FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Using Evolutionary Algorithms for Navigating through Maze-like Environments
Student:	Jiří Němeček
Supervisor:	Mgr. Miroslav Olšák
Study program:	Informatics
Branch / specialization:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	until the end of summer semester 2021/2022

Instructions

The student will use an evolutionary algorithm, such as the genetic neural network, to create an agent capable of navigating randomly generated maze-like environments.
The finished agent will be able to navigate through previously unseen environments and environments of a different size than it trained on.
Voluntarily, the student can compare with methods using gradient descent.
1) Study relevant materials about the topic
2) Set up a visualization program
3) Create a functional agent using Evolutionary algorithm
4) Evaluate the agent
5) Compare different (mostly neural) architectures

Electronically approved by Ing. Karel Klouda, Ph.D. on 16 September 2020 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Using Evolutionary Algorithms for Navigating through Maze-like Environments

Jiří Němeček

Department of Applied Mathematics
Supervisor: Mgr. Miroslav Olšák, Ph.D.

May 13, 2021

Acknowledgements

I thank my supervisor Mgr. Miroslav Olšák, Ph.D., for his insight and help. I would also like to thank all the great teachers at the Czech Technical University and the others too since they helped me appreciate the great ones. Lastly, I thank everyone who supports me on this journey of life.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 13, 2021 ....

Citation of this thesis

Němeček, Jiří. *Using Evolutionary Algorithms for Navigating through Maze-like Environments.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021. Also available from: ⟨`https://nemecekjiri.cz/BachelorsThesis`⟩.

Abstrakt

Tato práce se zabývá generalizací agenta pro vyhledávání cesty v bludišti, použitím neuronových sítí učených evolucí, nazývaných Genetické Neuronové Sítě (GNN). Práce zkoumá vícero evolučních přístupů a srovnává jejich vlastnosti. Nejlépe fungující GNN jsou poměřeny na složitějších bludištích a jejich výsledky ukazují potenciál v použití GNN pro zpětnovazební učení.

Klíčová slova evoluční algoritmus, evoluční strategie, genetické neuronové sítě, generalizace prohledávacího agenta, navigace v bludišti

Abstract

This thesis attempts to generalize a maze solving agent using Artificial Neural Networks trained by evolution, referred to as Genetic Neural Networks (GNNs). It explores multiple evolutionary approaches, giving their comparative review. The best configurations of GNNs are evaluated on more complex, previously unseen mazes. The results are signaling potential in the use of GNNs for Reinforcement Learning.

Keywords evolutionary algorithm, evolution strategy, genetic neural networks, search agent generalization, maze navigation

Contents

List of Figures

List of Tables

List of Listings

Introduction

The most important element of learning is the ability to generalize. Understanding several examples is good, understanding the underlying concept is better. Ever since the first models of Artificial Intelligence (AI), researchers are trying to create a model that generalizes best.

A simple example of the demonstration of learning is maze solving. Maze solving capabilities can be tested on living mice and artificial agents alike. It is also a great testing environment for the generalization properties since more complex, bigger, or otherwise trickier mazes can always be generated.

This thesis presents a specific approach to maze solving, using a combination of currently prevalent Artificial Neural Networks (ANNs) and Evolutionary Algorithms (EAs) into Genetic Neural Networks (GNNs). It combines them by taking the architecture of an ANN, and the black-box optimization capabilities of EA.

In reinforcement learning — that the maze solving problem is an example of — the question of how to efficiently learn parameters of a model has always been at the center of research. For supervised models, it is essential to have a clear result for every input. However, in the case of reinforcement learning, there often is no clear answer to a given input. EAs, being a black-box optimization technique, do not need labeled data. Thus, it can be helpful in cases like maze solving, where, even though there is one ideal path, with limited information of the agent that navigates through it, its ideal decision cannot always be the same, even when its surroundings (input, if you will) are.

Objectives of the thesis

The thesis aims to explore the capabilities of EAs on the optimization of ANNs applied to the generalization of a maze solver.

The first objective is to explore the theory behind Evolutionary Algorithms and their methods. The applications of EAs and other methods on the maze solving problem will be reviewed. After that, the topic of interest will be the implementation of the environment – the maze and the agent.

Thirdly, a swift discussion of the Visualization program, following with the actual implementation of various Evolutionary techniques. With that, the evaluation method will be explained, various EA configurations and Neural architectures will be tested and their results compared.

These results will also be compared to results created in a separate work, where instead of Neural Networks, Decision Trees were used as a model for the agent's behavior. They were also evolved using some similar evolutionary techniques as presented here.

Chapter 1

Theoretical Background

In the first chapter, the theoretical basis of the topic of this thesis will be explored. Starting with the Evolutionary Algorithms and through Memetic Algorithms (MAs) into Genetic Neural Networks. In the end, some theory on Maze solving will be discussed.

In this chapter, basic knowledge of ANNs and optimization is assumed.

1.1 Evolutionary Algorithms

According to Whitley [1], EAs are tools that simulate evolution to find solutions to search, optimization, or design problems.

There are many types of EAs, generally split into four categories:

- *Genetic Algorithms (GAs)*, initially developed in the 1970s by John H. Holland, are a multi-purpose tool utilizing all tools available. [2, 1] The Evolutionary Algorithm used in this thesis belongs to this category.
- *Evolution Strategies (ESs)* originate from Germany, were created in the 1960s. They prefer mutation to crossover and are more used in optimization. [1, 3] In this thesis, a popular algorithm Covariance Matrix Adaptation – Evolution Strategy (CMA-ES) will be used and its performance compared to the specific variant of GA.
- *Evolutionary Programming (EP)*, created in the 1960s by Lawrence J. Fogel, focused on evolving finite-state machines. [4]
- *Genetic Programming (GP)* was created in the 1990s and focuses mainly on evolving computer programs. [5]

Table 1.1: Common vocabulary for Evolutionary Algorithms

Term	Explanation
Gene	An element of genotype
Genotype	Individual's representation that is mutated
Phenotype	Translation of a genotype that can be evaluated
Individual	Carrier of genetic information, has a fitness value
Fitness	Evaluation of an individual, selection is based on fitness
Population	Set of individuals

1.1.1 Vocabulary

Before getting into EAs it is necessary to explain part of the common vocabulary. Using Beyer's glossary [6] a selection of a few core terms is presented in Table 1.1.

1.1.2 The generic algorithm

EAs are, at their core, quite simple algorithms, as seen in Figure 1.1. They consist of 6 main components:

- *Initialization* sets the initial population of μ individuals, μ being a hyperparameter of the algorithm. [4]
- *Evaluation* requires a fitness function. This function takes an individual and returns an orderable value so that the individuals can be sorted. In this step, all individuals are assigned their respective values. [4]
- *Selection* represents natural selection based on the fitness values of the individuals and implementation of the selection operator. It selects κ individuals from the μ in the population. κ is a hyperparameter, not constrained by μ since an individual can be selected multiple times. [4]
- In *crossover* κ selected individuals are combined between themselves, creating λ offspring. In each step, n individuals are combined to create m offspring. Typically $n = m = 2$. [4]
- *Mutation* operator makes a small random change in the genotype, according to a given probability distribution. It does not change the number of individuals. [6]
- *Replacement* happens after λ individuals are finally complete and evaluated. For the main population to not grow in size, replacement decides which individuals will advance into the next generation, choosing μ out of $\mu + \lambda$ individuals. [4]

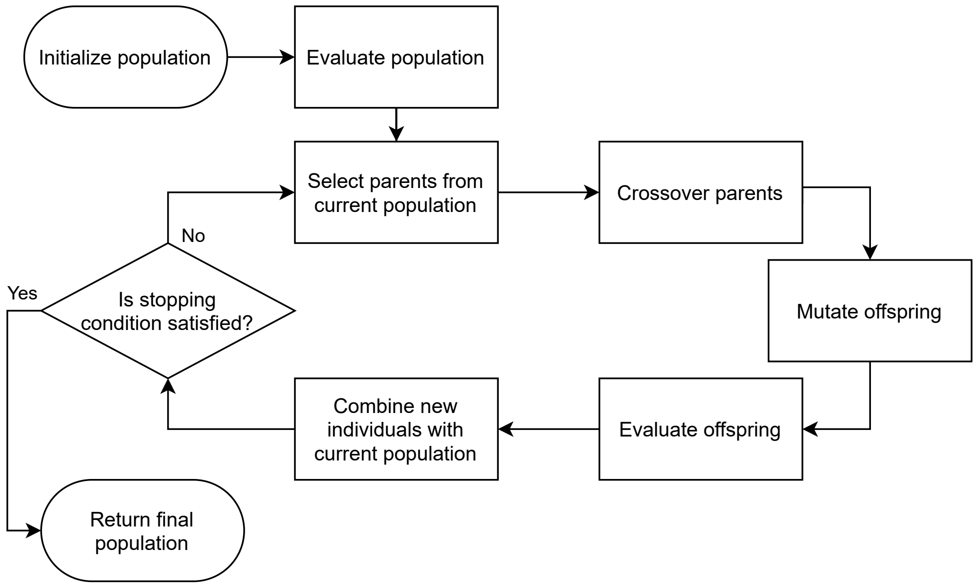


Figure 1.1: Diagram of a generic evolutionary algorithm

1.1.2.1 Initialization

In initialization, the first — initial — population of individuals is created. This step is very problem-dependent and can significantly influence the performance of the final population. If there is prior knowledge about the problem, the initialization can be adapted to reflect that knowledge and to direct the performance towards an optimum. If there is no such knowledge, the initialization is done randomly to cover as much of the search space as possible to boost exploration. [4]

1.1.2.2 Evaluation

Evaluation is even more problem-dependent. It must contain translation from genotype to phenotype and a fitness function to assign the individual a fitness value. If looking for a minimum of a function f, a likely choice is a genotype as a vector of input variables x where each variable is a gene, phenotype mapping is identity, and the fitness value of an individual x is just $f(x)$. Then in the selection, lower values of fitness are considered better.

1.1.2.3 Selection

In selection, better individuals, according to their assigned fitness, are selected more often than those with worse values. A prime example of a fitness proportional selection operator is Stochastic Universal Sampling (SUS). It samples

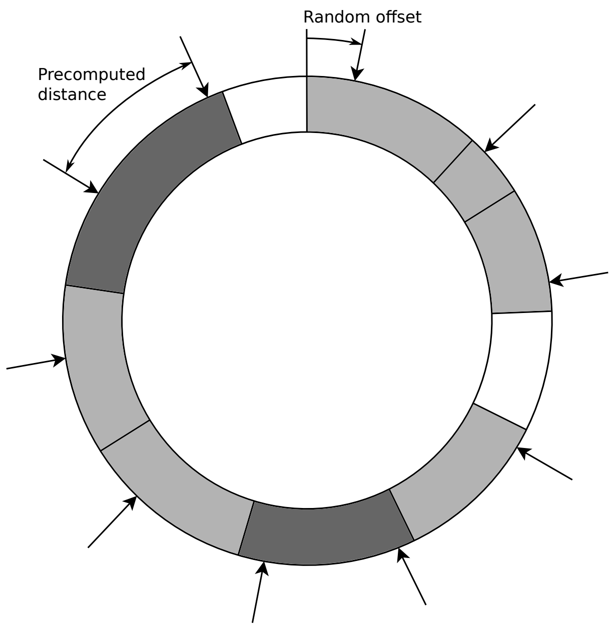


Figure 1.2: Stochastic Universal Sampling diagram. All $\mu = 10$ individuals take up space proportional to their fitness, random offset ensures stochastic selection, and $\kappa = 10$ equidistant pointers along the wheel ensure diversity of selected individuals. The darkest individuals are selected twice, while the lightest are not selected at all. Note that the individual with the worst fitness value was also selected.

individuals with a probability proportional to their fitness. The exact way this is done can be explained with an analogy of a roulette wheel, visualized in Figure 1.2. All individuals take up space along the wheel, proportional to their fitness value (supposing higher fitness is better). Then κ equidistant points around the wheel are sampled by randomly offsetting the first one. Each of the points selects the individual it points at. [4]

A common issue with fitness-proportional selection is that if one individual has much better fitness than all other individuals, the selection operator will choose the best individual multiple times. This effectively destroys diversity in the population, which leads to premature convergence to a sub-optimal solution.

There are ways of mitigating premature convergence, for example, the method of fitness sharing, where individuals with similar genotypes share their fitness and thus have a lower chance of getting all into the next generation. The fitness is recomputed using an equation

$$f_{sh}(i) = \frac{f(i)}{\sum_{j \in I} sh(d(i,j))}$$

where $f(i)$ is the original fitness value of an individual i, I is the set of all individuals, $d(i,j)$ is the distance between individuals i and j, and $sh(d)$ is a sharing function defined as

$$sh(d) = \begin{cases} 1 - (\frac{d}{\sigma_{sh}})^{\alpha_{sh}} & \text{if } d < \sigma_{sh} \\ 0 & \text{otherwise} \end{cases}$$

where σ_{sh} is a parameter called niche radius. It defines a radius of the neighborhood around an individual. The parameter α_{sh} can be used to decrease the final shared fitness, but it is often chosen as 1. [7]

Another way to approach this issue is that instead of changing the fitness values themselves, they are used to rank the individuals and these ranks are used for selection. Or the fitness can be used only to compare random individuals, like in Tournament selection. Tournament selection takes $k > 2$ random individuals from the population and selects the one with the highest fitness value. It repeats this process κ times. It is used often because it is faster and simpler to implement than proportional methods and has similar properties to methods using ranking. [4]

1.1.2.4 Crossover

Crossover is a process of taking at least two parents, combining their genotype, and creating offspring. Not all selected individuals need to become parents. Based on crossover probability or proportion, some individuals from the population become parents. The crossover process is typically stochastic, so two parents can combine into many different offspring. However, if the parents are identical, they will always create identical offspring to them.

It can happen that two good individuals will not generate as good offspring. Imagine an example with bimodal function, each parent at one local optimum and generated offspring between them, in the valley. This individual has a minor chance of surviving replacement or will have low odds of being selected in the next generation since its fitness is significantly lower.

