Evolutionary_Algorithms

December 22, 2020

Irakli Okruashvili, Marjan Chowdhury, Andrew Pantera

Professor Strigul

1 General: Evolutionary Algorithms

1.1 Introduction

Optimization problems are being increasingly solved with artificial intelligence. Using such methods we are today able to solve problems with ease that were impossible to solve decades ago. Evolutionary algorithms are no different; they provide a metaheuristic solution to optimization problems through biomimicry. Just as humans have been shaped and built by evolution to conquer and survive our landscape, evolutionary algorithms attempt to build solutions to modern problems with the tried and true technique of evolution. Evolutionary algorithms model many methods after successful attributes of evolution such as selection, genetic crossover, genetic mutation, and survival of the fittest. Evolutionary algorithms and genetic algorithms in general are among a larger set of AI algorithms that leverage biomimicry including neural networks, cellular automata, swarm intelligence, and reinforcement learning. These are indispensable in solving some of the most complex problems we face today. The study of genetic algorithms is a relatively new one, being first implemented in and pioneered in the 1960s. Today evolutionary algorithms solve problems from circuit design to stock market predictors and even an antenna that is currently aboard a NASA spacecraft.

1.2 History

We owe many of the strongholds of our current technological foundation to advancements made in World War Two. Evolutionary algorithms are not one of these advancements. They belong to a rarer set of advancements pioneered after WWII, in the case of evolutionary algorithms in the 1960s. Scientists and researchers started looking to the products of nature for inspiration for technological advancement, producing fruits such as velcro and airplanes. Later, however, they began to look to the processes of nature itself, rather than its products, for this inspiration.

Nils Barricelli began research into the field in the 1950s and was the first person to simulate reproduction and evolution on a computer in the 1960s. Alex Fraser began work in the field in the 1950s and contributed major innovation to the field with his landmark work "Simulation of genetic systems by automatic digital computers" published in 1958. Lawrence Fogel contributed to the field through applying evolutionary programming to real world problems in multiple industries from 1965 to 2007. Ingo Rechenberg developed the first widely recognized successful application of evolutionary algorithms to solve engineering problems in the 1970s. Rechenberg helped validate evolutionary algorithms such as viable and powerful optimization methods. Popularity of the method grew further through the works of John Holland such as his pioneering book "Adaptation in Natural and Artificial Systems". Overall, the early development of evolutionary algorithms led to the establishment of three major paradigms: genetic algorithms (GA) which is associated with genetic programming (GP) and classifier systems (CS), evolutionary strategies (ES), and evolutionary programming (EP).

Genetic algorithms were first developed by John Holland and his collaborators in the 1960s and 1970s. Determined to understand the principles of adaptive systems, Holland believed biological evolution would be an effective model to observe such principles including adapting to changing environments. Throughout these times Holland and his colleagues would simulate biological evolution through a representation of a population and environment in a computer program. The program would use population genetics operations such as crossover, mutation, inversion, and selection to create a new population generation that is more adapted to the environment. Consequently, this demonstration showed that GAs follow Darwinism and natural selection as a form of optimization. Throughout the process of creating new generations from the genetic operations, an optimal population would eventually be produced that would be ideal for the set environment. These concepts developed by Holland would become more popular over time, as multiple scientists would develop theses and projects utilizing such concepts to contribute to the field's growth and its applicability in different industries. Thus, GAs came to be one of the, if not the most well known form of the evolutionary algorithm.

Evolutionary programming was founded by Lawrence J Fogel in the 1960s. Fogel was tasked to research AI while serving at the National Science Foundation (NSF). An important aspect of an AI is that it must be capable of becoming more intelligent, and Fogel believed evolution modeled this concept of increasing intellect. Fogel would simulate this process through a population represented by finite state machines/automata (FSA) whose objectives were to identify its environment and to ideally respond and complete a given goal. The environment and goal would be in the form of input symbols for the FSAs. The output of the FSAs would then be evaluated through a function determining the fitness of each FSA. The more fit FSAs would be kept as parents that would produce the next generation of FSAs who were altered versions of the parents. These alterations would be the mutations the offspring FSA would possess, and the new population would repeat the reproduction process until an optimal solution can be produced. The results of Fogel's research would lead to EP being applied to experiments involving automata, optimization algorithms, and neural networks. Through conferences and projects in the early 1990s, EP has since been utilized in various fields such as medicine, defence, and mathematics.

