BENCHMARKING MODULAR GENETIC PROGRAMMING ON DEEP MEMORY TASKS

by

Mihyar Al Masalma

Submitted in partial fulfillment of the requirements
for the degree of Master of Computer Science

at

Dalhousie University
Halifax, Nova Scotia
March 2022

To my father, this work is a result of your passion for knowledge that you seeded in us.

Table of Contents

List of Tables

List of Figures

Abstract

Partially observable tasks require a learning agent to make decisions based on the previous state, hence a requirement for memory. There is a trade-off between the flexibility and specificity of the memory. This impacts the ability of the agent to solve specific tasks versus generalize to a range of tasks. Recently, a suite of 'deep memory tasks' was proposed to evaluate different approaches to partially observable problems. In this thesis, a canonical tree-structured genetic programming (GP) framework is assumed as the starting point, with memory taking the form of a list. The interface to memory requires that canonical GP is deployed as a modular co-evolutionary framework to support multiple outputs. An empirical evaluation is performed using three deep memory benchmarks to showcase the relative strength/weaknesses of this approach. We also compare our findings with neural solutions to distinguish between the relative contribution of GP versus list-based memory.

Acknowledgements

First and foremost, a huge thanks to Dr. Malcolm Heywood, my research supervisor. Plenty of idea were tried in the making of this thesis especially with the pandemic, it was comforting having Dr. Heywood with me during this journey.

To my family and friends who stuck by me through this interesting journey with patience and understanding, you've always pushed me to become a better version of myself and I hope I made you proud.

Chapter 1

Introduction

There are many different approaches to characterizing machine learning (ML) and the types of tasks to which they might be applied. One of the most widely assumed is from the perspective of the type of feedback available, leading to the concept of supervised, unsupervised, and reinforcement learning. Supervised learning implies that for each input (state) there is a known corresponding outcome (label). Typical examples of tasks that might be addressed using supervised learning are classification and regression (function approximation). In both cases, the underlying objective is to find some mapping from the input space to the label or function space such that a cost function is minimized. Unsupervised learning represents the subset of tasks for which there is no label information available, thus the underlying objective is to describe the original input only data in terms of a lower-dimensional representation, i.e. clustering. Again, after each input is presented, an update is performed such that some distance measure is minimized. The last category of a task, reinforcement learning, represents a decision-making problem in which the learning agent has to maximize the cumulative rewards over a (possibly infinite) sequence of state inputs. As such the action chosen by the learning agent at any point in time impacts what state the agent might experience next. Reinforcement learning tasks appear in applications such as making investments, decision making for autonomous agents, drones, cars or robots, and games. In this thesis, we are interested in reinforcement learning (RL).

When it comes to RL in particular, there are two types of problems that we might try to address, those where the description of state (input) provided by the task is complete and the second where state information is incomplete or partial. In the case of the first type of task, reactive learning agents are sufficient. This implies that an agent does not need to build a 'mental model' of the world (internal state) in order to make optimal decisions. Conversely, partially observable environments require the

agent to develop an internal state representation or memory in order to make optimal decisions.

An example of a task with complete state information would be the game of chess. This is to say that, all the information we need at any stage in the game, in order to make an optimal decision is available from the current configuration of the board. For tasks with partial information, we will need some sort of memory to keep track of past state values or what steps have we tried (in the past), and what results we have reached using such steps. An example of that would be card games, where you can only see cards that you hold and memorize past dealt cards. Another example would be first-person shooter games where the learning agent's view is limited by orientation and position relative to other objects in the environment, i.e. it is not possible to see through walls, trees, or out the back of the agent's body. We are interested in problems with partial information in this thesis.

RL tasks can also be subject to stochastic and deterministic properties. Deterministic tasks do not have a random component to the task, a good example would be the game of Chess. In Chess, we have a defined set of pieces, each of which has specific moves that can be performed and this will not change. Conversely, Stochastic tasks have a random component to them. A good example of that would be the game of poker where we do not know what is the next card to be dealt. Another example is games where dice are used, such as back-gammon, where the dice represent the source of randomization.

In general, if we wanted to invent a new Machine Learning (ML) algorithm we would need to answer three key questions: representation, credit assignment, cost function. Representation defines a language for expressing how the learning algorithm expresses possible solutions. Credit assignment defines how a possible solution can be modified once the performance is evaluated. The third question addresses the performance of a possible solution or a cost function. In our work we are interested in the specific case of Genetic Programming [17]. As such we see genetic programming addressing the previously mentioned questions in the following way:

- **Representation:** Genetic Programming (GP) assumes that solutions are described in terms of an a priori specified instruction set. In effect, we are conducting a combinatorial search over the set of instructions appearing in a candidate

solution in order to find solutions that optimize the performance function using the least number of instructions.

- **Credit Assignment:** represents the process by which the representation is manipulated. GP assumes that a 'population' of candidate solutions exists. Selection operates on the population to determine which solutions get to survive. Variation operators modify the 'parent' programs to produce 'offspring', typically through the application of crossover and mutation [17]. Finally, replacement defines how a new population is defined from the resulting parents and children, i.e. competition to survive in a fixed size population.

- **Performance Function:** is the metric used to rank the performance of individuals relative to each other. In the case of reinforcement learning tasks, it is typically defined in terms of maximizing the total reward experienced over a sequential set of interactions between the agent (a GP individual) and the environment. As such, the reward policy is task-specific.

In this thesis, we will assume that GP takes the general form of Koza's canonical tree-structured GP as implemented in DEAP [6]. In doing so, we can concentrate on the underlying issues of decision-making under incomplete information with varying amounts of stochastic noise.

1.1 Thesis Objectives

Canonical tree-structured GP is a reactive representation (no capacity for representing the previous state). In order to overcome such a limitation and work with partially observable tasks, we will take the approach of adding an external memory. Depending on how the external memory is designed, we might make the tasks more difficult or easier to solve. The agent will need to learn when to write and when to read to memory, given instructions that perform a write or read (from memory). Naturally, there are many approaches to designing the memory properties for GP. In this thesis, we develop our solution about the concept of a linked list.

Our second hypothesis relates to the instruction set an agent has access to. We assume that different instruction sets would have different effects on the agent's performance. We assume that having a small instruction set would lead to clear and

simple solutions on account of how quick or even how often a solution would be found. On the other hand, we assume having more diverse instructions set would allow for bigger search space and therefore agents would find a solution more often. This would come as a trade-off to speed and simplicity versus limiting the instruction set too much (i.e. not possible to find solutions).

The third hypothesis is related to how the task itself is defined. We assume that the interaction between task definition and instruction set has the potential to make the task easier or more difficult for the agent to solve. What we mean by that is, if a task has a clear definition in relation to the instruction set, then the agent will be able to find a solution to the given task very quickly. On the other hand, if the task definition has stochastic/noise properties, we anticipate that this will make it much more difficult for the agent to solve.

1.2 Thesis Outline

This thesis is structured as follows: In Chapter 2: Related Work, relevant topics are introduced to present prior work regarding the general approaches assumed for designing memory versus the flexibility of the resulting memory models. Specifically, we review developments from both neural networks and GP as well as from the perspective of internal versus external memory models.

Following that, in Chapter 3: List data structure with genetic programming, describes in detail our approach for external memory and the control signals that would be necessary for an agent to access and manipulate the memory. Also, we detail how to address the single output limitation of canonical tree-structured GP and the details of the evolutionary credit assignment process we are using.

Then in Chapter 4 Results, the different instruction sets used to evolve GP agents alongside the configuration file needed for the comparator neural evolutionary representation (NEAT) [27]. The benchmarking tasks and their results are then reported as follows:

- In section 4.2 Sequence Recall, describes the experiment performed and the results in both GP and NEAT compared. Results from different GP instruction sets are compared with each other, introducing different kinds of noise and

comparing results with the original GP setup, and the generalization properties of the original setup tested.

- In section 4.3 Sequence Classification, describes the task and experiments performed. Results for both GP and NEAT are compared. Results from different GP instruction sets are compared with each other, introducing different kinds of noise and comparing results with the original GP setup, and the generalization properties of the original setup tested.

- In section 4.4 Copy Task, describes the task and experiments performed. Results in both GP and NEAT are compared, with results from different GP instruction sets compared with each other. Different kinds of noise are introduced and compared, and the generalization properties of the original setup are tested.

Lastly, Chapter 5: Conclusion, wraps up the thesis with a summary of the thesis including directions in which future work can be done in this area.

Chapter 2

Related Work

In order to address partially observable state tasks, it is necessary that the representation support some form of memory (capable of expressing internal state). In the following, we provide a review of recent research of machine learning frameworks from the perspective of neural networks and genetic programming. Specifically, the recent developments in memory models for neural networks will be contrasted with the historical developments of memory in GP. This will then establish the motivation for using recent benchmarks from neural networks to assess the capability of GP augmented with an external memory structure; thus, forming the research question of this thesis.

2.1 Neural networks with Memory

Neural networks as used for machine learning tasks are often characterized in terms of layers of neurons in which the output(s) from one layer may only be connected to the inputs of the following layer (a feedforward architecture). This is sufficient for reactive/stateless tasks such as classification or regression. Memory (of the previous state) can be introduced by adopting connection topologies that allow the output of a neuron in layer l to appear as an input to neurons in the same or earlier layers.[1] Memory formed by recurrent connections tends to assume that all values appearing in the path of a recurrent connection should be retained. Under gradient-based credit assignment, this leads to pathologies such as vanishing versus exploding gradients [23]. As such, further developments from the neural network community attempt to address this shortcoming by adding additional controls. The most widely adopted model is that of the Long Short Term Memory (LSTM) [10]. LSTM supports additional functionality, such as gating the input to the recurrent feedback loop and enabling

[1] Implies that a unit delay that takes a value and reintroduce it in the next iteration is applied to such a recurrent connection.

the value of a recurrent connection to be reset.

Recently, Graves et al. introduced an alternative formulation for memory in neural networks called the 'Neural Turing Machine' (NTM) [7]. The motivation was based on the observation that although the distributed form of memory provided by recurrent connections are Turing-complete (may simulate arbitrary procedures), this is only realized if they are correctly connected. However, the connectivity patterns need to be pre-specified.[2] A NTM realizes memory by providing the neural network with an 'external memory' (synonymous with the indexed memory of a computer) that a neural network interacts with using a dedicated set of read and write operations. Graves et al. demonstrate that the proposed NTM is able to provide solutions to benchmark problems that defeat recurrent neural networks (LSTM).[3] The benchmarks included the copy task, finding n-grams, and sorting. Later research went on to generalize these results to planning tasks such as route discovery on the London underground [8].

Recurrent forms of memory are also present in frameworks for neuro-evolution. The NEAT (NeuroEvolution of Augmenting Topologies) paradigm represents a widely assumed approach, in part because the genotypic representation supports the application of sexual reproduction under a variable length representation [27]. Specifically, NEAT initiates evolution from a population of linear models (perceptrons) and incrementally increases complexity by adding (hidden) neurons and modifying the pattern of connections. Unlike gradient methods, neuro-evolutionary frameworks are therefore free to discover recurrent topologies specific to a particular problem. Indeed, the NEAT framework has been widely applied to tasks with temporal properties. The following results are identified as being significant to this thesis as they contrast and compare outcomes using a neuro-evolutionary approach to results from the above mentioned Neural Turing Machine:

- NEAT with LSTM [22]: provides NEAT with the LSTM as an atomic data structure for incorporating into neural networks as evolved by the NEAT framework. In addition, in order to scale the approach to tasks with increasing memory

[2] Gradient methods for credit assignment are good for optimizing weights within fixed topologies, not discovering the topology connectivity itself.

[3] Indeed, feedforward network, LSTM, and external memory can all be used in a single architecture for maximum performance [7].

'depth', NEAT is rewarded for evolving neural network topologies that maximize the information stored in the LSTM data structures. Benchmarking is performed using the sequence classification and recall tasks (representing tasks that can be scaled to different memory durations) with success rates in the order of $\approx 25\%$ on the 5-depth sequence recall task and 6-depth sequence classification task.

- Evolving Neural Turing Machines [9]: demonstrated that adopting a neuro-evolutionary approach to credit assignment enabled the interface to external memory to be substantially simplified. Moreover, the topology of the controlling neural network could also be directly evolved. Empirical evaluation on the Copy Task demonstrated generalization under much deeper tasks than demonstrated with NTM, with zero error and a solution identified with a single neuron (as opposed to the 100 neurons appearing in the NTM solution).

- Modular Memory Units [16]: proposes a specific configuration of input, read and write gates relative to a memory cell. The resulting 'modular memory unit' is embedded within the hidden layer of a feedforward neural network and can be trained using either neuro-evolutionary or gradient-based credit assignment mechanisms. Benchmarking was performed using the sequence classification and sequence recall tasks (also adopted in this thesis) with results typically better than reported by [22].

2.2 Genetic Programming with Memory

Recurrent (as opposed to external/indexed) memory is associated with the retention of a single variable's state. Koza originally proposed to support variables by having a single read/write operation to act on a specific variable reference in tree-structured GP [17]. Naturally, it was necessary to 'guess' the relevant number of variables. Conversely, under the so called 'linear GP' representation, instructions are expressed in an imperative programming language, e.g. `R[a] = R[a]` $\langle op \rangle$ `R[b]`; where `R[a]` is a reference to array/ register cell `a` and $\langle op \rangle$ is a two argument operator. Naturally, if the registers are not reset after execution of the program, all registers retain their

content and are therefore recurrent (or stateful) with respect to their previous value [19].

Silva et al. proposed a framework called 'Genetically Programmed Networks' (GPN) in which individuals consist of n programs [24]. Programs comprising an individual receive input in the form of task attributes, outputs from other programs in the same individual, or delayed outputs from other programs; the latter correspond to recurrent connections associated with a single program's output as fed back to an 'earlier' program in the same individual. Silva et al. demonstrate that the GPN framework is able to evolve solutions to the Tartarus (4×4) grid world task (introduced by Teller and Andre, see below). Memory in this case is distributed across the 'network' of n programs and only associated with the output of each program (as opposed to variables within programs).

Indexed memory for tree-structured GP was introduced by Teller ([28]) using a single argument 'read' operation (argument specified the memory 'address') and a dual argument write (value for writing and memory address). Naturally, the read operation returned the value read from memory, however, the write also returned a value, this time the value at the memory address before performing the write. This was necessary in order to ensure that values could continue to be passed up the tree (from the position of the write instruction). Teller went on to demonstrate that tree-structured GP could produce solutions to a grid world task in which blocks had to be pushed to the boundary of the grid (the Tartarus task). However, both Andre and Brave later pointed out that the role of indexed memory under Teller's experiments might be unrelated to the development of internal state representations for solving the Tartarus task. Agapitos et al. went on to consider the role of soft updates[4] to Teller's indexed memory model under a Financial Trading application [1].

Andre and Brave separately proposed tree-structured GP formulations for map making / planning in which the left and right hand branches were evaluated in independent 'phases'. During phase 1, the exploration branch is executed, implying for Andre that indexed memory can be written to, but the state of the environment cannot change [2, 3]. In short, the purpose of the exploration phase is to visit the 'interesting' states of the world and develop an appropriate representation in memory.

[4]Soft updates imply that writing the value `x` to memory location `i` is defined by the operation `M[i] = M[i]` $\times$ `x`. McPhee et al. develop the concept for tree-structured GP in general [20].

After completing phase 1, the same agent is allowed to execute its second branch, which in the case of Andre may only take input from memory. The agent, however, can now modify the world. Brave assumed the same division of duties, but went about developing the memory model using pointers expressed relative to the original state [4]. In both cases, the Tartarus grid world task was assumed (a 4×4 world), thus assuming that the agent could navigate the world without memory was feasible. Moreover, it was never necessary to develop a lower dimensional representation of state for efficiency purposes because the grid worlds were always sufficiently low dimensional. Indeed, isomorphic memory representations of state were the norm.

Langdon showed that tree-structured GP with indexed memory can evolve data structures such as queues and lists [18]. However, the explicit goal was to evolve the relevant operations to control indexed memory to operate as a queue/list. Langdon went on to demonstrate that given tasks that could benefit from specific data structures (such as queues or stacks), GP would use the data structures in preference to indexed memory (that would require the data structure to be evolved while solving the task). Finally, Langdon drew attention to the 'deceptive' nature of tasks involving a memory requirement. This was equated to the premature loss of useful primitives in the population. As such, diversity measures were introduced in an attempt to let primitives exist long enough for their respective purpose(s) to become apparent, i.e. have a measurable effect on performance.

Most recently, forms of memory have been developed for GP agents operating in both high dimensional (video frame buffer [26, 25, 14]) and low dimensional (1 or 2 inputs [13, 14, 15]) partially observable environments. Memory for other forms of GP has also been demonstrated, for example, recurrent 'connections' in Cartesian GP provides the basis for evolving solutions to low dimensional grid world and forecasting problems [29].

2.3 Discussion

Machine learning as applied to supervised learning and reinforcement learning[5] tasks need not be anything more than reactive. However, tasks that are in some way partially observable will need to augment (external) state information with internal state,

[5]Conforming to the Markov property of complete state information.

or memory. Section 2.1 provided a characterization of developments in memory mechanisms from the perspective of neural networks sufficient to motivate the research conducted in this thesis. Specifically, although recurrent connections are 'Turing-complete', they face practical limitations that limit their application. The Neural Turing Machine framework addresses this by providing an interface to indexed memory. Also noteworthy for this thesis was that neural-evolutionary paradigms could either build on the NTM or LSTM frameworks in order to more efficiently discover solutions to a set of scalable benchmark 'deep' memory problems.

Research using memory for genetic programming has taken a similar path (§2.2), with both recurrent/scalar memory and external indexed / data structures provided to enable tasks with partially observable state to be solved. However, to date, the tasks used to benchmark recent developments in neural networks have not been applied to GP. The approach of this thesis is to assume a tree-structured representation for GP but to revisit the role of external memory as a predefined data structure. We hypothesize that the style of credit assignment and representation assumed by canonical tree-structured GP will be sufficient for discovering efficient solutions to deep memory benchmarks that recent neural representations still find challenging. In the next section, the specific approach to deploying GP for these style tasks and the stack data structure itself will be detailed.

Chapter 3

List data structure with genetic programming

This thesis hypothesizes that we can draw on the approach adopted by Langdon [18] (reviewed in §2.2) to demonstrate that GP is also effective for solving benchmark 'deep' memory tasks (reviewed in §2.1). Specifically, we will assume the following:

- GP will take the form of the classical tree structure, as implemented in the open source DEAP Python distribution [6].

- External memory will take the form of a list data structure controlled by up to four commands: `push, pop_head, pop_tail, no_op`. Push and pop result in pushing and popping data to/from the list data structure (hence a **stack**). A data pushed will be the input state from the environment. Such a design decision is motivated, on the one hand, by external memory of the NTM [7] and neuro-evolution (e.g. [9]). On the other hand, Langdon demonstrated that GP could exploit data structures when provided. The machine learning **agent**[1] will have to develop a policy for controlling the stack given **reinforcement feedback** from the environment.

- Tree-structured GP is limited to a single output [17]. In order to support the four outputs necessary to control the stack, four populations will be co-evolved, i.e. a prior decomposition of the task. Individuals may only 'mate' with other individuals from the same population (ensuring a common context). The fitness of each program, however, is expressed relative to a sampling (without replacement) of one individual from each population.[2] Such a sampling across the four populations defines the GP reinforcement learning **agent**. Thus, fitness reflects individuals that are good at their respective tasks (push, pop, no-op) and

[1] By 'agent' we imply that any candidate solution that the machine learning framework suggests can be assumed. In this thesis, we compare different GP and neural evolutionary formulations.

[2] Applied repeatedly until no further individuals exist.

as well as good co-operators. Co-operation is therefore an emergent phenomena with only implicit support for 'team' or 'group fitness'.[3]

- Simple instruction sets will be assumed throughout. Unlike Langdon [18], no attempt will be made to construct instruction sets and functions that promote task-specific behavior (loops, functions shared between particular subsets of programs, etc). Instead, we limit the instruction set to a common set of arithmetic and logical operations for all benchmark tasks.

In the following, we provide the details for the above framework and conclude with a discussion and comparison of the proposed approach with earlier works.