This issue is addressed by modifying the crossover operator to generate offspring more similar to either of the parents. Or by restricting the parent selection to only similar parents so that the distance between them is less likely to contain a valley.

Basic examples of crossover operators for bit vector genotypes are one- and two-point crossover. A point or two points on the vector are randomly selected, and the parents cut into 2 or 3 parts. One part (the middle part in the case of two-point crossover) is then exchanged between them, creating two offspring. Both crossover operators are described in Figure 1.3. N-point crossover is a generalization of this principle. It is a simple and efficient crossover, especially if there are relations between neighboring genes. [4]

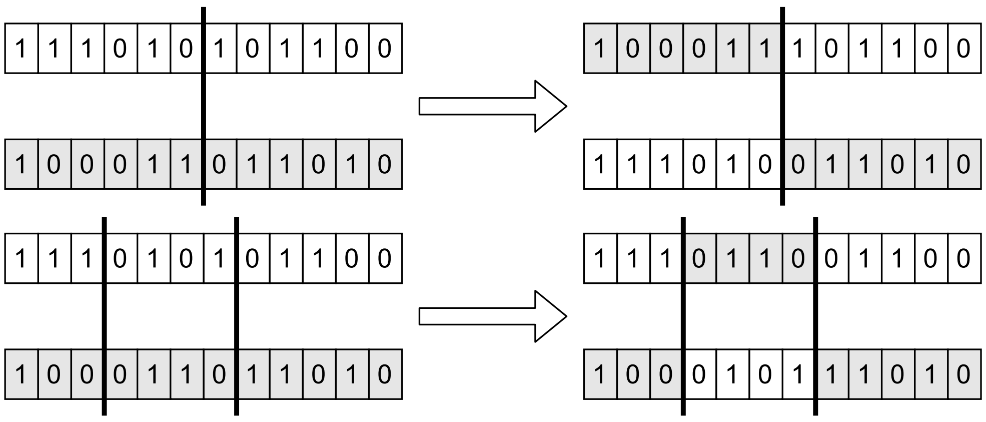


Figure 1.3: Bit vector crossover: on the top, one-point crossover; on the bottom two-point crossover, the middle part between the selected points is swapped. Different colors are there only to help identify the origin of a part of the individual.

1.1.2.5 Mutation

The mutation operator introduces a small random change to the individual. Just as the crossover, the proportion of mutated individuals or probability of mutation is set as a hyperparameter.

A simple example of mutation on a binary vector would be a bit-flip of a bit at a randomly selected position, see Figure 1.4. [4]

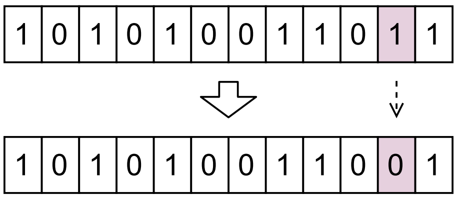


Figure 1.4: Bit-flip mutation: second to last bit in the vector was randomly selected and changed from 1 to 0. The grey color only signals the gene that was mutated.

1.1.2.6 Replacement

Replacement decides which individuals will advance to the next generation. There are also many ways to do this, the simplest being Generational replacement, which discards the entire old population and replaces it with the new one. There is a potential problem with this since it does not necessarily keep the best individual in the population.

The notion of keeping the best individual in the population for the next generation is called *Elitism*. A replacement method is elitist if it keeps the best individual in the population. Elitist strategies may significantly speed up the performance, especially in functions like convex functions, however in more difficult state spaces of, for example, multi-modal functions, non-elitist strategies can prove advantageous. [4]

1.1.3 Niching

Niching is a process in which the locality of a genotype in the search space is taken into account. It helps preserve the diversity of individuals in the population. [6] In niching, if two individuals have similar genotypes according to some relevant distance measure, they are less likely to both advance to the next generation. Niching can be done in many ways, for example, by fitness sharing, mentioned in Section 1.1.2.3. In this thesis, niching was implemented in the replacement operator.

1.1.4 Covariance Matrix Adaptation (CMA-ES)

As the name suggests, the CMA-ES is an Evolution Strategy that uses a multivariate normal distribution, defined by mean vector and covariance matrix. The algorithm keeps only the mean vector and covariance matrix, but each generation, μ individuals are sampled from the mutation distribution. The individuals are evaluated, and the mutation distribution is then adapted based on their fitness. [8]

Akimoto [9] shows that CMA-ES can be considered equivalent to a method that uses natural gradient. Thus it makes sense to use the CMA-ES to optimize weights of a Neural Network while not diverging from gradient descent. The use of CMA-ES for ANN optimization has been done before, for example, in online scheduling problems by Branke [10].

1.2 Memetic Algorithms

The term Memetic Algorithm was invented by Moscato, who defined it as *"a marriage between a population based global search and the heuristic local search made by each of the individuals"*. [11]

He argued that while Genetic Algorithms use the analogy of genetic information and genes are not the only determining factor of an individual's quality, an extension of GAs is required to achieve a more accurate simulation of what happens in nature. The memes, from which the Memetic Algorithms got their name, can be loosely explained as experiences, knowledge, or skill achieved by a person in their life. Moscato uses an analogy with martial arts. No one is born a kung-fu master. It is not in their genetic information, yet it helps the individual in the evolution process. [11]

Generally speaking, MAs are an extension of GAs that at a certain point in evolution use local search methods to improve the individuals. It can happen right after initialization, so the EA starts from a pre-optimized position, or during the evolution in every generation (or both). Then it is done typically before replacement so that every individual goes through the local search last. [12]

Cotta states in his book that the field of MA is very active, generating over 300 articles per year on average. Although the algorithms used nowadays are more complex than the basic MAs described here, some of them implementing self-adaptation, the concept of combining GAs with local optimizers works well. [12]

1.3 Genetic Neural Networks

Genetic Neural Network is a term used for an ANN that utilizes an EA to train its weights. Sometimes, a mix of evolution and gradient descent is used. [13] Typically, evolution is used at first to find a globally good solution in the search space, followed by short gradient descent to sharpen the result. Since Stochastic Gradient Descent (SGD) is a local search method, GNN optimization can be considered a Memetic Algorithm.

This approach to optimizing ANNs is motivated by one of their main problems, *overfitting*. This problem of getting into a local optimum can be addressed using a global optimization algorithm, such as GA. [14, 15] The use of GAs instead of gradient descent with backpropagation to train ANNs has been thoroughly studied since the 1990s. It has been shown that GA can outperform these methods in some cases. [16, 15] Porto and Fogel showed that gradient descent leads to sub-optimal solutions more often than stochastic optimization techniques, GA being one of the tested examples of a stochastic optimizer. [16]

1.3.1 Evolutionary operators

To implement a GNN, we need to specify the genotype of an ANN and some evolutionary operators that depend on it. Namely crossover and mutation operators.

The genotype can be represented as a list of real values which enables standard n-point crossover and a simple mutation, for example, by adding gaussian noise to the list or only to a few selected elements. [14]

Montana and Davis presented more potential operators, for example, mutating weights belonging to specific nodes or a crossover that swaps all weights of randomly chosen nodes instead of just some random weights. [14]

Interesting mutation and crossover operators do not consider only the structure of a genotype. They function in an informed way, augmenting the randomness. For example, *Mutate-weakest nodes* operator, presented by Montana and Davis, finds nodes with less significant influence on the result. These nodes are considered weak. Such a node can be found by executing the network on the same input with and without the node in the network. [14]

1.3.2 Use of CMA-ES

Even though the name Genetic Neural Networks suggests the use of GAs, we can utilize a different method of evolutionary computation, such as CMA-ES. Since the CMA-ES is equivalent to a method using natural gradient [9], its use in GNN is logical.

1.4 Maze Solving

Maze solving is a common task presented to computer agents [17] and biological organisms [18] alike. It is very suitable for studying the generalization capabilities of a given problem solver.

1.4.1 Common techniques

Firstly, let's look at common techniques used by maze solvers. In Graph theory, there are standard algorithms like Breadth- or Depth-first search, used for finding a path in a graph (maze). They, however, have non-constant memory requirements. A standard maze solving algorithm that is stateless is a *Wall follower algorithm.* [19]

Wall Follower (WF) is, as the name suggests, an agent that tracks either left- or right-hand wall. It is best described by a decision diagram, see Figure 1.5. A downside of the algorithm is that if there is no connecting wall from start to goal, the algorithm will never stop and never find the path. [20]

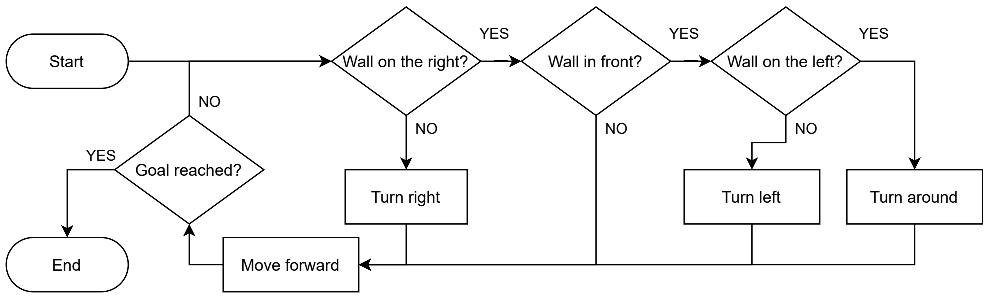


Figure 1.5: WF algorithm diagram: the right-hand version of the algorithm.

Nevertheless, it is a classical algorithm, sophisticated enough to make it a valuable baseline for intelligent behavior. In this thesis, only mazes solvable with WF algorithm are used. Regardless of which side the agent follows.

1.4.2 Use of Evolutionary algorithms

Nature, with its properties and processes, is often the initiator of scientific discovery. Evolution, being one of the greatest natural processes in the universe, is viewed as having great potential in achieving a level of AI equivalent to that of humans. [17]

Shorten and Nitschke have used Neuro-Evolution for this task. Neuro-Evolution is an extension of GNNs, where the architecture of a GNN is also optimized using an evolutionary algorithm. It is closer to what happens in a real brain than a basic ANN. [17]

Similarly, in this thesis, EAs were used to train Neural Networks to study generalization capabilities. However, the structure of a GNN is rigid, and only the weights are optimized. Their definitions of agent and maze vary significantly from those in this thesis. The agent has different inputs, mazes are vaster, and there are usually fewer of them in the evolution. The fitness function is also different, making the experiments of this thesis hardly comparable to the work of Shorten and Nitschke [17].

Chapter 2

Implementation

In this chapter, the specific implementation will be described. That is the specification of the task itself, the implementation of the maze solving problem, Graphical User Interface (GUI), the GNN models, and the EA methods used to optimize them. All of which were used to generate the results discussed in Chapter 3.

2.1 Task

The overall task to solve a maze sounds simple, but to be thorough, it is necessary to specify all components. There is a maze — an environment, if you will — and an agent.

2.1.1 Maze

For the purpose of this thesis, a maze is a rectangular two-dimensional grid made of tiles. Every tile (except bordering tiles) has four neighboring tiles. Some bordering tiles are separated by a wall. Connected tiles can be represented as a tree, meaning that each tile is connected to every other tile in the maze by exactly one path. It has finite width and height. For training — unless specified otherwise — mazes of size 10 by 10 tiles were used. There is always a starting tile and a goal tile, both chosen randomly from the four corners of the maze.

Maze generation

There are many maze-generating algorithms. Practically every algorithm constructing a minimum spanning tree — or a tree traversing algorithm — on a fully connected grid graph creates a maze that complies with the definition provided above. The only thing required is to introduce randomness to it, and a random maze generator is created.

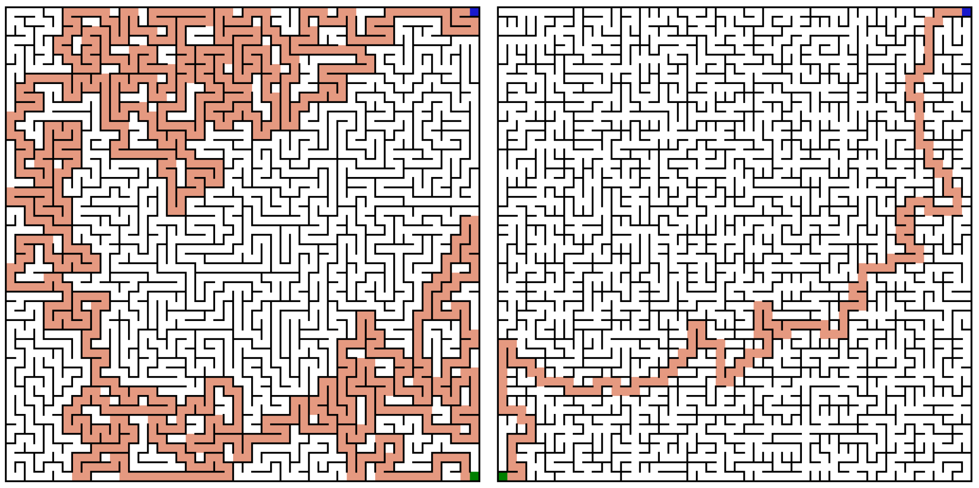


Figure 2.1: Maze comparison: on the left, there is a maze created with DFS; on the right, a maze created with Kruskal's algorithm. Even though DFS maze might look more difficult at first, notice that along the path, there are not as many points where the agent can make a wrong turn.

In this thesis, two maze-generating algorithms were implemented:

- *Kruskal's algorithm* starts with all walls in place and randomly selects two neighboring tiles that are not connected together (there is not any connecting path). It removes the wall in-between them and continues by selecting another pair of such tiles until all tiles are connected. It creates very good mazes, see the comparison in Figure 2.1.
- *Depth-First Search (DFS) algorithm* uses DFS to traverse the fully connected grid from a starting tile until all tiles are visited exactly once. In each step, a random neighboring unvisited tile is selected to ensure the creation of random mazes. All unused edges of the fully connected graph become walls. The shortest path is around twice as long compared to the Kruskal generated mazes, but in experiments, DFS mazes proved to be also significantly easier to solve.

