Facultad de Informática

Universidad de Alcalá

Incorporating transposons into Genetic Algorithms

Author: Samuel Santos Hernán

March 30, 2025

Table of contents

1 Problem Statement

This chapter presents the framework and the problem I aim to study, as well as stating the the main goals of such study.

1.1 Introduction

1.1.1 Context and Motivation

Genetic Algorithms (GAs) are optimization systems based on partial simulations of natural evolutionary mechanisms. They were proposed in the 60s by John Holland as a way to study natural adaptation, as well as to explore explore how these biological principles could be introduced in computational systems (Batista, Pérez, and Vega 2009). since their creation, GAs have been applied in a wide diversity of different fields (Shaikh 2021, Shevchenko 2024, Shahrabi 2024, Höschel 2018), due to their ability to explore large, complex solution spaces in an efficient way. These algorithms use a population-based search strategy, in which individuals —representing potential solutions— undergo selection, mutation, and crossover, mimicking biological evolution processes.

However, while GA are showing great potential when it comes to solving optimization problems, some important challenges are still implied, such as the tendency for the population to converge prematurely to suboptimal solutions, particularly in cases where genetic diversity diminishes too abruptly over successive generations. This limitation has sparked ongoing investigations, in the search of mechanisms capable of maintaining diversity as well as embracing exploration in the search space. In this thesis, I will explore the incorporation of transposons, mobile genetic elements, as a way to enhance GAs and address the challenge of premature convergence.

1.2 Proposed Solution: Incorporating Transposons

This section describes the key concept of trasposons, which is our many topic of study, and how they could be incorporated into GAs.

1.2.1 Biological Background of Transposons

Transposons, also known as "jumping genes", are sequences of non-encoding DNA that have the ability to travel to other parts of the genome and reinsert themselves there. This process, called transposition, is made possible by the enzyme transposase, which recognizes specific DNA sequences and permits the transposon to split and migrate. There are two kinds of transposons: DNA transposons, which move directly, and retrotransposons, which replicate themselves into RNA and then back into DNA before reinserting. In this study, we will focus on the former ones.

Transposons can create genetic diversity by moving, changing gene expression, or even disrupting gene normal function. These DNA sequences, which belong to the so called "junk DNA" category, are an important topic of study in genetics and medicine, even though they make only a minor portion of the human genome (Navarra Clinic 2023).

1.2.2 Integration of Transposons into Genetic Algorithms

One of the main concepts within GAs is the **genome**, an object —normally an array of elements but not exclusively— that contains all the essential information of an individual so that we can instantiate it. I must underline, that transposons could be incorporated to any kind of individual chromosome, but in this study I will only focus on Artificial Neural Networks (ANNs). In this regard, each gene will not only contain a value, but a list of values (and some other information described below), which will later constitute a certain neuron weights. This way we can create any ANN from its genome.

So far, all the elements of a genome have exclusively contained useful information, the so called **encoding genes** in biologic terms. But what if not all the genes were encoding? In this study I will research what would happen if these transposons could migrate and alter the genome of the individuals, allowing for more diversity. Specifically, this is the new proposed structure for the genome, in this case and as already stated, for ANNs:

- **Two arrays (would be the same for every problem):**
  - Array of G encoding genes
  - Array of T $<$ G transposons
- **Each encoding gene will contain (specifically using ANNs, but with easy adaptation to any other problem):**
  - **Values** of a neuron weights.
  - (boolean) Information about whether that weight is at the **beginning of a layer or not**.

  - (boolean) Information about whether that gene is **expressed** or not. **This is also a new proposal of this study**. It means that even though a certain gene may be present in a chromosome, it will not be reflected in the instantiated individual if that gene has this (newly defined) boolean field "expressed" set to false, but will still be able to transmit it to its offspring.

- **Each transposon will contain:**
  - **Gene index** that it is currently pointing at.

The role of the transposons will take place after the crossover phase:

1. First, each transposon will have a probability of migrating to another gene that is not already "occupied" by another transposon. This probability, as in real life, will depend on the surrounding environment and/or the population itself. In this study, I choose to let the probability rely on a **stagnation ratio** function, which will return a probability between 0 (population fitness mean still progressing) and 1 (population fitness mean completely stagnated). Nevertheless, other functions could be considered in future studies.
2. If this migration occurs, the transposon will randomly execute one (and only one) of the following actions on the current gene:
   - Modify its value(s) (in this study, it will modify one of its weights and setting it to a random value).
   - Toggle the expression of the gene (that is, toggle from True to False and viceversa the "expressed" field of the gene).
   - Create a new gene after the current one, initialized with random values (that is, instantiating a new gene with random value(s) or in this case weights, and placing it after the currently selected gene).
3. After the transposone phase ends, the traditional mutation phase would take place, potentially mutating an additional gene's value.

For example, imagine the following situation, where we have our list of genes, and a certain number of transposons associated to some of them (Figure 1.1):

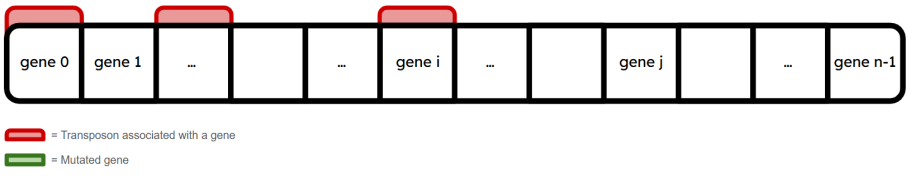

Figure 1.1: Transposons associated with genes in a genome.

