, G., Hart, G., Sarhangi, R. (Eds.), Proceedings of Bridges 2014: Mathematics, Music, Art, Architecture, Culture, Tessellations Publishing, Phoenix, Arizona. pp. 167–174.

[11] Kim, P.H., Crawfis, R., 2015. The quest for the perfect perfect-maze, in: Mehdi, Q.H., Elmaghraby, A., Marshall, I., Lauf, A.P., Jaromczyk, J.W.,

Ragade, R.K., Zapirain, B.G., Chang, D., Chariker, J., El-Said, M.M., Yampolskiy, R.V. (Eds.), Computer Games: AI, Animation, Mobile, Multimedia, Educational and Serious Games, CGAMES 2015, Louisville, KY, USA, July 27-29, 2015, IEEE Computer Society. pp. 65–72.

[12] Kim, P.H., Grove, J., Wurster, S., Crawfis, R., 2019. Design-centric maze generation, in: Proceedings of the 14th International Conference on the Foundations of Digital Games, Association for Computing Machinery, New York, NY, USA. pp. 1–9.

[13] Kozlova, A., Brown, J.A., Reading, E., 2015. Examination of representational expression in maze generation algorithms, in: 2015 IEEE Conference on Computational Intelligence and Games (CIG), pp. 532–533.

[14] Kruskal, J.B., 1956. On the shortest spanning subtree of a graph and the traveling salesman problem. Proceedings of the American Mathematical society 7, 48–50.

[15] Lee, H., Lee, C., Chen, L., 2010. A perfect maze based steganographic method. Journal of Systems and Software 83, pp. 2528–2535.

[16] McClendon, M.S., 2001. The complexity and difficulty of a maze, in: Bridges: Mathematical Connections in Art, Music, and Science, Bridges Conference. pp. 213–222.

[17] Okamoto, Y., Uehara, R., 2009. How to make a picturesque maze, in: Proceedings of the 21st Annual Canadian Conference on Computational Geometry, Vancouver, British Columbia, Canada, August 17-19, 2009, pp. 137–140.

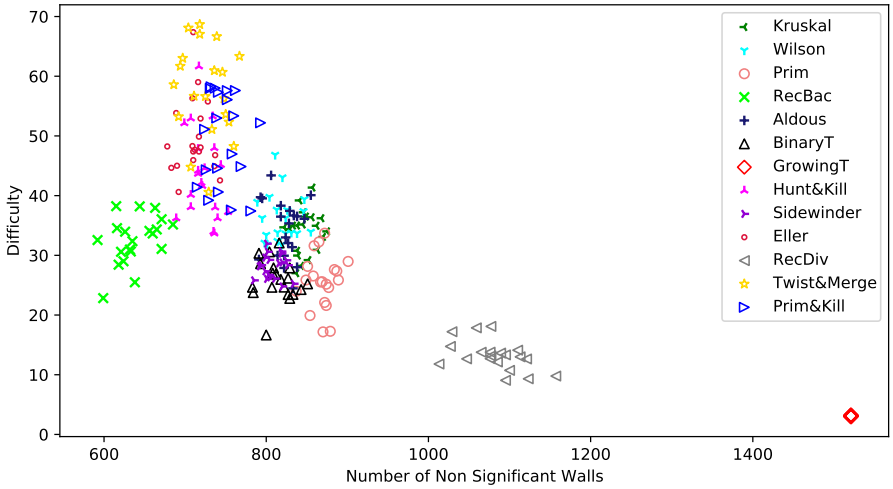
[18] Pedersen, H., Singh, K., 2006. Organic labyrinths and mazes, in: Proceedings of the 4th International Symposium on Non-Photorealistic Animation and Rendering, Association for Computing Machinery, New York, NY, USA. p. 79–86.

[19] Pullen, W.D., 1996. Think labyrinth! `http://www.astrolog.org/labyrnth.htm`.

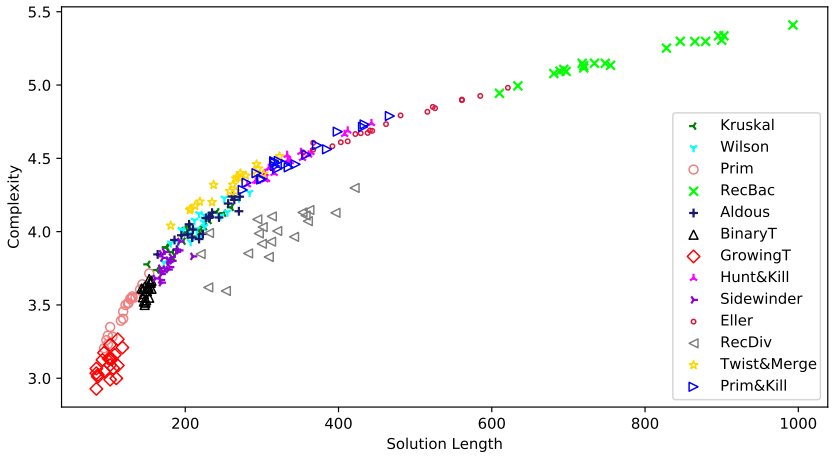
[20] Turan, M., Aydin, K., 2010. A dynamic terrain-spaced maze generation algorithm. Global Journal of Computer Science and Technology 10, pp. 9–14.

[21] Wan, L., Liu, X., Wong, T., Leung, C., 2010. Evolving mazes from images. IEEE Transactions on Visualization and Computer Graphics 16, pp. 287–297.

[22] Wilson, D.B., 1996. Generating random spanning trees more quickly than the cover time, in: Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, NY, USA. p. pp. 296–303.

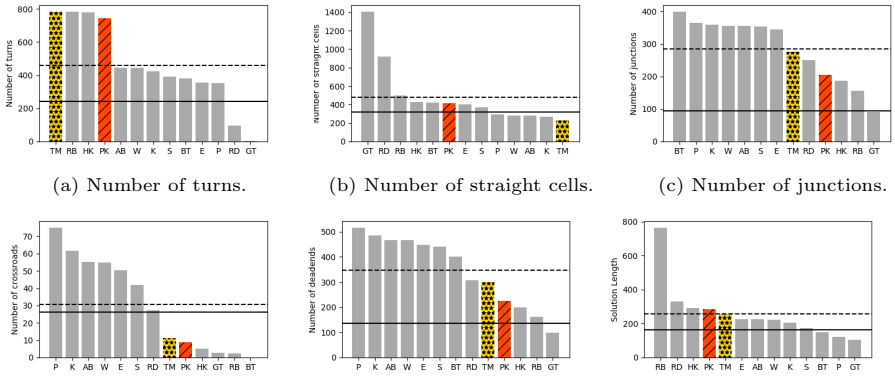


(a) 2D plotting of the algorithms using the number of non-significant walls and the difficulty introduced by McClendon [16].



(b) 2D plotting of the algorithms using the length of the solution path and the complexity introduce by McClendon [16].

Figure 6: 2D plottings of the algorithms using two mesures. 20 randomly generated 40×40 mazes are used for each algorithm. We chose only 20 mazes per algorithm which is representative on a readable plot.



(a) Number of turns. (b) Number of straight cells. (c) Number of junctions.

(d) Number of crossroads. (e) Number of dead-ends. (f) Length of the solution path.

Figure 7: Buck measures on 1000 generated 40×40 mazes per algorithm. The dotted line locates the mean while the plain one locates the standard deviation.