Genetic Algorithms

Computer Engineering, Universitat Jaume I
Castellón, Spain

David Dag Mora Zapata
Castellón, Spain
al152047@uji.es

Mihaita Alexandru Lupoiu
La Pobla Tornesa, Spain
al204332@uji.es

***Abstract*— In this article we are going to do a study on Genetic Algorithms, and how we can resolve problems really complex using the Pyevolve extra package of Python. We will also see diferent tipes problems and se how we can resolve them.**

Keywords—genetic; programs; Python; Pyevolve

I. INTRODUCTION

Evolutionary algorithms have been around since the early sixties. They apply the rules of nature: evolution through selection of the fittest individuals, the individuals representing solutions to a mathematical problem. [1]

A genetic algorithm is a form of evolution that occurs on a computer. Genetic algorithms are a search method that can be used for both solving problems and modeling evolutionary systems. With various mapping techniques and an appropriate measure of fitness, a genetic algorithm can be tailored to evolve a solution for many types of problems, including optimization of a function of determination of the proper order of a sequence.

II. SESSION 10: INTRODUCTION TO GENETIC ALGORITHMS WITH PYEVOLVE

A. *Introduction:*

In this session we were given an introduction to the most basic functions of Pyevolve, so we can make scripts for using genetic algorithms, and the use of graphics along with ways to analyze them.

B. *Pyevolve*

Pyevolve [2] was developed to be a complete genetic algorithm framework written in pure python. The main objectives were:

- Written in pure python, to maximize the cross-platform issue;
- Easy to use API;
- See the evolution, the user can and must see and interact with the evolution statistics, graphs, etc;
- Extensible, the API must be extensible, the user can create new representations, genetic operators like crossover, mutation, etc;
- Fast, the design must be optimized for performance;
- Common features, the framework must implement the most common features: selectors like roulette wheel, tournament, ranking, uniform; scaling schemes like linear scaling, etc;
- Default parameters, we must have default operators, settings, etc in all options;
- Open-source, the source is for everyone, not for only one.

C. *Task 1: First Example:*

In this task we had to follow the steps in a tutorial to resolve a problem. The problem was to find a simple 1D list of integers of n-size with zero in all position.

The script was already made, so we only had to copy and understand the functions that were used and practice.

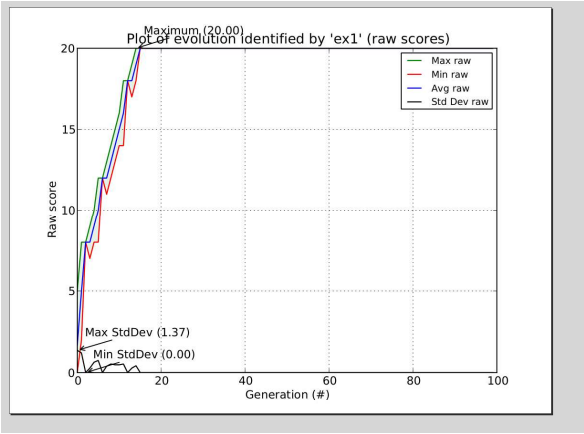

Fig. 1. Resulting Graphic of Evolution of the first script we created in the first task

The next part of this task was to use the “Interactive mode”, which you enter by pressing ESC key before the end of the evolution. When we press ESC, a python environment is loaded, and we can have some analysis functions and we can

interact with the population of individuals at the specific generation.

To understand this better, we downloaded a modified version of the previous program and executed it and analyze it's evolution.

D. Task 2: Graphical Analysis

In this task we had to get more graphics of the previous program, with different data and understand the results of those graphics.