As previously stated, each transposon does not contain any actual information of the individual, but if one of them jumps to another gene, it mutates it (Figure 1.2):

Figure 1.2: Same genome with a mutated gene by a transposon.

Note: Transposons will be inherited from an individual to its offspring, but a check should be executed to ensure that there are not more transposons than a certain maximum. This is important since we do not want the new chromosome to be built with too much randomness (we are still searching for convergence, at the end of the day).

1.3 The Snake Game Problem

1.3.1 Description of the Game Mechanics

The Snake Game (Angelos 2021) problem offers a simple yet sufficiently complex problem framework that will serve us as a test bench or target problem, and hopefully help to either demonstrate or refute the current hypothesis: do Transposons help in efficiency and convergence when applied to GAs?. The problem consists on a bi-dimensional board with a predefined width and height, in which a growing snake aims at eating as many apples as possible while avoiding the walls or its own body (Figure 1.3). There exist many variants of the game, with more or less restrictions, but for this study, the following conditions will be applied:

1. In each step, the snake will have to decide whether to keep forward or either turn left or right.
2. The snake will die if it either stumbles upon a wall or its own body.
3. The snake will gain **100 points** for each apple eaten, and the total sum will be divided by the **mean number of steps between eaten apples**. This will constitute the snake's final score and ultimately its fitness.

1.3.2 Challenges in Snake Game Optimization

This may look like a simple task, but finding a hard-coded, algorithmic strategy is actually non-trivial, bearing in mind the different obstacles that appear throughout a play, such

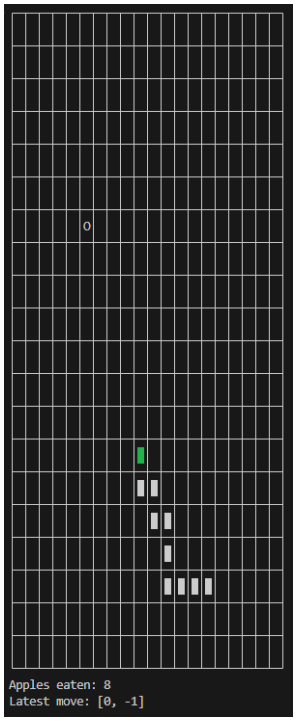

Figure 1.3: A certain state in the Snake Game.

as the increasing difficulty due to space reduction as the snake grows, and the need for the algorithm to adapt to dynamic environments. In order to address this, a neural network will be proposed, so that, for a given input (more details on this in what follows), the network takes one out of three possible actions (**LEFT**, **FORWARD**, **RIGHT**). Specifically, the network will be constituted by:

- An input layer with **11 input neurons**:
  - Length of the snake body
  - $[X, Y]$ distances to the middle point of the snake body.
  - $[X, Y]$ distances to the snake tail.
  - Distances to the closest obstacle in the three possible directions
  - $[X, Y]$ distances to apple.
  - Relative angle between the head of the snake and the apple (from -180º to +180º).
- A certain number of hidden layers, each with a certain number of neurons. These numbers will vary from an individual to another, mutated by **transposons** throughout generations).

- An output layer with **three output neurons**. This output will represent the snake's decision in a one-hot-encoded vector of length 3.

1.4 Research Objectives

I have considered one primary objective and three secondary objectives described as follows.

1.4.1 Primary Objective

The main goal of this research is to explore the potential benefits of applying transposons to genetic algorithms for the purpose of improving genetic diversity and preventing premature convergence in optimization issues, with specific application to an ANN built for the Snake Game problem.

1.4.2 Secondary Objectives

The secondary goals include the following:

- To implement a functional model of genetic algorithms with the integration of transposons and observe their effect on the Snake Game AI.
- To contrast the performance of normal genetic algorithms and transposon-enhanced algorithms based on fitness values and rates of convergence.
- To investigate the specific effects of transposon activities on the evolution and function of the population.

1.5 Conclusion

By exploring transposon effects on genetic algorithms, this study will show that such mobile genetic elements may substantially enhance diversity and sensitivity to evolution in the population. If this approach is successful, it can offer a new solution to the issue of premature convergence, which is a typical problem in most optimization contexts. Furthermore, the findings of this research could be used to advance research in incorporating other biological principles into computational models.

1.5.1 Summary of Key Points

- Transposons are mobile genetic elements that can be utilized to enhance the genetic diversity of populations in genetic algorithms.
- This project will examine the use of transposons in the genome representation of individuals in a Snake Game ANN, and their effects on the performance of the algorithm.
- The use of transposons is hoped to prevent premature convergence through the introduction of more varied evolutionary pathways.
- The goal of the project is to compare classical genetic algorithms with transposon-enhanced ones and compare their relative performance.

1.5.2 Next Steps

Moving forward, the next steps are to complete the implementation of the genetic algorithm with transposon integration and experiment on the Snake Game AI. Once the initial experiments are completed, a critical examination of the ensuing data will be undertaken in order to assess the effectiveness of transposons at preventing premature convergence. The outcome will then be compared with the performance metrics of conventional genetic algorithms and systematically recorded in the ensuing chapters of this dissertation.

2 Code Structure and Implementation

This chapter details the structure and implementation of the code developed for this thesis. The implementation was divided into various Python modules, each serving a specific role in the overall system, namely, genetic algorithms, snake behaviors, and simulation mechanics. Below, I outline the key components of the code.

