Masaryk University
Faculty of Informatics

Automated Maze Generation and Human Interaction

DIPLOMA THESIS

Martin Foltin

Brno, 2011

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Advisor: Mgr. Radek Pelánek, Ph.D.

Acknowledgement

I would like to thank Mgr. Radek Pelánek, Ph.D., the advisor of my master thesis, for his valuable comments, suggestions and time he spent helping me with this work. I would also like to thank my family for moral support.

Abstract

This thesis' aim is to study and contribute to the human problem solving analysis, which is a very complex domain. That is why it is helpful to perform an analysis of a logically easier comprehensible topic – mazes and maze solving in this case.

History of mazes, maze categorization as well as several graph based maze generation algorithms (e.g. Prim's algorithm, Kruskal's algorithm or DFS based algorithms) are presented and explained in detail. Relevant graph theory concepts are explained prior to the discussion of maze generation algorithms in order to make their logical intersection clear and observable.

Further subchapters study connections between automated maze generation (properties of a maze created by a specific algorithm) and its impact on human maze solving. Human interaction and behavior analysis is performed on a sample data gathered by Maze Generation Java applet, which I implemented for purposes of this master's thesis. Technical background of the implementation part is described in this thesis, too.

The applet let users solve predefined mazes or mazes generated by various randomized algorithms. Information about each solve attempt was stored and preserved for further analysis. The analysis was performed using several methods (some of them used probably for the first time – e.g. a maze density map) and it led eventually to some interesting conclusions (e.g. the definition of the so called got-lost ratio and its relationship to users' movement through the maze).

Keywords

maze, maze generation, maze generation algorithms, graph algorithms, Prim's algorithm, Kruskal's algorithm, recursive backtracker, hunt-and-kill algorithm, bacterial growth algorithm, maze solving, human problem solving

Contents

1 **Introduction to Mazes** 2
1.1 *Maze Concept* 2
1.1.1 What is a Maze? 2
1.1.2 History of Mazes 2
1.1.3 Application of Mazes 3
1.2 *Categorization of Mazes* 3
1.2.1 Dimension 4
1.2.2 Hyper-Dimension 4
1.2.3 Topology 5
1.2.4 Tessellation 5
1.2.5 Routing 6
1.2.6 Texture 7
1.2.7 Focus 8
2 **Maze Generation** 10
2.1 *Maze Generation Techniques* 10
2.2 *Introduction to Topic Relevant Parts of Graph Theory* 10
2.2.1 Basic Concepts 11
2.2.2 Graph Theory, Mazes and Maze Generation Algorithms 13
2.3 *Graph Based Maze Generation Algorithms* 15
2.3.1 Randomized Prim's Algorithm 15
2.3.2 Randomized Kruskal's algorithm 18
2.3.3 Recursive Backtracker 20
2.3.4 Hunt and Kill Algorithm 22
2.3.5 Bacterial Growth Algorithm 23
2.4 *Other Approaches of Maze Generation* 24
3 **Implementation** 26
3.1 *Back End* 26
3.2 *Front End* 28
3.2.1 User Interface 28
3.2.2 Relevant Classes and Data Structures 29
4 **Analysis of Human Maze Solving** 31
4.1 *Human Problem Solving* 31
4.2 *Maze Solving* 32
4.2.1 General Assumptions 32
4.2.2 Brute Force vs. Analytical Solving 33
4.3 *Data Collection* 35
4.4 *Data Analysis* 36
4.4.1 Analyzing Tools 36
4.4.2 Randomized Mazes Analysis 38
4.4.3 Predefined Mazes Analysis 40

4.4.4 User Behavior Analysis . . . 52
5 **Conclusion** . . . 58
Bibliography . . . 60
A **Figures** . . . 61
B **Kirchhoff's Theorem Example** . . . 69
C **Java Source Code Examples** . . . 70
D **Table Data** . . . 74
E **Electronic Attachment** . . . 76

Introduction

The topic of mazes is very old. Mazes originate in ancient history, when the first labyrinths were designed and built. Of course their primary purpose nowadays of fun and entertainment was not that relevant in those times, but it developed throughout the centuries. As the global education and wisdom level increased, mazes became very popular as a fun and entertainment tool and also as a very interesting domain from the mathematical point of view. Mazes have been applied more and more often within the real life, which brought up (especially parallel with the personal computer boom) the need to have mazes generated in an automated (i.e. algorithmic) way.

A logical question can come to a reader's mind: is there anything new such a paper can bring into the topic of mazes? There certainly is, but it is not really the aim of this thesis. Its aim is to summarize several maze generation methods (with focus on graph related techniques) and to perform an exploratory study of human interaction with mazes.

The first part of this thesis only briefly presents the basic concept of mazes, history of mazes, their perception by laic and professional public today and their real life applications. This chapter is followed by the chapter called Maze Generation presenting possible algorithmic ways of creating mazes. Since it focuses on graph theory related generation techniques, some basic concepts of graph theory are explained and semantically interconnected with related maze terms. The next chapter (Implementation) briefly presents the technical background of Maze Generation Java applet, which was implemented as a part of this thesis in order to gather statistical data usable for human interaction analysis. Finally, the statistical analysis of gathered users' results is performed and discussed in the fourth chapter.

Chapter 1

Introduction to Mazes

The task of this chapter is to present some basic concepts of mazes, their history and categorization before dealing with graph theory concepts and maze generation algorithms per se. It may contain informal terms and references, which will be formalized in further chapters. These terms are however usually intuitively understandable, since the subject of mazes is generally wide spread and comprehensible.

1.1 Maze Concept

1.1.1 What is a Maze?

A maze is a tour puzzle[1] consisting of variously branched passages. A solver's aim is to move from a starting point to an end point i.e. a valid passage between these two points has to be revealed. Since maze is a subtype of the puzzle concept, the same basic puzzle rule applies to it as well: a maze is a problem or enigma testing solver's diligence, ingenuity and ability to solve it. Mazes are of various shapes, kinds and difficulty. Section 1.2 focuses on differences between them.

1.1.2 History of Mazes

Some sources interchangeably refer to mazes also as labyrinths. However there is no consensus on that among the maze fan community. The word labyrinth has a deep historic background in ancient Crete. It was a structure elaborated by Daedalus for King Minos. Its purpose was to hold back the Minotaur. Daedalus' labyrinth was of a unicursal[2] sort and that is the aspect dividing the maze fan community into two fractions. Some believe the word labyrinth should be used only for unicursal mazes (ergo a labyrinth would be a subtype of mazes), the other fraction believes both terms are equal. For instance Oxford dictionaries[3] make no semantic difference between both labyrinth and maze.

---

1. Tour puzzle is a category of puzzles. Tour puzzles make the solver travel around a board (usually but not necessarily two-dimensional) in order to find a solution.
2. A unicursal maze is a maze without any junctions and choices. It is just one long passage which slows down the solver from reaching the end point but does not really make it difficult in any other usual way (e.g. loops, crossings where a decision is needed etc.). See also Section 1.2.5.
3. Labyrinth – a complicated irregular network of passages or paths in which it is difficult to find one's way, a maze. Maze – a network of paths and hedges designed as a puzzle through which one has to find a way.

1.1.3 Application of Mazes

Entertainment and brain training may be considered as primary goals of maze application. So called logic mazes are special mazes with specifically defined rules altering the usual way of maze solving. This typically involves rules regarding step sequence constraints, moving through some predefined checkpoints, etc.

In some countries (e.g. USA) there is a tradition to build life size mazes and promote them as tourist attractions. Lists of life size mazes across the world can be found for instance in the following sources: [10] – Life Size Mazes, [1], [16].

Another application of mazes can be found in the video game industry. Often there is a demand for random creation of interior and/or exterior environment. For example in case of creating an interior ground plan, maze cells can be mapped into rooms and maze paths into doors.

Mazes were also inspiration for many movies, e.g. Labyrinth (1986).

1.2 Categorization of Mazes

Walter D. Pullen did a great work on summarizing the topic of maze categorization. His work is one of the most quoted maze related sources on the Internet. According to him, there are seven categories mazes can be classified by:

- dimension
- hyper-dimension
- topology
- tessellation
- routing
- texture
- focus

Affiliation of a maze to some category (or categories) usually influences maze features related to the solving phase: difficulty, time needed to solve the maze, fun rate etc. Of course, all these properties are subjective depending on the human solver. See Chapter 4, "Analysis of Human Maze Solving" for more information on this topic.

This subchapter explains terms and concepts in an intuitive and informal form. The Chapter 2, "Maze Generation" makes everything clearer and more formal.

Due to the page number limitation, this master's thesis contains only few example figurations relevant for this chapter. See [10] – Maze Algorithms for deeper descriptions and more example figurations.

(http://oxforddictionaries.com)

1.2.1 Dimension

Dimension is a quite intuitive concept, but it is worth defining it in a formalized way in order to understand it properly. Eric W. Weisstein ([14] – Dimension) sees dimension of an object as *"a topological measure of the size of its covering properties. Roughly speaking, it is the number of coordinates needed to specify a point on the object. For example, a rectangle is two-dimensional, while a cube is three-dimensional. The dimension of an object is sometimes also called its dimensionality."*

This thesis focuses on two-dimensional mazes. They are easy to draw on a paper or to realize in a life size manner.

One can imagine a three-dimensional maze as a group of two-dimensional mazes placed "on each other" with an option to move upwards or downwards between them. Since there are many widely spread and understandable algorithmic ways of creating a two-dimensional maze, creating a three-dimensional maze is not really difficult as well. As already mentioned, virtual space (e.g. video game) designers use such methods to create randomized complex environments.

There are also so called weave mazes. Weave mazes enable overlapping or layering of paths within the maze, yet there is a limitation. The solver never has an option to move between layers when being positioned at one cell (from the ground plan point of view). One has to move inevitably between cells within the two-dimensional point of view in order to change the position within the 3rd dimension. Therefore weave mazes are somewhere between 2D and 3D (yet closer to 2D). Walter D. Pullen describes them as 2.5-dimensional [10].

Mazes of a higher dimension (>3) may be difficult to imagine at first sight. Yet one of the acceptable conceptions of higher-dimensional mazes is quite trivial. It describes them as a set of three-dimensional mazes interconnected with so called portals [10].

1.2.2 Hyper-Dimension

This category refers to the dimension of the object the solver moves through the maze. The following rule applies for dimensional relationships between mazes (both non-hypermaze and hypermaze) and objects being moved within them:

Let n be the dimension of an object to be moved through a maze (e.g. a 0-dimensional point). It is being moved along a path with dimension of n+1 (e.g. a 1-dimensional path). Then the environment the maze exists in has to be of dimension at least n+2 (a plane, a solid, etc.). [10]

The previous definition may not be intuitively and clearly seen in some graphical maze representations. For instance classic maze video games usually depict a 2-dimensional object being moved through a 2-dimensional environment leaving a 2-dimensional figure behind. Yet it is necessary to abstract from a classic point of view and concentrate on the topological point of view, which enables "reducing" all observed objects to dimensions fulfilling the definition above.

There are three items in this category: non-hypermaze, hypermaze and hyperhypermaze. The most intuitive way to distinguish between various dimensions of the object being moved within a maze is to observe the geometric figure the object "leaves behind".

The term non-hypermaze describes the case of a point-like (0-dimensional) object. When being moved through a maze, a line (1-dimensional object) is left behind. In case the moved object is a line, the geometric figure formed behind is a plane (2-dimensional object). It is described as hypermaze (of kth order, $k > 0$).

This master's thesis focuses on non-hypermaze. Hyperhypermazes are out of its scope.

1.2.3 Topology

According to Walter D. Pullen's work, the topology category divides mazes according to the space they exist in.

Normal

A normal maze is a maze existing in a standard Euclidean space. Such a space needs to fulfill all Euclid's postulates ([14] – Euclid's Postulates).

Plainair[4]

Plainair mazes are mazes that do not fit into a regular Euclidean space (by not fulfilling one or more of Euclid's postulates). For instance a set of interconnected plane mazes placed on a surface of a cube.

1.2.4 Tessellation

The tessellation categorization divides mazes into several groups according to the shape (and/or its regularity or irregularity) of their basic units they are composed of – cells.

Orthogonal

The implementation part of this master's thesis deals with mazes of orthogonal tessellation. A maze cell must have at least three edges in order to form a regular 2-dimensional plane. Orthogonal tessellation defines the condition of perpendicularity between all neighboring pairs of cell walls (logically implying that the cell has inevitably four walls and it is a rectangle or a square). Such a maze cell can have up to four neighboring cells intersecting at right angles (see Figure A.1 for an example).

Omega

Walter D. Pullen uses so called omega-like tessellation, which should cover any non-orthogonal mazes (e.g. delta or upsilon). Its aim is also to cover individually defined tessel-

4. Plainair – planes twisted through air ([10] – Maze and Labyrinth Glossary).

lations, since there are many possibilities how to do that. One just needs to define his own cell shape and create an adequate interconnecting system.

Theta

The theta tessellation forms a maze into a specific ring-like shape. Either the starting point or the end point is located in the center, the other is then located at the outer edge of the maze. All passages are aligned along circles with the same center but with different radius from it.

Crack

Crack ("shapeless" or "deformed") represents mazes without any consistent tessellation system. Automated generation of such mazes (e.g. pixel carving approach: [10] – Maze Algorithms) is different from algorithms primarily described in this master's thesis and is out of its scope.

Fractal

A fractal maze is a maze that consists of several mazes assembled together. Stanislav Vejmola deals with this kind of maze tessellation as well ([13], chapter 4.3).

Other Tessellations

- Zeta – an orthogonal tessellation with one addition. Besides regular four directions (oriented at the right angle) there is also an option to move in 45 degree angle. I.e. cell's diagonals can be designed as walls too.
- Delta – represents a triangle shaped maze cell that can have up to 3 neighboring cells.
- Sigma – a basic maze unit is formed into the shape of a hexagon. Such a cell can have up to six neighboring cells.
- Upsilon – octagon shaped cells that can have up to eight neighboring cells ergo eight possible directions to move to.

1.2.5 Routing

The Routing category classifies mazes into several subcategories according to the properties of their passage (path) system. The passage system pattern can sometimes help to determine the method by which the maze was created (see also Section 2.3).

Perfect

There are several constraints a perfect maze needs to fulfill: there must not be any passage loops, there must not be any isolated cells (so called islands) in the maze and there must be exactly one path between any pair of cells (i.e. the maze has exactly one solution). The concept of a perfect maze will also be discussed in further parts of this master's thesis.

Braid

A braid maze is a maze without any dead ends. One can imagine a passage pattern of such maze as a web of inter-circled paths. The absence of dead ends may radically increase the difficulty of the maze (in case it is designed in a clever way).

Unicursal

The concept of a unicursal maze has been already discussed. Its passage system consists of just one snake-like passage without any choices for the solver to be taken (see Figure A.3). Solving such a maze is generally easy. All it can cause is delaying the solver from reaching the end point. Yet this may be the desired effect in some cases.

Sparseness

A sparse maze is created by not carving and eventually not including all the grid cells (or pixels) in the maze. It means there can be inaccessible parts. (Do not confuse with an island – an inaccessible set of passages surrounded by walls.) Such a cell is not a part of any passage, thus it becomes a part of the wall.

Partial Braid

A partially braid maze is a combination of dead ends and loops. A dead-end-to-loop ratio[5] may be calculated and altered in order to analyze its influence on difficulty of the maze.

1.2.6 Texture

Identifying a maze's texture can be done either by just observing the maze or by expressing it using mathematical evaluation. The first alternative requires the maze texture to by quite obvious, i.e. its texture deviation should be more or less extreme.

The second alternative is pretty straight and requires a statistical expression of maze cells' properties. When comparing intensity of two states (e.g. occurrence of short and long passages), one needs to calculate the number of cells in each state. A case-specific and rational method should be chosen in order not to alter the results undesirably. This concerns especially cells under investigation which seem to be in both states (e.g. intersection cells).

5. Let the dead-end-to-loop ratio be the relationship between the number of cells belonging to a dead end path and the number of cells belonging to a loop.

Bias

A bias texture describes the maze paths' tendency to be directed in either a horizontal or vertical way. A horizontally biased maze has (relatively) long horizontal passages and (relatively) short vertical passages. The same but vice versa applies for vertically biased mazes.

Run

A high run-like maze is a maze characterized by (relatively) long passages (in either direction). To distinguish a short passage from a long one, a border line needs to be set. It should be determined ad hoc depending on the size of the analyzed maze.

Elite

Maze elitism is about the solution path. One can observe its two main characteristics: number of maze cells being a part of the solution path and the enlacement eventually the perplexity of the solution path. See the difference between solution paths of Figure A.8 and Figure A.9.

Symmetry

A maze is symmetric if one part of the maze is mirrored into another part. This effect can be achieved by applying rotational symmetry[6] or reflection using reflection axis[7]. The reflected part preserves the cell structure of the original part, but in reflected cell arrangement.

River

A maze with a high river factor is a maze created using an algorithmic or non-algorithmic method, which carves river-like paths. It means the path "flows" through the grid and a newly created cell is mostly a follower of the previously created cell. See for instance Figure A.9 or Figure A.10 for high river factor examples.

1.2.7 Focus

Although the title does not exactly express it, Walter D. Pullen uses this category to distinguish various methods of maze creation. There are two basic algorithmic ways: wall adders and passage carvers.

Wall adders are algorithms focusing on wall positioning. Virtual creation takes place usually on an empty canvas. The algorithm places walls on the canvas in an algorithm-specific way.

6. A rotationally symmetric object is an object that preserves its looks after a certain amount of rotation.
7. Mathematics uses the term fixed point or set of fixed points – an axis in our case. A fixed point is a point where a function maps to itself, i.e. $f(x) = x$ holds. The object is mirrored by the so called axis of reflection.

Passage carvers concentrate on paths and cell positioning. Related algorithms deal with cells and with their relationships. Virtually, one can imagine it as a canvas with a grid completely full of walls. A passage carving algorithm carves through the grid and destroys some of the walls (ergo creating a path between the cells which shared the destroyed cell). Again – it proceeds according to its algorithm-specific step order.

Chapter 2

Maze Generation

2.1 Maze Generation Techniques

There are two basic ways of maze generation: algorithmic and non-algorithmic.

Algorithmic methods create mazes according to a predefined step order. In order to create a randomly (more accurately – a pseudorandomly) structured maze, at least one or more steps need to be randomized (i.e. the decision making within the step has to use a function returning a random result). This thesis puts focus on graph based algorithms (while there are actually two aspects of mazes that are related to graphs: maze generation and maze structure per se).

Graph based maze generation algorithms create mazes by building a spanning tree, which is a kind of graph (see further chapters for a more formal description). At the end of the generation process a spanning tree topologically corresponding to a maze is created. The tree is either being gradually assembled by joining smaller fragmental tree structures (e.g. Kruskal's algorithm) or the creation can continuously expand from one specific randomly chosen spot (a maze cell designated as the root of the tree), e.g. Prim's algorithm.

There are two basic ways of automated maze generation: wall adding and passage carving (see Section 1.2.7). The recursive division method is an example of a wall adding algorithm. Some maze generation algorithms (e.g. Prim's algorithm) can even be implemented both ways, i.e. as a passage carver as well as a wall adder.

Fractal mazes (see Section 1.2.4) can be potentially created in an algorithmic way. It is a way of assembling individual fractal mazes into a whole. All fractal maze parts need to be topologically equivalent and their tessellation systems must fit. Also the border overpasses between them have to be synchronized in order to enable the solver to move within them.

The non-algorithmic methods are not interesting for the purpose of this thesis and will not be discussed any further.

2.2 Introduction to Topic Relevant Parts of Graph Theory

This subchapter may seem to contain too much mathematically oriented information. Yet it is essential (and mostly very intuitive and understandable) knowledge one needs to comprehend in order to study algorithmic maze generation further. Maze generation and mazes generally have a lot in common with mathematics. Czech "puzzle guru" Doc. Ing. Stanislav

Vejmola, CSc. for example considers mazes to be the most interesting part of fun mathematics [13].

2.2.1 Basic Concepts

This chapter cumulates and presents only basics and just those parts from graph theory relevant for this master's thesis. In no manner is this an attempt to sum up the graph theory in a complex way. This chapter mixes terminology from several sources, chosen according to the intuitive clearness of explanations.

Graph Theory

Graph theory is said to have its origins in the problem of citizens of Koenigsberg [4]. The city was divided by a river into four parts connected by seven bridges (see Figure 2.1). They were wondering whether it was possible to start walking from any land mass, cross all bridges and return to the starting point without crossing any bridge more than once. Swiss mathematician and physicist Leonhard Euler (1707–1783) analyzed this problem by simplifying the reality to basic graph concepts – vertices[1] and edges[2]. (Eventually he found out there was no way to make such a walk, since all graph nodes would have to be of an even degree[3].)

