Leiden University
Computer Science

Analysing the difficulty of mazes

using a web application

Name:	Tama McGlinn
Date:	16/08/2015
Supervisor:	Aske Plaat
2nd reader:	Siegfried Nijssen

Dissertation for a Bachelor of Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Analysing the difficulty of mazes using a web application

Tama McGlinn

Abstract

This dissertation presents a study of factors which make maze solution more difficult. It was conducted by constructing a website to gather data on visitors solving mazes, which is then processed by a custom-built analysis program. The data produced by this analysis was examined to shed light on the methods humans employ in solving mazes, which were then modeled in a simulator. This simulator can be used to automatically judge the difficulty of any given maze. It was found that the chosen specifications of mazes bear no statistical relevance to their difficulty to human participants, and that the simulation, based on Trémaux's algorithm, also cannot yet be used to directly estimate maze difficulty.

Acknowledgements

There are many people to thank; my wife, for enduring all the prattle I produced while chiseling out the theory and code, and for tireless proofreading; my parents, for proof-reading and for using their social network to reach many more test-subjects than I could have alone; my supervisor, for helping me narrow down my focus and for reassuring me of my topic's scientific importance; and of course, 355 test-subjects from all over the world doing mazes for science! Thank you all.

Figure 1: Visitors to the maze website, mapped by ip address

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In order to assess the difficulty of mazes, a website was built where test-subjects can complete mazes with varying properties. The path they walked was logged to a server. Reproducing the same mazes later, these properties were tested for statistical significance with the difficulty of the maze for the players. The difficulty was measured by counting the number of repeat steps on the same tiles. The results show that linear models of these properties are unlikely to yield anything worthwhile. Rather than looking directly at properties, we construct a simulator based on qualitative analysis of human paths.

1.1 Aim

Maze algorithms have long fascinated me, but in constructing maze algorithms and testing them, I always found I was flying blind in regards to their difficulty. The problem therein is that, in order to test the difficulty of mazes, we need people willing to try them, which severely impairs the speed of development of these algorithms. Not only that, we also need to have enough people solving each specific maze instance to gain statistically relevant results. For this reason, I set out to develop an algorithm to judge the difficulty of mazes *automatically*.

1.2 Applications

Many games, including rogue-likes and first-person shooters, are played on a map that is essentially a maze with certain desired properties — one of which is their level of difficulty. An application that can automatically assess the level of difficulty makes it possible to construct these maps with less human intervention,

which would speed up their development. It would also be possible to arrange a series of levels with increasingly difficult terrain. With a simulation, such as that presented in chapter 4, it is also possible to predict the likely places the player will visit in a level.

The results and tools developed during this study may also have applications in domains unrelated to gaming where the layout is not conventionally described as a maze. For instance, supermarket aisles are usually fully braided mazes (see section 2.2). With some modifications, the simulator could be used to predict the likely paths of people for different store layouts.

1.3 Dissertation Overview

Chapter 2 defines mazes and their components and properties, as well as a few mathematical relationships between those properties. Chapter 3 presents hypotheses relating the difficulty of a maze to various properties of the maze, and explains the tests on the data that led me to reject them. Chapter 4 describes the simulator I built to give a Monte Carlo estimation of the difficulty of a maze. Chapter 5 explains my methods of gathering data and processing them. Chapter 6 contains suggested topics for further research, as well as presenting the tools developed for this study that could be used in such research. Finally, the conclusions are presented in chapter 7.

Chapter 2

Definitions

For the purposes of this study, a maze is a rectangular grid of cells, with every cell connected to the starting cell by at least one path. Cells can only be directly connected to the cells in their Von Neumann neighbourhood. In mathematical terms, a maze is an undirected, planar, and possibly cyclic graph whose transitive closure is a fully connected graph.

An overlap-step is any move made onto a tile previously visited. Overlap-steps are used to measure the difficulty of a maze. I have assumed this to be more reliable than measuring the time taken to complete the maze.

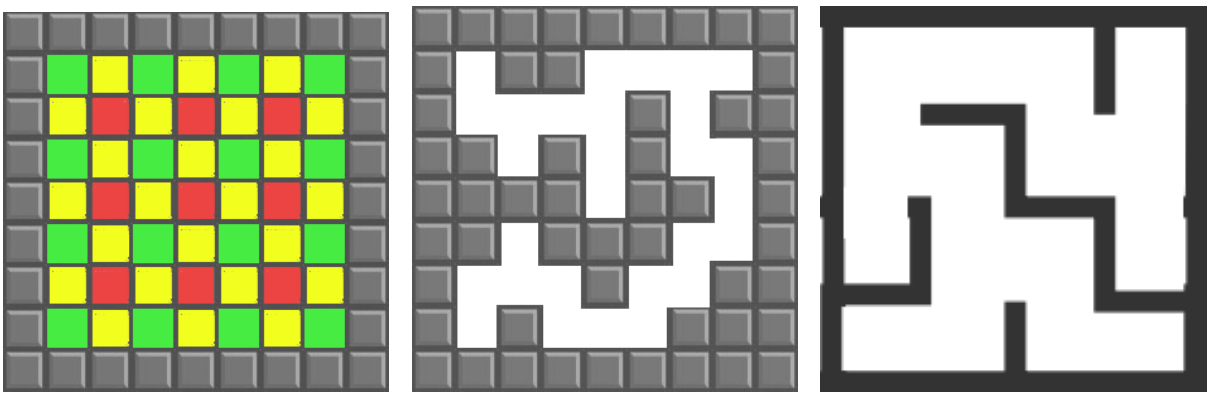
A dead end is any cell with exactly one connected neighbouring cell, but can also refer to the whole set connecting such a cell to the nearest split, in which case the size of the dead end denotes the number of cells from the end to the nearest split, excluding the split.

A split or junction is any cell with more than two connected neighbouring cells, with the exception of the starting cell, which is a split if there are two connected neighbouring cells. Using this definition allows us to equate splits with choices the player must make.

A passage is a set of cells between two junctions or between a dead end and a junction, including both end-points.

2.1 Representation

A maze as viewed by the test-subjects, consists of tiles, some of which are passable and some impassable. However, I have chosen to keep my mazes in a format such that every even-numbered tile in both dimensions is always impassable, and every odd-numbered tile in both dimensions is always passable — these are called



(a) Chosen maze representation (b) Restrictive maze representation (c) Functionally equivalent alternative

Figure 2.1: Representations of Mazes

cells. This is illustrated in figure 2.1(a), where cells are marked green, passages are marked yellow, and pillars — which are never passable — are marked red.

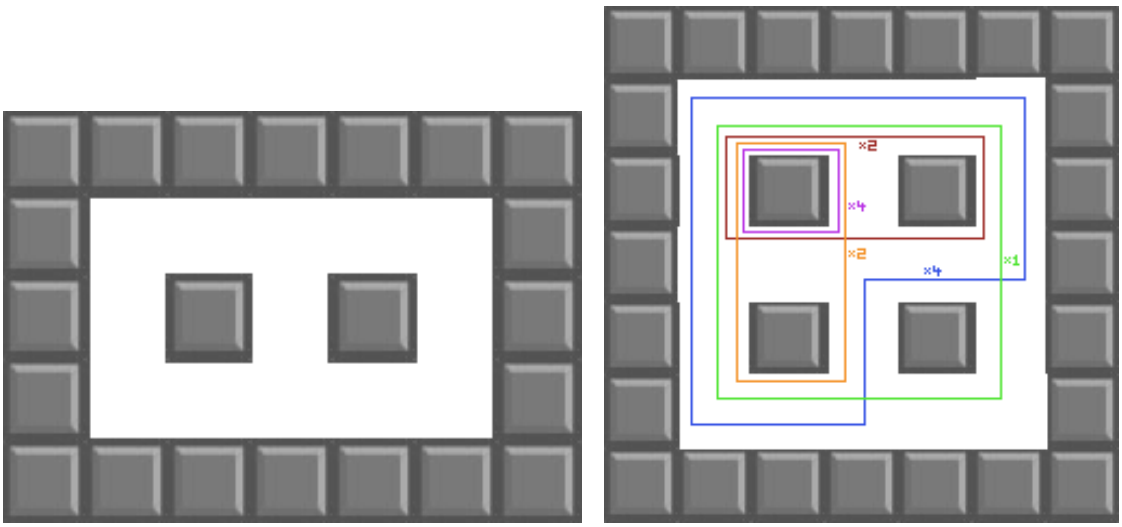
Two other commonly used representations were considered; the first, which can be seen in figure 2.1(b), differs from my representation by allowing turns at every cell, rather than only at odd-numbered cells in both dimensions. However, this actually restricts the number of mazes that can be represented because it is now required that at any corner, there be one cell that cannot be visited. Every maze represented as in figure 2.1(b) can be translated to a maze with the model using cells, pillars and passages (figure 2.1(a)), but not vice versa.

Another common representation is to draw lines for walls, as seen in figure 2.1(c) — this is a different visualization that is functionally equivalent. It can be achieved by stretching the tiles such that pillars are drawn small, cells are drawn as usual, and passages have one long and one short dimension.

The chosen maze representation I chose has the benefit of simplifying the implementation of the generation algorithms. Any maze generation algorithm using this representation is finished when all cells are connected to the beginning cell by some path.

2.2 Loops

Intuitively, a loop is a collection of distinct cells with each cell connected to the next and previous cells, and with the last cell also connected to the first. However, using this as the final definition for loops makes it difficult to count the number of loops when some of these intersect. Consider the maze in figure 2.2(a), where two minimal loops have been placed side by side. There are now three distinct such collections. A more extreme case is given in figure 2.2(b), where because of the intersection of 4 minimal loops, there are 17 possible looping paths.



(a) Two adjacent minimal loops (b) 3x3 matrix of minimal loops

Figure 2.2: Example of adjacent loops

To simplify, we count only *minimal loops* defined as followed; A minimal loop is a loop such that no subset of its cells also forms a loop. These can be calculated by $E - V + 1$, where E is the number of edges and V is the number of vertices in the maze graph [Ram12]. This equation follows from the Euler characteristic for planar graphs, which states that for planar connected graphs, $VE + F = 2$, where F is the number of faces, including the outer face [1]. We will henceforth refer to minimal loops simply as loops.