2.1 Overview of the System Architecture

The system is designed to simulate multiple snake agents that evolve over time by optimizing their neural network structures through a combination of genetic algorithms and transposons. The goal is to maximize their performance in a Snake game environment by improving their ability to navigate and collect apples. Each snake's behavior is dictated by its neural network (its "brain"), and the evolutionary process aims at discovering more effective neural network architectures.

The code is organized following this hierarchy:

- **./Genes/**: Package where all components related to DNA, Transposons, and more, are defined.
  - `SnakeDNA.py`: This module contains the class `SnakeChromosome`, which contains all the essential DNA information to instantiate a snake.
  - `SnakeGene.py`: This module contains the class `SnakeGene`, which contains all the essential DNA information of what will constitute a Snake Neuron, such as the **list of weights** the neuron will have, or whether it shall be placed **at the beginning of a new layer**, to mention a few.
  - `Transposon.py`: This module contains the class `Transposon`, which, as stated in Chapter 1, is a non-encoding gene responsible for environment-driven mutations.
- **./Snake/**: Package where all components related to the Snake Game are defined.
  - `Snake.py`: Contains the `Snake` class, which simulates the snake's behavior and actions in the environment.
  - `SnakeGame.py`: Handles the execution of a game, given a certain Snake instance and a board size.

  - `SnakeFarm.py`: Handles the evolution of a population of snakes, evaluating their performance, and applying genetic operations like mutation and crossover.

- **./Utils/**: Package where some miscellaneous data is defined.
  - `Constants.py`: Contains all the basic **hyperparameters** to play with and other settings of the system.
  - `Utils.py`: Provides utility functions for operations such as random number generation, file I/O, and plotting.crossover.

2.1.1 SnakeGene Class

Attributes

- `index`: Position of the gene in the sequence of the chromosome.
- `weights`: A list of float values representing the weights associated with the gene.
- `newLayer`: Whether the gene is part of a new layer of the ANN.
- `isExpressed`: Whether the gene is actively contributing to the final instantiated individual or not.
- `transposon`: A reference to a `Transposon` associated with this gene, allowing dynamic modification.

Methods

- `__init__(self, index, weights, newLayer, expressed)`: Constructor that initializes a `SnakeGene` with its index, weights, layer assignment, and expression state.
- `mutate(self, mutateVal=False, newLayer=None, expressed=None)`: This method applies mutations to the gene. It can randomly change the gene's weights, its layer, or whether it is expressed.

2.1.2 Transposon Class

Attributes

- `gene`: This stores a reference to an instance of `SnakeGene`, which represents a gene associated with this transposon.

```python
class SnakeGene:
    transposon: Transposon
    def __init__(self, index: int, weights: List[float]=None, newLayer: bool=None, expressed:bool=True):
        self.index   = index
        self.weights = weights

        if newLayer is not None: self.newLayer = newLayer
        else: self.newLayer = index == 0 or random.random() < Consts.PROB_NEW_LAYER

        self.isExpressed = expressed
        self.transposon  = None

    def mutate(self, mutateVal=False, newLayer=None, expressed=None):
        if mutateVal: self.weights[random.randint(0,len(self.weights)-1)] = Utils.random_unbounded_float()
        if newLayer is not None:  self.newLayer    = newLayer
        if expressed is not None: self.isExpressed = expressed
```

Figure 2.1: Class that implements the Snake Gene object.

Methods

- `__init__(self, gene=None)`: Constructor that initializes the `Transposon` instance. By default, the `gene` is set to `None`.
- `migrate(self, newGene, SnakeChromosome, mutate=True)`: Assigns a new `SnakeGene` to this transposon. If a valid `newGene` is provided, it will replace the current one. Additionally, if `mutate=True`, the transposon can mutate the gene associated with it.
- `mutateWeight(self, chromosome)`: Modifies the weight(s) of the gene's parameters.
- `toggleExpression(self, chromosome)`: Toggles whether the gene is expressed (i.e., active) by calling the `mutate` method on the gene.
- `mutateLayer(self, chromosome)`: Alters the layer the gene belongs to.
- `createNewGene(self, chromosome)`: Adds a new gene to the chromosome and generates new weights for it.
- `destroyGene(self, chromosome)`: Removes a gene from the chromosome, reducing its total number of genes. This is currently not used but prepared for future research.
- `mutateGene(self, chromosome)`: Randomly selects one mutation function from a predefined list (`mutateWeight`, `toggleExpression`, `mutateLayer`, or `createNewGene`) and applies it to the chromosome.

```python
class Transposon:
    def __init__(self, gene: SnakeGene = None):
        self.gene = None
        self.migrate(gene,mutate=False)

    def migrate(self, newGene: SnakeGene, chromosome: SnakeChromosome=None, mutate=True):
        if self.gene is not None: self.gene.transposon = None

        self.gene = newGene
        if self.gene is not None:
            self.gene.transposon = self
            if mutate: self.mutateGene(chromosome)

    def mutateWeight(self,chromosome: SnakeChromosome):
        self.gene.mutate(True)
    def toggleExpression(self,chromosome: SnakeChromosome):
        self.gene.mutate(expressed = not self.gene.isExpressed)
    def mutateLayer(self,chromosome: SnakeChromosome):
        self.gene.mutate(newLayer = not self.gene.newLayer)

    def createNewGene(self,chromosome: SnakeChromosome):
        if self.gene is not None:
            nextGenes = chromosome.genes[self.gene.index+1:]
            for gene in nextGenes: gene.index += 1

            w = [Utils.random_unbounded_float() for _ in range(len(self.gene.weights))]
            chromosome.genes = chromosome.genes[:self.gene.index+1] + [SnakeGene(self.gene.index+1, weights=w)] + nextGenes

    def destroyGene(self,chromosome: SnakeChromosome):
        nextGenes = chromosome.genes[self.gene.index+1:]
        for gene in nextGenes: gene.index -= 1
        chromosome.genes = chromosome.genes[:self.gene.index] + nextGenes
        chromosome.transposons.remove(self)

    def mutateGene(self, chromosome):
        actionsList = [
            self.mutateWeight,
            self.toggleExpression,
            self.mutateLayer,
            self.createNewGene,
            #self.destroyGene
        ]
        actionsList[random.randint(0,len(actionsList)-1)](chromosome)
```

