

"ALEXANDRU-IOAN CUZA" UNIVERSITY OF IAȘI  
**FACULTY OF COMPUTER SCIENCE IAȘI**



MASTER THESIS

**An overview of mazes: creation, solving and  
applicability**

propusă de

**Adrian Petercă**

**Sesiunea: iulie, 2024**

Coordonator științific

**Conf. Dr. Croitoru Eugen**

**"ALEXANDRU-IOAN CUZA" UNIVERSITY OF IAȘI**  
**FACULTY OF COMPUTER SCIENCE IAȘI**

# **An overview of mazes: creation, solving and applicability**

**Adrian Petercă**

**Sesiunea: iulie, 2024**

**Coordonator științific**

**Conf. Dr. Croitoru Eugen**

# Table of contents

<b>Purpose</b>	<b>2</b>
<b>Introduction</b>	<b>3</b>
<b>1 State of the Art</b>	<b>7</b>
1.1 Mazes . . . . .	7
1.2 Game Theory . . . . .	9
1.3 (Iterated) Prisoner's Dilemma . . . . .	10
<b>2 Generation</b>	<b>14</b>
2.1 Implementation details . . . . .	14
2.1.1 Depth first search . . . . .	16
2.1.2 Binary Tree . . . . .	16
2.1.3 Sidewinder . . . . .	17
2.1.4 Hunt and Kill . . . . .	17
2.1.5 Random Kruskal . . . . .	18
2.1.6 Aldous Broder . . . . .	19
2.1.7 Random Carving . . . . .	19
2.1.8 Spiral . . . . .	20
2.1.9 Random Prim . . . . .	20
2.1.10 Recursive Division . . . . .	20
2.1.11 Cellular Automation . . . . .	21
2.1.12 Genetic Algorithm . . . . .	21
2.2 Image trained GAN . . . . .	22
2.3 Numerical matrix trained GAN . . . . .	24
2.4 Future work . . . . .	25
2.4.1 Chunk based generation for infinite mazes . . . . .	25

2.4.2	Changing mazes . . . . .	25
2.4.3	Weaved mazes . . . . .	26
2.4.4	Wave function collapse . . . . .	26
2.4.5	Bitmap mazes . . . . .	26
<b>3</b>	<b>Solving</b>	<b>27</b>
3.1	Literature . . . . .	27
3.2	Implementation . . . . .	29
<b>4</b>	<b>Applicability</b>	<b>31</b>
4.1	Real life examples . . . . .	31
4.2	amazed - Python3 library . . . . .	33
4.3	CoLab - a maze based game . . . . .	35
4.3.1	CopyPlayer . . . . .	44
4.3.2	RememberMe . . . . .	44
4.3.3	Evolved strategies . . . . .	44
4.3.4	Future work & directions . . . . .	47
	<b>Conclusions</b>	<b>48</b>
	<b>Bibliography</b>	<b>58</b>

# Purpose

Mazes are seen as simple 2D problems that can be solved almost instantly using well researched and documented algorithms in a finite and easily calculable amount of steps. One could argue that mazes pose significance only in challenging the human mind and that big mazes (the ones that usually don't fit on paper) or infinite ones are of no real use. However, this paper tries to prove the contrary. It tackles three main subjects related to mazes: the creation part (also called *generation*), how to solve what has been created and where can these results be of interest.

The generation of a maze is usually done by *deterministic* algorithms such as **Depth First Search** or **Kruskal**. This paper aims to provide a new approach based on **Generative Adversarial Networks**, that tackle the problem first by using images as training data, then by using wall-encoded matrices. The second method will prove to be better.

The aspect of solving is considered in regards to the question "*what algorithm is good for what type of maze?*", taking into account aspects such as the number of steps, the solution's type, what restrictions are present (such as local discovery versus global discovery), whilst also analyzing known algorithms.

Regarding the applicability, this paper aims to provide two pieces of software for the general public. The first is called `amazed` and it is a PyPI library[24], which aims to provide a centralised utility for generating and solving mazes. Due to its architecture, the tool can be further expanded easily, if needed. The second piece of software is called `CoLab`[4], a game focusing on two agents that need to traverse an unknown maze using a predefined strategy. The game features a negotiation concept, where each agent can trade known information for new information. A good strategy trained using Genetic Algorithms is also presented.

# Introduction

In general, a maze can be defined as a finite space with at least 2 dimensions, where multiple individual cells are connected in an uniform manner. Usually, mazes are represented as 2D matrices and can also be seen as graphs, but they are only a particular subset of them, with specific restrictions set beforehand. The main challenge that a maze provides is finding a(n) (*optimal*) path from a starting point to a finish point, with respect to a distance metric, usually set as a total count of unique cells traversed.

The difficulty of a maze can be controlled by its restrictions. Most of the time, these consist of rules such as limiting the number of possible directions to 4 (*up, down, left and right*), being able to only evaluate (or otherwise "see") cells that are 1 move away, setting a certain "speed" for "moving" between cells, etc. These restrictions add a new parameter to the challenge, making use of mazes that could otherwise be seen as "too simple" (as seen in Fig.1).

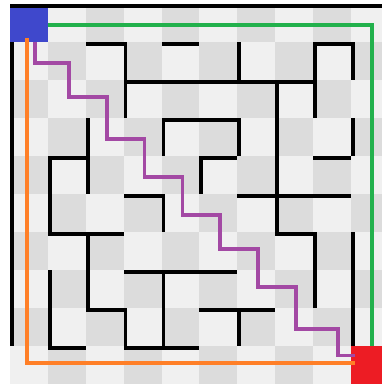


Figure 1: A simple maze. The purple path has the same number of steps as either the green or orange path, but can be faster by adding a new rule: *each cell has 8 possible directions*. Also, the other two paths can be faster by using the following rule: *gain momentum when maintaining the original direction of movement*.

This paper will focus on three main aspects: the generation process, for which a new method involving GANs is tested against the standard, deterministic algorithms,

the solvability of a maze and what makes it hard for which algorithm, along with a handful of example rules for adjusting the difficulty, and finally a part about applicability, where two pieces of software will be presented: a Python library which was created and used for this paper and a game inspired by The Iterated Prisoner's Dilemma where two agents must compete and cooperate at the same time.

The first part analyses twelve algorithms, four of which are personal implementation, comparing each one by underlying their capabilities and limitations and providing distribution tables in Chapter 4.3.4. After this, the paper analyses how GANs handle this type of information, in two different contexts. First, by training a network using a dataset of 10000 images of 64x64 mazes created by one of the algorithms above and analysing the results (one such maze can be seen in Fig. 4.9). The result of this analysis lead to the conclusion that image-based approaches tend to skip over important details and do not guarantee a uniform maze, usually resulting in bits and pieces of an actual structure. However, one could argue that this approach may work if the image sizes are equal to the maze sizes (meaning that, for a 64x64 maze, a black and white image of 64x64 pixels is used), thus leading to the second approach. This one focuses on retaining the relations between multiple cells in the maze (otherwise it would result in a randomly created grid) and trying to map collections instead of individual cells (for example, if a network can be trained to recreate a "T" intersection, that is better than randomly carving a maze and hoping for a coherent structure). The process works by assigning a value between 0 and 15 to each cell corresponding to the presence of walls (for a visual example, see Fig. 2), then using *convolutional layers* to extract information. This approach is guaranteed to create a correct maze (as opposed to the image-trained network) but tends to be very sparse (see Fig. 4.10).

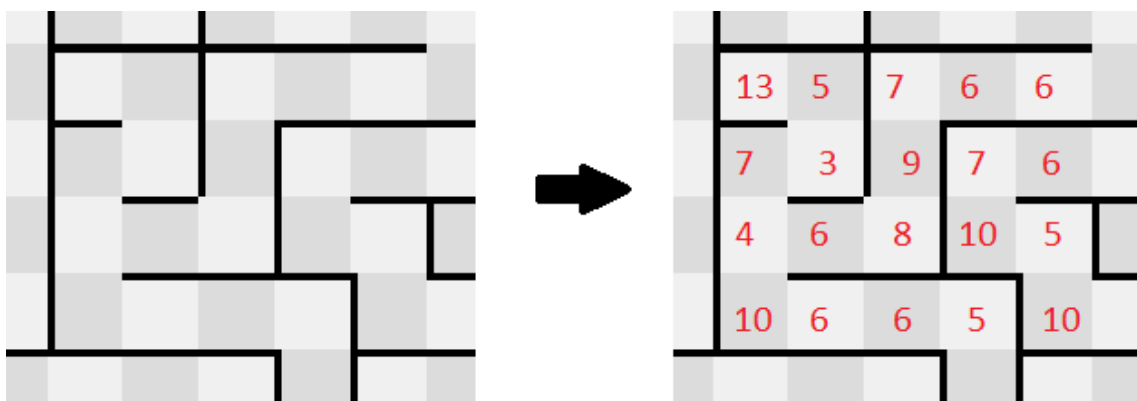


Figure 2: How each cell is mapped for a numerical-matrix based GAN

The second part of this paper looks at the *solvability* of a maze, focusing on identifying key characteristics for each type of maze. This is done in the idea that some algorithms can be faster if used on the right maze types (for example, there is no faster algorithm for single solution mazes than Heuristic DFS - a combination between a DFS approach that select the next "closest" cell instead of a random one). Another aspect that is taken into consideration is what exactly makes a maze "hard"? Is it the number of steps for the final solution? Is it the execution time? Is it the number of possible paths from start to finish? Finally, the paper attempts to provide a new way of determining a solution route by merging together open areas in sparse mazes and shortening long corridors.

The applicability is denoted by providing an open-source library that can manage all aspects of mazes and a two-player, maze routing game. The library aims to provide the end user with a friendly class hierarchy that can be expanded if needed, allowing custom cell constructors, various maze exporting methods, GIF and PNG/JPG formats for graphical uses (both for the generation part and for the solving part). The game is inspired by The Iterated Prisoner's Dilemma, attempting to provide a new environment where cooperation is key. Two agents compete to find the finish in an partially known maze by exchanging information (if the trade is *fair* - what *fair* means depends on the strategy at hand). There are several game modes which will be described in Chapter4, but this paper only uses the *almost-zero-sum-game* policy, which aims for a fair game for both agents. The policy selects at random two cells in the maze,  $cell_X$  and  $cell_Y$ , setting

$$playerA_{start} = cell_X, playerA_{finish} = cell_Y$$

$$playerB_{start} = cell_Y, playerB_{finish} = cell_X$$

ensuring that both players need to traverse the same path (thus removing situations where one player would win just because of a good spawn). One way to develop good strategies is by evolving them using evolutionary algorithms, such as genetic algorithms. This approach is tackled successfully, creating a strategy that is better than most human players.



# Definitions

For a better understanding, the following list defines a list of terms that will be used frequently in this paper:

- *fully connected maze* : a maze in which every pair of cells has at least one path between them.
- *blocked maze* : a maze in which, at least a pair of cells does not have a valid path between them.
- *warped maze* : a maze in which the margins are connected and where the player can go from  $cell_{0,columns-1}$  to  $cell_{0,0}$  if it chooses to go *East* (or right) and there is no wall present between the two cells.
- *single solution* : a maze in which, for each pair of (*START*, *FINISH*) cells, exists only one path.
- *multiple solution* : a maze in which, for each pair of (*START*, *FINISH*) cells, exists at least two paths that have at least one cell different. A maze with this property can also be called *cyclic* or *looped*.
- *dense maze* : a maze in which the ratio between the number of cells and the number of walls has a low value, generally smaller than 0.25. These types of mazes tend to have a small number of solutions.
- *sparse maze* : a maze in which the ratio between the number of cells and the number of walls has a value close to 1. These types of mazes tend to have a lot of solutions.
- *corridor* : multiple connected cells in the same orientation (either *North-South* or *East-West*) with no branching.
- *corridor-like path* : similar to *corridor*, but it does not take into account the orientation, just the idea of a long sequence of connected cells with no branching.

# Chapter 1

## State of the Art

### 1.1 Mazes

Mazes are constrained undirected graphs, and their generation is mostly inspired by graph traversal algorithms, with some novel implementations focusing on speed or memory efficiency. Among researchers, there is not a definitive accepted history of who invented or otherwise popularized a specific technique, with exception being algorithms such as Prim's 2.1.9, Eller's, Wilson's, Kruskal's 2.1.5, which were adapted to the task at hand - for example, the Kruskal Minimum Spanning Tree algorithm can be modified such that walls represent edges and non-connected areas represent disjoint sets. Researches also tried approaches that involve cellular automata [34], quantum annealing [35], mutated evolution [36] to improve upon this subject, many with great success. Nonetheless, the general consensus is mainly focused in the novelty of the generated maze, with aspects like patterns, fractal areas, balanced connectivity and others being of much more interest rather than generation speed or other such metrics. If technicalities are of interest, one could consult *Analysis of Maze Generating Algorithms* [41] by Peter Gabrovsek, where he offers a small, but detailed overview on six maze generation algorithms, along with four solving agent metrics, concluding that spanning tree based generation yields the hardest mazes.