Note that loops can be found in a maze by finding walls that are disconnected from the outer rim. An easy way to measure the size of such a loop would then be the number of pillars this group is comprised of. However, to the player, it is less important how many walls a loop is comprised of, and more important how many cells are involved in the loop, since this is what will cause players to forget that they have come back to the same point they were on. I define the size of a loop as the number of cells enclosed within the path, which can be calculated by counting the unique cells neighbouring each pillar enclosed by the loop.

The loopcount of a maze is the number of loops in the maze. A *perfect maze* has exactly one path between any two cells in the maze, and hence, has a loopcount of 0. By contrast, (partially) *braided mazes* contain pairs of cells between which there is more than one path, with fully braided mazes being the most extreme case; these never have any dead ends. However, even fully braided mazes also contain pairs of cells with only one path between them.

[1] Minimal loops in a maze equate to faces in their graphs

Chapter 3

Hypotheses

The act of creating mazes can be seen as a game, with rules limiting possible moves; we have a fixed number of cells which must all remain accessible - and an outcome to maximize; the number of overlap-steps players make while trying to solve the maze. Testing the following hypotheses should help us to establish the trade-offs to be considered when constructing difficult mazes.

3.1 Loops and dead ends

It seems obvious to consider the number of choices present, i.e. the number of junctions in a maze, as a good explanatory variable for the difficulty of the maze, since each choice can be wrong, and the more incorrect choices, the higher the number of overlap steps and the more difficult the maze will be. The regression results for hypothesis 3.1 are given in table 3.1.

Unfortunately, although the R-squared value, the F-statistic and the p-values all make the model seem reasonable, it must nevertheless be rejected because the Quantile-Quantile plot for the model (see figure 3.1) indicates that the errors are not distributed normally, which violates one of the assumptions underlying linear regression.

Hypothesis 3.1. *The difficulty of a maze can be explained with a linear model based on the number of junctions.*

Similarly, trying to explain the difficulty with a linear model of either loops or dead ends, independently, yields non normal distributions of residuals and so must be rejected. I have included the results and QQ plot of hypothesis 3.1 only to illustrate my reasons for rejecting these three hypotheses.

One possible explanation for the bad fit of the junction model is that we are considering all loops to be equal. One differentiation we could make is in their length. Consider figure 3.2, where the pillars of each loop

	Dependent variable:
	overlapsteps
I(deadEnds + loops)	8.692^{***}
	(0.560)
Constant	-211.417^{***}
	(17.353)
Observations	1,485
R^2	0.140
Adjusted R^2	0.139
Residual Std. Error	116.683 (df = 1483)
F Statistic	241.057^{***} (df = 1; 1483)
Note:	*p<0.1; **p<0.05; ***p<0.01

Table 3.1: Regression results for linear model based on the number of junctions

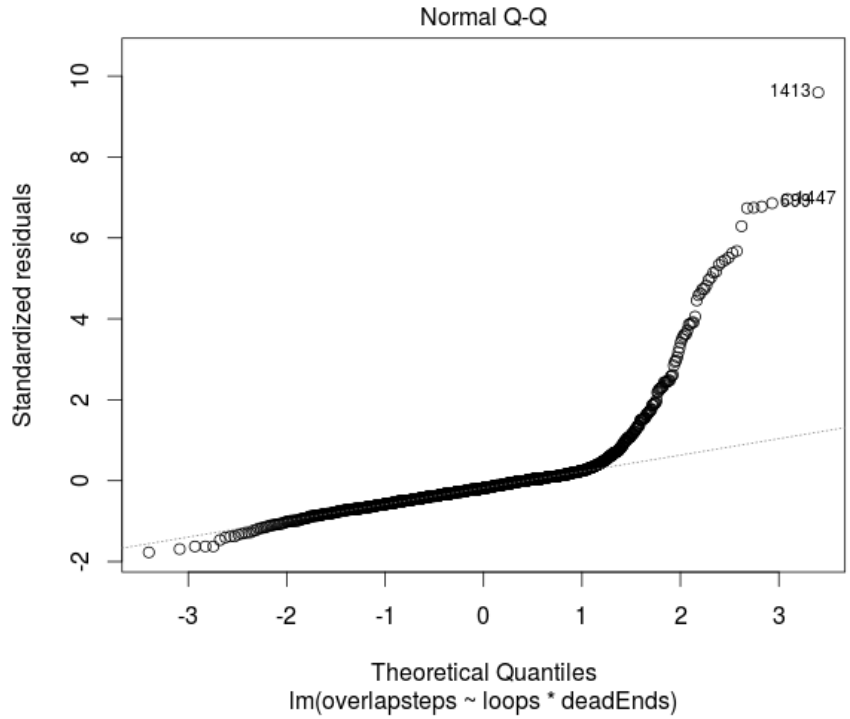


Figure 3.1: Quantile-Quantile plot for linear model based on the number of junctions. This plots the distribution of the actual residuals vertically against the theoretical distribution expected if the residuals were normally distributed.

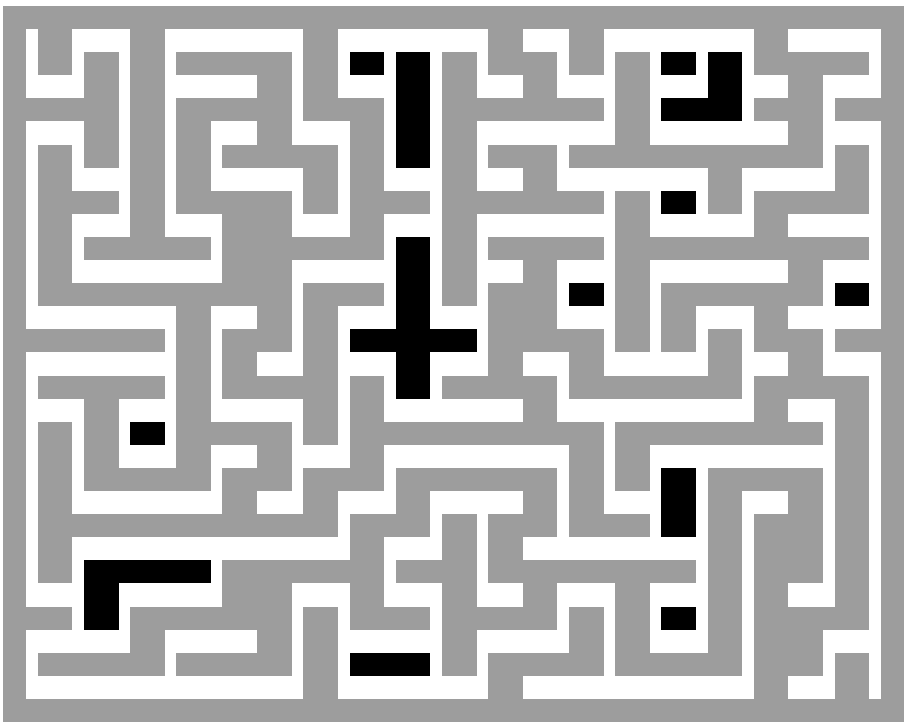


Figure 3.2: Some loops (coloured black) are larger than others

are coloured black. Surely, I thought, those tiny loops won't make the maze more difficult? This relates to the visibility range on the website — from any point you can see just beyond the cells in your Moore neighbourhood [1], so loops and dead ends of sufficiently short length should not fool players into making errors.

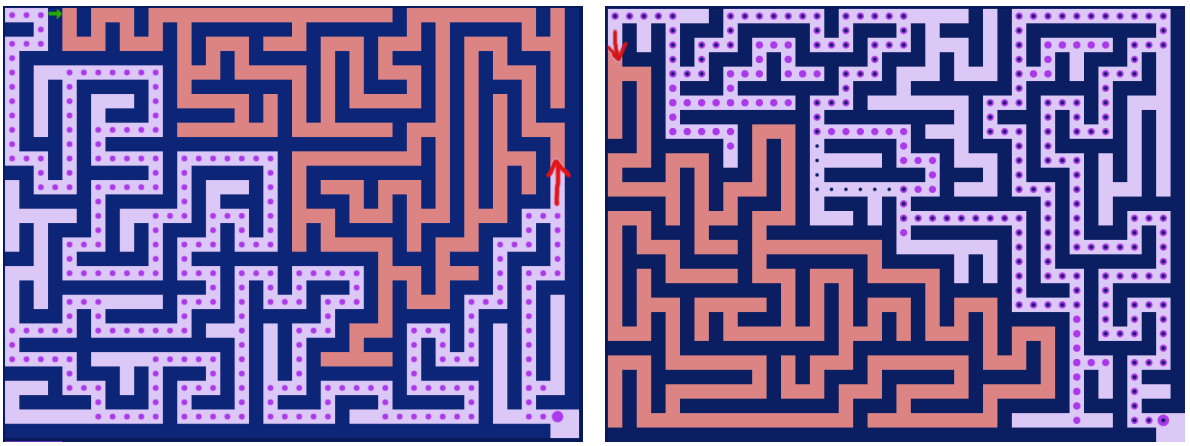
Hypothesis 3.2. *Small loops ($<$ 15 cells) and small dead ends ($<$ 5 cells) do not contribute to maze difficulty, while longer loops and dead ends do contribute.*

To test this hypothesis, a linear model was constructed that predicts the overlap steps based on only long loops, long dead ends, and their interaction. This model suffers the same fate as every model we have considered so far; the Q-Q plot has the shape of that in figure 3.1, so the model must be rejected.

Hypothesis 3.3. *Optimal difficulty is achieved by a balance between long dead ends and long loops; it is neither optimal to have 0 dead ends, nor 0 loops.*

It should be noted that this dichotomy is due to the definition of difficulty necessitating either dead ends or loops in order to achieve any difficulty at all, and that, in order to have more loops, you must necessarily have less dead ends. To test hypothesis 3.3 means to show that each of these variables does not predict the difficulty very well, but their interaction does. If the $Pr(> |t|)$ for each variable is significantly higher, this would support the model. Unfortunately, this would be an invalid comparison since, as stated before, the residuals are not normally distributed for any of the linear models considered thus far.

