

# *Extended Essay*

*Computer Science*

Using neural networks and genetic algorithms in a Pac-man game

Jaroslav Klíma  
Candidate D 0771 008  
Gymnázium Jura Hronca  
2003

*Word count: 3959*

**Abstract:**

*The theme of this essay was the application of neural networks and genetic algorithms. We were trying to partially program and partially evolve a neural network that would be able to function as the brain of Pac-man in the famous 2D computer game and compete with human players.*

*We have broken the problem down into four separate sub-problems and proper programs, networks, and/or genetic algorithms were designed to solve those. The resultant „brain“ was tested against human opponents and was found to be competitive.*

*This investigation has shown a possible approach to designing intelligent machines capable of independently learning to solve problems involving a high degree of uncertainty.*

## **Table of contents:**

<b>1. Introduction .....</b>	<b>3</b>
<i>1.1 The Pac-man game .....</i>	<i>3</i>
<i>1.2 The research question .....</i>	<i>3</i>
<i>1.3 The method.....</i>	<i>4</i>
<i>1.4 The testing method .....</i>	<i>5</i>
<b>2. Theory .....</b>	<b>6</b>
<i>2.1 Neural networks.....</i>	<i>6</i>
<i>2.2 Genetic algorithms.....</i>	<i>7</i>
<b>3. Crafting the brain.....</b>	<b>8</b>
<i>3.1 Crafting the “searching” section .....</i>	<i>8</i>
<i>3.2 Crafting the “gold” section .....</i>	<i>9</i>
<i>3.3 Crafting the “creatures” section .....</i>	<i>12</i>
<i>3.4 Crafting the „CPU“ section .....</i>	<i>15</i>
<b>4. Testing and analysis of the results .....</b>	<b>18</b>
<b>5. Conclusion.....</b>	<b>19</b>
<b>6. Bibliography .....</b>	<b>20</b>
<b>7. Appendix A: The breadth first search.....</b>	<b>21</b>
<b>8. Appendix B: Program code .....</b>	<b>22</b>
<i>8.1 Class CQueue .....</i>	<i>22</i>
<i>8.2 Evolution code .....</i>	<i>24</i>

# **1. Introduction**

The idea of artificial intelligence has been a widely discussed and popularized topic at all levels of professionalism. First came the sci-fi novels, the authors of which went crazy about intelligent, self-operated machines with all kinds of human abilities. Today, there is already a heavy mathematical theory concerning the topic of artificial intelligence and dreams about intelligent machines are slowly coming true...

## ***1.1 The Pac-man game***

You certainly remember the old computer game Pac-man. Just to refresh the memory, the game consists of a maze, in which Pac-man searches for gold coins and is hunted by a number of creatures. The player has to use arrow keys to tell Pac-man which direction to take in order to collect all of the coins while avoiding contact with the creatures. For the purpose of this essay, we will use the following set of rules:

- A tile represents either a wall or a room
- The maze consists of 15x15 tiles. The tiles on the border are always walls.
- Each “room” tile can be either empty or can contain gold, the player and/or a creature.
- The player and the creatures can move one tile in every turn or can remain where they are.
- There are one player, 3 creatures and 30 gold coins in every game
- The player and the creatures can “see” a square area of  $5^2$  tiles around themselves
- When the player and a creature stand on the same tile or try to switch their positions in one turn, the player gets eaten and loses the game
- When the player stands on a tile that contains gold, his score increases and the gold disappears. When there is no gold left, the player wins.

## ***1.2 The research question***

The interesting thing about this game is, that there is no way to compute the perfect move. Calculating the perfect move would be hard even if the player could see the whole maze, because of the unpredictable behavior of the creatures, but seeing only a

limited number of tiles around makes the calculation impossible. The player has to make the decision using his previous experience, which basically means running the current game state through his brain producing a certain output in form of the next move. While this approach is perfectly suitable for the human brain, it is apparently a complicated task for the computer. The high degree of uncertainty involved in the calculation makes this task a problem to be solved using so-called “artificial intelligence”. When solving the problem, most programmers only implement a function that works good enough for calculating the moves. What we want to do now is create an intelligent algorithm that will be able to develop itself into a player of Pac-man, which means that it will be able to first *train* itself and then use its *experience*. Because we want the program to simulate the human brain, we will use the concept of neural networks. Finally, the research question:

***“Are we able to create such a neural network and a “teacher” program that after some time of self-development the network will be able to play Pac-man as good as humans?”***

If we are successful, the resultant artificial player will be able to perform all tasks necessary to successfully play the game. It will effectively search through all the maze tiles and collect any gold in sight in the most effective manner while avoiding contact with the creatures.