Evolutionary strategies were first established by Ingo Rechenberg, Hans-Paul Schwefel, and Peter Bienert in the late 1960s. As students in university, the three wanted to develop a robot and wind tunnel where the robot would minimize its force of drag in the tunnel. Instead of creating a new optimization process or model for the robot and wind tunnel, the three researchers constructed methodologies that can be used to assist in finding an optimal solution to the problem. After a couple of failed attempts of finding an ideal setup, Rechenberg then decided to utilize the concept of random decisions to create the first known ES. This strategy would involve randomly changing the qualities of the wind tunnel's structure and this process would be repeated until the wind tunnel had an optimal structure. These random changes would serve as the mutations that occur in many evolutionary algorithms. Rechenberg and Schwefel would later improve upon this ES in the early 1970s. This became the foundation of new ESs the two would create that would be more useful. These ES would simulate evolution utilizing parent generations and offspring generations with mutations that could or could not be contributed by the parent. As time passed, researchers would develop new ES and discover new applications of such strategies that some evolutionary algorithms can follow.

1.3 Derivation

Evolutionary algorithms are derived from biological concepts and are very applicable in biology activities as well as for engineering problems. They are optimization algorithms that model evolution of a population (such as a set of data) composed of species to achieve an ideal solution to a given situation. Darwinism and natural selection are the foundation of these algorithms. A population would usually be given an environment or requirements where some species will have traits that satisfy the constraints better than others. A function can be used to gauge the fitness of the species by providing a rating and the species with the highest rating will most likely produce the next generation. This process of survival of the fittest contributes to the creation of a new generation through the genetic operations mentioned in genetic algorithms or other forms of mutations like the ones implemented in evolutionary programming. The process is then repeated until the ideal population is formed. As a result, each species' genetics play a major role in who is selected to create the new generation and its genetics. Because of natural selection, there is competition between species to survive and reproduce and the species with more optimal genetics will be more likely to survive. Those genes and mutations are then passed to the next generation who would better adapt and satisfy the constraints, much like how evolution and natural selection work in the real world.

The three paradigms which are genetic algorithms, evolutionary programming, and evolutionary strategies, are fundamental in the derivation of the structure of Evolutionary Algorithms. Genetic algorithms and the genetic operations crossover, mutation, inversion, and selection place emphasis of the most optimal genes. An evolutionary algorithm that is structured so that it prioritizes on selecting the best genes and qualities most likely uses genetic algorithms as its framework. The algorithm's results will reflect a representation of the solution to the given problem. On the other hand, Evolutionary programming does not use the crossover operation, so the passing of traits and mutations are more random and have less constraint. Evolutionary programming still follows survival of the fittest so the more optimal species will most likely be chosen to produce the next generation. An evolutionary algorithm having more emphasis on the link between the parent's and offspring's qualities and less emphasis on finding the best genes to carry on to the next generation is most likely based on evolutionary programming. The algorithm would provide a representation that is closer to the given problem instead of the solution. Evolutionary strategies use a criteria, usually based on functions, to determine whether certain populations and species, qualities, and genetics will be included or excluded in the experiment or program. As a result, the use of certain evolutionary strategies can create variations or affect the implementation of genetic algorithms and evolutionary programming as they can impact the result of parent and offspring populations. They can also be used standalone in other applications as well. Overall, these three paradigms allow evolutionary algorithms to be incredibly diverse in structure and implementation.

1.4 Applications

Evolutionary algorithms solve optimization problems. These methods thrive in areas where humans with conventional algorithmic approaches fail, often with stringent, unusual, or conflicting design constraints. In areas where conventional algorithms work, however, evolutionary algorithms are redundant. Evolutionary algorithms approximate solutions, and are extremely time and resource intensive. If a problem can be optimized conventionally, applying evolutionary algorithms in that instance will likely take more time and resources, and produce a worse result. Because of this many applications of evolutionary algorithms produce a result better than any human could. One famous example of this is the “evolved antenna”. In 2006 NASA launched a satellite with an X-band antenna designed by an evolutionary algorithm. The conditions were such that evolutionary algorithms thrived, there were strict design constraints concerning radiation patterns, and the result was an antenna better than any engineer could create. Since this famous example, evolutionary algorithms have been the goto method of designing radio antennas around complex design restrictions.