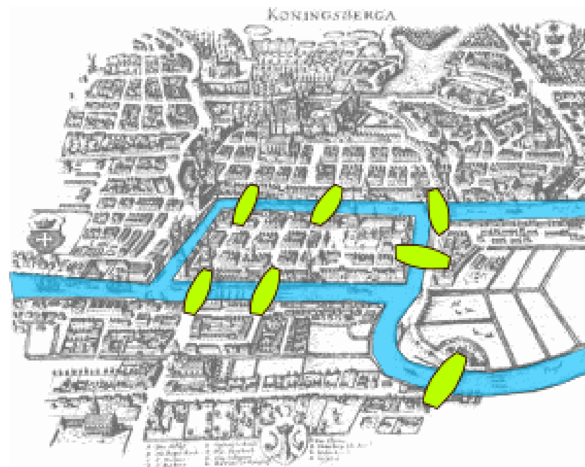


Figure 2.1: Seven Bridges of Königsberg [15]

1. Vertex – sometimes also referred to as node or point.
2. Edge – sometimes also referred to as connection or line.
3. The degree of a vertex of a graph is the number of edges incident to it.

Graph

A graph is the basic concept in graph theory. It is a triplet composed of a vertex set *V(G)*, an edge set *E(G)*, and a mapping (function) $\varepsilon : E \rightarrow V^2$ called *relation of incidence*. This function associates a pair of vertices {x, y} to each edge $e \in E$. Edge *e* and its associated vertices are *incident*. A vertex without any incidence is an *isolated* vertex.

A graph can be either *directed* or *undirected*. A directed graph contains oriented edges and the notation $e = (x, y)$ describes an ordered pair and its orientation from x to y (eventually depicting various relationships – precedence, superiority, time chronology etc.). On the other hand edges of undirected graphs have no orientation and incident vertices are organized in a set meaning there is no order (notation $e = \{x, y\}$). This is sufficient for purposes of maze generation algorithms and their graph presentation.

Another graph attribute is so called *connectivity* and it is used to distinguish connected and disconnected graphs. A connected graph is a graph interconnecting all vertices, i.e. there is not such a pair of vertices without being connected. Vertices are called *adjacent* if they are connected via a single edge.

Graph Comparison

Graphs $G = (V, E, \varepsilon)$ and $G' = (V', E', \varepsilon')$ are *equal*, if $V = V'$, $E = E'$ and $\varepsilon = \varepsilon'$.

Directed graphs $G = (V, E, \varepsilon)$ and $G' = (V', E', \varepsilon')$ are *isomorphic* [4] under following circumstances:

- let *f* be a function $f : V \rightarrow V'$
- let *g* be a function $g : E \rightarrow E'$
- $\varepsilon(e) = \{x, y\} \Leftrightarrow \varepsilon'(g(e)) = \{f(x), f(y)\}$

Informal hints to decide on isomorphism are as follows:

- isomorphic graphs have the same number of edges and vertices
- each vertex *x* of a degree *d* is mapped by isomorphism to a vertex *x′* of the exact same degree
- a couple of adjacent vertices *x, y* are mapped by isomorphism again only to a couple of adjacent vertices *x′, y′*

Another way of identifying structural similarities between graphs is the concept of *subgraph*. Graph *G′* is a subgraph of graph *G* if it is created by removing *n (n>=0)* edges or *m (m>=0)* vertices from graph *G*. This implies that graph *G* is also a subgraph of itself.

Factor *G′* of a graph *G* is a subgraph preserving the whole vertex set of graph *G* (also referred to as spanning subgraph). If $G = (V, E, \varepsilon)$ and $G' = (V', E', \varepsilon')$ then $V = V'$.

4. Graph isomorphism is a bijective map preserving graph structure (vertices, edges and their incidence).

Tree

A tree is an undirected connected graph without cycles[5] i.e. there is exactly one path[6] between each pair of vertices. A connected graph with *n* vertices and *n-1* edges is a tree. Removing any edge would cause the graph's disconnectedness whereas adding an edge would create a cycle.

Spanning Tree

If a tree is spanning, it means it includes all the graph vertices. A spanning tree of a connected graph *G* can also be defined as a *"maximal set of edges of G that contains no cycle, or as a minimal set of edges that connect all vertices"* [17]. Graph *G* with a finite number of vertices has a finite number of spanning trees *t(G)*. This invariant[7] number can be calculated using the Kirchhoff's matrix-tree theorem. A possible step sequence to calculate a graph's spanning tree number is to:

- construct the graph's degree matrix and adjacency matrix
- construct the Laplacian matrix *Q* (which is the difference of the two matrices mentioned above)
- calculate an absolute value of any (arbitrary) cofactor of *Q*

See Appendix B for a simple example calculation. Another alternative of calculating the number of spanning trees of a graph includes eigenvalues calculation of the graph's Laplacian matrix[8]. Jiří Sedláček also deals with this topic presenting a third calculation alternative ([11], chapter 7).

Weighted Graph

A graph may also be weighted. It means that a label (so called weight – usually a number) is associated with every edge in the graph. Then the so called minimum spanning tree is a spanning tree with weight less than or equal to the weight of every other potential spanning tree of the graph. Weight is not relevant for maze representation in general. Yet some maze generation algorithms are based on algorithms that deal with minimum spanning trees, that is why it is good to know at least the basic principle.

2.2.2 Graph Theory, Mazes and Maze Generation Algorithms

First of all an important thing needs to be mentioned. Graphs serve the purpose of some maze generation algorithms, but not only. A maze structure per se (aside from its generation process) can also be represented by some kind of a graph, since it is "just" a set of

5. Cycle – sometimes also referred to as loop.
6. A path is a list of *n* edges connecting two graph vertices $x, y \in V, x \neq y, n > 0$.
7. Invariant property is a property that does not change when applying specific (ad hoc) transformations to the object holding this property.
8. http://en.wikipedia.org/wiki/Kirchhoff's_theorem

vertices and a set of edges. I.e. even maze structure that was not created by a graph oriented algorithm may be represented by some sort of a graph.

Graph based maze generation algorithms described in this thesis are based on various graph related algorithms. These include creating, traversing or searching diverse graph structures. Applying an algorithm for maze creation usually involves deriving the algorithm it is based on, although the basic logic remains untouched.

The conceptual intersection of graph theory and graph based maze generation algorithms is quite intuitive since many concepts are named similarly or identically.

Consider a maze being a *grid* [9]. Then the graph theory related analogy to a grid is a graph. A grid consists of *cells* – graph vertices. Following maze parts are supposed to be turned into graph vertices when redrawing a maze into a graph: crossing, last cell of a dead end path, entrance cell, exit cell, isolated cell[10]. Numbering or another way of identification may be used in order to identify a maze cell with a graph vertex it is mapped into.

Grid cells (i.e. maze cells) are connected to each other via *paths* i.e. edges of a graph. The path between cells x and y denotes the ability to carve from x to y and vice versa. Possible formal denotation of a path between vertices x and y is $e_{x,y}$ ([13], page 75). If the number of edges (paths) between two cells is equal to 1, these cells can be referred to as *adjacent cells*. A path connecting cells *1* and *n* via a series of neighboring pairs of adjacent cells may be formulated as a sequence of several single path denotations $e_{1,2}, e_{2,3}, \ldots e_{n-1,n}$.

Maze presentation using a graph has several relevant advantages. First of all it is the most intuitive way for a graphical figuration of a maze. Another advantage is a graph's ability to abstract from scaling, original maze cells' positioning and shape. (Abstraction from the shape of maze cells may be very helpful especially in case of their extraordinary or unusual shape.) The vertex set and edge set and their connections are the only important aspects. That is why a redrawn positioning system in a form of a graph can make the maze cells' relationships clearer and more evident than a classic maze grid. Graph presentation can also assist at structural comparison of mazes using the concepts of subgraph and graph isomorphism.

The definition of a perfect maze (which might be used as a start up point and which can be easily derived to create definitions of other maze types[11]) includes the following constraints:

1. The maze has no loops.
2. The maze has no isolated areas.
3. Between any pair of cells there's exactly one path (i.e. there's only one possible solution).

9. Intuitive and wide spread understanding and graphical representation of a maze grid is a rectangular shape with orthogonal tessellation. Yet it is important to keep in mind that this is just one of the potential representations. The other ones are more or less also eligible to be interpreted by a graph structure of some sort.
10. An isolated group of interconnected maze cells is an island (isolated graph).
11. Perfect maze as a maze type is mentioned with reference to the routing category item (see Section 1.2.5).

According to its definition within graph theory, a tree as a specific graph structure is used for purposes of fulfilling conditions 1) and 3). A tree as a structure is therefore very close to fit the need to represent a perfect maze. To get a little closer, another conditional tree feature has to be deliberated – *spanning* [12]. A spanning tree of a connected graph *G* is a tree composed of all the vertices of *G* (and the 2nd condition is therefore fulfilled). Let *G* be a connected graph with *V* vertices. Then any spanning tree within *G* will have *V-1* edges.

If the maze is perfect, any cell can be designated as a root of a spanning tree matching the maze structure. If the starting cell of a maze (a cell, where the solving starts) is designated as the root, some algorithms (e.g. the depth-first search algorithm) can be applied to search the tree and find the way from the starting cell to the end cell.

2.3 Graph Based Maze Generation Algorithms

2.3.1 Randomized Prim's Algorithm

Prim's algorithm is an algorithm capable of finding a minimum spanning tree for a connected weighted undirected graph. This algorithm per se is not really interesting for the purposes of automated maze generation. What is more relevant is its randomized version. But some tutorial information regarding the Prim's algorithm should be presented first.

The algorithm was first developed by a Czech professor and mathematician Vojtěch Jarník in 1930. Another inventor of this algorithm is Robert Clay Prim, who invented it independently from Vojtěch Jarník in 1957. Eventually a Dutch computer scientist Edsger Wybe Dijkstra rediscovered it in 1959. Therefore this algorithm can also be referred to by the following names: Jarník algorithm, Prim–Jarník algorithm or DJP algorithm.

As already mentioned, this – so called greedy[13] – algorithm can be used to find a minimum spanning tree within a graph, i.e. a tree structure containing all the graph vertices. The limitations of the spanning tree structure have been discussed in the previous chapter. Jonathan Gross and Jay Yellen ([6], page 146) provide a quite intuitive example of Prim's algorithm pseudocode:

Input: a weighted connected graph *G*.
Output: a minimum spanning tree *T*.

1. Choose an arbitrary vertex *s* of graph *G*.

12. Some sources refer to the minimum spanning tree concept. This means that weighted (usually numbered) edges are taken into consideration. Yet it is not relevant for the maze generation process dicussed in this thesis, since it is not necessary for the paths (i.e. edges) to be weighted.

13. A greedy algorithm is an algorithm reaching locally (in each step) for the most optimal choice in order to approach the best (i.e. the most optimal) global choice [3]. (However, some greedy algorithms are heuristic, which means that the probability of finding the most optimal global solution is very high, but not guaranteed. Yet it is not the case of Prim's algorithm, since its correctness can be mathematically proven.)

2. Initialize the Prim tree T as vertex s.
3. Initialize the set of frontier edges for tree T as empty.
4. While Prim tree T does not yet span G
   (a) Update the set of frontier edges for T.
   (b) Let e be a frontier edge for T with the smallest[14] edge-weight.
   (c) Let v be the non-tree endpoint of edge e.
   (d) Add edge e (and vertex v) to tree T.
5. Return Prim tree T.

(For a more formal definition of Prim's algorithm see [5], page 41.)

The question is how can this algorithm be altered in order to generate random mazes? The answer is easy. The alternative algorithm should abstract from weighted edges i.e. it should consider the graph to be unweighted and the random factor would be therefore implemented within the step 4. More accurately – the algorithm would not choose the most minimally weighted edge. It would choose any (random) edge directly connecting a vertex within the tree (a cell already being a part of the maze) with a vertex outside the tree (a non-tree vertex – a cell not being carved into so far).

Randomized Prim's algorithm can be implemented from two points of view: as a passage carver or as a wall adder. Darryl Nester mentions three variations of randomized Prim's algorithm [9]. Walter D. Pullen uses a method dividing maze cells into three groups ([10] – Maze Algorithms): in-cells (cells already being a part of the maze), out-cells (cells not being a part of the maze as yet), frontier cells (cells not being a part of the maze yet but having at least one in-cell as a neighbor):

1. Start by picking a cell, making it "in", and setting all its neighbors to "frontier".
2. Proceed by picking a "frontier" cell at random, and carving into it from one of its neighbor cells that are "in".
3. Change that "frontier" cell to "in", and update any of its neighbors that are "out" to "frontier".
4. The Maze is done when there are no more "frontier" cells left (which means that there are no more "out" cells left, so they are all "in").

Finally, the pseudocode from Maze Generation Java applet related to this master's thesis:

1. Choose a random cell within the maze grid (given by its width and height) and design it as a start cell.

14. Substituting the word "smallest" with "highest" would result in creating a maximum spanning tree instead of a minimum spanning tree.

2. Add the start cell to (by now empty) *inCells* set.

3. Mark cells around the start cell as frontier, i.e. add them to *frontierCells* set.

4. While *frontierCells* set is not empty:

   (a) Choose a random frontier cell *cF* from *frontierCells*.

   (b) Choose a random in-cell *cI* adjacent to *cF*.

   (c) Add *cF* to *inCells*.

   (d) Mark all out-cells around *cF* as frontier.

   (e) Add a path between *cI* and *cF* to the maze paths set.

   (f) Remove *cF* from *frontierCells* set.

For Java source code see Example C.0.4 and for illustration see Figure 2.2.

When implementing this algorithm, only two data set structures storing cells are needed: a set of in-cells (*inCells*, grey color) and a set of frontier cells (*frontierCells*, yellow color). This alternative has two (optionally even three) randomizations. The first (one-shot only) randomization may consist in choosing an arbitrary vertex to start the algorithm from (step 1). Another randomization occurs repeatedly when choosing a random frontier cell (step 4a). And finally the third randomization (step 4b) happens (again repeatedly) when choosing an in-cell the selected frontier cell should be connected with (in case the selected frontier cell has more than one neighboring in-cell).

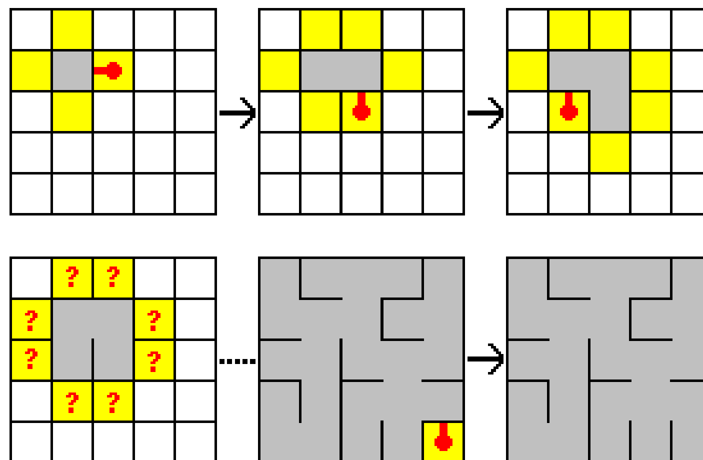


Figure 2.2: Randomized Prim's algorithm example

2.3.2 Randomized Kruskal's algorithm

Randomized Kruskal's algorithm used for maze generation purposes originates in Kruskal's algorithm. Joseph Bernard Kruskal, Jr. was an American (not only) mathematician and computer scientist. He first presented this algorithm in 1956.

Kruskal's algorithm is – as well as Prim's algorithm – a greedy algorithm able to find a minimum (or maximum) spanning tree within a graph. The difference between Prim's and Kruskal's algorithm is in the "area" of graph the algorithm operates in. As described in previous chapter, Prim's algorithm builds the spanning tree from one point up in an expansible way. On the other hand Kruskal's algorithm operates in the whole graph area creating and binding fragmental acyclic trees to each other (thus dealing with a forest). Eventually adding the last edge creates a spanning tree.

Alan Gibbons' interpretation of Kruskal's algorithm ([5], page 63):

1. Relabel the elements of E so that: *if* $w(e_i) > w(e_j)$ then $i > j$
2. $MWT \leftarrow \emptyset$
3. *for i = 1 to* $|E|$ *do*
   (a) *if* $MWT \bigcup \{e_i\}$ *is acyclic then*
   (b) $MWT \leftarrow MWT \bigcup \{e_i\}$

Commentary: MWT stands for minimum-weight spanning tree. Relabeling of elements of *E* means ordering the list of edges according to their weight. The third step represents iteration through the ordered list. If it starts with the smallest number, the algorithm returns the minimum spanning tree. If it starts with the highest number, the algorithm returns the maximum spanning tree.

How can one alter this algorithm in order to be usable for maze generation? Again (since the same holds for Prim's algorithm), the randomized version needs to abstract from graph edges being weighted. Although every wall has got to be processed, they are chosen in a random order. Choosing a wall means potentially creating a path between cells the wall divides. Unless there is another path between the two cells, the currently chosen wall is removed and the path is created (otherwise removing the chosen wall would result in a loop, which is highly undesired).

Darryl Nester presents his version of pseudocode of the randomized Kruskal's algorithm [9]:

1. Make a list of every available wall.
2. Choose a wall and remove it from the list.
3. If there is no path between the two cells on either side of that wall, remove it.
4. If all cells are connected, we are done; otherwise, return to step 2.

Step 3 is very intuitive at first sight, but keeping track of the actual cell relationship situation requires a specific solution. Walter D. Pullen's pseudocode ([10] – Maze Algorithms) is more specific and indirectly mentions sets for purposes of storing the forest system within the graph:

"Label each cell with a unique id, then loop over all the edges in random order. For each edge, if the cells on either side of it have different ids, then erase the wall, and set all the cells on one side to have the same id as those on the other. If the cells on either side of the wall already have the same id, then there already exists some path between those two cells, so the wall is left alone so as to not create a loop."

Finally – the pseudocode implemented within the Maze Generation Java applet (for Java source code example see Example C.0.5):

1. Initialize a list of tree sets, starting with each cell being in its own set.
2. Choose a random wall from the wall list and interpret both cells it divides.
3. If cells are not in the same tree set yet, create a path between these two cells and join their tree sets.
4. Remove the wall from the wall list
5. Unless the wall list is empty, loop back to step 2.

Eventually there is just one tree set left when the algorithm finishes. It contains the whole spanning tree corresponding to the resulting maze. Some sources recommend using disjoint-set data structure in order to operate on sets efficiently.

Figure 2.3 illustrates the randomized Kruskal's algorithm. Yellow colored wall is a wall which is randomly picked at the moment when it cannot be removed anymore (in order to prevent creating a loop).

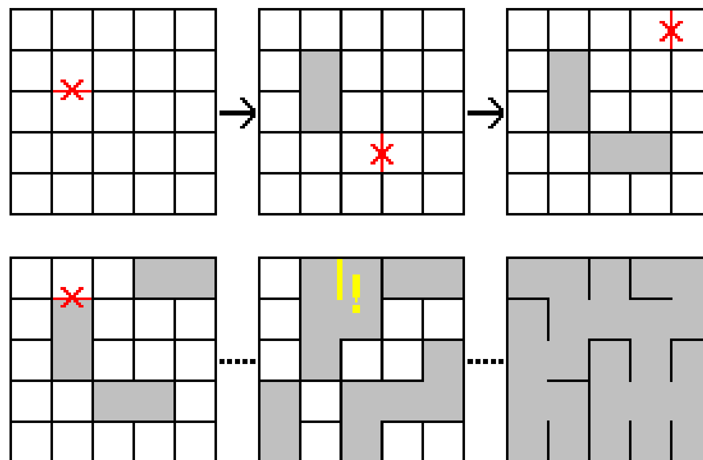


Figure 2.3: Randomized Kruskal's algorithm example

2.3.3 Recursive Backtracker

A recursive backtracker is a maze generation algorithm based on the so called depth-first search (DFS) technique. Graph search is not the only application of the DFS algorithm[15], that is why "search" is slightly misleading in its name. Yet there is a general adherence to this naming convention.

The DFS algorithm wanders through the graph in a depth oriented way. It means that if possible, the next visited vertex is a child of the previously visited vertex. Otherwise, in case the latest visited vertex has no more "available" children, the algorithm backtracks to the previously visited vertex and tries again. It is done when all vertices have been visited by the algorithm (regardless of the actual purpose of wandering through the graph). In case the algorithm builds a tree along the way (by registering and storing visited graph vertices and graph edges), a spanning tree is eventually returned.

The following pseudocode [6] formalizes the DFS algorithm:

Input: a connected graph G and a starting vertex v.
Output: a depth-first spanning tree T and a standard vertex-labeling of G.

1. Initialize tree T as vertex v.
2. Initialize the set of frontier edges for tree T as empty.
3. Set *df number(v) = 0*.
4. Initialize label counter *i:=1*.
5. While tree T does not yet span G
   (a) Update the set of frontier edges for T.
   (b) Let e be a frontier edge for T whose labeled endpoint has the largest possible *df number*.
   (c) Let w be the unlabeled endpoint of edge e.
   (d) Add edge e (and vertex w) to tree T.
   (e) Set df *number(w) = i*.
   (f) $i := i + 1$.
6. Return tree T with its df numbers

15. The DFS is not only an automated maze generation method, but it can also serve as an automated maze solving method (which is however out of this thesis' scope). See for instance [6], page 139.

Commentary: df number identifies the order the vertices are visited in. It can be observed, that if *"df number(x) < df number(y), then vertex x must be an ancestor of vertex y in tree T"* [6].

Recursive backtracker algorithm works in a similar way. It carves from the start point throughout the whole graph, using backtracking when necessary. The important aspect is randomization. The first randomization step occurs when a random cell is chosen to start the algorithm from. Another randomization happens when choosing a neighbor (child vertex) to carve into from the current cell.