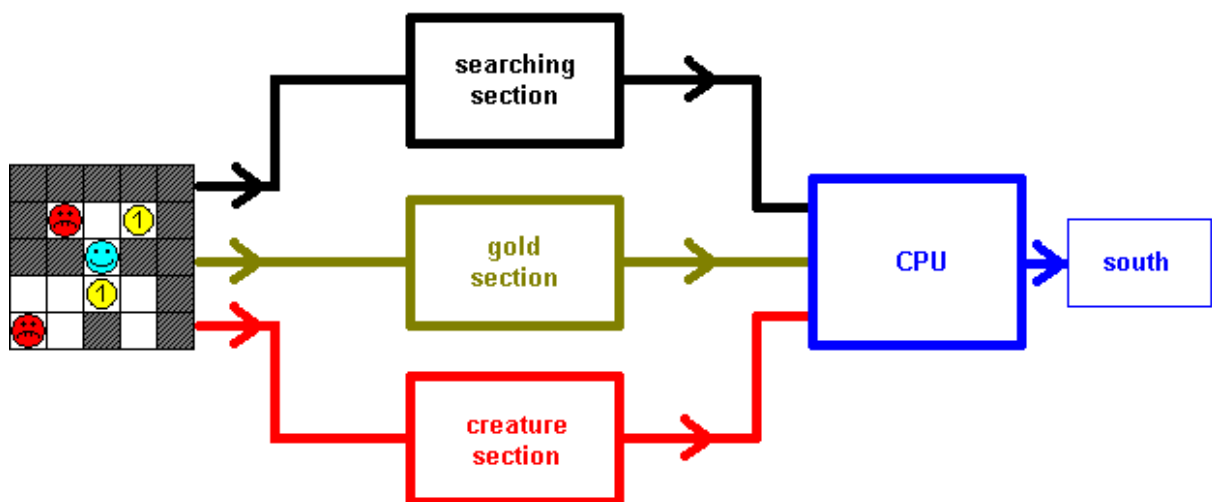
### ***1.3 The method***

Because the number of tiles that Pac-man can see is always  $5^2$ , this will also be the number of our network’s inputs. Based on those inputs, our network will have to decide which direction to take and produce one output that will represent this direction. The brain will have four sections, each crafted separately:

- “Searching” section will take care of searching through the whole maze to avoid staying on the same place and search all reachable tiles instead. This section will have to include some kind of memory, because Pac-man will have to be able to retrace his steps back and take a different path.

- “Gold” section will take care of collecting all visible gold. It should optimize every move for collecting the highest amount of gold possible in the shortest time.
- “Creature” section will take care of avoiding contact with creatures. In every turn, it will have to decide which move is the “safest”.
- “CPU” section (the Central Processing Unit) will consider the output of the previous three sections and produce the final output.

All put together, the brain will look as shown in figure 1.3.1. Throughout this essay, we will develop all the brain sections and design ways to train them.



***Figure 1.3.1:*** *The basic outline of the resultant network*

#### ***1.4 The testing method***

We will test our resultant neural network against a real human brain. We will create 10 game scenarios and let two humans and our network play them. If the cumulative score of our artificial player is higher than the score of one of the humans, we will declare our experiment a success, because we will have an artificial network capable of competing with human beings.

## **2. Theory**

Two of the most widely used practices in developing artificial intelligence are those of using “neural networks” and “genetic algorithms”.

### ***2.1 Neural networks***

The idea behind neural networks is simulating the function of a biological brain. A network consists of interconnected “*neurons*” with different *activations*. Each neuron has a number of “*dendrites*”, a “*soma*” and an “*axon*”. A dendrite is a weighted input to the neuron. The soma is an *activation function*, which produces certain output based on the inputs. The axon stores the neuron’s activation (output), the last value produced by the soma. [4]

The neurons are connected by their dendrites and axons. A dendrite connects the axon of one neuron to the soma of another making the outputs of several neurons the inputs to another one.

Some of the neurons in the network are declared the inputs and some are declared the outputs. The axons of the input neurons then store the input values of the network and the axons of the output neurons store the output of the network.

The output of every neuron is equal to the result of its activation function  $f(x)$ . The input  $x$  of the activation function is the sum of all inputs multiplied by their weights. The most frequent activation functions are step, linear and exponential. Step functions produce activations of either 0 or 1, and are used when we want the neuron to “fire” when a critical level of input is reached. Linear functions result in linearly dependent on the input and are used when we want the neuron to more closely follow the input activation. Exponential activation functions are used when we need the neuron to have non-linear response. [5]

## 2.2 Genetic algorithms

Genetic algorithms are used in solving problems that do not have a straightforward algorithmic solution or this solution would be too expensive. The idea is to *evolve* the solution. First, we have to generate an initial *population* (set of possible results) of given size and then evolve next *generations* of the population in the same manner as it is done in the nature. We “mate” random members of the current population producing *offspring*. This way, the population would increase, but we avoid this by keeping only the fittest of the offspring. The evolution is terminated when the terminal state is reached. [1]

The size of the population, the measure of fitness, the process of mating and the terminal state have to be defined for each specific problem separately and their definition is the determinant of the algorithm’s success.

### **3. Crafting the brain**

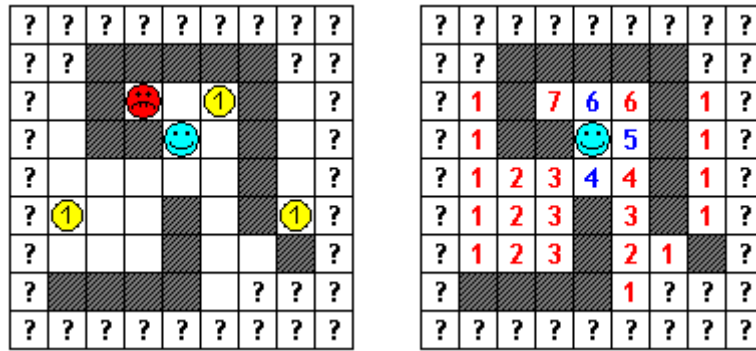
In every turn, there are five possible decisions to make. Pac-man can go north, east, west, or south or he can decide not to move. Every one of those five possibilities can be assigned its value in each turn. We will call this value the direction's score. The direction with the highest score will be returned as the resultant move. If two or more directions receive the highest score, one of them will be chosen at random. And how do we assign each direction its score? This is the task of all of the brain sections. Each one of the first three will receive its own input and produce its own output. The CPU will then consider the three outputs and make the final decision.

#### ***3.1 Crafting the “searching” section***

This section of the brain will tell Pac-man where to go regardless of the gold and the creatures. Its purpose is to make him search through the whole maze and not move back and forth on one place.

In order for this to be possible, Pac-man will have to remember where he has already been. Because this problem has a straightforward solution, we will not use a network, but a simple computer memory. In this memory, Pac-man will remember the whole maze, except that tiles that are not yet uncovered will be marked “unknown”.

Based on this memory, the “searching” section of the brain will make Pac-man want to move in such direction as to come as close to the unknown tiles as possible. Once again, this problem has a straightforward algorithmic solution and therefore we do not need to use a network. We will use the “breadth first search” algorithm (described in the appendix) instead to compute the distance to the closest unknown tile for every already uncovered tile and then see which direction will be the best. This is not the only method of finding the shortest path, but in our case it is the best, as we need to perform the search only once to obtain the shortest distance for all five possible moves. A sample situation is shown in figure 3.1.1.