Figure 2.2: Class that implements the Transposon.

2.1.3 SnakeChromosome Class

Attributes

- `genes`: A list of `SnakeGene` objects that represent the genetic sequence of the chromosome.
- `transposons`: A list of `Transposon` objects. These transposons dynamically interact with the genes to introduce mutations and structural modifications.

Methods

- `__init__(self, brain=None)`: This constructor initializes the genes and transposons. If a neural network brain is provided, it initializes genes with weights. Otherwise, it sets up the genes without weights.
- `getLayerSizes(self, chromosome)`: Computes the number of neurons (genes) in each layer of the chromosome.

- `migrateTransposon(self, transposonIdx)`: Selects a transposon and migrates it to a different gene, applying mutations as well.

```
class SnakeChromosome:
    genes: List[SnakeGene]
    transposons: List[Transposon]
    def __init__(self, brain=None):
        if brain is not None:
            self.genes       = []
            self.transposons = []
            idx = 0
            for layer in brain:
                self.genes += [SnakeGene(idx+i,weights,i==0) for i,weights in enumerate(layer)]
                idx += len(layer)
        else:
            #Init genes without assigning weights:
            self.genes = [SnakeGene(i) for i in range(Consts.SNAKE_INPUTS + Consts.INITIAL_NEURONS + Consts.SNAKE_OUTPUTS)]

            #Making sure the first layer has the right number of neurons:
            for i in range(Consts.SNAKE_INPUTS+1):
                self.genes[i].newLayer = (i % Consts.SNAKE_INPUTS) == 0
                self.genes[i].isExpressed = True

            #Computing the size of the different layers:
            layerSizes = SnakeChromosome.getLayerSizes(self.genes)

            #Assigning weights to all the neurons, depending on the layer they are in:
            index = 1
            for gene in self.genes:
                if gene.newLayer:
                    nextLayerSize = layerSizes[index]
                    index += 1
                gene.weights = [Utils.random_unbounded_float() for i in range(nextLayerSize)]
        self.transposons = [Transposon() for _ in range(Consts.INITIAL_TRANSPOSONS)]
        for i in range(Consts.INITIAL_TRANSPOSONS): self.migrateTransposon(i)

    def getLayerSizes(chromosome):
        layerSizes = []
        curLayer = 0
        for gene in chromosome:
            if curLayer > 0 and gene.newLayer:
                layerSizes.append(curLayer)
                curLayer = 0
            curLayer += 1
        layerSizes += [curLayer,Consts.SNAKE_OUTPUTS]
        return layerSizes

    def __str__(self):
        return f"<SnakeChromosome\n   Genes: {[[g.value,g.newLayer,g.isExpressed] for g in self.genes]}\n>"

    def migrateTransposon(self,transposonIdx):
        gene = None
        while gene is None or gene.transposon is not None: gene = self.genes[random.randint(0,max(len(self.genes)-4,0))]
        self.transposons[transposonIdx].migrate(gene,mutate=False)
```

Figure 2.3: Class that implements the Snake DNA.

2.1.4 Snake Class

The `Snake` class represents a snake agent, including its movement, brain, and fitness evaluation. The class has the following fields and methods:

Attributes

- `DNA`: This field stores the genetic material of the snake, represented by a `SnakeChromosome` object. This allows the snake to inherit its data to its offspring by encoding its

neural network.

- `brain`: This field is a 3D list of neural network weights (array of layers, each layer is an array of neurons, and each neuron is an array of weights). It is derived from the snake's DNA and is used to predict the snake's next move.
- `body`: A list representing the coordinates of the snake's body parts. The first element is the head, followed by the rest of the body.
- `tupleBody`: A set containing tuples of the body parts' coordinates, used for faster lookup.
- `lastMove`: Stores the last movement direction of the snake using values from the `Consts.Move.MOVES` array.
- `fitness`: An integer that represents the snake's performance (with higher fitness indicating a better performance).
- `applesEaten`: A counter to track how many apples the snake has eaten.