The pseudocode from Darryl Nester's point of view (he refers to this algorithm as "random walk" [9]):

1. Start in a randomly-chosen cell.
2. If the current cell has at least one unvisited neighbor, choose one at random, knock down the walls between the current cell and the new cell, move into the new cell, and repeat step 2.
3. If the current cell has no unvisited neighbors, backtrack into the cell from which we "broke into" the current cell, then return to step 2. The maze is finished when we backtrack into the cell in which we started.

Walter D. Pullen ([10] – Maze Algorithms) goes a bit further with his declaration and mentions an effective way of implementing such an algorithm using a stack:

"When carving, be as greedy as possible, and always carve into an unmade section if one is next to the current cell. Each time you move to a new cell, push the former cell on the stack. If there are no unmade cells next to the current position, pop the stack to the previous position. The Maze is done when you pop everything off the stack."

Stack is also the data structure used within the Maze Generation Java applet:

1. Design a randomly chosen cell as the current cell and add it to the stack.
2. IF the current cell has any unvisited neighbors THEN
   (a) randomly choose one of them and design it as the next cell
   (b) add next cell to the stack
   (c) create path between current cell and next cell
   (d) mark next cell as the current cell
3. ELSE pop a cell from the stack and design it as the current cell
4. WHILE stack is not empty
   (a) loop back to 2

The first move is performed outside the loop in order to initialize the stack, so that the while-condition can check for finalization of the maze creation. See Example C.0.6 for Java source example and Figure 2.4 for illustration. Yellow points depict cells where the new walk started from.

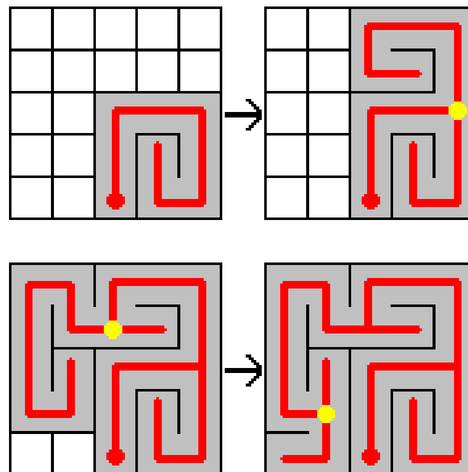


Figure 2.4: Recursive backtracker algorithm example

2.3.4 Hunt and Kill Algorithm

The Hunt and Kill algorithm is a modification of the recursive backtracker. However it cannot be logically compared to the DFS anymore, since its randomness breaks the basic principle of the DFS – the backtracking. When the algorithm reaches a cell with no more available unvisited neighboring cells to carve into, it does not backtrack to the previous cell. Instead a random already visited cell with available unvisited neighbors is found and the carving starts from this point again. Walter D. Pullen describes this state as the so called "hunt-mode" ([10] – Maze Algorithms). He describes the whole maze creation process using this algorithm as follows:

"It's most similar to the recursive backtracker, except when there's no unmade cell next to the current position, you enter hunting mode, and systematically scan over the maze until an unmade cell is found next to an already carved into cell, at which point you start carving again at that new location. The maze is done when all cells have been scanned over once in hunt mode."

Implementation of this algorithm within Maze Generation Java applet can be paraphrased with the following pseudocode:

1. Choose a random cell to start from, design it as the current cell and mark it as visited.

2. While not all cells have been visited:

(a) IF current cell has unvisited neighbors THEN
   i. randomly choose one of them and design it as the next cell
   ii. create path between the current cell and the next cell
   iii. mark the next cell as the current cell
   iv. mark the (newly assigned) current cell as visited
(b) ELSE
   i. randomly choose an already visited cell with available unvisited neighbors

See Example C.0.7 for a Java source code example and Figure 2.5 for illustration. Yellow points depict cells where the new walk started from.

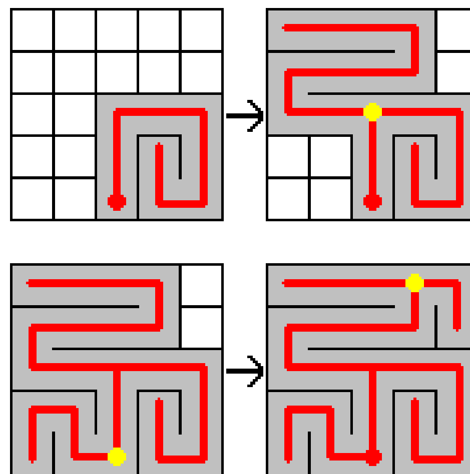


Figure 2.5: Hunt and Kill algorithm example

2.3.5 Bacterial Growth Algorithm

The bacterial growth algorithm is very similar to the principle of Prim's algorithm. Both build the spanning tree from one point up. The difference is in the "intensity" of adding new edges and vertices to the tree. Prim's algorithm adds one edge per each step of the loop, whereas bacterial growth algorithm adds an edge (and the corresponding vertex) to each cell with available (yet unvisited) neighbors. The number of visited cells can therefore be (at most) doubled after each step. This is where (most likely) the name of the algorithm comes from – bacteria population usually increases in an exponential way.

The pseudocode of such an algorithm can be stated as follows:

1. Design a randomly chosen cell as the current cell and mark it as visited.

2. Unless all cells have been visited:

(a) For each cell marked as visited:

(b) IF such a cell has any unvisited neighbors THEN

i. Randomly choose one, carve a path into it and mark it as visited

However this is not very efficient version of this algorithm. Creating, using and iterating only a set of already visited cells with unvisited neighbors would be more appropriate. See Example C.0.8 for a Java source code example and Figure 2.6 for illustration.

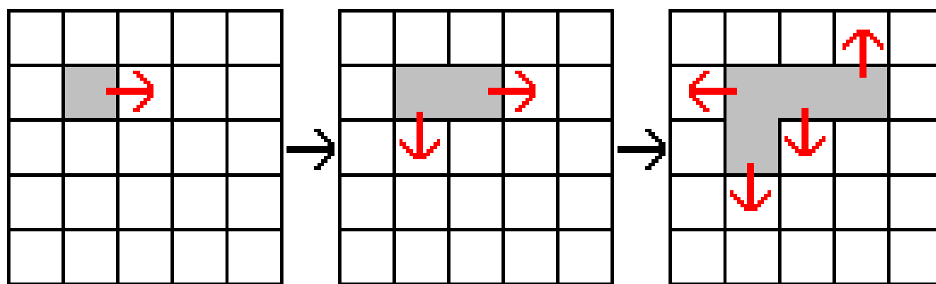


Figure 2.6: Bacterial growth algorithm example

2.4 Other Approaches of Maze Generation

There are also some other maze generation techniques worth being mentioned at least briefly. However inserting them into this subchapter does not mean they are not connected with graph theory at all. Logic principles of some of them do originate or are somehow connected with graph theory, whereas the terminal structure of all of them can be represented by some kind of a graph structure.

Eller's Algorithm

Eller's algorithm is very interesting because it is one of the fastest algorithms for maze generation purposes and it is also highly efficient with memory allocation. The principle is that only one row is being processed at once.

Walter D. Pullen ([10] – Maze Algorithms): *"It creates the Maze one row at a time, where once a row has been generated, the algorithm no longer looks at it. Each cell in a row is contained in a set, where two cells are in the same set if there's a path between them through the part of the Maze that's been made so far. This information allows passages to be carved in the current row without creating loops or isolations. . . . Creating a row consists of two parts: Randomly connecting adjacent cells within a row, i.e. carving horizontal passages, then randomly connecting cells between the current row and the next row, i.e. carving vertical passages."*

Wilson's Algorithm

Wilson's algorithm is another random walk based algorithm eventually creating a maze structure corresponding to a spanning tree. A random walk is performed between two randomly chosen cells. Then a third cell (outside of the already carved path) is chosen as a starting point for another random walk, which finishes when it hits a cell that is already a part of the maze. This cell thus becomes a root of a newly created subtree within the final spanning tree.

Darryl Nester [9] provides an intuitive solution, if an undesired loop is created during the random walk:

1. Choose a random cell, and mark it as "used".
2. Start in a randomly chosen unused cell (call it "START").
3. Do a random walk – keeping track of the path taken – until you stumble into a "used" cell (call it "END").
4. Remove walls along the path from START to END. If any cells on that path were visited more than once during the walk, only remove the wall in the most recent direction. Mark all cells on that path (including START) as "used".
5. If unused cells remain, return to step 2.

Recursive Division

The recursive division algorithm is a typical wall adding algorithm. Its connection with graph theory is not as obvious as by some other previously mentioned algorithms. Yet eventually a graph can be used to depict the final maze structure.

The algorithm starts with dividing an empty maze area horizontally or vertically, *"with an opening randomly placed along it. Then recursively repeat the process on the two subareas generated by the dividing wall. For best results, give bias to choosing horizontal or vertical based on the proportions of the area, e.g. an area twice as wide as it is high should be divided by a vertical wall more often."* [10]

Chapter 3

Implementation

A subtask of this master's thesis was to implement an application with the following basic features:

- The user can have a maze created using several algorithms.
- Mazes can be watched when being created, so that the differences between algorithms can be clearly observed.
- The user can interactively solve mazes.
- The application gathers data collected from users for further analysis.

In order to enable a potentially big amount of users to play and solve mazes, the application needed to be provided online. Several aspects needed to be considered: friendliness of UI design, available IDEs, behavior at the client's side (necessary web browser components, virtual machine behavior and efficiency), multithreading features (which eventually showed to be very important), etc. Since I had no previous experience with any alternatives (e.g. Flash), I chose Java applet technology.

When designing back end implementation, I had to take available technologies into consideration. The Faculty of Informatics, Masaryk University[1] provides its students (not only) with MySQL accounts. Because of the faculty's network restrictions and other potentially arising problems at the client's side, I have implemented server side PHP scripts to store and operate the data in the database based on requests sent by the applet. It showed to be a good and quite efficient solution for such a small sized project.

See Appendix E for electronic attachment description.

3.1 Back End

Data analysis regarding Chapter 4, "Analysis of Human Maze Solving" will be performed on user data, which consists of information about each attempt to solve a maze. The following information is connected with each solve attempt and can be retrieved from the MySQL database for analysis purposes:

1. http://www.fi.muni.cz

- Username – unique user identifier.
- Timestamp – time identifier.
- Maze finished – whether user has successfully solved the maze or not.
- Show visited cells – whether the option to show already visited cells was enabled during the solve attempt or not.
- Maze type – name of the predefined maze or name of the randomized algorithm which created the random maze.
- Maze size – size of the maze (maze width and maze height).
- Maze paths – set of all pairs of cells representing a valid path stored as a string with specific structure.
- Start cell – starting point of the maze.
- End cell – end point of the maze.
- User draw – list of cells ordered chronologically as the user visited them during the solving phase (including the time information for each turn) stored as a string with specific structure.
- Time – time in seconds.

An attempt is represented by a record, which can be eventually in 3 various states (see Figure 3.1):

- Initialization – user started solving but he neither clicked the "Give up" button nor did he solve the maze successfully *(win_time = -1 AND maze_finished = false)*
- User won – user solved the maze successfully *(win_time > -1 AND maze_finished = true)*
- User gave up – user gave up and clicked the "Give up" button *(win_time > -1 AND maze_finished = false)*

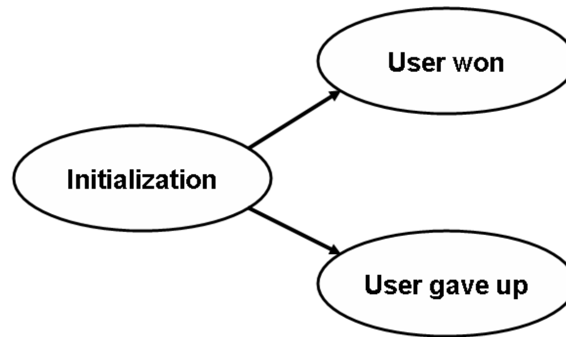


Figure 3.1: Database record states.

3.2 Front End

Maze Generation Java applet[2] was developed using NetBeans IDE 6.9.1 and compiled by Java 6 update 22. Applet enables user (solver) to:

- solve predefined mazes (using mouse or keyboard arrow keys, but exceptions may apply according to the predefined maze characteristics)
- have a maze created using various randomized algorithms
- watch the process of maze creation (or skip the watching phase and have it created and shown immediately) – applies to non-predefined mazes only
- have a maze exported to a bitmap file for further processing (printing, etc.)

Predefined mazes are static mazes with hard-coded properties (width, height, starting cell and end cell position, ability to see already visited cells while solving) and path structure. The point is to provide a bunch of identical mazes to a mass of users in order to increase the results' objectivity of human interaction analysis.

3.2.1 User Interface

At the moment (January 2011), application UI contains following controls:

- Slow motion checkbox – if checked, maze is created "live" in order to point out the characteristics of used algorithm (applies to non-predefined mazes only).
- Slow motion slider – adjusts the speed of the slow motion option.

2. http://www.fi.muni.cz/~xfoltin/mazes

- Generate button – starts generating the maze (if slow motion checked, it can be also used to skip the slow motion generation and show the maze immediately).
- Give up button – user gives up current solve attempt by clicking this button.
- Size combobox – shows the selected desired maze size (applies to non-predefined mazes only).
- Maze list – list of available mazes to choose from (predefined and non-predefined ones).
- Menu bar:
  - Actions / Export current maze – exports current maze to a bitmap file for further processing.
  - Options / Show visited cells – enables solver to see already visited cells while solving (usually applies to non-predefined mazes only, but there may be exceptions).
  - Help – contains Help and About dialog windows.

3.2.2 Relevant Classes and Data Structures

MazeCanvas

MazeCanvas extends Java AWT class Canvas. A Canvas component represents a blank rectangular area of the screen onto which the application can draw or from which the application can trap input events from the user (keyboard arrows and mouse clicks). MazeCanvas is the application component mazes are drawn onto.

Maze

A maze class represents a maze with its basic attributes: maze type (string description of its origin), width, height, start cell, end cell, set of paths.

Cell

A cell is a basic maze unit. It is characterized by its x-coordinate and y-coordinate (see Example C.0.1). The coordinates of the upper left cell in each maze are [0, 0]. X-coordinate represents the horizontal dimension of a maze, i.e. its width. Y-coordinate represents the vertical dimension of a maze, i.e. its height. Cell class is a supertype of following classes: CellIn (start cell of the maze), CellOut (end cell of the maze), CellWithTiming.

CellWithTiming (extends Cell) is a structure representing a cell including the information of time (in seconds, since the start of the solving phase) when solver visited the cell (see Example C.0.2). This enables the backtracking, replaying and following of a maze solve attempt for eventual analysis.

Path

A path is a structure consisting of two cells denoting the fact there is a connection between them (i.e. one can carve from one to the other and vice versa).

Pathset

A pathset is a *java.util.HashSet* structure each algorithm uses to store paths in. It is a convenient way to archive the maze paths structure and it enables the programmer to easily check presence or absence of a valid path within a maze using the *contains(Object o)* method.

MazeCreator

The MazeCreator class' aim is to provide a maze after the user has chosen his desired way of having it created (predefined maze or maze created by a randomized maze generation algorithm).

Randomized Algorithms Classes

Each randomized maze generation algorithm is implemented within its own class extending *java.lang.Thread* class. The multithreading feature is critical for user friendliness and efficiency of this application, since concurrently running subtasks enable for instance skipping the maze creation during the slow motion process. Multithreading is also necessary when contacting PHP scripts and sending data to the database server.

MethodUtil

MethodUtil is a class providing static methods frequently and commonly used by maze generation algorithms. A typical example is the *getRandomCell* method returning a random maze cell provided it knows the size of the maze (see Example C.0.3).

Chapter 4

Analysis of Human Maze Solving

4.1 Human Problem Solving

ProSo[1] is a research group at the Faculty of Informatics, Masaryk University studying various aspects of *"cognitive science – an exciting intersection of computer science, psychology, artificial intelligence and education."* According to one of ProSo's publications [12], an extensive study of human problem solving is important, because *"humans and computers solve problems in different ways and have complementary strengths; to build tools for human–computer collaboration we need to understand what makes problems difficult for humans."* ProSo researchers find inspiration for human problem solving analysis in various puzzle games: Sudoku, Sokoban or Replacement puzzle. See for instance [2], [7] (in Czech).

Problems from the human problem solving point of view can be categorized according to the complexity of their definitions into two groups: well-structured and ill-structured problems. Well-structured problems are (relatively) clearly defined and structured, whereas ill-structured problems are vaguer. The maze solving problem – as a logic puzzle problem – can be defined as well-structured. The basic principle and how-to-play is very intuitive and the problem can be observed relatively easily. The so called game state and state space are used for this purpose.

Concerning maze solving – game state is a possible state of game represented by the solver being positioned on one of the maze cells. State space is an undirected graph containing vertices representing all possible game states. An edge between vertices x and y means there is a direct transition from state x to y and vice versa. Since a maze state space is generally a graph, a perfect maze state space is then a spanning tree (which connects this topic again with graph theory concepts). Number of vertices in such a spanning tree equals to the number of cells in its corresponding maze. Figure 4.1 shows an example of a simple maze state space with the starting cell designed as the root of the tree (the starting cell is green and the end cell is red).

1. http://www.fi.muni.cz/~xpelanek/proso

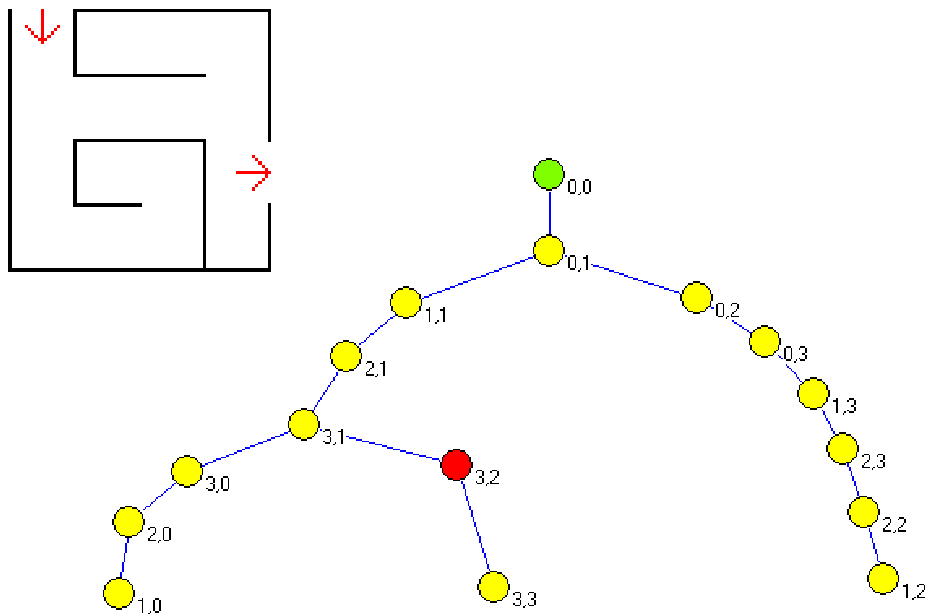


Figure 4.1: A state space example

4.2 Maze Solving

4.2.1 General Assumptions

Mazes can be generated and/or solved in a both automated and non-automated way (human in our case). The most interesting combination is automated generation vs. human solving or vice versa, since these two combinations show differences between humans and computers and their perception of a problem (generation or solving).

There are many aspects that influence the difficulty of the maze and the time needed to solve it. Walter D. Pullen deals also with this topic ([10] – Maze Psychology), yet focusing especially on life size mazes. There are many ideas helpful for our purpose as well, though.

A very typical approach to solve a maze is so called wall following ([13], [10] – Maze Psychology), but it is helpful only in some cases. Its principle lies in “following” either the left or the right wall, which leads into finding the solution. If the maze is infinite or if it contains loops, this technique might/will not work. Loops generally influence the solving phase a lot. If there are only dead ends in the maze (the case of all perfect mazes), the solver can find out relatively easily if he is on the wrong way. Loops can make the solver believe he is heading successfully towards the end cell.

Another aspect influencing maze difficulty is its size. The length of the solution path or complexity of maze tessellation and eventually the cell shape are relevant as well. As already mentioned, many categories (see Section 1.2) influence maze solving difficulty variously.

4.2.2 Brute Force vs. Analytical Solving

A solver's thinking and behavior is very vague and subjective but also a very interesting aspect worth observing, analyzing and studying. Let us assume and experimentally distinguish two basic approaches: brute force solving and analytic solving. The brute force approach should be observable by fast movement throughout the maze area, especially (but not only) directly towards the end cell. In case the solver happens to find and traces the solution path soon enough, it is a very fast approach. Analytic solving is an approach where the solver is able to stop for a moment and analyze a part of the maze (or even the whole maze) in order to find the solution path. It can also help to rule out a possible dead end. One can assume such an approach may be more helpful in large-sized mazes that are more likely to take longer to solve using the brute force method. Both methods seem to have their advantages and disadvantages.

Let us take a closer look at the difference between brute force and analytical solving. Figure 4.2 and Figure 4.3 show the distribution of a number of moves over time (do not confuse with the distribution function in probability theory; let us refer to it as the moves-over-time function). The move count is assigned to the interval of one second, which can be then perceived as a discrete variable. The last second is the moment when the solver carved into the end cell.