2.1.2 Agent

The agent has some observations of its surroundings and actions used to move through the environment. The agent also has a direction it is facing, so all observations change not only with the change of the agent's position but also with the change of the agent's direction. The agent starts on the starting tile facing upwards.

The possible observations of the agent's surrounding environment are:

- *Relative Angle* from the direction of the agent to the direction towards the goal tile. This has values in the interval $(-180, 180]$, 0 being the direction forward. Negative values mean that the goal tile is to the left.
- *Distance to the closest wall* in all four directions relative to the way the agent is facing.
- *Number of visits* the agent has made to each of the four neighboring tiles. This information is used only if the model has no internal state to remember this information.

And the possible actions of the agent are:

- *Turn right.* The agent stays at the same position but rotates clockwise.
- *Turn left.* The agent stays at the same position but rotates counter-clockwise.
- *Move one tile forward.* If the agent is facing a wall, the action is consumed, and the agent stays at the same position, facing in the same direction.

2.2 Graphical User Interface

For the implementation of GUI, a simple to use, minimalist library was sought. Ideally with a canvas to visualize a maze and basic form input elements to enable configuration within the application.

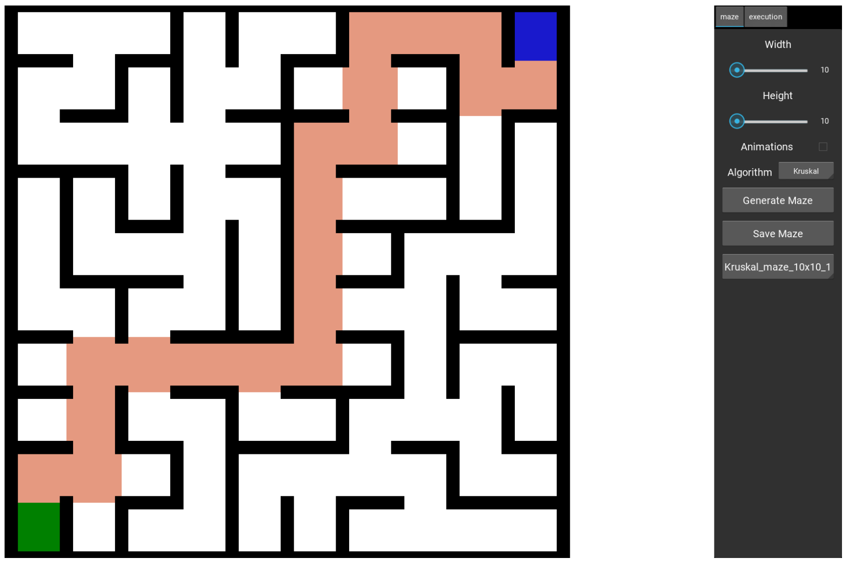
After attempts to work with Tkinter, PyQt, and PySimpleGUI, the decision was to go with Kivy [21], as it had the most intuitive structure, enabled division of logic and visual layout definition with special configuration files, and had a fresh look.

2.2.1 Maze page

The main thing to view is the maze. That is why all interaction was implemented into a small side panel to the right from the canvas on which the maze is shown. The layout can be seen in Figure 2.2.

There are two tabs: one handles mazes, their creation, saving, and loading, and the other handles execution. That means loading learned models and simulating their attempts at solving the maze.

In the maze tab's interface, two sliders determine the width and height of a generated maze. The generation process can be animated using a checkbox. This process needs to run in a separate thread. Otherwise, the library cannot reload the GUI while the walls are being removed from the maze.



Width 10
Height 10
Animations
Algorithm Kruskal
Kruskal_maze_10x10_1

Figure 2.2: GUI maze layout, with one of the training mazes loaded

The generated maze can be saved into a pre-defined folder, from which it can also be retrieved using the load selection tool. This way, the user can save specific mazes for later evolution.

2.2.2 Visualization of attempts

For the visualization of the agent's attempts at solving the maze, another tab was implemented, see Figure 2.3, where it is possible to choose a neural network, step through or animate its behavior, or test the maze solving skills of the user with a simple manual operation.

If the manual operation is enabled using the topmost checkbox, the user can navigate the maze using arrows. Arrow up is a forward signal for the agent. Arrows left and right control the turning of the agent.

2.3 Neural Networks

In this thesis, only the weight optimization of ANN was explored and not the evolution of the neural architecture itself. That is why four different neural network structures were used. Two of which are standard feed-forward networks, other two are recurrent. They were designed as comparable as possible. That is why two are small, with 55 parameters and two are bigger, with around 90 parameters (89 parameters in the feed-forward net, 91 in the recurrent).

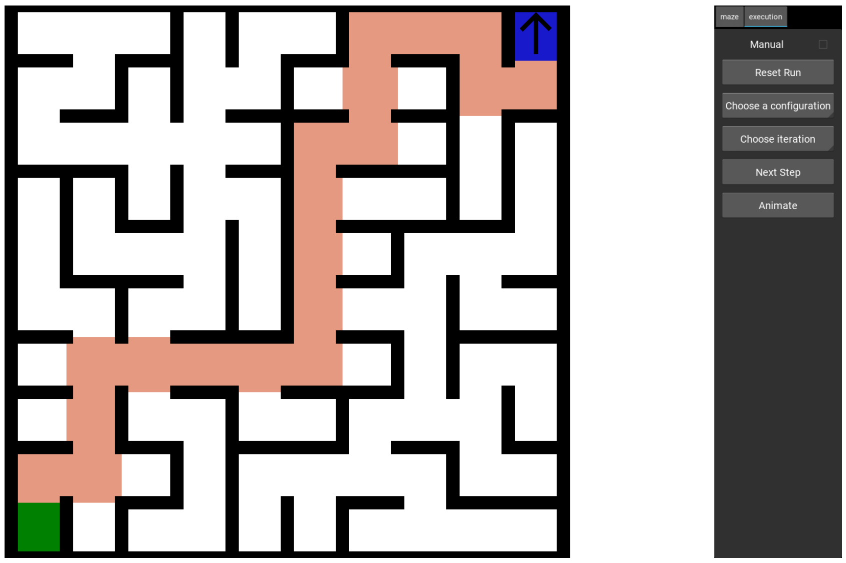


Figure 2.3: GUI execution layout; the agent is shown as an arrow on the starting tile.

Both architectures are implemented using only NumPy [22]. A specialized library, such as TensorFlow or PyTorch is not used since the gradient computation was not needed, except for Memetic Algorithm, as will be discussed further in Section 2.5.

For all hidden layers, ReLU was used as an activation function. Softmax was used for the output layer, as is a standard in classification (the problem can be viewed as a classification of possible situations).

2.3.1 Feed-Forward

For feed-forward architectures, 9 input neurons must be considered instead of 5 because of the information about visits to neighboring tiles. Thus the small version has 4 hidden neurons in one layer, and the bigger one has 5 in the first hidden layer and 4 in the second layer.

2.3.2 Recurrent

Recurrent architecture has fewer input neurons (5), but parameters concerning the recurrence of layers in the equation must be considered, so one hidden layer of 4 neurons creates the small architecture, and two layers of 4 neurons compose the bigger one. The computation of one hidden layer can be described by the equation

$$s_{curr} = \phi(v_{in} \cdot W_{in} + s_{prev} \cdot W_{state} + b)$$

where ϕ is an activation function, v_{in} is a row vector of input to the layer, s_{prev} is a row vector of the output of this layer in the previous time step, W_{in} and W_{state} are weight matrices of appropriate dimensions, for input and state vector respectively, and b is a bias vector. In the next time step, s_{prev} will have the value of s_{curr} in this time step. The output layer does not have a state and is computed in the same way as in feed-forward ANN. The initial state vector is equal to a zero vector. It is reset to zero vector before every new maze simulation.

The recurrent architecture was encountering problems with the state variables exploding towards infinity, so in later tests, an attempt was made to mitigate this using hyperbolic tangent as the activation function. However, it did not have a significant influence on the performance of Recurrent Neural Networks (RNNs).

2.3.3 Normalization

Based on a few initial tests, the normalization of the input values was implemented since the angle was spanning from -180 to 180 while other inputs barely exceeded the value 5. Furthermore, it was unclear how to set a value representing that on some neighboring tile, the number of visits is impossible to know since there is a wall separating the agent from it.

Angle was rescaled linearly — dividing by 180 — to the range $(-1, 1]$. Distances from a wall are mapped into the $[0, 1)$ interval using a function

$$\texttt{transform\_wall\_distance}(k) = 1 - 0.5^k$$

which for $k = (0, 1, 2, \ldots)$ generates a sequence $(0, 0.5, 0.75, 0.85, 0.9375, \ldots)$. Numbers of visits are mapped into the $[0, 1]$ interval using a function

$$\texttt{transform\_visits}(k) = 1 - 0.6^k$$

which for $k = (0, 1, 2, \ldots)$ generates a sequence $(0, 0.4, 0.64, 0.784, 0.8704, \ldots)$ that has a limit 1 for $k \to +\infty$ which makes a good upper bound to the case of unknown value when the knowledge is obstructed by a wall.

The reason for a different constant in the cases of wall distance and number of visits is that the difference between 3 or 4 visits is important to be noticeable since the agent can get to the same spot 3 or 4 times. However, in the case of wall distance, the difference between 3 and 4 is almost irrelevant. There, the difference between 0 and 1 is extremely important since it is the basis of understanding whether there is space to go.

2.4 Evolutionary Algorithm

The main EA have been implemented from scratch because it enabled the most modifiability. The code is not optimized for speed but rather for clarity.

All source codes are provided on an enclosed medium. Like Neural Networks, it uses NumPy library [22] for mathematical operations. In code listings, it is assumed to be imported as `np` as is common for the library. A common parameter in all Listings is `rng`. It is a random number generator, created by a function `np.random.default_rng`.

The whole implementation is object-oriented. Every operator implements a pre-defined method that the main algorithm calls. This enables modularity since, to implement a new operator, only the defined method needs to be implemented. As can be seen in Listing 2.1 of helper method to the mutation process, the instance of a class `Evolution` has a `mutation` operator it was given in the constructor. It only requires the `mutate` method. This logic is implemented similarly for all evolutionary operators.

```python
def mutate(self, children):
    for c in children.get_nets():
        if self.rng.uniform(0, 1) <= self.mutation_prob:
            # mutation is in-place
            self.mutation.mutate(c, self.rng)
    return children
```

Listing 2.1: Evolutionary algorithm core: Mutation helper method

2.4.1 Population

Population class contains basic logic concerning the population of individuals. It stores the networks, their fitness value, their distance used for niching, which will be discussed further in Section 2.4.7.1. The index of the best performing individual is remembered since it is often required for comparisons.

2.4.2 Initialization

Initialization operator class takes the number of hidden neurons in layers described as a list. A list `[6,4]` translates to 2 hidden layers, first with 6 neurons and second with 4 neurons. The final architecture then depends on the second argument, the type of the network. This argument can be either of `"recurrent"` or `"basic"` meaning recurrent or feed-forward architecture respectively. It also determines the number of neurons in the input layer. That is 9 or 5 respectively, as explained in Section 2.3. The output layer always consists of 3 neurons.

The neural nets are then randomly generated, their default weights and biases are sampled from a multivariate normal distribution. The parameters of this distribution were found using the weight distribution of resulting nets from a few initial tests. The parameters are shown in Table 2.1.

Table 2.1: Parameters of normal distribution used to initialize the network architectures.

	mean value	standard deviation
Recurrent	0	5
Feed-forward	0.5	1

The modularity of the implementation was useful when an initialization using precomputed nets was required in the MA discussed further in Section 2.5.

2.4.3 Fitness Evaluation

As a fitness function, a simulation of an attempt to solve a maze was selected. In experiments, evaluation on the same set of mazes and evaluation on a different set in each generation was tested. The changing maze sets generate issues in comparability, require multiple evaluations for each individual, and lead to non-monotone convergence to an optimum. The fact that it converges nonetheless shows an interesting property of EAs and will be discussed in the Experimental part of this thesis.

The `Evaluator` class takes following parameters:

- `n_mazes` sets the number of mazes an individual will be evaluated on;
- `max_actions` sets the maximum allowed actions, the maximal amount of time steps;
- `w` and `h` are integers, defining the width and height of the mazes in case mazes are to be generated;
- `generator` is one of `"kruskal"`, `"DFS"` or `"random"`, it sets the generator in case the mazes are to be generated or makes a random choice for every maze;
- `normalize` is a boolean value, signaling if the input values should be normalized or not;
- `folder` is a string with a path used to load a specific set of mazes stored in a folder defined by the path. There should be no other files, else the program crashes.

The choice of evaluating only based on success or failure is too simplistic and needs a lot of mazes for evaluation to get some variability in the results. Thus in this thesis, a more delicate measure of success was used. It can be described using the following equation:

$$f(i) = \frac{\sum_{m \in M} d_m(i) + t_m(i)/t\_tot_m(i)}{|M|}$$

```
 1 def evaluate(self, net, rng, generate_mazes=False):
 2     if generate_mazes:
 3         self.generate_new_mazes(rng)
 4     strat = NNStrat(net, self.normalize)
 5     results = []
 6     for maze in self.mazes:
 7         maze.refresh_maze()
 8         strat.restart()
 9         agent = Agent(maze)
10         shortest_dist = maze.distance_from(agent.pos)
11         shortest_i = 0
12         last_pos = agent.pos
13         stuck_t = 0
14         tot_actions = self.max_actions
15         for i in range(self.max_actions):
16             next_action = strat.next_action(agent)
17             agent.apply_action(next_action)
18             if next_action == Action.GO:
19                 d = maze.distance_from(agent.pos)
20                 if d < shortest_dist:
21                     shortest_dist = d
22                     shortest_i = i
23                     if d == 0:
24                         break
25             if agent.pos != last_pos:
26                 last_pos = agent.pos
27                 stuck_t = 0
28             else:
29                 stuck_t += 1
30                 if stuck_t > 4:
31                     tot_actions = i+1
32                     break
33         results.append(shortest_dist + (shortest_i / tot_actions))
34     return sum(results) / len(self.mazes)
```

Listing 2.2: Fitness evaluation method

where M is a set of all mazes to simulate on, $d_m(i)$ is the closest distance from the goal tile reached by individual i in maze m, defined as the number of tiles between the agent and the goal. $t_m(i)$ is the time step at which the closest tile was reached, and $t\_tot_m(i)$ is the number of time steps executed.