The following part will describe all distinct known maze generation algorithms as of writing this paper. The first eight approaches are also discussed in more detail in Chapter2, with each method having its own subsection.

- **Depth First Search 2.1.1:** using a stack and a controllable bias, perform a full walk on the maze, carving paths without walking over already visited cells. If

the bias is ignored, it tends to create mazes with deeper dead ends. It is also known as *Recursive Backtracker*.

- **Binary Tree** 2.1.2: for each cell, randomly carve either in *North* or *West*. Can also be made with a combination of *North*, *South* and *East*, *West*. However, it will create an empty corridor along the selected borders.
- **Sidewinder** 2.1.3: for each cell, randomly choose to carve *East* or not. If the choice is negative, select a random cell from set (containing all subsequent cells from the most recent negative choice) and carve *North*. It can also be reversed, carving from *South*, much like *Binary Tree*.
- **Hunt and Kill** 2.1.4: perform a random walk (ignoring visited cells) starting from a random cell, then start searching for another start cell from (0, 0) until all cells are visited.
- **Random Kruskal** 2.1.5: each cell is considered a "group". "Bind" each group by breaking the wall between two random, adjacent cells (one from each group). Repeat until all cells belong to one group.
- **Aldous Broder** 2.1.6: starts performing random walks until all cells are visited and only carves a path between two cells if the second is unvisited.
- **Random Prim** 2.1.9: it relies on a queue of *frontier* cells (that have both visited and unvisited neighbors), selecting one at random and carving a path to an unvisited cell. Usually, the generation starts from the center of the maze.
- **Recursive Division** 2.1.10: starting with a maze with no walls, split the current area with a centered wall with only one opening, recursively doing the same with the newly created subareas.
- **Random Eller**: it works by creating each row at a time with the help of sets. For the first row, each cell will be its own set. Then, random merges are performed between adjacent cells from distinct sets. After this, for each set, a path to *South* will be carved. Finally, assign each cell from the next row that is not already in a set its own set and make the next row the current one. For the last row, join all adjacent cells that belong in distinct sets and do not carve vertical paths.

- **Wilson:** start by adding a random cell to the list of carved cells, then select a random cell not already carved and perform a random walk until an already carved cell is found (adding all cells walked into the list of carved cells), repeating this step until all cells are carved. It is an improved version of *Aldous Broder*, converging faster and faster as more cells are carved.
- **Growing tree:** start by adding a random cell to a list of visited cells. Then, using a *selection function*, select a cell from the list and carve a path to random, unvisited neighbor cell, adding it to the list of visited cells. Repeat this step until all cells are visited. If *selection function* is set to *select the newest added cell*, it behaves exactly like *Depth first search*, while random selection leads to *Random Prim*.

Nonetheless, mazes can also be generated by rotating, mirroring, or otherwise cutting an already existing maze, but such methods rely on existing objects and are not considered generation algorithms. The same applies for combined techniques, such as one of the methods described above paired with a random mutations on each cell - this also relies on existing objects, and as such is not considered an *atomic* algorithm.

## 1.2 Game Theory

Game theory is a mathematical framework used for analyzing situations where multiple players make decisions that influence each other directly. It was inspired from the field of economics, presenting a way of mathematically studying what would be later called *zero sum games*, situations where a participant's gain means the opponent's losses and vice-versa, and later extending to *non-zero sum games* and behavioural science, focusing on studying interactions between agents in a competitive environment (be it natural or numerical).

At its core, game theory involves the study of games, which are defined by a set of players, a set of strategies available to each player, and a payoff function that determines the outcome based on the chosen strategies for each player. One of the fundamental concepts in game theory is the Nash Equilibrium [37], which occurs when players select strategies such that no player can benefit by unilaterally changing their strategy, given the strategies of the others. This concept guarantees the existence of a global stability point, where by evaluating each player's chosen strategy against all others, one can conclude that no improvement can be made for any player.

Applied to computer games, *game theory* can be split into cooperative and non-cooperative types. The first one is found in settings where parties can bind together to create a better strategy for a given obstacle, such as in *World of Warcraft* where players compete against each other but need to team up to participate in *raids* (hard challenges involving dungeons, bosses and quests), even if the prizes will be split. The second type is found in games such as *Fall Guys*, where players must race each other in an obstacle course, first to finish winning the tournament.

Another view on computer games is represented by symmetric and asymmetric games. The first one can be defined as the set of games in which each player has the same amount of knowledge as any other player (such as in chess, where each player can see the next possible moves of the opponent), while the second one describes games where a player only knows information related to it and predicts what actions the other players will do, without knowing for certain in advance (for example, a game of poker, where the cards for the other parties are hidden).

This paper will tackle the idea of a cooperative, asymmetric setting, in which two agents are spawned in an unknown maze and need to race each other to the finish, either by information exchange or by brute forcing a possible solution. The game is called **CoLab** and will be discussed in detail in Chapter 4.

## 1.3 (Iterated) Prisoner's Dilemma

**Prisoner's Dilemma** [25] is a widely known problem in behaviour science and game theory, first proposed in 1950, where two *rational agents* strive to gain as much points as possible in a competitive environment, each one being able to either cooperate for mutual gains or defect for personal gain. However, the actions of each agent are dependent on one another: if they both cooperate, both will increase their score by a normal amount, yet if one of them defects while the other cooperates, the defector will gain a significant advantage, while if both of them defect, they will get a very low score. The original score matrix is displayed in Figure 1.1.

**Prisoner's Dilemma**

		<b>Player A</b>	
		Cooperate	Defect
<b>Player B</b>	Cooperate	3 3	0 5
	Defect	5 0	1 1

Figure 1.1: Original IPD Matrix score

For a single iteration, without prior knowledge of the opponent's strategy, the best move for any agent would be to defect. This can be demonstrated by calculating the expected score values:

	Player B cooperates	Player B defects	Total score
Player A cooperates	3	0	3
Player A defects	5	1	6

Yet, for **iterated** scenarios, the strategies change significantly. Robert Axelrod [29] organized a tournament (detailed in his 1984 book *The Evolution of Cooperation*) in which he analyzed multiples strategies paired against one another, thus creating a set of four *attributes* that a good (yet not perfect) strategy must have:

- **nice** : the strategy will not defect before its opponent does. This encourages cooperation in the long term.
- **retaliating** : the strategy must sometimes retaliate, otherwise they will be taken advantage of. This punishes greedy strategies, such as *Always Deflect*.
- **forgiving** : successful strategies must be forgiving. This encourages cooperation re-balancing after a series of defects which would otherwise lead to a spiral of defections.
- **non-envious** : the strategy must not strive to score more than the opponent, which would lead to defections and loss of trust.

In the first tournament, 14 strategies [40] were submitted by various professors

and paired with each other. Some of those strategies considered atomic (not a combination of two or more strategies) are described below:

- **Tit-for-Tat** : one of the simplest, yet powerful strategies. It starts of by cooperating and deflects only if the opponent deflected in the round prior.
- **Tit-for-Two-Tats** : also called *Tit-for-Tat with forgiveness*, it applies the same strategy, but deflects only if the opponent deflected two rounds in a row.
- **Always Cooperate** : as the name implies, it will always cooperate.
- **Always Deflect** : as the name implies, it will always deflect.
- **Random** : it will randomly cooperate or deflect.
- **Grudger** : it starts of cooperating, but will switch to deflect (and stay that way) if the opponent deflects once.
- **Win-Stay-Lose-Shift** : starts by cooperating and will cooperate only if the last round was successful, otherwise deflects.
- **Tester** : starts of with a deflection, then adjusts itself based on the opponent's actions.
- **Majority** : starts of with a cooperation, then choose the next action as the maximum between the total number of cooperations and the total number of deflections made by the opponent.

Other strategies are more complex, such as the one proposed by Gordon Tullock, where the agent cooperates for the first 10 moves, that it cooperates 10% less that the opponent in the last ten moves. The tournament was ran 5 times each with 200 episodes, to ensure a stable average score. Despite many strategies taking into account multiple scenarios and measures, the best one was *Tit-for-Tat*, having three out of four essential traits, the missing one being *forgiveness*. Later studies [26] [27] [28] will further support this strategy and try to better understand it, but still none will find a general best strategy [38].

The second tournament, in which a total of 62 strategies were submitted and analyzed, the winner was still *Tit-for-Tat*. The main motive that explain this behaviour is that, in a population where individuals try their best to cooperate, this strategy will

flourish, but it will retaliate if necessary. Tools such as *The Evolution of Trust* [39] highlight this aspect by using evolving populations, showcasing how *Tit-for-tat* or similar strategies can outcompete greedy strategies focusing on cooperation not only on single episodes, but on cooperation as a group. In other words, if a small part of the population is focused on cooperation, they tend to outperform other strategies focused solely on individual gain.



# Chapter 2

## Generation

### 2.1 Implementation details

As stated in the Introduction, a maze is defined as a matrix of cells with  $no\_rows > 1$  and  $no\_columns > 1$ , with each cell having the possibility of a wall in four directions: *North*, *East*, *South* and *West* (or up, right, down and left). Another common approach to this problem is to define a maze as a matrix of cells, where each cell can be either a **passage cell** or a **wall cell**, and while it is true that this method simplifies some aspects (such as the double wall problem - in a typical implementation, adjacent cells must synchronize in wall states: if  $cell_{X,Y}$  has a wall to the *East*,  $cell_{X,Y+1}$  must also have a wall to the *West*), it also brings unreachable cells (if a cell has four **wall cells**, that space is lost information). Earlier implementation used this approach and found (in a subjective manner) that the graphical aspect was better perceived when walls were displayed separately of cells (which is a part of the final goal for this paper).

As data structures, mazes are usually represented as unoriented graphs, with each node being a cell and each wall being an edge, but this is not the only method. Another way would be to represent them as matrices, where each cell has a value between 0 and 15, denoting a binary representation of 4 bits, representing the presence of a wall in each direction. Although both methods are easier to process when dealing with algorithms, they store the same information twice (at least for some cells): each  $cell_{X,Y}$  stores the information for the wall with  $cell_{X+1,Y}$  and vice-versa. To counter this, one could store all the information in a binary string, as shown in Fig.2.1, thus leading to a significant reduction in memory size:

$$total\_bits_1 = 4 \cdot rows \cdot columns$$

$$total\_bits_2 = rows \cdot (columns - 1) + columns \cdot (rows - 1) + (2 \cdot rows + 2 \cdot columns)$$



Figure 2.1: Binary string maze encoding (for unwrapped mazes)

Worth mentioning is that, for the second method, if the maze is not wrapped, meaning that there will always be a wall surrounding the margin of the maze, then the total value comes down to:

$$total\_bits_2 = rows \cdot (columns - 1) + columns \cdot (rows - 1)$$

Each algorithm used for generation has a particular set of characteristics and biases. Some tend to create long corridors, while others are prone to branching (creating multiple splits from a "main" path), resulting in various patterns (see Fig. 4.11). However, what makes an algorithm "good" depends on the purpose of the maze: if it is intended for human challenges, it should rely on branching and twists to make it hard to follow, but if it is intended for computers that use heuristics to approximate a path, it should create "false paths" (routes that, at some point, need to distance themselves from the finish point in order to go on the right track and not end up in a dead-end). Some good metrics for scoring a maze are:

- $M_1 = \frac{len(solver(maze).steps)}{rows \cdot columns}$  : this metric gives a score based on how many steps a *solver* algorithm needs to perform in order to find a solution.
- $M_2 = \frac{max(steps(all\_deadends))}{rows \cdot columns}$  : this metric improves upon the first, however taking into account the maximum possible distance between any two cells. Only dead ends are calculated to improve performance and reduce unnecessary calculations.
- $M_3 = \frac{internal\_walls\_count}{rows \cdot columns}$  : this metric measures the "sparseness" of a maze. Denser mazes tend to be more difficult.
- $M_4 = \frac{T\_intersections\_count}{rows \cdot columns}$  : this metric follows the idea that multiple T intersections mean a vast branching of the maze, thus multiple paths to explore. A similar metric taking into account all types of cells with various modifiers can also be used in the general case.

- $M_5 = \frac{1}{\text{len}(\text{distinct\_areas})}$  : this metric favours mazes that have less disconnected areas. A complete maze would have only 1 connected area, so the highest score possible is 1.

The following subsections describe in detail each algorithm used, its set of characteristics and how it is implemented, along with examples and a cell type distribution table calculated for 100 10x10 mazes in Chapter 4.3.4.