To create the graphic, Pyevolve comes with a plotting tool. But to be able to use is we need to create a database file, and for that we had to introduce this statements in our script of the previous tas:

```
from pyevolve import DBAdapters
…
sqlite_adapter = DBAdapters.DBSQLite(identify="ex1")
ga.setDBAdapter(sqlite_adapter)
...
```

Where "ex1" is the output of the data. But if we create the graphic for the first time, we have to use in the following way:

```
sqlite_adapter = DBAdapters.DBSQLite(identify="ex1", resetDB=True)
```

With that instruction we reset all values and start with a empty database. To display the graphics, we had to run the program and after it finished, in the console we used commands like this:

```
pyevolve_graph.py -i ex1 -1
```

To achieve the graphics from the Fig. 1 and Fig. 2 we had to initiate the program from the Windows console, because it does not seem to work within Speder (our actual Integrated development environment)

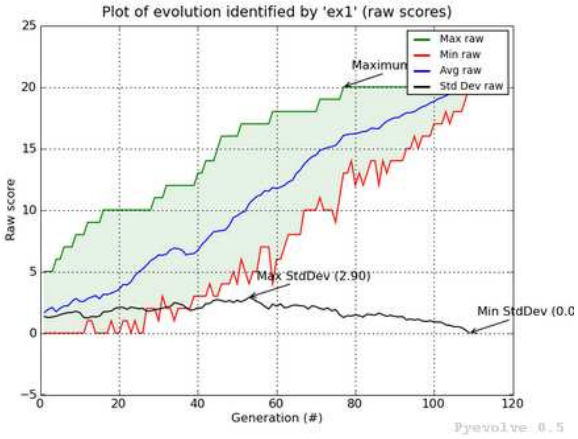

Fig. 2. Resulting Graphic of Evolution of the first script we downloaded in the first task

To end the second task had done the problem: "The Rastring function", where the script was already made, so we only had to run it and understand the graphics.

The Rastrigin function is a non-convex function [referencia a wikypedia] used as a performance test problem for optimization algorithms [http://en.wikipedia.org/wiki/Rastrigin_function] , and by changing some values and we had to draw a graph with the following options:

- Error bars graph (rough results).
- Error bars graph (fitness scores).
- Max / min / avg / std. dev. graph (raw scores).
- Max / min / avg graph (fitness score).
- Raw and min / max graph Fitness difference.

The result was the one we have in Fig. 3.

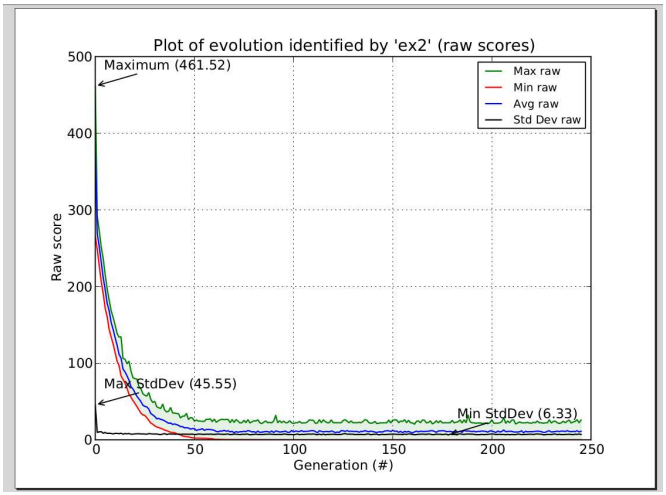

Fig. 3. Graphic result of the The Rastring function

E. Task 3: the Travelling Salesman Problem

The last task of this sesión was to run the program already made and use the plotting tool to generate the graphs of the evolution process, as in the previous task.

The problem we had to resolve was "The travelling salesman problem" (TSP), and asks the following question: Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?[3]

The task consisted two parts. The first part was the execution of the program already done, but reducing the number of iterations of the genes, because because it is a slow process, and because of the the available equipment. [Fig. 4 and Fig. 5]

The second part is to replace the cities that were created randomly in the first part by a txt file of 48 real cities (US state

capitals). To perform this task we modified the program to read these coordinates. [Fig. 6]

