IMPACT OF INDEXED MEMORY ON PERFORMANCE IN LINEAR GENETIC PROGRAMMING FOR CLASSIFICATION TASKS

by

Lauren Galbraith

Submitted in partial fulfillment of the requirements
for the degree of Bachelors of Computer Science, Honours in Computer Science

at

Dalhousie University
Halifax, Nova Scotia
December 2023

Table of Contents

List of Figures . . . . **iv**

Abstract . . . . **v**

Acknowledgements . . . . **vi**

Chapter 1 Introduction and Background **1**

1.1 Evolutionary Algorithms 1

1.2 Genetic Programming 2

1.3 Linear GP 2

1.4 Crossover 3

1.5 Mutation 3

1.6 Fitness Sharing 3

1.7 Memory 4

Chapter 2 Experimental Environment **5**

2.1 Tasks 5

2.2 Virtual Machines 6

2.3 Patterns 7

2.4 Instructions 7

2.5 Program Generation 7

2.6 Instruction Set 8
2.6.1 Read and Write 8

2.7 Memory Structure 8

2.8 Parsing and Execution 9

2.9 Reward 9

Chapter 3 Experiments **11**

3.1 Evolution 11

3.1.1 Mutation 12

3.2 Champion Selection 12

3.3 Expectations 13

Chapter 4 Results 14

4.1 Memory Usage 14
4.1.1 Iris Dataset 17
4.1.2 Ionosphere Dataset 17
4.1.3 Breast Cancer Dataset 17
4.1.4 Heart Dataset 18
4.1.5 Glass Dataset 18
4.1.6 Tictactoe Dataset 20

4.2 Performance 20
4.2.1 Iris Dataset 21
4.2.2 Ionosphere Dataset 22
4.2.3 Breast Cancer Dataset 22
4.2.4 Heart Dataset 22
4.2.5 Glass Dataset 23
4.2.6 Tictactoe Dataset 23

Chapter 5 Analysis 25

5.1 Memory Usage 25

5.2 Performance 26

Chapter 6 Conclusion 28

6.1 Further Research 28

Bibliography 30

List of Figures

Figure 4.1 Heatmap of write frequencies across all tasks 15

Figure 4.2 Heatmap of read frequencies across all tasks 16

Figure 4.3 Heatmap displaying read and write frequencies across all memory locations for the Iris task 17

Figure 4.4 Heatmap displaying read and write frequencies across all memory locations for the Ionosphere task 18

Figure 4.5 Heatmap displaying read and write frequencies across all memory locations for the Breast Cancer task 19

Figure 4.6 Heatmap displaying read and write frequencies across all memory locations for the Heart task 19

Figure 4.7 Heatmap displaying read and write frequencies across all memory locations for the Glass task 20

Figure 4.8 Heatmap displaying read and write frequencies across all memory locations for the Tictactoe task 21

Figure 4.9 Violin plot illustrating accuracy scores of each task with and without indexed memory 24

Abstract

In this work, the performance of a Linear Genetic Programming (LGP) model in the context of classification tasks is assessed with and without the use of indexed memory. Using a virtual machine with a fixed instruction set, the model addresses various classification problems. Read and Write instructions are introduced to the model for accessing a single shared instance of memory. Ultimately, the incorporation of indexed memory not only allows the model to achieve a more diverse distribution of results, often corresponding to higher accuracy scores, but also reveals a significantly task-dependent relationship between the volume and frequency of memory accesses. Furthermore, the degree to which memory increases performance is also demonstrated to be highly task-dependent, emphasizing the nuanced impact of memory on the model's adaptability and effectiveness in diverse classification scenarios.

Acknowledgements

Thank you to Dr. Malcolm Heywood for providing me the opportunity to explore this field. I've learned so much throughout the course of this research, and I'm incredibly grateful for your continued support and guidance.

I would additionally like to thank the National Science and Engineering Research Council for providing funding for this work.

Lastly, I would like to thank my many dear friends who supported me throughout the duration of this research and, at times, had more confidence in me than I did. I appreciate you all so much.

Chapter 1

Introduction and Background

This thesis examines the effect of introducing a representation of indexed memory to a Linear Genetic Programming model designed to solve basic classification tasks. Prior to the examination of this research, it is essential to establish several conceptual paradigms. The following concepts were foundational to the experiments conducted in this study, and are crucial for understanding the implementation of Linear Genetic Programming that was used in this work.

1.1 Evolutionary Algorithms

Evolutionary algorithms (EAs) are a subsection of Machine Learning (ML) algorithms that draw inspiration from the biological concept of evolution in order to address a variety of problems [9]. Most notably, they employ aspects of 'survival of the fittest' and natural selection as described by Charles Darwin [4]. In EAs, a group of individuals comprise a population of candidate solutions to a given task [9]. These individuals are evaluated against some criterion to determine their "fitness" and thus decide their probability of reproduction. The fittest individuals in a population are selected to undergo the process of reproduction, during which various operators of mutation and recombination (crossover) are employed to introduce variation into the produced "offspring" [9]. Each iteration of this process can be referred to as a "generation." As a result, it is anticipated that the population will evolve towards an optimized solution that aligns with the desired objective.

Evolutionary algorithms have been used to solve a variety of decision and optimization problems. They have been applied in contexts such as data mining, image processing, and game strategy optimization.

In this work, EAs will be employed to solve basic classification tasks.

1.2 Genetic Programming

Genetic programming (GP) is an application of evolutionary algorithms that is specifically characterized by the evolution of computer programs. In GP, the initial population is comprised of randomly generated computer programs that aim to solve a given task when executed. Each individual's fitness is determined by examining the result of the execution of the program and evaluating its success against some predefined criteria. Then, a new population is created by genetically manipulating copies of existing programs and recombining their parts to create altered individuals [8].

There are several ways GP has been represented across previous works. The traditional method of representing GP, as defined by John Koza, is known as Tree-based genetic programming (TGP) [8]. In TGP, the evolution of a computer program is depicted using a tree data structure containing the parse tree of the program, with functions residing in inner nodes, and input values and constants comprising the leaves [3] [9]. Mutation and recombination is performed by the manipulation of subtrees.

Aside from Tree-based GP, other representations of GP have been documented, such as Linear GP and Graph-based GP [2]. For these experiments, Linear GP was used.