### 2.1.1 Depth first search

The DFS algorithm is one of the first methods for maze generation, if not the first if random carving is put aside. It's a simple approach, creating single solution mazes with usually long corridor-like routes. The pseudocode is provided below.

```
start = random position in the maze
visited = empty list
stack.push(start)
while not all cells are visited:
    last_cell = current_cell
    current_cell = stack.pop()
    maze.break_wall(last_cell, current_cell)
    visited.add(current_cell)
    next_cells = all unvisited cells from current_cell
    # The order of cells dictates the bias of the output maze
    stack.push(next_cells)
```

### 2.1.2 Binary Tree

The binary tree algorithm is another simple approach that randomly carves a wall either on the North-South axis or on the East-West axis. Although very unlikely, in theory this approach can create blocked mazes. One downside of this approach is the presence of a empty border on selected carving directions. The pseudocode is provided below.

```
for each row:
    for each column:
        cell = maze[row][column]
```

```

if random() > 0.5:
    if North is a valid direction: carve path to North
    else: carve path to West
else:
    if West is a valid direction: carve path to West
    else: carve path to North

```

### 2.1.3 Sidewinder

Much alike to Binary Tree, the Sidewinder algorithm evaluates each cell and carves a passage either North or East, guaranteeing a complete maze. The difference between the two is that Sidewinder only has the North edge as a corridor from  $cell_{0,0}$  to  $cell_{0,columns-1}$ . The pseudocode is provided below.

```

run = empty list
for each cell in the top row:
    maze.path(cell, EAST)
for each cell in the maze:
    if cell in top row:
        skip
    run.add(cell)
    if random() > 0.5 and (cell + EAST) is valid position:
        maze.path(cell, EAST)
    else:
        random_cell = random_element(run)
        maze.path(random_cell, NORTH)
    run.clear()

```

### 2.1.4 Hunt and Kill

This method works like DFS, in the sense that it carves random paths from a start point, but it does not use a stack to select a start point. Instead, as the name implies, it "hunts" an unvisited cell starting from (0, 0) and selects it as the start point for a new walk, carving a wall between the newly found cell and a neighbour one that is already visited. The last steps ensures that the maze remains connected.

```

C = (0, 0)
while not all cells are visited:
    mark cell C as visited
    dirs = all possible moves from C
    if len(dirs) is 0:
        C' = grid_search()
        carve a path from C' to neighbors
    else:
        D = random select from dirs
        carve a path from C in direction D
        C = C + D # Moving from C in direction D

```

### 2.1.5 Random Kruskal

The random Kruskal algorithm is based of the Kruskal graph traversal method, which works by selecting sets of edges and merging them together to create a spanning tree. The algorithm for mazes follows the same idea, initializing all cells as a set with only one object, then randomly selecting two distinct cells  $X$  and  $Y$  from separate sets and carving a path between them, then merging the two sets into one. This operation is repeated until all cells are in the same set. The pseudocode is provided below.

```

list_of_sets = set(cell) foreach cell in maze
list_of_edges = maze.walls()
foreach edge in list_of_edges:
    C1, C2 = edge
    set_1 = list_of_sets.find(C1)
    set_2 = list_of_sets.find(C2)
    if set_1 != set_2:
        maze.path(C1, C2)
        list_of_sets.remove(set_1)
        list_of_sets.remove(set_2)
        new_set = set_1 + set_2
        list_of_sets.add(new_set)

```

### 2.1.6 Aldous Broder

This algorithm has a very simple implementation and has its place as a stepping stone to a better understanding of other algorithms, but in practice is the slowest choice, because it does not take into account that a cell has been previously visited when moving (only when carving paths). Thus, it often searches endlessly for an unexplored cell without an indicator of where such a cell may be. Mazes created by this algorithm tend to imitate Depth First Search. The pseudocode is provided below.

```
visited = empty list
cell = random(maze.cells)
while not all cells are visited:
    visited.add(cell)

    neighbor = random(cell.neighbors())
    if neighbor is not visited:
        maze.path(cell, neighbor)
    cell = neighbor
```

### 2.1.7 Random Carving

The aim of this algorithm is to provide a better way of a pseudo-random carving. Instead of each cell having the same probability, the algorithm relies on the following ideas:

- increase the chance that a cell/wall will be carved for each previous cell/wall that was **not** carved, thus reducing the number of blocked cells / inaccessible areas
- evaluate each wall from a cell, not the cell itself. This allows corridor-like paths to appear more often
- provide an **adaptive function** that will increase the chance.

With these rules, the algorithm build sparse, but almost complete mazes (which is a significant improvement over a random iteration over the set of cells/walls).

### 2.1.8 Spiral

As the name implies, this algorithm creates spiral mazes, using the same technique as Hunt and Kill. It searches for an unvisited cell, then starts going in a spiral motion, with a bias towards the "center" and a predefined maximum length. Not very useful on its own as it rarely generates complete mazes, but can be paired with the Random Carving for a multi-solution maze or a modified version of Random Kruskal that ensures only a single possible solution.

### 2.1.9 Random Prim

Similar to Kruskal's, Prim's algorithm works by joining together sets of cells. First, it selects a starting cell (usually the center of the maze) and builds a so called *frontier* of cells, where each cell has at least one unvisited neighbor. This leads to a maze with more short length dead ends. The pseudocode is provided below.

```
visited = empty set
frontier = empty set
visited.add(start_cell)
while not all cells are visited:
    for each visited cell:
        for each neighbor for cell:
            if neighbor is unvisited:
                frontier.add((cell, neighbor))
    shuffle(frontier)
    for cell1, cell2 in frontier:
        if cell2 is unvisited:
            break
    maze.path(cell1, cell2)
    visited.add(cell2)
```

### 2.1.10 Recursive Division

This algorithm is very different than the others, because it builds a maze backwards: it starts with a blank canvas and adds walls in a recursive manner. First, it divides the area into two equal subareas along the longer axis, then adds a wall between

subareas. After that, it randomly selects a place in the wall to break, such that the areas remain connected. Then it recursively does the same operation on each subarea, until the maze is fully created. All mazes created in this manner are single solution. The pseudocode is provided below.

```
def divide(area):
    if area is only one cell:
        return
    if area.cols > area.rows:
        divide along the column axis
        randomly break the wall in one place
        divide(area.left)
        divide(area.right)
    else:
        divide along the row axis
        randomly break the wall in one place
        divide(area.up)
        divide(area.down)
```

### 2.1.11 Cellular Automation

This algorithm evolves a starting maze based on the 110 Rule [1]. It uses the binary string encoding of a maze and applies the following rules to evolve a single cell:

Current neighbors	000	001	010	011	100	101	110	111
New value for cell	0	1	1	1	0	1	1	0

Table 2.1: Rule 110

The implementation lets the user choose its own rules if necessary.

### 2.1.12 Genetic Algorithm

Earlier paragraphs describe a method for storing only the internal walls of a maze by using  $rows * (columns - 1) + column * (rows - 1)$  bits. This method opens up the possibility of generating mazes using Genetic Algorithms, by mixing and evolving the



bit string. The implementation uses various parameter configurations detailed in Table 2.2, with examples for each set of values in Figure 4.12.

Another approach that, at first glance, seems simpler, would be to use the initial maze encoding, where each cell is represented by 4 bits. However, this comes with an underlying problem that is unnecessary difficult to resolve and leads to misleading results: because most of the walls are double stored, one must always make sure that adjacent cells match (meaning that, if  $cell_{X,Y}$  has the value 0100 - only the East wall present, the cell  $cell_{X,Y+1}$  **must** have the value \*\*\*1 - the West wall present). Due to how the crossover operator and mutation operator work, this need to match walls cannot be guaranteed, thus leaving the only viable encoding as described above.

The GA was tested with the previous scoring metrics, either alone or in combination. The best results were obtained when a metric such as  $M_5$  (that rewards complete mazes) was combined with a distribution metric, such as generalised  $M_4$  (meaning that each type of cell had a specific score - for exact values check Table 4.2), paired with elitism. This led to natural looking, complete mazes, but without a guaranteed single or multiple solution structure.

## 2.2 Image trained GAN

A Generative Adversarial Network is a type of Artificial Neural Network that uses two models, a generator and a discriminator, which compete to "fool" and "understand" each other. First, a vector is sampled from the latent space, which is then fed into the generator. Its task is to create fake information that the discriminator cannot distinguish from real information (meaning that the final confidence should be split 50/50).

The idea for this maze generation technique was inspired by original papers in this domain, such as **DCGANs**[6] and **CycleGANs**[7], which showed how effective can General Adversarial Networks be at understanding a concept from the training dataset and recreating it. The plan was to train a model based on 64x64 black and white mazes to observe what kind of patterns will emerge. However, the results were far less interesting than expected: many models suffered from model collapse and would often create incorrectly structured mazes, ignoring borders and cell delimitation.

One major problem that this approach has is that maze images are too strict to train a model on. The image itself does not represent the information behind the maze

ID	Gen.	Pop. size	Crossover	Mutation	K value	Best score
1	1000	100	0.9	0.01	10%	1.376
2	500	70	0.9	0.01	10%	1.358
3	1000	100	0.9	0.01	5%	1.347
4	500	70	0.9	0.01	5%	1.346
5	500	70	0.6	0.01	10%	1.318
6	1000	100	0.6	0.01	10%	1.324
7	1000	100	0.6	0.01	5%	1.303
8	500	70	0.6	0.01	5%	1.290
9	500	70	0.9	0.1	5%	1.273
10	500	70	0.9	0.1	10%	1.266
11	1000	100	0.9	0.1	10%	1.265
12	1000	100	0.6	0.1	5%	1.256
13	1000	100	0.9	0.1	5%	1.248
14	1000	100	0.6	0.1	10%	1.243
15	500	70	0.6	0.1	10%	0.768
16	500	70	0.6	0.1	5%	0.731

Table 2.2: Results & parameters for metric  $M_4^{general} + M_5$

(the internal data structure does), being only a graphical representation for better human understanding. GANs work best on images where fine details can be ignored, rather focusing on the main "theme" or "center of interest". Thus, training a model to recognise cell delimitation using pixels is too strict to yield good results. However, to mitigate this problem, one could use images where each pixel represents a cell in the maze: white means an empty cell, black means a wall. But this method comes with another problem: a pixel wall surrounded by four pixel walls is a lost pixel, which should not happen in this particular implementation. A cell can be inaccessible, but not a wall.

This last method lead to the conclusion that the model should focus on the paths between cells instead of a graphical representation. This is how the next approach came to be.

## 2.3 Numerical matrix trained GAN

Numerical matrix Generative Adversarial Networks focus on the internal data structure of the maze instead of its graphical representation. As shown in Fig.2, each cell has a value between  $[0, 15]$ , based on four bits - one for each wall. Using convolutional layers, the model is able to understand the relationships between those values (what type of neighbors a cell has) and recreate them in a sparse, but structurally correct manner.

The generator and discriminator models used are represented in Fig. 4.13. The latent space is represented by a float vector with size 100, each value being between zero and one. To keep the overall structure small, a maximum number of 1000000 parameters for both models was imposed. The training was done in multiple stages with various configurations, most of which behaved poorly, either due to insufficient training, bad parameters or model collapse (where the generator greatly surpasses the discriminator, which in turn will result in poor "fake" results). To counter the last problem, experiments on model training frequency were done, with a good overall improvement. The best combination was between  $v2.4.1.G$  and  $v2.3.6.D$  trained for 250 epochs (all models are available on GitHub [4]), which is available in the release version of the `amazed` utility. Training datasets were generated using Hunt and Kill / Binary Tree / Random Kruskal, with 1000 and 10000 samples.

In conclusion, the method described above is not focused on speed or control

over the maze, but rather as a novel way of performing a task usually associated with well defined and studied algorithms. The size constrain, the model's complexity, the library requirements and the overall quality of generated mazes represent aspects that one should consider carefully when choosing this technique.

## **2.4 Future work**

Maze generation is a field of study that can always be improved, not only by new ways of computing a simple 2D maze, but by using specific constrains that make such structures work differently. In addition to the algorithms detailed above, this paper presents some ideas for future improvement.

### **2.4.1 Chunk based generation for infinite mazes**

Bigger mazes tend to be very slow to generate, even with simple algorithms, due to the amount of computing needed, while also remaining unexplored for the most part. One approach that may solve this problem is to segment such a maze into smaller areas and individually generate them as needed or to generate all of them in a thread-safe environment, thus dispersing the work load in an equal manner. This approach works great for places where procedural content generation is needed, such as games focused on environment exploration.

Another way would be to use "scrolling" mazes, where the agent only sees a fixed part of the maze and each row/column is generated when the agent reaches the known border. To encourage exploration and fast movement, the maze could "scroll" even if the agent is stationary.

### **2.4.2 Changing mazes**

Another novel way that a single maze can be simply improved is by modifying its structure while an agent is walking it. One such method can be to use *Cellular Automata* or *Genetic Algorithm* generation and updating the maze for every fixed number of iterations.

One other way to approach this challenge would be to use Rubik cube generation, where 6 mazes with equal sizes are generated and "bound" together, fusing their ends

(so no "border" exists). Then, by "rotating" rows and columns, the maze that the agent is currently on will change, but in a controllable manner.