Evolutionary algorithms are applied in similar places as other artificial intelligence methods. They were applied to stock market prediction in the early 2000s with little success, but they are seeing more success right now, with multiple successful evolutionary algorithm predictors for sale in 2020. It seems every form of AI has been thrown at stock prediction, however. Evolutionary algorithms are not a great fit for stock prediction; they are used carefully as a tool. One field that has found evolutionary algorithms to be a good fit is integrated circuit design. There are often many complex constraints with complex interactions; a perfect use case for evolutionary algorithms. There is no large scale commercial implementation of evolutionary algorithms in integrated circuit design, but major research and breakthroughs in the field are being presented in papers published as late as September 2019 and the method has succeeded in a lab setting. Evolutionary algorithms are being researched for use in civil engineering, energy fuels, operations research management science, automation control systems, and other engineering, science, and technology fields. They have been demonstrated to significantly improve the power output of solar arrays when applied to design preparation. They reduced energy consumption by 13% when used to optimize energy consumption of a natural gas purification process.

Outside of the lab, evolutionary algorithms are being used to solve real world problems. They are being sold as market prediction tools as previously described, as well as representing rational agents in economic models such as the cobweb model. An evolutionary algorithm has been used to significantly improve the accuracy of a back propagation neural network used to successfully predict terrorist attacks. Evolutionary algorithms have been used in the field of natural language processing to accomplish goals such as grammar induction and word sense disambiguation. Applications exist of evolutionary algorithms in a litany of engineering fields. They are used to solve problems in CAD, quality control, sorting, timetabling, vehicle routing, creating floor layouts, quality control, communications infrastructure, container load optimization, revenue management, computer vision, and other optimization applications.

2 Original Research Component

2.1 Introduction

We focused our research on implementing evolutionary algorithms to solve real problems. We first created a general model for evolutionnary algorithms, then applied that model to solve an optimization problem, and finally we applied that model to solve a maze.

2.2 The Computational Model (the code)

We chose to implement our evolutionary algorithm computational model as an abstract class in Python. The class defines paramaters such as total population size, the number of generations, the proportion of the population that is killed in each generation, and thje proportion of the genomes that will replace the killed members of the population that are mutated, as opposed to being the result of the breeding of two parent genomes. The class can also be seeded with a given innitial gene pool, instead of generating the gene pool randomly. This could be useful for continuing the evolution of a previously computed gene pool, or to seed the gene pool with genomes that are already known to be effective. The class also defines methods to seed the gene pool with random genes, to perform the evolution within the given paramaters, and to display the result of the evolution. Finally, the class has abstract methods. These abstract methods need to be overridden by a subclass in order for an object to be instantiated. This informs the following behavior: in order to use this class to perform evolution, the class must be subclassed by a given implementation. The methods that need to be overridden, defining the functionalities specific to that implementation are measuring the fitness of a given genome, generating a random genome, and reproducing two parent genomes into an offspring genome.