This computation is implemented in a method `evaluate` shown in Listing 2.2. The first thing to notice is a `strat` variable on line 4. It is an instance of a class `NNStrat`. It handles the neural network, normalizes inputs if applicable, and translates the results of a forward pass through ANN into a respective `Action`.

On line 6 the loop over all mazes is defined. Each iteration, the maze is reset, which is important for feed-forward architecture when the maze remembers the number of visits to tiles. Then the strategy is reset. This is where RNN resets its state to 0. An agent gets created on line 9.

Variables `shortest_dist` and `shortest_i` contain values described in the mathematical equation as $d_m(i)$ and $t_m(i)$ respectively. Variable `tot_actions` is $t\_tot_m(i)$. It is changed only when the execution is prematurely finished, thanks to a simple stuck state detection. It does not change if the agent reached the end because the time when the goal was reached is also a measure worth optimizing.

The stuck state detection is rather basic. It detects if the agent has not moved for 5 time steps. In that case, the execution finishes with a lower than the maximal amount of time steps because higher fitness values are considered worse. And getting stuck is clearly worse than moving constantly. The position is checked even if the action was to move forward since the agent might walk into a wall. In that case, its position stays the same.

The feed-forward ANNs cannot get out of the stuck state because in every position, there are only 4 possible configurations of inputs, and they do not change. For RNN however, this does not hold because their inner state changes each time step and depends on the previous one. That is why the limit was set at a conservative 5 time steps. From initial testing was deduced that this stuck state detection influenced only the runtime speed and not the quality of solutions.

The results on mazes are stored in a list `results` and in the return statement, their average is computed exactly as in the mathematical statement.

2.4.4 Selection

Multiple selection operators were implemented but based on previous experience, only the Stochastic Universal Sampling was used as a well-performing selection operator.

Its implementation, seen in Listing 2.3, takes as arguments a list of fitness values and a number of individuals to select. It returns indices from the list of fitness values that were selected.

The `select` method contains 3 main blocks. In the first block (lines 3-7), the fitness values are transformed using a function

$$\texttt{transform\_fitness}(f) = 1.1 \cdot f_{max} - f$$

where f_{max} is a maximal value of f in the population. It reverses the order of the fitness values since the lowest original value should take most of the circle's circumference. The maximal value is increased by a tenth else the worst solution would have no chance of being selected. After that, the values are rescaled to sum up to 1.

```
 1  class StochasticUniversalSampling:
 2      def select(self, fitness, result_n, rng):
 3          # reverse the fitness values (lower fitness is better)
 4          max_f = np.max(fitness)
 5          f_rev = -fitness + (max_f*1.1)
 6          sum_f = np.sum(f_rev)
 7          elem_angles = f_rev / sum_f
 8
 9          angle_d = 1/result_n
10          angle = rng.random() * angle_d
11
12          chosen_indices = []
13          i = 0
14          elem_angle = 0
15          for _ in range(result_n):
16              # find appropriate individual to select
17              while elem_angle+elem_angles[i] < angle:
18                  elem_angle += elem_angles[i]
19                  i += 1
20
21              chosen_indices.append(i)
22              angle += angle_d
23          return np.array(chosen_indices)
```

Listing 2.3: Stochastic Universal Sampling implementation

In the second block (lines 9-10), a starting angle is sampled from a uniform random distribution, and an angle delta is computed as $1/n$ where n is the number of indices to select.

In the last block (lines 12-22), the angles are iterated over by adding the angle delta to the starting angle. For each angle, the index of the appropriate fitness corresponding to that angle is added to the list. This is done with parallel iteration over the rescaled fitness values, normalized, to sum up to 1, just like the maximal angle iterated in the `for` loop.

2.4.5 Crossover

The expected crossover operator has a method `crossover` which takes 2 parents and returns 2 offspring as a tuple. It must not change the parents, as they are expected to be intact.

Two crossover operators were implemented. `LayerCrossover` takes a random layer and swaps all weights and biases of that layer. The exact implementation is shown in Listing 2.4. `NeuronCrossover`, as the name suggests, takes a random neuron out of all neurons and swaps its input weights and bias. The implementation is very similar to that of `LayerCrossover`.

```python
def crossover(self, p1, p2, rng):
    c1, c2 = deepcopy(p1), deepcopy(p2)

    layer_i = rng.integers(p1.layers)
    c1.w[layer_i], c2.w[layer_i] = c2.w[layer_i], c1.w[layer_i]
    c1.b[layer_i], c2.b[layer_i] = c2.b[layer_i], c1.b[layer_i]
    if c1.nn_type == "recurrent" and layer_i != p1.layers-1:
        c1.sw[layer_i], c2.sw[layer_i] = c2.sw[layer_i], c1.sw[layer_i]

    return c1, c2
```

Listing 2.4: Layer crossover implementation

2.4.6 Mutation

A mutation operator has to contain a method `mutate` that takes one network as a parameter. Unlike crossover, a mutation operator is made to mutate the network instance itself and does not return any value.

There are 3 implemented and tested mutation operators. The smallest changes are done by the `OneNeuronMutation`, the biggest `LayerMutation` and as a compromise between them is `MultiNeuronMutation`. A common parameter of all of them is the standard deviation (`sd`), which sets the scale of the normal distribution that is sampled and added to the mutated weights.

The first two mutation operators are simple. A node or a layer is randomly selected, and then a vector or matrix sampled from the defined multivariate normal distribution is added to it.

The `MultiNeuronMutation` has an extra parameter, `proportion`. It is a decimal number between 0 and 1. It sets the proportion of all neurons that are to be mutated. The exact number of nodes to mutate is determined by the equation

$$n_{select} = \lfloor n_{neurons} \cdot \texttt{proportion} \rfloor$$

where $n_{neurons}$ is the total number of hidden and output neurons (all modifiable neurons). The exact implementation is shown in Listing 2.5.

In `mutate` method, on lines 7 to 10, all neurons' coordinates — index of layer and index of the neuron in that layer — are gathered into a list. It is then shuffled, and a proportion of them is selected for mutation. The number of nodes to mutate is computed, as specified above, on line 13.

The `__mutate_one` method is implemented in the same way as `mutate` method of `OneNodeMutation`, but the coordinates are given as parameters, instead of being randomly selected.

```python
class MultiNeuronMutation():
    def __init__(self, proportion, sd):
        self.sd = sd
        self.proportion = proportion

    def mutate(self, net, rng):
        all_coords = []
        for l in range(net.layers):
            for n in range(net.b[l].shape[0]):
                all_coords.append((l, n))

        rng.shuffle(all_coords)
        select_n = int(len(all_coords) * self.proportion)
        picked = all_coords[:select_n]
        for l, n in picked:
            self.__mutate_one(net, l, n, rng)

    def __mutate_one(self, net, layer_i, neuron_i, rng):
        net.w[layer_i][:, neuron_i] += rng.normal(scale=self.sd,
                size=net.w[layer_i][:, neuron_i].shape)
        net.b[layer_i][neuron_i] += rng.normal(scale=self.sd)
        # if the network is recurrent and selected layer is not output layer
        if net.nn_type == "recurrent" and layer_i != net.layers-1:
            net.sw[layer_i][:, neuron_i] += rng.normal(scale=self.sd,
                    size=net.sw[layer_i][:, neuron_i].shape)
```

Listing 2.5: Multi neuron mutation operator implementation

2.4.7 Replacement

Replacement operators require a method `select` which takes two populations, old and new one, and returns a population of the same size as the old one, created by selecting some individuals from both of them.

Multiple replacement operators were implemented, but two were mainly used. A simple elitist replacement operator, `BestOfAll` which combines both populations and takes the best individuals and a modified version with niching, named `BestNiche` explained in next Section 2.4.7.1.

The `BestOfAll` operator is implemented by first concatenating the lists of individuals, then an `np.argsort` function is called in the list of fitness values. This function returns a list of indices sorted by their fitness values in ascending order. Then the individuals at indices are taken from the start of the array (low fitness is better) until there are as many individuals as in the old population.

2.4.7.1 Niching

BestNiche operator is based on the BestOfAll replacement operator. But when an individual is added to the final population, only a few similar individuals within a distance delimited by a parameter radius are also added. These individuals are called leaders. Their number is parametrized by leader_count and is expected to be very low. Experiments used leader_count 1 or 2. Other close individuals are flagged as followers and taken out of the process. This repeats until every individual in the combined population is flagged. If there are not enough leaders o fill the resulting population, the rest is selected based on fitness from the followers.

This process needs a distance metric for individuals, more specifically for their genotypes. In this implementation, a mean of Manhattan distances between layers is used. The exact formula is

$$dist(net_1, net_2) = \frac{1}{N} \sum_{i=0}^{N-1} \|layer(net_1, i) - layer(net_2, i)\|_1$$

where N is the number of layers, function $layer(net, i)$ returns a vector of all weights and biases from a layer i in a neural network net. This metric considers a difference in a small layer more significant than the same difference in a bigger layer.

The implementation is shown in Listing 2.6. Lines 2-6 are the same as in the BestOfAll operator, the populations are combined, and a sorted array of indices is constructed. Then, a distance matrix is computed, arrays of leaders and followers are initialized with a flagging array assigned signaling if the individual is assigned to either followers or leaders.

The main loop starts on line 11. The unassigned individual is added to the list of leaders as a tuple of fitness value and an index. The reason for that is the possibility to sort the values by the fitness value. Then, on line 16, indices of individuals with distance smaller than radius are retrieved into a variable niche_indices.

These indices are then cycled through from the most distant to the closest. If there are any unassigned, they are assigned to leaders if they are not identical and if leaders for this niche radius are still accepted. That is if their number is lower than leader_count. Otherwise, they are added to the followers.

After the loop, individuals of the resulting population are selected. First from the leaders, and after that from the followers, in both cases ordered by fitness values. The approved_nets with their respective fitness values are then put into a population, and a resulting population is returned.

2.5 Memetic algorithm

The implementation of MA requires a combination of evolution and local search. This version of MA uses a population initialized by local search and

```python
def select(self, old, new, rng):
    # create an empty population with the same properties
    total = old.empty_new()
    nets = np.append(old.get_nets(), new.get_nets(), axis=0)
    total.set_nets(nets) # assign the nets (individuals) to the population
    fitness = np.append(old.fitness, new.fitness, axis=0)
    indices = np.argsort(fitness)

    dists = total.get_distances() # shape [n, n], where n = len(nets)
    assigned = np.full(fitness.shape, False)
    leaders, followers = [], []
    for i in indices:
        if not assigned[i]:
            assigned[i] = True
            leaders.append((fitness[i], i))
            niche_leaders = 1
            [niche_indices] = np.where(dists[i] < self.radius)
            # the farthest are assigned as leaders first
            for rel_i in np.argsort(-dists[i, niche_indices]):
                j = niche_indices[rel_i]
                if not assigned[j]:
                    if dists[i][j] == 0 \
                       or niche_leaders >= self.leader_count:
                        followers.append((fitness[j], j))
                    else:
                        niche_leaders += 1
                        leaders.append((fitness[j], j))
                    assigned[j] = True

    goal_size = int(indices.size // 2)
    approved = 0
    approved_nets = np.empty_like(old.get_nets())
    approved_fitness = np.empty_like(old.fitness)

    for _, i in sorted(leaders):
        approved_nets[approved] = nets[i]
        approved_fitness[approved] = fitness[i]
        approved += 1
        if approved >= goal_size:
            break

    for _, i in sorted(followers):
        if approved >= goal_size:
            break
        approved_nets[approved] = nets[i]
        approved_fitness[approved] = fitness[i]
        approved += 1

    res = old.empty_new()
    res.set_nets(approved_nets)
    res.fitness = approved_fitness
    res.best_i = np.argmin(approved_fitness)
    return res
```

Listing 2.6: Replacement with niching: `BestNiche` method implementation

follows with evolution. For neural networks, a standard method of local search is SGD, which was used in this thesis.

Backpropagation that was not implemented is required for SGD. Thus the PyTorch library was used by modifying a simple demo [23] to fit the problem. The trained neural network is then transformed to the pure NumPy implementation, and evolution is run as described earlier on a population filled with the pre-trained networks.

2.5.1 Data sampling

To gather labeled data for the training and validation, a special `Simulator` class was created. It is implemented similarly to the fitness evaluation. There are mazes and an agent that is trying to solve them. Each time step, the agent receives input and takes an action. The input is stored. Based on the position and orientation of the agent, the algorithm determines the correct action (the one in direction to the goal) and stores it as a label to the input. If the best action is to turn back, both left and turn are equally good actions to take. The implementation prefers right turns every time.

This process ran on 100 10x10 mazes, with 15 neural networks of the 10 best configurations found so far. That totals 150 different neural networks running for a maximum of 200 time steps on 100 different mazes. This simulation generated 1 565 056 labeled rows of data, 166 618 of which were unique.

2.6 CMA-ES

The CMA-ES is has many implementations, the chosen implementation is library cmaes [24] which is the most lightweight implementation available. It uses an ”ask and tell” interface to guide the optimization and contains a function for a warm start to the evolution. This warm start function takes several vectors and their fitness values, and based on their values and position in the n-dimensional space, estimates the distribution parameters. That helps us set the initial multivariate normal distribution in an informed way, hopefully leading to better results.

Since ESs are used for real value optimization, all weights and biases of the network had to be flattened and concatenated into a n-dimensional vector of real numbers.

The usage of this library is quite different from the GA implementation because it does not use crossover, mutation, selection, or replacement operators. The only element of the implemented GA used by CMA-ES is fitness evaluation. The library even has a heuristic estimating the appropriate size of a population (number of vectors sampled from the distribution in each generation) given the dimensionality of the search space.

