Machine learning - neuroevolution for designing chip circuits/pathfinding

JONATHAN RINNARV, PONTUS BRINK

Master in Computer Science
Date: June 3, 2017
Supervisor: Alezander Kozlov
Examiner: Örjan Ekeberg
Swedish title: Maskininlärning - nevroevolution for att designa kretskort/stigfinnare
School of Computer Science and Communication

Abstract

Neural Networks have been applied in numeral broad categories of work. Such as classification, data processing, robotics, systemcontrol e.t.c. This thesis compares using traditional methods of the routing process in chip circuit design to using a Neural Network trained with evolution.

Constructing and evaluating a chip design is a complicated thing, where a lot of variables have to be accounted for and therefore a simplified evaluation and design process is used in order to train the network and compare the results.

This was done by constructing simple test cases and running the algorithms BFS, A*Star and the neural network and comparing the paths each algorithm found using a fitness function. The results were that BFS and A*Star both performed better on complex circuits, but the neural network was able to create better paths on very small and niche circuits.

The conclusion of the study is that the neural network approach is not able to compete with the standard industry methods of the routing process, but we do not exclude the possibility that with a better designed Fitness function, this could be possible.

Sammanfattning

Neurala Nätverk används i flertal breda kategorier av arbete. Såsom klassificering, databehandling, robotik, systemkontroll e.t.c. Denna avhandling jämför traditionella metoder för routingprocessen i chip-kretsdesign med att använda ett neuralt nätverk utbildat med evolution.

Att konstruera och utvärdera en chipdesign är en komplicerad sak, där många variabler måste tas hänsyn till och därför används en förenklad utvärderings- och designprocess för att träna nätverket och jämföra resultaten.

Detta gjordes genom att konstruera enkla testfall och köra algoritmerna BFS, A * Star och det neurala nätverket och jämföra de sökvägar som varje algoritm fann med hjälp av en så kallad Fitness-funktion. Resultaten var att BFS och A * Star både fungerade bättre på komplexa kretsar, men det neurala nätverket kunde skapa bättre vägar på mycket små och nischade kretsar.

Slutsatsen av studien är att det neurala nätverkssättet inte kan konkurrera med routingprocessens standardindustrimetoder, men vi utesluter inte möjligheten att med en bättre utformad Fitness-funktion skulle detta vara möjligt.

Contents

Chapter 1

Introduction

A chip on a circuit board is a finely tuned and precise circuit map. Designing these chips depends on thousands of variables and would take a lot of computing power if you were to brute force every solution. The routing process of designing a chip is very precise, and must satisfy certain rules defined by chip foundries. The most important part of the routing process is to complete the required connections on the chip. After this, reducing wirelength and ensuring that the timing of each connection meets the conditions of the chip foundry rules. The routing process also has to take in parameters such as resistance, capacitance, wire width and spacing of each layer into account [5]. Modern circuits have reached such complexity that the use of computer tools are mandatory in design and evaluation of these circuits [1]. Finding the best path here isn't always the shortest possible one. In some situations, a path that is longer with less turns is favorable to the shortest possible one and sometimes a route may be required to take a longer path in order to satisfy some signal timing constraint. The environment has a large impact on which path is the most optimal one [13]. In critical sections of the circuit, some manufacturers even recommend manual routing [9].Therefore, traditional pathfinding algorithms that find the shortest path aren't guaranteed to give the best solution.

Using Neural Networks, a pathfinding algorithm that behaves and reacts differently according to the environment can be constructed. One could create a fitness function that generates a behavior required in one environment, which could generate a behaviour that is optimal for that situation. In order to do this, intensive knowledge about chip circuits is required, and the ability to score a design with a numerical value. With chip circuits being so complex, this is beyond the scope of this project and a simplified version of reality is used.

1.1 Problem statement

Is it possible to use a neural network to handle the routing part of designing chip circuits that are on-par with common industry methods?