[1] the 8 surrounding cells with offsets of (-1,-1) to (1,1)



(a) A maze that effectively wastes half of its cells (b) A choice near to the beginning excludes half of the maze

Figure 3.3: Structural traits in mazes that are disregarded by examining mazes as a whole

3.2 Better models through simulation

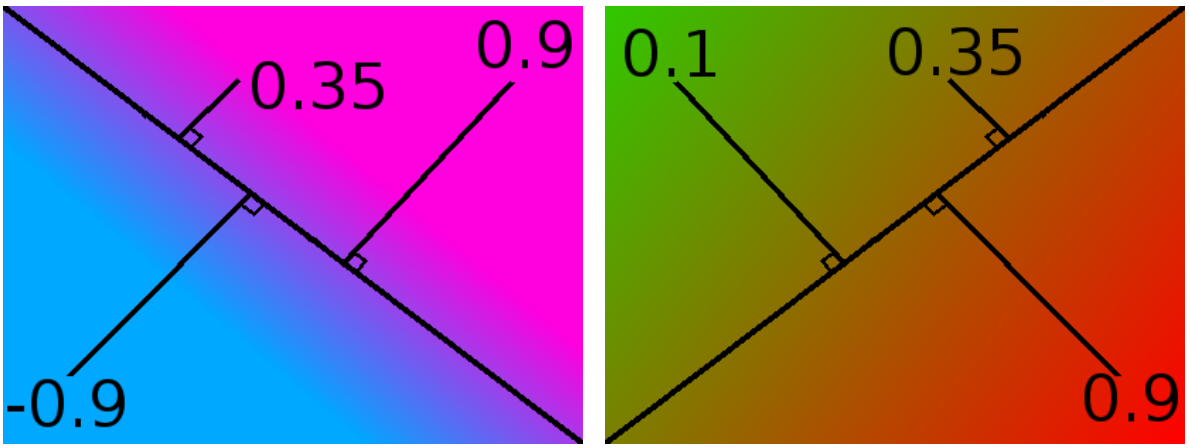
It occurs to me that the problem with the approach taken above lies in viewing the maze and its properties as a whole, because in doing so we discard the structure of the passages and junctions. It may be more fruitful to consider some base hypotheses about how people move inside a maze, and then construct a simulation that replays these in order to get an estimate on the difficulty. Such a stochastic model, even if somewhat inaccurate in the actual choices, will allow us to consider the morphological traits that can impact the difficulty of a maze.

To explain what this means, see the examples in figure 3.3. On the one hand, the maze may be offering choices such as the passage marked by a red arrow in figure 3.3(a), which the player is unlikely to choose because the exit is in the opposite direction. On the other hand, a maze may offer only perfectly reasonably choices, as in figure 3.3(b), but a choice near to the beginning could exclude half of the maze, creating a large variability in the actual difficulty of the maze. Each of these traits would go unnoticed when you examine the maze as a whole, which is why I have abandoned that approach in favour of simulation.

3.3 The first direction of choice

Hypothesis 3.4. *The direction chosen at a junction is correlated to the position of this junction within the whole maze.*

This hypothesis stems from the idea that when you are near the bottom-left of the maze, upon encountering a junction of some specific type, you will be more likely to choose to go right than down when compared to encountering the same junction from the same direction when you are near to the top-right of the maze, because the player knows which direction the exit is, and will greedily move towards the exit.



(a) Direction index, with purple representing 1 and blue representing -1

(b) Progress index, with green representing 0 and red representing 1

Figure 3.4: Two positional indexes for predicting players' first choices

However, nearer to the start of the maze, we expect to find less greedy strategies chosen because players will know that the maze is intended to be difficult, and will require unintuitive choices to solve. From examining some of the paths traced by players, I have noted a tendency to aim for the edges when near the beginning of the maze.

To test hypothesis 3.4, I calculate two floating-point indexes for each junction. The first is the direction index, and is calculated from the distance between the centre of the cell and a line stretching from the beginning to the end of the maze. The direction index ranges from $(-1, 1)$, with -1 in the bottomleft corner and 1 in the topright corner, as in figure 3.4(a). Similarly, the progress index seen in figure 3.4(b) ranges from $(0, 1)$ and represents the distance from the starting point using the other diagonal line for reference. This distance-to-line method is necessary to allow for non-square mazes.

Next, each junction is studied upon its first encounter by the player. It is split into one of 16 classes depending on the choices available at the junction, and the direction from which the junction was approached. It may be possible to improve the model presented in chapter 4 by replacing the assumptions with hard data on this hypothesis, which I leave unanswered.

Chapter 4

Difficulty by simulation

Using the results obtained in the previous chapter, it should be possible to estimate the difficulty of a given maze by predicting the paths taken by people and weighting each path's overlap steps by the probability of the path. This means basing our difficulty estimate for each maze on a Monte Carlo simulation, rather than on properties directly readable from the maze.

4.1 Approach

To predict these paths, we start by constructing a simulator for a player solving a maze, which navigates perfectly using the knowledge of only the paths it has already traversed. Then, we add more and more knowledge based on the data of real players and hopefully observe increases in the accuraccy of the simulator at predicting the paths of players.

To aid me in comparing human paths with simulated paths, I wrote a simple algorithm that draws the path as a randomized line with a shifting hue, as shown in figure 4.1. This makes it easy to distinguish the leading from the returning path, and allows me to quickly do qualitative analysis on a sample of human mazes done, as well as facilitating debugging the algorithm.

For gathering knowledge about how humans solve mazes, I looked in the data for instances of the same maze solved by many players. The mazes most often solved are listed in figure 4.6. For each of these, I traced the path with a different consistent colour for each person, and used a little ImageMagick [1] to compose them into single images — see figure 4.2 for an example.

[1] mogrify -format png -background transparent *.svg && convert -page *.png -background transparent -layers flatten composite.png

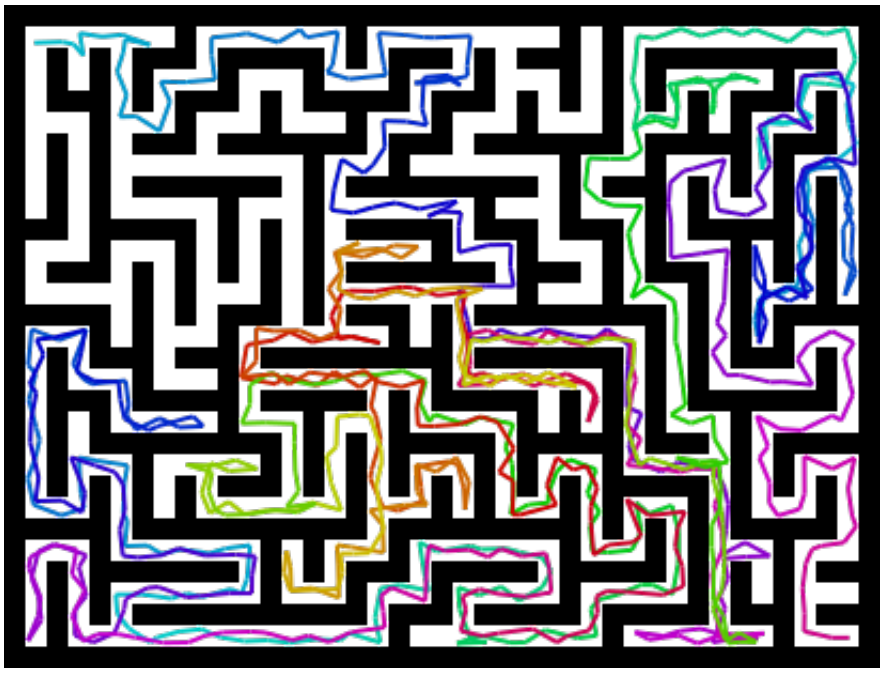


Figure 4.1: Example of path traced by a human participant

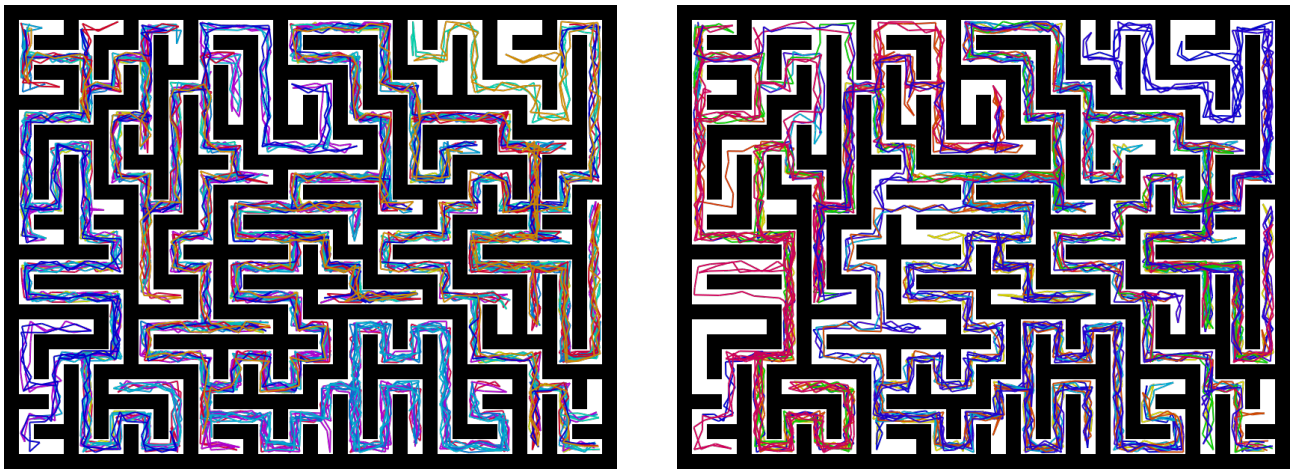


Figure 4.2: Paths taken by people in the same mazes are overlaid with a distinguishing colour for each player. Left; 0 loops, Right; 10 loops

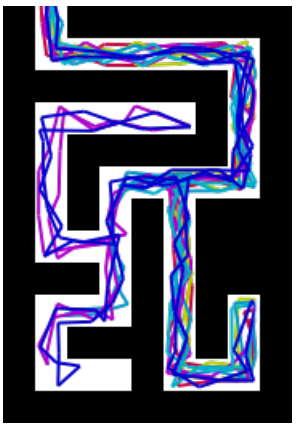


Figure 4.3: A corner of the maze with five players' routes traced

4.2 Cut off areas

Studying these images, I found a point of interest in the bottom-left of the maze. A simplified version is presented in figure 4.3. Here, we see 5 players entering the area from the top. Of these, 3 explore the left branch at the first junction, while 2 only explore the bottom branch.