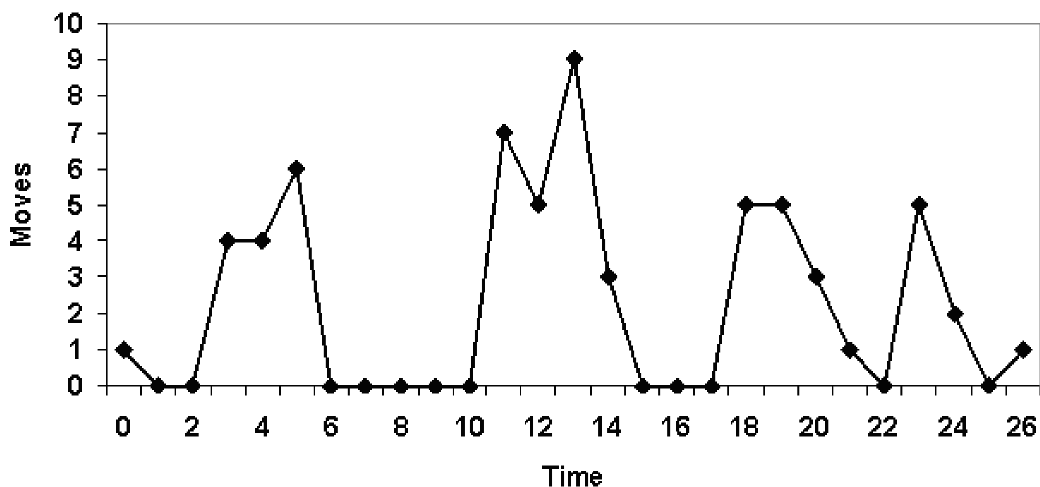


Figure 4.2: Analytic Example

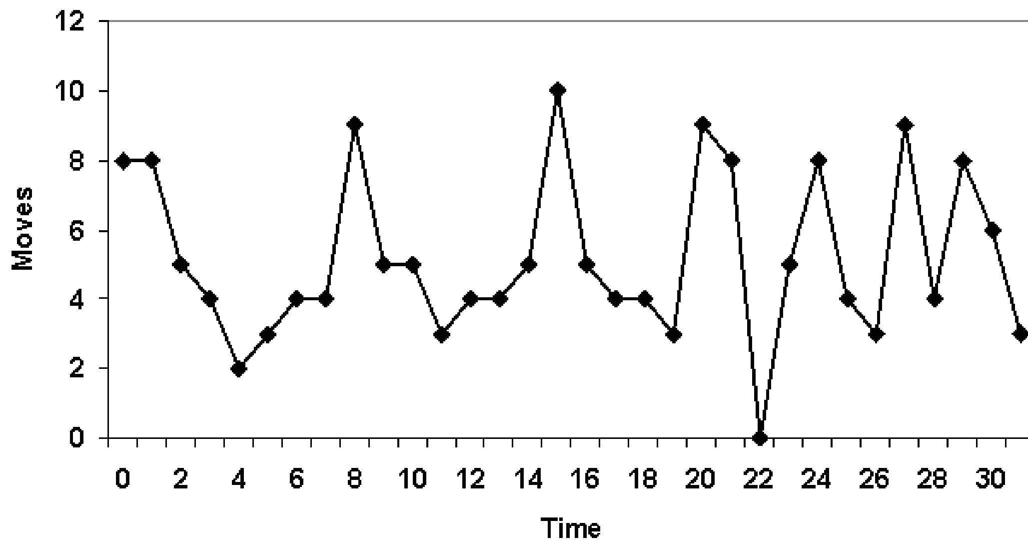


Figure 4.3: Brute Force Example

Both solving examples were synthetically produced when simulating assumptive user behavior. Let us observe some basic differences between both approaches first. The analytic example contains many time intervals without any moves, which could mean that the user stopped and analyzed the surroundings of his current position. (Of course that is just an assumption. He could have been disturbed or simply could have stopped for many other reasons.) Another interesting fact is that immediately after the last no-move interval (intervals 2, 10, 17, 22) the number of moves rapidly increases during the next interval. It could mean that the solver analyzed successfully and followed a path he was sure about (and that is why he moved faster).

On the other hand the assumption regarding brute force solving is based on high intensity and fast steady movement of the solver through the maze area. Time intervals with a low number of or no moves could be explained by the solver running into a dead end, which made him stop and analyze the situation (compared to an analytic oriented solver, who analyzes "voluntarily").

Both example mazes are medium sized. Despite the fact that they were different, let us calculate the average count and standard deviation of the number of moves throughout the time (Table 4.1). It might be helpful and illustrate the way I might study mazes in further subchapters.

	Average	Standard deviation
Brute force	5.19	2.38
Analytic solving	2.26	2.62

Table 4.1: Examples' statistics

A lower average number of the analytic solving example proves it to be more moderate and not so "aggressive". Standard deviation of the brute force example is lower despite the fact that its average is more than doubled. Ergo a relatively high standard deviation of analytic solving mirrors a high frequency of time intervals without any moves.

When analyzing predefined mazes and individual users' behavior, I'll try to prove (or reject) the assumption, that solvers usually use one of these two basic approaches (brute force or analytic solving).

4.3 Data Collection

Note: shortly after publishing the applet a bug regarding the movement evidence was revealed. Therefore numbers concerning successfully solved mazes come from the whole data sample, but any previous or upcoming data regarding moves count (average number of moves etc.) come from the part of users' data (approx. 82 % of all data) which was obtained after the bug has been fixed. Let's refer to these periods as applet version 1 (before the bug was fixed) and applet version 2 (after it has been fixed).

The gathering of user data took for over a month (43 days). It was quite essential for me to convince users to register and solve mazes while being logged in under their unique username. The point is for me to be able to study behavior of a specific person (regardless of his/her identity and characteristics). About three quarters of all mazes were solved by registered users. To increase users' motivation I organized a competition. (A prerequisite to enter the competition chart was to have all solvable predefined mazes solved.) One could gain 1 point for successfully solving a medium sized maze and 2 points for solving a large sized maze. Competition results have been evaluated and top 3 players were given a prize. According to the final standings, I suppose the competition did not really invoke as much interest as I expected however I am satisfied with the eventual extent of users' data and I find it sufficient for my exploratory data analysis. Table 4.2 contains some basic information about the whole data gathering process. Table 4.3 contains maze size measurements which are referred to in following subchapters.

Number of users who successfully solved at least one maze	48
Time spent by reg. users (successful and unsuccessful attempts)	24.24 hours
The most active reg. user spent	6.35 hours
Anonymous users spent (all together)	4.48 hours
Successfully solved mazes (all together)	1733

Table 4.2: Global statistics

Maze size	Width (cells)	Height (cells)	Total number of cells
Tiny	7	7	49
Small	15	15	225
Medium	37	25	925
Large	60	40	2400

Table 4.3: Global statistics

See Table D.1 and Table D.2 for global randomized and predefined mazes' statistics.

4.4 Data Analysis

4.4.1 Analyzing Tools

The following tools are used to express or visualize the statistical sample data:

- line charts (e.g. visualization of moves-over-time function)
- tables
- maze screenshots with various illustrations
- state space graphs
- maze density maps

State space graphs were created using Pajek[2] software, which is a highly powerful tool for graph visualization. When looking at such a figure, note that: the starting cell is green, the end cell is red, the rest of the cells is yellow; when focusing on (shortest or user's) solution path as well, cells which are a part of it are marked blue.

A maze density map (e.g. see Figure A.5) depicts the frequency of users' landing on a specific cell, i.e. the density map is usually created according to a sample containing several (>1) records. Density map can be imagined as a scatter plot representing the maze area. Each

2. http://pajek.imfm.si

cell would be depicted as a rectangular part of the area with dots; each dot representing one user landing onto this cell. Density map considers only the fact whether the user visited the cell or not; moves count involving this cell is not relevant. The more cell visits (i.e. the more dots), the higher the frequency is. On a density map the frequency is proportional to the darkness of the cell (see Figure 4.4). Cells being landed on not even once are white. Move evidence is needed for calculation of such a density map that is why only samples from applet version 2 are relevant.

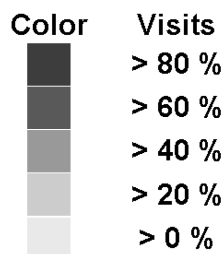


Figure 4.4: Density colors (100 % is the highest visits count per cell within the sample).

I derived and improved the Maze Generation applet in order to implement an analysis environment. It enabled me to "replay" any solve attempt of any user stored in the database. I also implemented export features within this environment, which enabled me to easily export maze screenshots, density maps and Pajek files necessary to create state space graphs.

Abbreviations, that are often used (and will be closer explained, if necessary) in following subchapters:

- *SA* – successful attempt(s)
- *SAC* – successful attempts count
- *AST* – average solution time
- *TT* – total time
- *SP* – solution path
- *SSP* – the shortest solution path
- *ANM* – average number of moves[3]
- *MNM*[4] – minimum number of moves

3. A move should be understood as a cell *c* where the user "lands" at a specific time *t*. (I.e. landing on the same cell in two different seconds counts as two moves.)
4. MNM of a maze is equal to the length of its SSP.

- *MPS* – moves per second

- *GLR* – got-lost ratio

- *AVG/MIN/MAX/STDEV* – average/minimum/maximum/standard deviation

4.4.2 Randomized Mazes Analysis

There were five randomized algorithms implemented at the moment of data gathering: randomized Prim's algorithm, randomized Kruskal's algorithm, recursive backtracker, hunt-and-kill algorithm and bacterial growth algorithm. Since I implemented bacterial growth algorithm only a few days prior to the evaluation, it will not be contained in the randomized mazes' data evaluation.

Implemented randomized algorithms always create a different pseudorandom maze, yet within the limitations of the size of the maze grid (i.e. within the limited number of cells). That is why if we want to analyze such user data, we might want to focus more on the global aspects, which we can find and study especially using statistical functions.

Some basic points are already revealed by Table D.1. We can observe that regardless of the number of successfully solved mazes per algorithm, the average time and the average number of moves increase with the size of the maze. It applies for the bacterial growth algorithm with less than 40 solved mazes as well as for the randomized Prim's algorithm with over 880[5] successfully solved mazes. A basic (yet a little bit generalized and relative) assumption is confirmed: the larger the maze, the more difficult it is (the more time is needed to solve it).

A proportional relationship between AST and ANM is quite intuitive and also shown in a graph (data from Table D.1) and by interpretation of Pearson product-moment correlation coefficient[6] (Figure 4.5).

5. The extremely high number of solved medium sized mazes created by Prim's algorithm has a reason in how competition rules were set. Users were awarded one point per each medium maze and two points per each large maze. Eventually users found out that a medium sized maze by Prim's algorithm was the fastest to solve, ergo the easiest way to gain points. Users obviously going for the prize winning positions mostly solved this kind of maze.

6. $r \in < -1; 1 >$: 1 – proportionality, -1 – inverse proportionality, 0 – no linear correlation

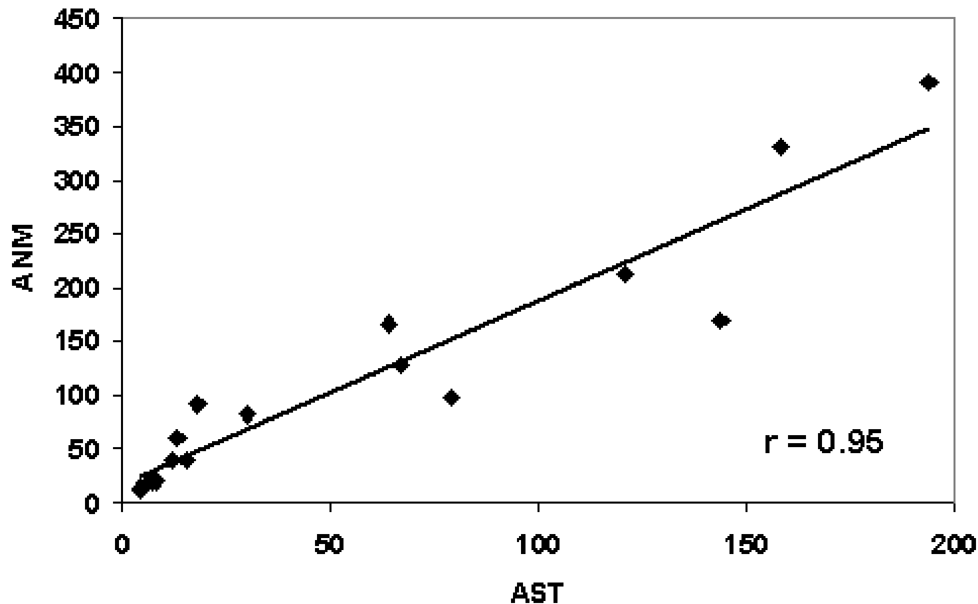


Figure 4.5: Randomized mazes: AST and ANM relationship

Users had an option to have all cells they had visited marked while solving any maze created by a randomized algorithm. However not more than 8.5 % of such attempts were started in this mode. There could be two main reasons for that: either the users did not know of the availability of such a feature, or they simply did not want the puzzle to be easier to solve (probably because of higher motivation). Humans usually do not like to be challenged with puzzles that are too easy or too difficult [12].

By altering the data composition from the base table, we can try to mine for some other data, which might help us compare algorithms among each other. (Tiny-sized maze results are excluded.) See Table 4.4.

Results are ordered descending according to the average solution time per maze size category. We can observe dominance of the DFS based algorithms (hunt-and-kill algorithm and recursive backtracker). This dominance is evident especially for large sized mazes, which accent the algorithm differences the most. (The higher the difference between the maze size category maximum and minimum is, the more relevant the sample appears to be). Comparison of statistics of Kruskal's and backtracker algorithm for medium sized mazes can be interesting as well. Despite a lower average solution time of the backtracker algorithm, the average moves count is higher. I interpret this as a consequence of a high river factor (and long dead end paths) of mazes created by recursive backtracker: when reaching a dead end, one needs to get back from where he got there, which is fast but increases the moves count quite rapidly.

On the other hand mazes created by the randomized Prim's algorithm eventually showed

	Allgorithm	Maze size	SAC	AST	ANM
1	Hunt And Kill	Large	6	194	392
2	Recursive backtracker	Large	3	158	331
3	Randomized Kruskal's	Large	14	144	170
4	Randomized Prim's	Large	29	67	129
1	Hunt And Kill	Medium	46	121	212
2	Randomized Kruskal's	Medium	18	79	99
3	Recursive backtracker	Medium	10	64	166
4	Randomized Prim's	Medium	861	30	82
1	Recursive backtracker	Small	54	18	92
2	Randomized Kruskal's	Small	9	15	39
3	Hunt And Kill	Small	8	13	61
4	Randomized Prim's	Small	9	12	40

Table 4.4: Randomized mazes' statistics

to be the least difficult to solve. As already mentioned users did notice that and took advantage of it when gaining competition points.

Walter D. Pullen summarized some maze characteristics ([10] – Maze Algorithms) into a table showing that the dead end count in the non-DFS based algorithms (Kruskal's and Prim's) is much higher than in the DFS based algorithms. According to results from Darryl Nester's applet [9], the dead end count of Prim's algorithm is approximately 2.7 times higher than the dead end count of a DFS based algorithm (both measured on the same maze size). Although DFS based algorithms have a relatively smaller count of dead ends, they tend to be longer (thus increasing the river factor of a maze). Short dead ends are easier to spot and avoid which accelerates the solving. Again – Darryl Nester applet's statistical feature proves the increased river rate of DFS based algorithms.

Maze difficulty is also influenced by the positioning of the starting cell and the end cell. Both cells may visually seem to be close to each other, but the path between them may be complicated and branched. Random mazes within Maze Generation applet have the starting and the end cell randomly positioned, but always opposite to each other.

However we can state that the river factor (parallel with its relationship to the dead end count) is the main cause of increased DFS based algorithms' difficulty comparing to non-DFS based algorithms.

4.4.3 Predefined Mazes Analysis

The point of creating and publishing predefined mazes was to have a data sample enabling a closer and more objective comparison of users' individual approaches when solving mazes.

All of them were static (unlike randomized mazes) and the same for everyone. To increase the results' objectivity, the basic sample data should be reduced. Table D.2 is based only on first attempts (per maze) of (only) registered users. Although it decreases the successful attempts count, I do believe such a sample is still good enough to perform an exploratory data analysis, which is one of the objectives of this thesis. (First 2–3 mazes have a higher successful attempt count most likely because they were the first ones to appear in the list without scrolling it down.)

Basic Observation

Table D.2 enables us to perform some basic observations. As expected - the smaller the maze, the lower the average solution time is. None of medium mazes took longer to solve than any of the two large mazes (at average). The same as with the average solution time applies for the average moves count, but with one exception – medium sized PredefinedMaze3. It has been created by the DFS based algorithm (recursive backtracker) and the starting cell and the end cell were positioned so that the solution path would fill up almost 40 percent of the whole maze area (see elitism column[7]). Ergo its average solution time is not the highest, but one needs to perform relatively many moves and pass many cells (in comparison to the maze size) to get to the end cell.

If we ordered the sample data by AST (and by maze size), we would find out – as expected – that non-DFS based algorithms (Kruskal's and Prim's) took the shortest time to be solved. Yet there is one exception – PredefinedMaze13 which was created by the recursive backtracker algorithm. It is related to PredefinedMaze3, which will be properly explained.

By deriving the sample data, we can also get another interesting observation. Let M_{avg} be maze's average moves count and let M_{min} be maze's minimum moves count necessary to solve the maze. Then let $(100 - (M_{min}/(M_{avg}/100)))$ be the got-lost ratio (GLR) which expresses the amount (in percent) of unnecessarily performed moves. Table 4.5 orders results according to GLR (ascending).

The largest examples (9 and 10) made users roam mazes the most. On the other hand PredefinedMaze3 and PredefinedMaze8 have the lowest GLR. It is related to their high elitism factor (i.e. a large portion of the maze is a part of the solution path, that is why it is "harder" to get lost). In general – inverse proportionality between elitism factor and GLR can be observed (see also Figure 4.6), which is proven also when interpreting the Pearson product-moment correlation coefficient.

7. Length of the shortest solution path compared to the maze size.

Maze type	Algorithm	ANM	MNM	Elitism	GLR
PredefMaze3	Recursive Backtracker	363	358	38.70 %	1.38 %
PredefMaze8	Recursive Backtracker	98	90	40.00 %	8.16 %
PredefMaze7	Kruskal's Algorithm	40	35	15.56 %	12.50 %
PredefMaze1	Prim's Algorithm	106	80	8.65 %	24.53 %
PredefMaze13	Recursive Backtracker	129	92	9.95 %	28.68 %
PredefMaze11	Prim's Algorithm	117	80	8.65 %	31.62 %
PredefMaze5	Hunt And Kill Algorithm	207	134	14.49 %	35.27 %
PredefMaze2	Kruskal's Algorithm	182	113	12.22 %	37.91 %
PredefMaze12	Kruskal's Algorithm	191	113	12.22 %	40.84 %
PredefMaze15	Hunt And Kill Algorithm	249	134	14.49 %	46.18 %
PredefMaze14	Hunt And Kill Algorithm	294	155	16.76 %	47.28 %
PredefMaze4	Hunt And Kill Algorithm	297	155	16.76 %	47.81 %
PredefMaze10	Kruskal's Algorithm	360	180	7.50 %	50.00 %
PredefMaze9	Prim's Algorithm	310	119	4.96 %	61.61 %
PredefMaze6	Kruskal's Algorithm	N/A	N/A	N/A	N/A

Table 4.5: Predefined mazes' statistics with got-lost ratio (GLR)

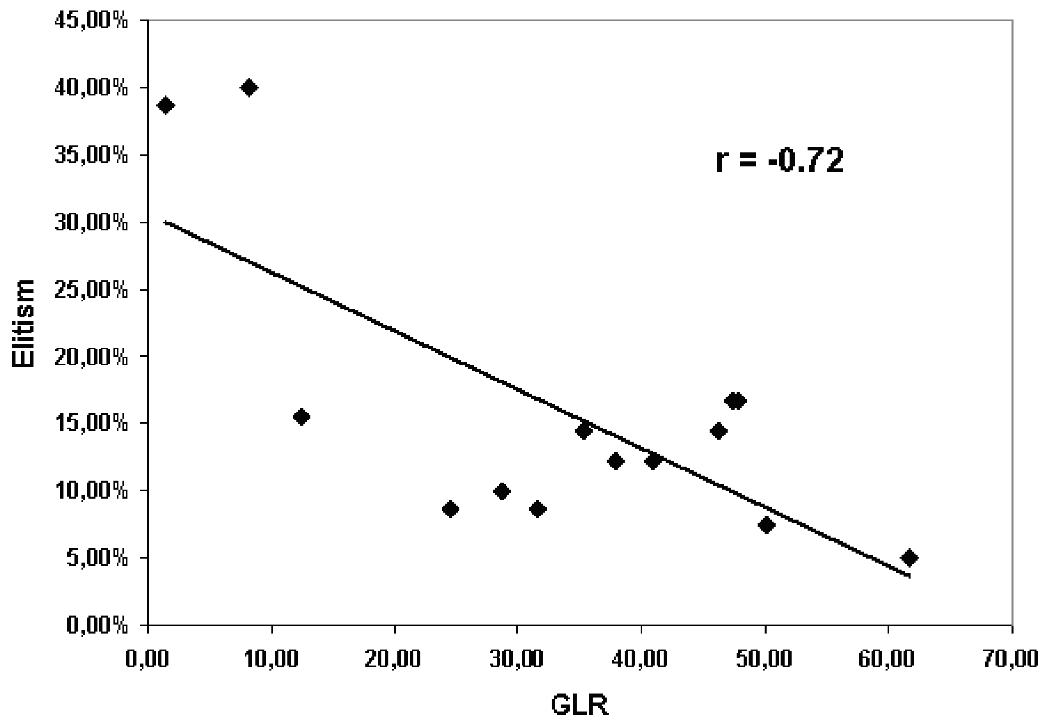


Figure 4.6: Predefined mazes: GLR and elitism relationship

Contextual Observation

Let us take a look at predefined mazes from the point of view I consider to be the most interesting within this experiment. It involves slightly altering the maze and studying the final impact on users' solving results. Some predefined mazes were coupled which means an original had its altered version. Such pairs involve predefined mazes 1–5 and their couples 11–15. Between mazes within each couple there is another 9 mazes to solve (in case users followed my request and solved them chronologically). This difference should make sure the solver forgets about previously solved mazes and perceives their couples as new mazes. Non-coupled solvable mazes (7–10) were created in order to compare various algorithms to each other. PredefinedMaze6 is a specific case. It is the only one without any solution (which was a priori not announced of course). It eventually showed to be a very interesting part of the experiment.