Methods

- `__init__(self, brain, DNA)`: The constructor initializes the snake's brain, body, and other parameters. If no `DNA` is provided, a new `SnakeChromosome` is created.
- `reset(self,boardWidth,boardHeight)`: This method resets the snake's body on the game board by randomly assigning a new position and direction.
- `move(self,input)`: Handles the movement logic of the snake. It takes input, predicts the next move using its brain, and updates its body accordingly.
- `addBodyPart(self,bodyPart)`: Appends a new body part to the snake, increasing its length, and updates `tupleBody`.
- `predict(self,input)`: This method performs a forward pass through the neural network (defined in `brain`) to determine the next move. Then, it uses a softmax function to convert the neural network output into a one-hot encoded vector representing the direction.
- `getPhenotype(self)`: Converts the genetic representation (`DNA`) into the `brain`, which is the snake's phenotype.

2.1.5 SnakeFarm Class

The `SnakeFarm` class manages a population of snakes. It handles the evaluation, selection, crossover, and reproduction of snakes, following the principles of genetic algorithms. The main goal of the `SnakeFarm` is to evolve snakes over generations to maximize their fitness.

Attributes

- `snakes`: A list of `Snake` objects that represent the current population of snakes.
- `game`: An instance of the `SnakeGame` class, which simulates the environment in which the snakes play.
- `meanScores`: A list that stores the average fitness scores for each generation.
- `bestScores`: A list that keeps track of the best fitness scores in each generation.
- `prevBestSnake` and `currBestSnake`: These fields store the best-performing snake from the previous and current generations, respectively in order to perform elite substitution.

Methods

- `__init__(self, snakes)`: Initializes the snake population and sets up the game environment. If no initial population is provided, it creates a default population of snakes.
- `saveCurSnakes(self, curGen, bestSnakeName)`: This method saves the current population of snakes to files, particularly after every 50 generations or for the best snake in each generation.
- `evaluateSnakes(self, g, display, displayGeneration)`: Evaluates the fitness of all snakes in the population. It runs each snake through multiple games and calculates an average fitness score. The method also updates the current best snake.
- `rouletteWheel(self)`: Implements the roulette wheel selection method, where snakes are selected for reproduction with a probability proportional to their fitness.
- `crossover`: Handles the crossover process by selecting pairs of parents and generating offspring. It uses the `rouletteWheel` method to select the parents.
- `generateOffspring(self, firstHalf, secondHalf)`: This method is responsible for generating new offspring by combining the genetic material of two parent snakes.
- `recombination(self, parent1, parent2)`: Combines the genetic information of two parent snakes to produce two offspring. It selects a random crossover point (based on the minimum number of layers in the parents' brains) and, if a random probability check (against PROB_CROSSOVER) passes, generates offspring by exchanging parts of the parents' neural network layers. Otherwise, it returns the original parents.

- `getStagnationRatio(self, x, y)`: Calculates a stagnation index for a subsection of the meanScores list defined by indices x to y. It computes the standard deviation (noise) and the slope of the mean scores trend, and then uses a normalized tanh function to return a value between 0 and 1—where 0 indicates active evolution and 1 indicates stagnation.
- `transpMutProb(self, firstTranspGen, minPeriod)`: Determines the probability of applying a transposon mutation based on the stagnation of the population. It checks if enough generations have passed (firstTranspGen), computes the stagnation ratio for a recent segment of meanScores, and if the ratio is above a threshold, scales it by some ratio lesser than 1. Otherwise, it returns 0, indicating no transposon mutation should occur.
- `transposonMutation(self)`: Applies transposon-based mutations to each snake in the population if transposons are enabled. For each snake, it iterates over its genes and, based on the probability returned by `transpMutProb`, migrates a transposon to a gene (ensuring the gene is not already associated with a transposon). It also resets the `newLayer` and `isExpressed` flags for the first few input genes.
- `mutation(self)`: Iterates through each snake and every neuron in each layer of the snake's brain. With a probability defined by PROB_MUTATION, it mutates a neuron's weight by assigning it a new random unbounded float value.
- `runGeneration(self, generation, display, displayGeneration, bestSnakeName)`: Executes one full generation cycle of the genetic algorithm. It sequentially evaluates the snakes, saves the current population, displays generation data, performs crossover, applies transposon mutations, and finally applies standard mutation. It updates the previous best snake record with a deep copy of the current best snake at the end of the generation.

2.1.6 SnakeGame Class

The `SnakeGame` class handles the execution of a game, given a certain snake. It is used to evalute the performance of the snakes population by executing several games for each snake.

Attributes

- `width`: The width of the game board, which is defined by the constant `Consts.BOARD_WIDTH`.
- `height`: The height of the game board, defined by `Consts.BOARD_HEIGHT`.
- `apple`: A list representing the current coordinates $[x, y]$ of the apple on the board.
- `snake`: An instance of the `Snake` class, representing the snake in the game.

- `lastMove`: A string that stores the last move made by the snake (initially set to "NONE").
- `score`: The current score of the game, initialized to 0 when the game is reset.
- `meanStepsPerFeed`: A list to store the average number of steps taken by the snake to reach the apple after each feed.
- `stepsSinceLastFeed`: An integer to track the number of steps since the snake last ate an apple.