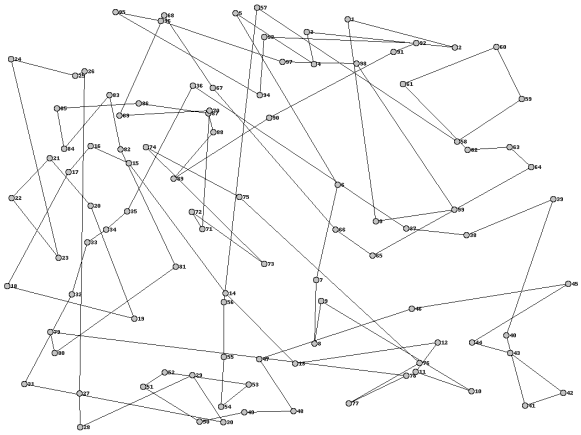

Fig. 4. The travelling salesman problem graphic of 2000 generations

F. Conclusion:

In this sesion we have seen that via the operations of selection, crossover, and mutation the GA will converge over successive generations towards the global (or near global) optimum. Why these simple operation should produce a fast, useful and robust techniques is largely due to the fact that GAs combine direction and chance in the search in an effective and efficient manner. Since population implicitly contain much more information than simply the individual fitness scores, GAs combine the good information hidden in a solution with good information from another solution to produce new solutions with good information inherited from both parents, inevitably (hopefully) leading towards optimality.

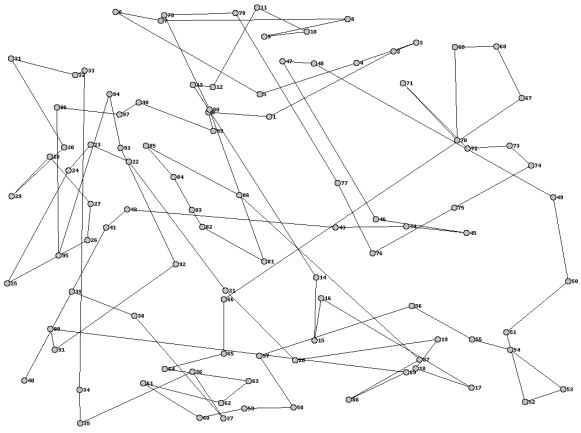

Fig. 5. The travelling salesman problem graphic of 5000 generations

The ability of the algorithm to explore and exploit simultaneously, a growing amount of theoretical justification, and successful application to real-world problems strengthens the conclusion that GAs are a powerful, robust optimisation technique.

III. Session 11: Automatic Generation of Crawling Behavior for a Humanoid Robot Using Genetic Algorithms

A. Introduction:

In this session we perform simulation of a child crawling with the robot "Nao" using artificial genetic evolution of the library we learned to use in the previous session, Pyevolve.

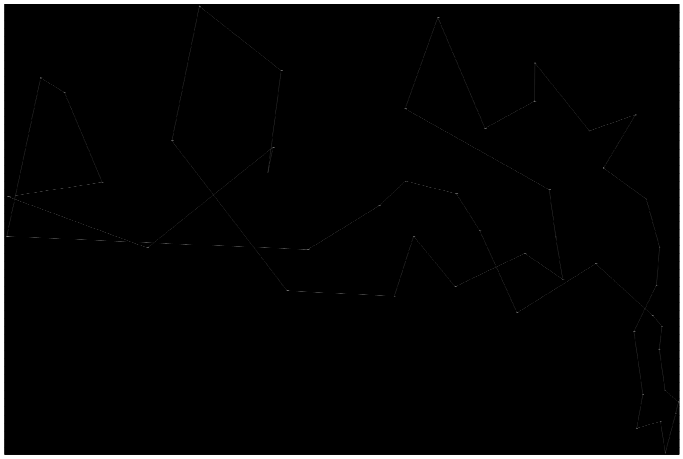

Fig. 6. The travelling salesman problem graph with real cities

B. The Crawling Gait

Some human-like movements are inherently periodic and repeat the same set of steps several times (e.g. walk, crawl, etc). Such a periodic function can be decomposed into a sum of simple oscillators as represented by the following expression:

$$f(t) = C + \sum_{n=1}^{N} A_n \sin\left(n\frac{2\pi}{T}t + \phi_n\right), \forall t \in \Re$$

Applying these oscillators to each joint, a crawling gait can be developed and tested with the simulated humanoid NAO.