When comparing mazes and mentioning their statistical properties, usually Table D.2 and Table 4.5 are referenced.

PredefinedMaze1 and PredefinedMaze11

This is a maze pair with the highest successful attempt count (SAC) when summed. However the SAC of PredefinedMaze1 is almost three times higher as the SAC of PredefinedMaze11. PredefinedMaze1 was created by randomized Prim's algorithm. PredefinedMaze11 is the same maze, but with an extra 8 walls down which ergo create loops (since the original one is a perfect maze). Figure 4.7 shows the state space of PredefinedMaze1 (blue vertices belong to the SSP).

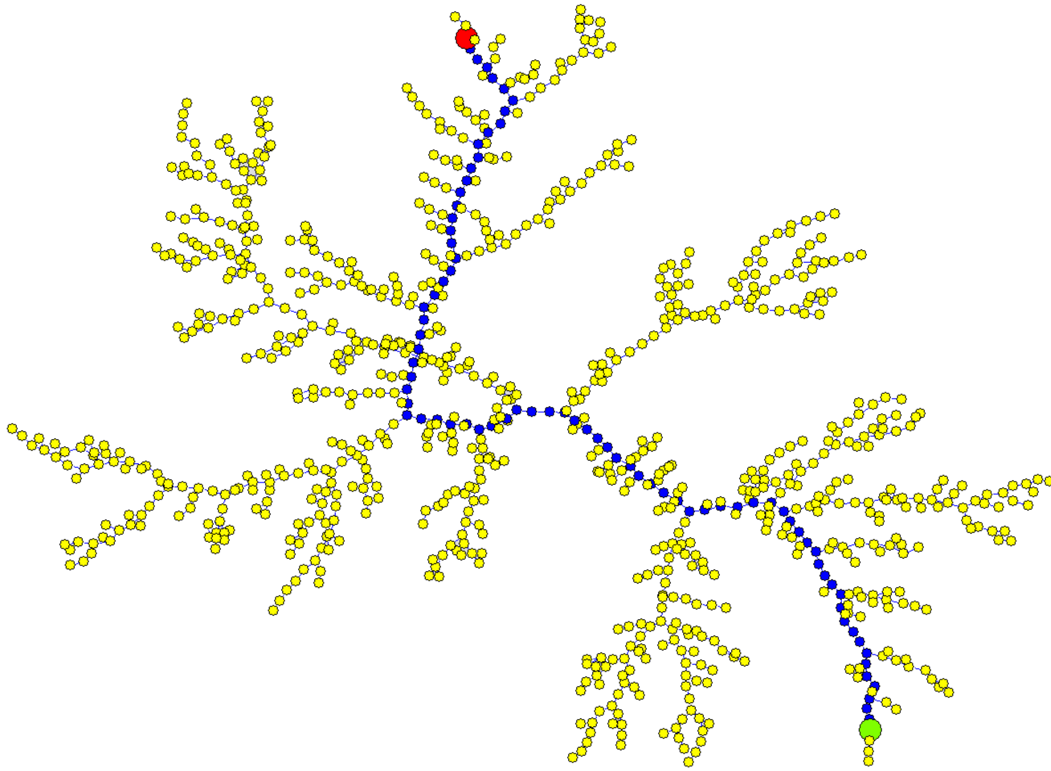
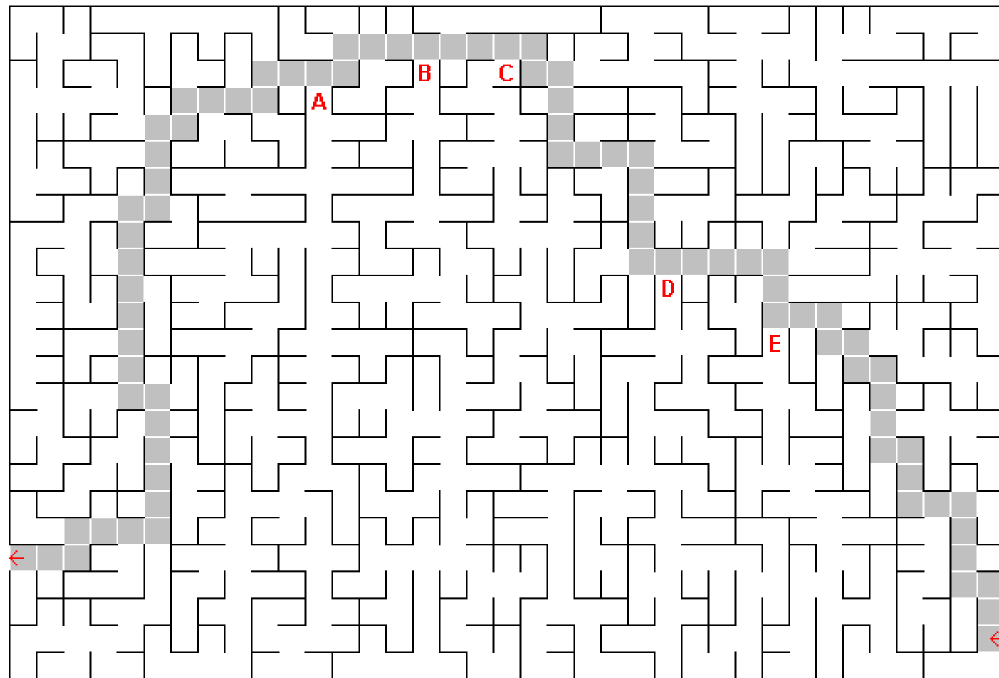
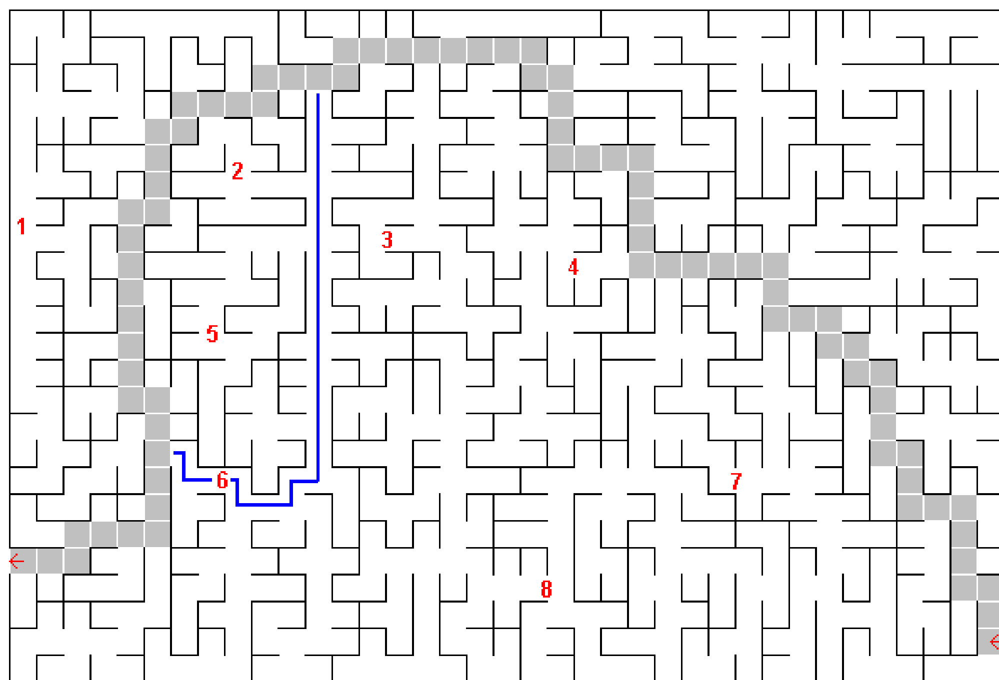


Figure 4.7: PredefinedMaze1 – state space

Figure 4.8 shows the original PredefinedMaze1 (a) with its shortest (and only) solution path and PredefinedMaze11 (b) with its shortest solution path (SSP), with the extra walls down (which are numbered) and with a (blue) often used alternative (will be explained).



(a)



(b)

Figure 4.8: PredefinedMaze1 and PredefinedMaze11

Approximately 27 users of PredefinedMaze1 followed the SSP with no or very short turnoffs. Letters A to E represent crossings where users tended to branch away from the SSP the most. Quite logically they almost never branched away to the right from the SSP. They branched away to the left with following frequency: A – 15 users, B – 7 users, C – 4 users, D – 16 users, E – 5 users (note that the mentioned sets of users are not necessarily disjoint). The closer to the top of the maze, the more attractive it became to turn off left.

2–3 users (out of 16) solved the PredefinedMaze11 exactly or in a very similar way to PredefinedMaze1, ergo they followed the shortest way. This makes me believe users actually did not notice they may have already solved a very similar (almost identical) maze. Removing wall no. 6 showed to have the greatest impact on solving PredefinedMaze11. The solution path of 12 users contained the blue cell sequence (check Figure A.4 and Figure A.5 for both density maps to observe the difference). Most of them actually copied the shortest solution path (SSP), went the "blue" way and then joined the SSP again. There was almost no looping. Only 2 people got trapped into a loop caused by walls 4 and 8.

The blue path increases the SSP only by 2 cells (however it is not SSP anymore). Since it is not really much of a difference, there must be some other motivation of using this alternative way. In my opinion it's a kind of psychological aspect that has been (indirectly) already mentioned: users see an approximately 15 cells long straight path leading towards the end cell, which makes them believe (or hope) it's a correct path.

The difference between average solution times (AST) of both mazes is quite high (26 seconds). The lower AST of PredefinedMaze11 might have been caused by users being more trained after solving several mazes, or simply by a smaller sample data. This may apply for other maze couples as well.

PredefinedMaze2 and PredefinedMaze12

PredefinedMaze2 was created by the randomized Kruskal's algorithm. Its couple – PredefinedMaze12 – has an extra 9 walls down. Figure 4.9 shows PredefinedMaze12 with its SSP (which is the same, since adding new passages had no impact). Numbers represent walls which were removed from the original PredefinedMaze2.

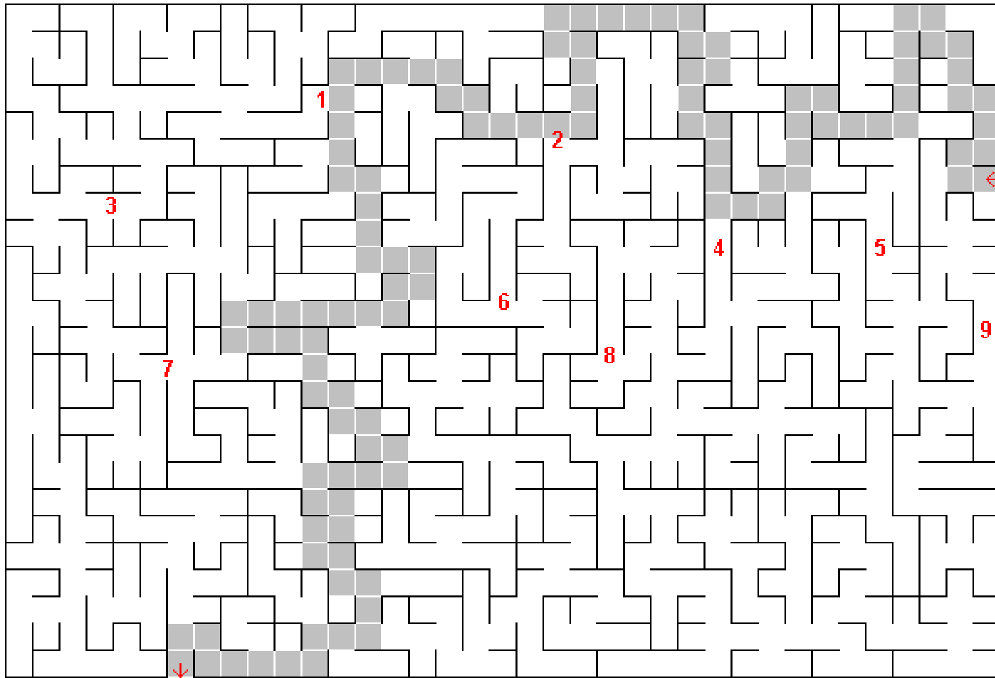


Figure 4.9: PredefinedMaze12

When we compare density maps of both mazes (Figure A.6 and Figure A.7), we come to the conclusion that the density map of PredefinedMaze12 obviously contains many more dark cells. Removing the walls increased the number of opportunities to roam throughout the maze (although the SSP remained unchanged), which is also mirrored in a higher average number of moves (ANM) of PredefinedMaze12. According to density allocation, walls 1, 4 and 5 seem to have been crossed the most often. Again – I believe users solved the second maze quicker because of previous training.

PredefinedMaze3 and PredefinedMaze13

Unlike in the case of the previous two predefined mazes, removing walls in the third case did change the length of the SSP. The minimum number of moves (MNM) decreased rapidly – by 71 %. However the AST decreased only by 43 % (from 134 to 76 seconds). The original PredefinedMaze3 was created by the recursive backtracker algorithm, which is specified by a relatively low dead end count and high river factor. The original SSP was designed to be very long (almost 40 % elitism factor), which – as already mentioned – caused a very low GLR. Figure 4.10 shows the SSP of PredefinedMaze13; numbers represent the walls removed from the original maze. (Check the density map of PredefinedMaze3 (Figure A.8) to see its SSP, which is exactly copied by the darkest cells set.)

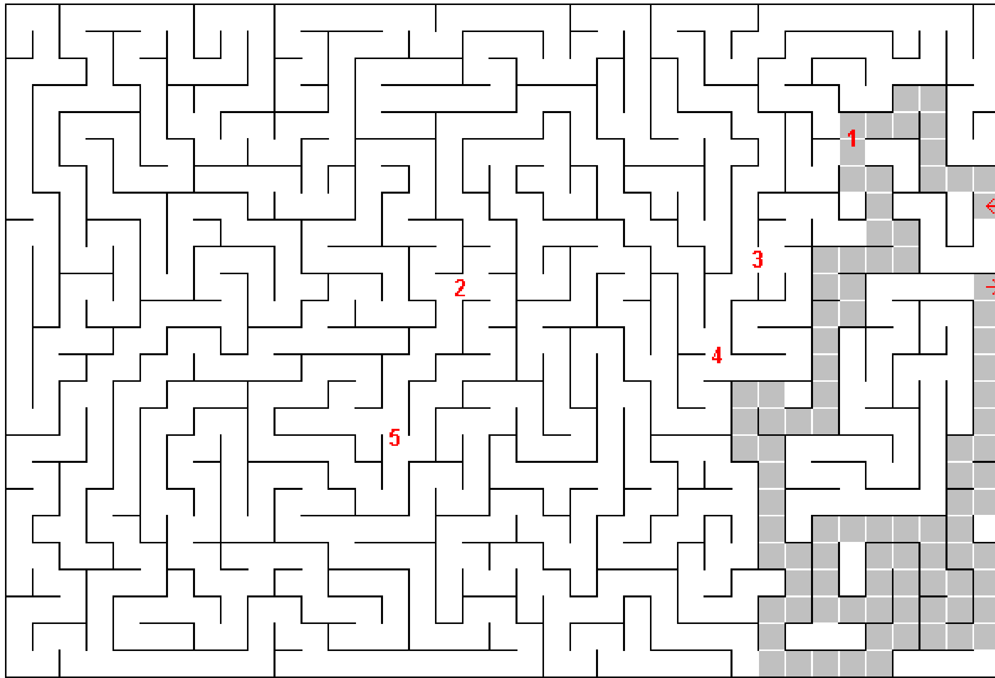


Figure 4.10: PredefinedMaze13

Density maps (Figure A.8 and Figure A.9) are quite clear and they depict the fact that both mazes have a very low GLR. Only 2 users out of 15 got lost or followed the original solution path when solving PredefinedMaze13. (Passages created by all walls – besides no. 1 – were used minimally or not at all.) The rest of the users found the SSP immediately or very quickly.

When observing the state space graph of PredefinedMaze3 (Figure 4.11) and comparing it with the state space graph of PredefinedMaze1 (Figure 4.7), we can see that mazes built by DFS based algorithms have a markedly lower dead end count than mazes created by non-DFS based algorithms. It is visualized by the branching intensity of both graphs. (Blue vertices represent mazes' SSP.)

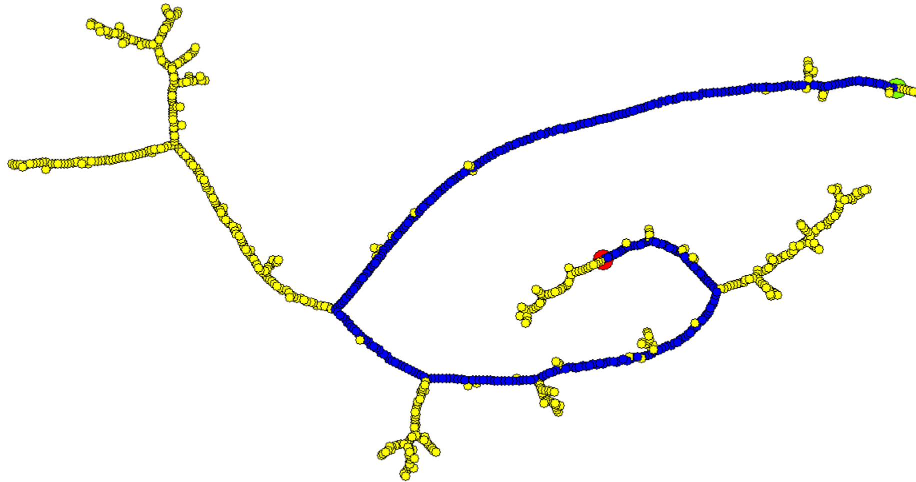


Figure 4.11: PredefinedMaze3 – state space

PredefinedMaze4 and PredefinedMaze14

Both mazes in this couple have the same SSP. The original one (PredefinedMaze4) was created using the DFS based hunt and kill algorithm, which already tells some basic features one should be expecting from such mazes. The altered maze has 9 extra walls down in comparison to its original.

Removing of several walls in the lower left corner had almost no impact on users' movement. As usual and expected – solvers mostly concentrate on the surroundings of a virtual straight line connecting the starting cell and the end cell. See Figure A.10 and Figure A.11 to compare both mazes' density maps.

PredefinedMaze5 and PredefinedMaze15

This maze couple consists of two totally identical mazes (created by the hunt and kill algorithm). The only difference was in the form of the solving process. Users solved the first maze in the mode enabling the previously visited cells to be marked. When solving PredefinedMaze15, this mode was turned off. It is quite difficult to decide whether turning this mode off had a relevant impact or not. The AST increased from 120 to 135 seconds (12.5 %) and the ANM increased from 207 to 249 (20 %). The advantage of seeing already visited cells could be probably more significant when observing a larger data sample of users solving a larger (than medium sized) braid maze (which the maze was not in this case). See Figure A.12 and Figure A.13 to compare both mazes' density maps and to observe their SSP (the darkest cells set).

PredefinedMaze6

PredefinedMaze6 was created by the randomized Kruskal's algorithm and it is a special

case of maze, since it had intentionally (and as the only one) no solution. The point was to study the impact of such a situation on users' behavior. I was warned before planning this and the warning was partially competent. The warning concerned potential loss of users' motivation after such an effortful and yet unsuccessful struggle. Some solvers did actually lose a lot of time while solving this maze. Solve attempt maximum was 16.8 minutes, which is obviously too much for such a medium sized maze.

Because of a (already mentioned) bug in the applet version 1, not all unsuccessful attempts were registered into the database during this time period; only those ones finished by clicking the "Give up" button (ergo excluding attempts finished by just closing the web browser tab or window). Let us just work with the data we have.