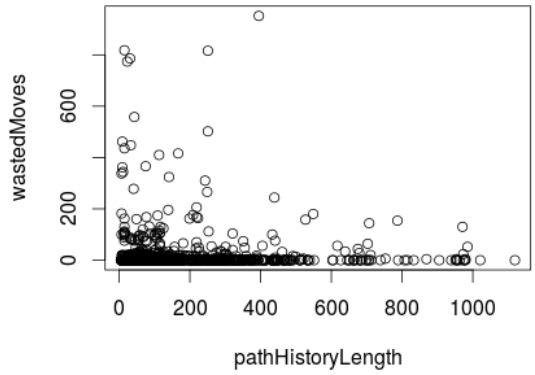
The phenoma I wish to study here is the human capacity to reason about areas of the maze that are cut off from the solution by the path already traced. In figure 4.3, if you have explored the bottom choice first, your path isolates the other choice from the solution location in the bottom right, and so one can reason that the choice is infeasible and may be skipped.

Conversely, if the solution location is *within* the area bounded by your path, one can reason that you must be on the path to the solution and will not need to backtrack further than the current point. It turns out that explicitly examining this case is not necessary, because if the solution is in an area bounded by the current path, then the alternative choice must also be bounded by the current path in an area that does not contain the solution.

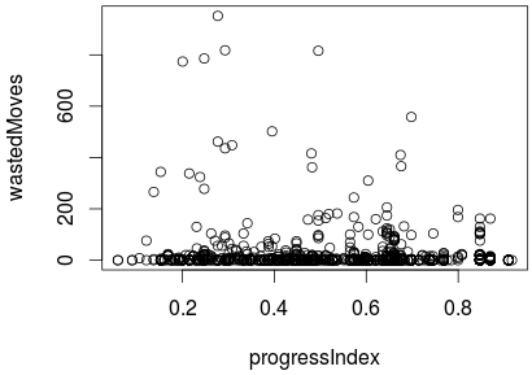
I propose a stochastic model for this based on the number of tiles in the bounded area such that a single cell is almost certain to be excluded from further search. The upper bound and probability should be based on statistical analysis of the human-made paths, as I have attempted in the following section.

4.3 Analysis of human performance

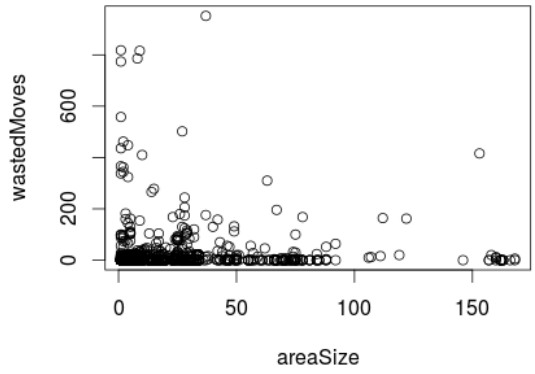
To research this, for each point in every path in the dataset, we determine whether the path bounds some area of the maze. If so, determine also; (1) whether that area includes the solution tile, (2) the size in tiles of the bounded area, (3) the oldest tile in the path we needed to remember in order to determine that the area is bounded, and (4) whether the player took the appropriate action (i.e. turn back, or don't examine a branch, depending on the situation).



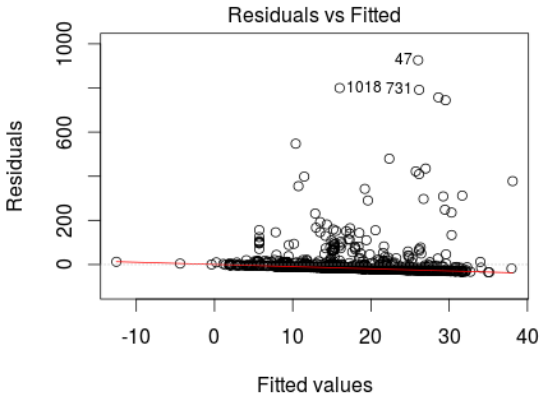
(a) Influence of the length of the path to be remembered



(b) Influence of the progress index on the moves wasted



(c) Influence of the size of the enclosed area



(d) Residuals of the 3-variable linear model

Figure 4.4: Results of modelling the number of moves wasted in a closed off area on properties of the decision tile

Turning the description above into an algorithm was no easy task. One simplification I applied was to disregard nested bounded areas, so only the largest of any set of nested bounded areas is examined. The implementation given in figure 8.2 determines for each tile in every path logged whether any available next tile is in a closed off area that does not contain the solution tile, and prints 4 numbers for each such instance; the progress index for the tile, the size in cells of the closed off area, the distance between the oldest tile needed to know the area is enclosed and the current choice tile, and the number of moves wasted on this closed off area. The results of the linear model constructed can be seen in figure 4.4. Unfortunately, the non-normally-distributed residuals indicate that the linear model cannot be used to explain the correlations present.

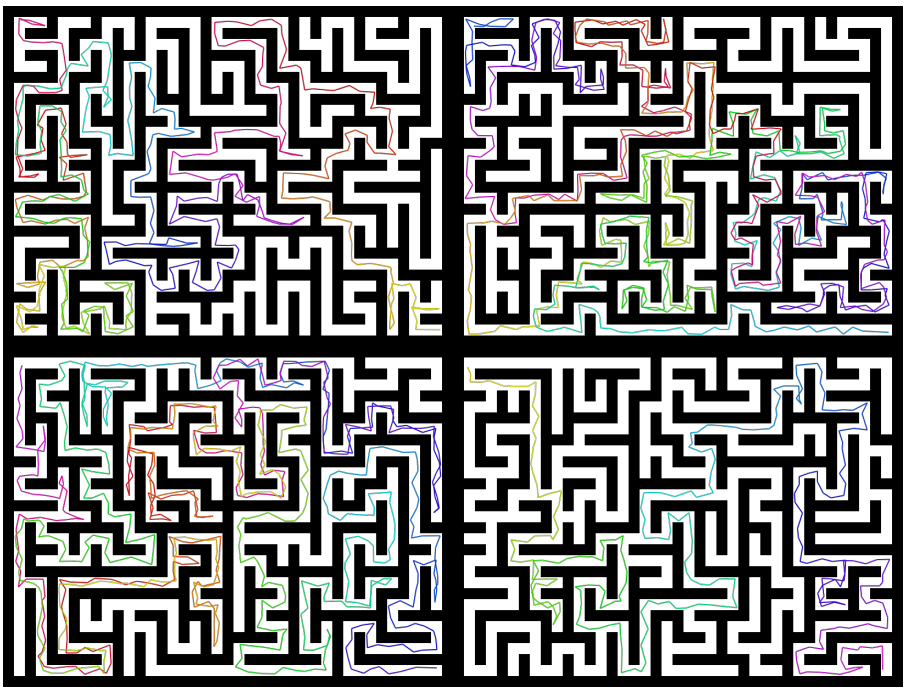


Figure 4.5: Four example paths generated by the smart directional simulator

4.4 Weighted Trémaux excluding bounded areas

I now present an algorithm to simulate human maze solving. It is based on Trémaux's maze solving algorithm [PT11], which proceeds thus; Whenever you arrive at or leave a junction, mark the direction you arrived from and the direction you departed in. A junction may have 0, 1 or 2 markings for each choice. When you arrive at a junction with any unmarked choices, pick from these unmarked choices at random. When you arrive at a junction with no unmarked choices, pick the choice marked once (there will only ever be one) to backtrack. If it does not exist, you must be back at the starting position, and there is no solution to the maze. There is one exceptional rule required only for (partially) braided mazes; if you are exploring a new path (i.e. at the last junction, there is only one marking for the choice you picked), but you encounter a junction with some markings already made, turn back and do not mark your entry or exit from this junction.

I made two additions to this algorithm; the first is a probability weight attached to each choice that is dependent on the directional and progress indices of the junction cell — based on the observation that humans tend to prefer to follow the heuristic of going in the direction of the solution whenever possible. This is called the directional bias. The second addition is the exclusion of all bounded areas not containing the solution tile from any search consideration. The modified algorithm is named the "smart directional simulator".

This algorithm, listed in figure 8.1, seems to generate reasonable walks through the maze, such as those in figure 4.5. But to properly test whether the simulation is representative of humans solving mazes, we need to first find a subset of the data for which many humans completed the same mazes. In figure 4.6 we see

```
+----------+----------+----------+
| mazetype | mazeseed | COUNT(*) |
+----------+----------+----------+
|        0 |        0 |       11 |
|        1 |        0 |        8 |
|        3 |        0 |        7 |
|        4 |        0 |       10 |
|        9 |        0 |        7 |
|       10 |        0 |       10 |
+----------+----------+----------+
52 rows in set (0.03 sec)
```

generated by hunt-and-kill

```
+----------+----------+----------+
| mazetype | mazeseed | COUNT(*) |
+----------+----------+----------+
|        9 |        1 |        7 |
|       10 |        1 |        7 |
+----------+----------+----------+
49 rows in set (0.00 sec)
```

generated by depth-first search

Figure 4.6: The most often solved mazes in the dataset

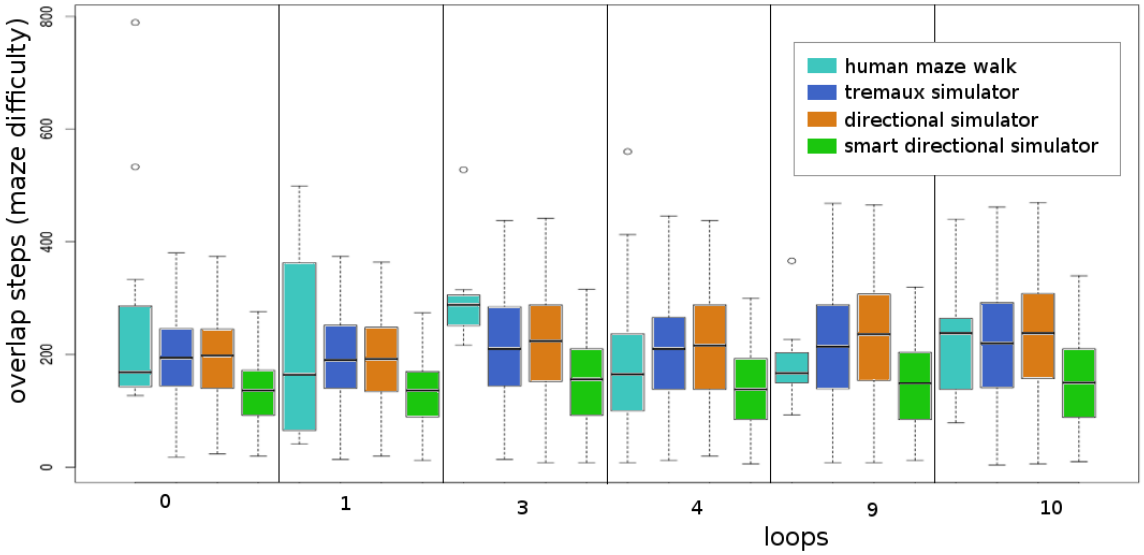


Figure 4.7: Comparison of the simulators versus the people solving the same 6 mazes

that of the 101 unique mazes solved in the second iteration of the algorithm, eight of these have at least seven participants. To further reduce the number of variables in the following comparison, I have only considered the results for mazes generated by the hunt-and-kill algorithm.