3.1 GP Structure

DEAP (Distributed Evolutionary Algorithm in Python [6]): assumes Koza's original canonical tree-like structure [17] to evolve programs (Figure 3.1). Thus, the external leaves of the tree represent zero-argument references to input state variables, $s_i(t)$, and constants (green color). A set of constants are initialized prior to evolution and then remain the same while the state variables, $s_i(t)$, define the state of the environment at time step t. Internal nodes are the operations that will be executed using the leaves as inputs from the bottom up to produce a value in the internal nodes and ultimately the single root node (red color). This structure is sufficient when we have a task consisting of a single output. However, when a task requires support for multiple outputs then the tree-structured representation represents a limitation. To overcome the single output restriction, a multi-population framework is assumed, one population per output (Figure 3.2).

All of the experiments in this paper assume a **reinforcement learning** context; which is to say that the machine learning agent (GP or neural network) experiences an environment through the vector of inputs describing the current state of the environment. Let $s_i(t)$ denote the i-th state input at time step t. The agent then suggests an action, a, in this case selected from the set {`pop, push_tail, push_head, no_op`}. After which a reward might be received, $r(t+1)$, and the input state updates, $s_i(t+1)$,

[3] Explicit formulations for co-evolution through Teaming (e.g. [5, 12, 14]) might be assumed in future research.

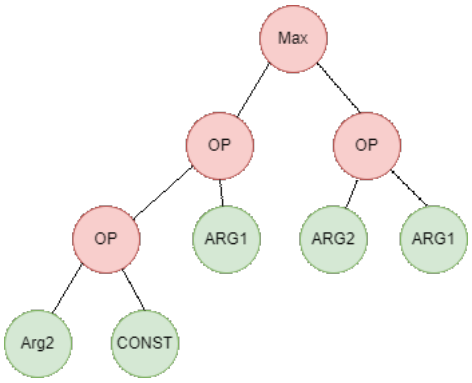

Figure 3.1: GP tree structure

to reflect the impact of the agent's action on the environment. If an action is deemed to result in the agent entering a terminal state (by the environment), the agent will receive a lower cumulative reward, $\sum_t r(t)$, than an agent that completes more of the underlying task.

An agent is the composition of GP individuals as sampled from each population, thus an agent has the same number of individuals as the possible actions. For instance, given a task with the `push`, `pop_head`, `push_head` and `no_op` actions, the agent will have four members, one from each of the four different populations. The composition of an agent has the following form: the first agent will be composed by selecting individual $k = 0$ from each population, the second agent will be composed by selecting individual $k = 1$ from each population and so on until $k = N - 1$; implying N agents have been constructed. Each member of the agent will be assigned the same fitness, i.e. fitness reflects the group performance of the set of programs comprising the agent. Each individual may only appear in one agent for each fitness evaluation. Moreover, index k does not imply any particular ordering on the respective populations, so $k = 0$ does not imply the best-performing program from each population.[4] The following pseudo-code describes the formation of a team:

[4]Biases could be investigated in the future. Instead, we will rely on mate selection to result in the best-performing programs receiving higher reproductive rights, thus appearing with more frequency within the population as generations progress.

```
# N = population size
pop1 = [tree1, tree2, ..., treeN]  # Represent Push
pop2 = [tree1, tree2, ..., treeN]  # Represent Pop Head
pop3 = [tree1, tree2, ..., treeN]  # Represent Pop Tail
pop4 = [tree1, tree2, ..., treeN]  # Represent No-Operation

# Create a list of teams
teams_list = []
for I = 1 to N
    teams_list.push([pop1[I], pop2[I], pop3[I], pop4[I]])
Next I
```

Listing 3.1: Agent Team formation

Each population is of a constant size, *'N'*. Having established the fitness of each agent, all programs participating in that agent receive the same fitness. Fitness evaluation is the only step performed using programs from each population. Selection is the process by which a population is seeded[5], so advancing each population from generation g to $g+1$. To do so, a tournament of size *'n'* is repeatedly applied (with replacement) to identify parents, one per tournament. In each case, the individual from the tournament with the highest fitness[6] is copied to the next population. This process continues until we have seeded a new population of size *'N'*. The following pseudo-code represents the process of selecting individuals:

```
# N = population size
# n = tournament size
chosen = []
for i in range(N):
    tournament = [random.choice(individuals) for i in range(n)]
    # attrgetter is a method to return individual attribute
    chosen.append(max(tournament, key=attrgetter(fitness))
```

Listing 3.2: Population Tournament Selection

Now that we have a population of parents with the best fitness of their respective tournaments, we introduce diversity. The variation process consists of two steps: 1) mate pairs of consecutive individuals with probability, `cxpb`. Cross-over selects the material from the parents to exchange [17], otherwise the parents remain unchanged

[5]The multi-population model implies that there are as many populations as actions.
[6]As inherited from the agent a program participated in.

2) mutate every individual with probability `mutpb`. Mutation implies that a randomly selected node is flipped to a matching argument operator/state index / constant. The following pseudo-code summarizes both the crossover and the mutation procedure:

```
chosen = individuals.copy()
# cxpb = Crossover Probability
# mutpb = Mutation Probability
for i in range(1, len(offspring), 2):
    if random.random() < cxpb:
        offspring[i-1], offspring[i] = mate(offspring[i-1],
    offspring[i])
        del offspring[i-1].fitness.value, offspring[i].fitness.
    value

    for i in range(len(offspring)):
        if random.random() < mutpb:
            offspring[i], = mutate(offspring[i])
            del offspring[i].fitness.value
```

Listing 3.3: Population Crossover and Mutation

The multi-population approach implies a decomposition of the task, while the same (state) inputs are used for each (GP individuals will identify which inputs to actually use). We use the value in the root node of the champion from each population, the population argument with the maximum value expresses the choice of output under the current state (input). For instance, if we have 4 populations and we get the following values from our champion root nodes (0.2, 0.1, 0.6, 0.25) then the resulting choice is the one expressed by the third population, or

$$a^* = \arg\max_{i \in \mathcal{A}}(GP_i(p_k)) \tag{3.1}$$

where $GP_i(p_k)$ is the numerical value returned from the root node of program i after execution w.r.t. input state $\vec{s}(t)$. $\mathcal{A}$ is the number of populations (and therefore the number of outputs specified in the task). a^* is the action suggested by the 'winning' program.

For instance, in the experiments below we assigned the first population to represent action *push to memory*, second population to *pop from memory* and third population to abstain from doing any action, i.e. *no-operation*. When evaluating fitness we take the output of each of those populations and if the value of the first population is the

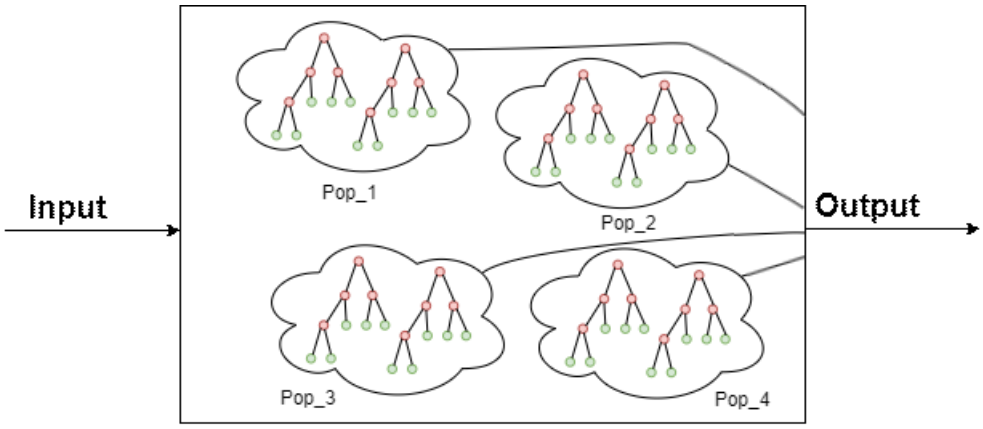

Figure 3.2: GP Multi-Population Structure.

highest then we execute the *push to memory* operation. The execution would then commence from the new state, $\vec{s}(t+1)$, given that the execution of state interacts with the task and does not result in a terminal state.

Memory: For memory we assumed the list data structure, where the first word added to the list is the first word retrieved from memory. However, in order to provide additional degrees of freedom to the resulting functionality, we provided the agent with the ability to retrieve from the head of the list or the tail. Function D is a delay function that looks at the head of the list and passes it to input in the next generational cycle without changing the value of the list head. An agent can interact with the memory and change it without the memory being able to affect the agent. We use function D to complete the loop between the agent and the memory, where the memory would have an impact on the agent as it is fed as input to the agent. Figure 3.3 summarizes the relationship between all of these components. Inputs are fed to GP individuals from each population. Based on the output from GP individuals comprising the team under evaluation, we get the argument associated with the 'winning action'. That action will update the state of the world, and the process iterates until some terminal condition is encountered.

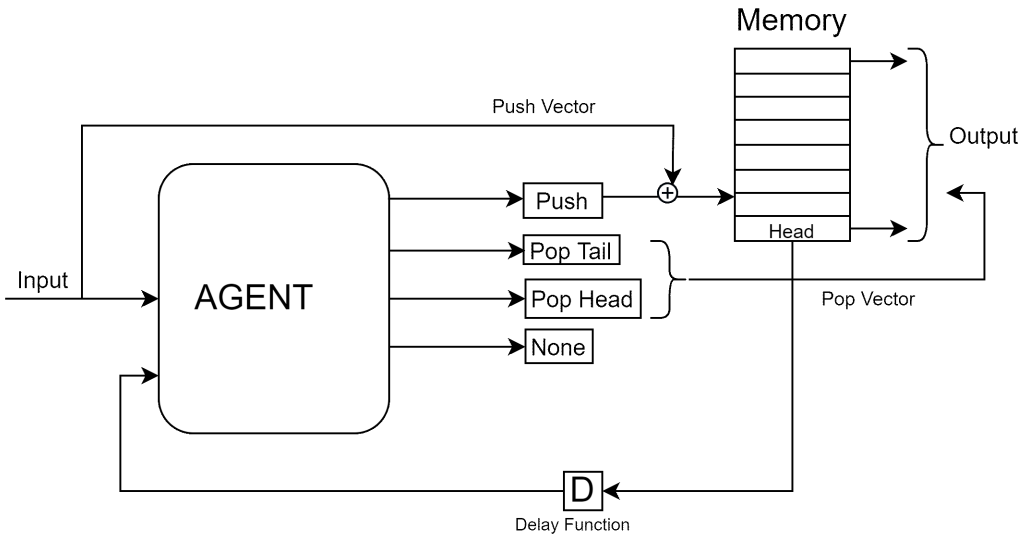

Figure 3.3: GP Stack Structure.

3.2 Evolution Algorithm

We assume the following overall process for the evolutionary algorithm, the algorithm maintains a list of candidate solutions (the population, size N) selects survivors (parents) before applying variation operators to maintain diversity. The algorithm consists of.

- each individual undergoes a fitness (performance) evaluation.
- individuals are selected to be parents using tournament selection. So $n(<< N)$ individuals are selected with uniform probability. The fittest (from each tournament) is copied to the next population. N tournaments are performed to identify N parents.
- variation operators are applied to the parents to introduce diversity.[7] This takes the form of Crossover (sexual variation) and Mutation (asexual variation).

The process then iterates, so the new individuals have their fitness evaluated. When the generational loop is done, the algorithm returns the final population. The following pseudo-code shows in more detail the steps the modified algorithm takes.

[7]Otherwise, the best individual at generation zero would ultimately just copy itself throughout the population.

```
# N = population size
pop1 = [tree1, tree2, ..., treeN]
pop2 = [tree1, tree2, ..., treeN]
pop3 = [tree1, tree2, ..., treeN]
pop4 = [tree1, tree2, ..., treeN]

# New trees have fitness set to invalid, add new trees of each
    population in a list to be evaluated
new_individual1 = [new_tree1, new_tree2, ..., new_treeN]
new_individual2 = [new_tree1, new_tree2, ..., new_treeN]
new_individual3 = [new_tree1, new_tree2, ..., new_treeN]
new_individual4 = [new_tree1, new_tree2, ..., new_treeN]

for I = 1 to N
    evaluate_fitness(new_individual1[I], new_individual2[I],
    new_individual3[I], new_individual4[I])
Next I

# halloffame is used to store the best performing individual in
    population.
halloffame1.update(pop1)
halloffame2.update(pop2)
halloffame3.update(pop3)
halloffame4.update(pop4)

For each generation evolving include:
# Select the next generation individuals (select entire population):
    offspring1 = select(pop1, len(pop1))
    offspring2 = select(pop2, len(pop2))
    offspring3 = select(pop3, len(pop3))
    offspring4 = select(pop3, len(pop4))

# Vary the pool of individuals:
    offspring_pop = clone(parent_pop)
    For I = 1 to len(offspring_pop)
        If mate_probability then
            child1, child2 = mate(offspring_pop[I], offspring_pop[I
    +1])
            offspring_pop[I] = child1
```

```
            offspring_pop[I+1] = child2
        End
    Next I
    For I = 1 to len(offspring_pop)
        If mutate_probability then
            mutate_child = mutate(offspring_pop[I])
            offspring_pop[I] = mutate_childtemp_data

# Select new trees with invalid fitness then (number of invalid tree
    != N):
    for I = 1 to N
        evaluate_fitness(new_individual1[I], new_individual2[I],
   new_individual3[I], new_individual4[I])
    Next I
# Update the hall of fame with the generated individuals.
# Replace the current population with the offspring.
# Return a list of final populations
```

Listing 3.4: Modified Simplest Evolutionary Algorithm

We note the use of a 'hall of fame' which is an elitism function to keep a record of the best individual through the generations. As each population (in the multi-population framework) converges the hall of fame will retain the champion agent.

3.3 Discussion

The proposed framework assumes the availability of a list that can be controlled using a set of actions: {`pop`, `push_tail`, `push_head`, `no_op`}. The goal of the agent is to determine the relevant policy to control the list to perform some tasks under the reward signal received by the environment. Given that we are using tree-structured GP, a separate population is employed for each action. A weak co-evolutionary framework will be assumed in which one individual is sampled from each population without replacement to build an agent. Fitness is measured relative to the agent's performance, and it is this fitness that each program comprising that agent receives. Once N agents are evaluated, we have the fitness for $\mathcal{A} \times N$ GP individuals. Selection, reproduction, and replacement can then be performed relative to each population.

The use of an external data structure is common to GP with indexed memory

(reviewed §2.2. However, the typical approach is to include instructions within the instruction set of programs, and have these manipulate memory. Instead, we assume the approach of NTM, neuro-evolution, and Langdon and have the action of a program be a signal sent to control a feature of external memory. This implies that there can only be one action per time step/interaction with the environment. However, as per NTMs, this is still sufficient for solving potentially difficult partially observable problems.

Chapter 4

Results

An empirical study will be performed with the weakly coupled coevolutionary GP formulation for controlling a list as a queue (§3) with three parameterizable 'deep' memory tasks: Sequence Recall (§4.2), and Sequence Classification (§4.3), and Copy Task (§4.4). Moreover, in addition to comparing our results with those previously published ([7, 9, 22, 16]), we will also run additional experiments using the NEAT framework for evolving neural networks. These experiments will assume the same interface to the list data structure as the weakly coupled coevolutionary GP formulation; hence, will give some insight as to how much is due to GP and how much is due to the data structure.

4.1 Experiment Parameterizations

A GP framework allows a user to use a wide variety of operators and user-defined primitives [6]. Using this feature gives the power to customize the population and come up with a solution that best fits the problem at hand. We varied the type and the number of operators used to solve memory problems. The hypothesis is that not only will smaller instruction sets result in discovering solutions faster, but also that including 'protected division' will have a significant impact when the task definitions from previous benchmarking studies are assumed [7, 9, 22, 16]. Table 4.1 shows the difference between the approaches taken, while table 4.2 lists parameters used by GP to configure the evolution algorithm:

Operators	Div	Multiplication	Full
Addition	×	×	×
Subtract	×	×	×
Protected Division	×		×
Multiplication		×	×
Boolean			×
if-then-else			×
And			×
Or			×
Not			×
Less than			×
Equal			×
Random Constant			×

Table 4.1: Table shows different operators used by different populations in attempt to best solve Tasks using GP.

Parameter	Value
Tournament size	5
Crossover Probability	0.5
Mutation Probability	0.4

Table 4.2: Table shows parameters used by GP to configure the evolution algorithm.

Protected division (%) was introduced by Koza as a mechanism to ensure that a real-valued number was always returned following application of arithmetic division [17]. Specifically, Koza proposed the following definition for protected division:

$$a\%b = \begin{cases} \frac{a}{b} & b > 0.0 \\ 1.0 & b = 0.0 \end{cases} \tag{4.1}$$

where in practice the use of floating point arithmetic implies that it is necessary to trap overflows resulting from $b \longrightarrow 0.0$

Several authors have noted that such a definition introduces a discontinuity into the division operation that works against its use in regression/ function approximation problems [21, 11]. We hypothesize that for the 'deep' memory benchmarks as typically defined, the above definition will actually make the task easier. That is to say, the division operation will act as if it is a conditional statement returning unity if the second argument is zero, and perform the division operation otherwise.

As remarked above, we will also repeat experiments in which the proposed weakly coevolutionary GP formulation is replaced by an approach to evolving neural networks. Specifically, the neural evolution of augmented topologies (NEAT) framework will be assumed on account of: 1) its use in prior research in the 'deep' memory benchmarks [8, 22], and; (2) to establish to what degree the results we achieve are due to the data structure versus the GP formulation. Naturally, the NEAT framework assumes a configuration file that governs how the network evolves.

Parameter	**Value**
fitness_criterion	Max
fitness_threshold	100
pop_size	400
reset_on_extinction	1
num_inputs	Task Related
Initial # hidden neurons	1
num_outputs	Task Related
initial_connection	partial_direct 0.5
feed_forward	False
compatibility_disjoint_coefficient	1.0
compatibility_weight_coefficient	0.6
conn_add_prob	0.1
conn_delete_prob	0.1
node_add_prob	0.1
node_delete_prob	0.1
activation_default	sigmoid
activation_options	sigmoid
activation_mutate_rate	0.1
aggregation_default	sum
aggregation_options	sum
aggregation_mutate_rate	0.0
bias_init_mean	0.0
bias_init_stdev	1.0

Parameter	Value
bias_replace_rate	0.1
bias_mutate_rate	0.7
bias_mutate_power	0.5
bias_max_value	30.0
bias_min_value	-30.0
response_init_mean	1.0
response_init_stdev	0.1
response_replace_rate	0.1
response_mutate_rate	0.1
response_mutate_power	0.1
response_max_value	30.0
response_min_value	-30.0
weight_max_value	30.0
weight_min_value	-30.0
weight_init_mean	0.0
weight_init_stdev	1.0
weight_mutate_rate	0.4
weight_replace_rate	0.5
weight_replace_power	0.5
enabled_default	True
enabled_mutate_rate	0.01
compatibility_threshold	3.0
species_fitness_func	max
max_stagnation	20
elitism	2
survival_threshold	0.2

Table 4.3: Table shows parameters used in configuration file in attempt to best solve tasks using NEAT.

Table 4.3 lists the values used in the configuration file. In our attempt to make NEAT work better we have changed the values of some of those parameters. Starting with the population size, we have tested with both higher and lower numbers in population and we didn't notice much of a difference. The initial number of hidden neurons was another parameter that we experimented with, a higher number didn't produce better results. We also changed the activation function, response mutate and weight mutate.

4.2 Sequence Recall

For our first experiment, we introduce the Sequence Recall Task, which is also known as T-maze navigation task [16, 22]. The Sequence Recall task requires an agent to start at one end of a T-maze and is given an instruction. The agent then moves in the 'maze' along a corridor until it reaches a junction. At this point, the agent needs to make a decision to go left or right, depending on the instruction given at the beginning of the task. In order to solve the task, the agent has to correctly and accurately remember the instruction given at the start of the task. The agent would also need to maintain this information and retain it to be used when encountering the junction as opposed to earlier points along the corridor.