The figure [Fig. 7] shows the humanoid structure and the referential axis considered. The main idea behind the definition of this gait is to place an oscillator on each joint we pretend to move in order to define its trajectory. The oscillators are placed on the following joints of the left (L) and right (R) sides: ShoulderPitch, ShoulderRoll, ElbowRoll HipPitch, HipRoll, and KneePitch (note: the "hip" is referred to as "thigh" in the figure). [4]

Therefore, 12 single-frequency oscillators are used. Since each single-frequency oscillator will have 4 parameters to define, 48 parameters are needed to completely define the gait. It is common to assume a sagittal symmetry, which determines the same movements for corresponding left and right sided joints with a half-period phase shift. Hence, it is possible to reduce the number of parameters by half of the original size, resulting on 24 parameters. Additionally, the period of all oscillators should be the same to keep all the joints synchronized by a single frequency clock. This consideration reduces the number of parameters to 19. If the parameters are defined on the left-sided joints, the right-sided joints can be

readily obtained: for roll joints the left and the right side perform the same trajectories over the time; only the sign of the offset needs to be changed. For pitch joints, the right side can be obtained by adding a phase, π, to the corresponding oscillator. The unknown parameters together form the genome that will be used by the genetic algorithm to generate the gait.

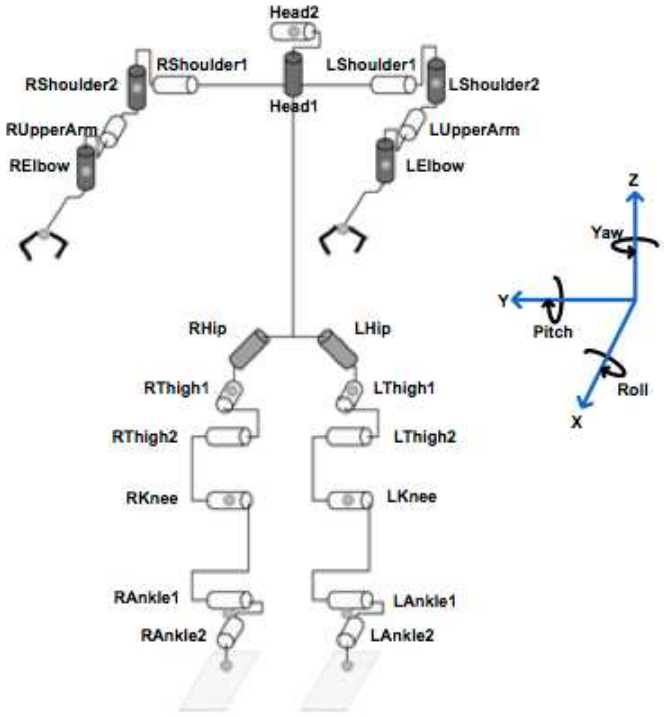

Fig. 7. Humanoid structure and the referential axis considered

C. Simulation setup:

Before we start working we had to prepare the simulator, and for that we downloaded from the aulavirtual a compressed folder that contained several scripts and Python programs to run the simulator and control the Nao robot.

Using the Nao class to control the robot: the file Nao.py contains a class named Nao with the following methods:

- Constructor:
- >>> nao = Nao()
- Standing up:
- >>> nao.stand_up()
- Setting the crawling posture:
- >>> nao.initCrawling()
- Getting the actual posture:
- >>> nao.getActualPose()
- Run a crawling cycle:
- >>> nao.crawl(params, seconds=5)
- Where params is a list with the period, and the amplitude, offset and phase of each joint oscillator, and the result is the distance travelled by the robot during the specified time in seconds:
    - period
    - shoulderPitchA
    - shoulderPitchK
    - shoulderPitchPhi
    - shoulderRollA
    - shoulderRollK
    - shoulderRollPhi
    - hipPitchA
    - hipPitchK
    - hipPitchPhi
    - hipRollA
    - hipRollK
    - hipRollPhi
    - elbowRollA
    - elbowRollK
    - elbowRollPhi
    - kneePitchA
    - kneePitchK
    - kneePitchPhi