24 users performed 37 solve attempts ergo some of them tried it more than once. One solver reported the absence of the solution by creating a bug in the project's trac system. Approximately 5 users contacted me directly and reported that this maze had no solution, whereby I always kindly apologized and explained the reason. After the first report I placed a note under the applet informing that an absence of a solution is not a bug, yet some users probably did not see it.

8 users attempted to solve this maze at least twice. Most of the repeated attempts were registered with very minimal or no moves count, which might imply they checked the maze only visually and wanted to make sure there really is no solution. When analyzing attempts with a significantly higher moves count from applet version 2, it is worth mentioning that the average number of moves (ANM) was 402 and the average time before clicking the give-up button was 231 seconds. None of the rest of the predefined mazes had a higher ANM or a higher average solution time (AST).

Figure 4.12 shows the density map of PredefinedMaze6. The red line depicts the critical wall that was added to the algorithmically created original maze and it also shows the (only) way which would normally lead towards the end cell.

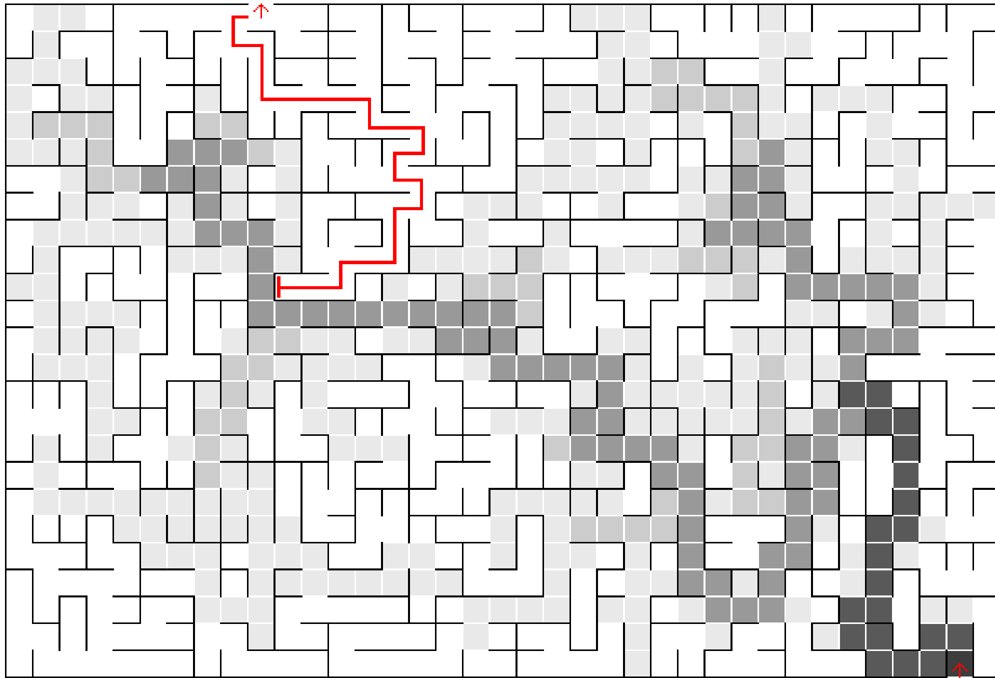


Figure 4.12: PredefinedMaze6

PredefinedMaze7 and PredefinedMaze8

This is a couple of small sized yet differently created mazes. PredefinedMaze7 was created using the non-DFS based randomized Kruskal's algorithm and PredefinedMaze8 by the DFS based recursive backtracker.

I asked users to solve all predefined mazes chronologically. That is why I decided to design these two mazes as very easy in order to increase users' motivation and to let them relax a little bit, since the upcoming two mazes were designed as large sized.

PredefinedMaze9 and PredefinedMaze10

Both mazes are large sized and created by a different algorithm. PredefinedMaze9 was created using randomized Prim's algorithm, PredefinedMaze10 using randomized Kruskal's algorithm. None of the rest of the predefined mazes has a lower elitism factor or a higher AST. See Figure A.14 and Figure A.15 containing density maps of both mazes (sets of the darkest cells copy both SSPs).

At this moment, we can try to compare medium sized and large sized mazes created by Prim's and Kruskal's algorithms (Table 4.6). We can observe a relatively low increase of the SSPs in comparison to the increase of the rest of the parameters.

		Randomized Prim's algorithm			Randomized Kruskal's algorithm		
	Maze size	SSP	AST	ANM	SSP	AST	ANM
Medium	925	80	71	106	113	112	182
Large	2400	119	166	310	180	224	360
Diff.	+ 159.5 %	+ 48.8 %	+ 133.8 %	+ 192.5 %	+ 59.3 %	+ 100 %	+ 97.8 %

Table 4.6: Comparison of mazes created by randomized Prim's and Kruskal's algorithm

4.4.4 User Behavior Analysis

User behavior analysis will be based on the study of moves (moves count, average moves count per maze or moves-per-second ratio, moves-over-time function). Small sized predefined mazes (7 and 8) are excluded and only applet version 2 sample data will be used. In this subchapter I will try to point out some interesting patterns or habits I've observed while analyzing the predefined mazes.

The moves-per-second[8] (MPS) ratio is the average moves count per second of one maze solving attempt. User's SAC is the count of different predefined mazes successfully solved by the user (only first attempts count in). User's AVG MPS is the MPS average of all user's maze solving attempts. Maze's AVG MPS is the average of all solving attempts of a particular predefined maze. MPS MIN and MPS MAX should be taken by analogy.

Table D.3 (ordered by the descending user's AVG MPS) shows users who had solved at least 5 predefined mazes during the applet version 2 time period.

Table 4.7 (ordered by the descending maze's AVG MPS) shows that users tend to move faster in mazes with a higher river factor, which means less crossings/decision points. (It applies especially for DFS based algorithms.)

The assumption is that brute force solving is specified by a high MPS and GLR rate. A solver relatively randomly (and quickly) roams through the maze area hoping to be on the correct solution path. Such behavior is supposed to lead to a high got-lost ratio because the probability of being on a dead end path (without analysis of the surroundings) is higher. This applies especially for mazes created by non-DFS based algorithms which have a higher dead end ratio.

Analytic solving should be specified by a low MPS and GLR rate. Solvers are expected to move rationally after analysis of the surroundings of their current position. This results in longer time intervals without any moves, ergo decreasing the MPS rate. Analysis should lead to a higher probability of finding the correct solution path. Therefore the GLR should be low. Let us see whether we can observe a proportional relationship between the MPS and GLR (Figure 4.13).

8. Moves per second (MPS) ratio = overall moves count / win time [s]

Maze	Algorithm	Maze's SAC	Maze's AVG MPS
PredefMaze13	Recursive backtracker	10	3.71
PredefMaze3	Recursive backtracker	15	3.42
PredefMaze11	Randomized Prim's	11	3.09
PredefMaze14	Hunt And Kill	11	2.98
PredefMaze12	Randomized Kruskal's	11	2.73
PredefMaze4	Hunt And Kill	14	2.63
PredefMaze15	Hunt And Kill	12	2.59
PredefMaze9	Randomized Prim's	10	2.57
PredefMaze5	Hunt And Kill	15	2.23
PredefMaze2	Randomized Kruskal's	19	2.02
PredefMaze1	Randomized Prim's	22	2.01
PredefMaze10	Randomized Kruskal's	11	1.98

Table 4.7: Predefined mazes and their AVG MPS

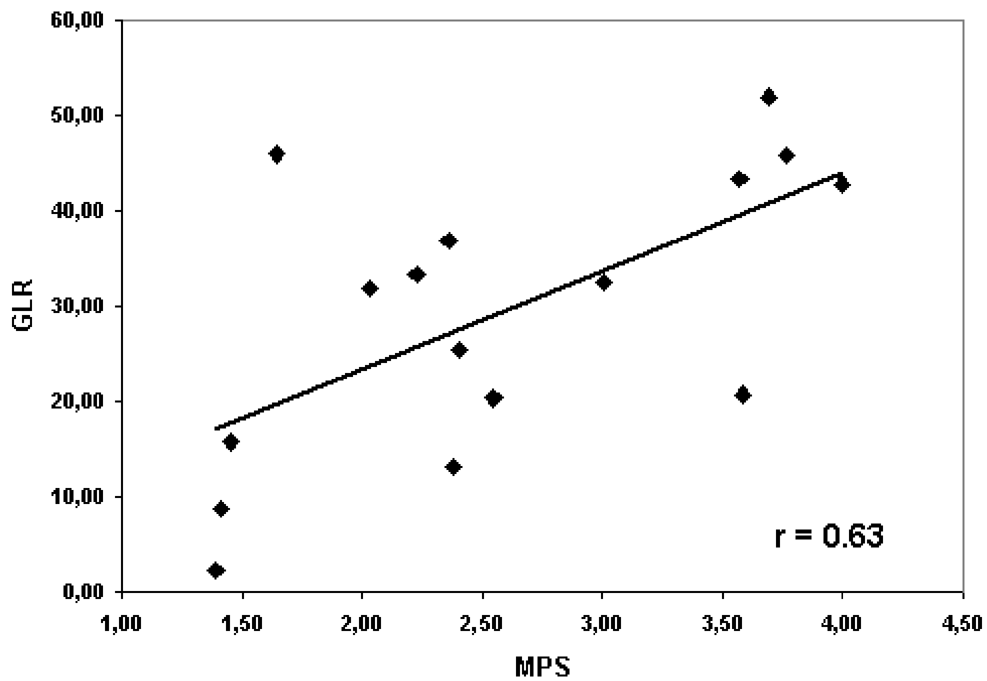


Figure 4.13: User's GLR and MPS relationship

Basically the trend line is the evidence of a very irregular but yet mostly proportional relationship between MPS and GLR rates. Proportional tendency is proven also by interpreting the Pearson product-moment correlation coefficient r. A deep comparison of the

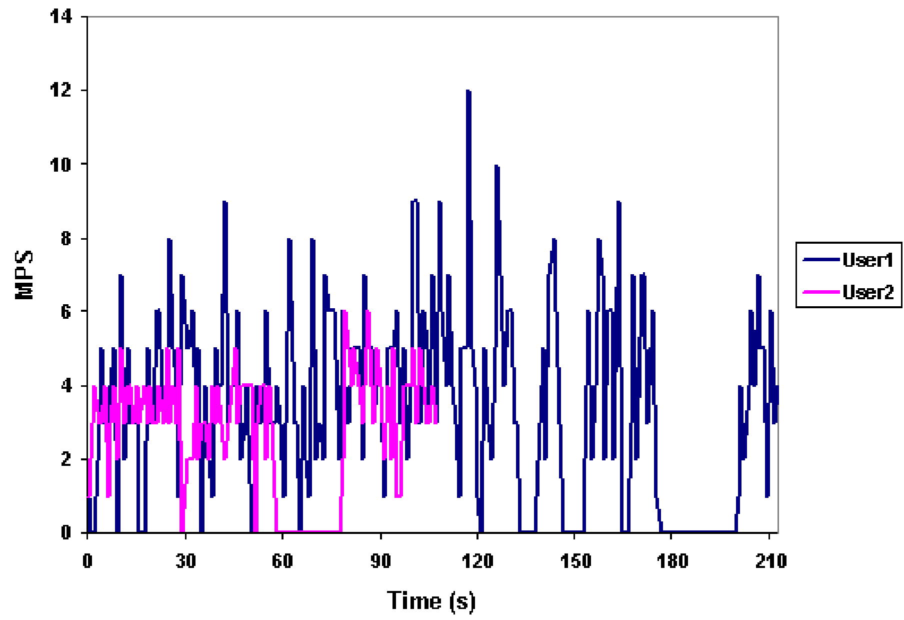
MPS vs. GLR relationship demands a more complex calculation and problem formalization. The cause is their dependency on external aspects. The GLR depends on static (objective) aspects of maze size and SSP length, whereas the MPS rate depends on a dynamic (subjective) aspect of total solution time, which varies from user to user. Such a complex analysis will not be performed within this master's thesis, yet it might be a good suggestion for further papers concerning this topic.

Attempts with a low MPS and high GLR are quite specific and they have a negative impact on proving my primary assumptions. These are attempts of – with all due respect – not very successful users, who – as I interpret the results – solved slowly (low MPS) and yet got lost or mislead quite often (high GLR). Of course – statistics are supposed to help to mirror the reality and not vice versa. However I believe the undesired irregularity would continuously retreat (but not completely disappear) as the sample data got more extended.

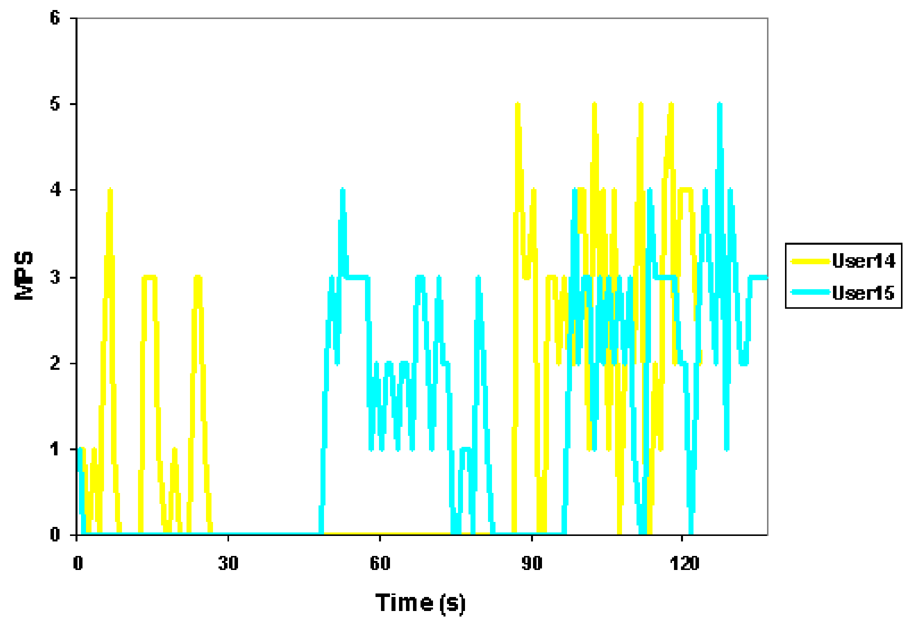
Individual users' approach

The extremely low MPS and GLR rates of User16 are caused by his specific solving approach. He always started solving a maze after a relatively long while (when an average of 47 % of the total time had passed). He used this gap most likely to analyze and find the correct solution path. Therefore his every attempt had a minimal or zero GLR.

Let us take a closer look at some users and predefined mazes. We will use the moves-over-time function to visualize large sized predefined mazes (9 and 10) of users User1, User2, User14, and User15 (ergo 2 users with the highest and the lowest MPS; each with all 12 attempts available for statistics). Observe and compare graphs within Figure 4.14 and Figure 4.15.

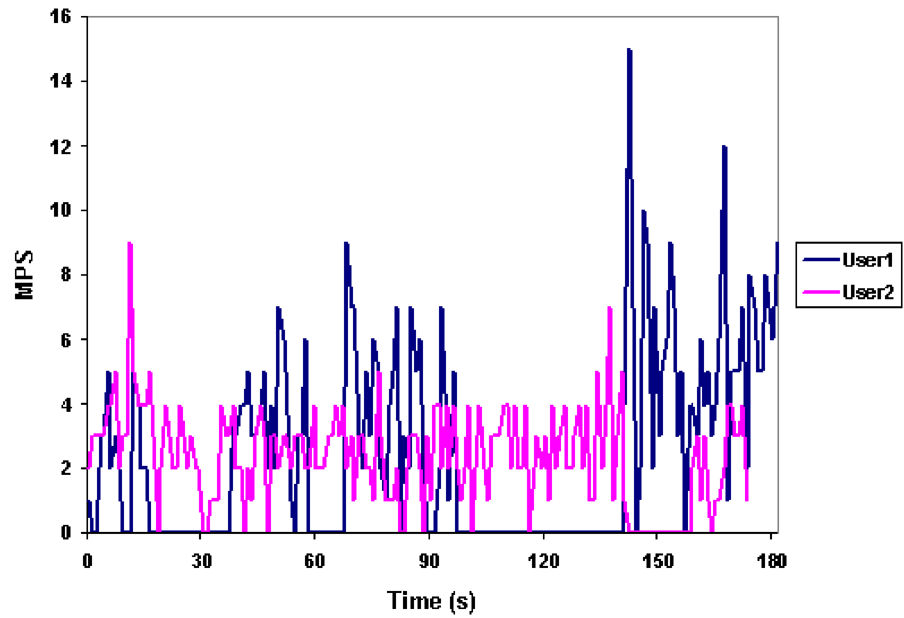


(a)

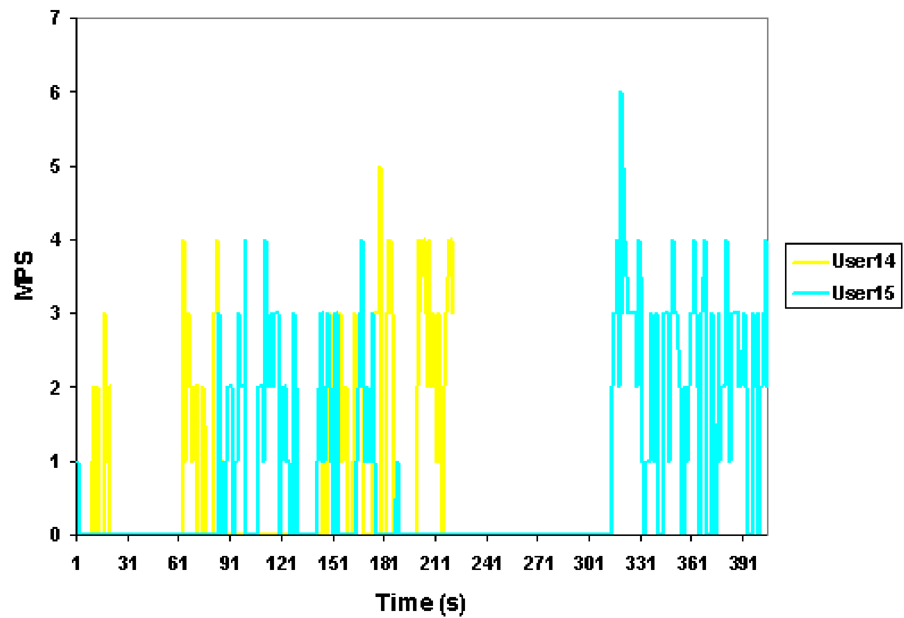


(b)

Figure 4.14: PredefMaze9 – users with high MPS (a) and low MPS (b)



(a)



(b)

Figure 4.15: PredefMaze10 – users with high MPS (a) and low MPS (b)

Maze	User	AVG MPS	STDEV
PredefMaze9	User1	3.38	2.60
	User2	2.72	1.70
	User14	1.03	1.53
	User15	1.20	1.37
PredefMaze10	User1	2.42	2.95
	User2	2.34	1.54
	User14	0.83	1.31
	User15	0.76	1.24

Table 4.8: Users' individual results – PredefMaze9 and PredefMaze10

Large sized mazes should be the most suitable for observing differences in users' solving approach. Yet it is getting more and more difficult to prove that users actually do use either the analytical or brute force approach. More accurately we can observe properties we use to distinguish the analytical solving approach. But brute force approach properties as defined in my assumptions are not possible to be observed clearly and distinctly. One would expect brute force attempts to have a higher average MPS (User1 and User2) and to be more steady (lower deviation – Table 4.8), which is not true in either of our cases. We also assumed that the analytical approach would bring better winning time results for large sized mazes. However this cannot be clearly decided as well (see graphs).

In my opinion, all these discoveries lead to a conclusion, that a distinct and clear difference cannot be observed. Most users seem a priori to lean to the analytical approach, but actually it is more like a mix of both approaches. Even the most brute force like attempt (PredefinedMaze9 – User1) eventually needed a longer pause to finally find the correct solution path. Generally we should speak more about the changing of phases within an attempt. And eventually characteristics of such phases can (and actually do) lean to characteristics of either of the two approaches. Probably a deep study of such a phase interchange might be another interesting inspiration for further research within this topic.

Chapter 5

Conclusion

This master's thesis was my first experience with such a kind of research. Eventually I found it much more interesting than I expected. It proves that maze generation and mazes generally are a very interesting phenomenon that deserves a deeper studying (not only for purposes of human problem solving).

Since there are not many sources concerning such a specific topic as maze generation is, I decided to include some summarizing information in this thesis as well, thus enabling any – even a laic – reader to reveal the fascinating world of mazes. One can start with easier tutorial chapters followed by chapters containing more of mathematics and formalization. Finally, a minor (and very laic) look into the psychological aspect of this topic is provided by the last part of this work.

The topic of maze generation is much more extensive than it seems at the first sight. There are still many things to explore, many experiments to perform. There are also things I would do differently, for instance a better analysis and preparation prior to implementing and publishing the maze generation applet, which would probably avoid some bugs or problems (that however have not caused any critical or blocking situations).