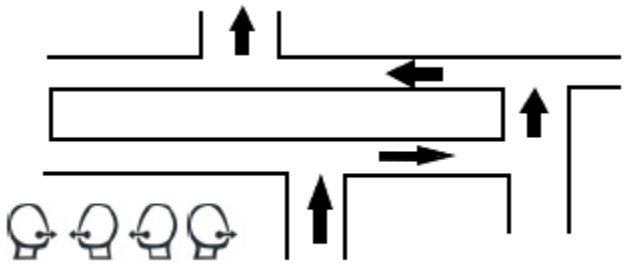

Figure 4.1: **Sequence Recall Task.** An agent listens to a series of instructions at the beginning of the trail, then travels along multiple corridors. The agent needs to select which direction it will go at a junction based on the direction received at the beginning.

Starting with the simple form of the Sequence Recall task, it can be extended to solve more complex higher depth Sequence Recall tasks, i.e. consist of multiple junctions and corridors of variable length, Figure 4.1. At the start of the maze, the

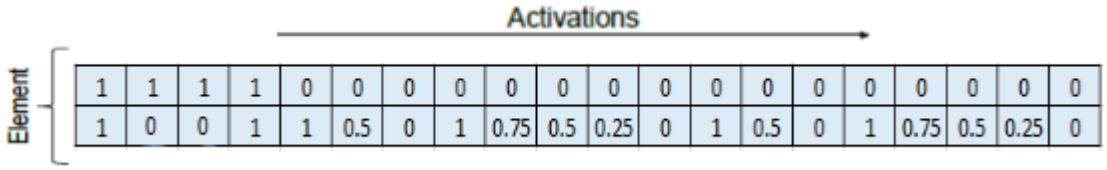

Figure 4.2: **Sequence Recall Input Sequence Example.** The ones in the first line indicate the agent receiving instructions. The accompanying second line is the instructions, where 1 means right and 0 means left. From here on, the second line will indicate the length of the corridor. When the corridor reaches zero, this would indicate the agent reached a junction.

agent receives a *set of instructions* before moving along the current corridor. When the agent encounters a junction, it will need to decide on the direction it will move, based on the instructions received at the beginning of the task to reach the end goal. Figure 4.2 is the input corresponding to Figure 4.1.

The input consists of 2D array where the first input indicates the 'mode' for interpreting the second bit. Thus, only when the first bit is set to '1' should the agent treat the second input as a direction. Thereafter, the first input takes a value of '0' indicating that the agent needs to navigate the maze. The second input can either be a direction or a countdown to the end of a corridor. Once a corridor length reaches zero, this would mean the agent has reached a junction and has to make a decision based on the instruction received at the beginning of the task. In order for the agent to successfully solve that task, it needs to accurately memorize the instructions in the original order and then retrieve this information in sequence, through multiple corridors of varying lengths.

The fitness of the program is calculated by how well it can reuse the input information to navigate the maze. For instance, the example given in 4.2 should follow the flow shown in the following step by step table:

index	input	**Description**	**Action**
0	[1,1]	an instruction bit of going right, commit to memory	Push
1	[1,0]	an instruction bit of going left, commit to memory	Push
2	[1,0]	an instruction bit of going left, commit to memory	Push
3	[1,1]	an instruction bit of going right, commit to memory	Push

index	input	**Description**	**Action**
4	[0,1]	Moving in the corridor	None
5	[0,0.5]	Moving in the corridor	None
6	[0,0]	Reached a corner and retrieve an instruction from memory	Pop_Head
7	[0,1]	Moving in the corridor	None
8	[0,0.75]	Moving in the corridor	None
9	[0,0.5]	Moving in the corridor	None
10	[0,0.25]	Moving in the corridor	None
11	[0,0]	Reached a corner and retrieve an instruction from memory	Pop_Head
12	[0,1]	Moving in the corridor	None
13	[0,0.5]	Moving in the corridor	None
14	[0,0]	Reached a corner and retrieve an instruction from memory	Pop_Head
15	[0,1]	Moving in the corridor	None
16	[0,0.75]	Moving in the corridor	None
17	[0,0.5]	Moving in the corridor	None
18	[0,0.25]	Moving in the corridor	None
19	[0,0]	Reached a corner and retrieve an instruction from memory	Pop_Head

Table 4.4: Table shows step by step simulation for agent processing sequence recall input

Leaving us with the following list of actions:

[Push] [Push] [Push] [Push] [None] [None] [Pop_Head] [None] [None] [None] [None]
[Pop_Head] [None] [None] [Pop_Head] [None] [None] [None] [None] [Pop_Head]

Each of these actions carries a weight of 1, the fitness will be calculated up until the performed action does not match the expected action. Training is performed on 50 different sequences with random corridor lengths between 10 and 20 steps. The list data structure serves as a memory to store the instruction given to the agent. The agent will have the freedom to pick an action from the head or tail of the queue or four outputs consisting of: `push, pop_head, pop_tail, no_op`. When evolving the programs there are a few settings that can be tweaked to control the complexity

of the task:

- **Sequence Depth:** One parameter that can change is the number of instructions given to an agent at the beginning of the task. This will impact the number of corridors and junctions in a maze. The more instructions the agent receives at the beginning of the task, the more corridors and junctions the agent will face to solve the maze. In our experiment, we have experimented with depths of 4, 5, 6, 15, and 21.

- **Corridor Length:** Number of steps the agent has to take to reach the junction. The length is determined randomly and ranges between 10 and 20. This will help ensure that the agent cannot simply memorize when to retrieve info from the memory structure, but it has to react to reaching the junction. The agent would also need to decide on what to retrieve from the memory structure, either from the head of the queue or the tail of the queue. If we were to use fixed-length corridors, the agent would memorize when to turn rather than learn what constitutes a junction, and when faced with a different layout the agent will not succeed in finding a solution.

- **Iterations:** we have chosen to work with 50 separate tasks of random sequences with varied depths *per fitness evaluation*. This was done to ensure that solutions do not optimize for one specific length, but can generalize to different depths.

4.2.1 GP versus NEAT

We begin by comparing the GP configuration with that of NEAT, both assuming the same interface an external list. Results shown in Figure 4.3 represent 20 runs of 250 generations in GP and 500 generations in NEAT. The GP fitness of each generation was calculated by evaluating the champion of said generation on a new randomly generated sequence and recording the evaluation results (not used for fitness evaluation-selection-replacement). While NEAT fitness was calculated based on the fitness of the champion during training. The GP run assumes the Div operators from Table 4.1 $(+, -, \%)$ and NEAT configuration file had the parameters from 4.3 with

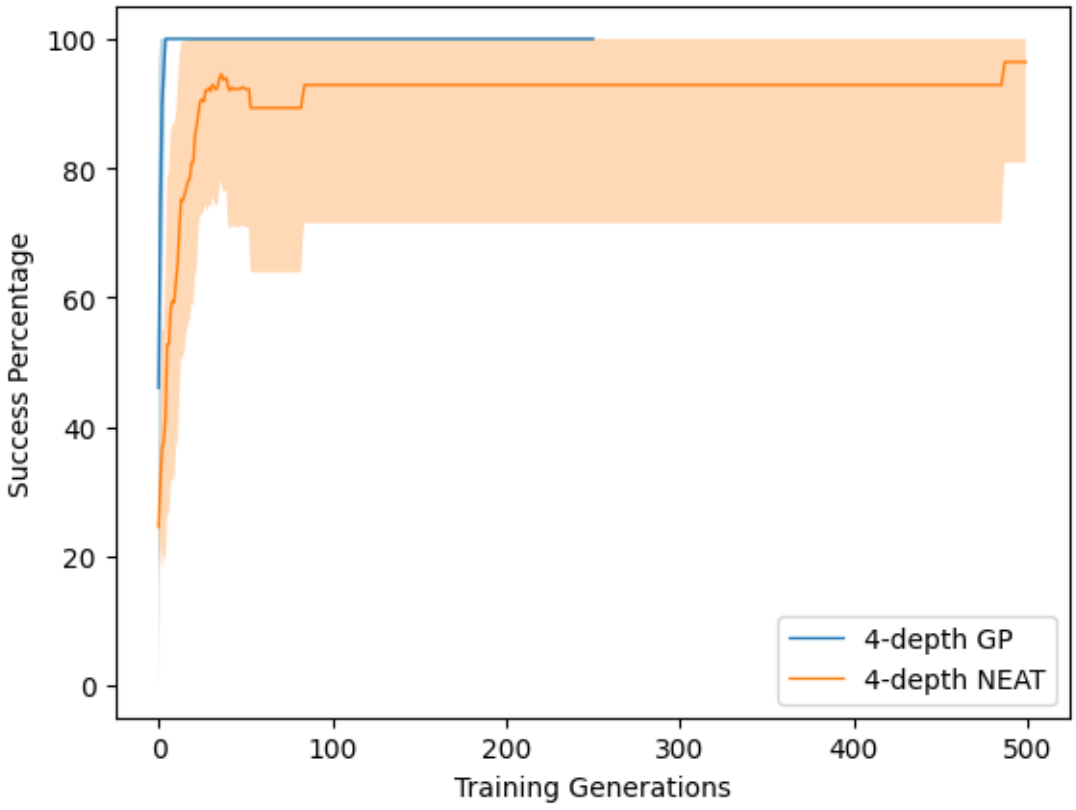

Figure 4.3: **GP vs NEAT 4-depth Sequence Recall Task Results.** Figure shows an evolving solution for a 4-depth sequence of varied corridor lengths using mean and standard deviation of 20 runs. The x-axis shows the number of generations while the y-axis shows the fitness of the generation. Colored lines show the mean of the 20 runs. The shaded area shows the standard deviation of the 20 runs.

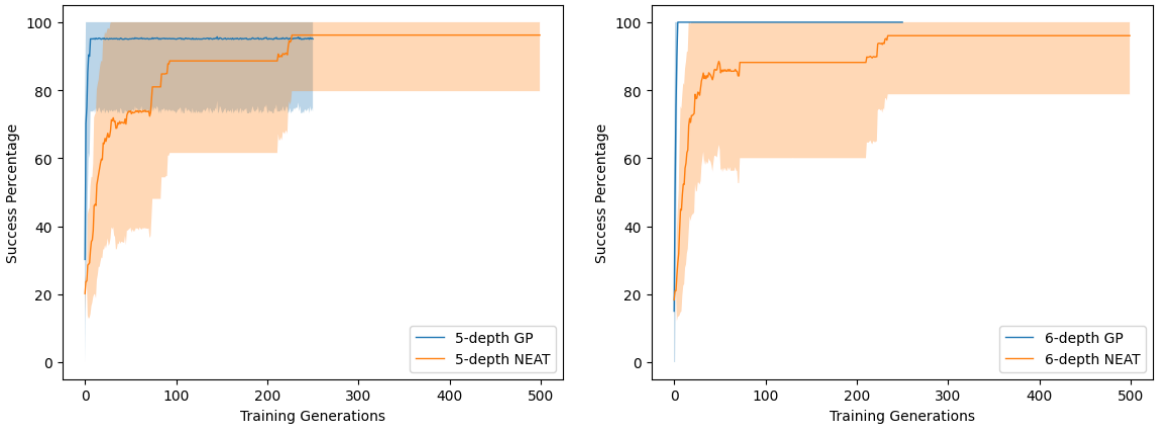

Figure 4.4: **GP vs NEAT 5- and 6-depth Sequence Recall Task Results.** Figure shows an evolving solution for 5-depth and 6-depth sequence of varied corridor lengths using mean and standard deviation of 20 runs. The x-axis shows the number of generations while the y-axis shows the fitness of the generation. Colored lines show the mean of the 20 runs. The shaded area shows the standard deviation of the 20 runs.

num-inputs = 2 (Instructions and data) and num-outputs = 4 (Push, Pop Head, Nothing and Pop Tail).

Figure 4.3 shows the mean and standard deviation of the 20 runs combined. We will start with sequences of 4 junctions depth [1], showing that GP has the upper hand in this number of solutions and how fast it was reached. The following code represents the functions of the champion reached in GP:

```
# Tree1 represent Push action
tree1 = add(ARG0, ARG0)
# Tree2 represent Pop Head action
tree2 = protected_div(ARG0, ARG1)
# Tree3 represent No Action
tree3 = add(ARG0, ARG1)
# Tree4 represent Pop Tail
tree4 = add(ARG1, ARG0)
```

Listing 4.1: Sequence Recall Champion Code

[1] Testing was done on a different set of 4-depth sequences.

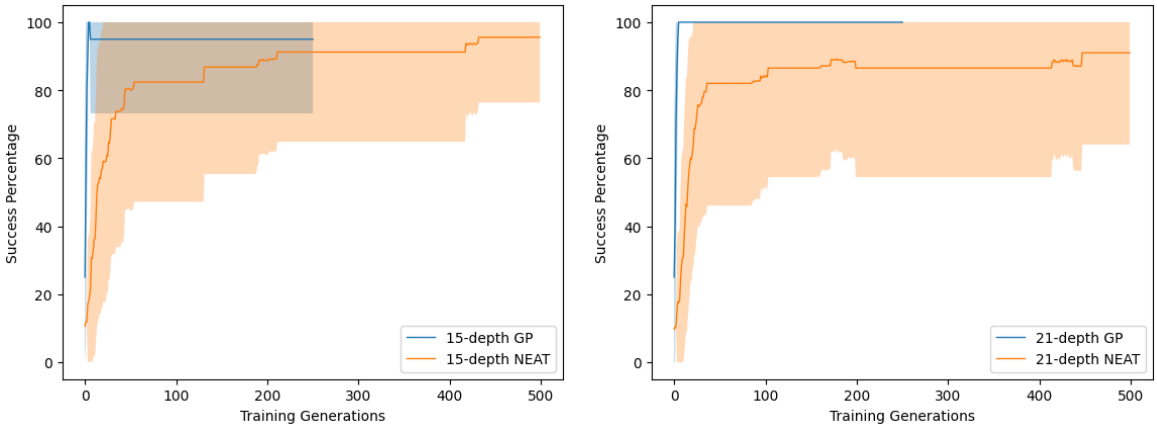

Figure 4.5: **GP vs NEAT 15- and 21-depth Sequence Recall Task Results.** Figure shows an evolving solution for 15-depth and 21-depth sequence of varied corridor lengths using mean and standard deviation of 20 runs. The x-axis shows the number of generations while the y-axis shows the fitness of the generation. Colored lines show the mean of the 20 runs. The shaded area shows the standard deviation of the 20 runs.

Looking at results in Figure 4.4 we can see that NEAT starts to fall behind GP at the beginning as the task becomes more complex and then catches up and find a solution. The difference is not that significant after the first 250 generations as the standard deviation starts to become closer to the mean of 20 runs. Following the same pattern we can see in Figure 4.5 NEAT falls even farther behind as the task gets more complicated while GP continues to perform better even in the 21-depth task. The standard deviation for NEAT does not become any better as generations progress in 15-depth and 21-depth.

Table 4.5 summarizes the respective GP and NEAT model outcomes under all the different depth limits associated with the Sequence Recall task.

Generalizing Task: Taking the champions from the GP 21-depth runs and testing it with even higher depths we managed to solve the task every single time. We tested with 50-depth and 100-depth. Which led us to the conclusion, once the task is being solved the champion can solve higher depth with no issues.

	GP (test)		NEAT (training)	
	Mean	Standard Deviation	Mean	Standard Deviation
4-depth	100	0.0	96.42	15.59
5-depth	95.1	21.35	96.21	16.5
6-depth	100	0.0	96.04	17.25
15-depth	95.0	21.79	95.59	19.20
21-depth	100	0.0	91.0	26.97

Table 4.5: Table shows final generation Mean and Standard Deviation for sequence recall NEAT and GP.

4.2.2 Replacing the protected division instruction

In the next experiment, we investigate the impact of the protected division on our instruction set, since a protected division will return '1' in case of dividing by '0'. We hypothesize that such an operator effectively represents a conditional statement that returns '1' when the denominator is '0' and therefore particularly useful for memory recall task definitions as typically assumed. We swapped Div operators with Multiplication operators from Table 4.1 and we ran the experiment again. Results in Figure 4.6 show the difference between Div and multiplication operators used to solve this task. We can see immediately the effect multiplication is having on the solution. In both 4-depth and 5-depth cases, a solution was not found using Multiplication operators.

Table 4.6 reports the resulting mean and standard deviations under test conditions. Replacing the protected division operator with multiplication clearly 'breaks' the capacity of GP to solve the Sequence Recall task, whereas under the Copy Task, replacing protected division with multiplication was preferable.

	Div		Multiplication	
	Mean	SD	Mean	SD
4-depth	100	0.0	0.0	0.0
5-depth	95.1	21.35	0.0	0.0

Table 4.6: Table shows final generation Mean and Standard Deviation for sequence recall Div and Multiplication instruction sets in GP.

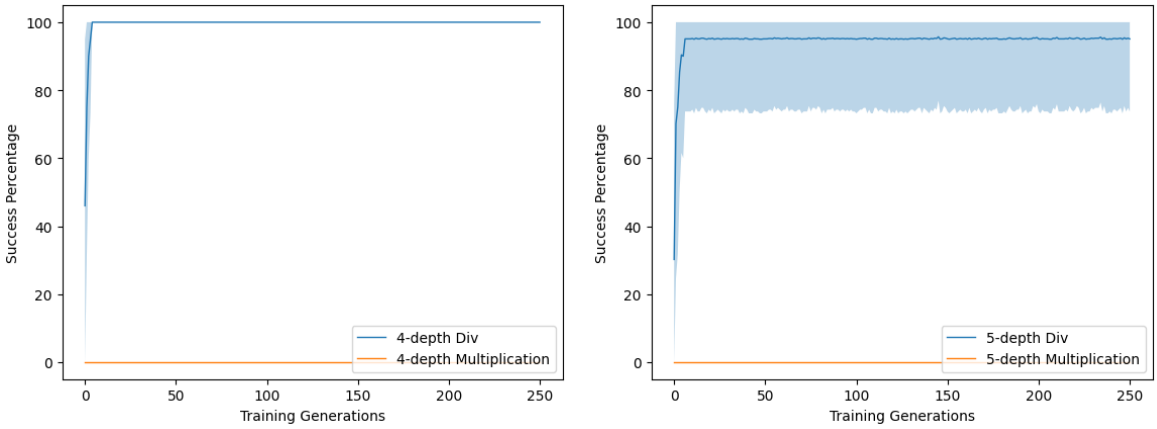

Figure 4.6: **Div vs Multiplication 4- and 5-depth Sequence Recall Task Results.** Figure shows the difference between Div and Multiplication operators evolving solution for 4-depth and 5-depth sequence of varied corridor lengths using mean and standard deviation of 20 runs. The x-axis shows the number of generations while the y-axis shows the fitness of the generation. Colored lines show the mean of the 20 runs. The shaded area shows the standard deviation of the 20 runs.

4.2.3 Complex instruction set

Having established that the concise instruction set requires protected division, we expand the instruction set to include arithmetic, Full operators, and constants (Table 4.1) and we repeated the experiment once again.

Results in Figure 4.7 show that there is no difference between the two different instruction sets under the 4-depth setup. Moving on to the 5- and 6-depth task parameterizations (Figure 4.8) the Full instruction set have better performance and find solutions every single time. We also notice how consistent the standard deviation is for the mean of those solutions.