D. Example:

>>> from Nao import Nao

>>> nao = Nao()

>>> nao.stand_up()

>>> nao.initCrawling()

>>> nao.crawl([1.28,0.1,0.21,0.,0.035,0.039,-2.,0.12,-0.86, pi,0.06,0.33,pi/2,0.005,-0.11,-2.,0.008,1.8,0.])

E. Task 1: Genetic Algotithm Configurationt

We had to develop a Pyevolve application for the crawling problem, with the following restrictions:

- Use a 1D list chromosome
- An initial population of 10 chromosomes initialized randomly
- Roulette method for selection
- Real range method for mutation, with a probability defined by pm=0.5
- Uniform method for crossover

- Fraction of the population created by crossover defined by pc=0.8

The fitness function had to be chosen carefully in order to achieve good results. A simple but effective fitness function to minimize can was the travelled distance during a fixed time.

The 1D list chromosome accepts the rangemin and rangemax parameters, but they are shared by all the elements of the genome. We had to define a standard range (e.g. [0,1]) and then scale each element to any other pre-defined range, prior to sending the crawl motion to the robot. Scaling was easily calculated by the expression:

robot_parameter = genome_value * (max - min) + min

The suggested ranges for each joint parameter that we were given are shown in the following table:

TABLE I. SUGGESTED RANGES

	min	max
period	*0,8*	*1,8*
shoulderPitchA	*0*	*0,5*
shoulderPitchK	*-0,5*	*0,5*
shoulderPitchPhi	*-3,14*	*3,14*
shoulderRollA	*0,2*	*0,4*
shoulderRollK	*-0,1*	*0,1*
shoulderRollPhi	*-3,14*	*3,14*
hipPitchA	*0*	*0,4*
hipPitchK	*-1,2*	*0*
hipPitchPhi	*-3,14*	*3,14*
hipRollA	*0*	*0,2*
hipRollK	*0*	*0,5*
hipRollPhi	*-3,14*	*3,14*
elbowRollA	*0*	*0,1*
elbowRollK	*-0,5*	*-0,2*
elbowRollPhi	*-3,14*	*3,14*
kneePitchA	*0*	*0,2*
kneePitchK	*1,6*	*1,9*
kneePitchPhi	**-3,14**	**3,14**

F. *Task 2: Experimantal Reslts*

After finishing the script, we run it and generated the the graphs with the evolution process. [Fig. 8 and Fig. 9]

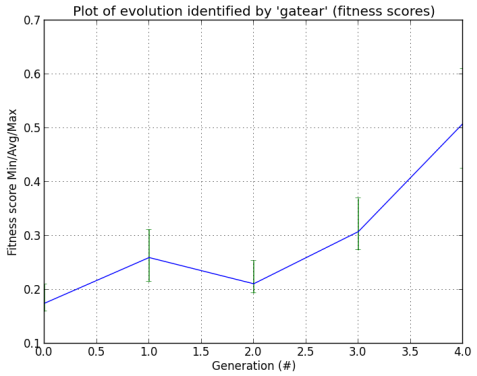

Fig. 8. Error bars graph (fitness scores).

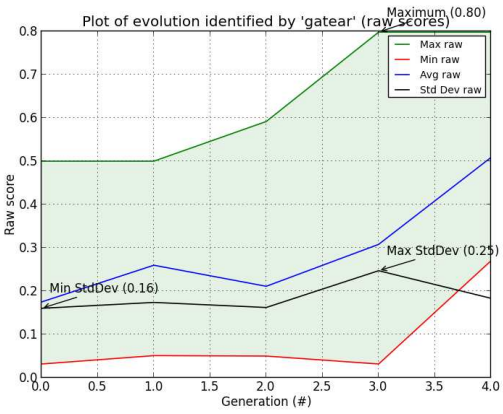

Fig. 9. Max/min/avg/std. dev. graph (raw scores).