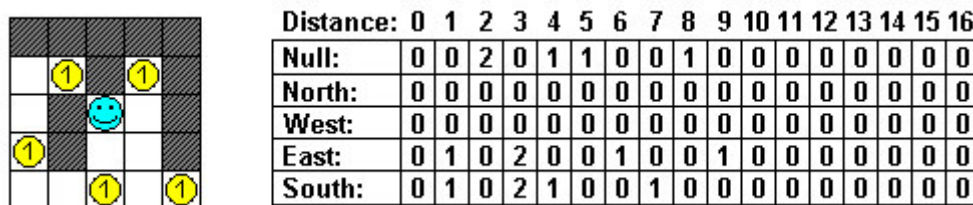
**Figure 3.1.1:** A sample situation and how it is handled by the “searching” section of the brain

When we have the distance in all directions, we will create the output by taking the least of the distances, and call it a “numerator”. The output for every direction will then be the numerator over the distance in that direction. For our example, the (Null, North, West, East, South) output for our sample situation would be  $(4/5, 4/6, 0, 4/5, 4/4) = (0.80, 0.68, 0.00, 0.80, 1.00)$ .

This section will make Pac-man explore all tiles that are possible to uncover, because it will always move towards the nearest unknown tile and when he comes close enough, the tile will be uncovered.

### 3.2 Crafting the “gold” section

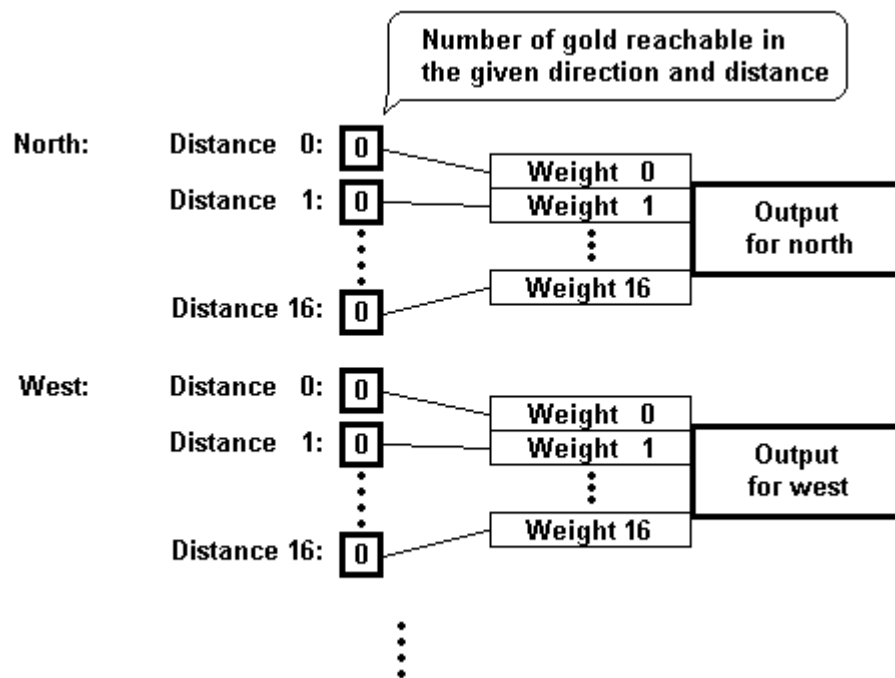
This part of the brain should be able to tell us in which direction to move in order to obtain gold. Computing the distances to gold in every direction can be done using the breadth first search algorithm again. A sample situation is shown in figure 3.2.1.



**Figure 3.2.1:** A sample situation and the corresponding input to the “gold” section

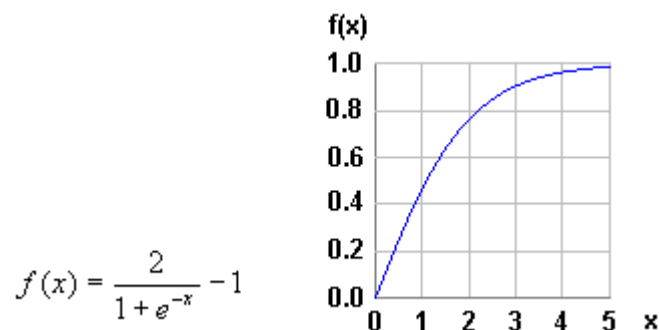
Once we have the input in this form, we have to find a way to evaluate every possible move. Because we do not know how to evaluate the presence of gold in different distances, we will create a neural network that will learn to make the best decisions.

This network will look as shown in figure 3.2.2. There are only 17 dendrites for each distance, because within a 5x5 tile square, gold is always reachable in 16 steps or less.



**Figure 3.2.2:** The structure of the "gold" section of the brain

The activation function for every neuron will be exponential, because we need to obtain a value between 0 and 1 and the response should not be linear. We don't want the output to be much different when there are 21 coins in one direction and 20 in another, but we want it to differ greatly when the difference is 1 and 2 coins. For those reasons, we will use the equation  $f(x) = \frac{2}{1 + e^{-x}} - 1$  shown in figure 3.2.3. [5]



**Figure 3.2.3:** The activation function of the neurons in the "gold" section

The weights are the same for every direction, so we only need to find 16 values. One thing we can say for sure about them is that weight  $W_A$  will always be greater than weight  $W_{A+1}$ . It is because we know that gold which lies closer should always attract Pac-man more than gold that lies farther away.

Based on those facts, we will use a genetic algorithm to evolve the most suitable set of weights. The initial population will be 100 sets of weights evenly distributed throughout the scope.

The measure of fitness will be the cumulative number of gold coins collected in thirty randomly generated games. The set of games will stay the same throughout the testing of the whole generation to avoid problems with uneven testing conditions.