Conversely, under the 6-depth task parameterization, both lines almost match. Looking more into the results we found that in one of the runs a solution was not reached which caused the wider standard deviation in the Div instruction set under the 5-depth task. The following code represents the functions of the champion reached in the GP Complex instruction set using Full operators:

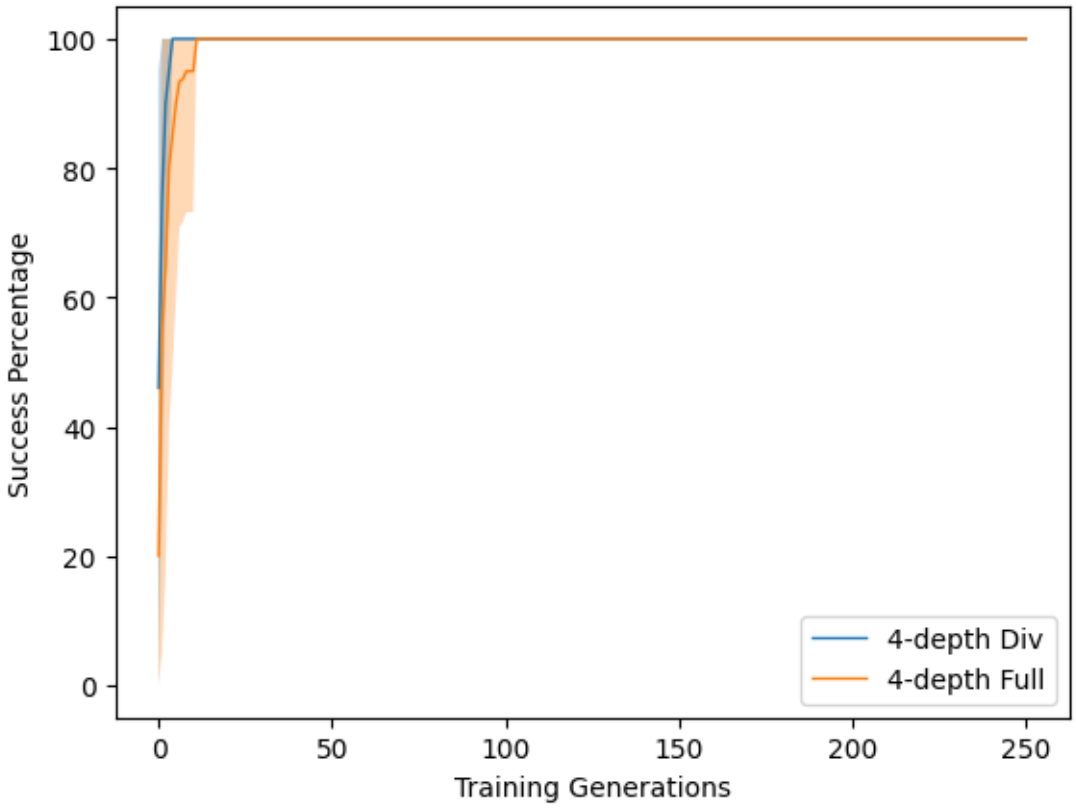

Figure 4.7: **Div vs Full 4-depth Sequence Recall Task Results.** Figure shows the difference between Div and Full operators evolving solution for a 4-depth sequence of varied corridor lengths using mean and standard deviation of 20 runs. The x-axis shows the number of generations while the y-axis shows the fitness of the generation. Colored lines show the mean of the 20 runs. The shaded area shows the standard deviation of the 20 runs.

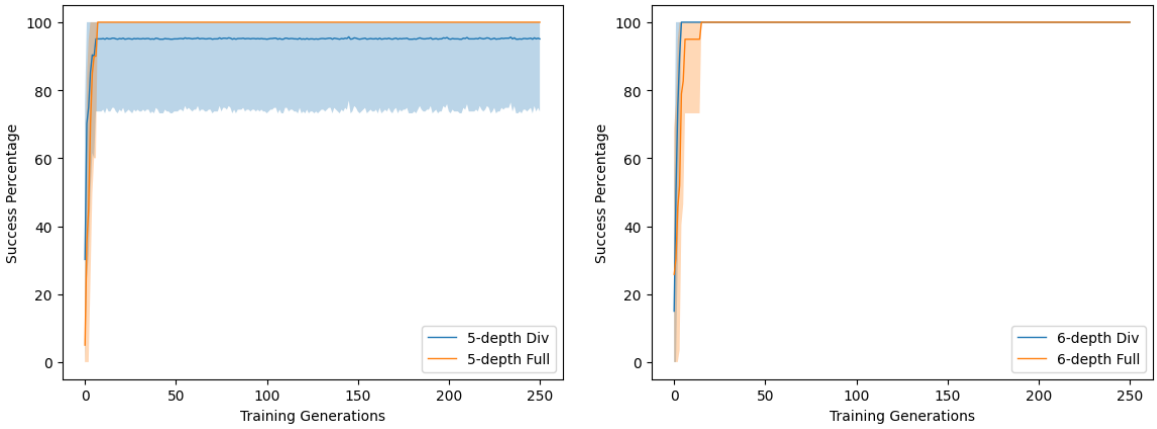

Figure 4.8: **Div vs Full 5- and 6-depth Sequence Recall Task Results.** Figure shows the difference between Div and Full operators evolving solution for 5-depth and 6-depth sequence of varied corridor lengths using mean and standard deviation of 20 runs. The x-axis shows the number of generations while the y-axis shows the fitness of the generation. Colored lines show the mean of the 20 runs. The shaded area shows the standard deviation of the 20 runs.

```
# Tree1 represent Push action
tree1 = mul(mul(ARG0, ARG0), if_then_else(False, ARG0,
    92.26174280394346))
# Tree2 represent Pop Head action
tree2 = protected_div(ARG0, ARG1)
# Tree3 represent No Action
tree3 = mul(if_then_else(lt(ARG1, ARG0), mul(ARG0, ARG0), mul
    (57.58064261386541, ARG1)), if_then_else(True, ARG1, ARG0))
# Tree4 represent Pop Tail
tree4 = mul(mul(ARG0, 21.05549475191497), if_then_else(True, ARG0
    , ARG0))
```

Listing 4.2: Sequence Recall Full Champion Code

The same theme appears when looking at 15-depth (Figure 4.9), Full instruction set is having an easier time finding a solution in every single run. The Div instruction set did not find a solution in one instance which led to the wider standard deviation. Looking at 21-depth we find no difference in performance, both of our instruction sets have performed exceptionally, almost immediately a solution was found regardless of the complexity of the task.

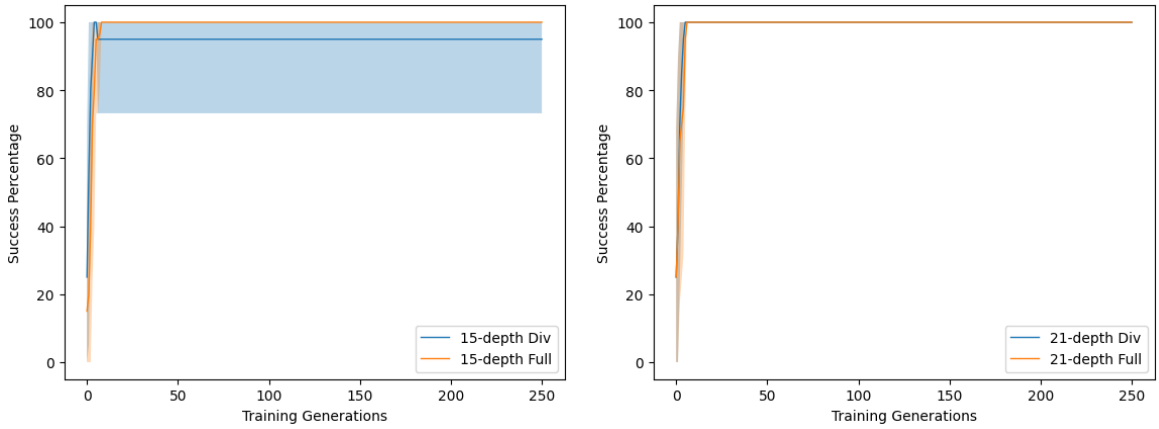

Figure 4.9: **Division vs Full 15- and 21-depth Sequence Recall Task Results.** Figure shows the difference between Div and Full operators evolving solution for 15-depth and 21-depth sequence of varied corridor lengths using mean and standard deviation of 20 runs. The x-axis shows the number of generations while the y-axis shows the fitness of the generation. Colored lines show the mean of the 20 runs. The shaded area shows the standard deviation of the 20 runs.

Table 4.7 summarizes the generalization performance of the Div versus Full instruction sets. The consistency of the Full instruction set is again very obvious. The implication is that the protected division might lead to local solutions appearing when the instruction set is reduced.

	Div		Full	
	Mean	Standard Deviation	Mean	Standard Deviation
4-depth	100	0.0	100	0.0
5-depth	95.1	21.35	100	0.0
6-depth	100	0.0	100	0.0
15-depth	95.0	21.79	100	0.0
21-depth	100	0.0	100	0.0

Table 4.7: Table shows final generation Mean and Standard Deviation for sequence recall Div and Full configurations in GP.

4.2.4 Noisy data with protected division instruction set

For our last change to the Sequence Recall task, we introduce changes to make the task harder for GP to solve. Specifically, we introduced the following changes to the original task:

- switched the zeros with -1. So now the pairs for instructions would be 1 to go right and -1 to go left rather than the original of 1 to go right and 0 to go left.

- change corridor lengths by adding 1 to every step. Starting from 1 + corridor_length and counting down till reaching 1. Meaning that an action must be taken when the corridor countdown reaches 1 now rather than taking action when the countdown reaches zero in the original setup.

These changes were intended to make the task harder to solve with the protected division operator. We ran the modified task with the same parameters as the original setup for 20 runs and recorded the results.

Looking at results in Figure 4.10 for the 4-depth parameterization we can see the effect of those changes as compared to runs performed under the original task definition. Both setups find an immediate solution to the task, however, a wider standard deviation appears under the modified task definition, suggesting that some champions failed to generalize.

We then started to look at deeper versions of the task. Figure 4.11 summarizes outcomes for the 5- and 6-depth task parameterizations. We observe an exchange in performance between the two setups. First, in the 5-depth runs the Original setup is having a wider standard deviation with almost none for the Modified setup, while the complete opposite is happening in 6-depth; Modified results in the wide standard deviation. Looking deeper into the results we can note that one case caused the wider variance in both cases.

Running the same setups on more complex mazes like 15-depth and 21-depth (Figure 4.12) the agent has no issues in solving either of those setups. We can see a bit of fluctuation at the earlier generations in 15-depth but working solutions are reached within the first 10 to 25 generations in both cases. As for the 21-depth test case Figure 4.12 shows a smooth performance from the beginning to the end with the modifications having no effect on the performance.

In conclusion we can certainly say that GP is having no issue finding a solution in any case as long as we do not use Multiplication instruction set (no protected division). Moreover, as summarized by Table 4.8 we note that attempting to remove the use of '0' in the problem definition has no impact on the ability of GP solutions to

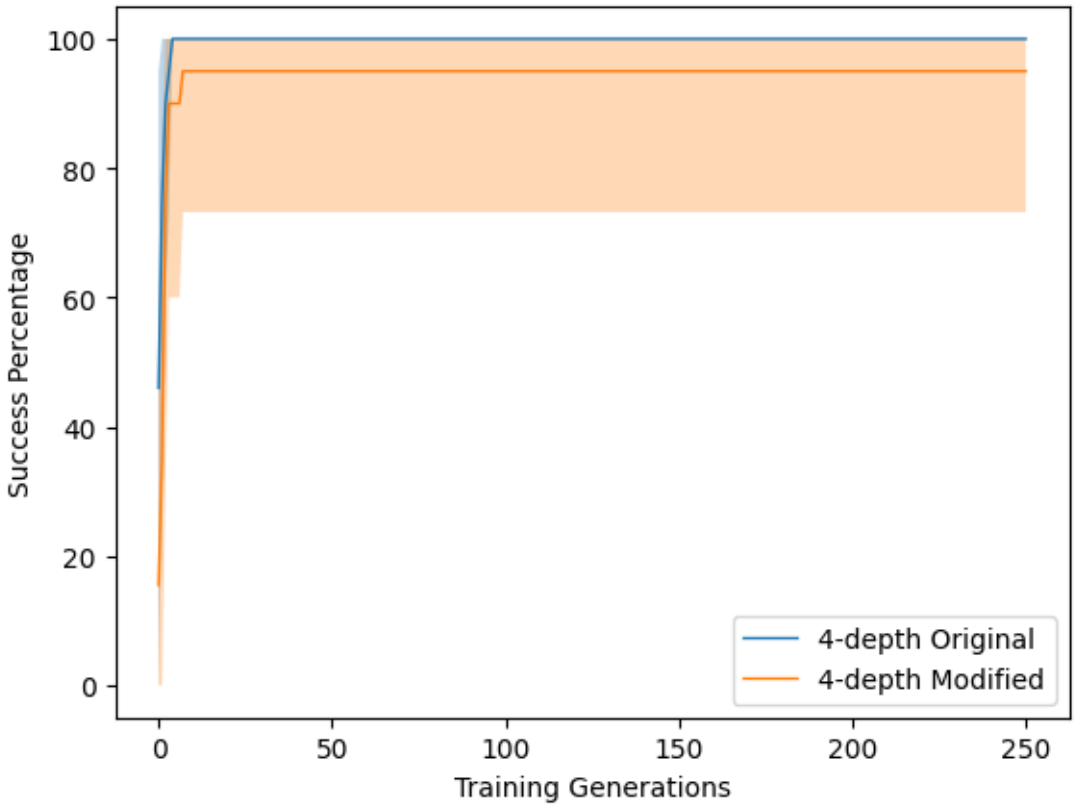

Figure 4.10: **Original vs Modified 4-depth Sequence Recall Task Results.** Figure shows the difference between Original and Modified setups evolving solution for a 4-depth sequence of varied corridor lengths using mean and standard deviation of 20 runs. The x-axis shows the number of generations while the y-axis shows the fitness of the generation. Colored lines show the mean of the 20 runs. The shaded area shows the standard deviation of the 20 runs.

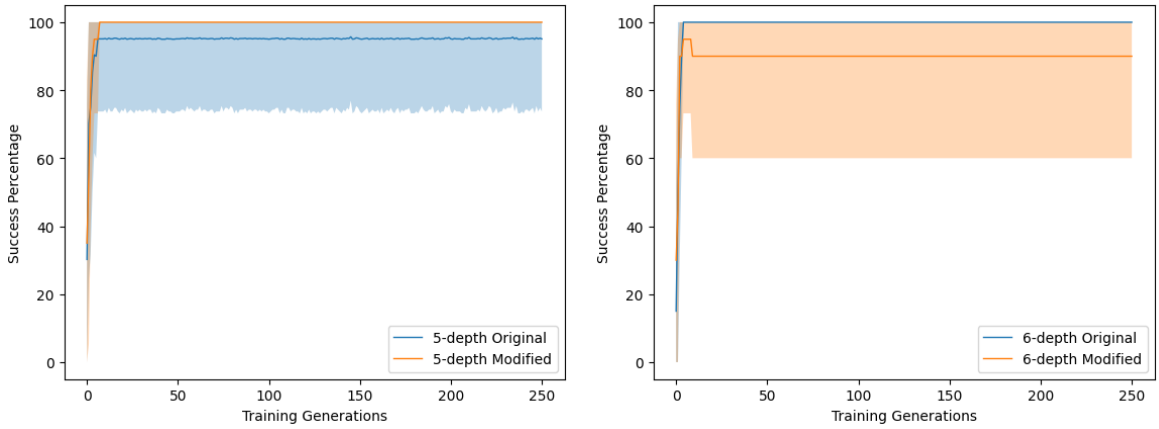

Figure 4.11: **Original vs Modified 5- and 6-depth Sequence Recall Task Results.** Figure shows the difference between Original and Modified setups evolving solution for 5-depth and 6-depth sequence of varied corridor lengths using mean and standard deviation of 20 runs. The x-axis shows the number of generations while the y-axis shows the fitness of the generation. Colored lines show the mean of the 20 runs. The shaded area shows the standard deviation of the 20 runs.

be consistently identified. On the other hand, NEAT is unable to consistently solve the task, there was instability in the performance presented by the wider standard deviations in the charts.

	Original		Modified	
	Mean	Standard Deviation	Mean	Standard Deviation
4-depth	100	0.0	95.0	21.79
5-depth	95.1	21.35	100	0.0
6-depth	100	0.0	90.0	30.0
15-depth	95.0	21.79	100	0.0
21-depth	100	0.0	100	0.0

Table 4.8: Table shows final generation Mean and Standard Deviation for sequence recall Original and Modified task setup in GP.

4.2.5 Generalization

In order to test the capacity to generalize the solution we did the following. First, we are using the original task setup using corridor length value of 10 and Div instruction set from table 4.1. Second, we trained 20 different champions. Each of these champions will be tested on 50 and 100 depth. Third, we generate 50 random test

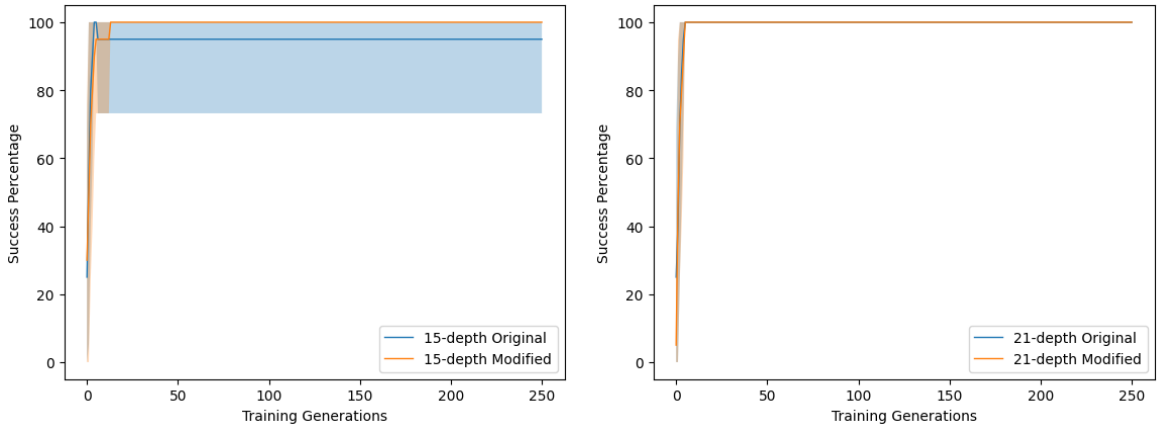

Figure 4.12: **Original vs Modified 15- and 21-depth Sequence Recall Task Results.** Figure shows the difference between Original and Modified setups evolving solution for 15-depth and 21-depth sequence of varied corridor lengths using mean and standard deviation of 20 runs. the x-axis shows the number of generations while the y-axis shows the fitness of the generation. Colored lines show the mean of the 20 runs. The shaded area shows the standard deviation of the 20 runs.

for each champion on each length. Then, we ran the tests and recorded how well the champions managed to solve the task.

Table 4.9 shows the Mean and Standard Deviation of the test of the champion. Looking at the results from 50 and 100 depths we can say that the champion had no problem scaling up to a more complex layout. The high success rate of 98% on both layouts with the low standard deviation leads us to state that the champion can scale up without any issues.

	Mean	**SD**
50 depth	97.7	9.9
100 depth	97.7	9.8

Table 4.9: Table shows Mean and Standard Deviation for sequence recall GP 20 champions generalizing on 50 and 100 depth.

4.3 Sequence Classification

For our second experiment, we are using Sequence Classification from [16, 22]. Sequence Classification is a parameterizable 'deep' memory task where the agent has to

Input Sequence	Target Output
-1 0....0 1 0....0 -1	-1 .... 1 .... -1
-1 0....0 -1 0....0 1	-1 .... -1 .... -1
-1 0....0 1 0....0 1	-1 .... 1 1
1 0....0 1 0....0 -1	1 .... 1 1

Figure 4.13: **Sequence Classification input.** The agent receives a sequence of input signals (1/-1) mixed with noise (0's) of variable length. At each introduction of signals, the agent has to determine if it has received more 1's or -1's. Agent receives a sequence as input and output -1 if the number of -1's is greater than the number of 1's else output 1's. [16]

keep track of the classification target through a long sequence of signals mixed with noise. The agent is given a sequence of -1/1 signals with a random number of zero's in between. The agent has to decide if it received more 1's or -1's at the end. The depth of the task is determined by the number of 1's and -1's in the sequence. The agent has to learn to ignore the noise represented by 0's in the sequence. Traditional recurrent neural networks have a hard time with this task especially as the sequence grows longer [22, 16].

In order to successfully solve the task, the agent has to concentrate on the 'non-zero' information throughout the sequence. For instance, for a 6-depth sequence, the network has to make the right classification for every sub-sequence. If any sub-sequence classification is incorrect, then the entire sequence classification becomes incorrect. The number of zeros in a sequence is determined randomly following each signal (1/-1) and ranges from 10 to 20. This inconsistency adds to the complexity of the task, as the agent cannot simply memorize when to classify signals and ignore the noise. An intelligent decision-making system needs to have the capability to filter out distractions and concentrate on useful signals. Mastering this property is a necessity to be able to generalize the task to solve more complex higher depth Sequence Classification tasks.