### 2.4.3 Weaved mazes

Considered to be a middle ground between 2D and 3D mazes (they are not considered 3D mazes because by separating each "level" it does not yield two independent mazes), weaved mazes represent a challenge to the human mind but not to computer agents, due to their corridor-like nature. Because of their generation constraints, most weaved mazes do not have many intersections (which represent the most important point in an agent's algorithm) and rely solely on their crooked structure.

### 2.4.4 Wave function collapse

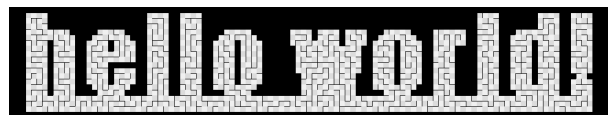
The wave function collapse[3] method improves upon the idea of generating a similar maze based on some input image or information.

### 2.4.5 Bitmap mazes

A bitmap maze is a novel way of creating mazes based on a predefined structure: instead of using a simple rectangle as the canvas, a bit mask is constructed based on the input provided (either as a bitmap or an ASCII map) and the maze will have the desired shape, like in Fig. 2.2.



(a) Original bitmap



(b) Generated maze

Figure 2.2: Bitmap generation

# Chapter 3

## Solving

This chapter covers the study, understanding and applicability in the current context of related work regarding the maze solving problem, along with how these techniques were integrated into the `amazed` library, followed by a novel compression method, finishing with future updates and directions on the matter at hand.

### 3.1 Literature

The problem of maze solving is often reduced to a graph traversal problem, in part due to the overall understanding and studying of graphs, and secondly due to how they are stored as data structures. However, one could also approach the problem from a different angle, such as with mazes stored as images that need a solution - in situations like this, it is difficult to say what algorithm is the best, because the focus is not only on a correct solution, but also on the preprocessing step required to transform the maze into a data structure that can support solving algorithms.

The problem can be resolved by either simple (otherwise known as "uninformed") or complex (otherwise known as "informed") algorithms. The first category covers two basic approaches: Depth First Search and Breath First Search. **DFS** relies on a stack where it adds unvisited cells in either a random or predefined manner, while **BFS** queues unvisited cells in a similar fashion. Both algorithms perform poorly on very large mazes, mainly due to the lack of *targeting*, but tend to be a good choice on single solution mazes, where their simplicity (which offers a minimal computational cost for each move) can be of use. One other uninformed technique would be random, chaotic searching, but due to the uncontrollable nature and behaviour, this method does

not pass as an algorithm.

The informed algorithms are mainly represented by Dijkstra's or A\*, but adaptations of uninformed methods to use heuristics can be made. The Dijkstra's algorithm is guaranteed to find the best solution and can be of use when cells have different moving costs: if, for example, a constrain is applied that allows straight moves - the ones that keep the current direction - to cost less than ones that require turning, but is very memory consuming and unnecessary if all cells have the same cost. On the other hand, A\* uses a heuristic approach: for each cell, it computes an approximate cost  $cost(next\_cell) = steps(current\_cell) + distance(next\_cell, goal)$  for each neighboring cell, selecting the one with the lowest possible cost. In general, the *distance* metric is either Euclidian Distance[9], Manhattan Distance (also known as Taxi Cab)[10] or Chebyshev Distance[11], but for mazes, the Taxi Cab is the best choice:

$$X = (2, 6), Y = (8, 1)$$

$$A = (1, 5), B = (2, 15)$$

$$TaxiCab(X, Y) = |2 - 8| + |6 - 1| = 11$$

$$TaxiCab(A, B) = |1 - 2| + |5 - 15| = 11$$

$$Euclid(X, Y) = \sqrt{(2 - 8)^2 + (6 - 1)^2} = 7.81$$

$$Euclid(A, B) = \sqrt{(1 - 2)^2 + (5 - 15)^2} = 10.04$$

The heuristic aspect can also be applied to uninformed methods, such as Depth First Search. During testing, this method proved to be an excellent compromise between speed and distance minimization, by performing slightly slower than DFS (due to distance computing and sorting) and being less precise than A\* (results are presented in Table 3.1 - each maze is shown in Fig.4.14).

Research in this topic can suggest other methods as well, but they tend to be of use only when a preprocessing step is necessary - otherwise, the algorithms shown above outperform any method. For example, Yoshitaka Murata and Yoshihiro Mitani analyzed[12] a modified version of A\* that focused on a preprocessed area of the maze in order to cut down on exploring dead areas. The research shows that, for some cases, this approach is surprisingly better than the standard A\*. They also compared it with A\* with thinning, which also involves a simpler preprocessing method, but concluded that the heavy preprocessing cost added was not worth it for this technique.

Maze size	DFS		DFS Heuristic		A*	
	Speed	Steps	Speed	Steps	Speed	Steps
16x16	0.0007	49	0.0013	49	0.0034	49
64x64	0.0113	305	0.1257	305	0.5538	305
128x128	0.7110	773	0.3107	773	11.4676	773
M16x16	0.0001	53	0.0002	31	0.0061	31
M64x64	0.0057	617	0.1648	143	1.0897	127
M128x128	0.0357	1819	0.0020	277	21.4251	255

Table 3.1: Solving speed measured in seconds. *M* stands for multiple solutions

## 3.2 Implementation

This section refers to the implementation aspect and functionality of the `amazed` Python library. It will discuss available methods, aspects to consider when choosing one and a development plan for future releases regarding the topic at hand.

The library offers the user the following `MazeSolver` classes:

- `DFS` : it is the uninformed approach where each neighboring cell is evaluated in this order: *North, East, South, West*. Because of this aspect, mazes that start in the top left corner and need to exit in the bottom right corner tend to be easy to solve with this method.
- `DFSRandom` : similar to `DFS`, but add neighbors in a random fashion. It behaves far poorly than the standard method, but could be of use if the exploration aspect rewards more than knowledge of the maze's layout (such as in an exploration focused game).
- `Lee` : this class implements the Lee's algorithm[13], being slow even for small mazes, but providing the user with a full distance map from the start cell to every other cell. The algorithm is also used in a function called `flood_fill` that counts how many connected areas are in a given `Maze` object.
- `AStar` : standard implementation of **A\***. The user can provide a *heuristic function*, by default being the Taxi Cab distance.
- `DFSHeuristic` : similar to `AStar`, it also accepts a custom distance function, otherwise falling back to Taxi Cab distance.



When choosing a solving method, one must consider various aspects. If the maze is a simple one, with no additional constraints besides valid moves and neighborhood peeking, for single solution small mazes **DFS** would be a good choice, but **DFSHeuristic** is more versatile if the best solution can be approximated (otherwise,  $A^*$  is a must). However, if the environment is changed in such a way that the agent:

- can see multiple cells ahead
- can move to any cell from any cell (as in a BFS manner)
- can gain/lose momentum when keeping the same direction / taking turns
- can remember only a limited amount of visited cells
- can accumulate a cost each time they move to a new cell
- can cut corners (moving diagonally)
- can find objects of interest scattered in the maze
- can ignore the time constrain
- can ignore the steps constrain

the overall problem changes significantly and each algorithm needs to be adapted to the situation at hand. Such aspects are the scope of future development, in order to provide to the user a way of controlling a maze based on a set of predefined rules of solving. By doing this, a previously generated maze can be evaluated in multiple instances with different constraints.

Another aspect of future research and development is the idea of maze compression, applicable mainly for sparse mazes. This method works by effectively squashing areas in a maze, such as transforming a corridor starting from  $cell_{X,Y}$  and going to  $cell_{X,Y+N}$ , where  $N > 2$  to only consist of two cells,  $cell_{X,Y}$  and  $cell_{X,Y+2}$ , or by merging together open areas where no walls are presents (for example, a 3x3 area with no walls will be converted in only one cell, with all connections from the original border now bound to it). The motivation for this method is to create an efficient maze data structure such that Dijkstra can be applied to it and, hopefully, yield better results.

# Chapter 4

## Applicability

This chapter will cover examples of applicability regarding maze generation and solving algorithms. It will focus on real-life examples encountered during research but also provide theoretical applications, along with a game called *Both*, which uses constraints to create an environment where agents need to rely on negotiations to win. Finally, it will present a winning strategy evolved using Genetic Algorithms as a starting point for human vs computer challenges.

### 4.1 Real life examples

Mazes are thought as a theoretical concept, usually present only as human challenges, but with no real applicability behind them. While this may be true for simple cases, studies show a variety of use cases, ranging from unknown area mapping [14] (where the focus is first to understand the layout - which may not always be a simple rectangle - then come up with a solving algorithm), guiding blind people [15] based on natural language given directions, sequential goal completion [16], to robotics challenges such as MicroMouse[17](where autonomous agents have to find a path to the center of an unknown 16x16 maze using limited computational and physical resources), city/warehouse mapping for travel cost minimization or cognitive evaluation in aging and neurodegenerative diseases[18].

On the theoretical side, maze generation has been used as means of controlling a training dataset[19], where the authors argue that their ability to fine tune differences in dataset instances is an important aspect when it comes to understanding how models (such as transformers) learn.

In addition to this example, a rather novel esoteric language can also be created using mazes, by using a three part information to "run" a program. The language itself is just a rough working concept, capable of simple, but interesting programs, but a well-developed prototype can be built upon this foundation. The language works as follows: a maze will define a possible execution graph and encode a specific action in each cell, while a solution token will indicate the start & end points, along with an ordered list of steps, followed by an input token, encoded in binary. Based on the following rules:

- always start at index 0 of the input
- always start with a standard file opened for output
- the execution can start and end at any cell in the maze (as described by *the solution token*)
- a stored value is always available

and the following actions:

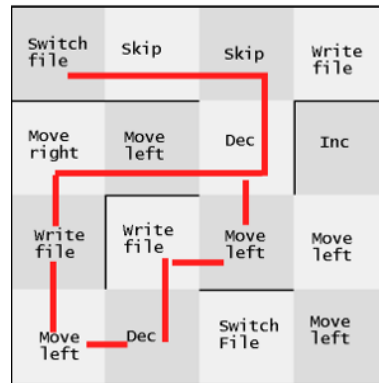
- **SWITCH FILE** : read the current position in the input and expect a number X. The next X cells will describe a path for a file. Open that file for writing.
- **SKIP** : read the current position in the input and expect a number X. Move the indicator for the current position by X cells to the right.
- **WRITE FILE** : write the current stored value to the current opened file.
- **DEC** : decrease by 1 the current stored value
- **INC** : increase by 1 the current stored value
- **MOVE LEFT** : move the indicator for the current position to the left.
- **MOVE RIGHT** : move the indicator for the current position to the right.

one could create two different programs based on the same maze. Figure 4.1a paired with the input token `b'16'dlrow_olleh.txt` and solution token `3,1 > 3,2 >`  
`3,3 > 2,3 > 2,2 > 2,3 > 2,2 > 2,1 > 2,2 > 2,1 > 2,2 > 2,1 > 2,2 >`  
`2,1 > 2,2 > 2,1 > 2,2 > 2,1 > 2,2 > 2,1 > 2,2 > 2,1 > 2,2 > 2,1 >`  
`2,2 > 2,1 > 2,2 > 2,1 > 2,2 > 2,1 > 2,2 > 2,1 > 2,2 > 2,1 > 2,2 >`

2,1 > 2,2 will write hello\_world in hello\_world.txt. Figure 4.1b paired with the input token b'15' /tmp/caesar.txtb'3' PGOJ and the solution token 0,0 > 0,1 > 0,2 > 1,2 > 1,1 > 1,0 > 2,0 > 3,0 > 3,1 > 2,1 > 2,2 > 1,2 > 1,1 > 1,0 > 2,0 > 3,0 > 3,1 > 2,1 > 2,2 > 1,2 will write into /tmp/caesar.txt the text INFO.



(a) Solution for hello\_world



(b) Solution for INFO

## 4.2 amazed - Python3 library

This paper and all the data provided with it make use of *amazed*, a Python library capable of both generating and solving mazes. The motivation behind Python as the language to use is dictated by the wide availability of libraries for various scopes, ranging from machine learning platforms to data structures architectures. *amazed* is based on several other libraries:

- NumPy[20] : widely used as the most versatile library for matrix data manipulation, it handles internal representation of mazes and greatly speeds up calculations.

- `Pillow`[21] : a known fork of `PIL` (Python Image Library[22]) - which was discontinued in 2009 - capable of working with various image formats, mainly `.png`, `.jpg` and `.gif`, it is used to export mazes for a better graphical visualization.
- `Keras`[23] : along with `sklearn` & `pytorch` is one of the industry standard libraries when it comes to machine learning concepts, allowing the user to both build a model and train it using custom made datasets. It is responsible of loading the Generative Adversarial Model in release builds and for training & debugging in development builds.