1.3 Linear GP

Linear GP (LGP) differs from TGP in that programs are depicted as sequences of instructions in an imperative programming language [3]. This concept can be likened to the execution of machine code on a CPU. Although the specific composition of these imperative instructions may vary across LGP implementations, as a general formula each imperative instruction contains an operation, operand(s), and a destination. Typically, it is assumed these instructions will be executed on some representation of a register-based machine [3]. The size of the set of registers allocated to a machine is user-defined. One or more of these registers are designated as "output registers" that will store the result of the program's execution. It is important to note that, given the ability to assign several output registers, it is possible for a Linear GP program to have multiple program outputs, whereas only one output is possible in Tree-based

GP [3].

1.4 Crossover

During the reproduction phase of a generation, one of the genetic operators that is applied to programs is crossover. Crossover entails selecting two "parent" programs, creating copies of them, and exchanging parts of each program with each other [2]. In Linear GP, this means selecting sequences of instructions from each parent and swapping them between each other. The length of the exchanged sequences may be fixed or randomly defined. The application of crossover

1.5 Mutation

Mutation is another genetic operator that is applied to programs during the reproduction phase. Unlike crossover, mutation operates on only one parent. Furthermore, mutation allows for material that is not present in either parent to be introduced into the population. This is distinct from crossover, which simply exchanges existing material between offspring. Typically, mutation is applied to individuals after they have already undergone crossover. Probability of mutation is user-defined, and may vary among types of mutation operations [2]. In Linear GP, mutation is applied by randomly selecting a single instruction from a program and altering its behaviour. This may mean changing the source or destination registers, changing the operation, or changing the addressing mode.

Specific details regarding mutation operators used in this work are explored in Chapter 2.

1.6 Fitness Sharing

Fitness sharing is a method of balancing the performance of evolutionary algorithms by encouraging diversity within the population. The basic principle of fitness sharing is that individuals may receive a greater reward for certain problems based on the perceived difficulty of the problem for the entire population. In the context of this work, programs that could correctly classify patterns that other programs failed to solve were favoured during the fitness evaluation. Rosin and Belew define a method wherein an individual's "simple fitness" (e.g. the number of correct classifications) is divided by the sum of the simple fitness scores of the rest of the population [10]. Thus, individuals receive a higher reward for classifying items that others could not. A similar function was used in this work, explored in Section 2.9. The objective of fitness sharing is to prevent a population from converging upon a single behaviour, and to promote unique solutions [10].

1.7 Memory

In this work, "memory" refers to a data structure that is separate from the virtual machines used in the Linear GP model. It is a collection, of user-defined size, of values that are accessible by index. Programs in the population may access the memory array using Read and Write instructions. Changes to the state of the memory array are preserved between generations. In other words, memory is not cleared as the population evolves. This is important as it introduces the possibility of "shared" values and inter-generational communication between programs, both of which are not naturally present in traditional implementations of LGP.

Chapter 2

Experimental Environment

2.1 Tasks

Prior to exploring into the experimental setup, it is crucial to establish a clear understanding of the model's designated tasks. The problems addressed by the model involve categorizing individuals within a dataset into classes based on their input values. When selecting tasks for this study, it was crucial to ensure sufficient diversity to test the model's effectiveness across various classification contexts. Some datasets featured binary outputs; the classification was a simple 'yes' or 'no.' Others had several outputs, meaning the model made a distinct prediction from a range of possible classes for each input. Furthermore, the number of features varied across tasks, ranging from as few as 4 to as many as 34. A total of six different tasks were attempted in these experiments:

Iris Dataset: This task involves classifying types of irises based on their petal and sepal measurements [5]. It contains four features, and has three possible outputs. There are 150 examplars, evenly distributed among the three categories. This dataset is widely used for machine learning tasks, and is largely considered to be an 'easy' classification task.

Ionosphere Dataset: With 34 features, this dataset contains the highest number of any other task chosen. It is a binary classification task that determines whether radar data collected from the earth's ionosphere can be considered 'good' or 'bad' [11]. It was chosen for these experiments because of its large number of features.

Breast Cancer Dataset: Like the Ionosphere dataset, this binary classification task is characterized by its extensive feature set, comprising 30 features. It contains data derived from images of breast masses and categorizes them as either 'malignant' or 'benign' [12].

Heart Dataset: Of the multi-output tasks, the Heart Disease dataset contains the most features, with 13. The features consist of a variety of health data with the intention of predicting the presence of heart disease in a patient. Its five outputs indicate the severity of heart disease, from 0 (not present) to 4 (significant presence) [7]. It was chosen for this study because of its high feature count for a multi-output task.

Glass Dataset: This task was selected for its high output count of seven. The dataset uses nine features to define types of glass in terms of their oxide content, and categorizes them into seven groups [6]. Benchmarking data for this task varies significantly, indicating that this task can be considered more 'difficult' than others.

Tictactoe Dataset: This binary classification task enumerates configurations of a game of tictactoe where 'x' is assumed to play first. The classification is 'true' when x wins, and false otherwise. It contains 958 exemplars, which is almost double the size of the next largest dataset used, with 569. Additionally, it is considered to be more of a 'difficult' task, based on benchmarking data from other machine learning models [1].

2.2 Virtual Machines

This implementation of LGP first established Virtual Machines (VMs) capable of executing groups of instructions (i.e. programs) in an imperative programming language. Within each VM, an array of floating-point values was used to represent the array of "registers." When choosing a size for a register array, it is necessary to ensure that the number of registers is at least as large as the maximum number of outputs for any given task. This is because the state of the registers is used to determine the result of the classification: each output corresponds to a register, and the highest value indicates the "result" of the classification function. A register array larger than the number of outputs may be used, and will improve performance in many cases. Thus, the ideal number of registers is variable for different tasks.

A single array size of 8 was selected for these experiments. This decision aimed to maintain consistency across tasks, ensuring that additional alterations to the simulation would not obscure the impact of adding or removing indexed memory. This particular size was selected because it meets the criteria of being at least as large as the number of outputs for any task attempted, and demonstrated reasonable performance in all cases. At initialization, all register values are set to 0.

2.3 Patterns

The VM has access to a single data point (pattern) at a time while a program is executing. These patterns are retrieved sequentially from an input file, and each program completes execution once for every pattern in the dataset. Register values are reset to 0 between executions, although values written to memory persist.

2.4 Instructions

Each instruction is represented as a group of four parameters: Mode, Target, Opcode, and Source. These parameters describe the operations that are carried out on the values present in the VM's registers, and dictate where the result should be stored. Each field is represented by a single integer, and the instruction as a whole is represented as an array of its components.

The 'Target' parameter holds the index of the register used to store the result of the instruction, and 'Source' holds the index of the second operand of the instruction, should it be necessary. The 'Opcode' identifies the operation that is to be performed on the two values. The 'Mode' parameter indicates the addressing mode. The model supports two addressing modes: "register addressing" and "pattern addressing." In register addressing, the model retrieves a value directly from the register indicated by the Source parameter. Alternatively, in pattern addressing, the model retrieves a value from the current pattern being tested at the specified index.