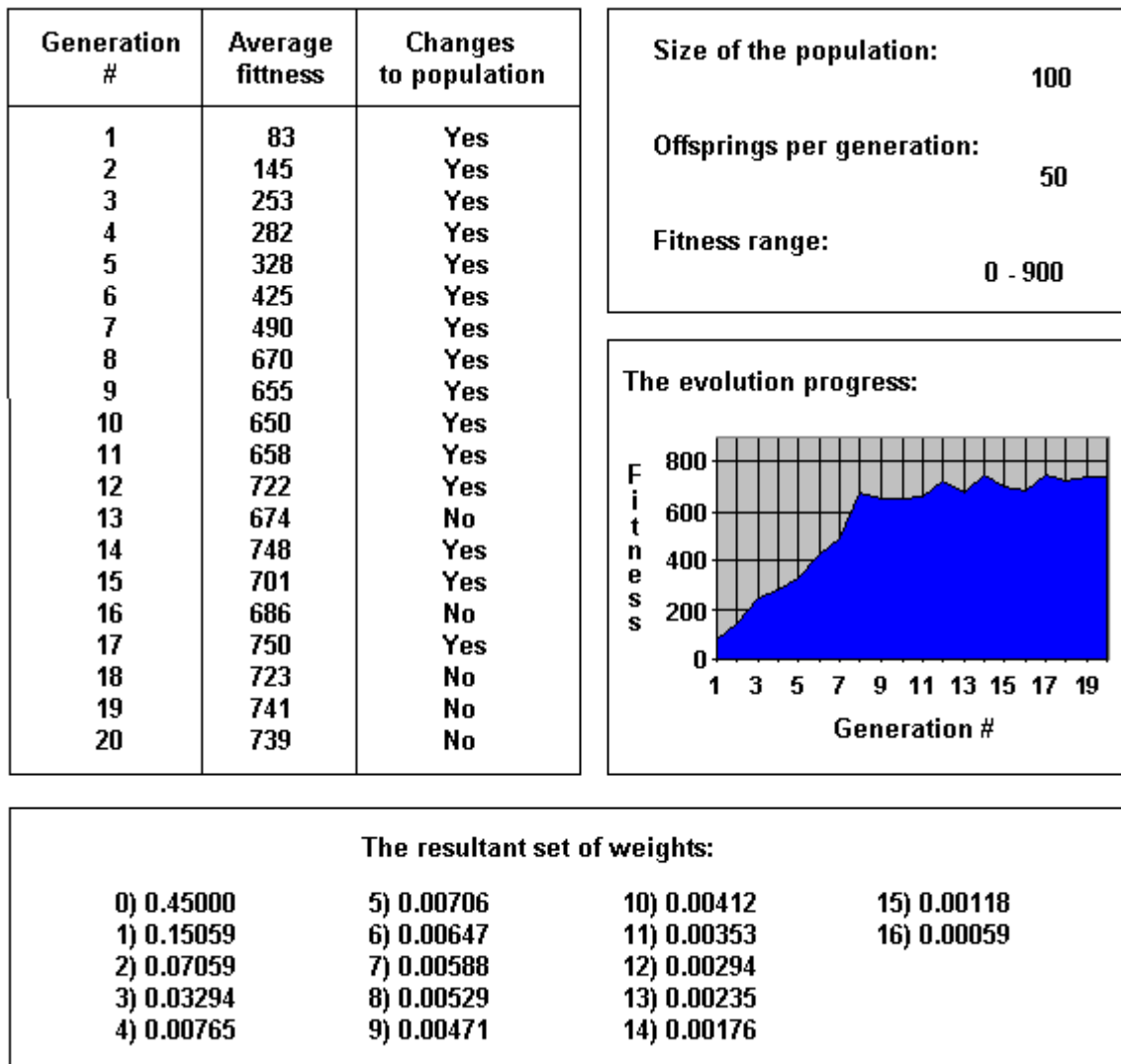
The mating process will be done as follows. We will choose two random “parents” from the population. Then we will choose randomly one of the parents as the donor of each of the 16 weights. The newly created set will then be added to the population.

The terminal state will be reached when the population does not change during 3 generations, which means that during 3 generations no offspring performed better than any of the current members.

To decide which way to move, we will use this section alone. The direction that will receive the highest score in each turn will be used. If two or more directions will receive the same score, one of them will be chosen randomly. If all directions receive the same score, we will use the “searching” section to uncover new locations.

Because there will be no creatures, we will limit one game to 50 turns to avoid problems with weight sets that never collect all gold. This way, those sets will score less and will not end up stuck in one game forever.

A sample run of the evolution of the “gold” section of the brain is shown in figure 3.2.4. There are some interesting facts about the evolution process. The most apparent is that the fitness decreased several times during the process. The cause of this was that a new set of games was created for every generation and therefore even a “better” generation could score less than the “worse” one. Another interesting fact was that the average fitness of a generation never exceeded 800. This was caused by the 50-turn limit – Pac-man simply did not have enough time to collect more coins.



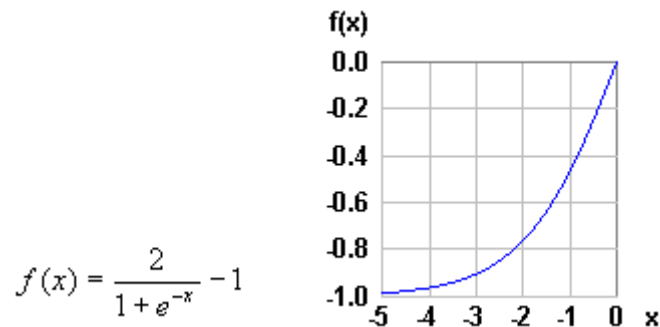
**Figure 3.2.4:** A sample run of the evolution of the “gold” section

### 3.3 Crafting the “creatures” section

This problem is very similar to that of hunting gold, except for the fact that the presence of a creature in a certain direction will decrease the chance of going in that direction instead of increasing it. The resultant values should therefore be negative or zero.