Next, the variance and mean is compared between human paths and those generated by simulation. It would be nice to do a Kolmogorov-Smirnov test of goodness of fit, but unfortunately there is not enough data from human participants to attempt such analysis. This was due to an error in judgement when constructing the second iteration of the website, where I was still under the impression that I would be studying direct correlation between maze properties and difficulty. In retrospect, I could dismissed depth-first-search and sorted into two buckets, one with few loops and one with many.

Running the three algorithms considered — Trémaux's original algorithm, the modification with directional bias, and the smart directional simulator — on the mazes given on the left side of figure 4.6 yields the results seen in figure 4.7. One thousand walks through each maze are done by each algorithm. Unfortunately, little can be said about the suitability

Chapter 5

Implementation

You should have received an accompanying .zip file containing the source code for the website, the analyzer, the simulator, and the scripts and readme's that accompany them. If not, please email t.mcglinn@gmail.com for a copy. Using this code it is possible to reproduce all of the results presented in this dissertation.

5.1 A unique approach

Studying the behaviour of humans is conventionally done with pen and paper, manually reviewing each test subject as he/she completes the task at hand. In that respect, my approach differs, using the more scalable tools available with the web to gather a much larger dataset than would be possible by conventional means, especially given my zero budget constraint. By constructing a website that logs data using asynchronous database calls, the player's experience is seamless and easy; all they have to do is follow a url and then navigate the mazes presented on-screen.

To advertise my website, I posted it on facebook, reddit, twitter, and a few forums related to mazes or gaming. The first version of my website reached 232 people, with 82 users completing at least 10 mazes.

5.2 Building the website

5.2.1 Language choice

Originally, I wanted to create the maze server-side in C++, using a php script that fetches the data and outputs to an html/js website which renders to the user and keeps talking to the server to fetch more mazes,

and say where you are etc. Jonathan Vis uses this technique in his Dou Shou Qi AI research. It's a very powerful stack for performance and security, but I realised early on that my project wasn't going to need much in the way of performance, and by moving everything to the client side, the solution I ended up with became a lot more scalable.

I also considered a radically different alternative; Unity webplayer would allow me to code in C# and quite quickly construct a 3d world, placing my test subjects in a more real environment. I chose not to implement this for two reasons; First, unity webplayer is not supported in all browsers, doesn't work on any linux based operating systems, and requires installation on the client computer, so I would lose quite a significant portion of my audience. Second, looking down on a maze and solving it is less frustrating, because you have a clear sense of progress - you can see how far and what direction the exit is. I was worried that if I placed users inside the maze, many would give up before long.

5.2.2 Security

The lack of security means visitors to my website could open up a javascript console and mess with all the variables and potentially change the maze. There are two ways to work around this problem; the first is to validate every move server side - this would be implemented in websites where cheating presents either high risks or high potential losses. The second is to let javascript store the moves the player does in an obscure, compact format to discourage players from altering it. This is called security through obscurity, and has the advantage that it is much easier to implement.

There are two possible problems with this solution; The first problem is players who cheat by changing their position in the maze. The analysis program I wrote nullifies this problem by verifying the paths traced by the player to ensure they do not walk through walls or off the map, and that they finish the maze in the correct end position. The second problem is the possibility to view the entire maze by turning off the visibility mask. This would lead to entries in the database reflecting that players immediately find the exit without retracing their steps at all. This is a valid issue, and the reason I strived to promote my website among non-programmer circles.

Due to the nature of my requirements, and the target demographic, I opted for the security through obscurity approach, combined with path verification. The client-side javascript implementation writes the full path the user walked to a database for later analysis, which is done with a standalone program written in C++ (MazeAnalyzer).

5.2.3 Ease of use

Early prototypes of my website required one keypress for every move in the maze. Having tested the website with a few friends as guinea pigs, I concluded that this was going to severely decrease the number of mazes people will be willing to complete. My subsequent searching on google turned up no solutions to this apparently simple and, one would think, common problem. The solution I settled on was to track the state of each of the relevant keys by listening to the keyup / keydown events. This implementation can be found in code listing 5.1.

Another issue I faced was that *people don't read* [Kruoo, p. 21]. I received a few responses from visitors saying they did not understand how to walk in the maze, and one possible explanation was that they did not read the very short introduction on the frontpage explaining which keys move your character. With the new version, I created a help section on the same page as the maze — not on the index page which has been skipped unread — which may be unfurled at the click of a button, but which unfurls itself after 3 seconds if you are still on the starting tile.

I figured that the best way to catch the attention of a user unable to operate the page is with something that moves. Similarly, when you complete a maze, a new button appears right beside your character that allows you to load the next maze. This is necessary because of the added functionality that displays your path right after completing a maze — instead of being instantly transported to the next maze as soon as you hit the solution cell.

5.3 A new version

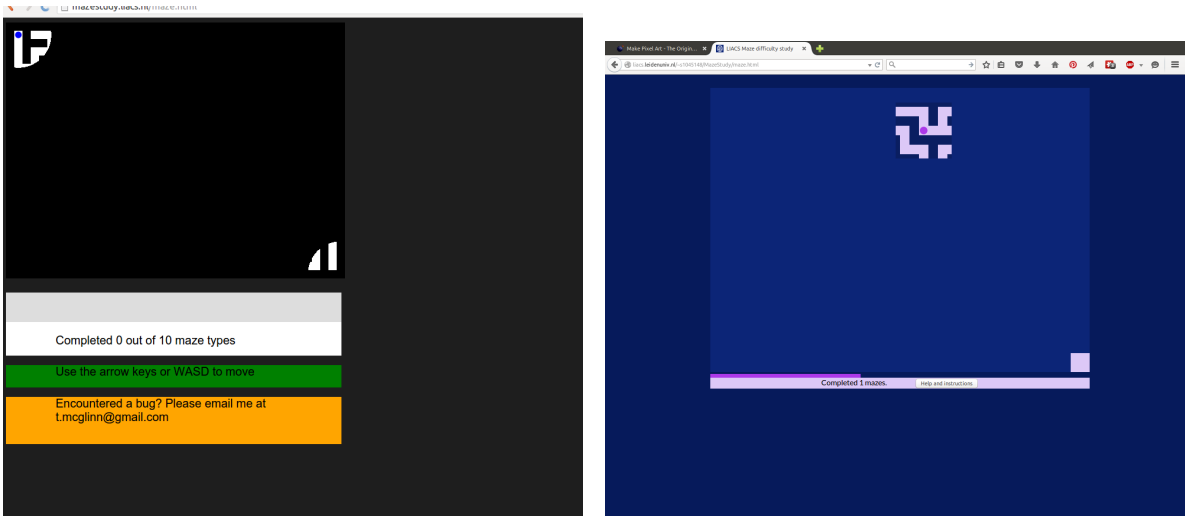
A few problems with the website prompted me to made a second, improved version. My first website version made the mistaken assumption that I would gather so much data that with it that it would possible to analyse many different explanatory variables without needing to restrict the mazes shown at all — all mazes were generated at random, so few mazes were ever completed by more than one visitor. In the second version, I made all users view the same sequence of mazes, with only the number of loops being randomized. With several users making the same choices, I can say something statistically relevant about what choices people make at each junction, as well as being able to test my hypotheses relating to dead ends versus loops.

A second problem with the data was that I was not capturing the path completely; the point at which a user turned back and went the other direction was not registered, so there was no way to tell exactly how many steps the user had taken down a passage.