Thus, an instruction using register-addressing mode would produce the following effect:

```
Registers[Target] <-- Registers[Target] <Operation> Registers[Source]
```

Similarly, an instruction using pattern addressing:

```
Registers[Target] <-- Registers[Target] <Operation> Datum[Source]
```

2.5 Program Generation

Upon initialization, the model generates a population of 100 random programs, each comprising a random number of instructions. The number of instructions per program is within the range of `n*2` to `n*n`, where `n` is the number of registers. Individual parameters are randomly generated and concatenated into a single line, forming the array of instructions. This process is repeated 100 times until a full population has been generated.

2.6 Instruction Set

For these experiments, the base instruction set consisted of four primitives: add, subtract, multiply by two, and divide by two. Multiplication and division operators were limited for simplicity and computational efficiency. Since multiplication and division do not require a second operand, their execution will produce the following effect on the registers:

```
Registers[Target] <-- Registers[Target] * 2
```

or

```
Registers[Target] <-- Registers[Target] / 2
```

2.6.1 Read and Write

When memory was enabled, the model added Read and Write instructions to the pool of valid opcodes generated at the initialization stage.

Read instructions were executed according to the following structure:

```
Registers[Target] <-- Memory[Source]
```

Write instructions are performed as:

```
Memory[Target] <-- Source
```

Where "Source" is either Registers[Source] or Datum[Source], depending on the Mode.

2.7 Memory Structure

Memory for the model is simply represented as an array of 100 floating point values. At the beginning of a simulation, each position in the array is initialized with a value of 0. As the execution progresses, values will be written to and read from memory. Unlike registers, the memory retains its contents between patterns and persists through successive generations during the evolutionary progression of the simulation.

2.8 Parsing and Execution

Once the population of programs is randomly generated, the VM proceeds to execute each program on every pattern within the dataset. The simulation initiates by selecting a pattern and a program and subsequently iterating through each instruction within the chosen program. The registers will be manipulated according to the four parameters of each instruction, and at the end of execution, the state of the registers will be examined for classification. The last n registers of the register array are considered "output registers," where n is the number of outputs for the current task. The output registers are then analyzed separately to determine which of them contains the greatest value. The register that contains the maximum value corresponds to the output that was selected by the execution of the program. For example, if the classification registers were in the following state:

```
[0.34, 0, 6.45]
```

The index of the maximum value (6.45) is 2, indicating that output 2 was selected. This result is compared with the expected output for the selected pattern, and if the program's classification aligns with the anticipated outcome, the program receives a reward.

2.9 Reward

When a program correctly classifies a pattern, it receives a reward. Since this model makes use of fitness sharing, a program's fitness is discounted relative to the performance of every other program on each pattern in the dataset before it is finalized. The formula used to calculate fitness sharing for these experiments was as follows:

The initial reward, G, for an individual program g_i in the population being executed on an individual pattern p_k in the dataset can be described by:

$$G(g_i, p_k) = \begin{cases} 1 & \text{if } g_i \text{ classifies } p_k \text{ correctly} \\ 0 & \text{otherwise} \end{cases}$$

This binary value is then discounted by the performance of the rest of the population on the same pattern. If a particular pattern is labelled correctly by the entire population, then the classification of that pattern is not considered particularly 'valuable' behaviour. To encourage diversity in the population, the highest reward is granted when a program correctly labels a pattern that few others can. The discounted reward P is given by

$$P_{ik} = \frac{G(g_i, p_k)}{\sum_{j=1}^{n} G(g_j, p_k)}$$

where n is the number of programs in the population.

The final fitness score F_i of the program is acquired by summing the discounted reward for each pattern in the dataset, denoted by m.

$$F_i = \sum_{k=1}^{m} P_{ik}$$

After calculating the final fitness sharing score for each individual program, the population is sorted in descending order based on this value. Thus, the program with the highest fitness sharing score is considered the 'best' program in the current population.

Chapter 3

Experiments

Now that it has been established how the simulation generates and executes programs, the next step is examining the experiments themselves. Two experiments were conducted: one where the model was given access to an indexed memory structure and one where it was not. The following section will discuss the details of how each experiment was executed.

3.1 Evolution

In these experiments, one 'run' of the program included 300 generations of evolution before a 'champion' was selected. A generation consisted of executing each program in the population on each pattern in the dataset once, followed by a cycle of cloning and mutation. The result was a new population for the subsequent generation.

To create a new generation of programs, a subset of the highest performing individuals was chosen to 'breed' new individuals. This process commenced by selecting an eligible subset of 'parent' programs, determined by a 'gap percent.' The gap percent indicates the percentile of the population above which a program is considered eligible to be used for breeding. The lowest-performing individuals, those whose scores fell outside the range of the gap percent, were discarded in favour of newly generated 'children.' Each child underwent a mutation process wherein it had a chance of being altered, thus introducing new behaviours than those of its parent. This approach maintained a consistent population size while introducing novel behaviors in each generation.

For these experiments, a gap percent of 80 was used. Since the population size was 100, this meant that the bottom 20 programs were discarded at each generation. Then, 20 'parents' were randomly selected and duplicated from the surviving portion of the population. These programs then underwent a mutation process before being reintroduced to the population.

3.1.1 Mutation

During this phase, the children had the chance to be altered by crossover and mutation. Each mutation operator had a 'threshold value' indicating the probability of that mutation being applied. Crossover was always applied in each generation. The details of the five genetic operators used in these experiments are as follows:

Crossover: A random subsection of four consecutive instructions from one program are exchanged with another. A threshold value was not used for crossover during these experiments. Instead, crossover was always performed 30 times per generation, with each call selecting two random programs to exchange instructions. Because of this, it was possible that the same programs were selected to undergo crossover more than once, and some not at all.

Replace Instruction: One randomly selected instruction in a program is replaced with a newly generated instruction. Threshold value: 0.7.

Remove Instruction: One randomly selected instruction is deleted from a program. Threshold value: 0.5.

Add Two Instructions: Two newly generated instructions are inserted at a randomly selected index This mutation operator is particularly significant because it allows programs to increase in length. Increasing the length of a program may increase its computational complexity. Threshold value: 0.5.

Modify Parameter: One randomly selected parameter of a randomly selected instruction is regenerated. Any of the four parameters (Mode, Target, Opcode, or Source) is eligible to be replaced. Threshold value: 0.7.