Even if Python does not enforce an object oriented design, the library offers a friendly class hierarchy, having as base classes 1)`Sculptor`, for every algorithm that can generate a maze, capable of exporting either the final product as either an image, a text representation, an adjacency list, a graph matrix or, in a more dynamic fashion, as a GIF, and 2)`MazeSolver`, which all solving algorithms inherit from, capable of finding a path (if one exists) from a `self.start` cell to a `self.finish` cell, then possibly exporting the process either as an image or as a GIF. Careful attention was given to the graphical aspect because of two main reasons: first, this method helps greatly in the debugging process, allowing errors to be caught and fixed early on, and second, because most other libraries either provide only image representation (most of the time with one pixel wide cells), without the ability to generate GIFs. Some of the most downloaded libraries from PyPI are listed and discussed below:

- `maze` : generate weaved mazes using Kruskal and also solve them. Can be exported as SVG.
- `maze-dataset` : focused more on dataset generation, has only four generation algorithms and one solver algorithm.
- `maze-world` : only has the Wilson's algorithm, using Dijkstra to solve it.
- `maze-nd` : generates mazes in N dimensions using Prim's algorithm.
- `mazely` : generates mazes with DFS and solves them with BFS. Mazes can be displayed as an image.

The code for `amazed` can be viewed on GitHub[24] or on PyPI[24], and it can be installed with `pip install amazed`.

As for future releases, `amazed` is in active development as of writing this paper. Plans include **1)** a total rework on the underlying data structures to eliminate the problem of double walls, thus reducing the overall memory impact and increasing generation & solving time; **2)** a port to C++ for calculations with Python bindings is also considered, but not guaranteed - tests are required to determine the overall impact of this modification; **3)** multi-threaded support which would decrease generation time for very large mazes - with the mention that the impact could be lower than expected because of Python's way of implementing threads; **4)** in the same manner for solving, a multi-agent solver for very large mazes is also considered; **5)** adding a user interface for generating mazes using the keyboard arrow keys; **6)** adding a way to upload a maze (with constraints!) to a centralized "arena" where other users can upload "solvers" for them (thus creating a centralized space where new environments can be uploaded & tested); **7)** on the applicability side, adding support for photo pre-processing (given a photo with a hand drawn maze, recreate it in memory is one of the main future goals, along with **8)** video pre-processing (given a video with a first person perspective, recreate the maze), **9)** geolocation pre-processing (given a list of coordinates, map and recreate a maze according to them) and **10)** language pre-processing (given a text describing a set of directions, recreate the maze). However, to help keep development organized, the last 4 points would be implemented in four separate libraries based on the original one, mainly to not overcomplicate it.

### 4.3 CoLab - a maze based game

**Cooperative Labyrinth**, shorten to **CoLab**, is a two players maze based game, in which agents must race to accumulate points in an unknown environment by being the first to reach the finish cell. The game is focused on exploration and collaboration, initially inspired by The Iterated Prisoner's Dilemma[25].

As described in Chapter 1, the IPD problem presents itself as a series of episodes in which two *rational agents* have two choices: either cooperate for a mutual benefit or defect for a (usually bigger) individual gain. For the original score matrix (see Fig.1.1), when there is only one episode and no additional information is provided, the theory states that each agent must defect to guarantee a maximum personal reward. Yet, for an *iterated* version, the theory states that no perfect solution (one that will get the maximum score possible) exists, and this is because a solution must always be

evaluated against another one. A simple yet powerful solution came out to be *Tit-for-Tat*, which checks 3 out of the 4 attributes that Robert Axelrod described as *necessary*: 1) **the strategy is nice**, meaning that it won't defect first, 2) **the strategy is retaliating**, meaning that it won't stand being taken advantage of, 3) **the strategy is non-envious**, meaning that it won't try to gain an advantage over the opponent.

Thus, researches began experiments and published various methods of supporting this idea, either by direct comparison with known strategies or by evolving a population of such strategies and seeing what characteristics it nourishes. Again, the best strategy was proved to be *Tit-for-Tat* or similar variations (such as a *Tit for Two Tats*, in which the agent retaliates if the opponent deflected twice).

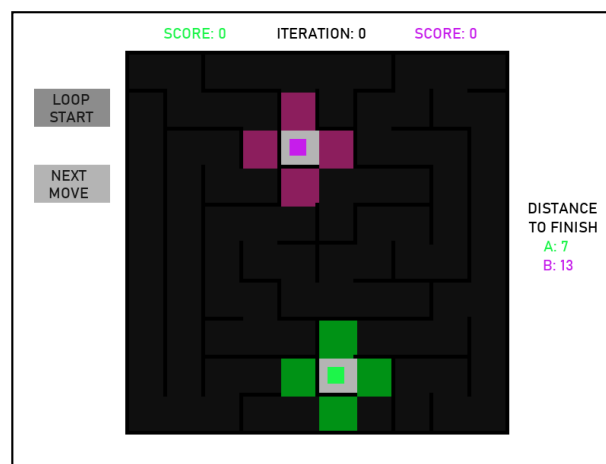


Figure 4.2: CoLab GUI

The game **CoLab** sets out to test this theory in a more complex environment. With an interactive GUI (see Fig. 4.2), two agents are spawned in a randomly generated 10x10 maze, seeking as fast as possible a finish cell, either by random & fortunate discovery of the path or by jolly cooperation. The interface provides two buttons: **LOOP**, which makes the game run until either the round is over or the button is pressed again, and **NEXT STEP**, which only runs one step, along with some other metrics, such as current score for each player (calculated over the total number of rounds), a current step counter and an approximate distance based on what each player knows. The logic loop is split into two parts:

- first, the negotiation part, only if both players do not have a fully known path that starts from their current position to the finish position:
  - each player decides if they want to cooperate or not. If both answered *Yes*, this process continues to the next step. Otherwise, the negotiation ends.

- then, each player receives their opponent’s **proposal**, which they evaluate and decide if the information exchange will happen or not (an important note to mention is that the exchange will only happen between each other’s offer and will not be influenced by the request - for example,  $Player_A$  can come up with  $Offer_A = cell_{X_1, Y_1}$ ,  $Request_A = cell_{X_2, Y_2}$  and  $Player_B$  can come up with  $Offer_B = cell_{X_3, Y_3}$ ,  $Request_B = cell_{X_4, Y_4}$ ; if they both accept the proposal,  $Player_A$  will know  $Offer_B$  and  $Player_B$  will know  $Offer_A$ )
- if the players decided against the proposal, this negotiation step is retried two more times (a total of 3 possible negotiation can happen, at the end of which an information exchange is not required). Otherwise, if they both agreed, the negotiation stops and the information is exchanged,
- with the information known at this point, each player decides what is the best possible move in the current context - the default logic uses *DFSHeuristic* to create a potential list of steps to the end goal, taking into account both visible cells and unknown cells.
- next, the current situation is evaluated:
  - has any player won?
  - is it a draw?
  - have the maximum number of iterations been reached?

If none of these conditions are met, the loop repeats.

The game overall configuration went through a trial and error phase to balance out the scores and multipliers. First, the idea was to let the game run for as long as possible, but this yielded poor results for autonomous agents, almost all getting stuck with no will to negotiate. After this, a maximum allowed number of 1000 iterations was tested, then lowered to 500 and finally setting to 300, following the idea that the worst possible strategy (the one that does not cooperate at all) would take about 100 iterations to map the maze, then the rest to reach the goal. An empirical test was then run where various levels of cooperation were implemented and averaged on 100 10x10 generated mazes. The results are displayed in Table 4.1, indicating that between 80 and 150 iterations is enough to cover both cooperating agents and brute-force strategies.



Unknown cells	$P_0$	$P_{25}$	$P_{50}$	$P_{75}$	$P_{100}$
95	134.26	84.91	84.52	83.81	90.32

Table 4.1: Average number of rounds required for finishing a 10x10 maze for random agents with varying levels of cooperation, where  $P_X$  means that the agent discovers a cell through cooperation  $X\%$  of the time.

One interesting aspect that this table reveals it that more cooperation does not always mean faster goal finding. This is because not all known cells are relevant (for example, as in Fig.4.5c), yet known cells are (almost) always chosen over unknown ones, despite them possibly leading to a dead end.

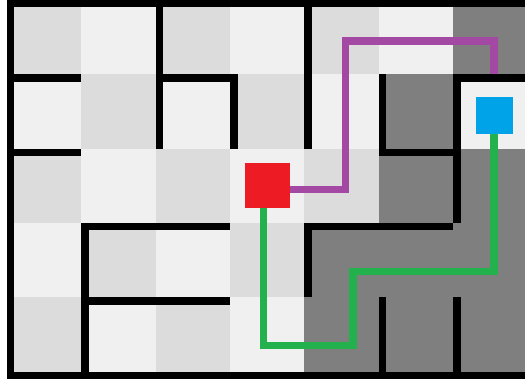


Figure 4.3: Taking the known path (purple) leads to (an unknown) dead end.

Next, the initial idea was to split the all cells of the maze in half, such that each player knows exactly  $\frac{\text{rows} \cdot \text{columns}}{2}$  cells (as seen in Fig.4.4). However, this method proved to be unbalanced because it introduces too much randomness: many times one player would know a good part (if not all!) of the best possible route to the end goal right from the start (as seen in Fig.4.5a), which in turn discouraged any form of negotiation. To counter this, each player was set to start with just five known cells - their respective neighbors, plus the final goal. This led to more balanced experience, because agents now rely also on exploration to have a better pool of knowledge to exchange from.

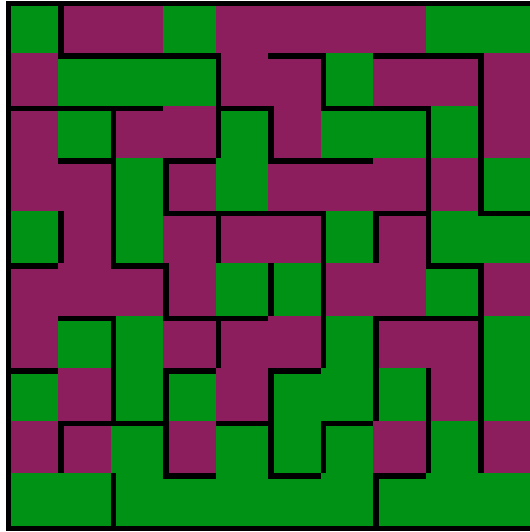
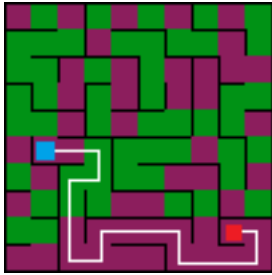
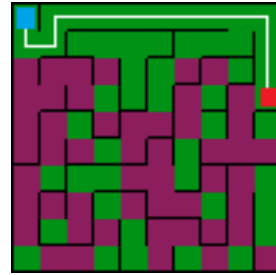


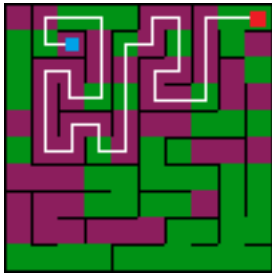
Figure 4.4: Evenly split maze



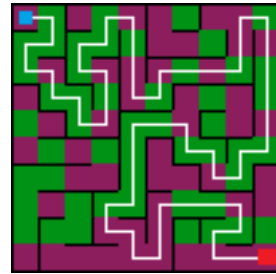
(a) Great advantage for **purple**



(b) Instant solve for **green**



(c) Unfair distribution - many cells for **green** are useless.



(d) Fair distribution - both **green** (35) and **purple** (23) have close enough knowledge

Figure 4.5: Various cases with even splitting a maze

Another aspect that was modified during development was how players react to "moving" into unknown cells when such a move was impossible. The initial idea was to leave the player with a simple "error" that the strategy should take into consideration - nevertheless, this idea was changed because it led to very little exploration incentive: agents would tend to just try and exchange any form of information they could (even running in circles to *not stand on the same cell*) instead of actually trying to explore new cells. In this version, if a player tries to move to an unknown position and

this move turns out to be impossible, then that cell becomes visible.