The structure of the network will be the same as that used in the “gold” section, but the weights will be different. Also the activation function will differ (in its scope), since we need the output neurons in this section to fire negative values

We will use the same method as we did with the “gold” section. We will again evolve 16 weights for a network that will make Pac-man decide where to go based on the distances to creatures in each direction.



**Figure 3.3.1:** *The activation function of the neurons in the “creatures” section*

The rules for the evolution of the set of weights will be defined as follows:

The initial population will be a hundred sets of weights evenly distributed throughout the scope.

The measure of fitness will be the cumulative number of steps made before being “eaten” by the creatures in thirty randomly generated games. The set of games will stay the same throughout the testing of a whole generation to avoid problems with uneven testing conditions.

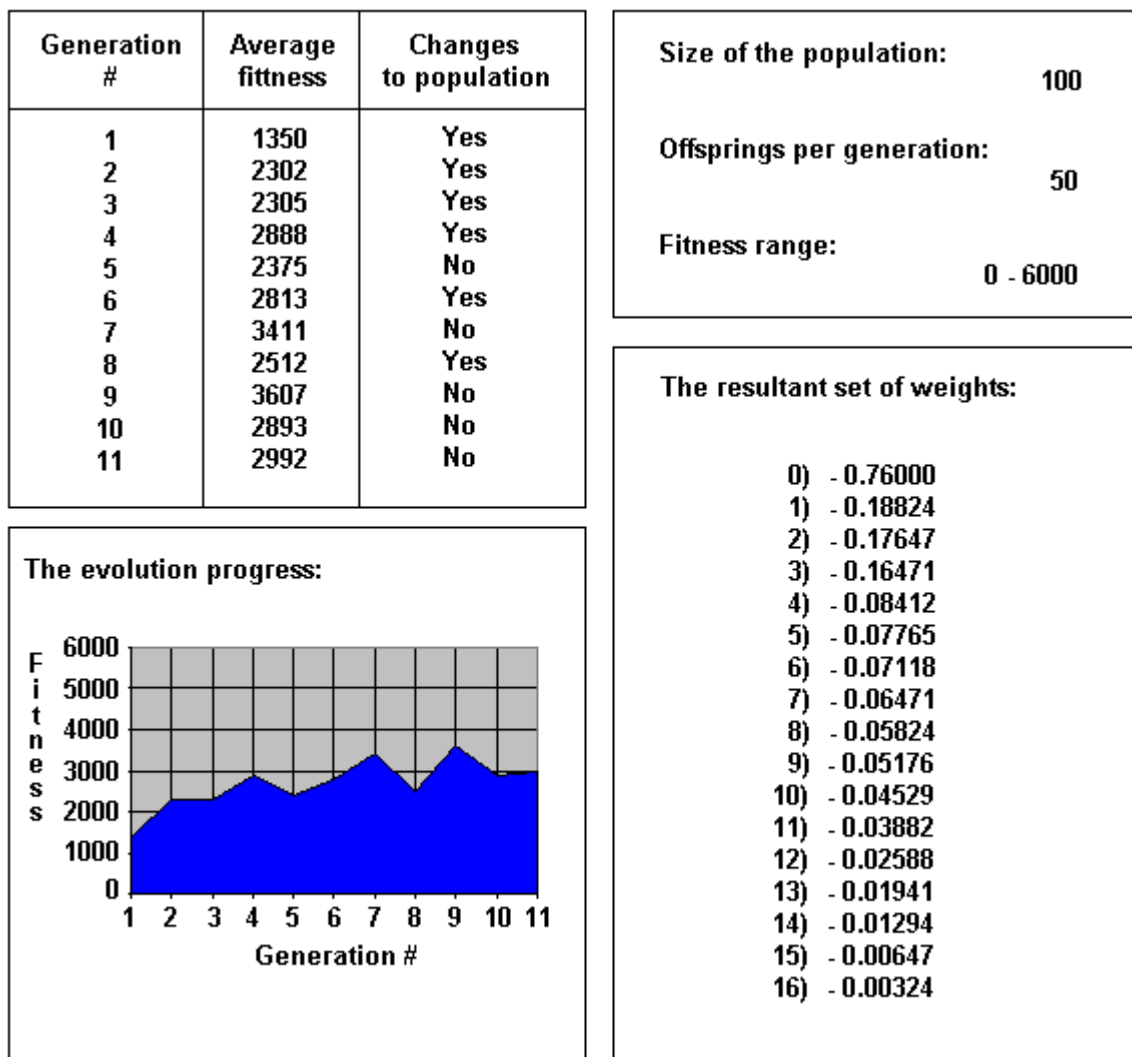
The mating process will be done as follows. We will choose two random “parents” from the population. Then we will choose randomly one of the parents as the donor of each of the 16 weights. The newly created set will then be added to the population.

The terminal state will be reached when the population does not change during 3 generations, which means that during 3 generations no offspring performed better than any of the current members.

To decide which way to move, we will use this section alone. The direction that will receive the highest score in each turn will be used. If two or more directions will receive the same score, one of them will be chosen randomly. If all directions receive the same score, we will use the “searching” section to uncover new locations.

We will limit one game to 200 turns, which will be the best possible fitness, to avoid getting stuck with a set of weights that will be able to avoid creatures forever. Such set will only receive the highest fitness rating.

The results of the evolution are shown in figure 3.3.2. The evolution was a little less successful than when we were creating the “gold” section, but still sufficient. When we look at the table of the resultant weights, we see, that a creature standing right beside a player is VERY important, one standing two or three tiles still has a big impact, but creatures standing farther away are not considered too dangerous.