3.2 Champion Selection

After the model has finished its full 'run' of 300 generations, a champion must be selected from the population. While fitness sharing scores were the metric for selection during training, the final champion was chosen based on accuracy. This is because once the model is fully trained, evaluating programs in terms of their performance relative to their peers becomes less relevant, and their overall success on the task takes precedence.

To select champions from the population, each program was executed repeatedly on test partition of the dataset and its average performance was determined. In the case of the memory-enabled experiment, to initiate this process, the state of memory after training was preserved. The post-training state of memory was then used as the initial state of memory for each program during the champion selection phase. This was to ensure that any stored values 'used' by the programs during training remained available to them while they were executed on the test partition. Each program was executed 20 times on the test partition and tested for raw accuracy (i.e. number of correct classification). The individual with the best average accuracy after 20 executions was selected as the overall champion of that run.

3.3 Expectations

If memory is enabled on the model, some subset of each program's instructions will involve interacting with memory. In theory, this will allow the program to store useful data statically for later access. The definition of 'useful' is task-dependent, but some possibilities include preserving significant features or storing constants beneficial to calculations. It was hypothesized that high-performing programs would develop the behaviour of storing and subsequently reading values that were found to be beneficial to the task.

Since the programs are generated randomly, it was expected that not every read or write to memory would be deemed 'useful' to the execution of the program. However, it was anticipated that programs that generated some degree of useful behaviour would be favoured during the selection process, resulting in a population that could be said to use memory in a meaningful way.

Since memory persisted between generations during training, there was also a possibility that programs would find success by reading values that had been previously written to memory by other programs. This was acknowledged as acceptable behaviour for the model.

Chapter 4

Results

4.1 Memory Usage

After the model had been trained, several patterns emerged from examining how the champions interacted with memory. After the champions had been selected, their instructions were examined to determine the frequency and locations of memory accesses. The resulting data was visualized through a heat map of indexed memory, illustrating the accessed locations and their corresponding frequencies. Interestingly, the patterns that emerged in the memory accesses were vastly different across each task.

There were, however, several elements that appeared to be consistent throughout each task. Notably, each task displays a prominent spike in memory accesses at index 0, which can be attributed to the initial state of each VM when programs are executed. Given that register values are initialized with a value of 0, any early read or write instruction in the program's execution naturally targets index 0. As registers gradually accumulate values from patterns and computation results, memory accesses to other indices become more prevalent.

It was also observed that many tasks tended to exhibit a higher frequency of memory access in the initial and final 10 locations in memory. This behaviour was more pronounced in some tasks than in others. Figures 4.1 and 4.2 visually depict this pattern, revealing that five out of the six attempted tasks exhibit a discernible 'hot spot' near the beginning and end of the memory pool.

In the following sections, each task will be explored individually in order to highlight specific patterns and behaviours that appeared during the experiments.