Chapter 3

Experiments

The most important part of this thesis is the experiments. They consisted of over 100 tests. The volume of gathered data is above the limit of reason, so only a few selected notable results are compared.

All presented tests ran 15 times to ensure statistical reliability. They were typically evaluated on 10x10 mazes, with 200 maximal time steps, with a population of 100, and for 100 000 fitness function evaluations. For all tests (where relevant) SUS was used as a selection operator. All test configurations, even those not presented here, are specified in the appended files, in `src/test_definitions.ods`.

To compare the performance of various configurations and algorithms, 3 main methods were used.

Evolution process is visualized using a plot of the fitness value of the best so far (BSF) solution. The x-axis of this type of plot uses the number of maze run simulations as a metric since the simulation of a maze solving attempt is the most time-consuming part of the algorithm. Another option was the number of generations or fitness evaluations. But the time complexity of those values depends on the population size or the number of mazes used in the evaluation, respectively.

The performance of the best solution, typically the best individual in the final population, is visualized using two kinds of plots.

One is the *shortest distance from the goal tile reached* within the assigned maximal time steps, averaged over all mazes. This plot typically uses only mazes generated by Kruskal's algorithm (Kruskal mazes), as they are more complex and the maximal distance value is more consistent. It is a box-plot containing the data from all — typically 15 — iterations of a given configuration.

The second plot is a similar box-plot, but the plotted values are *number of mazes solved* by the given iteration. In this plot, both Kruskal and DFS mazes are usually used to give a better sense of the performance of the given configuration.

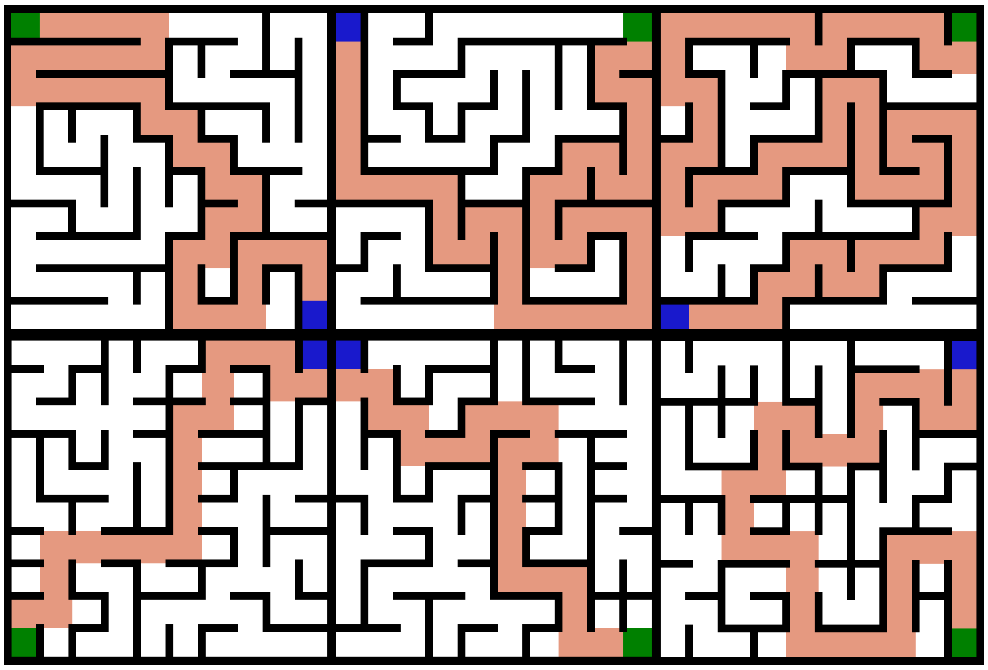


Figure 3.1: Static mazes used in the experiments: 3 on the top were generated by DFS and 3 on the bottom were generated by Kruskal's algorithm.

For most runs, a set of six 10x10 mazes was generated. Three were generated with Kruskal's algorithm, and three with DFS algorithm. These static mazes are preset and do not change throughout the evolution. They are shown in Figure 3.1. Many tests used only the 3 Kruskal-generated mazes. It is important to note that this is significantly less than in Shorten's experiments where the individuals in the evolution process were evaluated on 100 mazes [17].

3.1 Neural Architectures

The first comparison is of the different Neural Architectures. The evaluation was done on all 6 of the static mazes discussed earlier. All parameters, except the size and the architecture of the neural nets, were the same. The differences in architectures are described in Section 2.3.

As can be seen in the comparison of convergence progress in Figure 3.2, the convergence speed of RNNs is hardly comparable to that of feed-forward ANNs. From the comparison of best solutions in Figure 3.3, the feed-forward ANNs also dominate the RNNs in both distance reached and mazes solved, where RNNs perform around the level of a randomized agent.

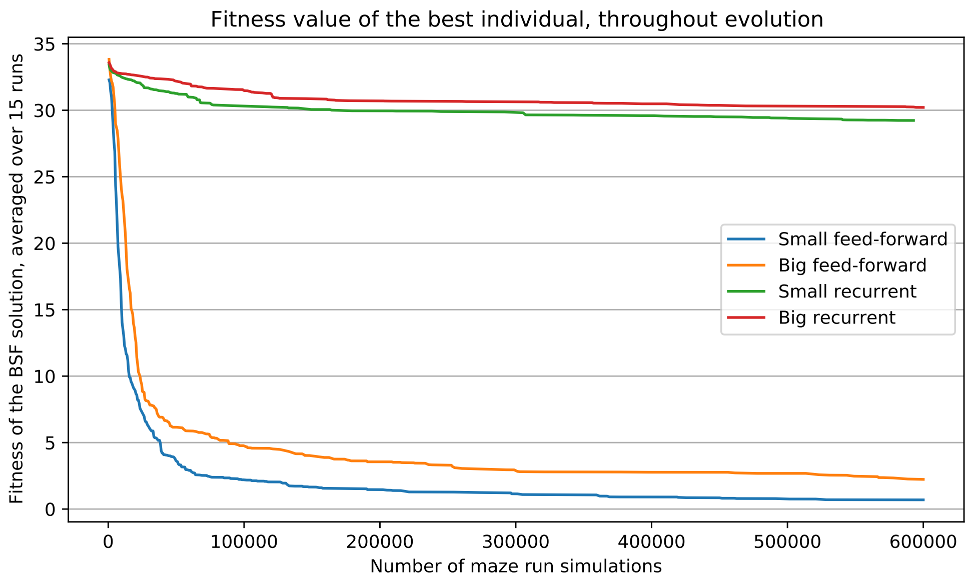


Figure 3.2: *Neural Architectures convergence comparison.* Feed-forward ANNs have a significantly better convergence rate. RNNs stagnate at almost initial values.

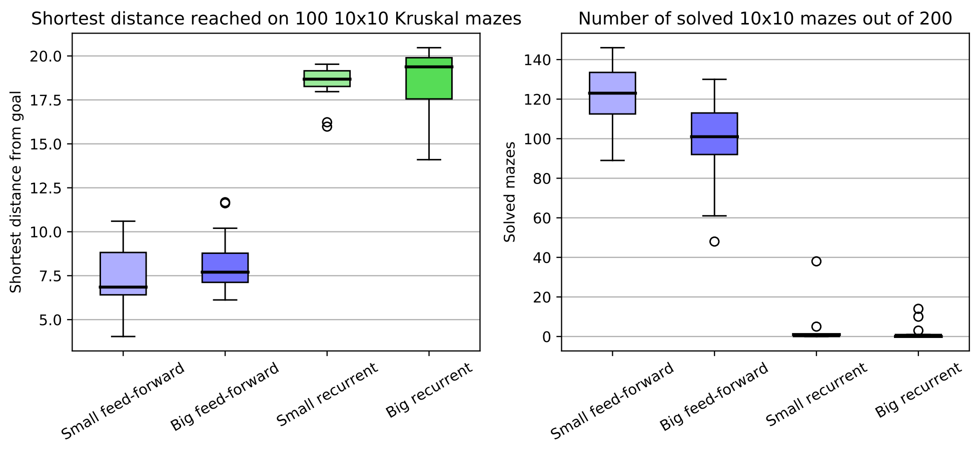


Figure 3.3: *Neural Architectures performance comparison.* Feed-forward ANNs generalize significantly better. RNNs have the performance comparable to a random action picker.

These disappointing results of RNNs are the reason that the rest of the experiments presented here focus mainly on the Feed-forward architecture. That does not mean that RNNs were not tested further. The best performing RNN — found using CMA-ES — is presented at the end of this chapter.

3.2 Different maze generators

As was already mentioned, two maze generators were implemented. The effect of evaluation performed on only one type of mazes is noticeable and was significant in the testing process. For this comparison, Layer crossover was used for big ANNs and Neuron crossover for small ANNs. Other than that, the same parameters were used. Except, of course, whether they were evaluated on DFS or Kruskal mazes only.

In these runs, the mazes were regenerated at the start of each generation. This caused the fitness value of a BSF solution to fluctuate throughout the evolution. That is why floating average was used to flatten the curve in the convergence plot, see Figure 3.4.

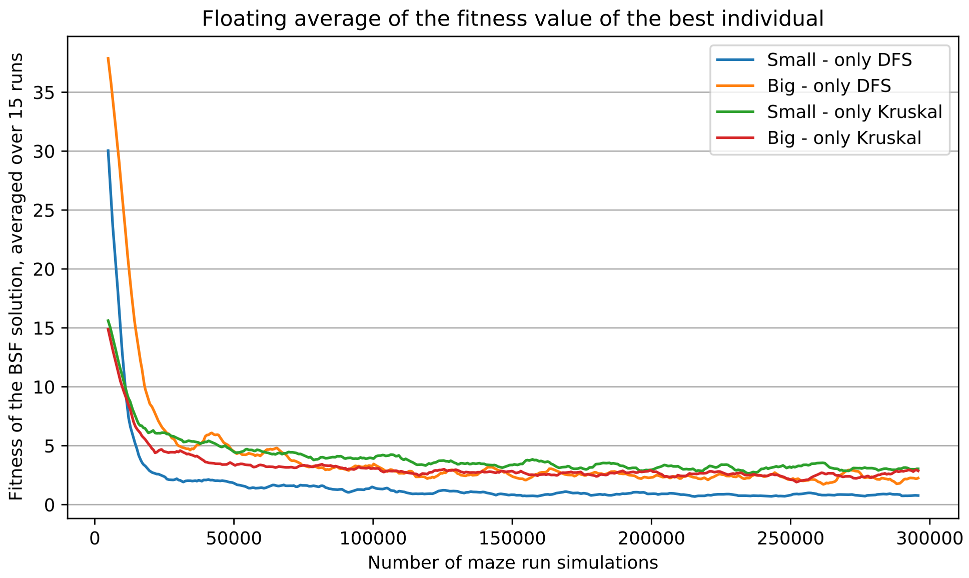


Figure 3.4: *Evaluators on different mazes convergence comparison.* Clearly visible is a better starting point for those evaluated on Kruskal mazes, as they typically have shorter distances from start to goal. However, they are harder to solve, which is noticeable from the fact that DFS evaluated individuals outperform those evaluated on Kruskal.

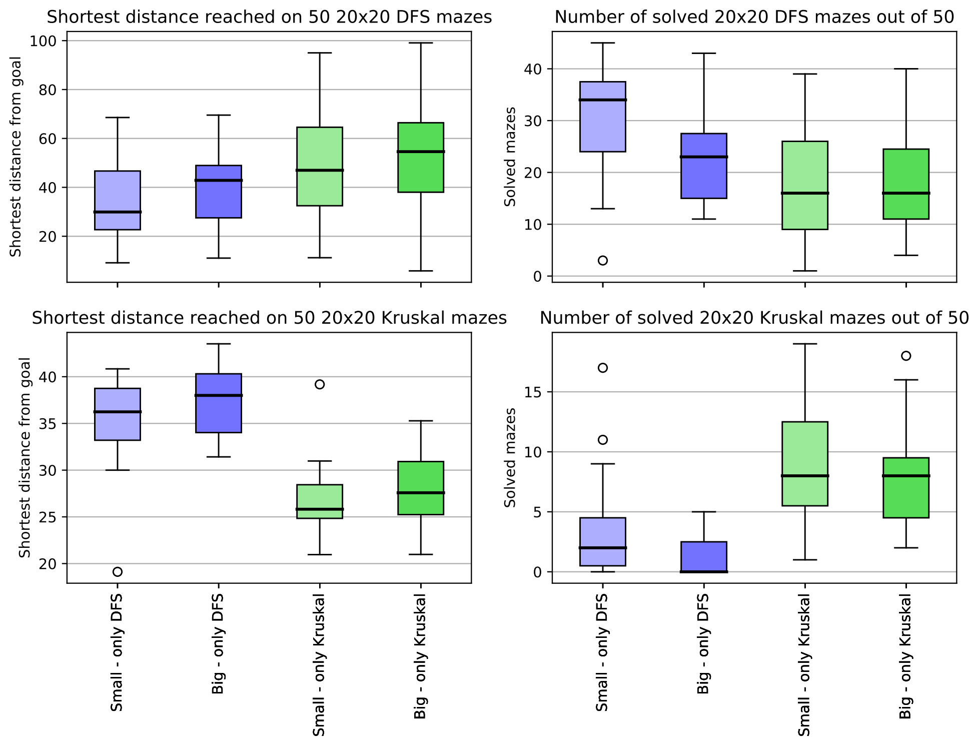


Figure 3.5: *Evaluators on different mazes performance comparison.* Comparison was done on 20x20 mazes to reflect generalizing properties of the implementations. While individuals evaluated on mazes of either generator perform better on that kind of mazes, the Kruskal evaluated individuals are more versatile, reaching lower, but not abysmal performance on DFS mazes. DFS evaluated individuals, on the other hand, have significantly greater difficulties in solving Kruskal generated mazes.

3.3 Regenerated mazes

In the previous section, individuals evaluated on regenerated mazes were discussed. This was an alternative approach, where mazes were not static throughout the evolution but were regenerated each generation. This meant that all individuals in the old population had to be reevaluated every generation to make them comparable to the offspring. The individual's fitness is not constant, and thus the plot of BSF solution fitness in the evolution process does not decrease monotonously. To better visualize the convergence, a floating average is used.