[1]:
```
from abc import ABC, abstractmethod
import random
import pandas
import matplotlib

class EvAlgo(ABC):
    def __init__(self, popSize, generations, genePool=[], propKilled=.5,␣
 ↪propMutated=.5, graph=[]):
        self.popSize = popSize
        self.generations = generations
        self.genePool = genePool
        self.memo = {}
        if propKilled >= 1 or propKilled <= 0:
            raise ValueError(
                "Invalid proportion killed, must be between 0 and 1")
        if propMutated >= 1 or propMutated <= 0:
            raise ValueError(
                "Invalid proportion mutated, must be between 0 and 1")
        self.propKilled = propKilled
        self.propMutated = propMutated
        self.graph = graph

    def seedGenePool(self):
        for i in range(self.popSize):
            self.genePool += [self.randomGene()]

    def evolve(self, printFrequency=0):
        # Seed gene pool if one wasn't provided
        if not self.genePool:
            self.seedGenePool()
        # Perform all generations
        for i in range(self.generations):
            # Measure fitness of entire gene pool, and sort least fit to most␣
 ↪fit
```

```
            self.genePool.sort(key=self.measureFitness)
            numKilled = int(self.popSize * self.propKilled)
            # Kill propKilled proportion of the population, and replace them
            for j in range(numKilled):
                if random.uniform(0, 1) < self.propMutated:
                    # Mutate
                    self.genePool[j] = self.randomGene()
                else:
                    # Reproduce
                    self.genePool[j] = self.reproduce(self.genePool[random.
↪randint(
                        numKilled + 1, self.popSize-1)], self.genePool[random.
↪randint(numKilled + 1, self.popSize-1)])
            if printFrequency and i != self.generations-1:
                if type(printFrequency) == int:
                    if (i+1) % printFrequency == 0:
                        self.result(i+1)
                if type(printFrequency) == tuple or type(printFrequency) ==␣
↪list:
                    if (i+1) in printFrequency:
                        self.result(i+1)
            self.graph += [self.measureFitness(self.genePool[-1])]

    def result(self, generation=0):
        if not generation:
            generation = self.generations
        print("The most fit gene in generation "+ str(generation) +": ", end='')
        self.printGene(self.genePool[-1])
        print("Achieved measure of fitness: ",
            self.measureFitness(self.genePool[-1]))
        if generation == self.generations:
            pandas.DataFrame(data = self.graph, columns=['Fitness']).plot()

    def printGene(self, gene):
        print(gene)

    @abstractmethod
    def measureFitness(self, gene):
        """return a higher number the more fit the geneome is"""
        pass

    @abstractmethod
    def randomGene(self):
        """return a random gene (tuple)"""
        pass

    @abstractmethod
```

```python
    def reproduce(self, parentA, parentB):
        """Return a child gene (tuple) resulting from two parents"""
        pass
```

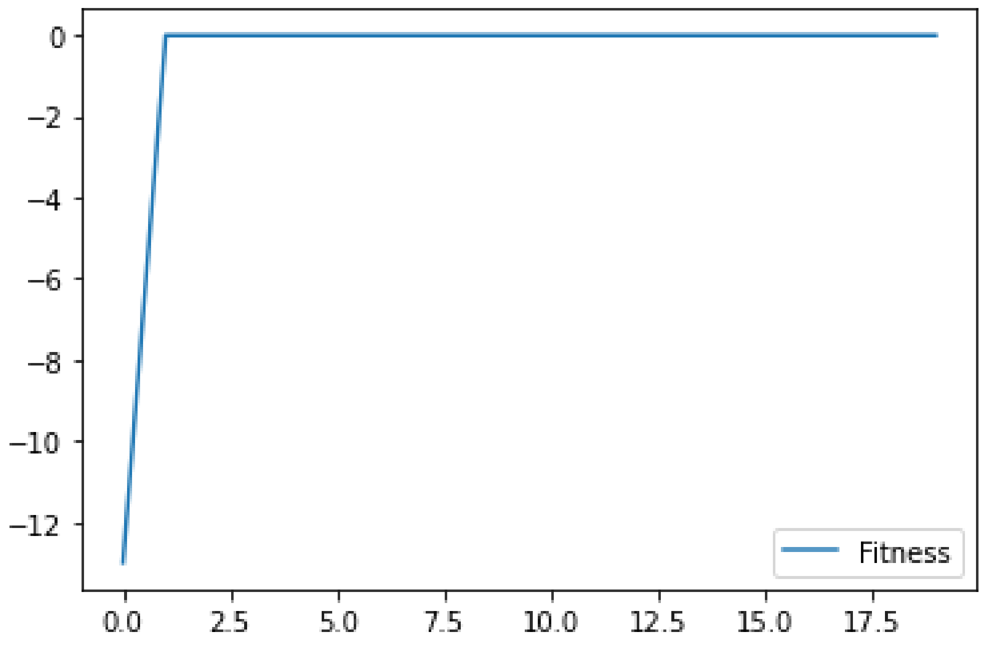
2.3 Application to Non-Differentiable functions

Our first implementation of this evolutionary algorithm was to solve a simple non-differentiable function. Evolutionary algorithms are often applied to problems that can't be solved using easier, cheaper methods, such as differentiation. In our case, we chose a basic non-differentiable function in order to simply showcase the functionality of the computational model: y = -|x-4|

Here, we are trying to maximize the value of y, so we are looking for an x value of 4. To implement this, we simply subclass the EvAlgo class, and override the abstract methods. We then provide paramaters of a population size of 5 genes, for 20 generations. We print the most fit genome and its fitness every 4th generation, and graph the fitness of the most fit genome vs the generation number.