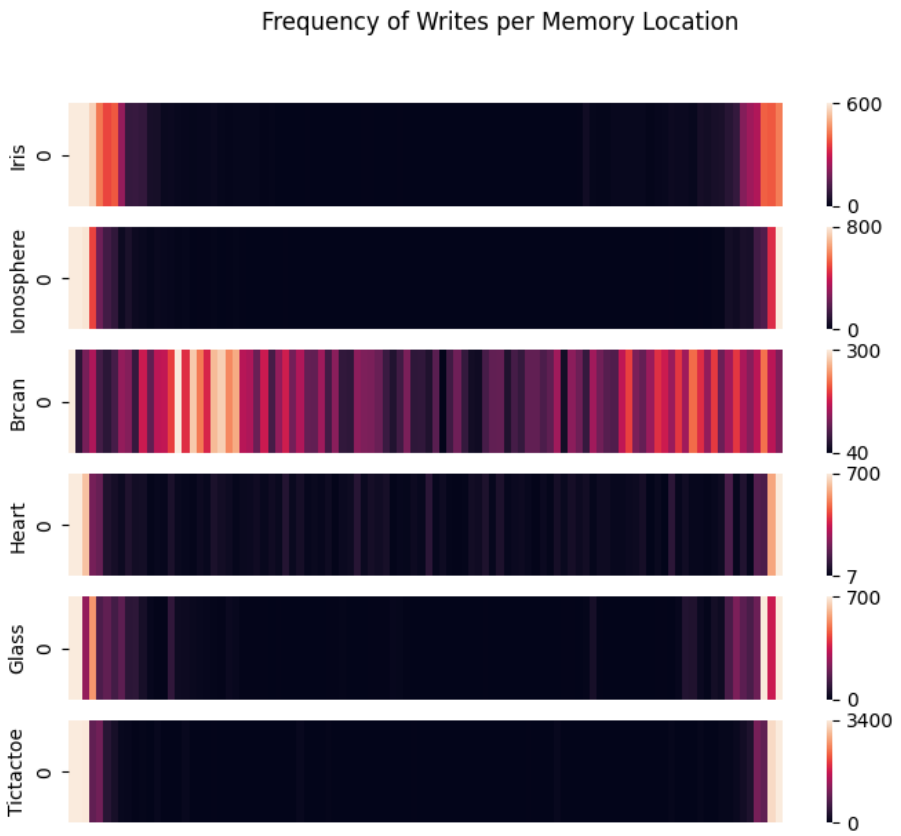

Figure 4.1: Heatmap of write frequencies across all tasks

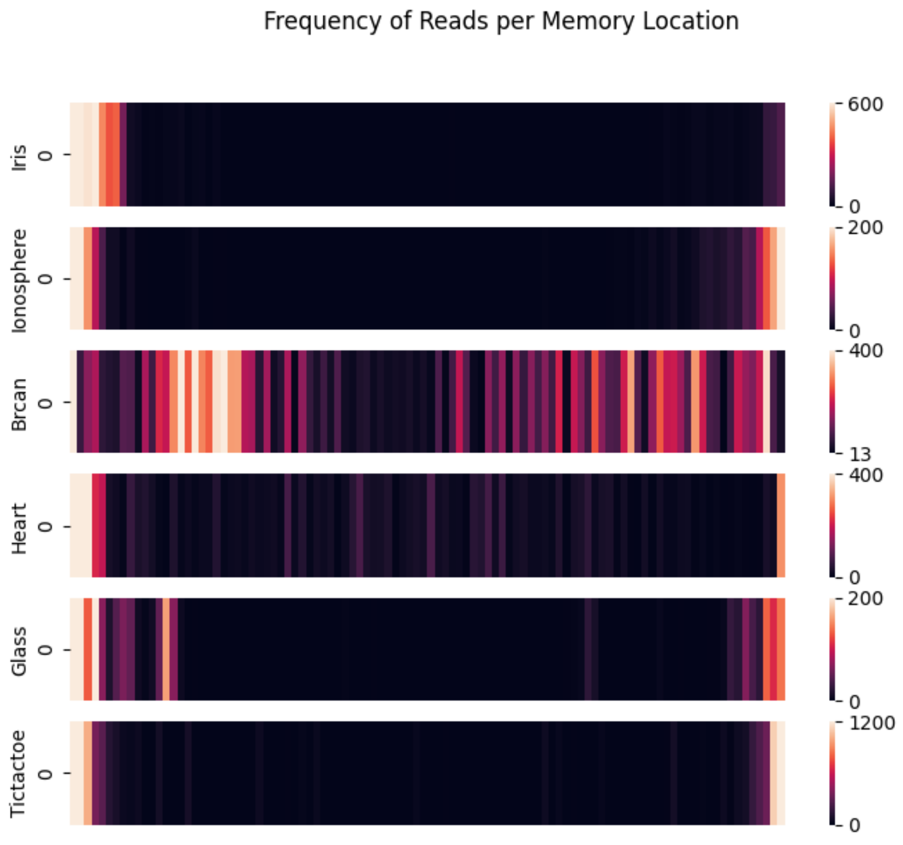

Figure 4.2: Heatmap of read frequencies across all tasks

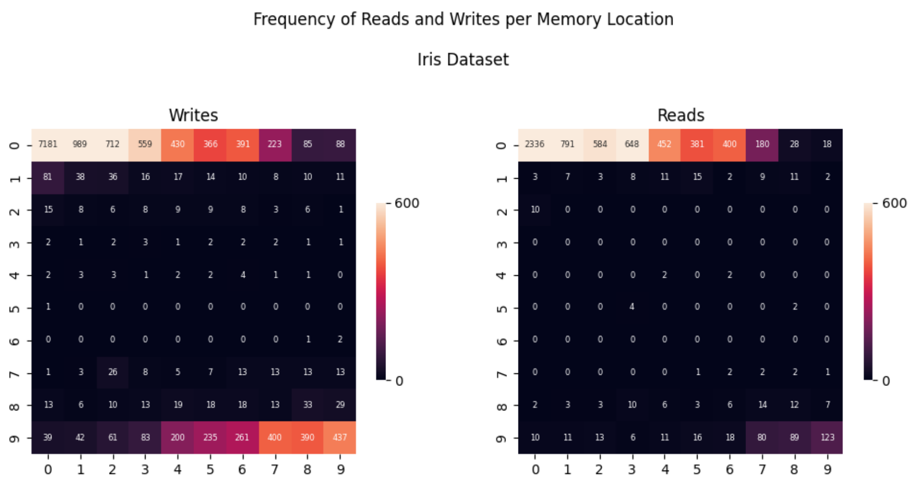

Figure 4.3: Heatmap displaying read and write frequencies across all memory locations for the Iris task

4.1.1 Iris Dataset

The Iris task (found in Figure 4.3) displayed the aforementioned pattern of pronounced memory usage concentrated in the initial and final 10 memory slots. Notably, there was very little memory usage outside of those locations. It is important to note that the distribution of reads and writes is asymmetrical; the champions had a tendency to write values more frequently than they were read.

4.1.2 Ionosphere Dataset

The pattern of memory usage in the Ionosphere task (Figure 4.4) closely resembles that of the Iris task, characterized by a substantial frequency of both reads and writes at the beginning and end of memory, with fewer accesses in the middle. The majority of memory accesses were most concentrated within the first and last five spaces of memory. Like the Iris task, champions of this task also performed writes at a much higher frequency than reads.

4.1.3 Breast Cancer Dataset

The Breast Cancer task (Figure 4.5) exhibited the most extensive use of memory of any other task. It is important to highlight that the location of the hot spots in the

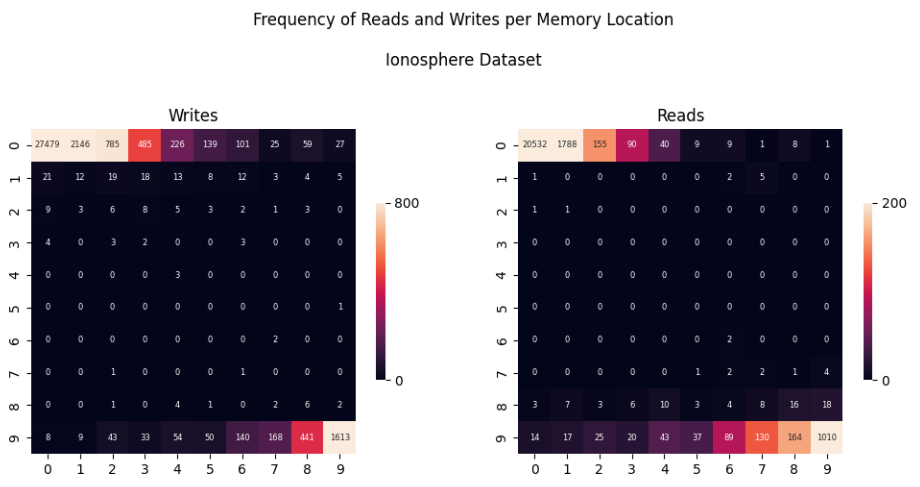

Figure 4.4: Heatmap displaying read and write frequencies across all memory locations for the Ionosphere task

read and write data indicate that the champions were accessing the same locations with a high frequency for both operations. This illustrates some evidence of the model storing and later accessing values deemed beneficial. Furthermore, the concentration of read operations exceeded that of write operations in several locations. This could potentially indicate the model's behavior of establishing constants that are frequently read from memory to enhance the classification process.

4.1.4 Heart Dataset

The pattern of memory accesses in the Heart task (Figure 4.6) exhibits similarities with the previously examined tasks. Like Iris and Ionosphere, there are noticeable hot spots near the beginning and end of memory. And, like Ionosphere, this task displayed significant activity across a majority of memory addresses. Furthermore, there is evidence of hot spots appearing in the same locations for both read and write operations, indicating some level of association.