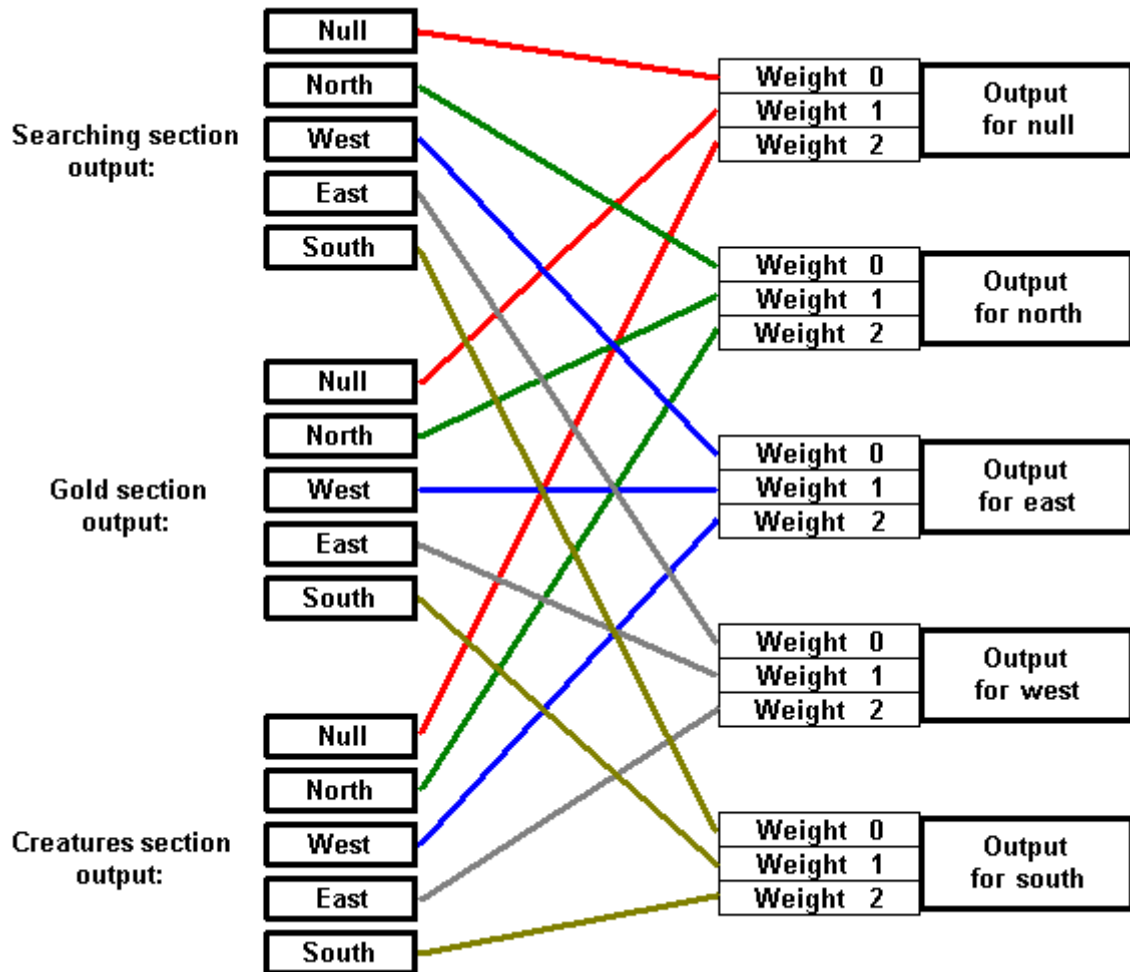
**Figure 3.3.2:** A sample run of the evolution of the “creatures” section

### ***3.4 Crafting the „CPU“ section***

Now that we have all the parts of the brain functioning, we need to decide about their order of importance. That will be the task for the CPU section. This section will take the outputs of the previous three sections as its input and produce the final rating for every one of the five possible moves. Its structure will therefore be as shown in figure 3.4.1.

There are only three dendrites for each direction, and yes, their weights will be always the same, which means that we only have to find three values. To do so, we will use a genetic algorithm, again. This time, the neurons will have a linear activation function, because we want the response to depend linearly on the output activations of the previous three sections. The function can happily be  $f(x) = x$ , because it is linear and we do not need to scale it, as the results will not be used anywhere else.

We will start with a set of a hundred sets of three random weights, each in range between 0.0 and 99.9. The lower boundary is zero because we only want positive numbers (including zeros) and the upper boundary was set to 99.9 just to make sure that there is enough space for experimenting. Theoretically, the upper boundary should be infinite, but because the sum of the inputs is always less than three, a hundred will surely be enough.



**Figure 3.4.1:** *The structure of the CPU section*

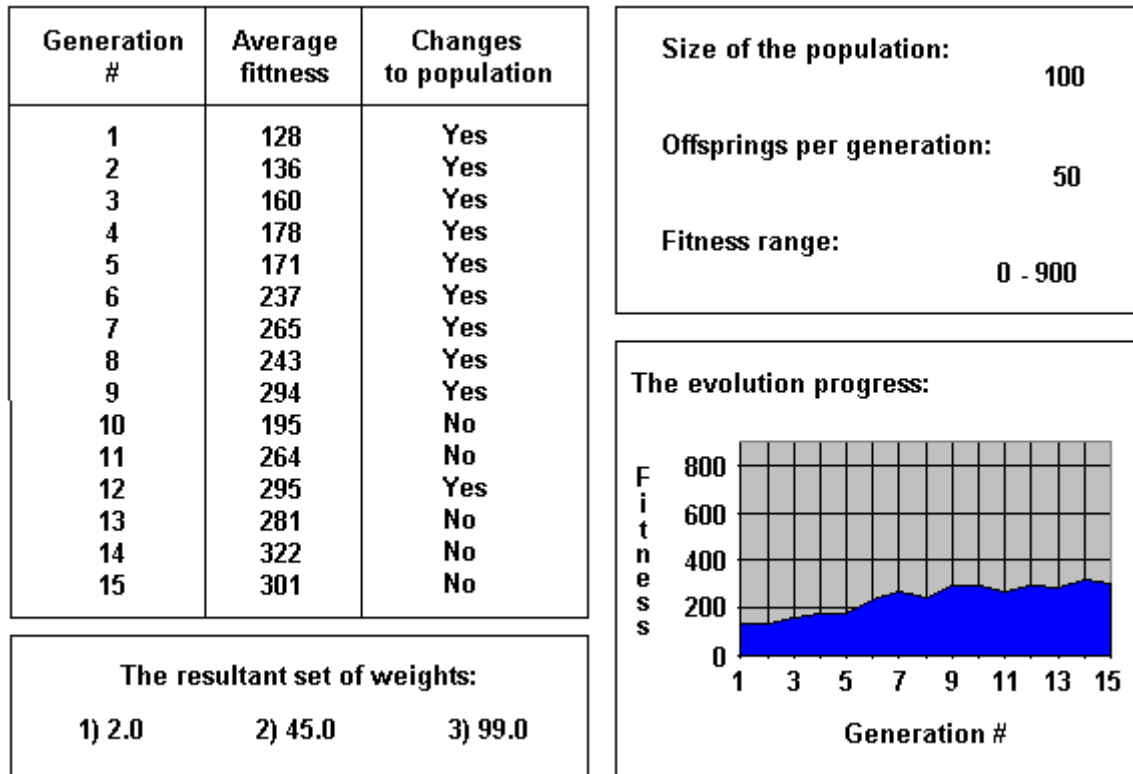
The measure of fitness will be the cumulative score from 30 randomly generated games with full set of features.