It is an interesting property of the EA that the individuals converge nonetheless to a value close to 0, meaning that the individuals optimize a behavior that is able to solve or almost solve any random maze. To enable stability of the evaluation, 3 mazes are generated each generation and used in the evaluator. All mazes in these experiments use the Kruskal generator. To compare the capabilities of evolving on regenerated and static mazes, two configurations, one small and one big feed-forward architecture were used. They were being evaluated on 3 regenerated mazes, on 3 static Kruskal mazes (from the pre-defined 6), and lastly on 10 static Kruskal mazes, not shown here.

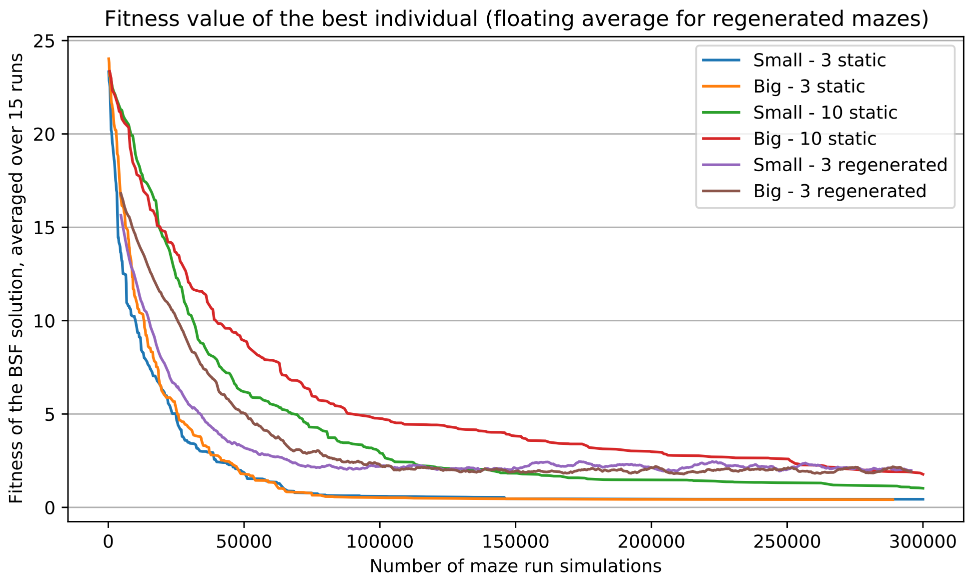


Figure 3.6: *Regenerated and static mazes convergence comparison.* The slow convergence on 10 mazes is due to longer generations with respect to maze run simulations. The floating average of fitness oscillates around 2, eventually being surpassed by both static evaluated configurations.

Since all runs were calibrated for the total of 300 000 maze run simulations, it meant less than a third of generations for the runs evaluated on 10 Kruskal mazes in comparison to the runs on 3 static mazes. This is the reason for the slower convergence seen in Figure 3.6. An interesting result to note is the comparability of the performance of the individuals evaluated on only 10 mazes compared to those evaluated on a couple of hundred mazes (though not every generation).

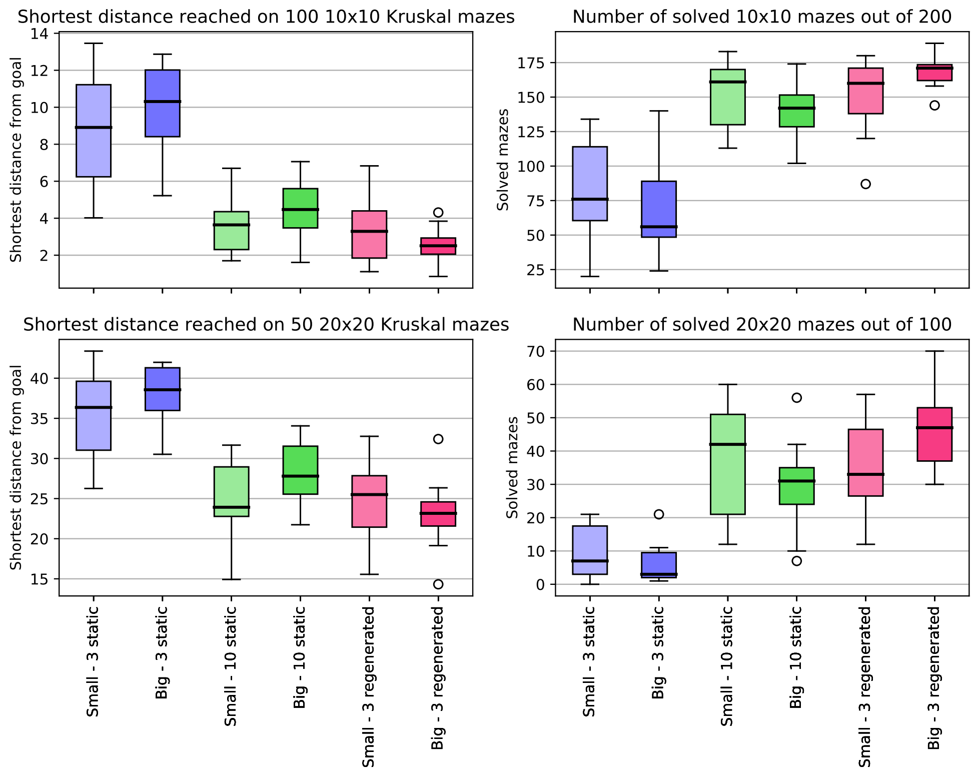


Figure 3.7: *Regenerated and static mazes performance comparison* also on 20x20 mazes since the regenerated evaluation could have used some of the 10x10 mazes in the comparison.

3.4 Niching

After a couple of initial tests, a closer look at the final populations lead to a discovery that up to 90% of the individuals were precisely the same. This was a major issue, as the population is very unlikely to progress if crossover has no effect.

A replacement operator with niching was implemented, and multiple combinations of niching parameters were tested until the most effective were found. These are `niche_radius`=0.5 and `leader_count`=2.

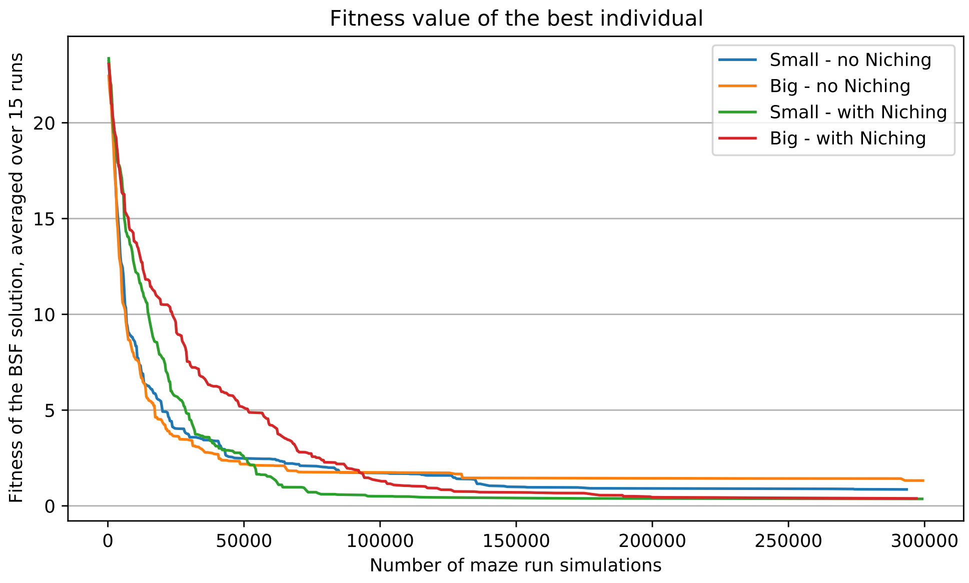


Figure 3.8: *Niching convergence comparison.* Niching prolongs the convergence, but the final value is lower than without niching.

Niching was tested on recurrent mazes too. However, a significant effect on the convergence rate was not found.

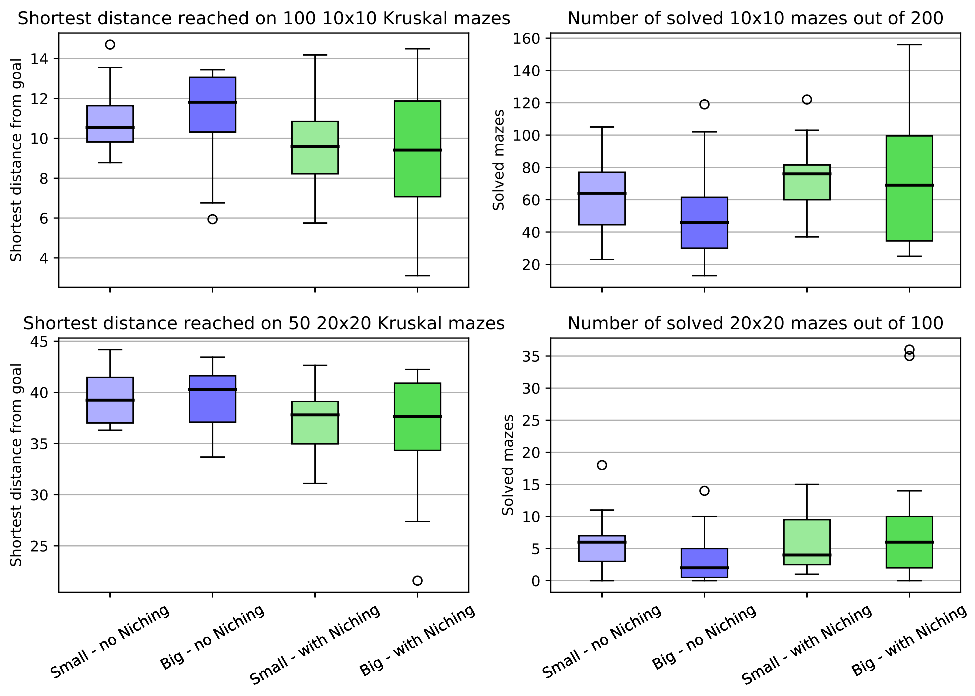


Figure 3.9: *Niching performance comparison.* The most improvement is visible in the overall shortest distance that the niched individuals can reach. However, some increase in solving capabilities is also noticeable.

3.5 Population size

Generally, an important hyperparameter in EA is population size. In most tests, a population size of 100 was used. However this comparison, see Figures 3.10 and 3.11, of 6 configuration different only in architecture and population size shows, that in some cases lower population size can be better.

It is important to mention that all these configurations used niching. If not, the results on lower population size would stagnate easier and probably not reach the results they did here.

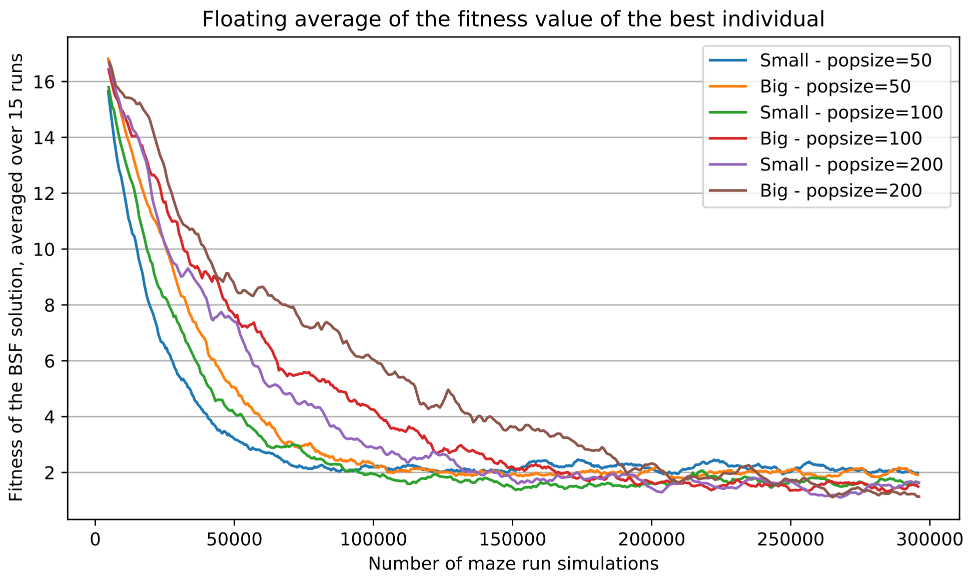


Figure 3.10: *Population size convergence comparison.* This shows well that even though all configurations converge to a similar value, the smaller mazes with a lower population converge faster.

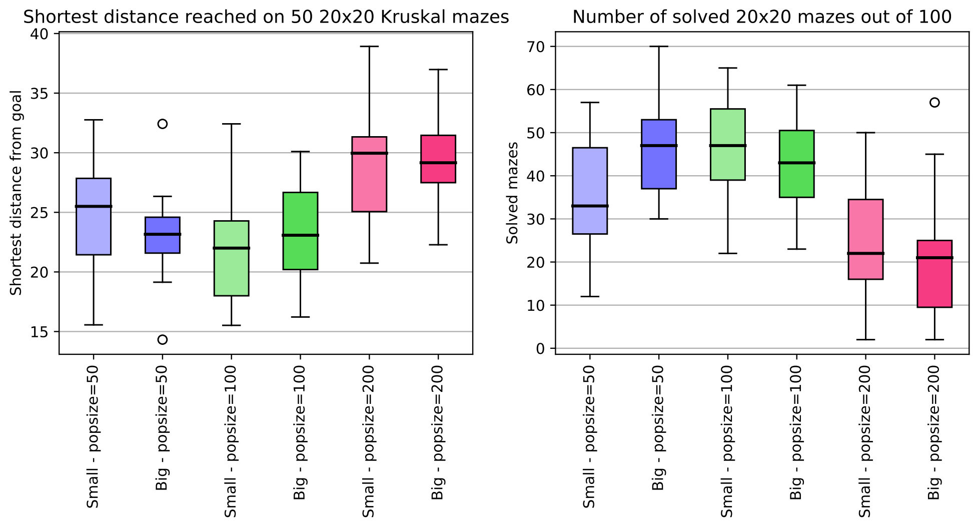


Figure 3.11: *Population size performance comparison.* While population size 100 is best for the small architecture, one could argue that population size 50 is better for the big architecture, based on the results in the number of solved mazes.