Methods

- `reset(self, snake)`: Resets the game state by reinitializing the snake (either to the provided one or a new one) and placing a new apple. Also resets the score and tracking attributes like `meanStepsPerFeed`.
- `placeNewApple(self)`: Places a new apple on the board at a random location, ensuring it does not overlap with the snake's body or its head.
- `getAStarDistance(self, snake, goal)`: Computes the distance from the snake's head to the goal (e.g., the apple) using the A* algorithm. It returns the shortest path length in terms of the number of moves or "infinite" if no valid path exists. Currently not used, but considered for the distance to apple.
- `getBodyPartsDistances(self, numParts)`: Calculates the relative distances between the snake's head and certain body parts. These distances are based on dividing the body into segments. It returns a list of distances or an empty list if the number of parts is less than one.
- `getInput(self)`: Generates the input for the game based on the snake's current state. It calculates the distances to obstacles (such as the board edges or the snake's body) in various directions, and also computes the distance and angle to the apple.
- `getAngleToApple(self)`: Computes the angle between the snake's head and the apple, and adjusts the angle relative to the snake's last movement direction. The angle is returned in degrees, normalized between -180 and +180 degrees.
- `checkGameOver(self)`: Checks whether the game is over based on whether the snake has collided with the edges of the board, with its own body, or if too many steps have passed since the last apple was eaten.
- `updateBoard(self, oldTail, curGen)`: Updates the visual representation of the board. It places the snake, apple, and relevant game information on the console and updates the game display with the latest state.
- `placeCharacter(self, char, x, y)`: Places a character (like the snake's body, the apple, or empty space) at the given x and y coordinates on the console display.

- `checkAppleEaten(self, newTail)`: Checks whether the snake has eaten the apple by comparing the position of the snake's head with the apple. If eaten, a new apple is placed and a new body part is added to the snake.

- `drawEmptyBoard(self)`: Draws the initial empty game board with walls using special characters for borders and spaces. It's primarily used at the start of a new game.

- `start(self, display, curGen, forcePlay)`: This is the main function that starts the game. It handles the game loop, including resetting the game, displaying the board, and checking game-over conditions. It updates the snake's fitness score at the end, based on the number of apples eaten and the number of steps between each apple.

2.1.7 Constants and Utils

`Constants.py` and `Utils.py` are files that define essential parameters for the game board, snake behavior, and evolutionary settings, as well as utility functions. These constants include board dimensions, points awarded per apple, the size of the neural network's input and output layers, and probabilities for genetic operations like crossover and mutation. These parameters provide flexibility, allowing the system to be tuned for different experimental setups.

2.2 Evaluating Transposons against traditional GAs

In order to demonstrate how good, bad (or useless) Transposons are, a series of different experiments will be executed. The flow of all of them will be the same. Each experiment will consist on running the complete algorithm a decent number of times, so as to get a sufficiently confident statistic measure. This is because running the algorithm just once is not enough to prove that a certain configuration is good or bad. In this case, I choose 20 as the number of executions per configuration. That is, each experiment will take 20 times the number of generations I let the population evolve (in this case, more than 400), to complete; otherwise, the size of the population, the selection function, the (traditional) mutation probability and needless to say the fitness function, will remain the same throughout all the experiments.

For each experiment, two graphs will be obtained: one depicting the mean scores of each execution, and the other one depicting the evolution of the best scores of each iteration.

2.2.1 Traditional GA, with no transposon in use

In this configuration, I used a traditional configuration, with a population of 100 snakes, a roulette wheel selection function, a 1% chance of having a traditional mutation and running each iteration for 400 generations; as already stated, the following experiments will also share this parameters.

The shape obtained in the mean scores (fig. 2.4) graph turns out to be as expected, with a logarithmic drawing, save for an outlier that got a score way above the rest. Curiously, it was the very first iteration the one that got this anomalous score. The average of the best scores with this approach ends up being 2189.65.

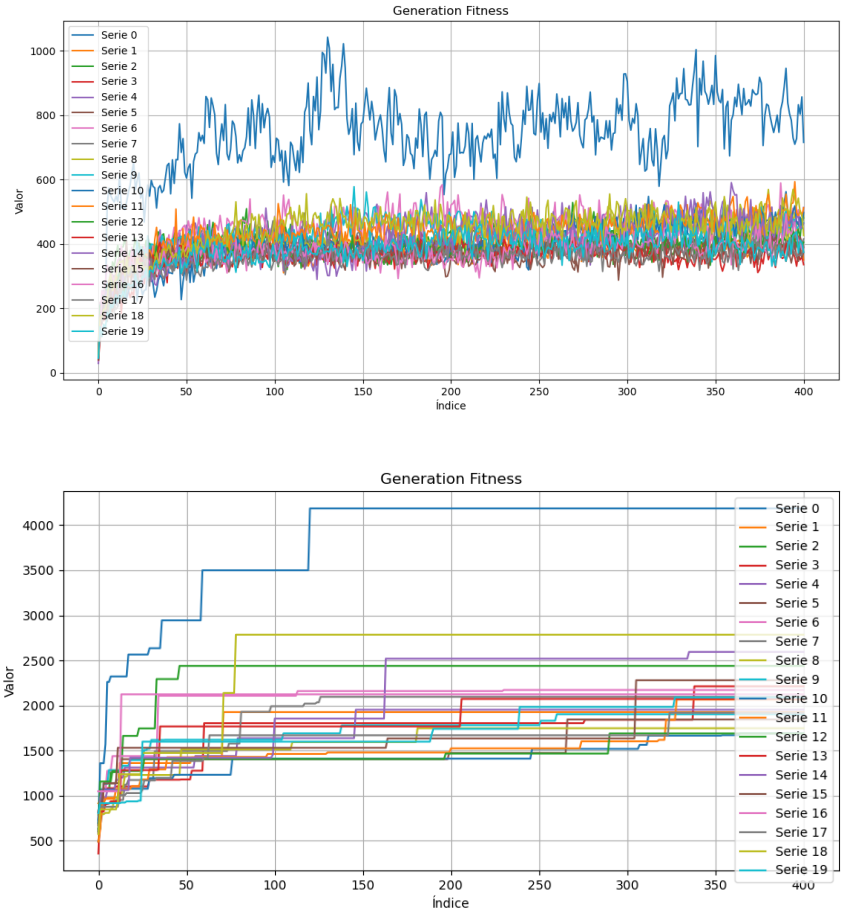

Figure 2.4: Evolution of the mean scores and best scores per generation and execution in traditional GA.