Starting with the simple form of the Sequence Classification task we can extend

it to solve more complex higher depth Sequence Classification tasks which consist of multiple signals. Figure 4.13 represents examples of input sequences and expected output sequences. The input represents a 2D-array the 1st bit comes from the input sequence while the 2nd bit comes from memory controlled by the agent. The fitness of the program is determined by how well it classifies the input. Each of these actions carries a weight of 1, the fitness will be calculated up until the performed action does not match the expected action. Training is being performed on 50 separate iterations of sequences with intervening noise (zeros) randomly selected between 10 and 20. As in the earlier experiments, the list data structure serves as a memory to store the classification as the agent advances. When evolving the programs there are a few settings that can be tweaked to control the complexity of the task:

- **Signal Depth:** One parameter that can change is the number of signals given to an agent in a sequence. This will impact the number of signals and noise in sequence. The more signals and noise the agent receives, the more classifications the agent will have to make to solve the task. In our experiment we've experimented with 4-, 5-, 6-, 15- and 21-depth signals.

- **Noise Length:** Number of classifications the agent has to take to reach the signal. The length is determined randomly and ranges between 10 and 20. This will help ensure that the agent cannot simply memorize when to retrieve info from the memory structure, but it has to react to reaching the signal.

- **Iterations:** we have chosen to work with 50 separate evaluations of sequences with common signal depths per experiment. This was done to ensure that the program does not optimize toward one specific length of noise but can generalize to multiple noise lengths.

4.3.1 GP versus NEAT

We start our experiment by comparing the results between NEAT and GP. We run the task on 20 runs of 500 generations for NEAT and 250 generations for GP. GP fitness of each generation is calculated by taking the champion of said generation and testing it on new randomly generated sequences (of the same signal depth) and

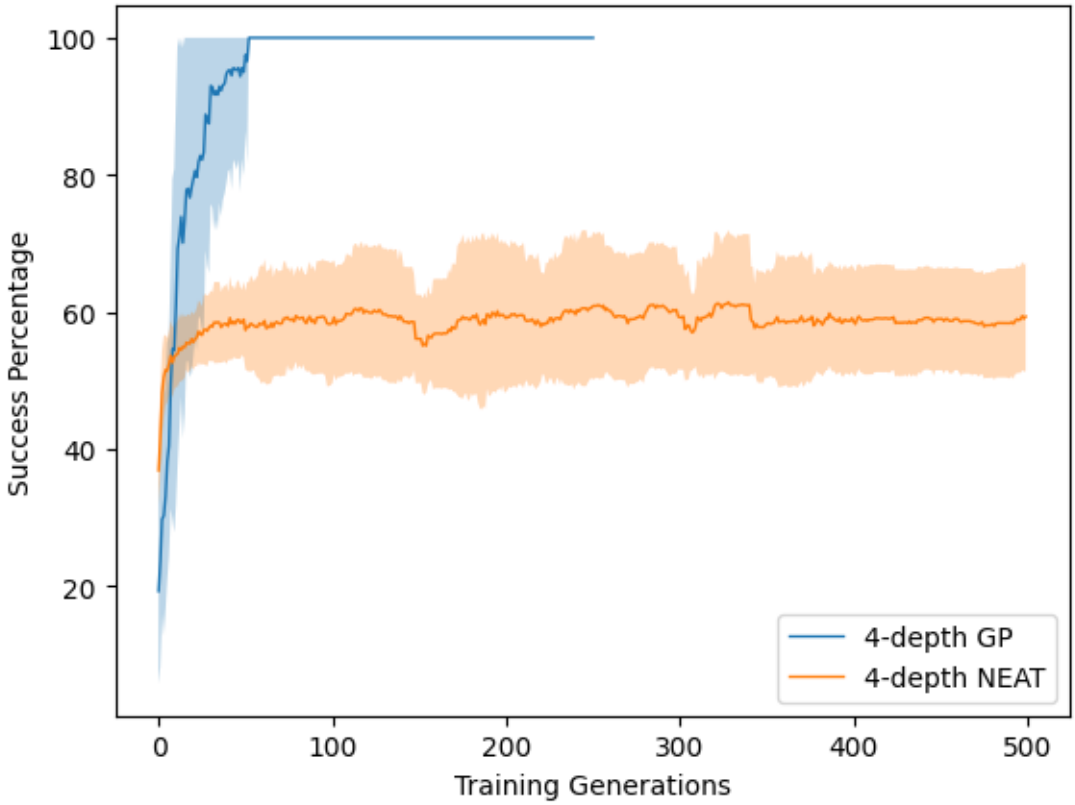

Figure 4.14: **GP vs NEAT 4-depth Sequence Classification Task Results.** Figure shows the evolving solution for a 4-depth sequence of varied sequence lengths using mean and standard deviation of 20 runs. the x-axis shows the number of generations while the y-axis shows the fitness of the generation. Colored lines show the mean of the 20 runs. The shaded area shows the standard deviation of the 20 runs

recording the results. On the other hand, NEAT results come from the generation training fitness.

GP runs were done using the Div operators from Table 4.1 and NEAT configuration file had the parameters from Table 4.3 with num-inputs = 3 (sequence and input from memory stack) and num-outputs = 4 (`Push, Pop Head, Pop Tail, No_op`). Looking at Figure 4.14 we can immediately see the difference in performance between the two platforms. GP performs significantly better than NEAT as it is able to solve the task in every single run, while NEAT is unable to reach a solution.

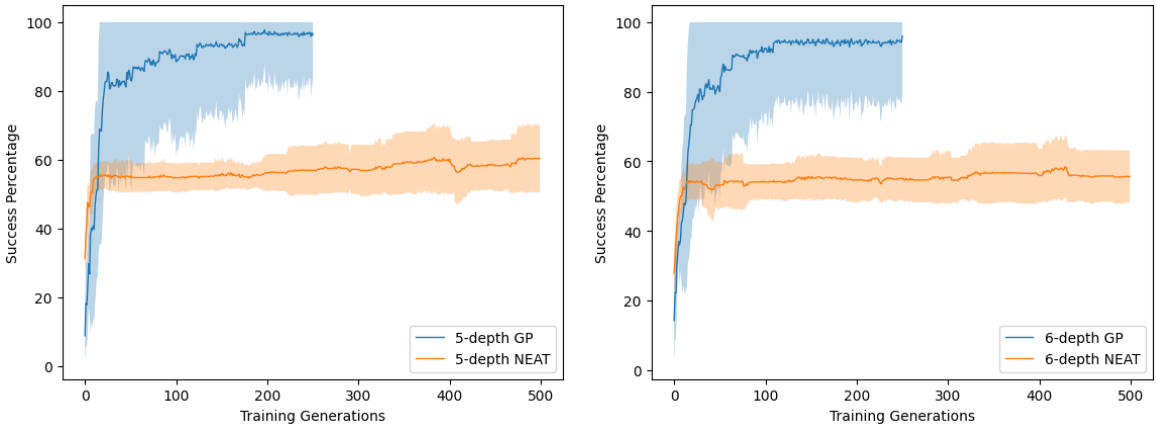

Figure 4.15: **GP vs NEAT 5- and 6-depth Sequence Classification Task Results.** Figure shows the difference between GP and NEAT setups evolving solution for 5-depth and 6-depth sequence of varied sequence lengths using mean and standard deviation of 20 runs. the x-axis shows the number of generations while the y-axis shows the fitness of the generation. Colored lines show the mean of the 20 runs. The shaded area shows the standard deviation of the 20 runs.

The following code represent the functions of the champion reached in GP:

```
# Tree1 represent Push action
tree1 = sub(add(ARG0, ARG1), ARG1)
# Tree2 represent Pop Head action
tree2 = sub(ARG0, protected_div(add(sub(ARG0, ARG0), add(add(ARG0
    , ARG1), ARG1)), protected_div(ARG0, protected_div(ARG0, ARG0)
    )))
# Tree3 represent No Action
tree3 = protected_div(protected_div(ARG0, ARG0), ARG0)
# Tree4 represent Pop Tail action
tree4 = protected_div(sub(add(ARG0, ARG1), add(add(ARG1, ARG0),
    ARG0)), ARG0)
```

Listing 4.3: Sequence Classification Champion Code

As the task becomes more complex we see the performance trend continue between the two platforms. Figures 4.15 and 4.9 verifies the findings we had so far. GP continues to perform better than NEAT regardless of the extra number of generations provided to NEAT. Optimal solutions were reached in almost every run by GP, while NEAT still lags behind and can not find a solution.

Generalizing Task: As we did with other tasks, we took the champion of GP

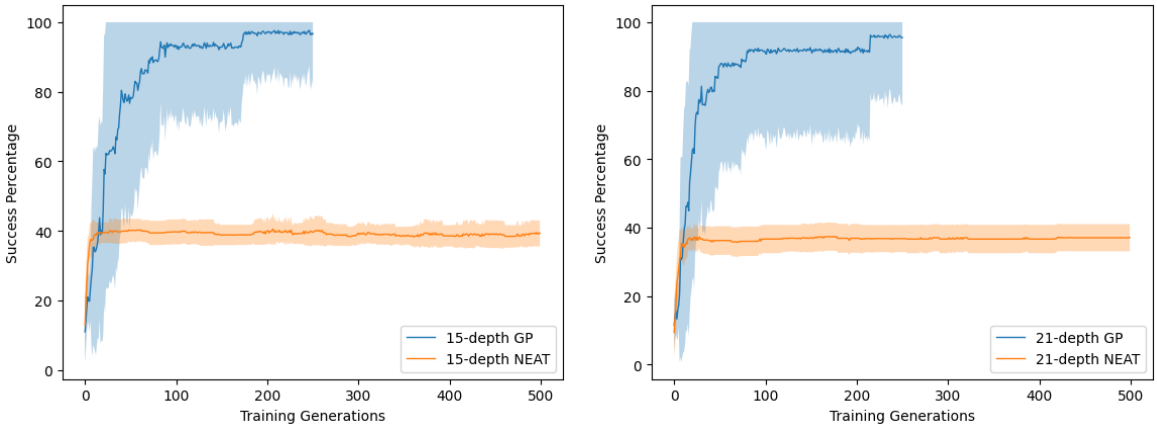

Figure 4.16: **GP vs NEAT 15- and 21-depth Sequence Classification Task Results.** Figure shows the difference between GP and NEAT setups evolving solution for 15-depth and 21-depth sequence of varied sequence lengths using mean and standard deviation of 20 runs. the x-axis shows the number of generations while the y-axis shows the fitness of the generation. Colored lines show the mean of the 20 runs. The shaded area shows the standard deviation of the 20 runs.

for the 21-depth task and tested it with even higher depths. Results showed GP champions had the ability to generalize to more complex tasks. We tested with 50-depth and 100-depth and the agent managed to find solutions. In conclusion, once a solution is found it can be generalized without any issues.

Table 4.10 summarizes the generalization performance of GP versus NEAT. The supremacy of GP is again very obvious, the results from NEAT is nowhere to be compared with how well GP is handling the task.

	GP (test)		NEAT (training)	
	Mean	SD	Mean	SD
4-depth	100	0.0	59.33	7.88
5-depth	96.3	16.12	60.44	9.87
6-depth	96.0	17.43	55.60	7.52
15-depth	96.6	14.82	39.44	3.64
21-depth	95.5	19.61	37.14	3.96

Table 4.10: Table shows final generation Mean and Standard Deviation for sequence classification NEAT and GP.

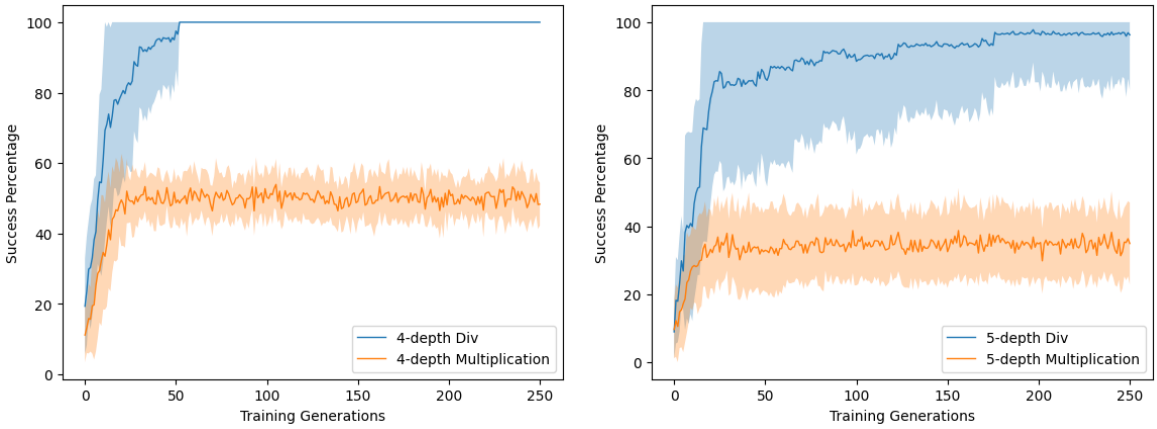

Figure 4.17: **Div vs Multiplication 4- and 5-depth Sequence Classification Task Results.** Figure shows the difference between Div and Multiplication setups evolving solution for 4-depth and 5-depth sequence of varied sequence lengths using mean and standard deviation of 20 runs. the x-axis shows the number of generations while the y-axis shows the fitness of the generation. Colored lines show the mean of the 20 runs. The shaded area shows the standard deviation of the 20 runs.

4.3.2 Replacing the protected division instruction

Just like our other experiments, we wanted to study the effect of the tools given to the agent on the performance of the agent. We start by switching instruction set from Div to Multiplication (Table 4.1). We tested only with 4-depth and 5-depth tasks (Figure 4.17). It is readily apparent that only when the protected division operator appears can GP solve the task. Table 4.11 shows in numbers the difference in performance between div and Multiplication instructions sets.

	Div		Multiplication	
	Mean	SD	Mean	SD
4-depth	100	0.0	48.3	6.07
5-depth	96.3	16.12	34.9	12.0

Table 4.11: Table shows final generation Mean and Standard Deviation for sequence classification Div and Multiplication instruction sets in GP.

4.3.3 Complex instruction set

Now we compare the minimalist 'Div' instruction set to the 'Full' instruction set, i.e. arithmetic, logical and constants (Table 4.1). The Full instruction set contains a wide variety of options for the agent to choose from, giving us the opportunity to study the effect of having too many choices on the complexity of the agent. Figure 4.18 shows the difference between the two instruction sets in an attempt to solve the 4-depth Sequence Classification task. The figure shows that Full instruction set has a slight edge over Div instruction set in terms of how fast it can reach a solution. This edge comes with the price of the complexity of the champion agent. An example champion in GP using Full operators has the form:

```
# Tree1 represent Push action
tree1 = mul(ARG0, if_then_else(True, ARG1, ARG1))
# Tree2 represent Pop Head action
tree2 = mul(mul(if_then_else(True, ARG0, ARG1), protected_div(
    ARG0, ARG1)), ARG1)
# Tree3 represent No Action
tree3 = protected_div(mul(if_then_else(True, ARG1, ARG1),
    if_then_else(eq(17.256515950005948, protected_div(
    protected_div(ARG1, 91.88959726562793), protected_div(ARG0,
    ARG0))), mul(ARG1, ARG0), mul(ARG0, ARG0))), ARG0)
```

Listing 4.4: Sequence Classification Full Champion Code

Continuing with the same trend, data from Figure 4.19 shows the difference between the two instruction sets become more obvious as the complexity of the task increases. That fact is confirmed when we look at Figure 4.20 for the 15- and 21-signal depth versions of the task. As we reached the 21-depth sequence the difference between the two instruction sets become clearer. Full instruction set has the upper hand in this task not only in terms of the number of generations to reach a solution but also by the consistency with which solutions are found (illustrated by the size of the standard deviation).

Table 4.12 illustrate how close the results of Div and Full instructions set toward the end. Both of these instruction sets are performing at a close result range.

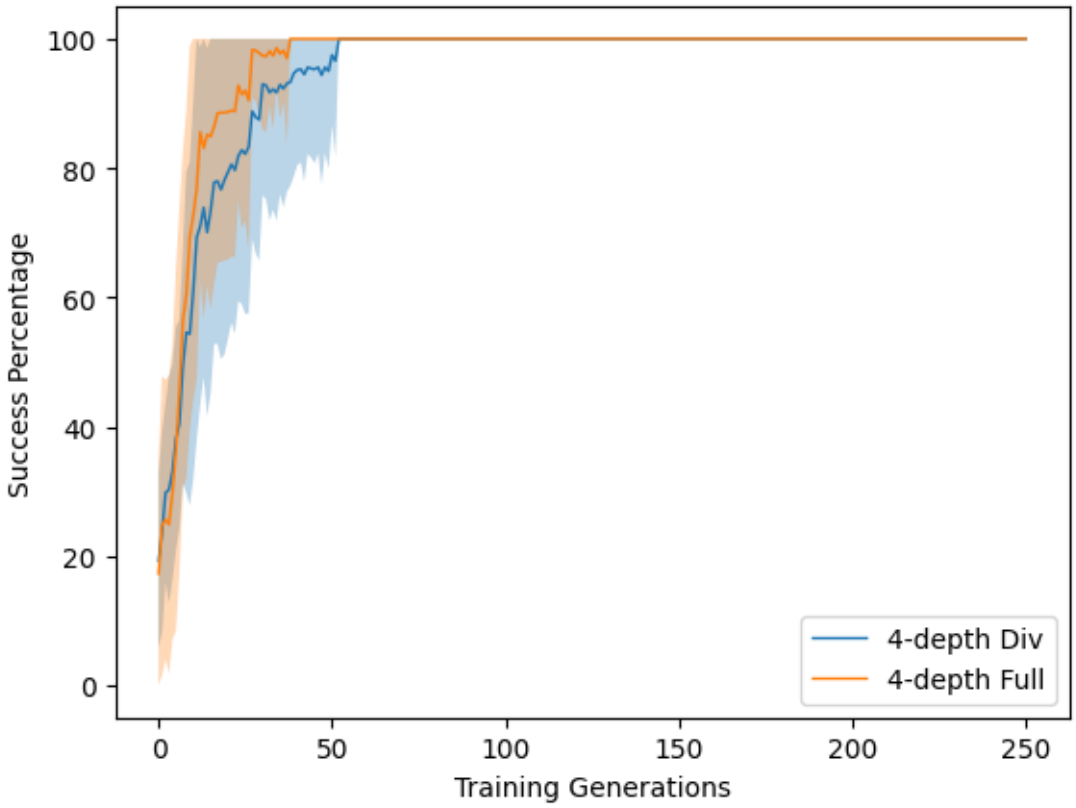

Figure 4.18: **Div vs Full 4-depth Sequence Classification Task Results.** Figure shows the difference between Div and Full setups evolving solution for a 4-depth sequence of varied sequence lengths using mean and standard deviation of 20 runs. the x-axis shows the number of generations while the y-axis shows the fitness of the generation. Colored lines show the mean of the 20 runs. The shaded area shows the standard deviation of the 20 runs.

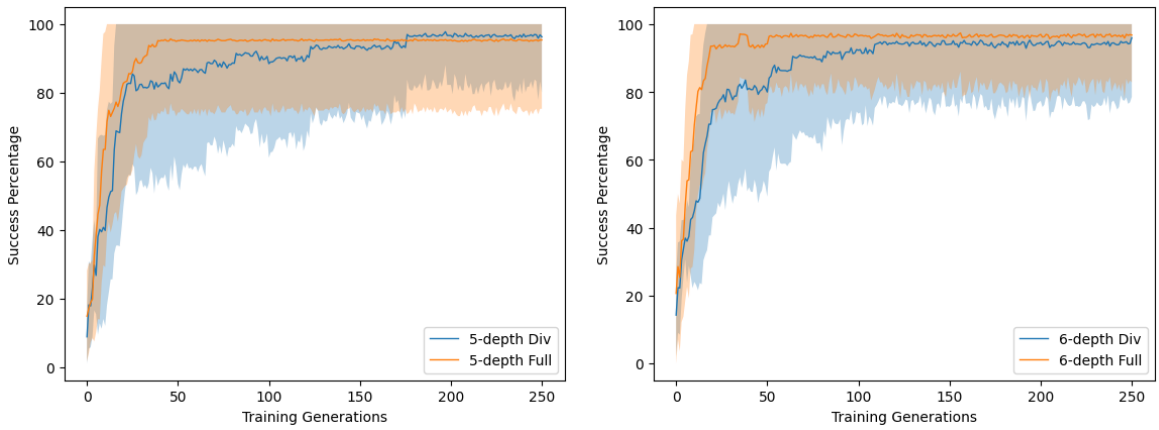

Figure 4.19: **Div vs Full 5- and 6-depth Sequence Classification Task Results.** Figure shows the difference between Div and Full setups evolving solution for 5-depth and 6-depth sequence of varied sequence lengths using mean and standard deviation of 20 runs. the x-axis shows the number of generations while the y-axis shows the fitness of the generation. Colored lines show the mean of the 20 runs. The shaded area shows the standard deviation of the 20 runs.