We will seek to answer this question by comparing the neural network to algorithms commonly used as a basis for industrial solutions.

1.2 Scope

General pathfinding functions such as A*star and BFS are covered in this paper. As well as a comparison between these algorithms and a Neural Network trained for pathfinding.

Constructing and evaluating a chip design is a complicated thing, where a lot of variables have to be accounted for and therefore a simplified evaluation and design process is used in order to train the network and compare the results. We consider the chip circuit to be a grid where two tiles are to be interconnected. There can also be tiles that the path can not be routed through. simplifications are described in detail in section 3.1.

1.3 Purpose

The purpose of this study is to compare the results of industry standard algorithms for the routing process in chip circuit design to what a trained neural network can generate. This is interesting because no study was found trying solve this problem with a neural network before.

Chapter 2

Background

2.1 Neural Networks

A neural network is a mathematical model of a brain and can be described as a layer of states. The first layer is the input layer for the given problem. This layer feed its information forward to the next layer called hidden layer. The information progress through all hidden layers until it reaches the last layer, the output layer, which is the predicted solution for the given input. How the information flow through the network is not meant to be designed by an engineer, instead the network gets trained through rewarding it when it gets a prediction right and punishing it when it gets it wrong. Depending the amount of hidden layers a neural network can make more abstractions of the information and learn to analyze more complex tasks.

A popular method in training a network is supervised learning, that is when the network is fed a solution or a part of a solution and the input for that solution and update the network's internal state using the backpropagation algorithm. The algorithm calculates how far off the network's prediction was from the expected value and backpropagates the error backwards through the network and adjust the weights between the layers to minimise the error [12][2].

Neural networks is not a new invention and have been around for a very long time [14]. Due to increase in computing capability of modern computers neural networks has been able to solve a wide range of complex tasks such as character recognition and stock market prediction [15].

2.2 Genetic Algorithms

Genetic algorithms are evolution inspired algorithms used to find an optimal solution to a problem. Often very expensive in form computing power but they have been found to solve difficult problems like Database Query Optimization [3] and optimizing wind turbine farms [11].

The idea behind genetic algorithms is that you start with simple creatures trying to solve the problem at hand. You let the best performing creatures reproduce and kill the rest. You allow creatures to mutate and compete for survival. Given enough generations you hopefully have a creature able to solve the problem efficiently.

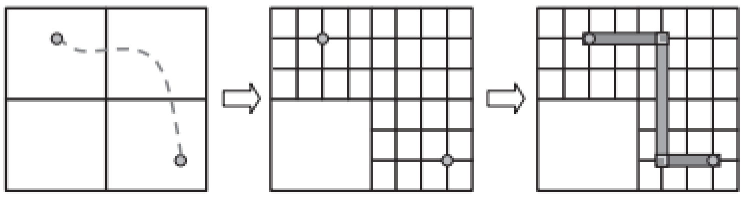
2.3 Neuroevolution of Augmented topologies

One of the main problems with genetic algorithms is when a creature innovates and make a drastic change it will most likely perform worse than its parents did initially because it needs time to optimize and perfect its innovative niche. This often leads to the death of the creature and a potential solution is lost.

In 2002 Kenneth O. Stanley and Risto Miikkulainen released a paper tackling this very issue, they call it Neuroevolution of Augmented topologies, or NEAT for short [17]. They argued that dividing the population of creatures into species and thereby letting creatures compete within their own niche will preserve innovation and will result in faster learning. This way they were able to create neural networks faster they other genetic algorithms [18]. They also developed a new way of creature crossover (mating) by tracking genes to their historical origin. Genes that two creatures share are always passed down to offspring and genes that differs are passed down from the fittest parent. This way children has lower chance of losing expertise which their parent obtained [19]. This led to NEAT being able to start with a minimal neural network (only input and output) and not a random array of starting networks as other neuroevolution systems requires. This makes NEAT have a bias towards minimalistic solutions which is important to be able to keep the search space to a minimum to increase performance [20].