[2]:
```python
from EvAlgo import EvAlgo
import random

class FindFour(EvAlgo):
    def measureFitness(self, gene):
        # Return the y value of our equation
        if gene in self.memo:
            return self.memo[gene]
        else:
            result = -abs(gene[0]-4)
            self.memo[gene] = result
            return result

    def randomGene(self):
        # Return a random x value
        return (random.randint(-100, 100),)

    def reproduce(self, parentA, parentB):
        # Return the average of the 2 x values of our equation, plus some␣
 ↪mutation
        return ((parentA[0]+parentB[0])//2 + random.randint(-10, 10),)

    def printGene(self, gene):
        print(str(gene[0])+". ", end='')

def main():
    x = FindFour(5, 20)
    x.evolve(4)
    x.result()

if __name__ == "__main__":
    main()
```

```
The most fit gene in generation 4: 4. Achieved measure of fitness:  0
The most fit gene in generation 8: 4. Achieved measure of fitness:  0
The most fit gene in generation 12: 4. Achieved measure of fitness:  0
The most fit gene in generation 16: 4. Achieved measure of fitness:  0
The most fit gene in generation 20: 4. Achieved measure of fitness:  0
```



Our fitness, which in this case is the difference between our gene (x value), and 4, increased as the generations went on.

2.4 Application to Maze Solving

For our final application of our computational model, we set out to solve a maze. We need to provide much more functionality because the problem is more complex, so things like measuring fitness is more complex. Other functionality addded includes things like printing the maze. Here, a maze is a two dimensional grid of cells, some of which are walls and some of which aren't, and a gene represents a possible path through the maze in the form of another two dimensional grid of cells, some of which are in the path, and some of which aren't. The fitness is measured differently if a path completes the maze or not. For a path that does not complete the maze, the fitness is a measure of distance from the end of the maze to the connected path that includes the beginning of the maze. Unconnected cells in the path are not measured. All paths that comlete the maze are more fit than all paths that do not complete the maze. For paths that do complete the maze, the fitness is measured as the inverse of the number of cells occupied by the path. Paths that complete the maze with less cells are more fit than paths that complete the maze with more cells. When breeding two genomes (paths), a cell in the child path has an 80% chance of being occupied with a path if both parents have a path there, 50% if one parent has a path there, 20% if neither parent has a path there, and 0% if there is a wall there. To generate a random genome, we generate a two dimensional grid of cells where each cell has a 40% chance to contain a path if it is not a wall, and a 0% chance to contain a path if it is a wall. We call our computational model with a population size of 500, for 1000 generations. We see the most fit genome displayed atop the maze for the 1st, 10th, 100th, and 1000th generation, and we will see the fitness of the most fit genome graphed against the generation number.