G. *Conclusion:*

The described application is a good example for the tasks which research on genetic algorithms should deal with: Because of the nondeterministic behaviour and the long runtimes, genetic algorithms will never succeed against other optimization methods for low complexity problems that allow fast greedy solutions. But for high complexity problems without any known sophisticated solution techniques, a genetic approach is well suited.

IV. SESSION 12: NEUROEVOLUTION

A. *Iroduction*

Neuroevolution (NE), the artificial evolution of neural networks using genetic algorithms, has shown great promise in complex learning tasks. Neuroevolution searches through the space of behaviors for a network that performs well at a given task.

In traditional NE approaches, a topology is chosen for the evolving networks before the experiment begins. Usually, the network topology is a single hidden layer of neurons, with each hidden neuron connected to every network input and every network output. Evolution searches the space of connection weights of this fully-connected topology by allowing high-performing networks to reproduce. The weight space is explored through the crossover of network weight vectors and through the mutation of single networks' weights. Thus, the goal of fixed-topology NE is to optimize the connection weights that determine the functionality of a network.

However, connection weights are not the only aspect of neural networks that contribute to their behavior. The topology, or structure, of neural networks also affects their functionality.

In this session we used a novel NE method called NeuroEvolution of Augmenting Topologies (NEAT) , that is designed to take advantage of structure as a way of minimizing the dimensionality of the search space of connection weights.

B. Task 1: The "Evolving-XOR" experiment

XOR is not linearly separable, a neural network requires hidden units to solve it. The two inputs must be combined at some hidden unit, as opposed to only at the output node, because there is no function over a linear combination of the inputs that can separate the inputs into the proper classes. These structural requirements make XOR suitable for testing NEAT's ability to evolve structure.

To compute fitness, the distance of the output from the correct answer is summed for all four input patterns. The result of this error is subtracted from 4 so that higher fitness will reflect better network structure. The resulting number is squared to give proportionally more fitness the closer a network is to a solution.

TABLE II. RESULTS OF THE EVOLVING-XOR EXPERIMENT:

	Media
Generación	***39.2***
Evaluación	***5498.5***
Nodos Ocultos	***5.2***
Conexiones	***12.1***
Fallos	***0***
Peor Generación	***100***
Peor Evaluación	***13534***

The initial generation consists of networks with no hidden units. The networks has 2 inputs, 1 bias unit, and 1 output. The bias unit is an input that is always set to 1.0. There are three connection genes in each genome in the initial population. Two genes connect the inputs to the output, and one connects the bias to the output. Each connection gene receives a random connection weight.

The work consisted in analysing 10 runs of the XOR script already created and see the difference of the then 10 results.

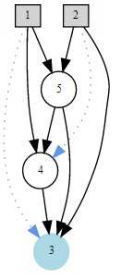

Fig. 10. First run resulting network.

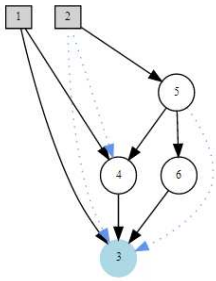

Fig. 11. Fifth run resulting network.

C. Task 2: The "Double Pole" experiment

There are many control learning tasks where the techniques employed in NEAT can make a difference. Many of these potential applications, like robot navigation or game playing, present problems without known solutions.