4.1.5 Glass Dataset

Interestingly, the Glass task (Figure 4.7) displayed distinctive asymmetry between read and write patterns. Several 'read' hot spots were observed in locations that were

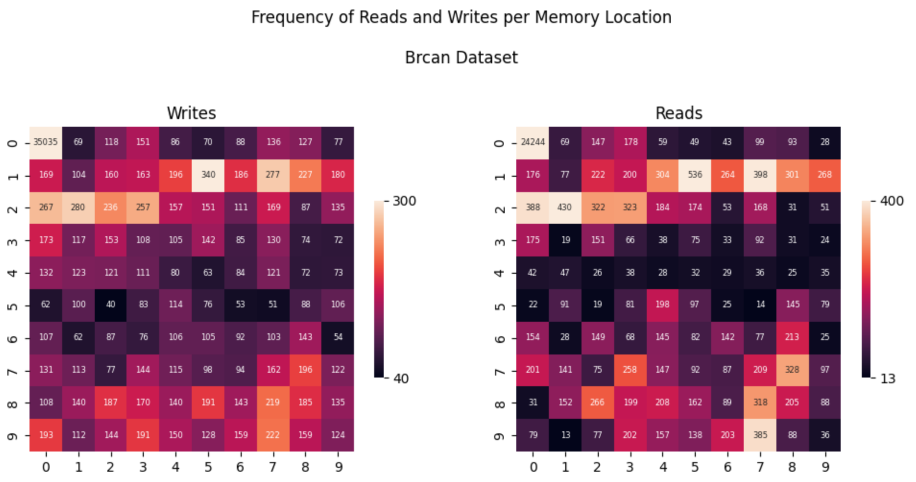

Figure 4.5: Heatmap displaying read and write frequencies across all memory locations for the Breast Cancer task

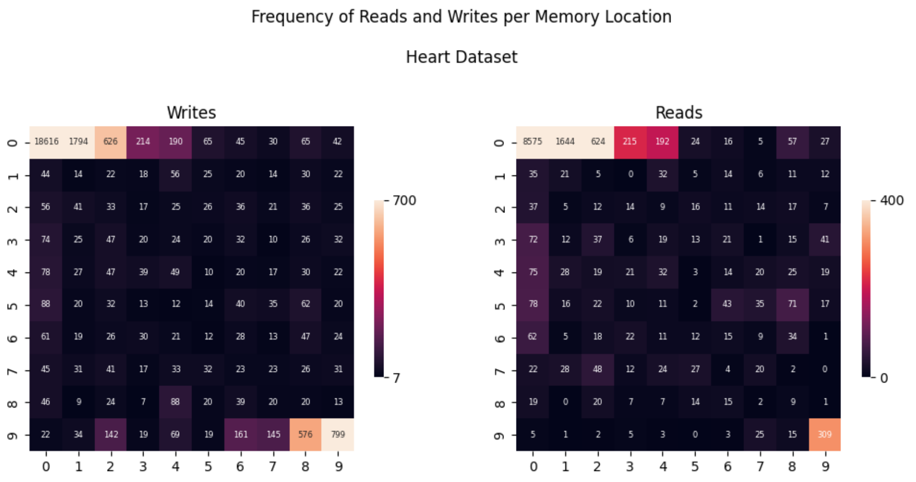

Figure 4.6: Heatmap displaying read and write frequencies across all memory locations for the Heart task

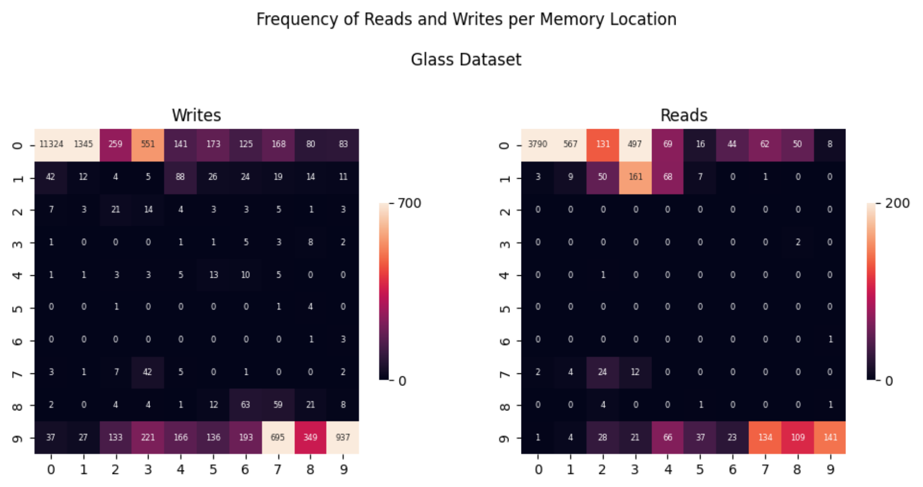

Figure 4.7: Heatmap displaying read and write frequencies across all memory locations for the Glass task

written to very infrequently. This behaviour is interesting because it implies that programs frequently accessing those locations were favoured despite the fact that the value seldom changed.

4.1.6 Tictactoe Dataset

The Tictactoe task (Figure 4.8) followed the pattern observed in several other tasks and displayed a preference for values at the beginning and end of memory. Notably, in this instance, the magnitude of memory accesses in that range was particularly striking. The memory locations within the initial and final 5 slots experienced remarkably high-frequency access, ranging from triple digits to quintuple figures. The middle portion of memory was mostly neglected. As seen in several other tasks, the number of write operations exceeded the number of reads.

4.2 Performance

In terms of performance (number of correct classifications), there were several trends that could be observed across all tasks. These trends can be seen in Figure 4.9, which displays the distribution of accuracy scores among the champions of both experiments. Most consistently, the introduction of indexed memory created a wider distribution

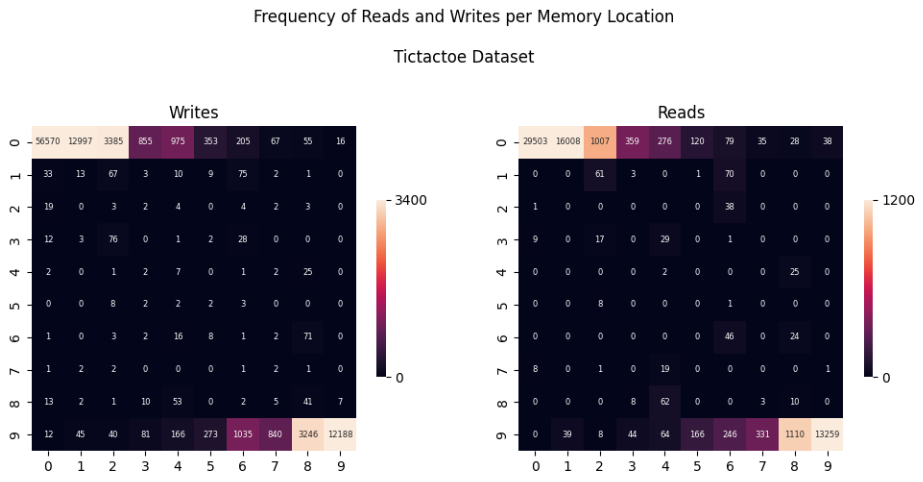

Figure 4.8: Heatmap displaying read and write frequencies across all memory locations for the Tictactoe task

of performance scores across the population of champions. Where the experiments without memory tended to cluster in certain niches, those with memory demonstrated more diverse characteristics. In all but two cases, the highest performing champion in the indexed-memory pool achieved a higher accuracy score than the corresponding champion that did not use memory. Of the two cases where this observation does not hold, one of them achieved 100 percent accuracy without memory and thus could not be improved further. Similarly, in all cases except two, the worst performing champion in the indexed-memory pool had a lower accuracy score than the worst of the no-memory pool.