2.2.2 GAs with transposons, and a max. probability of 20%

In this configuration, apart from enabling transposons, I set a maximum probability of transposon mutation as 20%. This might seem like way too much for a mutation probability, but since at the end of the day we are trying to allow the population to escape from the current local minimum and explore further the space of solutions, we want to give an important "push", so to say, to the snakes so that it can overcome a local minimum "valley" and go downhill through another one. The expected behavior of the mean score graphs should be that of a staircase, with steps that may be softened with generations.

The shape obtained in the mean scores graph (fig. 2.5), nevertheless, is still logarithmic. The plot twist is that we are no longer talking about a single outlier but many of the snakes seem to bee above the mean. And not only that, but as per can be seen in the best scores graph, only three of the executions (in contrast with eight executions in the traditional approach) end up scoring below 2000 points. This leaves this approach with a final mean of the best scores of 2615.55. This is promising, but we can do better.

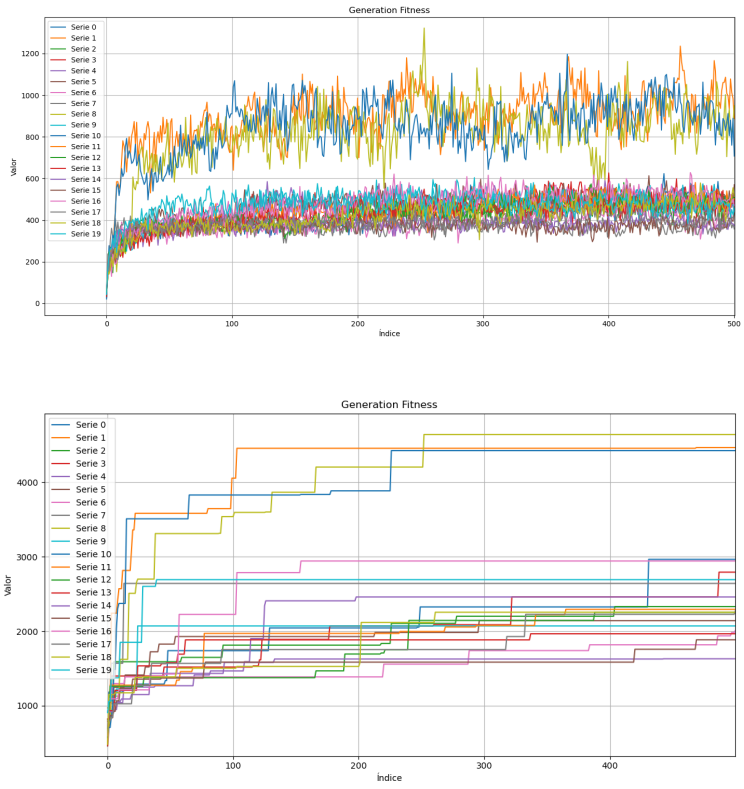

Figure 2.5: Evolution of the mean scores and best scores per generation and execution in GA with transposons.

2.2.3 GAs with transposons, and a max. probability of 40%

In this final configuration, I tried increasing the max. probability of a transposon mutation even more, up to 40%. To my surprise, even more snakes started to score above the mean (fig. 2.6), and regarding the best scores, once again only three snakes ended up scoring below 2000 points, an even higher best score (4929 points) and an even higher mean of the best scores (2987.45 points). Moreover, if we take a look at the best scores graph, we can observe that many of the executions (around 6, specifically) end up being above 4000 points.