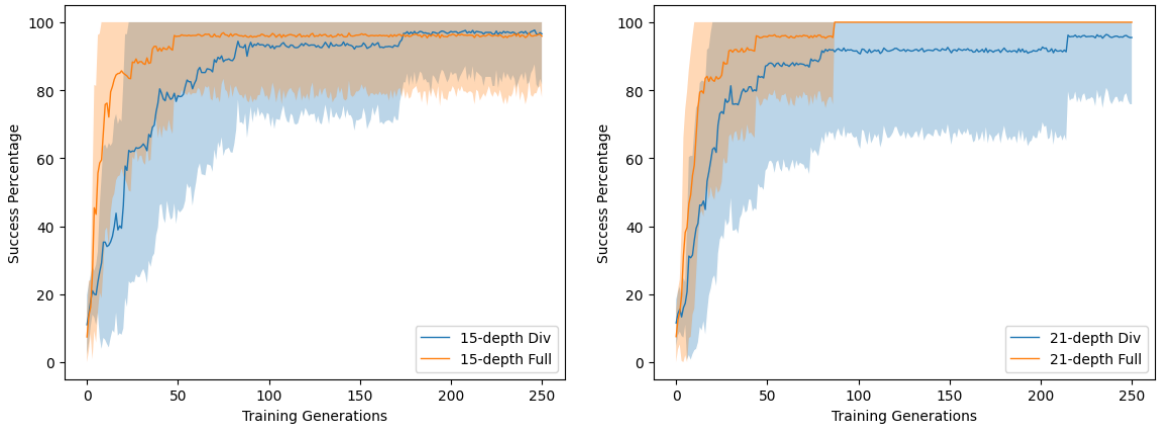

Figure 4.20: **Div vs Full 15- and 21-depth Sequence Classification Task Results.** Figure shows the difference between Div and Full setups evolving solution for 15-depth and 21-depth sequence of varied sequence lengths using mean and standard deviation of 20 runs. the x-axis shows the number of generations while the y-axis shows the fitness of the generation. Colored lines show the mean of the 20 runs. The shaded area shows the standard deviation of the 20 runs.

	Div		Full	
	Mean	SD	Mean	SD
4-depth	100	0.0	100	0.00
5-depth	96.3	16.12	95.4	20.05
6-depth	96.0	17.43	96.8	13.94
15-depth	96.6	14.82	95.9	17.87
21-depth	95.5	19.61	100	0.00

Table 4.12: Table shows final generation Mean and Standard Deviation for sequence classification Div and Full instruction sets in GP.

4.3.4 Noisy data with protected division instruction set

Finally, we altered the setup of the task to remove the opportunity to explicitly trap the zeros in the input sequences. To do so, we replaced the inputs taking a value of zero and replaced them with a random number generated between $[+\beta, -\beta]$ and ran the programs. The process was repeated over three sets of $\beta \in \{0.5, 0.25, 0.125\}$. Figure 4.21 shows the results of the original task run along with the three runs of the β-noise tasks for a signal sequence depth of 4. We can immediately see that the original 'clean' version of the task setup outperformed all of the other variations. The range of the β-noise introduced has an effect on the difficulty of the task. However, there is not a simple 'linear' relationship between the amount of β-noise and task difficulty. From the perspective of average performance reached under test: $\beta = 0.25 > 0.125 > 0.5$.

We continue our test and add to the complexity of the task by running it at a higher signal sequence depth. Figure 4.11 shows results from running tests on 5-depth and 6-depth respectively. The same trend continues here too, we can see that the original task still performs best (i.e. is easiest) while the agent never returns all 20 runs with an ideal solution.

In the 5-sequence depth task we do not see any change from the previous figure, 0.125 still performs better than 0.5 but worst than 0.25. We believe that 5-depth sequence wasn't long enough for the range noise to show full impact on agent performance. On the other hand, looking at the 6-depth figure we can see a change in the pattern. Notice how 0.125 now performs better than 0.25 and 0.5 in all runs. As the complexity increases in Figure 4.23 we see the previous trend grows clearer, the smaller the range used the better performance we get. That said, we also see a decline in the overall performance. The deeper the sequence, the harder it is for the

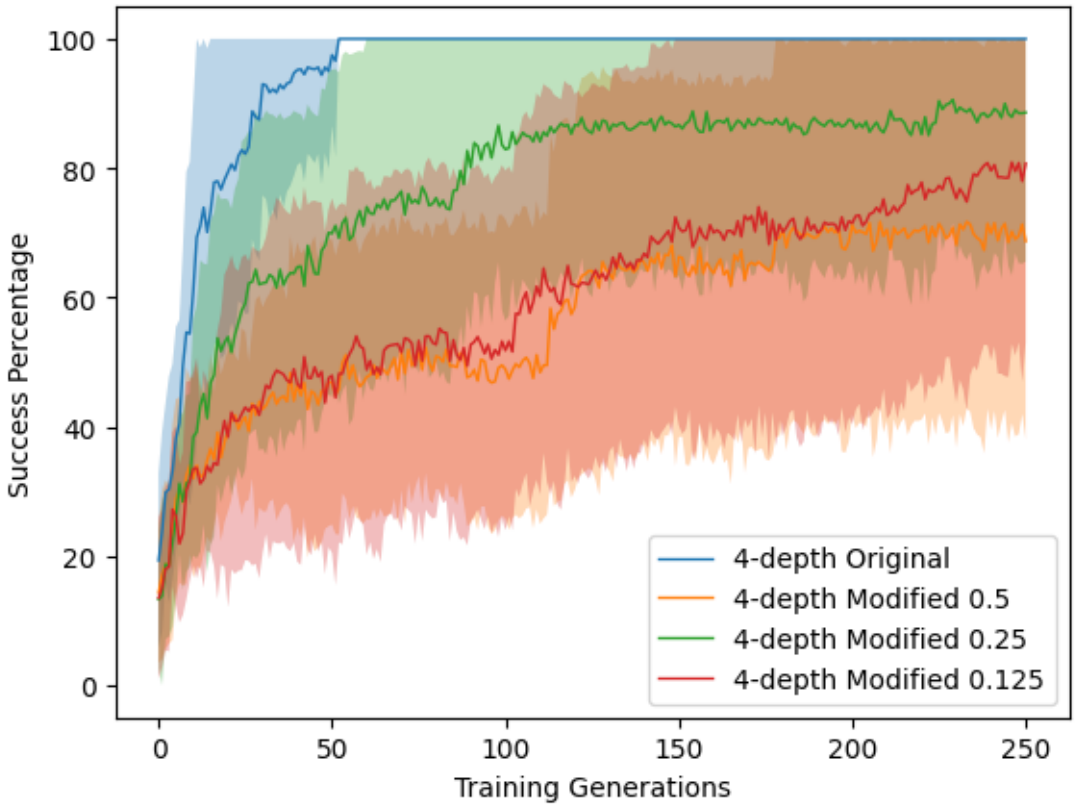

Figure 4.21: **Original vs Modified 4-depth Sequence Classification Task Results.** Figure shows an evolving solution for a 4-depth sequence of varied sequence lengths using mean and standard deviation of 20 runs including modifications. the x-axis shows the number of generations while the y-axis shows the fitness of the generation. Colored lines show the mean of the 20 runs. The shaded area shows the standard deviation of the 20 runs

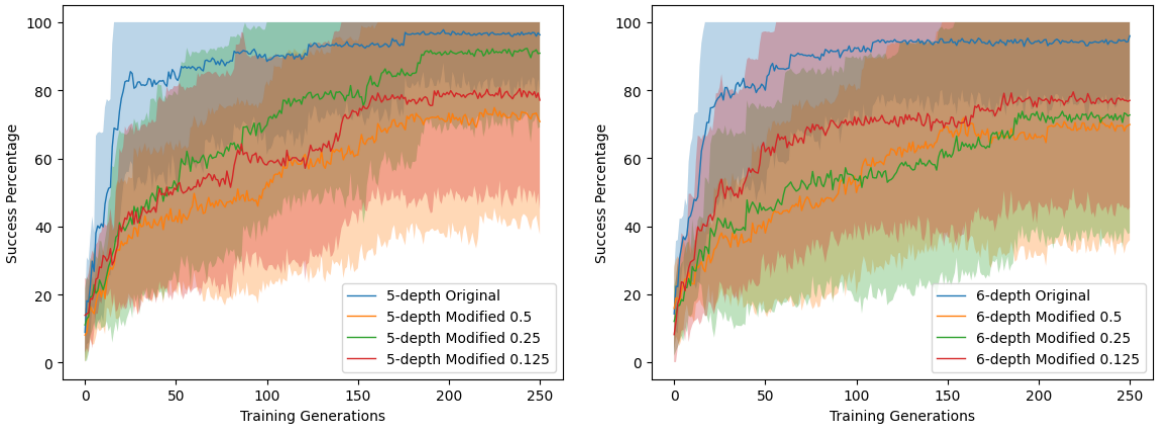

Figure 4.22: **Original vs Modified 5- and 6-depth Sequence Classification Task Results.** Figure shows the evolving solution for 5-depth and 6-depth sequence of varied sequence lengths using mean and standard deviation of 20 runs including modifications. the x-axis shows the number of generations while the y-axis shows the fitness of the generation. Colored lines show the mean of the 20 runs. The shaded area shows the standard deviation of the 20 runs

agent to find a solution.

Table 4.13 shows in numerical values the results seen in figures 4.21, 4.22 and 4.23. We can see as the task is in lower complexity (4-, 5- and 6-depth), modification with range '0.25' has the better performance. But, as the task gets more complicated (15-depth and 21-depth) the smaller the range the better results we are getting.

	Original		Modified 0.5		Modified 0.25		Modified 0.125	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD
4-depth	100	0.0	68.7	30.66	88.6	23.02	80.7	27.62
5-depth	96.3	16.12	70.8	33.07	90.9	21.68	77.1	31.94
6-depth	96.0	17.43	69.9	34.0	72.7	34.67	77.0	31.45
15-depth	96.6	14.82	47.6	38.88	60.5	39.68	63.9	40.08
21-depth	95.5	19.61	36.3	37.46	37.5	36.56	70.6	40.22

Table 4.13: Table shows final generation Mean and Standard Deviation for sequence classification Original and Modified instruction sets in GP.

4.3.5 Generalization

In order to test the capacity to generalize the solution we did the following. First, we are using the original task setup using noise value of 10 and Div instruction set from

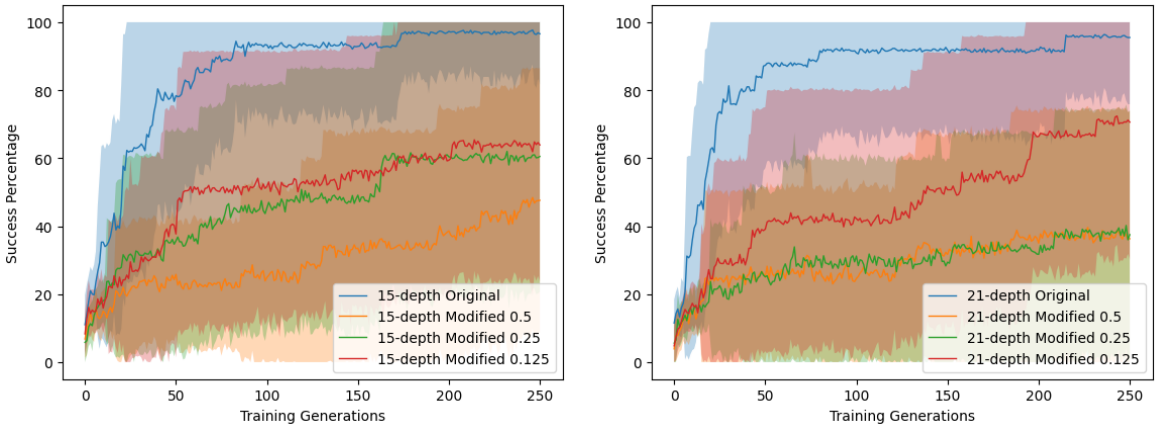

Figure 4.23: **Original vs Modified 15- and 21-depth Sequence Classification Task Results.** Figure shows the evolving solution for 15-depth and 21-depth sequence of varied sequence lengths using mean and standard deviation of 20 runs including modifications. the x-axis shows the number of generations while the y-axis shows the fitness of the generation. Colored lines show the mean of the 20 runs. The shaded area shows the standard deviation of the 20 runs

table 4.1. Second, we trained 20 different champions. Each of these champions will be tested on 50 and 100 depth. Third, we generate 50 random tests for each champion on each length. Then, we ran the tests and recorded how well the champions managed to solve the task.

Table 4.14 shows the Mean and Standard Deviation of the test of the champion. Looking at the table we can see that the champion kept the success rate and managed to successfully scale with the problem. Both the 50 and 100 depth layouts presented a challenge judging by the slightly higher standard deviation but the champion managed to score high in both layouts.

	Mean	**SD**
50 depth	95.1	15.9
100 depth	93.7	19.4

Table 4.14: Table shows Mean and Standard Deviation for sequence classification GP 20 champions generalizing on 50 and 100 depth.

4.4 Copy Task

In the copy task [7, 9] the agent has to memorize and then recall a long sequence of random binary vectors (words). The program is given a single bit delimiter to indicate the beginning of the memorization process, a sequence of random bit vectors to be memorized, then a single bit delimiter to indicate the beginning of the recall process. Figure 4.24 shows this process in more detail. The first step is to initialize the program with the start bit set to unity. In each following step, the program receives a random bit vector (element bits, Figure 4.24) until a single delimiter bit is set to unity, indicating the end of the memorization phase and the beginning of the recall phase. Once the delimiter bit is set, then there are no further inputs, hence the 'element' bits are also set to unity.

The program was trained to copy a sequence of random 8-bit vectors, where the sequence length was randomly generated between 10 and 20 vectors long. The target sequence is simply a copy of the input sequence without the delimiter or starts flags. The task is abstracted in a way that delimiters are the only thing that matters, based on the delimiter passed the program needs to maintain awareness of which phase is currently in effect and act on it. The goal is to be able to retain arbitrary length sequences of bit vectors. Testing will be performed over multiple lengths of sequence.

The fitness of a program is calculated by how well it can perform actions based on the delimiters. For instance, the example in Figure 4.24 should follow the flow of the following step by step table:

index	input	**Description**	**Action**
0	[1,0]	indicates the beginning of memorization	None
1	[0,0]	commit info to memory	Push
2	[0,0]	commit info to memory	Push
3	[0,0]	commit info to memory	Push
4	[0,0]	commit info to memory	Push
5	[0,1]	indicates the beginning of recall	None
6	[0,0]	retrieve from memory	Pop_Head
7	[0,0]	retrieve from memory	Pop_Head
8	[0,0]	retrieve from memory	Pop_Head

index	input	Description	Action
9	[0,0]	retrieve from memory	Pop_Head

Table 4.15: Table shows step by step simulation for agent processing copy task input

Leaving us with the following list of actions:

None | Push | Push | Push | Push | Push | None | Pop_head | Pop_head | Pop_head | Pop_head | Pop_head

Each of these actions carry a weight of 1, the fitness will be calculated up until performed action does not match the expected action. Training is repeated for 50 separate iterations of sequences with random lengths between 10 and 20. The list data structure was used as memory for this task with 4 actions: `push, pop_head, pop_tail, no_op`. When evolving the program there are a few setting to modify the copy task complexity:

- **Sequence Length:** One parameter that can control the difficulty level of the task is the length of the sequence the program has to memorize and recall. In the experiment by [7] the length was set between 1 and 20. On the other hand, in [9] experiment it was set to be between 1 and 10 to address the generalization question. In our task the length is set to be randomly chosen between 10 and 20, the random generation was chosen to prevent the program from over-fitting for the training data and be able to generalize to longer sequences that were never seen before.

- **Bit Vector Size:** This parameter represents the number of bits in each vector, i.e the number of values that must be stored in each step. In [9] experiment 1-bit, 2-bit, 4-bit, and 8-bits long were experimented with. In [7] experiment only worked with 8-bit in their program. In our experiment, we also assumed 8-bits vectors.

- **Iterations:** Just like [9] we have chosen to work with 50 separate iterations of random sequences with varied lengths. This was done to ensure that solutions do

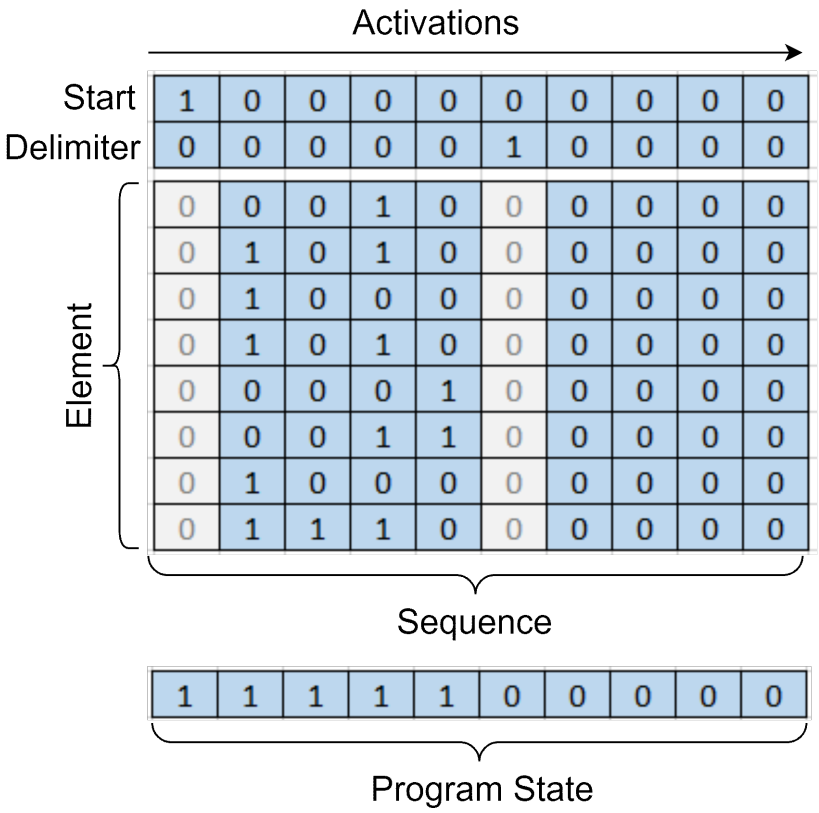

Figure 4.24: Copy Task Sequence example showing Program State of every step.

not optimize toward one specific length, but can generalize to different sequence lengths.

Program State: In tasks where it is important to know what state the program is in and act based on that, we introduced the state as an extra output (making the total of 5 outputs for the task). Program state will be figured out by the agent and will be fed back as input. Let's take Copy Task as an example, the programs are given delimiters for memorizing, and everything after that until the recall delimiter will have to be pushed to memory. Program state will be used as a reminder of what state the programs are in (memorize or recall). Figure 4.24 shows an example sequence and the value of 'Program State' produced by the agent when processing the sequence. Figure 4.25 shows Copy Task GP structure, in it, we can see the interaction between agent and memory and how programs state is passed from one evolutionary cycle to another.