The other problems I addressed were related to the usability and visual appeal of the website — see figure 5.2

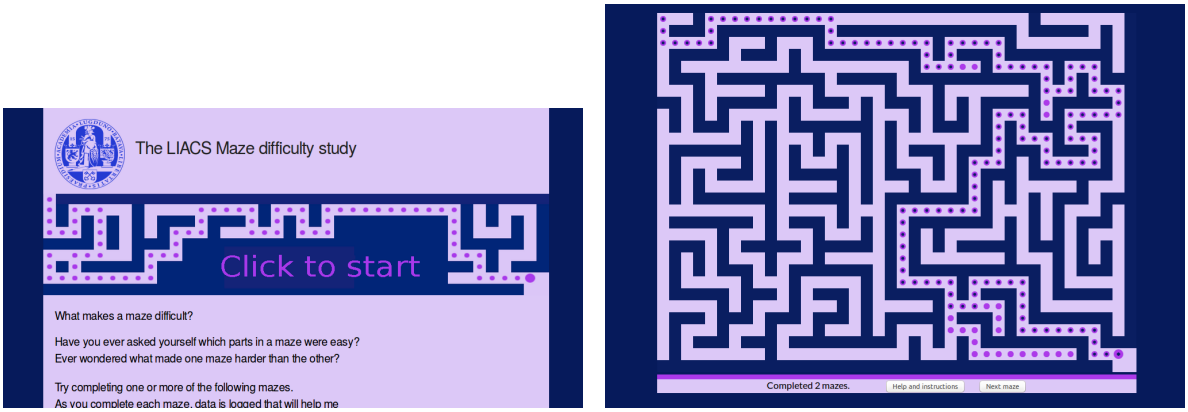
```
var keyList = [];
var moveIfKeyDownIntervalId;
var keysToCheck = [38, 87, 37, 65, 40, 83, 39, 68];

function clearMoveRepeat(){
  if(moveIfKeyDownIntervalId){
    clearInterval(moveIfKeyDownIntervalId);
  }
}

function clearAllKeys(){
  for(var i = 0; i < keysToCheck.length; ++i) {
    keyList[keysToCheck[i]] = false;
  }
}

function onKeyDown(e) {
  if (!e) e = window.event;
  if( keyList[e.keyCode] ){
    return; //disable repeat keys of user's OS
  }
  clearMoveRepeat();
  keyList[e.keyCode] = true;
  moveRect(e.keyCode);

  moveIfKeyDownIntervalId = setInterval(moveIfKeyDown,150);
}

function onKeyUp(e) {
  if (!e) e = window.event;
  keyList[e.keyCode] = false;
}

function isKeyDown(keyCode) {
  return keyList[keyCode];
}

function moveIfKeyDown(){
  for(var i = 0; i < keysToCheck.length; ++i) {
    if( isKeyDown(keysToCheck[i]) ){
      moveRect(keysToCheck[i]);
    }
  }
}

window.addEventListener("keydown", onKeyDown, false);
window.addEventListener("keyup", onKeyUp, false);
```

Figure 5.1: Excerpt from maze.js that handles key state registration



(a) First version (b) Second iteration

Figure 5.2: Maze page, old and new versions



(a) New landing page (b) Display after maze is completed

Figure 5.3: Features of the new maze website

for a side-by-side comparison of the old and new maze page. With the new version, I also took extra care to entice new visitors with a more appealing frontpage explaining my motivation for the research, and to reward each completed maze by showing the path traced versus a shortest path (see figure 5.3).

Unfortunately, I received limited response to my second version of the website. Despite receiving slightly more front page hits (280 versus 232), only 240 mazes were completed, versus 1235 with the old version. I attribute this mostly to the increased difficulty of the new mazes, which means that for the same time spent fewer mazes were actually completed, and that some people would have given up before completing the first maze. However, it is impossible to tell what the real reasons are, since I changed many things at once.

5.4 Maze generation

Each maze is first generated in javascript and presented to the user to solve on the website, and later re-generated by the analysis program to examine the properties of the maze, and to retrace the steps of the player. It turned out to be a major challenge to replicate the exact same behaviour in so many parts of the codebase. In retrospect, it would have been an excellent idea to generate the javascript code from the C++ implementation using e.g. cheerp [che], rather than trying to manually maintain consistence between two different code bases.

```
void dfsSkewed(maze<xSize, ySize> &target, int skewNeighbours[]) {
  std::stack<coord> cellStack;

  coord currentCell(1, 1);
  coord newCell;

  cellStack.push(currentCell);

  target.set(currentCell, false);

  while (true) { // NB: exit point when cellStack.empty() below
    const std::vector<coord> options =
        target.getAvailableNeighbours(currentCell, skewNeighbours);
    if (options.empty()) {
      if (cellStack.empty()) {
        return; // DONE
      } else {
        currentCell = cellStack.top();
        cellStack.pop();
        continue;
      }
    }

    newCell = options[randomNumber() % options.size()]; // random element
    target.connect(currentCell, newCell);
    cellStack.push(newCell);
    currentCell = newCell;
  }
}
```

Listing 5.1: Skewed Depth-First maze generation algorithm

In the first iteration, I was focusing on differences between mazes, so I created 10 different maze algorithms, where each user completes one of each type (ideally). The maze algorithms are all based on depth-first

search, and differ in two respects. The first is the nature of their loops. Some add loops depending on the length of the current passage, others on the current position. The second difference is in the probabilities assigned to each direction. The code in listing 5.1 relies on a set of positive integers indicating how many of each neighbour to insert (when available).

After the first round of statistical analysis, however, I realised the real topic of study here is the variance among humans solving the mazes, so I trimmed down the implementation to just two maze algorithms — in hindsight, one would have sufficed. With these two maze algorithms, I still wanted to generate various numbers of loops. The original had some mazes with many loops, and some mazes with none at all but very little in-between, making accurate statistical analysis impossible. To address this, I changed the meaning of the mazetype to be the number of loops added after a perfect maze has been generated. This way, I did not have to modify the data structure or the SQL code — which would make it a pain later to study both datasets together. I also got rid of the directional skewing in the generation algorithms — it would be interesting to pick just this to vary among mazes, to see how humans respond to such morphological differences in mazes.

5.4.1 Reversing wasted turns

In the maze generating code, you will find a post-processing step I added after the first iteration of the website that I call "reversing wasted turns". Refer back to figure 3.3(a) for an example. This junction has been wasted, I deem, because it offers an unreasonable alternative - going up, when the exit is below. In the second iteration of the website, I added the `reverseWastedTurns` function to make the mazes a little bit harder. For each junction on the first[1] shortest path to the solution, the function checks for junctions that offer options going up or to the left.

These are **replaced**, which means that another connection must be found that also connects the area — coloured pink in figure 3.3(a) — that would be cut off from the rest of the maze by removing this junction. The stand-in tile is determined as the first neighbour of the given solutionpath that can offer a choice into the area, such that that choice points either down or to the right. In figure 3.3(a), the small green arrow in the top left would replace the big red one in the bottom right of the maze.

[1] Effectively random, when there are multiple shortest paths

Chapter 6

Further research

The dataset provided with this dissertation may be further exploited to find correlations I was unable to find, and to further improve the simulation algorithm to be more humanlike. In a more general sense, my approach and toolset could be applied to different areas of research. In order to do so, this chapter will help you find your way around the data, and will explain in further detail how to use the tools presented. In conclusion I explain the mistakes I made, but you can avoid.

6.1 Suggested topics

The dataset on the first decisions people pick upon encountering a junction probably contains much information that could be used to improve the simulator, which currently uses a guesswork probability function. Another possible variant of the simulation strategy could take other heuristics into account, such as: "if the last junction was much closer to the solution than my current position, turn back and try a different option". Although I spotted this in some paths traced by humans, it is difficult to adapt the algorithm so much and still guarantee termination. For the same reason, I have kept the original exception in Trémaux's maze solving algorithm for encountering an already visited junction from a new path, even though this is unlikely to be observed in real human cases. Further development of the simulation algorithm in such paths may lead to better estimation, especially of the outliers present in the dataset.

It would be interesting to see what happens to hypothesis 3 when the length of each loop or dead end is split into more than 2 buckets. In other words, modify the program to output loops of length $> X_i$ for some vector X of lengths, and see if the model becomes more or less statistically relevant for different sizes of loops to be considered.

In my analysis, I have ignored the fact that players can see the cells in their Moore neighbourhood. It would be interesting to expand the smart directional solver algorithm to take this information into account, as that would allow it to reject even more paths. It may also help to explain some outliers where humans appear to perform extremely well compared to the algorithms tested against them.

Using the time between each decision, the website could be modified to log more information that may be used to determine when a player is thinking about his choices and when he is "blindly" searching for the correct path. I have not used time in any of my research, although I do log the total time it takes to complete each maze.

It would be very interesting to consider the same research I am doing with a different maze topology, because with square two dimensional mazes, splits can only have either 3 or 4 available neighbours. If you would allow bridges that can span over one tile leading to the next - as are frequently used in real-life corn mazes - that becomes 3-8, and so the variance in splitting factor could be studied for its contribution to maze difficulty.

Another totally different approach could be attempted. By my definition of maze difficulty, a labyrinth - i.e. a maze with no junctions at all - should have a difficulty of zero. Instead of analysing the difficulty of a whole maze, which necessarily means dealing with a lot of noise, you could try analysing the difference in difficulty caused by various transformations, such as the difficulty increasing transformation "reverseWastedTurns" given in section 5.4.1. A naieve starting point would be to assume equal chances for all options at each junction. This may be enough to predict the practical difficulty[1] of mazes with more accuraccy than I have done so here, although more thought will need to be given for loops.

6.2 Setting up the website

With the sql statements in listing 6.1, you can set up a database locally or on your server. The relevant code is all in `code/htmlmaze/website/database`; the javascript code running on the client will call ajaxSend.js, which invokes enterdata.php asynchronously. The ajaxSend code can be extremely simple because the client javascript doesn't actually care about the results of the query, it's strictly fire-and-forget.

```
1 CREATE TABLE visits (ip BINARY(16),time DATETIME) ENGINE MyISAM;

3 CREATE TABLE mazewalks (ip BINARY(16),mazetype SMALLINT,mazeseed int,
  finishtime int,steps SMALLINT,overlapsteps SMALLINT,previousmazes SMALLINT)
5 ENGINE MyISAM;
```

Listing 6.1: SQL statements to create the two tables I used to gather data

[1] The average overlap steps of many people completing the same maze

The best way I have found for developing such code is to connect it to a local MySQL server that is opened on the terminal, and open the webpage in Google Chrome, since it has the best javascript console, featuring auto-completion and code-stepping.

6.3 Data formats

While the MazeAnalysis and Simulator source code contains no platform specific code, I have not tested compiling under Visual Studio for the latest versions of the code, so the instructions I give here for Linux computers may need adjustment to compile under other operating systems. The only dependency is to the Templatized C++ Command Line Parser (TCLAP), which can be downloaded from sourceforge, and installed with the standard "`./configure && make install`" on Linux distributions. Next, open the command line in the Debug/ directory of the accompanying source code, and issue "`make && make sim`" to compile both the MazeAnalysis and the Simulator programs.