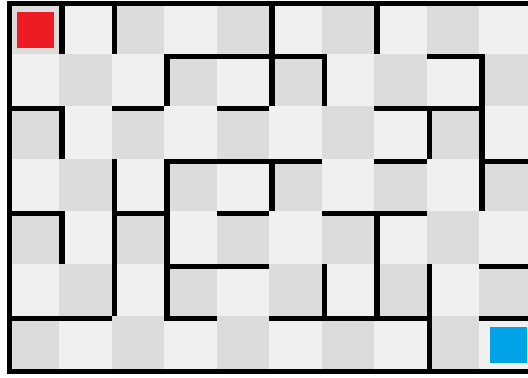
How each start and finish point is selected was also tested and balanced. First, the idea was to select four random points, also called *truly random selection*, but this method proved very hard to control and balance: one player could spawn tens of steps away from finish, while the other could spawn right next to the finish. Again, the uncontrollable random aspect influenced too much the main focus of the game, so the next idea tried to fix that. Instead of 4 total random points, generate only 2 random start points and another random distance for the finish, then put each player's goal at the same distance from the start. The idea (called *partial random selection*) itself is rather interesting, but the implementation brings a big performance impact, because a breath-first-search needs to be run for each player. A modified version (called *inverse partial random selection*) would be to select a random finish point and a random distance, then perform a BFS up until said distance is reached in two different points, which will be the start points for each agent. The final and most simple solution (called *fair random selection*) was to select two random points in the maze  $X, Y$  and set

$$start_A = X, finish_A = Y$$

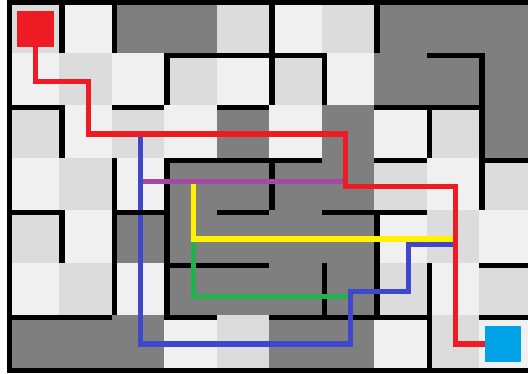
$$start_B = Y, finish_B = X$$

thus insuring the same distance, the same complexity and a reasonable computing time.

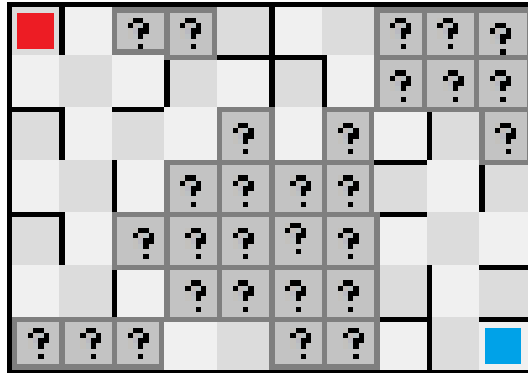
With the core functionally fully working and balanced, the next step was to provide a way of using autonomous agents, either trained or hard coded. The class *Player* was created, serving as a template for user crafted strategies. By default, it selects a random cell to offer while also requesting a random unknown cell. The best move is defined as the next step as dictated by a DFS heuristic approach (for a graphical example, see Fig.4.6) in an attempt to approximate a possible solution, which is recalculated at periodic intervals or when a move leads into a wall. For the negotiation part, each agent calculates two possible solutions every time they need to reroute: first, by ignoring all unknown cells, trying to find a direct path to the goal - if they succeed, no more negotiations will be accepted (the idea is that an alternative, more efficient solution is not worth helping the opponent) - and second, by taking into account all cells, with a slight preference towards known ones.



(a) Full overview



(b) Possible routes

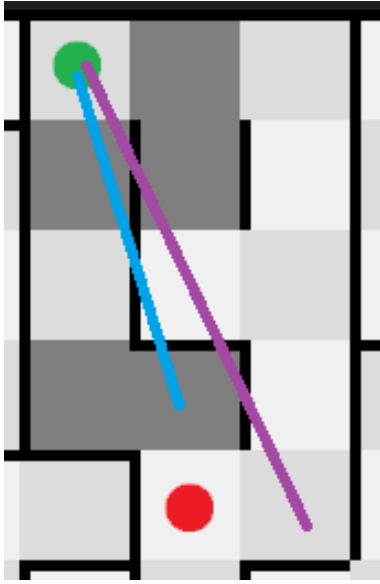


(c) Player's view

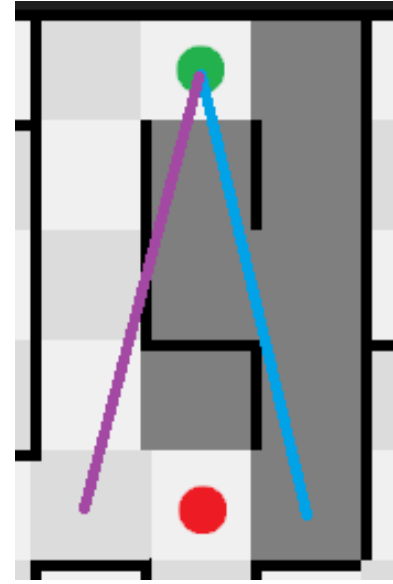
Figure 4.6: Different views on possible routes for a player.

The idea to add a form of penalty to unknown cells arose because initial algorithms behaved poorly on unexplored areas, most of the time choosing to circle in place. For example, in Figure 4.7a, the blue distance is clearly lower than the purple one, and it would be the best possible choice, yet in Figure 4.7b, when both distances are equal, there is no benefit in exploring.

As a more complex environment of the Iterated Prisoner's Dilemma, one must ask provide an answer to the question: *what does it mean to cooperate and what does it mean to deflect?* Shortly put, in the current environment, these two terms cannot be dis-



(a) Figure 4.7a



(b) Figure 4.7b

Figure 4.7: Agent is red, goal is green. Dark gray marks unknown cells.

cretely defined and must be interpreted as a continuous term, mainly because of the vast number of situations where not even a human can determine if a negotiation can be marked as either cooperation or deflect. The analysis shows four possible combinations:

- both players offer and request each other cells of similar value (for example, a cell closer to the current position or exactly the requested cell) - this category contains even "poor" deals, because both players are affected in equal form by such an exchange
- one players offers a high value cell (for example, one closer to the opponents position) while the other offers a low value cell (for example, one that would not be of use for the player)
- one player refuses cooperation (either because it has found a path or because its strategy says so)
- both players refuse to collaborate

A trivial idea for a "cell score" function would be to calculate a distance between the requested cell and the offered one, like this:

$$score = \frac{1}{C + |cell_{REQ} - cell_{OFF}|}$$

where  $C$  is a small constant to account for division by 0. However, this score function only takes into account information known by the agent, without accounting for the possibility that the opponent is willing to offer a cell of greater value in exchange for a cell of equal value. In other words, players will tend to value less an offer different than what they requested, even if said offer could be better for both players.

Another simple metric can take into account the distance between the offered cell and the player's current position. This aspect follows the idea that closer cells are more important than further cells, so if the opponent offers a cell closer to the current position, it should be better. The function can be written as this:

$$score = \frac{1}{C + ||player - cell_{REQ}|| - ||player - cell_{OFF}||}$$

where  $C$  is a small constant as before. Nonetheless, this method comes with its own caveats. If the opponent offers a cell closer to the player, but in the opposite direction of where the player was headed, this will result in a better score, but a bad trade. A quick solution to this would be to average multiple points from the current path, thus ensuring that offered cells tend to be closer to the player's direction.

Counting the total neighbors of a cell and computing a score based on how many of them are known is another idea that favours "frontier cells": the ones that are near the border of knowledge. One such function could be:

$$score = \frac{1}{C + |cells_{known} - cells_{unknown}|}$$

which filters useful frontier cells from useless ones (for example, if an unknown cell is surrounded by 8 known cells, it is not of particular interest, but if a cell has 4 known and 4 unknown neighbors, then it is a valuable piece of information).

Various other metrics and score functions can be taken into consideration: how useful previous cells were, how big are the areas of known space that will be connected (or for which the distance will be smaller) if the trade takes place, how likely this cell is to be of use in the future and many more. As stated earlier, each metric will have a different impact on the game based on how the *cooperation* and *deflection* actions are defined.

### 4.3.1 CopyPlayer

One tested strategy is called `CopyPlayer`. It is modeled after `Tit-for-Tat`, defining the *cooperation* as a successful trade and the *deflection* as an unsuccessful one. In the idea of cooperation, this strategy will always try to offer the requested cell or accept the offered cell, regardless of "personal" directions, otherwise it will always request and offer random cells.

One key difference between this and the *Tit-for-Tat* strategy is that, two `CopyPlayer` agents will never cooperate, because they won't be able to synchronize. This problem is addressed in the next hard coded strategy, `RememberMe`.

### 4.3.2 RememberMe

The idea behind `RememberMe` is to always maintain a list of cells that were needed as some point. If the opponent offers the a cell from this set, immediately accept, regardless of the requested cell. Will dominate and take advantage of `CopyPlayer` because of this "remember" mechanic.

### 4.3.3 Evolved strategies

Similar to how other researches used Evolving Algorithms to find the best strategy for the Iterated Prisoner's Dilemma, this paper uses a Genetic Algorithm approach to evolve a similar strategy. A GA is a simple yet effective algorithm that relies on a population of individuals that get better over time by chromosome operators (*selection*, *crossover* and *mutation*), with respect to a *fitness function*.

Multiple runs with various hyper-parameters were done, and the chromosome changed over time, starting from basic multipliers to fine tuned arguments that encode what a "good" cell is. The chromosome structure as of writing this paper is described below:

- `DNEXT` : defines which unknown cell in the current stack should be requested.
- `DXNEXT` : defines which known cell in the current stack should be requested.
- `MOFF` : multiplier used to express how important it is to be requested the offered cell.

- `MREQ` : multiplier used to express how important it is to be offered the requested cell.
- `SCORE_THRESHOLD` : defines a maximum "bad" score. If the current proposal score is under or equal to this value, it will be accepted, otherwise denied.
- `EXPLORATION_CHANCE` : defines a chance to take the a "suboptimal" route when calculating a solution with `DFSHeuristic`. Inspired by the *temperature* used in Simulated Annealing algorithms[30].
- `ATTEMPT_MODIFIER` : how will each proposal score change based on what the current negotiation round is.
- `UNKNOWN_CELL_MODIFIER` : defines how the value for an unknown cell will change when a path is calculated.

The fitness function was also adapted during training. The initial method was

$$fitness_1 = \frac{total\_rounds\_won}{total\_games\_played}$$

but this lead to many agents depending too much on the starting state of the maze, thus the overall population did not evolve much. The next method tries to rework the binary aspect of a game (either the agent wins, and is rewarded, or it gets nothing) by treating differently each possible scenario:

$$fitness_2 = \begin{cases} +2 & \text{if the player won,} \\ -2 & \text{if neither player won,} \\ +1 & \text{on a draw,} \\ -1 & \text{on a lose.} \end{cases}$$

$$fitness_2 = \frac{fitness_2}{total\_games\_played \cdot 2} \in [-1, 1]$$

The squashing is done to ensure an even score even if the total number of games played is changed (this in turn helps to fine tune the hyper parameters of the GA). Yet, this method did not find a best strategy for all possible cases: if a player knows a a fully connected border (meaning that each pair of cells can be accessed with the current known information) around an unknown part of the maze, it will still consider offers for that region, despite it being of no use (for a graphical explanation, see Fig.4.8).



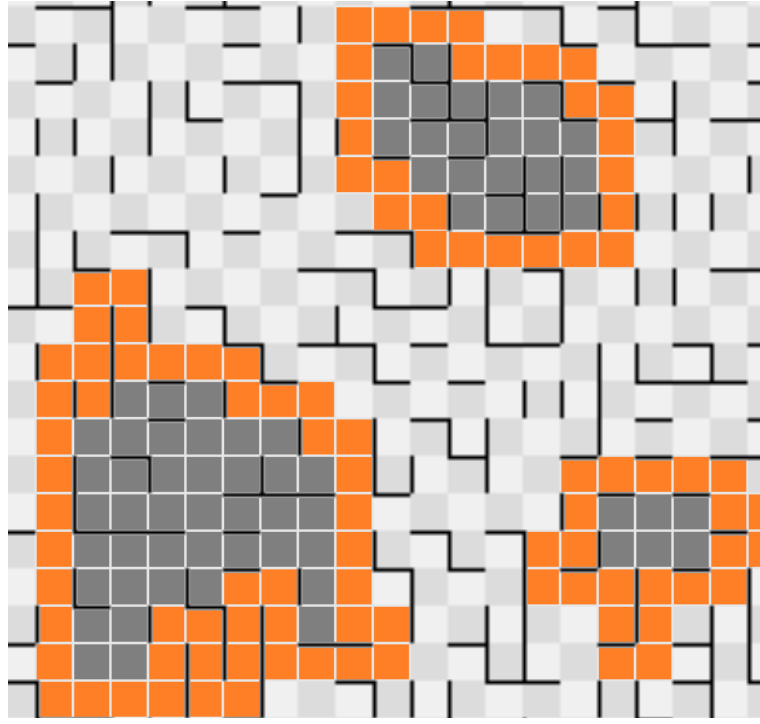


Figure 4.8: Fully connected border around an unknown region of no interest. Gray marks an unknown cell, while orange marks the border.

The final tested fitness function tries to combine both using  $A^*$ . The function can be rewritten as follows:

$$fitness_3 = \begin{cases} +1 & \text{if the player won,} \\ \frac{+1}{1+A^*_{steps}(current\_cell, goal\_cell)} & \text{if the player lost or the game finished} \\ +0.5 & \text{if it is a draw} \end{cases}$$

This approach also takes into consideration how far away an agent is from the goal cell, ensuring that, for example, a game that ends with *Player A* winning and *Player B* losing only by a 1-2 steps is different that a game where *Player A* wins and *Player B* is at 20 steps away from the goal cell.