My master's thesis may (and hopefully will) serve as a starting point for other authors and papers. As already mentioned, there are still many things to try and prove or reject. I have tried to establish some basic concepts (analytical vs. brute force approaches, got-lost ratio, etc.), but of course in a very laic and simple way. Now it is up to other people to elaborate or develop them, eventually to reject them and to find another proper way of formalizing this topic. Other people are more than welcome to bring some more formalization (functional dependencies, algorithms, probability evaluations) into this topic, since in my opinion it might help a lot especially in the field of human problem solving.

However the exploratory part of this research is very important as well. The data collecting system I used was quite simple and not perfect at all. My sample unfortunately did not consist of many various sorts of humans (speaking of age, education level etc.). It might also be very interesting to be able to physically see and observe the solver while solving specific types of mazes (focusing for example on altering various maze properties and analyzing the impact) or maybe even other types of puzzle games.

Since I have already put so much effort into this work, I am going to keep the web project running. After some time I will either rerun the scripts, recreate charts and re-evaluate the results or I will provide all the obtained results to someone willing to make further research (of course after prior agreement).

Bibliography

[1] Abbott, R.: *Logic Mazes*, available at URL <http://www.logicmazes.com>. 1.1.3

[2] Bouda, O.: *Sběr záznamů postupů řešení logických úloh*, Masarykova univerzita, Fakulta informatiky, 2010. 4.1

[3] Demel, J.: *Grafy a jejich aplikace*, Academia, 2002, ISBN 8020009906. 13

[4] Dickson, A.: *Introduction to Graph Theory*, Utah Math Circle, October 2006, <http://www.math.utah.edu/mathcircle/notes/MC_Graph_Theory.pdf>. 2.2.1

[5] Gibbons, A.: *Algorithmic graph theory*, Cambridge University Press, 1994, ISBN 0521288819. 2.3.1, 2.3.2

[6] Gross, J. and Yellen, J.: *Graph theory and its applications*, CRC Press, 1999, ISBN 0849339820. 2.3.1, 2.3.3, 15, 2.3.3

[7] Křepský, T.: *Generátor a simulátor logické úlohy Replacement puzzle*, Masarykova univerzita, Fakulta informatiky, podzim 2009. 4.1

[8] Astbury, M.: *Mike's Mazes*, available at URL <http://www.mikes-mazes.com>. A.3

[9] Nester, D.: *Maze Profiler*, Bluffton University, Bluffton, Ohio, available at URL <http://www.bluffton.edu/~nesterd/java/mazeprofiler.html>. 2.3.1, 2.3.2, 2.3.3, 2.4, 4.4.2

[10] Pullen, W.: *Think Labyrinth!*, available at URL <http://www.astrolog.org/labyrnth.htm>. 1.1.3, 1.2, 1.2.1, 1.2.2, 4, 1.2.4, 2.3.1, 2.3.2, 2.3.3, 2.3.4, 2.4, 2.4, 4.2.1, 4.4.2

[11] Sedláček, J.: *Úvod do teorie grafů*, Praha, 1981. 2.2.1

[12] Jarušek, P. and Pelánek, R.: *Human Problem Solving: Sokoban Case Study*, Faculty of Informatics, Masaryk University, April 2010, <http://www.fi.muni.cz/reports/files/2010/FIMU-RS-2010-01.pdf>. 4.1, 4.4.2

[13] Vejmola, S.: *Chvála bludišť*, Státní pedagogické nakladatelství, Praha, 1991, ISBN 80-04-25235-4. 1.2.4, 2.2, 2.2.2, 4.2.1

[14] Weisstein, E.: *MathWorld – A Wolfram Web Resource*, available at URL <http://mathworld.wolfram.com/>. 1.2.1, 1.2.3

[15] Wikipedia: *Seven Bridges of Königsberg*, available at URL <http://en.wikipedia.org/wiki/Seven_Bridges_of_Königsberg>. 2.1

[16] Wikipedia: *Maze*, available at URL <http://en.wikipedia.org/wiki/Maze> . 1.1.3

[17] Wikipedia: *Spanning tree*, available at URL <http://en.wikipedia.org/wiki/Spanning_tree> . 2.2.1

Appendix A

Figures

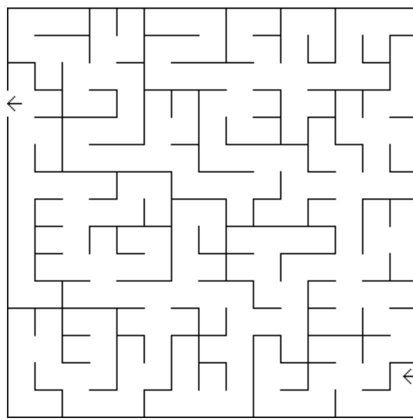


Figure A.1: A maze created by Kruskal's algorithm (Maze Generation applet).

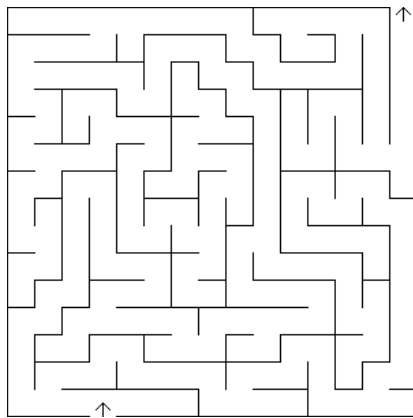


Figure A.2: A maze created by the recursive backtracker algorithm (Maze Generation applet).

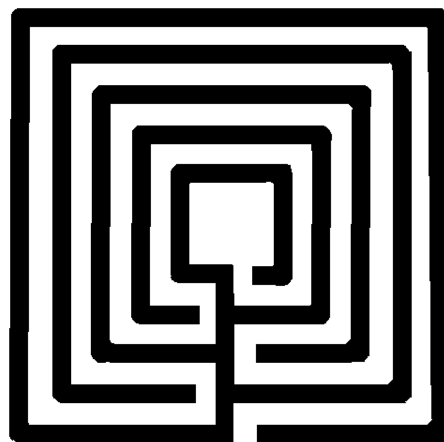


Figure A.3: A unicursal maze ([8] – http://www.mikes-mazes.com/graphics/unicur.gif).

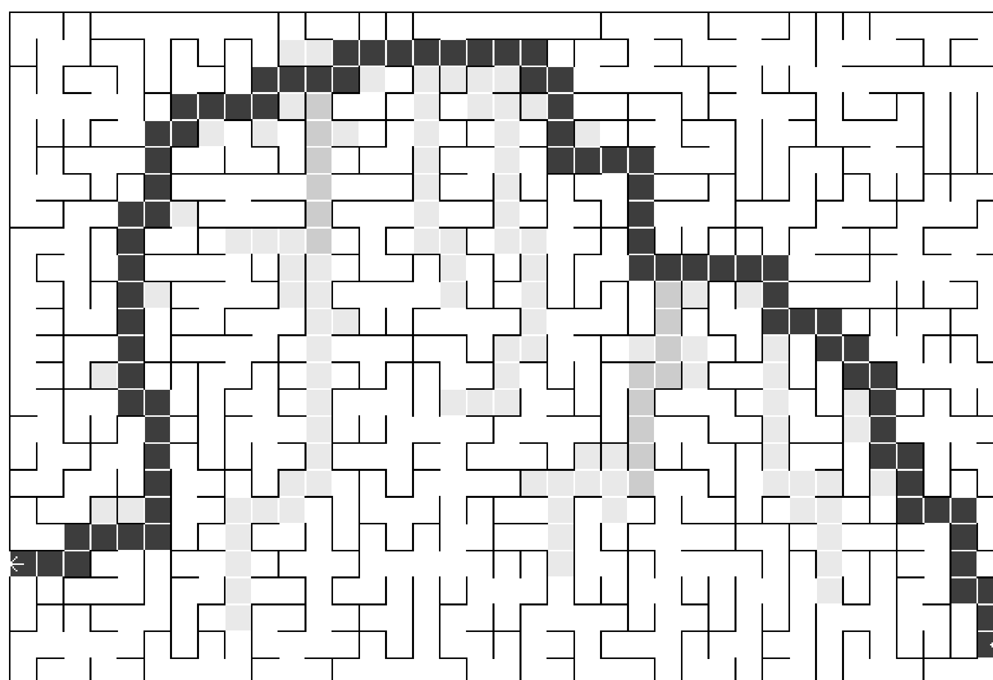


Figure A.4: Density map – PredefinedMaze1.

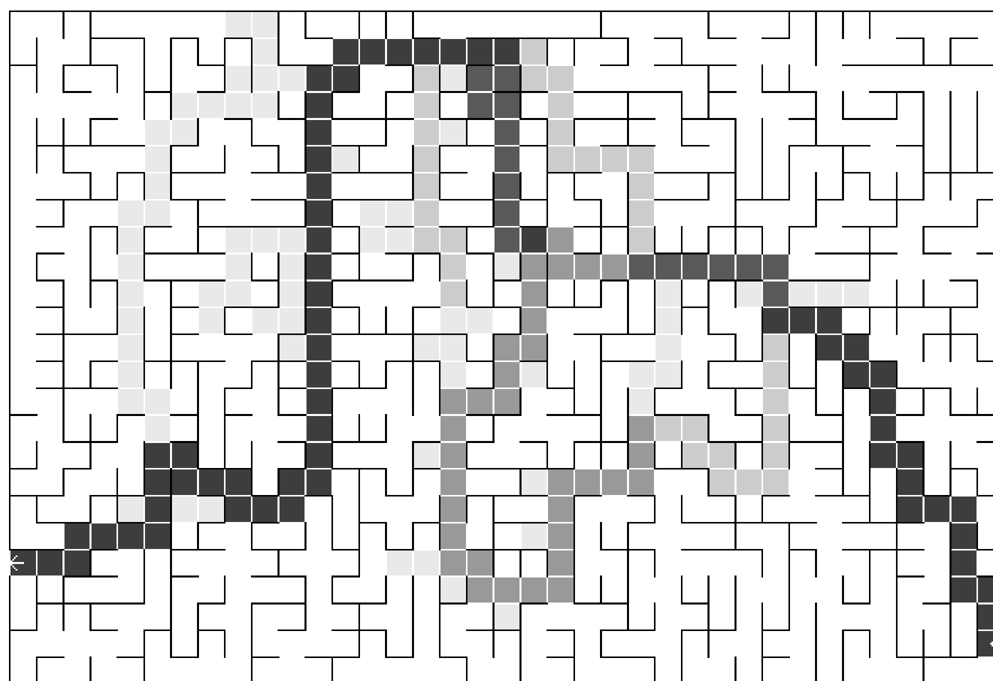


Figure A.5: Density map – PredefinedMaze11.

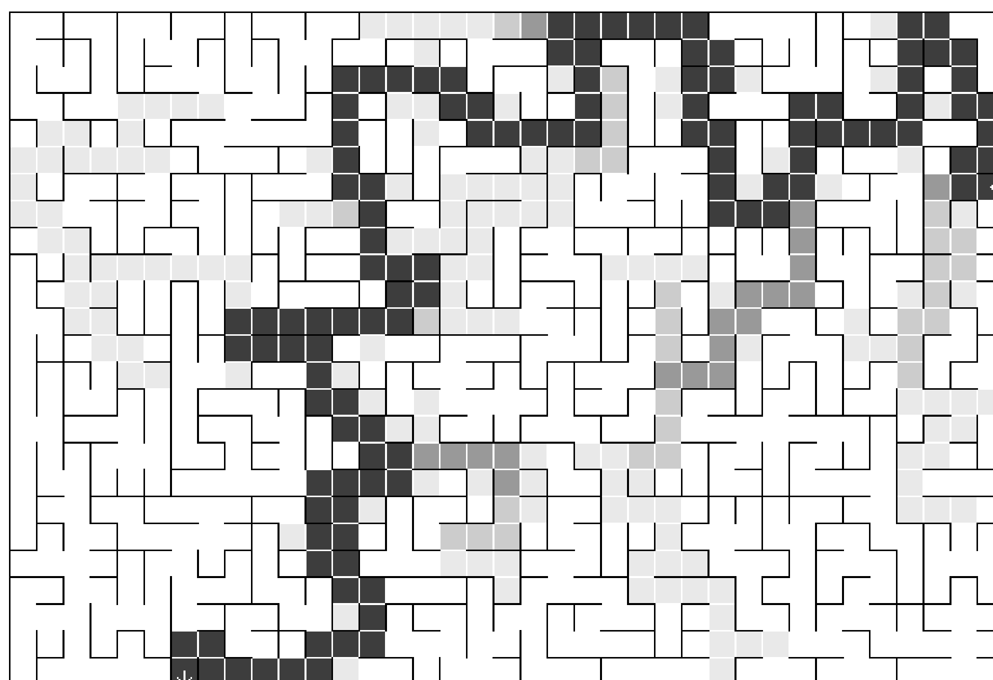


Figure A.6: Density map – PredefinedMaze2.

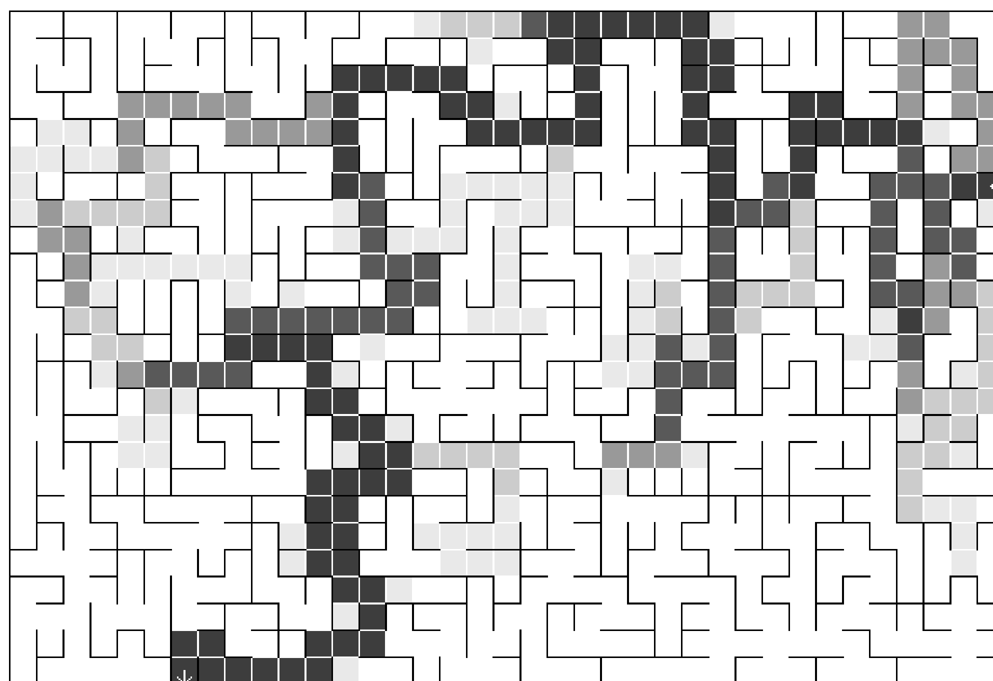


Figure A.7: Density map – PredefinedMaze12.

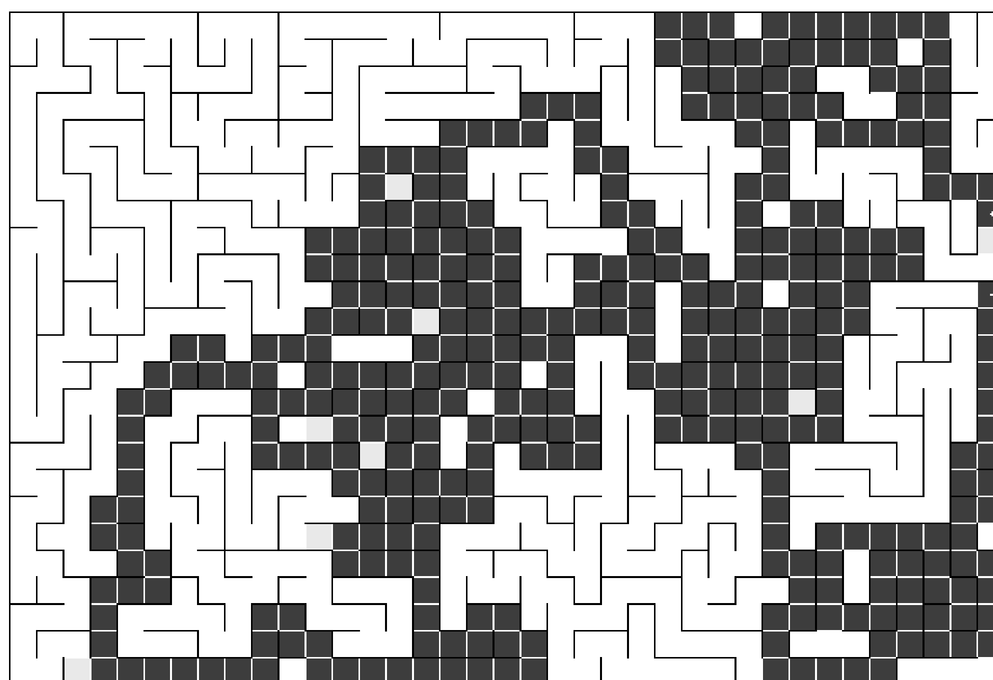


Figure A.8: Density map – PredefinedMaze3.

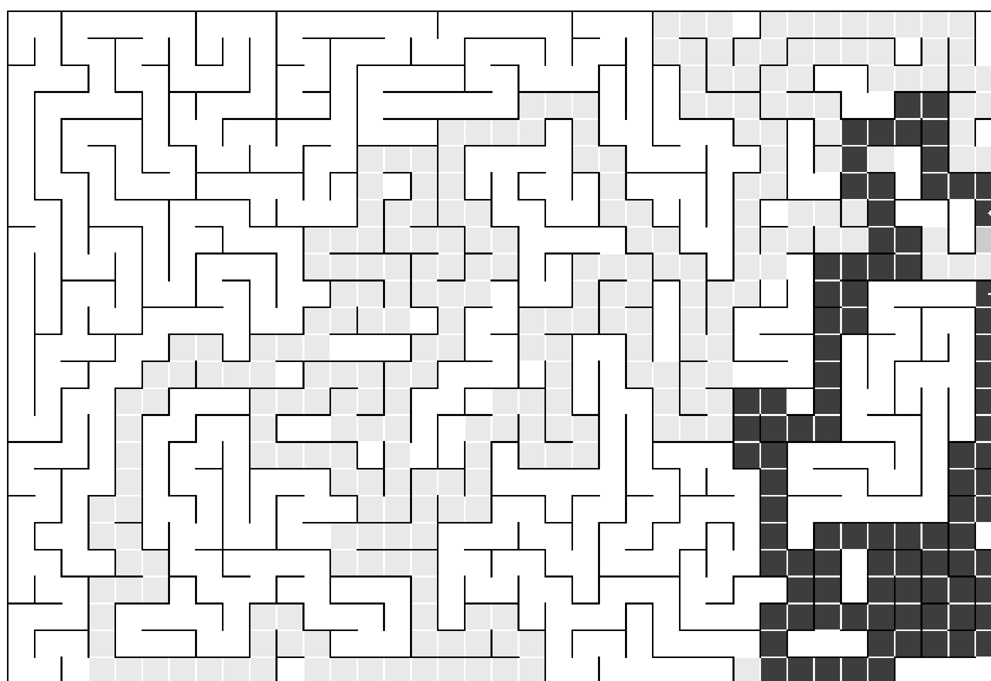


Figure A.9: Density map – PredefinedMaze13.

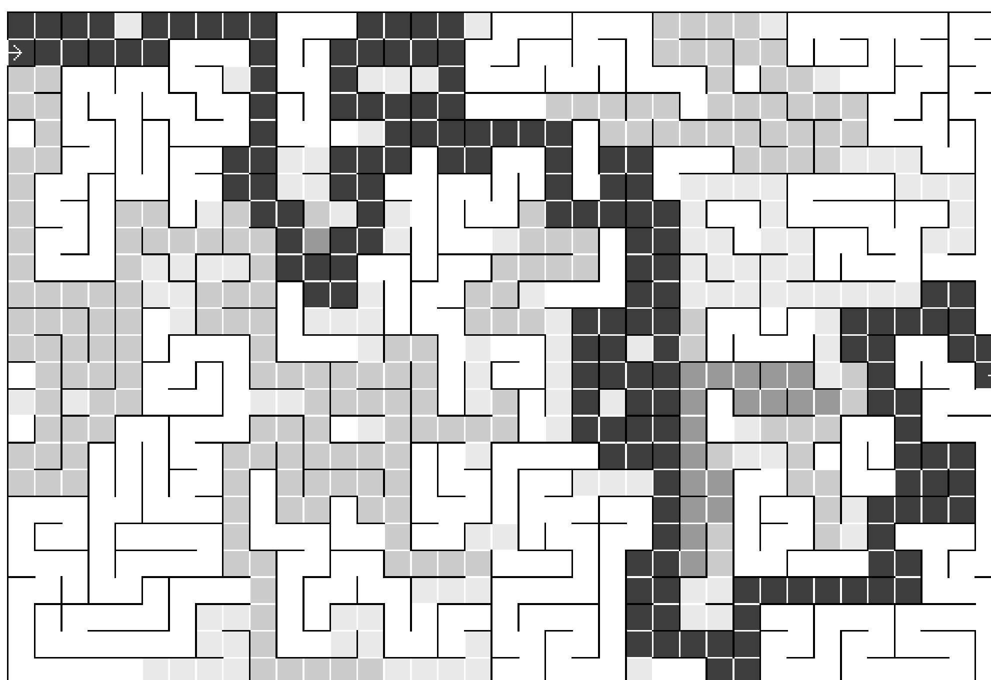


Figure A.10: Density map – PredefinedMaze4.