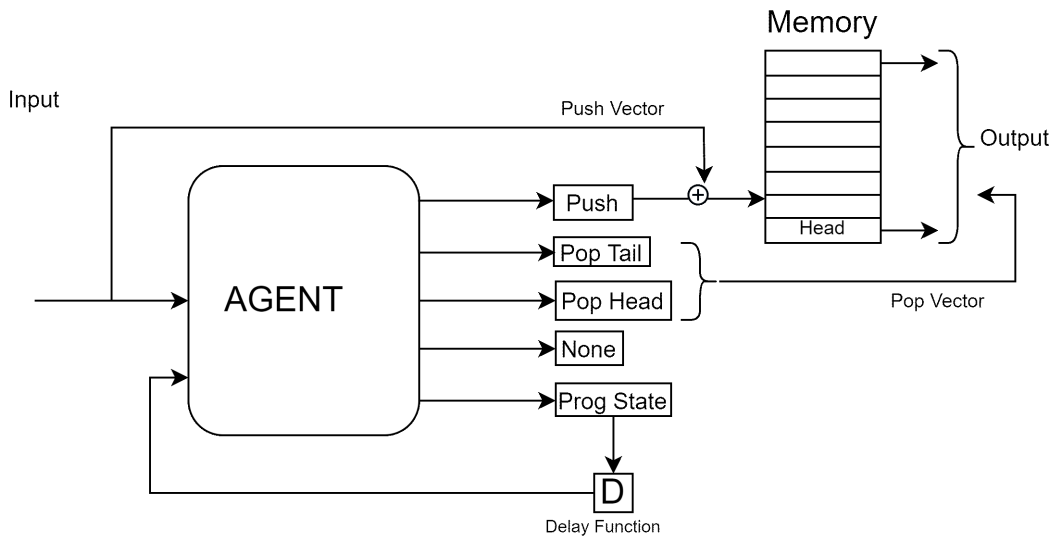

Figure 4.25: GP Copy Task Stack Structure.

4.4.1 GP versus NEAT

Results shown in Figure 4.26 represent 20 runs of 250 generations in GP and 500 generations in NEAT. The GP fitness of each generation was calculated by evaluating the champion of the training generation on a new randomly generated sequence and recording the evaluation results, i.e. the GP curve reflects test performance. While NEAT fitness was calculated based on the fitness of the champion during training, i.e. the NEAT curves reflect training alone. GP run was done using the Div operators from Table 4.1 and NEAT configuration file had the parameters from Table 4.3 with num-inputs = 3 (two delimiters and input from memory stack) and num-outputs = 5 (Push, Pop Head, Nothing, Pop Tail and Progress State).

Figure 4.26 shows the mean and standard deviation of the 20 runs combined. Looking at the figure we can see a solution was reached within the first 50 generations in the case of GP while NEAT struggled with this task and was unable to reach a full solution. Taking the solution from GP, we were able to test the capacity to generalize by testing it on newly generated sequences of lengths 20, 50, and 100, i.e. only the case of sequences of length 20 were encountered during training. The following code represents the solution discovered by a GP champion:

```
# Tree1 represent Push action
tree1 = sub(ARG1, ARG1)
# Tree2 represent Pop Head action
tree2 = sub(protected_div(protected_div(ARG0, ARG2), ARG2), ARG1)
# Tree3 represent No Action
tree3 = add(add(add(ARG1, ARG1), ARG0), protected_div(ARG0, add(
    protected_div(ARG1, ARG1), protected_div(ARG1, ARG1))))
# Tree4 represent Pop Tail
tree4 = add(ARG2, ARG1)
# Tree5 represent Progress State
tree5 = sub(add(ARG2, ARG1), ARG0)
```

Listing 4.5: Copy Task Champion Code

Framework	Mean	SD
GP (test)	95.0	21.79
NEAT (training)	53.40	0.65

Table 4.16: Table shows final generation Mean and Standard Deviation for copy task NEAT and GP.

Table 4.16 defines test performance of the champions from the 20 GP runs with the 'Div' instruction set from Table 4.1. Clearly, test performance of GP exceeds the training performance under NEAT. Moreover, this is achieved in a tenth of the number of generations than under NEAT.

4.4.2 Replacing the protected division instruction

Having established the utility of GP and the list data structure, we now turn our attention to the role of the instruction set (currently add, subtract, and division). The 'protected division' operation is supposed to act as a conditional statement under the Copy Task. In order to test this, the protected division will be replaced with multiplication (i.e. an unprotected operator) or the Multiplication operators from Table 4.1

Figure 4.27 compares the two GP instruction sets (20 separate runs). Both were able to find solutions within the first 50 generations. Moreover, the 'multiplication' instruction set was even more effective, i.e. faster convergence with lower standard

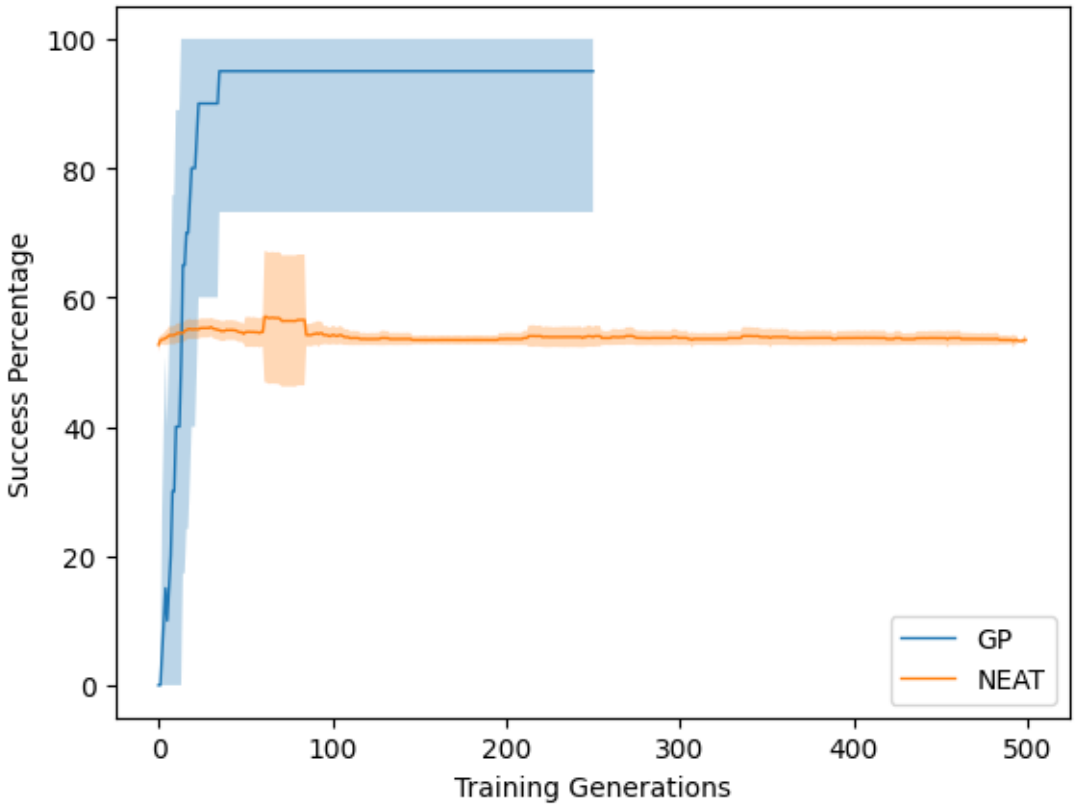

Figure 4.26: **GP vs NEAT Copy Task Results.** Figure shows the evolving solution for 8-bit vectors of varying lengths using mean and standard deviation of 20 runs. the x-axis shows the number of generations while the y-axis shows the fitness of the generation. The colored line shows the mean of the 20 runs. The shaded area shows the standard deviation of the 20 runs.

deviation (Table 4.17). The following code represents the solution found by GP using Multiplication instructions set to solve Copy Task:

```
# Tree1 represent Push action
tree1 = sub(sub(ARG2, ARG1), add(ARG1, mul(mul(ARG0, ARG2), ARG1)
    ))
# Tree2 represent Pop Head action
tree2 = mul(ARG2, mul(ARG0, ARG0))
# Tree3 represent No Action
tree3 = add(add(ARG2, ARG0), add(add(ARG2, ARG2), ARG1))
# Tree4 represent Pop Tail
tree4 = mul(ARG2, sub(mul(ARG1, ARG1), add(ARG2, ARG2)))
# Tree5 represent Progress State
tree5 = sub(ARG2, add(mul(ARG0, ARG2), ARG1))
```

Listing 4.6: Copy Task Multiplication Champion Code

instruction set	Mean	SD
Div	95.0	21.79
Multiplication	100	0.0

Table 4.17: Table shows final generation Mean and Standard Deviation for copy task GP Div and Multiplication instruction set.

4.4.3 Complex instruction set

At this point, we have established that simple solutions to the 'Copy Task' can be discovered efficiently under the proposed approach. Conversely, using NEAT in place of the weakly cooperative GP framework (GP) did not manage to discover such solutions. We now introduce a 'complex' instruction set in which all arithmetic and logical operators appear (Table 4.1).

The hypothesis is that this would lead to a larger search space and likely more complex solutions being discovered after more generations. Results in figure 4.28 shows that Div instruction set was able to find a solution almost immediately, while Full instruction set struggled with the task and in some runs were unable to find a solution. Looking at table 4.18 which shows the last generation testing results of 20 different runs, we can see the difference in performance between Div and Full instruction sets both in 'Mean' and 'Standard Deviation'. The solution found was

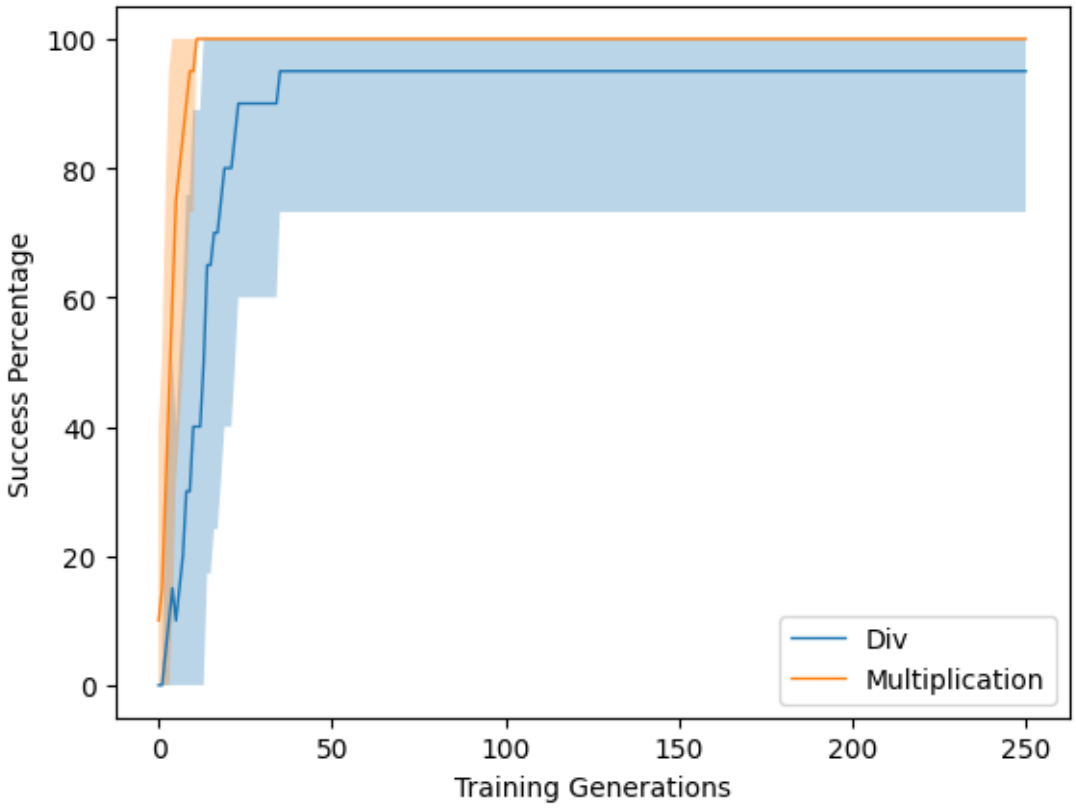

Figure 4.27: **Div vs Multiplication Copy Task Results.** Figure shows the difference in performance between Div and Multiplication instruction sets. Results represent mean and standard deviation of 20 runs each with varied vectors length. The colored line shows the mean of the 20 runs. The shaded area shows the standard deviation of the 20 runs.

in some cases over complicated. The following code represents the functions of the champion using Full operators in GP:

```
# Tree1 represent Push action
tree1 = mul(if_then_else(lt(mul(protected_div(ARG1, ARG2), ARG0),
    mul(ARG1, 71.99986108166674)), mul(mul(ARG0, if_then_else(True,
    ARG0, ARG1)), if_then_else(False, 7.860610953699998, ARG2)),
    protected_div(12.090939012041579, ARG2)), ARG1)
# Tree2 represent Pop Head action
tree2 = protected_div(mul(protected_div(if_then_else(True, ARG0,
    ARG0), if_then_else(True, 2.1023390046363333, ARG0)),
    protected_div(ARG2, ARG0)), mul(ARG2, ARG2))
# Tree3 represent No Action
tree3 = if_then_else(eq(mul(ARG0, 60.12377982639531), if_then_else(
    lt(ARG0, ARG1), mul(ARG1, 59.03890437566138), if_then_else(True,
    ARG0, mul(if_then_else(True, ARG1, ARG2), if_then_else(True,
    25.77960635715084, ARG1))))), ARG1, 56.123860744871514)
# Tree4 represent Pop Tail
tree4 = mul(if_then_else(True, ARG2, mul(mul(ARG2, mul(ARG2, ARG1)),
     ARG2)), mul(mul(70.98917616469029, ARG1), mul(ARG0,
    63.148039312409196)))
# Tree5 represent Progress State
tree5 = if_then_else(not_(True), if_then_else(True, mul(ARG2, ARG1),
     ARG0), protected_div(mul(41.70397912251834, ARG0), protected_div
    (ARG1, ARG2)))
```

Listing 4.7: Copy Task Full Champion Code

instruction set	Mean	SD
Div	95.0	21.79
Full	80.0	40.0

Table 4.18: Table shows final generation Mean and Standard Deviation for copy task GP Div and Full instruction set.

4.4.4 Noisy data with protected division instruction set

In this experiment under the Copy Task, we return to the minimalist instruction set based on the 'protected division' operator (or Div in Table 4.1). This time, however, we replaced the zero in the delimiter and start files to be -1 in an attempt to disrupt

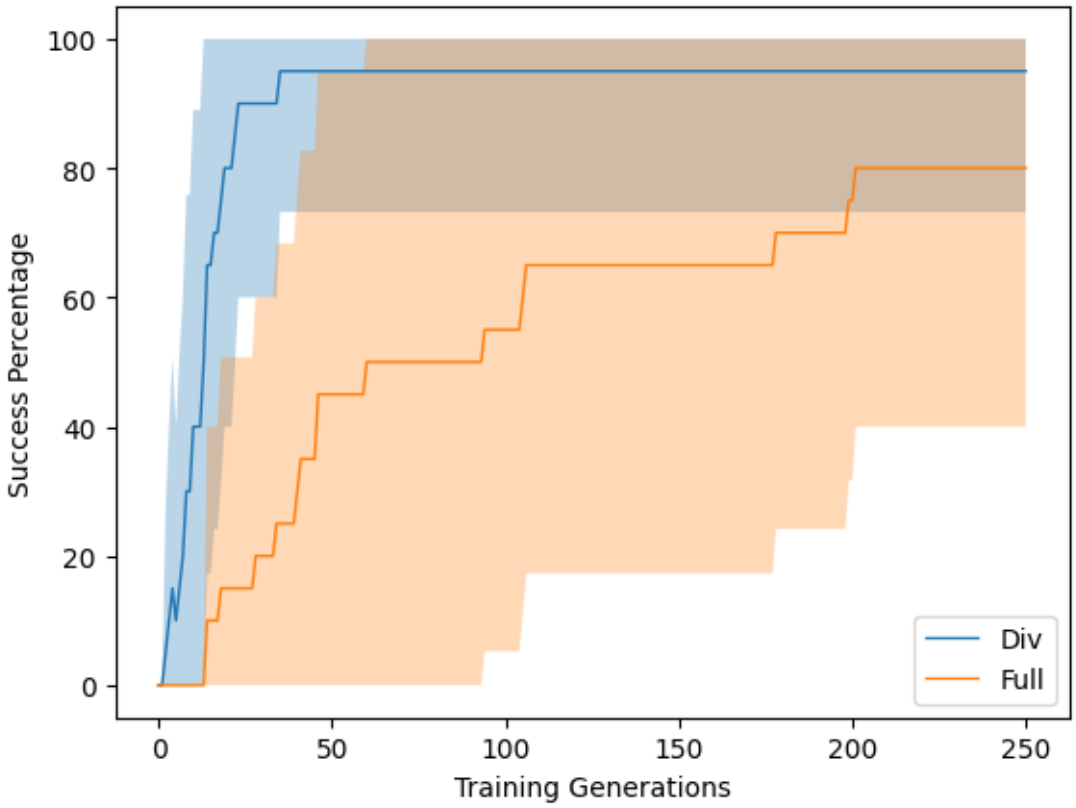

Figure 4.28: **Div vs Full Copy Task Results.** Figure shows the difference in performance between Div and Full instruction sets. Results represent mean and standard deviation of 20 runs each with varied vectors length. The colored line shows the mean of the 20 runs. The shaded area shows the standard deviation of the 20 runs.

the role of protected division in building a 'conditional' statement. Having a zero in the input allows the 'protected division' operator to behave as a conditional operator, replacing the zero with -1 will take away that advantage and forces the agent to find another way to deal with the input. Thus, the start of the memorization delimiter is now [1,-1] instead of [1,0] and the recall delimiter is now [-1,1] rather than [0,1] (see Table 4.24). We used the Div instruction set from table 4.1 and ran the experiment for 20 once again.

Results in figure 4.29 show that in both cases the agent was able to find a solution to solve the problem. What is interesting now is that the modified version of the task slows down the pace of evolution, taking in the order of 125 generations to solve the task, but all runs appear to generalize. That is to say, all 20 champions generalize to solve all test cases (Table 4.29). Looking at table 4.19 which shows the last generation testing results of 20 different runs, we can see the results in numerical expression.

instruction set	Mean	SD
Original	95.0	21.79
Modified	100	0.0

Table 4.19: Table shows final generation Mean and Standard Deviation for copy task GP Original and Modified instruction sets.

Combined results in figure 4.30 summarizes all the different experiments performed with the Copy Task using GP. We note that the agent performed best using Multiplication instruction set from table 4.1 and had the worst results using the Full instruction set. Div instruction set came third. However, applying the Div instruction set with '-1' replacing '0' in the task definition resulted in slower convergence, but better generalization than Div under the original task definition. This tends to suggest that the 'conditional' statement operation that appears with protected division and the '0' style task definition results in early convergence to sub-optimal champions. Introducing a problem definition without '0' results in a slower rate of convergence, but all runs then perform optimally. The 'multiplication' instruction set appears to achieve this under the task definition based on '0'.