We use the pole balancing domain for comparison because it is a known benchmark in the literature, which makes it possible to demonstrate the effectiveness of NEAT compared to others. It is also a good surrogate for real problems, in part because pole balancing in fact is a real task, and also because the difficulty can be adjusted. Earlier comparisons were done with a single pole [Fig. 12], but this version of the task has become too easy for modern methods. Balancing two poles simultaneously is on the other hand challenging enough for all current methods. [Fig. 13]

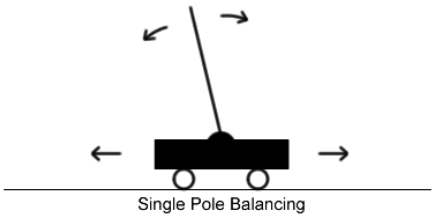

Fig. 12. Single Pole Balancing

In this problem, two poles are connected to a moving cart by a hinge and the neural network must apply force to the cart to keep the poles balanced for as long as possible without going beyond the boundaries of the track. The system state is defined by the cart position and velocity, the first pole's positionand angular velocity, and the second pole's position and angular velocity. Control is possible because the poles have different lengths and therefore respond differently to control inputs.

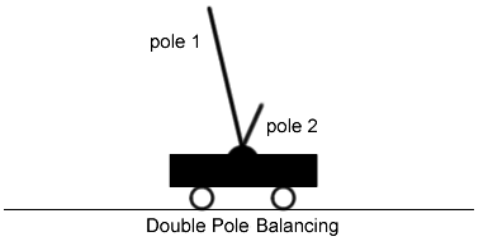

Fig. 13. Doble Pole Balancing

TABLE III. RESULTS TWO POLES

	Media
Generación	***164***
Evaluación	***24865***
Nodos Ocultos	***1.4***
Conexiones	***11.2***
Fallos	***0***
Peor Generación	***260***
Peor Evaluación	***39252***
Score	***299.7***

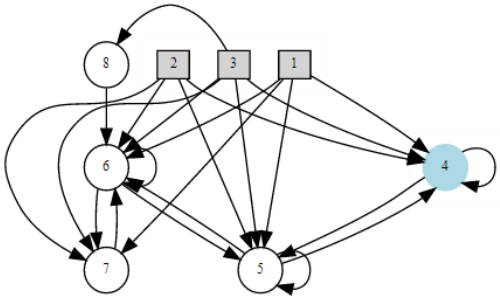

Fig. 14. Fifth run resulting network.

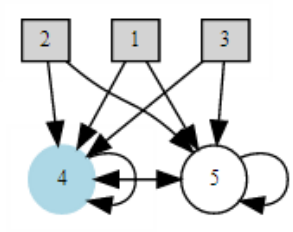

Fig. 15. Fifth run resulting network.

We also did a special experiment that consisted in initiate que poles in a different angle then right and try to evolve until it gets to 1000 generations and the resultant network was Fig.16

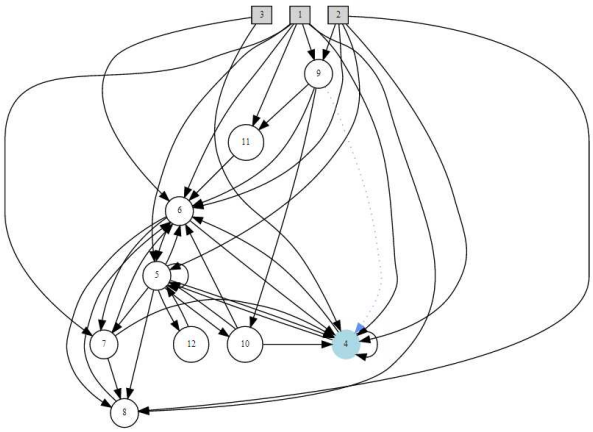

Fig. 16. Resulting network andfter 1000 generations

D. Conclusion:

Neuroevolution is a method of creating artificial intelligence. It is a very powerful tool and it's limits are yet to be found. The more time it evolves the better the result will be. But with the equipment we possess the process is slow, not as slow as it is in the real nature process(billions of years).

REFERENCES

[1] http://subsimple.com/genealgo.asp
[2] http://pyevolve.sourceforge.net/0_6rc1/index.html
[3] http://en.wikipedia.org/wiki/Travelling_salesman_problem]
[4] https://aulavirtual.uji.es/