3.6 CMA-ES

CMA-ES was one of the main approaches tested in this thesis. As was mentioned earlier, it was the most successful in optimizing the RNN architectures. But here are the best results on feed-forward architecture achieved by CMA-ES and implemented GA. Both were tested extensively, and the best performing was chosen for this comparison.

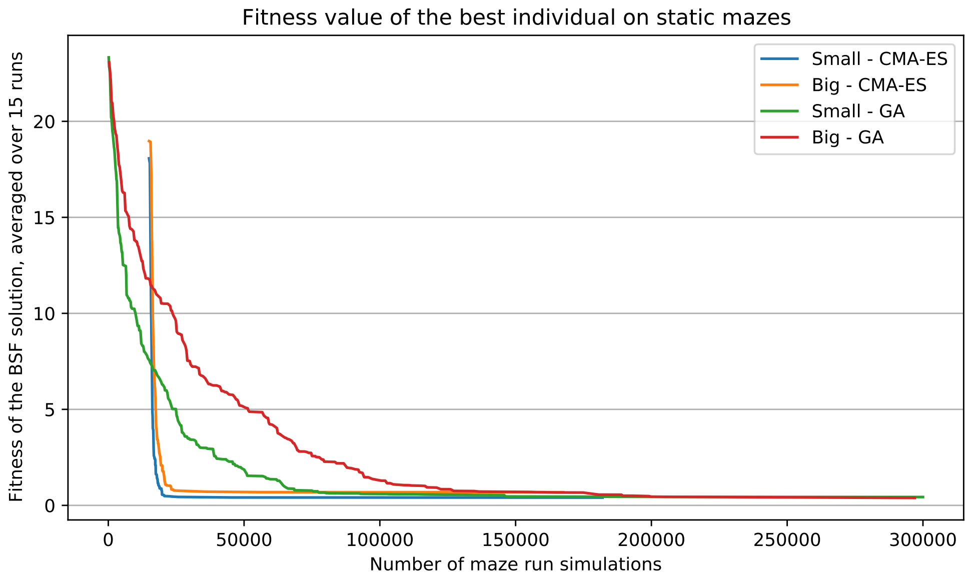


Figure 3.12: *CMA-ES convergence comparison.* The CMA-ES has an amazing speed of convergence. The volume of available maze run simulations was not required. A seemingly late start is created by the warmup phase before the CMA-ES. In this configuration, the algorithm samples 5 000 individuals to estimate the distribution. That equals 15 000 maze run simulations before the CMA-ES even starts.

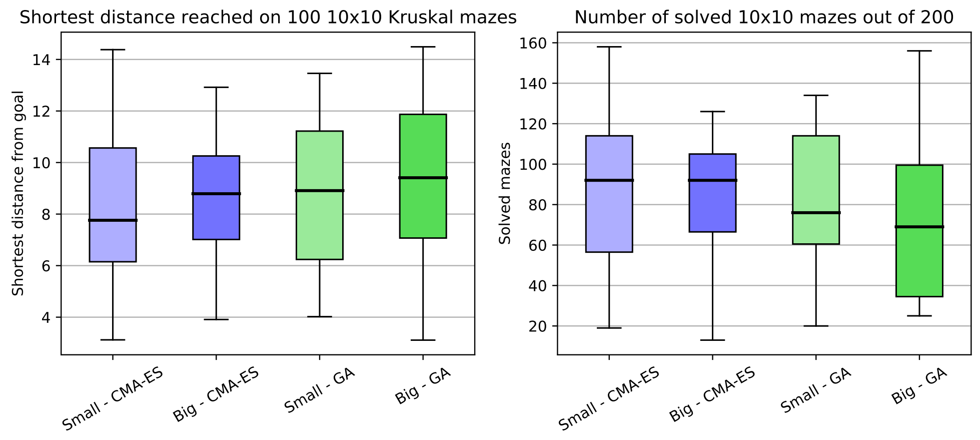


Figure 3.13: *CMA-ES performance comparison on static mazes.* The CMA-ES has superior performance in optimizing on static mazes.

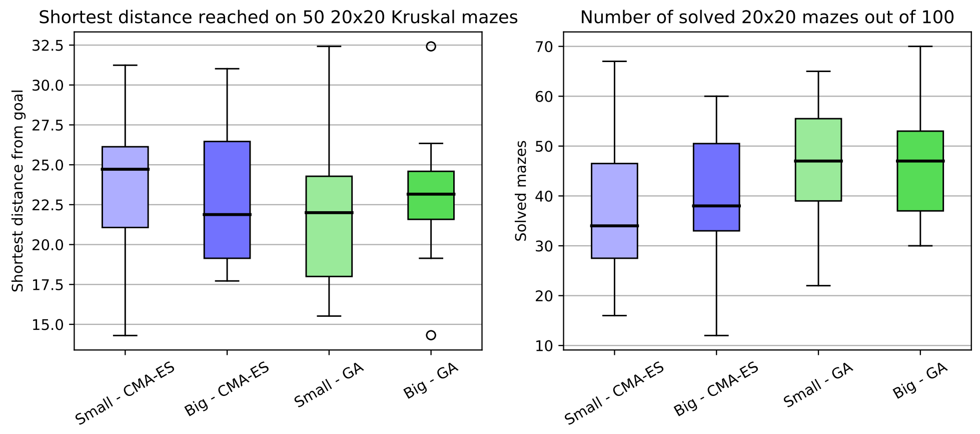


Figure 3.14: *CMA-ES performance comparison on regenerating mazes.* CMA-ES seems to have difficulties — especially compared to its performance on static mazes — to optimize the weights on an unstable state space created by constantly regenerating mazes.

3.7 Memetic algorithm

The implementation of a Memetic Algorithm required pre-optimized individuals. These were generated using standard SGD and backpropagation. 15 such individuals were trained on all 1.5 million rows (divided into training and validation dataset) and another 15 individuals on the unique rows (there were around 160 000 of those). Combined, they generated 30 individuals of small architecture and 30 individuals of big architecture. Their performance is visualized in Figure 3.15.

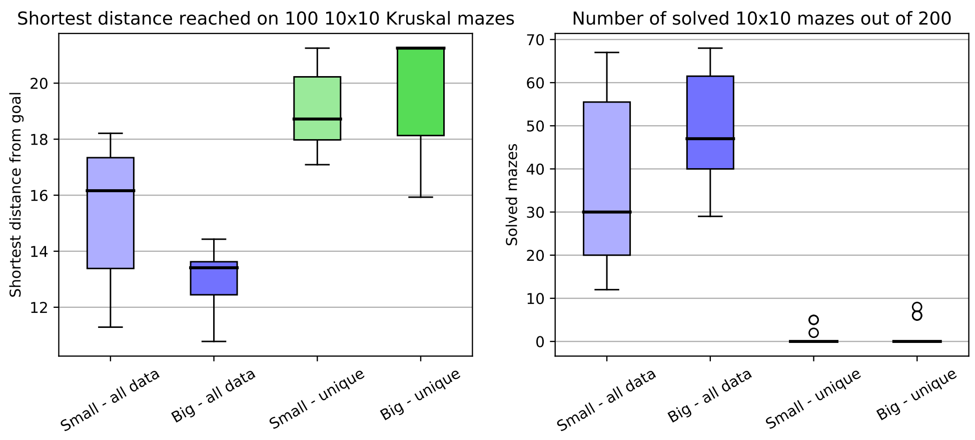


Figure 3.15: *Performance of individuals generated by SGD and backpropagation.* The staggering incapability of the individuals trained on unique data was unexpected. On the other hand, on all data, similar performance to that of the best obtained recurrent networks can be seen.

To make a good comparison with the results of MA, two exactly the same configurations were used. They all use niching and have a lower standard deviation on the mutation operator because smaller modifications were expected to be required. The only difference is in the initial population. Both have size 30, but one is made up of pre-optimized individuals as seen in Figure 3.15 and the other population was initialized randomly as with a generic GA.

The results, see Figures 3.16 and 3.17, were not too impressive, so CMA-ES was tested with warm starting the distribution on the 30 pre-trained mazes. However, this did not lead to any performance boost, rather the opposite.

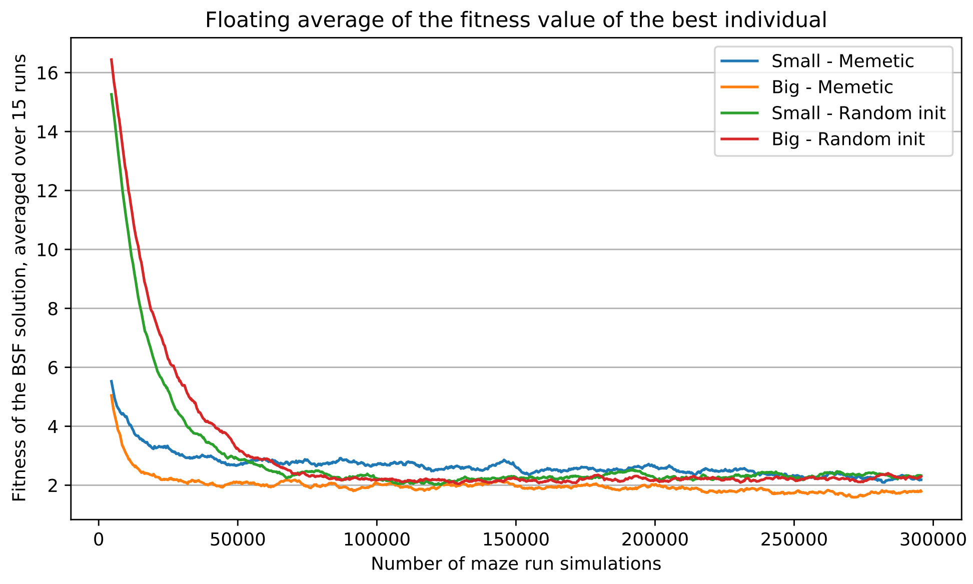


Figure 3.16: *MA and GA convergence comparison.* The MA have a significant advantage in the starting fitness value of BSF solution. The big architecture seems to improve even in the last third of the evolution, which is not common in these experiments.

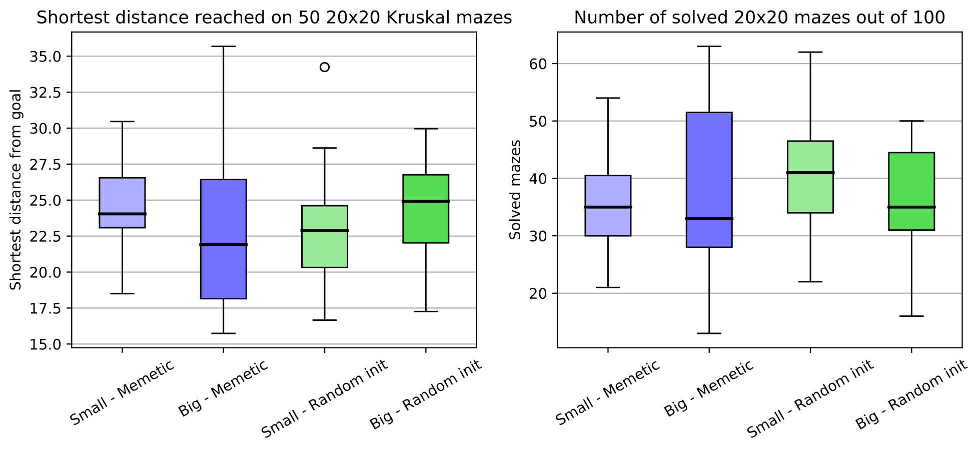


Figure 3.17: *MA and GA performance comparison.* Similarly to the convergence plot, the big architecture seems to favor the MA. It also reflects the relatively good quality of the SGD results on big mazes.

3.8 Final Comparison

In this final section, the best results of various configurations and approaches are compared. Two feed-forward networks, one trained on 10 static and the other on 3 regenerated Kruskal mazes. Both architectures are small and use niching. Then there is the best achieved RNN, using CMA-ES warm started on 5000 individuals, evaluated on 3 regenerated Kruskal mazes. The RNN was run for fewer maze run simulations because the CMA-ES proved to have faster convergence. This change had no negative impact on the solutions.

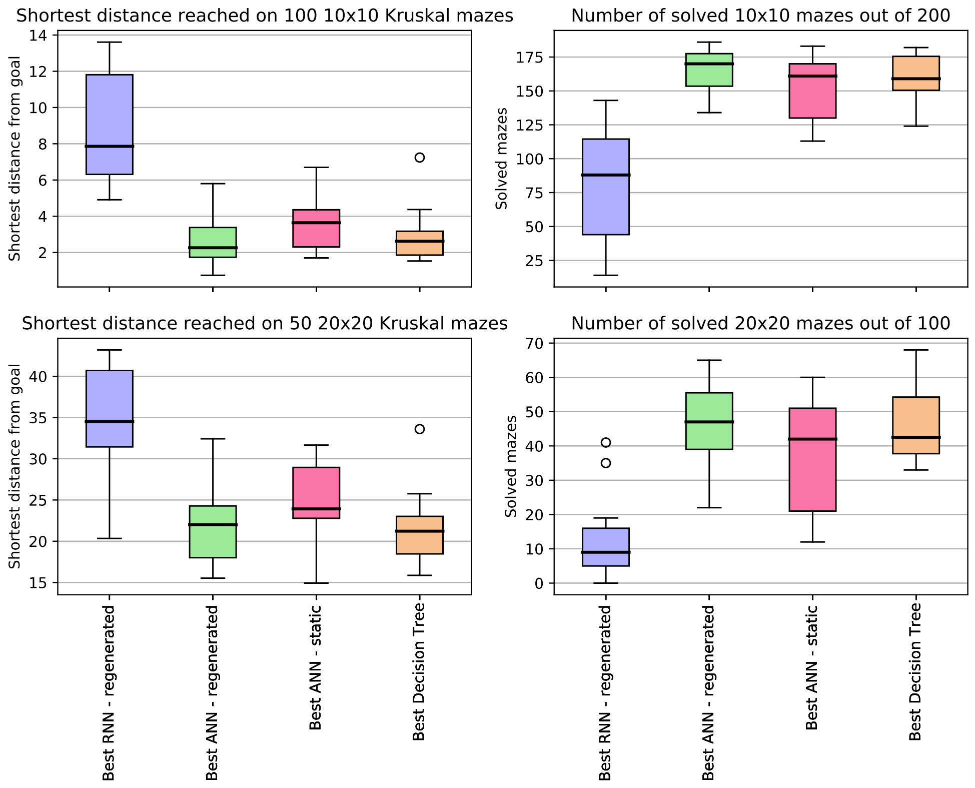


Figure 3.18: *Best configurations performance comparison.* Except for RNN, the results are comparable, showing around 40-50% chance of solving a previously unseen maze of 4 times the size.