The MazeAnalysis program reads input from stdin and outputs three sections of data. The first section is formatted like in table 6.1, and has been put into files that can be identified by the suffix "results.txt". The first 7 parameters, from ip (Identifier for Person) through walkedpath, are the input to the program, while the trailing parameters are generated by the program. So, the corresponding input file always consists of the first 7 columns of the output file. I chose this format in order to eliminate the need for keys that need to be indexed between files. With the input included in the resulting data, analysis using R becomes easy, even when you want to combine input and output parameters in your model (e.g. ip and loops, to assess the variance in how different people deal with loops).

	ip	timestamp	oldornew	mazetype	mazeseed	previousmazes	walkedpath
3	visitor141	1,432,192,972	0	3	4,866	2	ttttrrrrtldr
4	visitor141	1,432,192,996	0	2	534	3	rrttttttttdttltdrttddd
5	visitor160	1,432,190,306	0	0	6,152	0	rrdruutdruurrrrd
6	visitor160	1,432,190,332	0	1	1,718	1	ddddrrrtdrdrrddr

	overlapsteps	deadEnds	longDeadEnds	startSplit	splits	averageSplitFactor	loops	longLoops
3	5	25	1	0	25	3	0	0
4	17	27	3	0	31	3	2	2
5	6	29	2	0	29	3	0	0
6	2	0	0	0	59	3.017	30	11

Table 6.1: Combined result format

The second section of the MazeAnalysis output contains the first choices people have made for each junction. This is split into four columns; `junctiontype directionIndex progressIndex directionPicked`. The junction type is an integer between 0 and 19, dependent on the choices available and the direction from which it is approached. See listing 6.2 for the mapping between junctions and their indexes.

```
std::vector<FirstChoice> &getChoiceStorage(const std::vector<coord> &choices,
                                           coord currentTile,
                                           direction incomingDir) {
    if (choices.size() == 4) {
      return choiceTypes[(int)incomingDir];
    }
    if (std::find(choices.begin(), choices.end(), currentTile + coord(0, -1)) ==
        choices.end()) {
      return choiceTypes[4 + (int)incomingDir];
    }
    if (std::find(choices.begin(), choices.end(), currentTile + coord(0, 1)) ==
        choices.end()) {
      return choiceTypes[8 + (int)incomingDir];
    }
    if (std::find(choices.begin(), choices.end(), currentTile + coord(-1, 0)) ==
        choices.end()) {
      return choiceTypes[12 + (int)incomingDir];
    }
    assert(std::find(choices.begin(), choices.end(),
                     currentTile + coord(1, 0)) == choices.end());
    return choiceTypes[16 + (int)incomingDir];
  }
```

Listing 6.2: Code mapping junction types to indexes

The third section of the MazeAnalysis output contains the decisions made by players when encountering a junction for which one junction leads to an area cut off from the solution tile by the path thus far, as described in more detail in section 4.2. The format is `progressIndex areaSize pathHistoryLength wastedMoves`.

There are a number of data files in the accompanying archive of the form [somename]1000-2-0.txt. These are paths generated by the corresponding algorithm, and may be regenerated by the command `./Simulator -l 2 -s 0 -a -w` On a modern computer, this should take less than 10 minutes, but you can decrease the sample size n for faster results. Consult `./Simulator --help` for more information on usage.

If you wish to view some paths traced by people in the dataset, you can print a selection visually with a command such as `head -n 5 ../../../../data/newinput.txt ./MazeAnalysis -d|`. This will print four mazes to the traces folder (which must exist beside the executable) with names identifying the visitor and the time when the path was logged to the database. Similarly, the Simulator program supports the -d option

6.4 Tools

The perl script given in code listing 6.5 is intended for replacing the ip-address at the start of each line of a dataset with a string identifier, in order to include the dataset in a publication without compromising your users' security. First, generate the keyfile, which should map an ip to an identifier string and should be named "obfuscationkeys.txt" and be in the same directory as the script. Next, run the script with a data file as the first and only parameter, piping the output to a new file, ready for inclusion in your publication. See listing 6.3 for example commands for the supplied dataset.

In processing the data into a form usable for statistical analysis, I have made extensive use of awk and sed one-liners. A few examples are given in listing 6.4, but interested readers should consult [DR].

```
cat olddata.txt newdata.txt | awk ' { print $1 } ' | sort | uniq > iplist.txt
cat iplist.txt |  awk ' BEGIN { id = 1 }; { print $0":visitor"(id++); }
 ' > obfuscationkeys.txt
./obfuscate olddata.txt > obfuscatednewdata.txt
```

Listing 6.3: Obfuscate usage example

```
cat data.txt | awk ' { print $1 } ' | sort | uniq > ipadresses.txt
cat newinput.txt | awk ' { if( $4 == 10 && $5 == 0 && $6 % 2 == 0 ){ print $0 } } ' > common
    -10-0.txt
sed -i '1i ip timestamp etc. header' fileneedingheader.txt
```

Listing 6.4: Some useful awk commands for transforming data

```
#!/usr/bin/perl
# found on StackOverflow, written by asf107
# http://stackoverflow.com/questions/11617402/search-replace-using-keyvalues-from-file-in-bash

use strict;
my %keys2values;

# read through the pairs file to get the key:value mapping
open PAIRS, "cat obfuscationkeys.txt |" ;
while(<PAIRS>) {
    chomp $_;
    my ($key,$value) = split(":",$_);
    $keys2values{$key} = $value;
}

open KEYS, "cat @ARGV[0] |";
while(<KEYS>) {
    chomp $_;
```

```
19      my $inputline = $_;
        my @list = (split /\s/, $inputline);
21      my $key = @list[0];
        if(defined $keys2values{$key}) {
23          print "$keys2values{$key} ";
        }
25      # if a key:value pair isn't defined, just print the key
        else {
27          print "Error obfuscating $key\n";
        }
29      foreach my $i (1..$#list) {
            print "@list[$i] ";
31      }
        print "\n";
33  }
```

Listing 6.5: Obfuscate script

6.5 Improvement

Think carefully about what data you want to include in the database calls, and note that it is almost certain that these requirements will change once you examine some of the data. In fact, you should start by just inventing some data, so that you can first run through the analysis phase. It is only when you have to draw conclusions that you realise what was missing from the data in the first place. In retrospect, I should have discarded the idea of correlating first properties of mazes to their difficulty much earlier, and geared my data gathering towards getting as many people as possible doing exactly the same mazes.

The code for logging and replaying decisions becomes significantly more complex when you allow a split to occur on the very first tile of the maze, because this creates a cell that is a junction even though it only has two neighbouring cells connected to it. In retrospect, I should have taken the simple route and fixed the starting cell to always be connected to one of its neighbours.

Chapter 7

Conclusions

Having studied the most promising properties of mazes — loops, dead ends and junctions — for correlation with maze difficulty, we must conclude that such a simple approach will not suffice. The simulation models, based on Trémaux's maze solving algorithm, are found also to be insufficient, especially for reproducing the outliers present when humans solve mazes, although the smart directional simulation algorithm is useful in its own right for maze solving with partial information.