[3]:
```python
from EvAlgo import EvAlgo
import random
import functools


class Maze(EvAlgo):
    def __init__(self, popSize, generations, maze, genePool=[], propKilled=.5,␣
 ↪propMutated=.5):
        super().__init__(popSize, generations, genePool, propKilled,␣
 ↪propMutated)
        # The maze will always start at the upper left and end at the lower␣
 ↪right
        # 1 represents a wall, 0 represents no wall
        self.maze = maze
        self.borderChar = '  '
        self.wallChar = '  '
        self.pathChar = '  '
        self.emptyChar = '   '
        self.printRed = lambda x: print('\033[91m' + str(x) + '\033[0m', end='')
        self.printCyan = lambda x: print('\033[96m' + str(x) + '\033[0m',␣
 ↪end='')
        self.printGreen = lambda x: print('\033[92m' + str(x) + '\033[0m',␣
 ↪end='')
        self.height = len(self.maze)
        self.width = len(self.maze[0])
        self.numCells = self.width*self.height
        self.countCells = lambda z: functools.reduce(lambda x,y: x+sum(y), z, 0)

    def printGene(self, path=[]):
        # If no path provided, seed an empty one
        if not path:
            path=[[False]*self.width for x in range(self.height)]
        print("\n",(self.width+2) * self.borderChar, sep = '')
        for i,row in enumerate(self.maze):
            print(self.borderChar, end='')
            for j,cell in enumerate(row):
                # If it is the start or end, print a cyan path char
                if (i == 0 and j == 0) or (i == self.height-1 and j == self.
 ↪width-1):
                    self.printCyan(self.pathChar)
                # If there is both a wall and a path there, print a red path␣
↪char
                elif cell and path[i][j]:
                    self.printRed(self.pathChar)
                # If there is just a path there, print a green path char
                elif path[i][j]:
                    self.printGreen(self.pathChar)
                # If there is nothign there, print the empty char
                elif not cell:
                    print(self.emptyChar, end='')
                # If there is a wall there, print the wall char
                else:
                    print(self.wallChar, end='')
            print(self.borderChar)
        print((self.width+2) * self.borderChar)

    def measureFitnessHelper(self, gene):
        """return a higher number the more fit the geneome (path) is.
        If a path is invalid, it will recieve a fitness score lower than the␣
↪lowest possible valid path score.
        A path will recieve a higher score the closer it is to the exit.
        A path will recieve a lower score the more cells it occupies.
        The most fit genome will be a path that completes the maze, and␣
↪occupies the minimum number of cells.
        """
        # Step 1: Start at the start cell, and create a new path consisting of␣
↪every cell you can get to by just moving adjacently to the start cell.
        # Step 2: If the path found in step 1 contains the end state, the path␣
↪is a solution, the fitness is the
        #            total number of cells in the maze times 2, minus the␣
↪number of cells occupied by the path. Do not continue.
        # Step 3: Start from the end cell, move along the maze in every␣
↪direction there isn't a wall. For example, if you can move up and left, do␣
↪both during this one iteration.
        # Step 4: Make n iterations of Step 3, until you hit the first cell␣
↪occupied by the new path found in step 1. The fitness is the total number of␣
↪cells in the maze minus n.

        # Step 1
        newPath = [[False]*self.width for x in range(self.height)]
        newPath[0][0] = True
        cellsVisited = [(0,0)]
        foundNewCels = True
        while foundNewCels:
            newCellsVisited = []
            for cell in cellsVisited:
                for adjacent in [ (cell[0],cell[1]-1) , (cell[0],cell[1]+1) ,␣
↪(cell[0]+1,cell[1]) , (cell[0]-1,cell[1]) ]:
                    if adjacent[0] >= 0 and adjacent[1] >= 0 and adjacent[0] <␣
↪self.height and adjacent[1] < self.width and not self.
↪maze[adjacent[0]][adjacent[1]] and not newPath[adjacent[0]][adjacent[1]] and␣
↪gene[adjacent[0]][adjacent[1]]:
                        newPath[adjacent[0]][adjacent[1]] = True
                        newCellsVisited += [(adjacent[0], adjacent[1])]
            if not newCellsVisited:
                foundNewCels = False
            else:
                cellsVisited = newCellsVisited

        # Step 2
        if newPath[self.height-1][self.width-1]:
            return 2*self.numCells - self.countCells(gene)

        # Step 3
        visited = [[False]*self.width for x in range(self.height)]
        visited[self.height-1][self.width-1] = True
        cellsVisited = [(self.height-1, self.width-1)]
        foundPath = False
        iterations = 0
        while not foundPath:
            iterations += 1
            newCellsVisited = []
            for cell in cellsVisited:
                for adjacent in [ (cell[0],cell[1]-1) , (cell[0],cell[1]+1) ,␣
↪(cell[0]+1,cell[1]) , (cell[0]-1,cell[1]) ]:
                    if adjacent[0] >= 0 and adjacent[1] >= 0 and adjacent[0] <␣
↪self.height and adjacent[1] < self.width and not self.
↪maze[adjacent[0]][adjacent[1]] and not visited[adjacent[0]][adjacent[1]]:
                        visited[adjacent[0]][adjacent[1]] = True
                        newCellsVisited += [(adjacent[0], adjacent[1])]
                        if newPath[adjacent[0]][adjacent[1]]:
                            foundPath = True
                            break
            cellsVisited = newCellsVisited

        # Step 4
        return self.numCells-iterations

    def measureFitness(self, gene):
        if gene in self.memo:
            return self.memo[gene]
        else:
            result = self.measureFitnessHelper(gene)
            self.memo[gene] = result
            return result

    def randomGene(self):
        path = [[False]*self.width for x in range(self.height)]
        path[0][0] = True
        path[self.height-1][self.width-1] = True
        for i in range(self.height):
            for j in range(self.width):
                if not self.maze[i][j]:
                    if random.uniform(0,1) < .4:
                        path[i][j] = True
        return tuple(map(tuple,path))

    def reproduce(self, parentA, parentB):
        # Probability that cell will be occupied if:
        # Both parents: .8
        # One parent: .5
        # No parent: .2
        path = [[False]*self.width for x in range(self.height)]
        path[0][0] = True
        path[self.height-1][self.width-1] = True
        for i in range(self.height):
            for j in range(self.width):
                if not self.maze[i][j]:
                    if parentA[i][j] and parentB[i][j]:
                        if random.uniform(0,1) < .8:
                            path[i][j] = True
                    elif parentA[i][j] or parentB[i][j]:
                        if random.uniform(0,1) < .5:
                            path[i][j] = True
                    else:
                        if random.uniform(0,1) < .2:
                            path[i][j] = True
        return tuple(map(tuple,path))

def main():
    x = Maze(500, 1000, [
            [0, 0, 0, 0, 0, 0],
            [0, 1, 1, 1, 1, 1],
            [0, 1, 0, 0, 0, 0],
            [0, 0, 0, 1, 1, 0],
            [0, 1, 0, 0, 1, 0],
            [0, 0, 0, 0, 1, 0],
        ])
    x.evolve([1,10,100,1000])
    x.result()
```

```python
if __name__ == "__main__":
    main()
```