The mating process will be done in the same way as it was done in the previous sections. We will choose two random “parents” from the population. Then we will choose randomly one of the parents as the donor of each of the three weights. The newly created set will then be added to the population.

The terminal state will be reached when the population does not change during 3 generations, which means that during 3 generations no offspring will have performed better than any of the current members.

Sample results of the evolution process are shown in figure 3.4.2. We have managed to evolve a brain that collects one third of the available coins on average before being eaten by the creatures. As we can see in the table looking at the resultant weights,

avoiding creatures has the biggest priority. When there are no creatures too close, Pac-man will try to grab any visible gold and if there is no gold, only then will he go and explore new areas. The question is, if our brain can compete with human beings?



***Figure 3.4.2:*** Sample result of the evolution of the CPU section.

## 4. Testing and analysis of the results

The testing was done in the way described in the testing method. Two humans (players A and B) and a computer using our artificial brain (player C) were given a set of 10 games to play. Every player had the exact same set of games with the same initial position of Pac-man, the creatures and gold to make sure that all players play under the same conditions. Player A was chosen to be the person with more experience in playing computer games, Player B was less experienced in this area.

Game #	Player A		Player B		Player C	
	Score	Turns	Score	Turns	Score	Turns
1.	20	54	14	42	13	20
2.	13	48	9	23	17	36
3.	16	30	9	17	15	32
4.	6	10	16	48	16	46
5.	30	83	14	53	13	31
6.	22	50	23	63	21	35
7.	25	43	18	46	4	7
8.	19	46	20	52	14	38
9.	21	63	16	41	15	31
10.	19	62	17	64	18	41
<i>Average</i>	<i>19.1</i>	<i>48.9</i>	<i>15.6</i>	<i>44.9</i>	<i>16.0</i>	<i>31.7</i>
<i>% of max.</i>	<i>64%</i>	-	<i>52%</i>	-	<i>53%</i>	-

***Figure 4.1:*** *The results of testing*

As we can see on the table of results, our network performed well. Its score was only about 2.5% better than the score of the worse of the humans – player B – but the artificial brain proved his ability to compete with human beings. What is more, when we look at the scores and the numbers of turns played, we see that the artificial brain was the most efficient: it has collected one gold coin in two turns on average, compared to 2.5 turns in case of player A and 2.8 turns in case of player B. On the other hand, efficiency is not the goal of this game, but it reveals some characteristics of our network. It is less afraid of the creatures than people are, which results in higher rate of being eaten by monsters, but is much more efficient in collecting the gold coins.

## **5. Conclusion**

The goal of this essay was to design a neural network and a set of algorithms that would be able to learn to play Pac-man good enough to be competitive with humans. This goal was fulfilled, as our artificial brain was able to obtain better score in a set of games than one of the human players. Still, there is a lot of space left for future investigation. For example, the brain shows some insufficiency in his ability to escape the creatures by running into areas from where there is no way to escape. Maybe a different approach should be used in order to solve this.

To wrap up, we can say that in his essay we have shown a new way of teaching the computer mathematically incomputable things that involve a high degree of uncertainty and are based mostly on experience. This approach is not applicable only to turn-based eat-and-run computer games, but can be very well used in practical life. Machines could replace humans in performing tasks that involve a degree of uncertainty and experience, for example a builder-robot could be taught to adopt itself to a new or changing environment by „evolving“ his command set.

Neural networks and genetic algorithms take us one step closer to simulating the function of real human beings and even if this is probably the sound of far future, the can today already be of great help.

## **6. Bibliography**

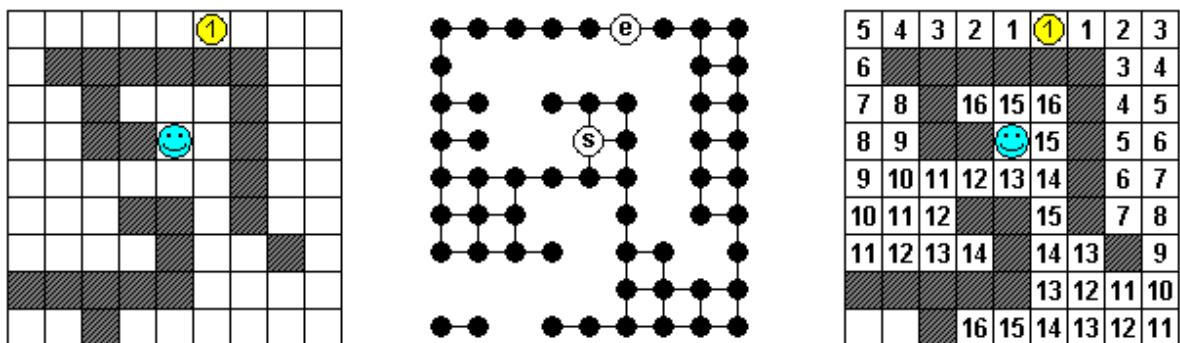
- [1] LaMothe, A.: Tricks of the Windows Game Programming Gurus . USA, Sams, 1999
- [2] Petzold, C.: Programming Windows . Praha, Computer press, 1999
- [3] Rychlík, J.: Programovací techniky . Ceske Budejovice, KOPP, 1994
- [4] LaMothe, A.: Neural Netware . URL: [www.GameDev.net](http://www.GameDev.net)
- [5] Generation<sup>5</sup>: Neural network essays . URL: [www.Generation5.org](http://www.Generation5.org)