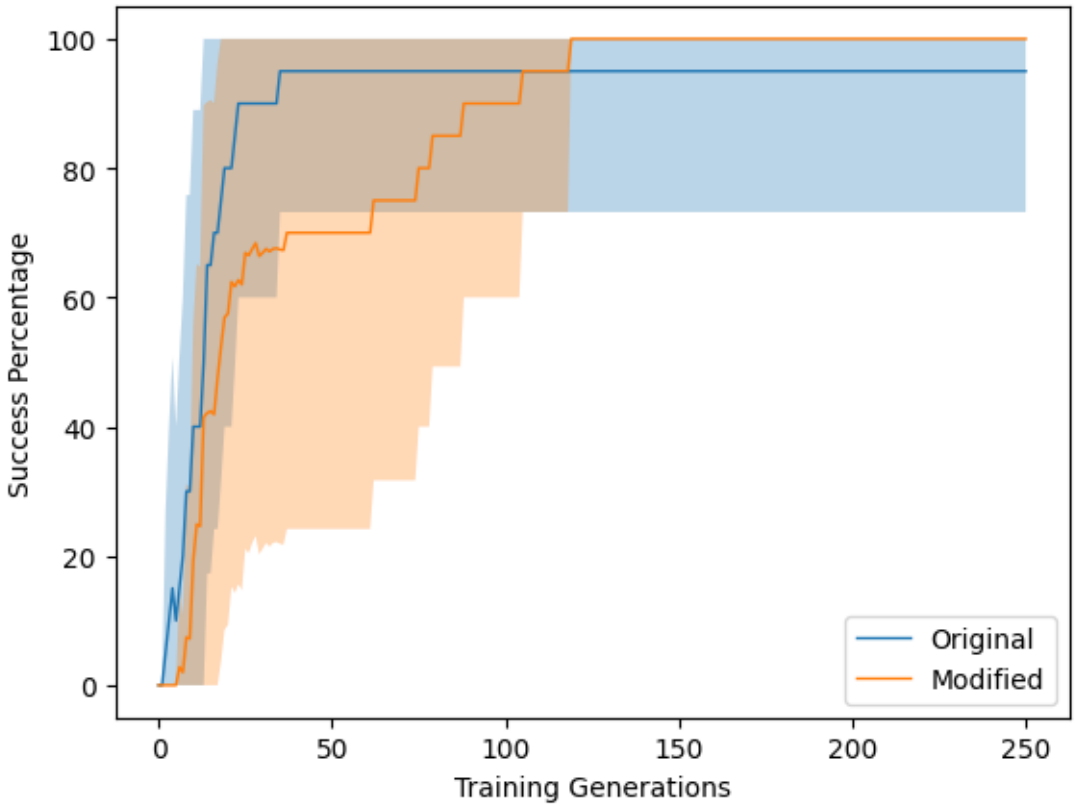

Figure 4.29: **Original vs Modified Copy Task Results.** Figure shows the difference in performance between Original and Modified settings of Copy Task. Results represent mean and standard deviation of 20 runs each with varied vectors length. The colored line shows the mean of the 20 runs. The shaded area shows the standard deviation of the 20 runs.

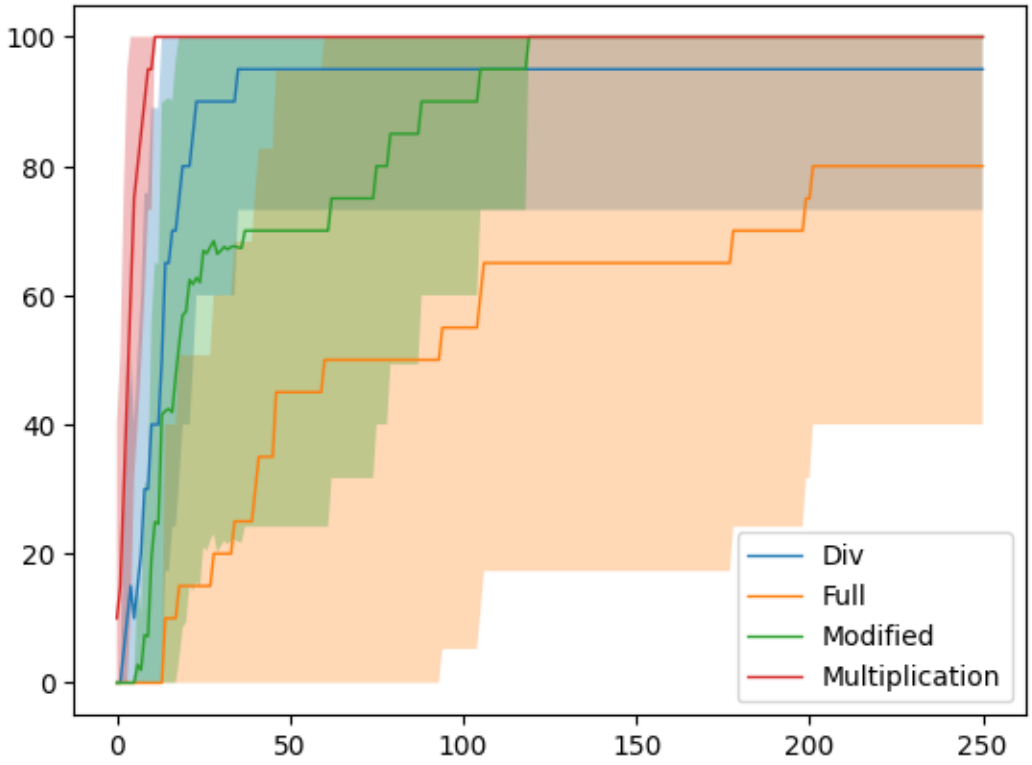

Figure 4.30: **GP Combined Copy Task Results.** Figure shows all the different variations of the Copy Task done in GP. Results represent mean and standard deviation of 20 runs each with varied vectors length. The colored line shows the mean of the 20 runs. The shaded area shows the standard deviation of the 20 runs.

4.4.5 Generalization

In order to test the capacity to generalize the solution we did the following. First, we are using the original task setup using 8-bit length and Div configuration from table 4.1. Second, we trained 20 different champions. Each of these champions will be tested on 50 and 100 sequence lengths. Third, we generate 50 random tests for each champion on each length. Then, we ran the tests and recorded how well the champions managed to solve the task.

Table 4.20 shows the Mean and Standard Deviation of the test of the champion. We can clearly see that even with extremely long sequences the champion agent managed to ace those tests with no issues at all. Having a success rate of 100% and a standard deviation of 0.0 on both 50 and 100 length sequences proves that the agent can generalize with no problems at all.

	Mean	**SD**
50 length	100	0.0
100 length	100	0.0

Table 4.20: Table shows Mean and Standard Deviation for copy task GP 20 champions generalizing on 50 and 100 Sequence Length.

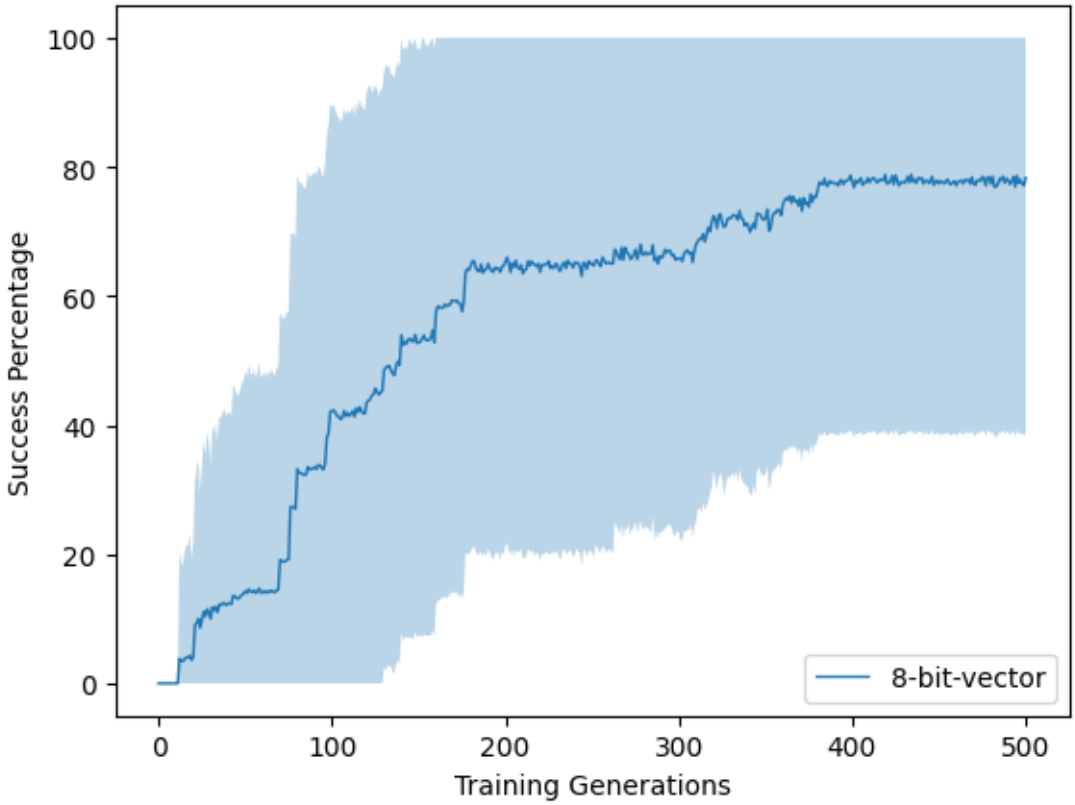

Figure 4.31: **Full Vector Copy Task Results.** Figure shows full vector variation of the Copy Task done in GP. Results represent mean and standard deviation of 20 combined runs random length vectors. The colored line shows the mean of the 20 runs. The shaded area shows the standard deviation of the 20 runs.

4.4.6 Full vector with protected division instruction set

In this last experiment under the Copy Task, we are using the minimalist instruction set based on the 'protected division' operator (or Div in Table 4.1). This time, however, we are using the full vector of 11-bit (8-bit to be copied, 2-bit as delimiters, and 1-bit 'progress state') as an input. With this setup, it will be harder for GP to evolve an agent since the agent needs to learn that the first two bits are the delimiters then learn what each of these delimiters mean. Due to the complexity of this setup, we altered the number of generations used from 250 (used in all previous tasks and setups) to 500. Looking at figure 4.31 we can see how challenging this variation was for GP demonstrated by the wide 'standard deviation'. despite that, GP managed to find solutions in most cases and solved the task.

In order to test the capacity to generalize the solution we did the following. First,

we trained 20 different champions. Each of these champions will be tested on 50 and 100 sequence lengths. Second, we generate 50 random tests for each champion on each length. Then, we ran the tests and recorded how well the champions managed to solve the task. Table 4.21 shows the Mean and Standard Deviation of the champions test.

	Mean	**SD**
50 length	90.13	15.99
100 length	90.05	19.66

Table 4.21: Table shows Mean and Standard Deviation for copy task GP 20 champions generalizing on full vector 50 and 100 Sequence Length.

4.5 Discussion

Looking back at the results found by this study and comparing with previous studies that tackled the same tasks, we can best illustrate these findings as follows:

- **Sequence Recall:** The previous work was done with 1- to 6-depth only, while our work expanded to 15-depth and 21-depth.

	GP 4.2	**NEAT 4.2**	**MMU [16]**
4-depth	100%	96.42%	≈64.0%
5-depth	95.1%	96.21%	≈55.0%
6-depth	100%	96.04%	46.3%
15-depth	95.0%	95.59%	NA
21-depth	100%	91.59%	NA
# Gen	250	500	10,000
Pop Size	100 x 4	400	100

Table 4.22: Compare results between our findings and work done by others.

- **Sequence Classification:** Previous work done on this task covered 1- to 6-depth only, we expanded that to work with 15-depth and 21-depth also.

	GP 4.2	NEAT 4.2	MMU [16]
4-depth	100%	59.33%	≈94.0%
5-depth	96.3%	60.44%	≈90.0%
6-depth	96.0%	55.60%	87.6%
15-depth	96.6%	39.44%	NA
21-depth	95.5%	37.14%	NA
# Gen	250	500	1,000
Pop Size	100 x 4	400	100

Table 4.23: Compare results between our findings and work done by others.

- **Copy Task:** In Graves et al. [7] didn't have comparable numbers with what we have so we are only listing our findings.

	GP 4.4	NEAT 4.4	GP Full Vector 4.4.6
8-bit	95.0	53.40	90.13
# Gen	250	500	500
Pop Size	100 x 5	400	100 x 5

Table 4.24: Compare our findings in two different variations and results of full vector.

Chapter 5

Conclusion

This thesis investigates the ability of GP to perform a suite of 'deep' memory tasks as popularized for benchmarking neural networks (Sequence Recall, Sequence Classification, and the Copy Task). We begin by recognizing that there are two basic approaches (§2): internal memory models and external memory models. An internal memory implies that some form of 'recurrent connectivity' exists in the representation that lets the learning agent retain internal state. Such a representation is capable of Turing complete computation (§2.1), but faces the difficulty of separating computation to suggest an action from computation to control memory. In this work, we therefore adopt an external memory model (§2.2). This means that the properties of memory have to be 'engineered in', thus another set of flexibility/ efficiency trade-offs. The agent was given control signals to manipulate the memory in terms of adding to memory, retrieving from memory, and not interacting with the memory. Having external memory made the task easier for GP to handle, especially in observable tasks like the one we have. Otherwise, an internal memory management would give the agent more flexibility in dealing with non-observable tasks.

Our approach was designed to operate within constraints set by canonical Tree structure GP (§3), i.e. programs may only produce a single output. In order to support multiple outputs, we assume a multi-population model in which a feasible learning agent can only be defined by sampling one program from each population or a modular coevolutionary framework. External memory was designed around the concept of a list that can be accessed as a first-in, first-out, or a first-in, last-out data structure.

The resulting empirical evaluation was designed to answer three basic questions (§4): 1) can a neural representation perform as effectively as GP with the external memory model we assume; 2) what influence will support for different instruction sets have; 3) how does changing the *task* representation affect the ability of GP to

discover solutions. With regards to question 1, GP was never worse than the neural network and was explicitly better in the Sequence Classification and Copy Task.

In the case of question 2, we provided three different types of instruction sets and assessed the impact this had on the performance of the agent. Protected Division played a very important role in evolving an agent that can solve our test cases, when replaced with multiplication the performance dropped drastically in two of the task while the multiplication set performed better on another task. We also showed the results of Full instruction set and how that would affect the agent. We showed that having a minimized instruction set will lead to clear outcomes and that including more instructions will lead to longer solutions but we will always find one.

Finally, with regards to question 3, this thesis investigated the effect of different types of noise on our deep memory tasks. The first type of noise was done by replacing zeros with another number (-1 or 1). Our motivation was to try to remove the perceived advantage that Protected Division was perceived to play. We implemented this type of noise in two tasks. Observed results were not majorly impacted by this change. The second type of noise was done by replacing zeros with a range of noise. We experimented with three different ranges (-0.5 to 0.5), (-0.25 to 0.25) and (-0.125 to 0.125). Ranged noise has a bigger impact on the complexity of the found solution and the performance of the agent. Results showed that in higher depth the narrower the range the better the results. Looking at 15-depth and 21-depth the best performance we got was with a noise range between -0.125 and 0.125.

Future work could be done by providing the agent with more options for the memory data structure. Rather than assuming a list data structure, the agent would have to learn to pick the right type of data structure as memory and then learn how to handle that type of memory. For example, associative memory represents a very different approach that would be useful for solving 'image' recall tasks as opposed to the sequence recall style tasks investigated in this thesis.

In this thesis, we only tested Neural Network with the original setup of tasks. It would also be interesting to see results from previous tasks in Neural Network using the modified setup with the range-type noise. GP solution can get complicated if the wrong instruction set is used. The tree structure could grow fast and consume a lot of memory and CPU cycles with very few benefits. We would recommend the start

with a small instruction set and adding more options slowly along the way of the development to keep complexity to a minimum. Selecting what functions would be used is completely related to the task at hand and how it's represented.

Bibliography

[1] Alexandros Agapitos, Anthony Brabazon, and Michael O'Neill. Genetic programming with memory for financial trading. In *Proceedings of European Conference on Applications of Evolutionary Computation: Part I*, volume 9597 of *LNCS*, pages 19–34. Springer, 2016.

[2] David Andre. Evolution of mapmaking: Learning, planning, and memory using genetic programming. In *Proceedings of the Conference on Evolutionary Computation*, pages 250–255. IEEE, 1994.

[3] David Andre. The evolution of agents that build mental models and create simple plans using genetic programming. In *Proceedings of the International Conference on Genetic Algorithms*, pages 248–255. Morgan Kaufmann, 1995.

[4] Scott Brave. The evolution of memory and mental models using genetic programming. In *Proceedings of the Annual Conference on Genetic Programming*, pages 261–266. MIT Press, 1996.

[5] John A. Doucette, Peter Lichodzijewski, and Malcolm I. Heywood. Hierarchical task decomposition through symbiosis in reinforcement learning. In *Genetic and Evolutionary Computation Conference*, pages 97–104. ACM, 2012.

[6] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. DEAP: evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175, 2012.

[7] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *CoRR*, abs/1410.5401, 2014.

[8] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwinska, Sergio Gomez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John P. Agapiou, Adrià Puigdomènech Badia, Karl Moritz Hermann, Yori Zwols, Georg Ostrovski, Adam Cain, Helen King, Christopher Summerfield, Phil Blunsom, Koray Kavukcuoglu, and Demis Hassabis. Hybrid computing using a neural network with dynamic external memory. *Nat.*, 538(7626):471–476, 2016.

[9] Rasmus Boll Greve, Emil Juul Jacobsen, and Sebastian Risi. Evolving neural turing machines for reward-based learning. In Tobias Friedrich, Frank Neumann, and Andrew M. Sutton, editors, *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference*, pages 117–124. ACM, 2016.

[10] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, 1997.

[11] Maarten Keijzer. Improving symbolic regression with interval arithmetic and linear scaling. In *Proceedings of the European Conference on Genetic Programming*, volume 2610 of *LNCS*, pages 70–82. Springer, 2003.

[12] Stephen Kelly and Malcolm I. Heywood. Emergent solutions to high-dimensional multitask reinforcement learning. *Evolutionary Computation*, 26(3), 2018.

[13] Stephen Kelly, Jacob Newsted, Wolfgang Banzhaf, and Cedric Gondro. A modular memory framework for time series prediction. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 949–957. ACM, 2020.

[14] Stephen Kelly, Robert J. Smith, Malcolm I. Heywood, and Wolfgang Banzhaf. Emergent tangled program graphs in partially observable recursive forecasting and vizdoom navigation tasks. *ACM Transcations on Evolutionary Learning and Optimization*, 1(3):1–41, 2021.

[15] Stephen Kelly, Tatiana Voegerl, Wolfgang Banzhaf, and Cedric Gondro. Evolving hierarchical memory-prediction machines in multi-task reinforcement learning. *Genetic Programming and Evolvable Machines*, 22(4):573–605, 2021.

[16] Shauharda Khadka, Jen Jen Chung, and Kagan Tumer. Neuroevolution of a modular memory-augmented neural network for deep memory problems. *Evol. Comput.*, 27(4):639–664, 2019.

[17] John R. Koza. *Genetic programming - on the programming of computers by means of natural selection*. Complex adaptive systems. MIT Press, 1993.

[18] William B. Langdon. *Genetic Programming and Data Structures*. Kluwer Academic, 1998.

[19] Xiao Luo, Malcolm I. Heywood, and A. Nur Zincir-Heywood. Evolving recurrent models using linear GP. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1787–1788. ACM, 2005.

[20] Nicholas Freitag McPhee and Riccardo Poli. Memory with memory: soft assignment in genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1235–1242. ACM, 2008.

[21] Ji Ni, Russ H. Drieberg, and Peter I. Rockett. The use of an analytic quotient operator in genetic programming. *IEEE Transactions on Evolutionary Compution*, 17(1):146–152, 2013.

[22] Aditya Rawal and Risto Miikkulainen. Evolving deep lstm-based memory networks using an information maximization objective. In Tobias Friedrich, Frank Neumann, and Andrew M. Sutton, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 501–508. ACM, 2016.

[23] H. Sebastian Seung. Continuous attractors and oculomotor control. *Neural Networks*, 11(7-8):1253–1258, 1998.

[24] Arlindo Silva, Ana Neves, and Ernesto Costa. Building agents with memory: An approach using genetic programmed networks. In *Proceedings of the Conference on Evolutionary Computation*, pages 1824–1833. IEEE, 1999.

[25] Robert J. Smith and Malcolm I. Heywood. Evolving Dota 2 shadow fiend bots using genetic programming with external memory. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 179–187. ACM, 2019.

[26] Robert J. Smith and Malcolm I. Heywood. A model of external memory for navigation in partially observable visual reinforcement learning tasks. In *Genetic Programming - 22nd European Conference, EuroGP 2019, Held as Part of EvoStar 2019, Leipzig, Germany, April 24-26, 2019, Proceedings*, volume 11451 of *Lecture Notes in Computer Science*, pages 162–177. Springer, 2019.

[27] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural network through augmenting topologies. *Evol. Comput.*, 10(2):99–127, 2002.

[28] Astro Teller. Turing completeness in the language of genetic programming with indexed memory. In *Proceedings of the Conference on Evolutionary Computation*, pages 136–141. IEEE, 1994.

[29] Andrew James Turner and Julian Francis Miller. Recurrent cartesian genetic programming. In *Proceedings of the International Conference on Parallel Problem Solving from Nature*, volume 8672 of *LNCS*, pages 476–486. Springer, 2014.