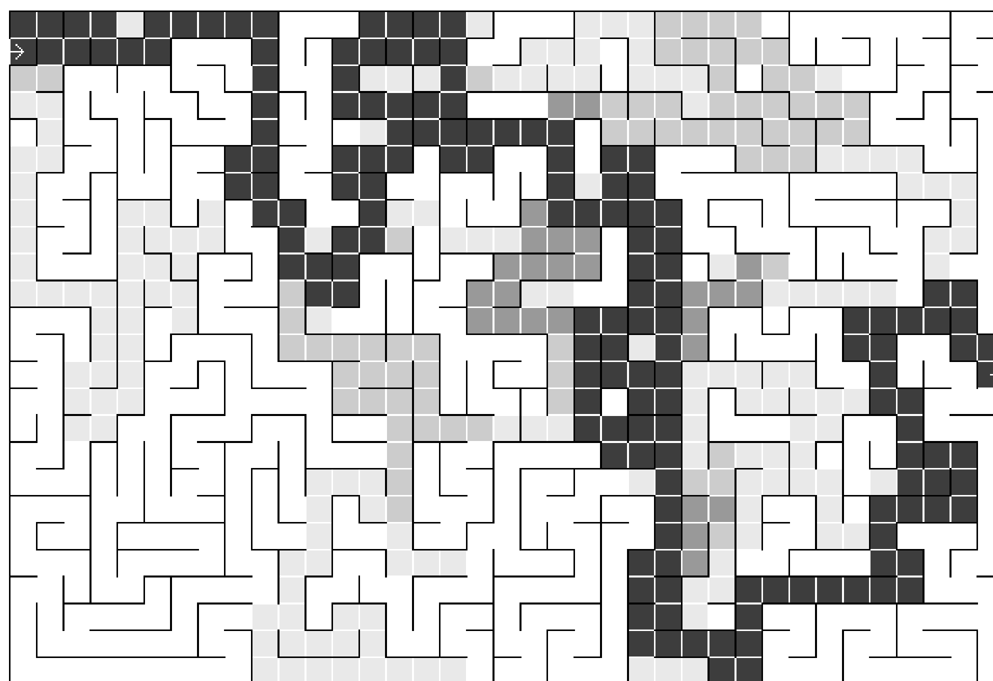


Figure A.11: Density map – PredefinedMaze14.

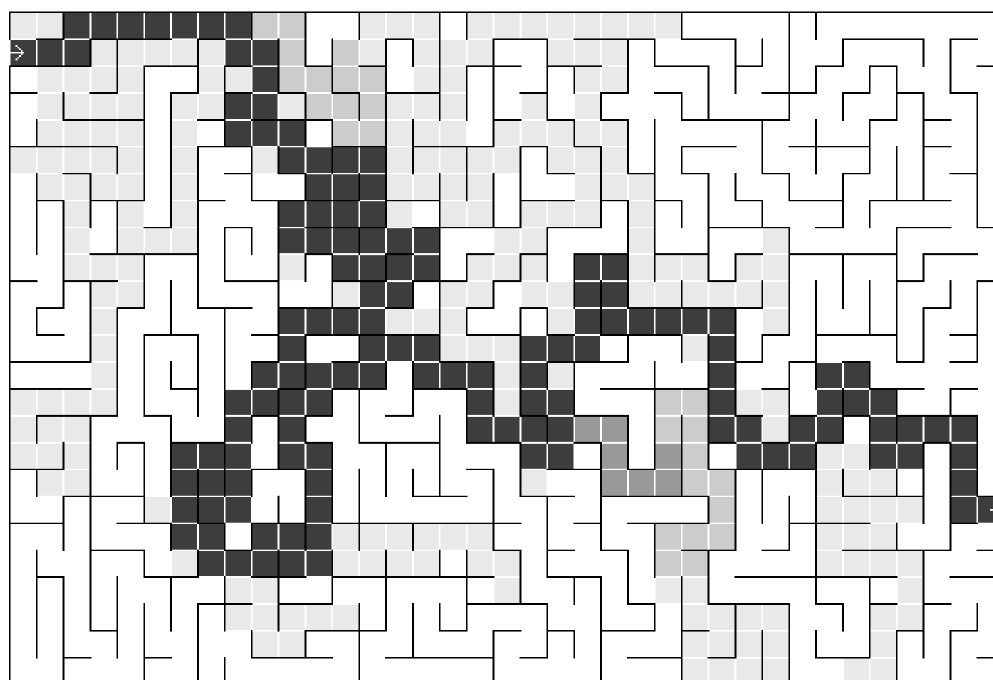


Figure A.12: Density map – PredefinedMaze5.

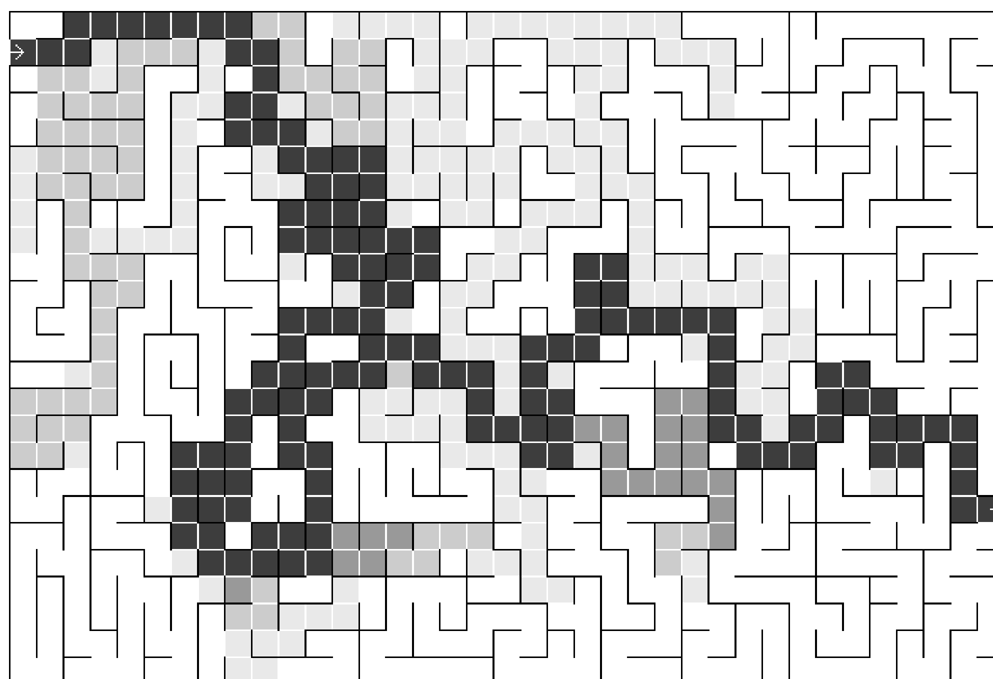


Figure A.13: Density map – PredefinedMaze15.

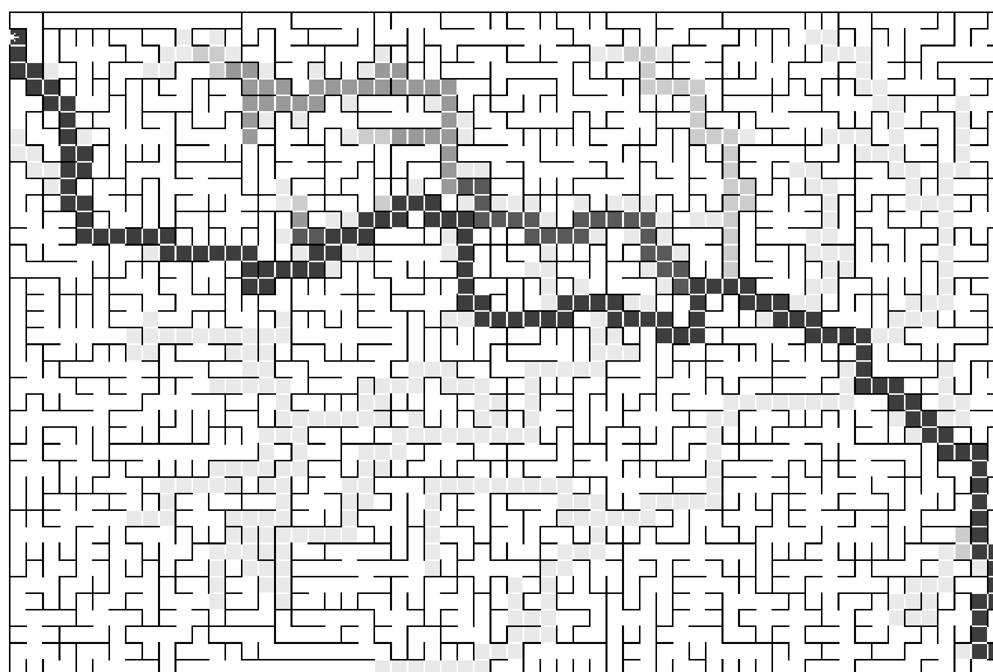


Figure A.14: Density map – PredefinedMaze9.

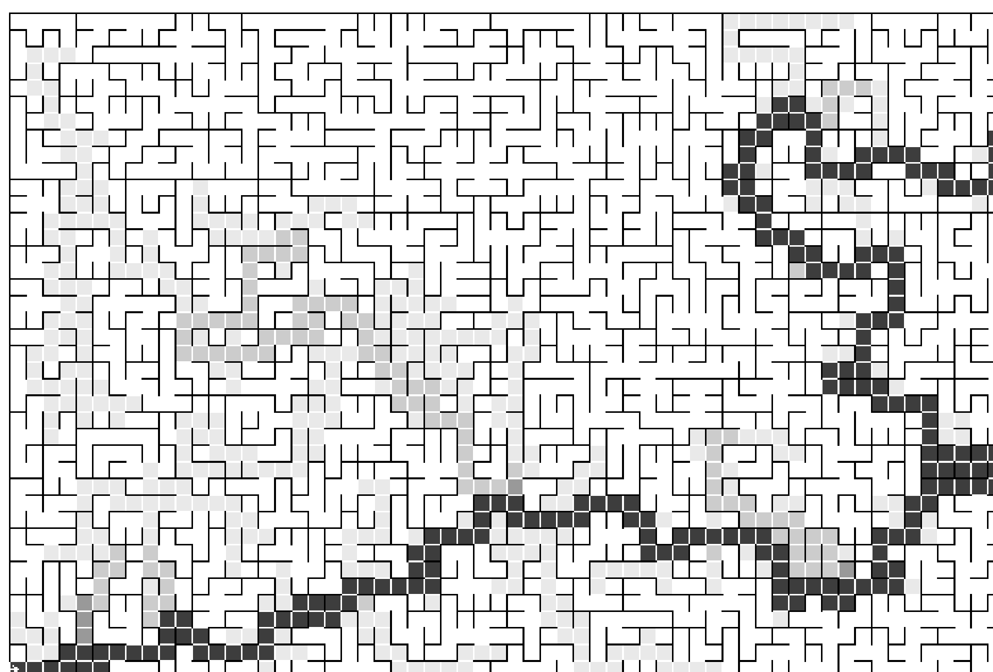


Figure A.15: Density map – PredefinedMaze10.

Appendix B

Kirchhoff's Theorem Example

I decided to mention this example because I was unable to find another relevant source directly interconnecting the Kirchhoff's theorem with the concept of perfect mazes. It shows, how to calculate the number of spanning trees of a connected graph (ergo the number of different perfect mazes on a specific maze grid – see bottom of the figure). The basic graph is actually a cycle graph, for which applies that the number of spanning trees is equal to the number of vertices of the cycle graph. Using the Kirchhoff's theorem might be helpful especially when dealing with more complex maze grids.

$$\begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix} - \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 2 & -1 & -1 & 0 \\ -1 & 2 & 0 & -1 \\ -1 & 0 & 2 & -1 \\ 0 & -1 & -1 & 2 \end{pmatrix}$$

$$C_{11} = (-1)^{(1+1)} M_{11}$$

$$C_{11} = \begin{vmatrix} 2 & 0 & -1 \\ 0 & 2 & -1 \\ -1 & -1 & 2 \end{vmatrix} = 4$$

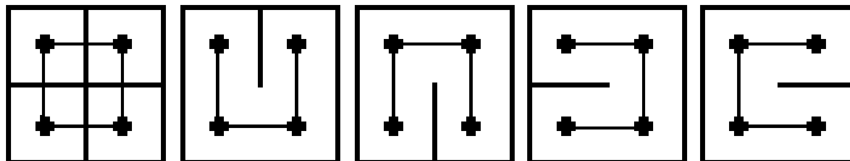


Figure B.1: Kirchhoff's theorem example

Appendix C

Java Source Code Examples

```
public String toString(){
        return "[" + this.getX() + "," + this.getY() + "]";
}
```

Return example: “[14,6]”

Example C.0.1: Cell – *toString()* method

```
public String toString(){
        return "[" + this.getTimeOfClick() + ":" + this.getX() + ","
            + this.getY() + "]";
}
```

Return example: “[25:14,6]”

Example C.0.2: CellWithTiming – *toString()* method

```
public static Cell getRandomCell(int mazeWidth, int mazeHeight){
        Random r = new Random();
        Cell c = new Cell(r.nextInt(mazeWidth),r.nextInt(mazeHeight));
        return c;
}
```

Example C.0.3: MethodUtil – *getRandomCell(int, int)* method

```
Cell startCell = MethodUtil.getRandomCell(mazeWidth, mazeHeight);
Cell cF = new Cell();
Cell cI = new Cell();
inCells.add(startCell);
markAsFrontierAround(startCell);
while(!frontierCells.isEmpty()){
            cF = getRandomFrontierCell();
            cI = getRandomInCellAround(cF);
            inCells.add(cF);
            markAsFrontierAround(cF);
            addPathBetween(cI, cF);
            frontierCells.remove(cF);
}
```

Example C.0.4: Randomized Prim's algorithm

```
while(!wallList.isEmpty()){
            currentWall = getRandomWall();
            currentPath = currentWall.innerWallToPath();
            c1 = currentPath.getC1();
            c2 = currentPath.getC2();
            if (getSetIndexOf(c1) != getSetIndexOf(c2)){
                joinSetsOf(c1, c2);
                addPathBetween(c1, c2);
            }
            wallList.remove(currentWall);
}
```

Example C.0.5: Randomized Kruskal's algorithm

```
Cell currentCell = MethodUtil.getRandomCell(mazeWidth, mazeHeight);
visitedCells.add(currentCell);
if (hasUnvisitedAround(currentCell)) {
    newCell = getRandomUnvisitedAroundCell(currentCell);
    stack.add(currentCell);
    addPathBetween(currentCell, newCell);
    currentCell = newCell;
} else {
    currentCell = stack.get(stack.size() - 1);
    stack.remove(stack.size() - 1);
}
while(!stack.isEmpty()){
    visitedCells.add(currentCell);
    if (hasUnvisitedAround(currentCell)){
        newCell = getRandomUnvisitedAroundCell(currentCell);
        stack.add(currentCell);
        addPathBetween(currentCell, newCell);
        currentCell = newCell;
    } else {
        currentCell = stack.get(stack.size()-1);
        stack.remove(stack.size()-1);
    }
}
```

Example C.0.6: Recursive backtracker algorithm

```
Cell currentCell = MethodUtil.getRandomCell(mazeWidth, mazeHeight);
visitedCells.add(currentCell);
while(visitedCells.size() < cellCount){
    if (hasUnvisitedAround(currentCell)){
        newCell = getRandomUnvisitedAroundCell(currentCell);
        addPathBetween(currentCell, newCell);
        currentCell = newCell;
        visitedCells.add(currentCell);
    } else {
        currentCell = getRandomUncompleteCell();
    }
}
```

Example C.0.7: Hunt-and-kill algorithm

```
Cell startCell = MethodUtil.getRandomCell(mazeWidth, mazeHeight);
visitedCellsCopy.add(startCell);
visitedCells = new HashSet<Cell>(visitedCellsCopy);
while(visitedCells.size() < cellCount){
    for (Cell cell : visitedCells) {
        if (hasUnvisitedAround(cell)) {
            newCell = getRandomUnvisitedAroundCell(cell);
            visitedCellsCopy.add(newCell);
            addPathBetween(cell, newCell);
        }
    }
    visitedCells = new HashSet<Cell>(visitedCellsCopy);
}
```

Example C.0.8: Bacterial growth algorithm

Appendix D

Table Data

Algorithm	Maze size	SAC	AST (sec.)	TT – SA (min.)	ANM
Bacterial Growth	Large	3	27	1.37	127
	Medium	32	14	7.53	78
	Small	1	11	0.18	40
	Tiny	2	2	0.08	11
Hunt And Kill	Large	6	194	19.47	392
	Medium	46	121	93.45	212
	Small	8	13	1.83	61
	Tiny	40	7	4.77	21
Randomized Kruskal's	Large	14	144	33.70	170
	Medium	18	79	23.83	99
	Small	9	15	2.38	39
	Tiny	35	4	2.45	15
Randomized Prim's	Large	29	67	32.55	129
	Medium	861	30	443.68	82
	Small	9	12	1.80	40
	Tiny	62	4	4.97	13
Recursive Backtracker	Large	3	158	7.90	331
	Medium	10	64	10.73	166
	Small	54	18	16.58	92
	Tiny	42	8	5.65	21

Table D.1: Randomized mazes' global statistics

Maze type	Size	SAC	AST (sec.)	TT – SA (min.)	ANM	MNM	Elitism
PredefMaze1	Medium	47	71	56.33	106	80	8.65 %
PredefMaze2	Medium	39	112	73.07	182	113	12.22 %
PredefMaze3	Medium	29	134	65.03	363	358	38.70 %
PredefMaze4	Medium	25	126	52.70	297	155	16.76 %
PredefMaze5	Medium	24	120	48.17	207	134	14.49 %
PredefMaze6	Medium	N/A	N/A	N/A	N/A	N/A	N/A
PredefMaze7	Small	18	17	5.32	40	35	15.56 %
PredefMaze8	Small	18	30	9.02	98	90	40.00 %
PredefMaze9	Large	16	166	44.52	310	119	4.96 %
PredefMaze10	Large	14	224	52.45	360	180	7.50 %
PredefMaze11	Medium	16	45	12.25	117	80	8.65 %
PredefMaze12	Medium	16	85	22.88	191	113	12.22 %
PredefMaze13	Medium	15	76	19.07	129	92	9.95 %
PredefMaze14	Medium	16	114	30.57	294	155	16.76 %
PredefMaze15	Medium	17	135	38.37	249	134	14.49 %

Table D.2: Predefined mazes' filtered statistics (only first attempts of only registered users)

User	SAC	AVG MPS	MPS MIN	MPS MAX	MAX–MIN diff.	AVG GLR
User1	12	4.00	2.43	6.93	4.50	42.83
User2	12	3.77	2.35	5.75	3.40	45.72
User3	5	3.69	2.85	4.48	1.63	51.95
User4	12	3.59	2.64	6.13	3.49	20.74
User5	12	3.57	1.76	5.22	3.46	43.42
User6	12	3.01	2.52	4.23	1.71	32.46
User7	5	2.55	1.65	3.33	1.68	20.29
User8	7	2.40	2.09	2.85	0.76	25.41
User9	5	2.38	1.61	4.59	2.98	13.17
User10	5	2.36	1.09	3.15	2.06	36.88
User11	12	2.23	1.62	2.99	1.37	33.31
User12	5	2.03	1.45	2.63	1.18	31.85
User13	6	1.64	1.10	2.30	1.20	45.89
User14	12	1.45	0.84	2.07	1.23	15.81
User15	12	1.41	0.76	2.24	1.48	8.81
User16	5	1.39	0.92	1.76	0.84	2.23

Table D.3: Statistics of registered users solving predefined mazes (only users with SAC > 4 from appet version 2 period are shown). Ordered by AVG MPS descending.

Appendix E

Electronic Attachment

Folder structure of the electronic attachment:

- *db_data*
  - *tblMazes.csv*[1] – sample data until December 7th 2010
  - *tblMazes.sql* – MySQL script to recreate an empty table with corresponding field data types and structure
- *MazeGeneration* – NetBeans 6.9.1 project folder containing all the necessary Maze Generation applet source files
- *PHP_scripting*
  - *MazeGeneration.php* – a relic file because of ongoing development purposes
  - *MazeGeneration2.php* – the actual PHP script file accepting requests from Maze Generation applet

Comments:

- The applet is versioned. Each database record contains the applet version number it has been created by.
- NetBeans project contains more algorithm classes than a user is actually shown in the list. Only classes of algorithms available for solving are correctly implemented!
- *ANONYM* identifies records of anonymous users, ergo users who had not been logged while solving.
- As already mentioned, records from applet version 1 contain invalid data in columns *user_draw* and *user_draw_timing* and therefore should be excluded from any movement analysis.

1. Vertical bar is the column separator. Column *user_draw* is a relic and it contains no relevant data or (mostly) no data at all and may be ignored.