## 7. Appendix A: The breadth first search

The breadth first search is one of the techniques described in the mathematic theory of graphs. It is used to find the shortest path from one node of the graph to another. The searching is performed as follows: [3]

- 1) Create an empty queue
- 2) Take the target node and insert it into the queue and mark it “done” and assign it distance 0
- 3) Withdraw a node from the queue and call its assigned distance “d”. Insert all nodes connected to this node and not marked “done” into the queue, mark them “done” and assign them distance d+1.
- 4) Repeat step 3 until the queue is empty
- 5) If a path exists from the starting to the target node, the distance assigned to the starting node is the length of the shortest path to the ending node.

This algorithm can be easily used for searching mazes consisting of tiles. We just declare every “room” tile to be a graph node connected to the nodes of all bordering tiles. An example of such search is shown in figure 7.1.1



**Figure 7.1.1:** A sample situation and how it is handled by the breadth first search algorithm.

## 8. Appendix B: Program code

This appendix includes the pseudo-code of the queue class and the evolution algorithm. The queue class implements a queue used in the breadth-search algorithm. The evolution algorithm evolves a set of data with the highest fitness. The algorithm is written in pseudo-code and shows the general procedure. Details of the implementation are described in the body of this essay.

### *8.1 Class CQueue*

```
\\----- Queue.h -----  
  
class CQueue  
{  
    private:  
        DATA* m_pQueue;  
        DWORD m_dwLength;  
        DWORD m_dwSize;  
        DWORD m_dwFirst;  
        DWORD m_dwLast;  
  
    public:  
        CQueue(DWORD size);  
        virtual ~CQueue();  
  
        void Enqueue(DATA Data);  
        void Withdraw(DATA* Data);  
  
        bool Empty();  
};  
  
//----- Queue.cpp -----  
  
#include "Queue.h"  
  
// Default constructor  
CQueue::CQueue(DWORD size)  
:m_dwSize(size), m_dwLength(0), m_dwFirst(1), m_dwLast(0)  
{  
    m_pQueue = (DATA*)malloc(size * sizeof(DATA));  
}  
  
// Virtual destructor  
CQueue::~~CQueue()  
{  
    free(m_pQueue);  
}
```

```
// Returns if the queue is empty
bool CQueue::Empty()
{
    if (m_dwLength>0)
        return false;
    else
        return true;
};

// Enqueues new data in the queue
void CQueue::Enqueue(DATA Data)
{
    m_dwLength++;
    m_dwLast++;
    m_dwLast = m_dwLast % m_dwSize;
    m_pQueue[m_dwLast] = Data;
};

// Withdraws data from the queue
void CQueue::Withdraw(DATA* Data)
{
    *Data = m_pQueue[m_dwFirst];
    m_dwFirst++;
    m_dwFirst = m_dwFirst % m_dwSize;

    m_dwLength--;
};
```

## 8.2 Evolution code

```
DATA EvolveData()
{
    // Create a buffer for the initial population
    DATA Population[100];

    // Create a buffer for saving the fitness of the
    // population
    int Fitness[100];

    // Create the initial population
    for (int i=0; i<100; i++)
    {
        Population[i] = CreateData[i];
    };

    // Begin evolution
    for (Evolution = 0; Evolution<20; Evolution++)
    {
        // Create a buffer for thirty games
        CGame Games[30];

        // Generate the thirty games
        for (int g=0; g<30; g++)
        {
            Games[g].Generate(false);
            Games[g].SaveToFile(g);
        };

        // Clear fitness
        memset(Fitness, 0, 100*sizeof(int));

        // Let the population play the games
        for (CurrentData=0; CurrentData<100; CurrentData++)
        {
            // play the games
            for (int g=0; g<30; g++)
            {
                Games[g].LoadFromFile(g);
                Fitness[CurrentData]
                    += Games[g].Play(CurrentData);
            };
        };

        // Mate random data
        for (i=0; i<50; i++)
        {
            // Choose the parents
            Parent1 = rand() % 100;
            Parent2 = rand() % 100;
            DATA OffspringsData;

            // Create the offspring
            OffspringsData = MateData(Population[Parent1],
                Population[Parent2]);

            int OffspringsFitness = 0;
```

```
// Let the offspring play the games
for (g=0; g<30; g++)
{
    Games[g].LoadFromFile(g);
    OffspringsFitness
        += Games[g].Play(OffspringsData);
};

// Find the minimum fitness among the current
// population
int MinFitness = 0;
for (int f=0; f<100; f++)
    if (Fitness[f]<Fitness[MinFitness])
        MinFitness = f;

// If the minimum fitness is lower than the fitness
// of the offspring, replace the member of
// the population with the offspring
if (OffspringsFitness > Fitness[MinFitness])
{
    Population[MinFitness] = OffspringsData;
    Fitness[MinFitness] = OffspringsFitness;
};
};

// Find the set with the best fitness and return it
int MaxFitness = 0;
for (int s=0; s<100; s++)
{
    if (Fitness[s]>Fitness[MaxFitness]) max = s;
};
return(Population[MaxFitness]);
};
```