Even with these improvements, the GA did not yield a best strategy, regardless of how the hyper parameters were set. Multiple runs provided strategies that win in about 2 out of 3 matches, but none were perfect. Handcrafted initial individuals also did not improve in a remarkable way, even with elitism (set between 5% and 25%). One possible direction that is worth exploring is a simpler chromosome structure, focused only on offers and a score function for them, without relying on distances, the possibility of future uses or actual usability.

Overall, the Genetic Algorithm seems promising in creating a winning strategy,

with multiple possible fitness function that each focus on different aspects, but further studying and experimentation is required with various chromosome configurations to determine the best match. Careful consideration needs to be taken when choosing the *fitness function*, as this step, as shown above, has vastly more influence on the final outcome than other aspects.

#### 4.3.4 Future work & directions

One main aspect that **CoLab** lacks is separability between classes. Both the `Player` class and the `Maze` class are interdependent, which opens up possibilities for cheating. Because the intent is to release the game in an online environment where anonymous users can upload agents to compete, this aspect must be fixed as soon as possible, leaving the `GameMaster` class exposing only necessary APIs.

Another part that will need an update are the available hard coded strategies. The Iterated Prisoner's Dilemma ranked *Tit-for-Tat* as the best strategy, but others (such as *Tit-for-Two-Tats*[31], *Win-Stay-Lose-Switch*[32], *Grim Trigger*[33]) ranked in the top must also be considered and tested, because, due to the continuous aspect of the environment, they may yield vastly different results in contrast to a discreet implementation.

Finally, the negotiation area can be improved. As of writing this paper, the game will always perform at most three attempts at negotiation, which comes up to 900 maximum information exchanges over a 10x10 maze, indicating a bias towards negotiation over brute force cooperation. Thus, research is required to determine a balanced number of such events, with respect to the size of the maze.

# Conclusions

Maze studying is an area of research that has applications in many branches of society, both physical and theoretical, and cannot be considered a fully understood chapter in computer science. Being simple in design, they represent a way to model various problems and study them in controllable environments, ranging from shortest path challenges to area mapping and mutual cooperation.

This paper aimed to provide an overview on the state of the art methods and algorithms, to challenge concepts and test them, to provide various applications and to bring develop tools that facilitate the described scope. It showed how General Adversarial Networks can be used to generate mazes, it provided methods to "rank" a maze based on its internal structure, it discussed data representation and compression methods, it built a complex game using a handcrafted library, it developed strategies for the game, finally providing a road map for future work on each chapter. Based on the presented results, it can be concluded that the paper achieved its scope that it set out to, thus bringing to light new research on the topic at hand.

## Resources

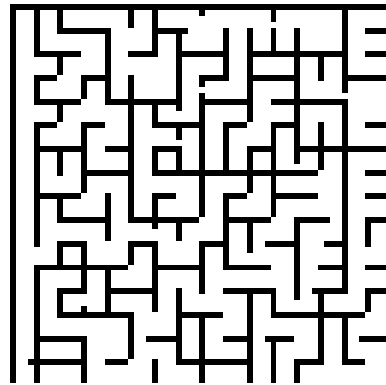


Figure 4.9: Early iteration of a maze generated using image-trained GANs

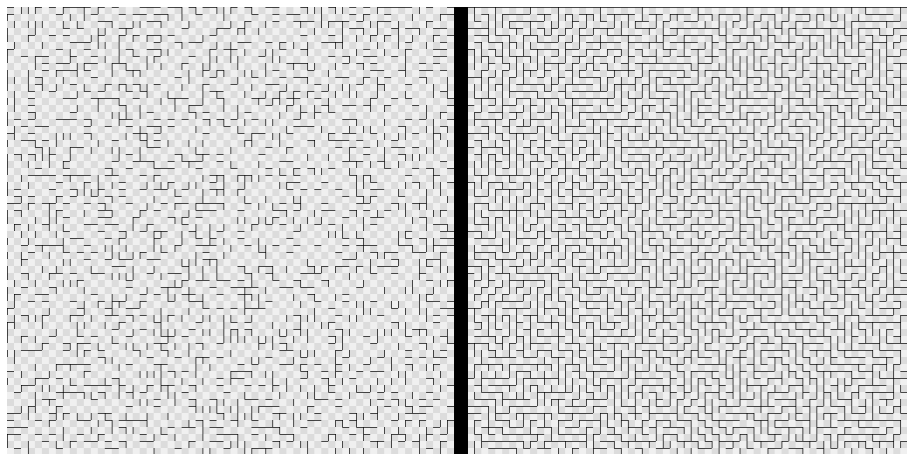


Figure 4.10: A comparison between a GAN created maze and a DFS one

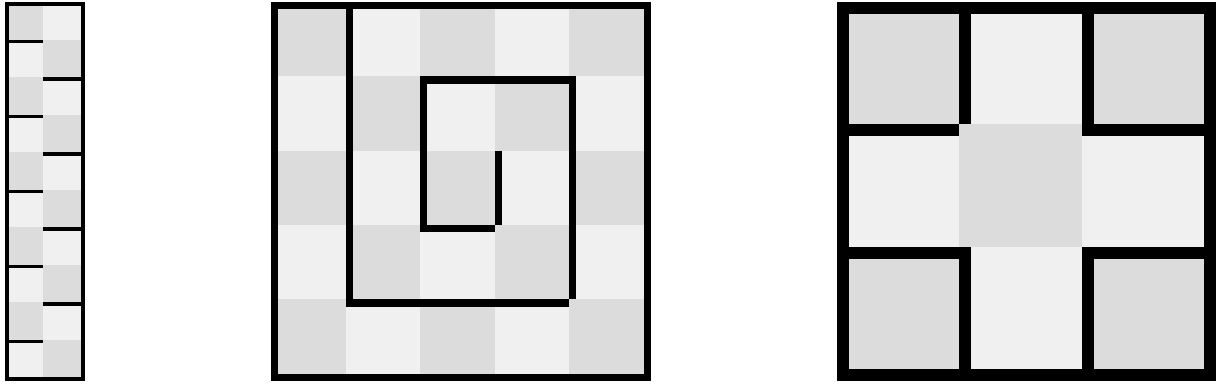


Figure 4.11: Maze patterns. From left to right: snake, spiral, T intersection

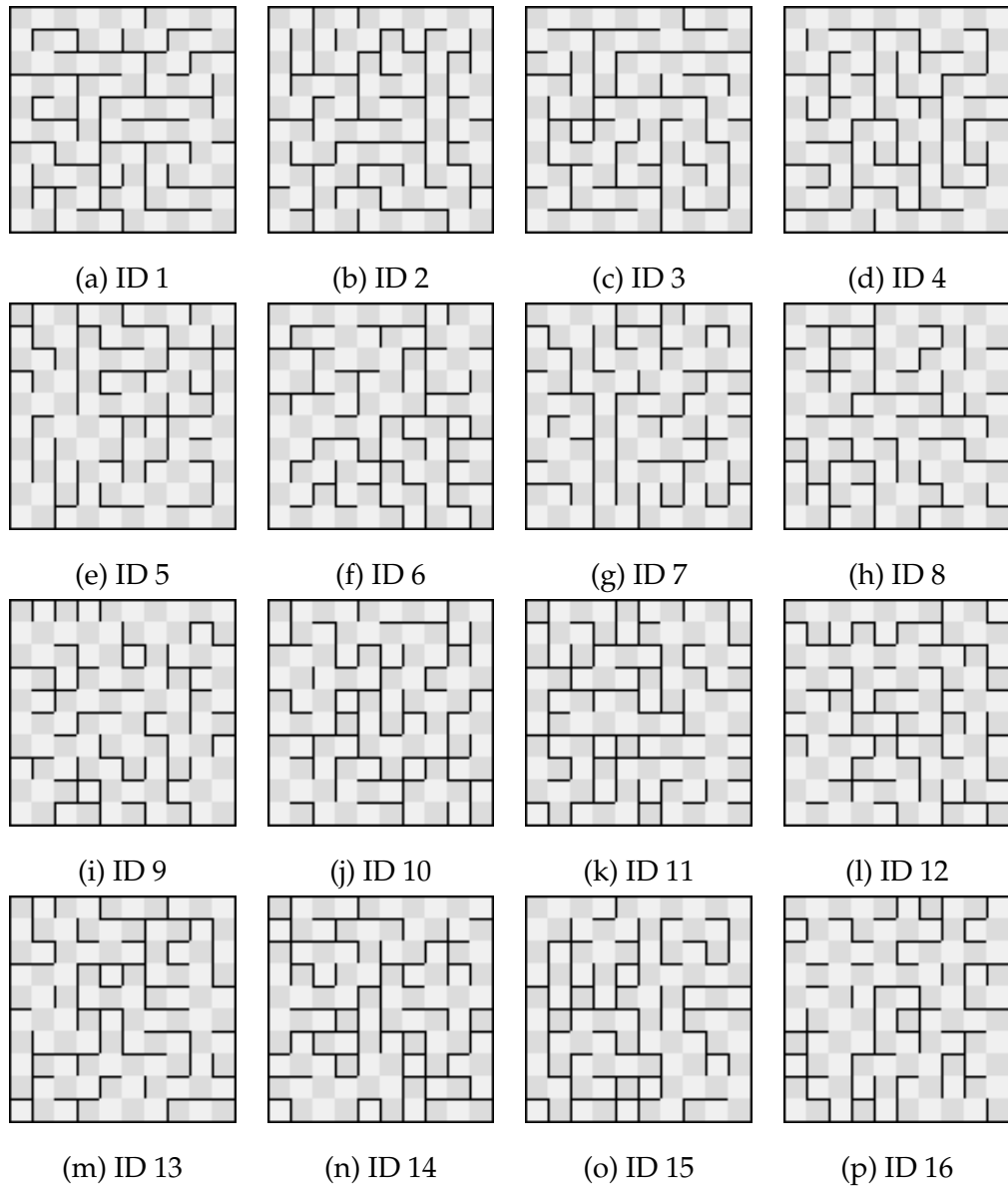


Figure 4.12: Mazes evolved by GA

Cell type	Multiplier
0 walls	-0.1
1 wall	0.1
2 walls	0.4
3 walls	0.2
4 walls	-1

Table 4.2: Values for  $M_4^{general}$  used in GA. They should favor mazes with long, straight corridors more.

















Cell type	Distribution	Cell type	Distribution
	0.06 %		2.40 %
	2.58 %		2.27 %
	2.45 %		11.84 %
	15.74 %		10.73 %
	11.49 %		16.77 %
	11.85 %		2.74 %
	3.29 %		3.38 %
	2.41 %		0.00 %

Table 4.3: Distribution of cell types for Depth First Search mazes

















Cell type	Distribution	Cell type	Distribution
	5.26 %		5.24 %
	8.27 %		7.23 %
	5.95 %		4.95 %
	8.77 %		6.76 %
	8.33 %		8.91 %
	5.49 %		3.98 %
	7.35 %		8.08 %
	4.00 %		1.43 %

Table 4.4: Distribution of cell types for Random Carving mazes

















Cell type	Distribution	Cell type	Distribution
	3.23 %		5.69 %
	5.59 %		5.33 %
	5.28 %		7.09 %
	8.25 %		7.01 %
	6.59 %		8.54 %
	7.05 %		8.23 %
	7.32 %		7.41 %
	7.39 %		0.00 %

Table 4.5: Distribution of cell types for Random Kruskal mazes

















Cell type	Distribution	Cell type	Distribution
	5.14 %		5.14 %
	5.30 %		5.66 %
	5.22 %		5.66 %
	8.82 %		5.50 %
	5.74 %		8.78 %
	5.44 %		8.19 %
	8.61 %		8.73 %
	8.07 %		0.00 %

Table 4.6: Distribution of cell types for Random Prim mazes

















Cell type	Distribution	Cell type	Distribution
	9.47 %		5.36 %
	11.93 %		5.18 %
	11.61 %		5.80 %
	2.03 %		5.98 %
	5.82 %		15.68 %
	6.12 %		6.19 %
	1.25 %		6.17 %
	1.41 %		0.00 %

Table 4.7: Distribution of cell types for Recursive Division mazes



















Cell type	Distribution	Cell type	Distribution
	0.00 %		0.00 %
	0.00 %		0.00 %
	0.00 %		4.49 %
	39.92 %		4.00 %
	5.00 %		40.08 %
	4.51 %		0.00 %
	0.98 %		1.02 %
	0.00 %		0.00 %

Table 4.8: Distribution of cell types for Spiral mazes

















Cell type	Distribution	Cell type	Distribution
	0.12 %		3.07 %
	1.43 %		1.57 %
	2.66 %		12.55 %
	15.76 %		11.19 %
	12.44 %		15.74 %
	12.50 %		2.76 %
	2.91 %		2.46 %
	2.84 %		0.00 %