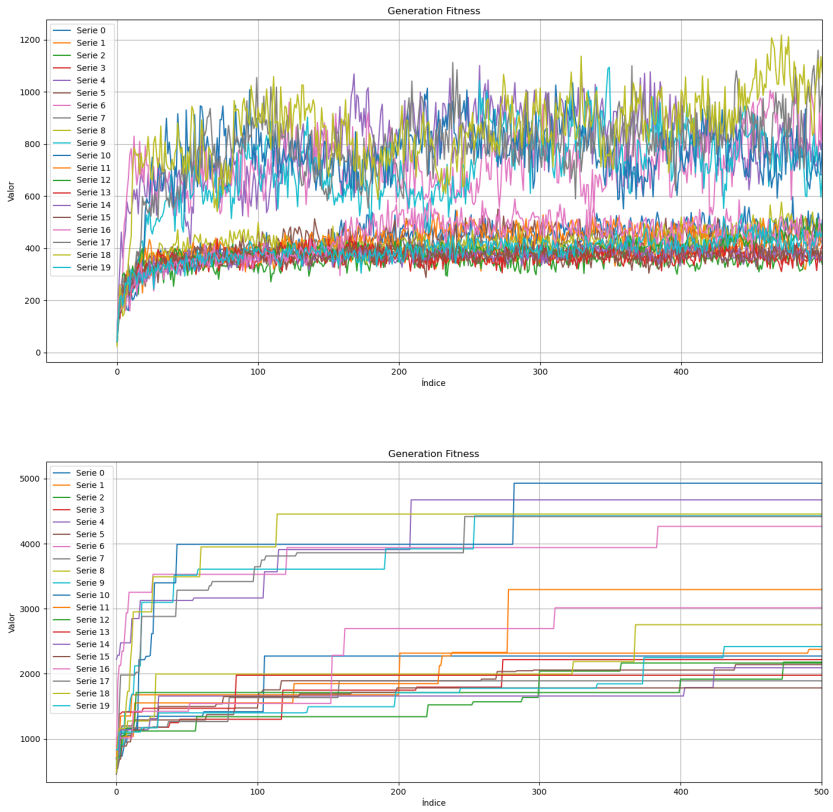

Figure 2.6: Evolution of the mean scores per generation and execution in GA with transposons.

3 Conclusions and Future Work

In this final chapter I summarize the entire study and propose some future work that could be done afterwards.

3.1 Conclusions

In the current thesis, we have proposed the application of transposons with Genetic Algorithms (GAs) to counter the problem of premature convergence, particularly in multi-objective optimization problems like the Snake Game. The experiments we have performed indicate that transposons can significantly enhance the performance of GAs by maintaining genetic diversity through generations and beyond. Specifically, the inclusion of transposons allowed the algorithm to explore over a broader range of possible solutions, reducing the risk of converging too early on inferior solutions.

The experiments conducted also had a point of comparison between the standard GAs and those incorporating transposons with varying probabilities of occurrence (20% and 40%). It was observed that in either case, the transposon-enhanced GAs outperformed standard GAs in terms of solution quality and overall stability. Specifically, the results showed that GAs with transposons had increased fitness values and greater adaptability in solving the Snake Game, a problem identified by its dynamic and changing environment.

The moral here is that the use of transposons introduces a level of genetic diversity that circumvents the pitfalls of premature convergence, usually the byproduct of an exponential decline in genetic diversity. By enabling targeted non-encoding genes to "jump" to a new location somewhere within a chromosome, transposons produce diversity without introducing unwanted amounts of randomness, and they balance exploitation with exploration.

3.2 Future Work

While the outcomes from the above approach are promising, there are several avenues of research that could refine and optimize the application of transposons to GAs even further:

3.2.1 Additional features to implement in the code

For starters, some of the following capabilities weren't possible to put into place due to the lack of time, but could be valuable to research in the future, such as utilizing A* algorithm for computing the distance instead of Manhattan, polishing the fitness function so the snake is taught how to move around obstacles (the snake is perfect in learning how to reach the apple efficiently, but isn't quite able to manage avoiding its own body), or introducing a function that displays graphically how does a specific snake ANN look like.

3.2.2 Testing on More Complex Problems

Although the Snake Game provides a suitable test bed, it is rather trivial compared to real optimization problems. Future research could extend transposon-augmented GAs to more sophisticated domains such as multi-objective optimization, dynamic environments, or real-time systems. Such applications would stress the algorithm to learn and find optimal solutions in more difficult scenarios, once again demonstrating the robustness of the methodology.

3.2.3 Exploring Other Types of Junk DNA

This thesis focused solely on the introduction of DNA transposons. Future work would be to discover how to utilize other types of "junk DNA", since there exist a big diversity of non-encoding genes. The implications of these genetic elements for GAs performance remain to be fully understood and could provide insights into increasing genetic diversity in evolutionary algorithms.

3.2.4 Final words

In conclusion, while the current research has put the potential for transposon use to improve GAs on the map, there is a great deal of room for research and experimentation. Exploring the proposed areas of future research will continue to refine this technique and push its application to more difficult and dynamic optimization problems.

Bibliography

Angelos, Ayla (2021). "The history of Snake: How the Nokia game defined a new era for the mobile industry". In: URL: https://www.itsnicethat.com/features/taneli-armanto-the-history-of-snake-design-legacies-230221.

Batista, Belén Melián, José A. Moreno Pérez, and J. Marcos Moreno Vega (2009). "Algoritmos Genéticos. Una visión práctica". In: *NÚMEROS* 71, pp. 29–47. URL: https://funes.uniandes.edu.co/wp-content/uploads/tainacan-items/32454/1209011/Meli25C325A1n2009AlgoritmosNumeros71.pdf.

Höschel, Kaspar (2018). "Genetic algorithms for lens design: a review". In: URL: https://link.springer.com/article/10.1007/s12596-018-0497-3.

Navarra Clinic, University of (2023). *MEDICAL DICTIONARY: Transposon*. URL: https://www.cun.es/diccionario-medico/terminos/transposon.

Shahrabi, Shahriar (2024). *Procedural Paintings with Genetic Evolution Algorithm*. URL: https://shahriyarshahrabi.medium.com/procedural-paintings-with-genetic-evolution-algorithm-6838a6e64703.

Shaikh, Taimur (2021). *How I made a genetic algorithm that generates music*. URL: https://medium.com/dc-csr/how-i-made-a-genetic-algorithm-that-generates-music-67b90cd0d05e.

Shevchenko, Eugenii (2024). *How to solve games with Genetic Algorithms (Building an Advanced Neuroevolution solution)*. URL: https://medium.com/@eugenesh4work/how-to-solve-games-with-genetic-algorithms-building-an-advanced-neuroevolution-solution-71c1817e0bf2.