The following sections will outline the performance results of each task individually, to highlight unique and noteworthy behaviours.

4.2.1 Iris Dataset

For this task, the vast majority of champions across both pools were able to achieve 100 percent accuracy on the test data partition. This is unique to this task; there were no others that achieved similar levels of performance.

4.2.2 Ionosphere Dataset

The introduction of memory notably improved performance in this task. Leading champions in the memory pool achieved an approximate accuracy of 90 percent, with a significant portion of the remaining distribution scoring around 80 percent. Notably, a substantial section of this pool surpassed the highest-performing individual in the pool without memory. In the absence of indexed memory, the majority of the champions scored below 80 percent, with the highest reaching approximately 82 percent. This was one of two tasks where the lower bound of performance was raised following the introduction of indexed memory. This task saw the most improvement with the introduction of indexed memory.

4.2.3 Breast Cancer Dataset

Similar to the Ionosphere task, the introduction of memory demonstrated an impact on both the upper and lower performance bounds for the Breast Cancer dataset. However, the magnitude of improvement was less pronounced compared to the aforementioned task. The top performers in the memory-enabled pool attained an accuracy of approximately 93 percent, while their counterparts in the no-memory pool achieved roughly 92 percent. Interestingly, the distributions are roughly the same width in both pools, with similar shaped curves characterizing their distributions.

4.2.4 Heart Dataset

In this problem, the distribution was significantly wider after the introduction of indexed memory. As observed in most other tasks, the top performers within the memory-enabled pool surpassed their counterparts in the memory-disabled pool. However, the number of individuals who achieved this was significantly smaller for this task. It is noteworthy in this case that the pool without memory had a higher average performance than the memory pool. Furthermore, the least successful individuals in the memory-enabled pool demonstrated significantly lower performance than the lowest performers in the no-memory pool, achieving only around 43 percent accuracy. The distribution in the no-memory pool was quite narrow, however, which may indicate a low level of diversity between champions.

4.2.5 Glass Dataset

Excluding the Iris task, which achieved 100 percent accuracy, the Glass dataset stood out as the only task where the introduction of indexed memory failed to enhance the upper limit of accuracy scores among the champions. Instead, the highest performing champions in the memory pool achieved the same accuracy as their counterparts in the opposite group. The distribution of accuracy scores was much wider in the memory-enabled experiment, with the lowest performers achieving approximately only 30 percent accuracy. This behavior is not entirely unexpected, considering that the Glass task is generally recognized as more challenging.

4.2.6 Tictactoe Dataset

Like the Ionosphere task, the Tictactoe task appeared to receive a significant boost in performance from the addition of indexed memory. The highest-performing champions in the memory pool achieved significantly higher accuracy scores than their no-memory counterparts. However, like the Heart task, the lower bound of performance was also extended by the addition of memory.

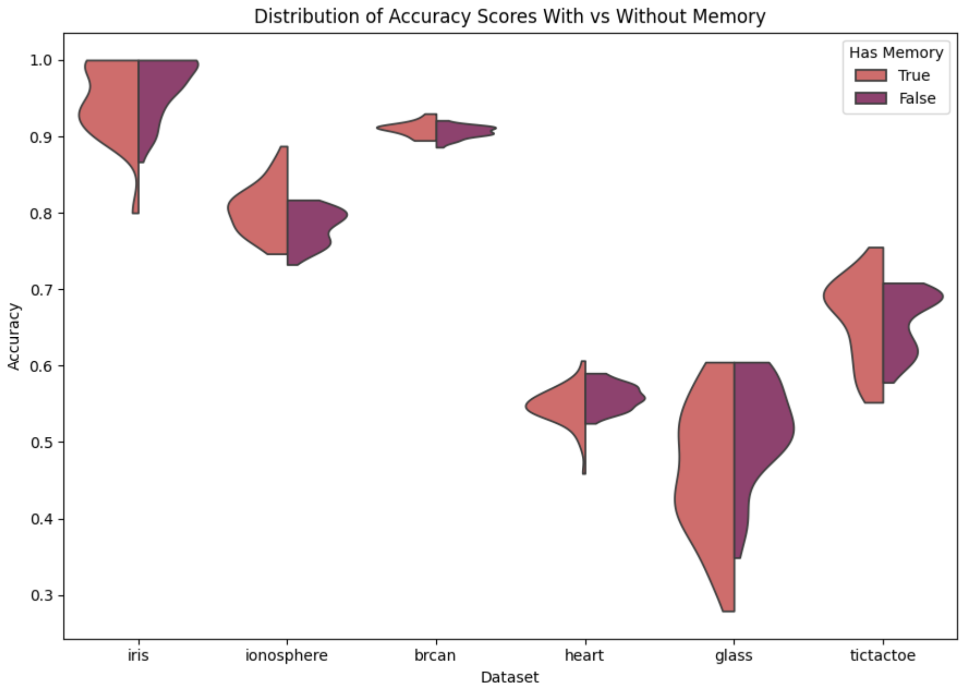

Figure 4.9: Violin plot illustrating accuracy scores of each task with and without indexed memory

Chapter 5

Analysis

The results of the two experiments provide a variety of insights about the intricate relationship between Linear GP and memory. Each task utilized memory in unique ways, though several similarities shone through. This has some implications about how memory was used for each task, as well as about the adaptability of the model. Additionally, the exploration of the model's performance during the experiments revealed several patterns and provided insights into the influence of memory on the model.

5.1 Memory Usage

The examination of memory access patterns revealed intriguing insights into the utilization of memory across different tasks. One of the more interesting observations was that some tasks, although very different in nature, displayed similar behaviours in terms of memory access, notably Iris and Ionosphere. These two tasks are very different; one is a small, balanced multi-output task with only four features, and the other is a binary classification task with more features than any other task examined. Yet, both tasks exhibited a pattern of a high frequency of reads and writes at either end of the memory array, with minimal activity in between.