Chapter 8

Code listings

8.1 Weighted trémaux algorithm excluding bounded areas for simulating players' paths

```
1
void chooseNext(coord &previousPosition, coord &currentTile, coord c) {
3   previousPosition = currentTile;
    currentTile = c;
5 }

7 template <typename PickFunc>
void genericTremauxWalk(const maze<xSize, ySize> &grid,
9                         std::vector<coord> &visitedTiles,
                          PickFunc directionPicker, bool walkIntoCutoffAreas) {
11   coord currentCell = coord(1, 1);
    const coord invalidPosition = coord(-1, -1);
13   coord previousPosition = invalidPosition;

15   while (!grid.isMazeEnd(currentCell)) {
      if (previousPosition != invalidPosition) {
17      visitedTiles.push_back(coord::avg(previousPosition, currentCell));
      }
19    visitedTiles.push_back(currentCell);
      auto options = grid.getWalkableNeighbours(currentCell);
21
      // essential in mazes with loops; otherwise, we may leave out paths and
23    // backtrack too early
      options.erase(
```

```
25        std::remove_if(options.begin(), options.end(),
                         [currentCell, &visitedTiles](coord candidate) {
27                         // return false; // if you don't believe me, try this
                           // out with srand(5); mazetype = 3; mazeseed = 2;
29                         const coord boundary =
                               coord::avg(currentCell, candidate);
31                         if (std::find(visitedTiles.begin(), visitedTiles.end(),
                                         boundary) ==
33                             visitedTiles.end()) { // unvisited boundary
                             // we're on a new path; treat any already seen
35                           // junctions as dead ends and remove them from the
                             // options
37                           if (std::find(visitedTiles.begin(),
                                           visitedTiles.end(), candidate) !=
39                               visitedTiles.end()) { // to visited cell
                             return true;
41                           }
                           }
43                         return false;
                         }),
45        options.end());

47    // exclude options cut off from the solution if we've been told to ignore
      // such options
49    if (!walkIntoCutoffAreas && visitedTiles.size() > 1) {
        options.erase(
51          std::remove_if(
                options.begin(), options.end(),
53              [&visitedTiles, &grid, currentCell](coord possibleOption) {
                  if (std::find(visitedTiles.begin(), visitedTiles.end(),
55                              possibleOption) != visitedTiles.end()) {
                    // don't consider options we've already visited for the
57                  // expensive EnclosedArea calculation below;
                    // if it were enclosed, we couldn't have already visited it.
59                  // (reconsider this if the skipping becomes probabilistic)
                    return false;
61                }
                  PathIter oldestNeededPosition;
63                auto area =
                      getEnclosedArea(grid, visitedTiles, visitedTiles.end() - 1,
65                                    possibleOption, oldestNeededPosition);
                  if (std::find(area.begin(), area.end(),
67                              grid.solutionPosition()) == area.end()) {
                    // TODO add factor for "intelligence" instead of always
```

```
69                  // recognising cut off area,
                    // possibly using area.size() and
71                  // distance(oldestNeededPosition,visitedTiles.end())
                    return true;
73                }
                  return false;
75              }),
          options.end());
77    }

79    size_t choiceCount = options.size();
      if (choiceCount > 2 ||
81        (choiceCount == 2 && previousPosition == invalidPosition)) {
        // std::tuple<int,coord> leastVisitedPosition { 9999, invalidPosition };
83      std::vector<coord> neverVisited;
        std::vector<coord> onceVisited;
85      for (auto c : options) {
          if (c == previousPosition) {
87          continue;
          }
89

          // the count is made based on the boundary tile, not either cell
91        // because the cell on the other side may also have multiple options
          // that been visited before
93        // In Tremaux's algorithm, this count represents the marking of the
          // junction for each direction
95        int visitCount = std::count(visitedTiles.begin(), visitedTiles.end(),
                                      coord::avg(currentCell, c));
97        // if( visitCount < std::get<0>(leastVisitedPosition) ){
          //  leastVisitedPosition = std::make_tuple(visitCount,c);
99        // }
          if (visitCount == 0) {
101         neverVisited.push_back(c);
          } else if (visitCount == 1) {
103         onceVisited.push_back(c);
          }
105     }

107     if (!neverVisited.empty()) {
          // prefer places never visited before
109       chooseNext(previousPosition, currentCell,
                     directionPicker(neverVisited, currentCell));
111     } else {
          // otherwise, there must be at least 1 place only visited once
```

```
113         // (which is the way back, once this whole area has been searched)
            if (onceVisited.empty()) {
115           return;
            }
117         chooseNext(previousPosition, currentCell,
                       directionPicker(onceVisited, currentCell));
119       }
      } else if (choiceCount == 2) {
121       for (coord c : options) {
            if (c != previousPosition) {
123           chooseNext(previousPosition, currentCell, c);
              break;
125         }
          }
127     } else {
          assert(choiceCount == 1);
129       // hit a dead end
          chooseNext(previousPosition, currentCell, options[0]);
131     }
    }
133   visitedTiles.push_back(currentCell);
  }
135
  coord pickUniform(std::vector<coord> choices, coord currentTile) {
137   return choices[rand() % choices.size()];
  }
139
  coord pickBiased(std::vector<coord> choices, coord currentTile) {
141   float preferenceMultiplier = progressIndex(currentTile); // range [0 ; 1]
    float directionalPreference =
143       directionIndex(currentTile); // range [-1 (bottom left) ; 1] (top right)
    float positionalModifier =
145       preferenceMultiplier *
          directionalPreference; // range approximately [-0.5 ; 0.5]
147
    // TODO these values and this calculation is due to a lot of guesswork
149 // it could be used to create more customised algorithms with preferences
    float downChance = 4;
151 float rightChance = 4;
    float leftChance = 2.5;
153 float upChance = 2.5;

155 downChance += positionalModifier * 6;
    rightChance -= positionalModifier * 3;
```

```
157   leftChance -= positionalModifier * 3;
      upChance -= positionalModifier * 6;
159
      // cout << downChance << ", " << rightChance << ", " << leftChance << ", " <<
161   // upChance << endl;

163   float totalChance = 0;
      std::vector<std::tuple<float, coord>> choiceChanceValues;
165   for (coord c : choices) {
        const coord diff = currentTile - c;
167     if (diff == coord(2, 0)) { // left
          totalChance += leftChance;
169       choiceChanceValues.push_back(std::make_tuple(leftChance, c));
        } else if (diff == coord(-2, 0)) { // right
171       totalChance += rightChance;
          choiceChanceValues.push_back(std::make_tuple(rightChance, c));
173     } else if (diff == coord(0, 2)) { // up
          totalChance += upChance;
175       choiceChanceValues.push_back(std::make_tuple(upChance, c));
        } else {
177       totalChance += downChance;
          assert(diff == coord(0, -2)); // down
179       choiceChanceValues.push_back(std::make_tuple(downChance, c));
        }
181   }

183   float choice = randFloat(0, totalChance);
      auto it = choiceChanceValues.begin();
185   // stack the choices on top of each other, with each slice getting the
      // probability assigned to its direction
187   // end()-1 so that last choice would get any extra probability due to floating
      // point inaccuraccies
189   while (it != choiceChanceValues.end() - 1 && choice > std::get<0>(*it)) {
        choice -= std::get<0>(*it);
191     ++it;
      }
193
      return std::get<1>(*it);
195 }

197 void tremauxWalk(const maze<xSize, ySize> &grid,
                     std::vector<coord> &visitedTiles) {
199   genericTremauxWalk(grid, visitedTiles, pickUniform, true);
    }
```

```
201
void humanWalk(const maze<xSize, ySize> &grid,
203                std::vector<coord> &visitedTiles) {
  genericTremauxWalk(grid, visitedTiles, pickBiased, false);
205 }

207 void smartWalk(const maze<xSize, ySize> &grid,
                std::vector<coord> &visitedTiles) {
209   genericTremauxWalk(grid, visitedTiles, pickBiased, false);
}
```

code/playerSimulator.h

8.2 Excerpt from MazeAnalysis source code that finds tiles in a path that are cut off from the solution tile

```
typedef struct {
2   std::string ip;
    float progressIndex;
4   int areaSize;
    int enclosingPathLength;
6   int tilesBeforeTurnBack;
} CutOffInstance;
8

typedef std::vector<coord>::const_iterator PathIter;
10

std::vector<coord> getEnclosedArea(const maze<xSize, ySize> &grid,
12                                 const std::vector<coord> &path,
                                   PathIter pathPosition, const coord tipTile,
14                                 PathIter &oldestNeededPosition) {
    std::vector<coord> enclosedArea;
16  std::queue<coord> toProcess;
    toProcess.push(tipTile);
18  oldestNeededPosition = pathPosition;
    while (!toProcess.empty()) {
20    coord current = toProcess.front();
      toProcess.pop();
22    if (std::find(enclosedArea.begin(), enclosedArea.end(), current) !=
          enclosedArea.end()) {
24      continue; // already seen this one
      }
26    if (grid.outofbounds(current)) {
```

```
      continue;
    }
    auto foundInPath =
        std::find(std::vector<coord>::const_reverse_iterator(pathPosition + 1),
                  path.rend(), current);
    if (foundInPath != path.rend()) {
      if (std::distance<PathIter>(path.begin(), foundInPath.base()) <
          std::distance(path.begin(), oldestNeededPosition)) {
        oldestNeededPosition = foundInPath.base();
      }
      continue; // found this one in the path leading to *it
    }
    // still a valid member of the enclosed area
    enclosedArea.push_back(current);
    // add all neighbours for evaluation
    grid.forNeighbours(
        current, [&toProcess](coord neighbour) { toProcess.push(neighbour); });
  }
  return enclosedArea;
}

void analyseClosedOffAreas(const maze<xSize, ySize> &grid,
                           const std::vector<coord> &path,
                           const std::string &ip) {
    // keep track of seen closed off areas in the path so we don't display them
    // twice
    std::vector<std::tuple<PathIter, PathIter>> wastedWalks;

    for (PathIter it = path.begin() + 2; it != path.end(); ++it) {
      const coord c = *it;
      if (!isCell(c)) {
        // not a choice cell
        continue;
      }
      int currentDistance = std::distance<PathIter>(path.begin(), it);
      auto encounteredBefore = std::find_if(
          wastedWalks.begin(), wastedWalks.end(),
          [currentDistance, &path](
              std::tuple<PathIter, PathIter> previousWalk) {
            return currentDistance >
                       std::distance<PathIter>(path.begin(),
                                               std::get<0>(previousWalk)) &&
                   currentDistance <
                       std::distance<PathIter>(path.begin(),
```

```
                                                std::get<1>(previousWalk));
72          });
        if (encounteredBefore != wastedWalks.end()) {
74        // it is contained in a previously encountered closed off area walk,
          // skip to the end of that walk
76        // it = std::get<1>((*encounteredBefore)-1); // premature optimization;
          // nevermind
78        continue;
        }
80
        auto options = grid.getWalkableNeighbours(c);
82      for (coord next : options) {
          // for testing, you can use this instead to show up all closed off
84        // areas, even when there's no option to walk that direction
          // const coord iter_dir[] = { coord(0,-2), coord(0,2), coord(-2,0),
86        // coord(2,0) };
          // for(int i = 0; i < 4; ++i){
88        //  coord next = c + iter_dir[i];
          //  if( grid.outofbounds(next) ){
90        //    continue;
          //  }
92        if (std::find(std::vector<coord>::const_reverse_iterator(it),
                      path.rend(), next) != path.rend()) {
94        continue; // this "choice" is on the path up to here; skip it
          }
96        // this tile is a possible next choice; see if it is in enclosed space
          PathIter oldestNeededPosition;
98        auto area = getEnclosedArea(grid, path, it, next, oldestNeededPosition);
          if (std::find(area.begin(), area.end(), grid.solutionPosition()) ==
100           area.end()) {
            // choosing this direction cannot lead to the exit given the knowledge
102         // in path up to it
            int d = std::distance(oldestNeededPosition, it);
104
            auto boundaryToCutOffArea = coord::avg(c, next);
106         auto walkedThere = std::find(it, path.end(), boundaryToCutOffArea);
            if (walkedThere == path.end()) { // never walked in
108           cutOffInstances.push_back(
                  CutOffInstance{ip, progressIndex(c), (int)area.size(), d, 0});
110         } else { // walked in at some point
              // find where we come back; the first *cell* that is not in the
112           // enclosed area
              PathIter returnBoundary = std::find_if(
114               walkedThere, path.end(), [&area](const coord tileInWastedPath) {
```

```
                    if (!isCell(tileInWastedPath)) {
116                   return false;
                    }
118                 return std::find(area.begin(), area.end(),
                                     tileInWastedPath) == area.end();
120               });
              if (returnBoundary == path.end()) {
122             // nevermind this option; we cannot determine when the player
                // walked out again
124             // (path must have been cut off)
                assert(*(path.rbegin()) == grid.solutionPosition());
126             continue;
              }
128           // log the tuple so that each area is not mentioned multiple times
              wastedWalks.push_back(std::make_tuple(it, returnBoundary));
130
              int distance = std::distance(it, returnBoundary);
132           cutOffInstances.push_back(CutOffInstance{
                  ip, progressIndex(c), (int)area.size(), d, distance});
134         }
          }
136     }
    }
138 }
```

Bibliography

[che] Cheerp: C++ compiler for the web.

[DR] Dale Dougherty and Arnold Robbins. *sed & awk - UNIX power tools*. O'Reilly.

[Kruoo] Steve Krug. *Don't Make Me Think: A Common Sense Approach to Web Usability*. New Riders Press, 2000.

[PT11] Jean Pelletier-Thibert. Charles trémaux (1859 — 1882) ecole polytechnique of paris (x:1876), french engineer of the telegraph. In *Annals academic*, march 2011.

[Ram12] Robert Ramirez. Applications of random mazes and graphs generated via markov chain monte carlo methods. 2012.