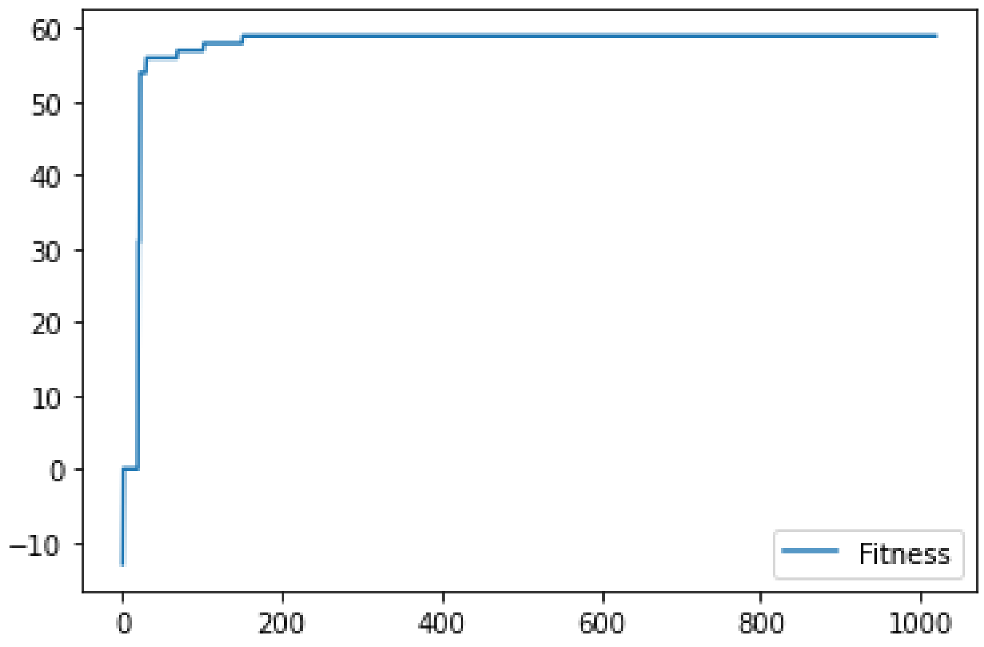
```
The most fit gene in generation 1:
```

```
Achieved measure of fitness:  31
The most fit gene in generation 10:
```

```
Achieved measure of fitness:  54
The most fit gene in generation 100:
```

```
Achieved measure of fitness:  58
The most fit gene in generation 1000:
```

```
Achieved measure of fitness:  59
```



Our initial random paths were scattered and incomplete, and as generations went on the path became more complete, and once the path was complete, the paths became more minimal as generations went on. We also see a clear diminishing return, with great progress in fitness being made in the first generations, and less and less progress being made as generations go on.