Table 4.9: Distribution of cell types for Hunt And Kill mazes

















Cell type	Distribution	Cell type	Distribution
	0.00 %		12.01 %
	0.00 %		0.00 %
	11.47 %		10.33 %
	14.92 %		1.00 %
	0.00 %		14.20 %
	10.59 %		12.75 %
	0.00 %		0.00 %
	12.73 %		0.00 %

Table 4.10: Distribution of cell types for Binary Tree mazes

















Cell type	Distribution	Cell type	Distribution
	0.08 %		2.37 %
	2.40 %		2.45 %
	2.58 %		12.84 %
	14.42 %		12.16 %
	12.44 %		13.76 %
	12.54 %		3.04 %
	3.11 %		3.10 %
	2.71 %		0.00 %

Table 4.11: Distribution of cell types for Aldous Broder mazes

















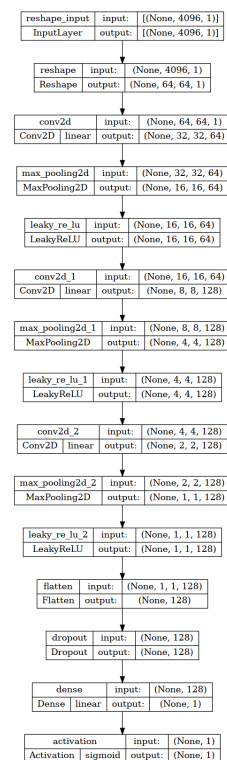
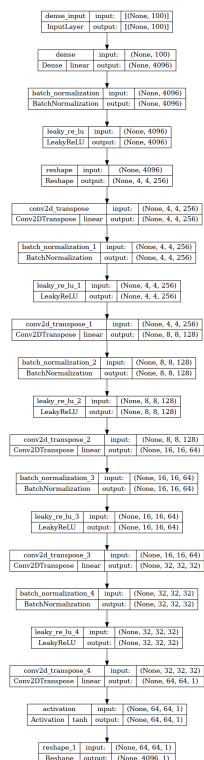
Cell type	Distribution	Cell type	Distribution
	2.18 %		10.03 %
	4.01 %		2.45 %
	4.47 %		7.31 %
	11.69 %		7.20 %
	4.73 %		13.88 %
	4.73 %		12.63 %
	7.17 %		0.00 %
	7.52 %		0.00 %

Table 4.12: Distribution of cell types for Sidewinder mazes



(a) Generator

(b) Discriminator

Figure 4.13: Model architecture

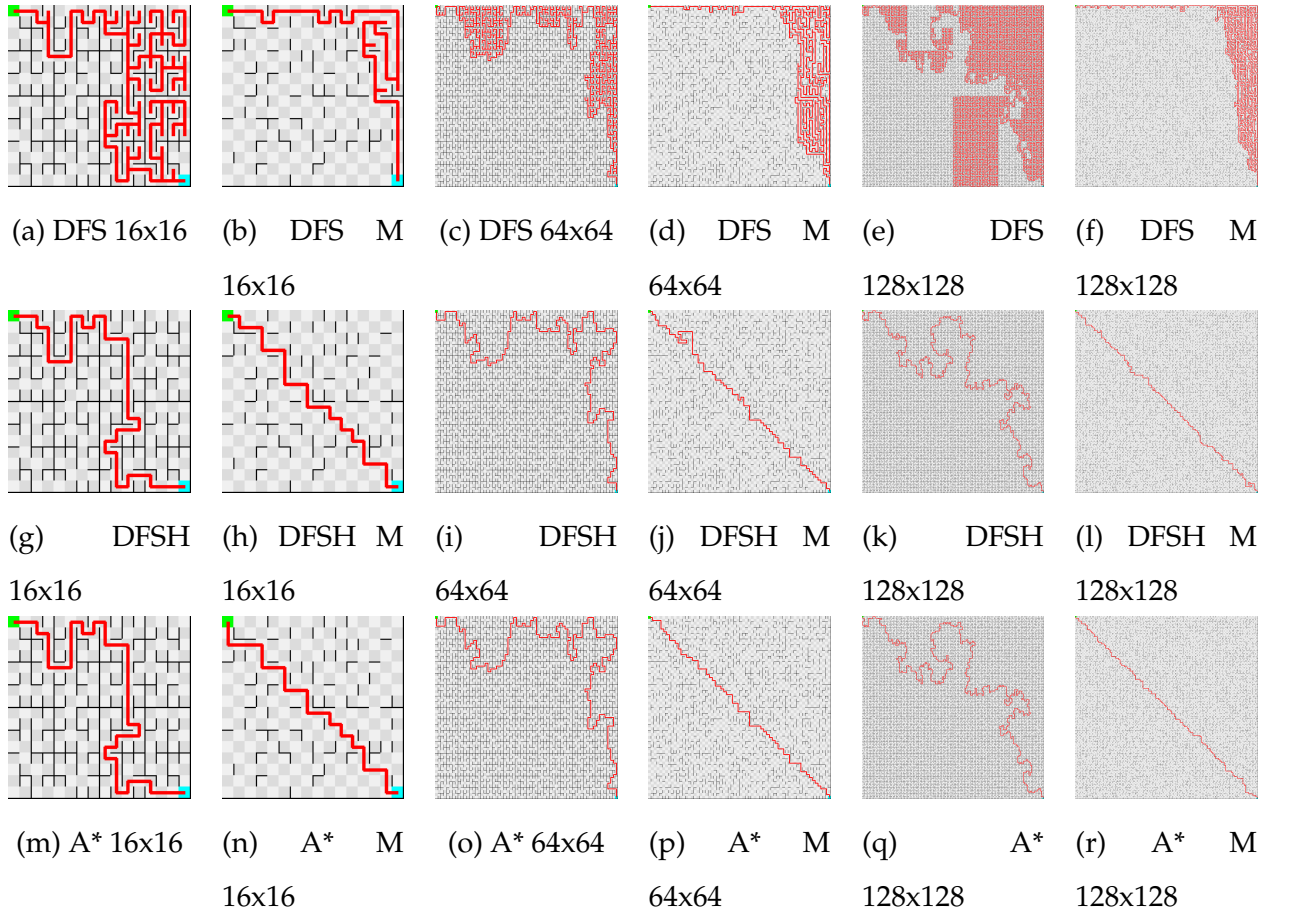


Figure 4.14: Mazes evaluated in Table 3.1

# Bibliography

[1] Rule 110

Retrieved from [https://en.wikipedia.org/wiki/Rule\\_110](https://en.wikipedia.org/wiki/Rule_110)

[2] Maze rule for cellular automaton

<https://conwaylife.com/wiki/OCA:Maze>

[3] Wave function collapse

GitHub repository of author

<https://github.com/mxgmn/WaveFunctionCollapse>

[4] GitHub Repository with various sources

<https://github.com/adipeterca/master-research>

[5] A Review of Various Maze Solving Algorithms

Based on Graph Theory

[https://www.researchgate.net/publication/331481380\\_A\\_Review\\_of\\_Various\\_Maze\\_Solving\\_Algorithms\\_Based\\_on\\_Graph\\_Theory](https://www.researchgate.net/publication/331481380_A_Review_of_Various_Maze_Solving_Algorithms_Based_on_Graph_Theory)

[6] Unsupervised representation learning

with Deep Convolutional Generative Adversarial Networks

2015

<https://arxiv.org/pdf/1511.06434>

[7] Unpaired Image-to-Image Translation

using Cycle-Consistent Adversarial Networks

2020

<https://arxiv.org/pdf/1703.10593>

[8] Mazes for Programmers, *Jamis Buck*

Book website

<https://pragprog.com/titles/jbmaze/mazes-for-programmers/>

Personal website

<https://weblog.jamisbuck.org/under-the-hood/>

[9] Euclidian distance

[https://en.wikipedia.org/wiki/Euclidean\\_distance](https://en.wikipedia.org/wiki/Euclidean_distance)

[10] Manhattan distance

[https://en.wikipedia.org/wiki/Taxicab\\_geometry](https://en.wikipedia.org/wiki/Taxicab_geometry)

[11] Chebyshev distance

[https://en.wikipedia.org/wiki/Chebyshev\\_distance](https://en.wikipedia.org/wiki/Chebyshev_distance)

[12] A Fast and Shorter Path Finding Method for  
Maze Images by Image Processing Techniques  
and Graph Theory

by Yoshitaka Murata and Yoshihiro Mitani

<https://www.joig.net/uploadfile/2014/0516/20140516035349809.pdf>

[13] Lee's algorithm

[https://en.wikipedia.org/wiki/Lee\\_algorithm](https://en.wikipedia.org/wiki/Lee_algorithm)

[14] Distributed maze exploration using  
multiple agents and  
optimal goal assignment

by Manousos Linardakis, Iraklis Varlamis,  
Georgios Th. Papadopoulos

<https://arxiv.org/pdf/2405.20232>

[15] Memory-Maze: Scenario Driven Benchmark  
and Visual Language Navigation Model for  
Guiding Blind People

by Masaki Kuribayashi, Kohei Uehara,  
Allan Wang, Daisuke Sato, Simon Chu, Shigeo Morishima

<https://arxiv.org/pdf/2405.07060>

[16] Modular, Hierarchical Machine Learning  
for Sequential Goal Completion

by Nathan R. McDonald

<https://arxiv.org/pdf/2404.19060>

[17] Micromouse competition

<https://en.wikipedia.org/wiki/Micromouse>

<https://micromouseonline.com/>

[18] Development and Evaluation of  
Maze-Like Puzzle Games to Assess Cognitive  
and Motor Function in Aging  
and Neurodegenerative Diseases

<https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7188385/>

[19] A configurable library for generating  
and manipulating maze datasets

<https://arxiv.org/pdf/2309.10498>

[20] The NumPy Library

<https://pypi.org/project/numpy/>

<https://numpy.org/>

[21] The Pillow Libray

<https://pypi.org/project/pillow/>

[22] The Python Imaging Library

[https://en.wikipedia.org/wiki/Python\\_Imaging\\_Library](https://en.wikipedia.org/wiki/Python_Imaging_Library)

[23] Keras Framework

<https://pypi.org/project/keras/>

<https://keras.io/>

[24] The amazed Library

<https://github.com/adipeterca/amazed>

<https://pypi.org/project/amazed/>

[25] (Iterated) Prisoner's Dilemma

[https://en.wikipedia.org/wiki/Prisoner's\\_dilemma](https://en.wikipedia.org/wiki/Prisoner's_dilemma)

[26] When is tit-for-tat unbeatable?

<https://arxiv.org/pdf/1301.5683>

- [27] Properties of Winning Iterated Prisoner's Dilemma Strategies  
<https://arxiv.org/pdf/2001.05911>
- [28] Intelligent tit-for-tat in the iterated prisoner's dilemma game  
<https://arxiv.org/pdf/0807.2105>
- [29] Robert Axelrod  
[https://en.wikipedia.org/wiki/Robert\\_Axelrod](https://en.wikipedia.org/wiki/Robert_Axelrod)
- [30] Simulated Annealing  
[https://en.wikipedia.org/wiki/Simulated\\_annealing](https://en.wikipedia.org/wiki/Simulated_annealing)
- [31] Tit-for-Tat strategy  
[https://en.wikipedia.org/wiki/Tit\\_for\\_tat](https://en.wikipedia.org/wiki/Tit_for_tat)
- [32] Win-Stay-Lose-Switch strategy  
[https://en.wikipedia.org/wiki/Win%E2%80%93stay,\\_%E2%80%93switch](https://en.wikipedia.org/wiki/Win%E2%80%93stay,_%E2%80%93switch)
- [33] Grim-trigger strategy  
[https://en.wikipedia.org/wiki/Grim\\_trigger](https://en.wikipedia.org/wiki/Grim_trigger)
- [34] Meta-Learning an Evolvable Developmental Encoding  
<https://arxiv.org/pdf/2406.09020>
- [35] Individual subject evaluated difficulty of adjustable mazes generated using quantum annealing  
<https://arxiv.org/pdf/2309.04792>
- [36] Mutation Models:  
Learning to Generate Levels by Imitating Evolution  
<https://arxiv.org/pdf/2206.05497>
- [37] Nash Equilibrium  
[https://en.wikipedia.org/wiki/Nash\\_equilibrium](https://en.wikipedia.org/wiki/Nash_equilibrium)



[38] Properties of Winning Iterated

Prisoner's Dilemma Strategies

<https://arxiv.org/abs/2001.05911>

[39] The Evolution of Trust

<https://ncase.me/trust/>

[40] GitHub repository from Axelrod-Python

[https://github.com/Axelrod-Python/Axelrod/blob/dev/axelrod/strategies/axelrod\\_first.py](https://github.com/Axelrod-Python/Axelrod/blob/dev/axelrod/strategies/axelrod_first.py)

[41] Analysis of Maze Generating Algorithms

<https://ipsitransactions.org/journals/papers/tir/2019jan/p5.pdf>