Furthermore, both demonstrated a higher overall concentration of writes compared to reads. This behaviour was not uncommon; it was noted also in the Tictactoe, Glass, and Heart tasks. This may indicate that the programs are frequently writing values to memory that are not subsequently used. Since performing a 'write' operation does not alter the state of the registers, it has no immediate effect on the program when executed. If the value written is not later referenced by the program, then that write instruction can be considered effectively useless. Nevertheless, there is no penalty for performing this action. Since there is no performance impact, programs that exhibit this behaviour along with other beneficial behaviours are likely to remain in the population during the selection process. The result of this is a population of programs who perform write operations more often than they perform reads.

In contrast, reading a value from memory has an immediate impact on the registers and, consequently, the program's execution path. As a result, programs that contain excessive 'bad' reads are more likely to be eliminated during the selection process. Theoretically, the ideal behavior for a model using indexed memory is to write crucial values to memory and subsequently retrieve them during execution. There was some evidence of this behaviour throughout the tasks, with Breast Cancer being the most notable example. Breast Cancer stood out as the sole task where the frequency of read operations surpassed writes. Moreover, the hot spots in the data suggested that the programs were frequently reading from the same indices that were being written to.

Though there were indeed some similarities among the memory access patterns of the tasks, each task ultimately exhibited its own unique characteristics. This highlights the key observation derived from these experiments: the model's adaptability to the specific task at hand. Throughout the evolution process, the survival of the most beneficial behaviors, including memory operations, shapes the resulting population. The result of this is a population that uses memory in a way that best suits the task being attempted. Whether it involves utilizing only a few provided memory slots or extensively exploiting the entire array depends entirely on what is demonstrated to be most effective during the evolution process.

5.2 Performance

Most tasks saw some degree of improvement with the introduction of indexed memory. Those that did not display an explicit improvement had their peak performance matched. Across all cases, the introduction of indexed memory led to a broader distribution of accuracy scores, potentially indicating an increased diversity within the populations.

Interestingly, all three of the binary classfication tasks, Ionosphere, Breast Cancer, and Tictactoe, appeared to derive the greatest benefit from the addition of indexed memory. This observation may suggest that indexed memory may be particularly effective in improving the performance of binary classification tasks compared to other contexts.

The examination of the performance of the model with and without indexed memory further solidifies the notion that the impact of memory is highly task-dependent.

Chapter 6

Conclusion

The most important observation that was produced from these experiments was that the effect of memory on Linear GP varies significantly based on the attempted task. In the binary classification tasks attempted during this study, introducing indexed memory was demonstrated to improve performance significantly. Across all tasks, introducing indexed memory was able to at least match the performance of models without memory.

Across all tasks, the utilization of memory was unique and custom to the specific requirements of the task. The evolutionary model is capable of converging on a way to utilize memory that is most beneficial to its current task. The result of this is a significant variance in the way memory is utilized for each task.

Ultimately, the behaviour of the model underscores the task-specific nature of memory utilization, with each task manifesting a unique and discernible approach aligned with its inherent computational demands in order to produce the best results.

6.1 Further Research

The scope of this study was confined to investigating the impact of enabling indexed memory on performance and analyzing the location and frequency of memory usage. This illuminated the ability of the model to tailor its memory usage to specific contexts while maintaining positive performance outcomes. However, the observed patterns in memory usage data open up various potential avenues for further research.

One such avenue involves delving deeper into the values written to and retrieved from memory. Additional tracing of the execution path could reveal the flow of data through the instructions in each program, potentially revealing the effects of each operation. This would provide richer insights about how the model utilizes memory to accomplish the given task.

Another potential direction for further research is the observed imbalance between read and write instructions. Throughout the experiments, a prevalent pattern emerged where models exhibited a higher frequency of write instructions compared to read instructions. This suggests that many of these instructions may introduce values into memory that are not subsequently accessed. It would be worthwhile to investigate the degree to which these seemingly unused write instructions impact the performance of the model, and if there is a way to prevent their occurrence.

In addition, a compelling area for further exploration involves understanding how tasks influence the model. While a diverse range of tasks was chosen for these experiments to assess the model's performance across various contexts, there is an opportunity to delve deeper into the specific characteristics of each task that influence the model's behavior in terms of memory usage.

In summation, the results of this study may serve as valuable stepping stones towards further studies in this field. The implications of indexed memory on Linear GP present a rich opportunity for further exploration.

Bibliography

[1] David Aha. Tic-Tac-Toe Endgame. UCI Machine Learning Repository, 1991. DOI: https://doi.org/10.24432/C5688J.

[2] Wolfgang Banzhaf, Peter Nordin, Robert Keller, and Frank Francone. *Genetic Programming: An Introduction on the Automatic Evolution of computer programs and its Applications*, pages 122–127. Morgan Kaufmann Publishers Inc., 01 1998.

[3] Markus F. Brameier and Wolfgang Banzhaf. *Linear Genetic Programming*. Springer New York, NY, 2006.

[4] Charles Darwin. *On the Origin of Species by Means of Natural Selection*. Murray, London, 1859. or the Preservation of Favored Races in the Struggle for Life.

[5] R. A. Fisher. Iris. UCI Machine Learning Repository, 1988. DOI: https://doi.org/10.24432/C56C76.

[6] B. German. Glass Identification. UCI Machine Learning Repository, 1987. DOI: https://doi.org/10.24432/C5WW2P.

[7] Andras Janosi, William Steinbrunn, Matthias Pfisterer, and Robert Detrano. Heart Disease. UCI Machine Learning Repository, 1988. DOI: https://doi.org/10.24432/C52P4X.

[8] John R. Koza. Genetic programming - on the programming of computers by means of natural selection. In *Complex Adaptive Systems*, 1993.

[9] Peter Nordin. *Evolutionary Program Induction of Binary Machine Code and its Applications*. PhD thesis, der Universitat Dortmund am Fachereich Informatik, Germany, 1997.

[10] Christopher D. Rosin and Richard K. Belew. Methods for competitive coevolution: Finding opponents worth beating. In *Proceedings of the 6th International Conference on Genetic Algorithms*, page 373–381, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[11] V. Sigillito, S. Wing, L. Hutton, and K. Baker. Ionosphere. UCI Machine Learning Repository, 1989. DOI: https://doi.org/10.24432/C5W01B.

[12] William Wolberg, Olvi Mangasarian, Nick Street, and W. Street. Breast Cancer Wisconsin (Diagnostic). UCI Machine Learning Repository, 1995. DOI: https://doi.org/10.24432/C5DW2B.