The last approach compared here used Decision Trees and was created by the author in a separate work. The crossover and mutation operators were different, but SUS and a simple elitist replacement strategy — like the `BestOfAll` — were used. It was also evaluated on 3 regenerating mazes and for the same amount of maze simulations. The specific configuration of parameters is not discussed here since it was a part of a different assignment. It is, however, the best found configuration among those tested.

3.8.1 Comparison to Wall Follower

Wall Follower (WF) is a common strategy used to navigate mazes. An interesting property to look for in a maze solver is its effectiveness compared to the WF algorithm.

The best performing configurations were thus tested to see if they solve mazes faster or slower than WF. Since all mazes generated by the implemented generators are solvable by WF, the number of actions it takes a right-hand wall follower was taken as a threshold. Then those iterations that succeed in solving the maze with fewer actions are flagged as better than a WF on the given maze. Finally, by seeing if more iterations succeed in beating the WF or not, the configuration is considered faster or slower than WF. By aggregation over multiple mazes, data in Table 3.1 were collected. The table is sorted by success in beating the WF. The mazes in question are 10x10 Kruskal mazes since they were the baseline mazes.

Table 3.1: *Comparison to right-hand Wall Follower (WF).* A major portion of the lower performance results were individuals unable to solve the maze. In attempts when the maze was solved, the dominance of an evolved agent compared to WF was evident.

Number of mazes, where WF was	**Slower**	**Same**	**Quicker**
Best ANN - regenerated	34	0	66
Best Decision Tree	29	7	64
Best ANN - static	29	1	70
Best RNN - regenerated	12	1	87

Conclusion

The aim of the thesis was the exploration of evolutionary methods and their capabilities in their use with GNNs. More specifically, the generalization of maze solving behavior of an agent was pursued. This task consisted of exploring the theoretical background, implementing a visualization program with some evolutionary methods, and finally testing multiple evolutionary approaches and comparing their properties.

All objectives were successfully completed, starting with a revision of the theory behind EAs, MAs and GNNs. This included a simple glossary with the most important terms used in this thesis. On top of that, a description of various evolutionary operators with examples, including some special operators for GNNs. Maze solving problem and commonly used techniques were also discussed.

In Chapter 2, the implementation was described in detail for environment definition, maze generation, GUI layout, and various used evolutionary operators, which were necessary for extensive testing in the next chapter.

The experiments compared multiple different evolutionary approaches to the optimization of a maze solver using neural architectures. Although the experiments did not achieve as significant results as the author hoped they would, the GNN approach proved to work better than the decision tree approach. A wide range of configurations was tested, compared and the benefits of some configurations were outlined. Over the weeks of CPU time that the tests took, hundreds of megabytes of resulting neural nets were generated and are freely accessible on the attached medium.

The resulting generalization properties were quite significant, considering the performance of individuals evolving in mere 10 mazes when encountering a wast space of unseen, bigger, and harder mazes.

The results of this thesis can prove to be a useful starting point in the search for an appropriate evolutionary algorithm since the spread of tested approaches is very wide. There are many things that could be tested further.

Further Improvements

One potential focus would be the RNNs since the results of those were very poor. Based on prior research, it is likely that RNNs, without additional information about the number of visits to neighboring tiles, should give better results than those achieved in this thesis. An increase in size and depth of the network might help since additional information needs to be remembered.

Another direction that would be interesting to explore is a more sophisticated testing of the MA approach. The extent to which this approach was explored in this thesis does not do justice to the potential the field of MA presents. Specifically, in this case, more effort would require to be put into the pre-training phase, or a more hybrid approach with one epoch of learning in every generation might be a good approach to try.

A questionable decision was to test the configurations on only 3 static mazes. The results clearly show that the fitness value converges quickly to the optimal value at around 0, while the overall capabilities stay mediocre at best. This hyperparameter was set too low, and its increase would be a great direction to take in exploring the subject further.

Bibliography

[1] Whitley, D. An overview of evolutionary algorithms: practical issues and common pitfalls. *Information and Software Technology*, volume 43, no. 14, dec 2001: pp. 817–831, doi:10.1016/s0950-5849(01)00188-4. Available from: `https://www.sciencedirect.com/science/article/pii/S0950584901001884`

[2] Holland, J. H.; et al. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992. Available from: `https://books.google.cz/books?id=wS0LEAAAQBAJ`

[3] Beyer, H.-G.; Schwefel, H.-P. Evolution strategies – A comprehensive introduction. *Natural Computing*, volume 1, no. 1, 2002: pp. 3–52, doi:10.1023/a:1015059928466. Available from: `https://link.springer.com/article/10.1023/A:1015059928466`

[4] Alain Petrowski, S. B.-H. *Evolutionary Algorithms*. John Wiley & Sons, Apr. 2017, ISBN 9781848218048. Available from: `https://ebookcentral.proquest.com/lib/cvut/detail.action?docID=4841876`

[5] Koza, J. R. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992. Available from: `https://www.researchgate.net/profile/Riccardo-Poli/publication/229091441_Genetic_Programming/links/0046352a190b0a465c000000/Genetic-Programming.pdf`

[6] Beyer, H.; Brucherseifer, E.; et al. Evolutionary algorithms - terms and definitions. *VDI/VDE-Richtlinie-3550, Blatt*, volume 3, 2002. Available from: `https://homepages.fhv.at/hgb/ea-glossary/`

[7] Goldberg, D. E.; Richardson, J.; et al. Genetic algorithms with sharing for multimodal function optimization. In *Genetic algorithms and their applications: Proceedings of the Second International Conference on Genetic Algorithms*, Hillsdale, NJ: Lawrence Erlbaum, 1987, pp. 41–49. Available from: `https://books.google.cz/books?id=MYJ_AAAAQBAJ&pg=PA41`

[8] Hansen, N.; Ostermeier, A. Completely Derandomized Self-Adaptation in Evolution Strategies. *Evolutionary Computation*, volume 9, no. 2, jun 2001: pp. 159–195, doi:10.1162/106365601750190398. Available from: `https://direct.mit.edu/evco/article-pdf/9/2/159/1493523/106365601750190398.pdf`

[9] Akimoto, Y.; Nagata, Y.; et al. Bidirectional Relation between CMA Evolution Strategies and Natural Evolution Strategies. In *Parallel Problem Solving from Nature, PPSN XI*, Springer Berlin Heidelberg, 2010, pp. 154–163, doi:10.1007/978-3-642-15844-5_16. Available from: `https://link.springer.com/chapter/10.1007/978-3-642-15844-5_16`

[10] Branke, J.; Hildebrandt, T.; et al. Hyper-heuristic Evolution of Dispatching Rules: A Comparison of Rule Representations. *Evolutionary Computation*, volume 23, no. 2, jun 2015: pp. 249–277, doi:10.1162/evco_a_00131. Available from: `https://direct.mit.edu/evco/article/23/2/249/985/Hyper-heuristic-Evolution-of-Dispatching-Rules-A`

[11] Moscato, P.; et al. On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. *Caltech concurrent computation program, C3P Report*, volume 826, 1989. Available from: `https://www.researchgate.net/profile/Pablo-Moscato/publication/2354457_On_Evolution_Search_Optimization_Genetic_Algorithms_and_Martial_Arts_-_Towards_Memetic_Algorithms`

[12] Cotta, C.; Mathieson, L.; et al. *Memetic Algorithms*. Cham: Springer International Publishing, 2018, ISBN 978-3-319-07124-4, pp. 607–638, doi:10.1007/978-3-319-07124-4_29. Available from: `https://doi.org/10.1007/978-3-319-07124-4_29`

[13] Paolucci, R. Genetic Artificial Neural Networks. [online], Feb 2020, [Last Accessed: 22/04/2020]. Available from: `https://medium.com/swlh/genetic-artificial-neural-networks-d6b85578ba99`

[14] Montana, D. J.; Davis, L. Training feedforward neural networks using genetic algorithms. In *IJCAI*, volume 89, 1989, pp. 762–767. Available from: `https://www.ijcai.org/Proceedings/89-1/Papers/122.pdf`

[15] Sexton, R. S.; Dorsey, R. E.; et al. Toward global optimization of neural networks: A comparison of the genetic algorithm and backpropagation. *Decision Support Systems*, volume 22, no. 2, feb 1998:

pp. 171–185, doi:10.1016/s0167-9236(97)00040-7. Available from: https://www.researchgate.net/publication/222477189_Toward_global_optimization_of_neural_networks_A_comparison_of_the_genetic_algorithm_and_backpropagation

[16] Porto, V.; Fogel, D. Alternative neural network training methods [active sonar processing]. *IEEE Expert*, volume 10, no. 3, June 1995: pp. 16–22, ISSN 2374-9407, doi:10.1109/64.393138. Available from: https://ieeexplore.ieee.org/document/393138

[17] Shorten, D.; Nitschke, G. Evolving Generalised Maze Solvers. In *Applications of Evolutionary Computation*, edited by A. M. Mora; G. Squillero, Cham: Springer International Publishing, 2015, ISBN 978-3-319-16549-3, pp. 783–794. Available from: https://link.springer.com/chapter/10.1007/978-3-319-16549-3_63

[18] Bega, D.; Samocha, Y.; et al. The effect of maze complexity on maze-solving time in a desert ant. *Behavioural Processes*, volume 166, sep 2019: p. 103893, doi:10.1016/j.beproc.2019.103893. Available from: https://www.sciencedirect.com/science/article/pii/S0376635719301111?via=ihub

[19] Sadik, A. M.; Dhali, M. A.; et al. A Comprehensive and Comparative Study of Maze-Solving Techniques by Implementing Graph Theory. In *2010 International Conference on Artificial Intelligence and Computational Intelligence*, IEEE, oct 2010, doi:10.1109/aici.2010.18. Available from: https://ieeexplore.ieee.org/abstract/document/5656597/

[20] Saman, A. B. S.; Abdramane, I. Solving a Reconfigurable Maze using Hybrid Wall Follower Algorithm. *International Journal of Computer Applications*, volume 82, no. 3, nov 2013: pp. 22–26, doi:10.5120/14097-2114. Available from: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.402.1370&rep=rep1&type=pdf

[21] The Kivy Authors. Welcome to Kivy. [online], 2018, [Last Accessed: 23/04/2020]. Available from: https://kivy.org/doc/stable/

[22] The SciPy community. NumPy v1.20 Manual. [online], 2020, [Last Accessed: 29/04/2020]. Available from: https://numpy.org/doc/stable/

[23] PyTorch. PyTorch: optim. [online], 2021, [Last Accessed: 29/04/2020]. Available from: https://pytorch.org/tutorials/beginner/examples_nn/polynomial_optim.html

[24] Shibata, M. cmaes. [online], Feb 2021, [Last Accessed: 30/04/2020]. Available from: https://pypi.org/project/cmaes/

Appendix A

Acronyms

AI	Artificial Intelligence. 1, 12
ANN	Artificial Neural Network. 1, 3, 9, 10, 12, 16, 18, 21, 32, 33, 49
BSF	best so far. 31, 33, 36, 46
CMA-ES	Covariance Matrix Adaptation – Evolution Strategy. 3, 9, 11, 29, 33, 43–45, 47
DFS	Depth-First Search. 14, 31–35
EA	Evolutionary Algorithm. 1–4, 10, 12, 13, 19, 20, 36, 41, 51
EP	Evolutionary Programming. 3
ES	Evolution Strategy. 3, 29
GA	Genetic Algorithm. 3, 9–11, 29, 43, 45, 46
GNN	Genetic Neural Network. 1, 10–13, 51
GP	Genetic Programming. 3
GUI	Graphical User Interface. 13, 15, 16
MA	Memetic Algorithm. 3, 10, 20, 27, 45, 46, 51, 52
RNN	Recurrent Neural Network. 18, 21, 32–34, 43, 47–49, 52
SGD	Stochastic Gradient Descent. 10, 27, 45, 46
SUS	Stochastic Universal Sampling. 5, 31, 47
WF	Wall Follower. 11, 12, 49

Appendix **B**

Contents of enclosed medium

```
 README.md ............................ the file with directory description
 src .................................. the directory of source codes
   app.py ............................. the main run script
   comparator.ipynb ................... the Jupyter notebook with comparisons
   requirements.txt ................... the required libraries definition
   test_definitions.ods ............... the file with configuraions of all tests
   mazes .............................. the saved static mazes used in evaluations
   nets ............................... the optimized nets, test results
   plots .............................. the plots generated by the comparator.ipynb
   supervised_data .................... the labeled data used for backpropagation
 thesis ............................... the directory of the thesis text
   acronyms.tex ....................... the glossary definition
   BachelorsThesis.pdf ................ the thesis text in PDF format
   BachelorsThesis.tex ................ the thesis source in LaTeX format
   FITthesis.cls ...................... the slightly modified FIT TeX theme file
   iso690.bst ......................... the BibTeX formatting definition file
   ref.bib ............................ the BibTeX references file
   chapters ........................... the chapters sources in LaTeX format
   include ............................ the graphics used in the thesis
   sources ............................ the PDFs of some cited (and other relevant) sources
```