2.4 Routing process

Routing is an important step of the integrated chip design process. It generates the wiring that connect pins of the same signal while obeying the manufacturing design rules [5]. With the technology advancing, the number of transistors on a single chip can reach into the billions, this challenges physical design and therefore the routing process. The routing process is split into global routing and detailed routing. In the global routing stage a routing region is constructed for each net on the chip and each net is given a tentative route through the set. The global routing part does not handle drawing how each wire connects, that is handled in the detailed routing process [8]. The global routing process essentially creates a grid with interconnected points that the detailed routing process then defines more clearly using channels and layers and also taking into account all the various parameters that affect the circuit such as interference (or crosstalk), density, heat and damage that can cause discharges [5]. The following pictures illustrates the global routing process (picture a and b) and the detailed routing process (picture c) with a simple example

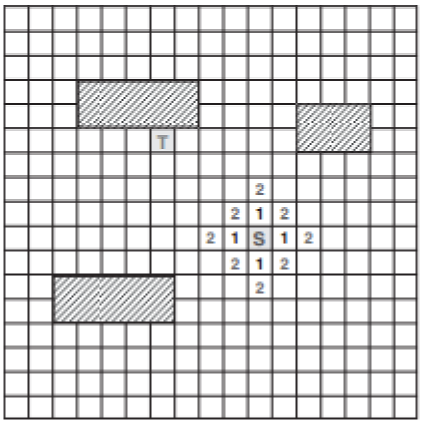


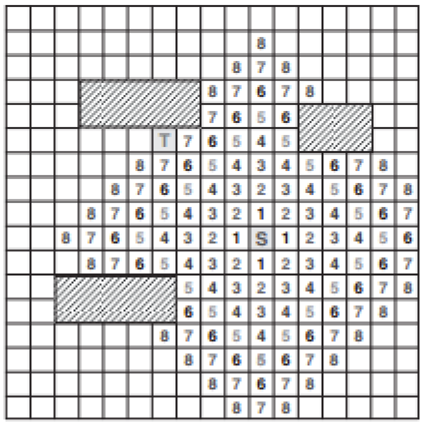
In picture A, regions that the net will use are defined. In picture B, the tentative route is given by the generated grid, the wire is allowed to be drawn through these nodes. In picture C, the detailed routing process has found a path through the grid.

Both the global and detailed routing process use general pathfinding algorithms so the following sections describe two popular algorithms for pathfinding used in both these processes.

2.5 Breadth First Search

Breadth first search (BFS) is a general pathfinding algorithm. It is suitable to use a BFS when you have a unweighted graph with one source node that needs to go through a number of destinations or a number of source nodes that all have the same destination [6]. These types of problems appear in both the detailed and global routing processes, making BFS a useful algorithm for pathfinding [5]. This algorithm was invented in 1950 by E.F Moore, but later in 1961 C.Y. Lee discovered it independently and used it as a routing algorithm [4][10]. How BFS works is that expands outward from the starting node. It checks each neighbour node of the visited node until it finds its destination. Each time it checks a neighbour node, it increments a value on that node in order to find the shortest path to it from the starting node. For multiple destinations, the algorithm would continue from the starting node until all nodes has been reached. For multiple starting nodes, the algorithm would start from the destination instead and work the same way. The following pictures illustrates how a BFS algorithm works, where S is the starting node, T is the destination node and shaded areas are blocked and can not be routed through.





BFS guarantees to find a path between the starting point and the destination given that such a path exist, and it also guarantees that the found path is a shortest path. It is however slow and memory consuming, its time complexity is O(mn) where m and n is the height and width of the grid.

2.6 A*Star

A*star is a heuristic pathfinding algorithm, meaning that it uses a evaluation function to determine which node to visit next. A*star is best fitted to use when you only have one starting node and one destination node. The function used is f(x) = g(x) + b(x) where g(x) is the cost of visiting the next node, and b(x) is the estimated cost from that node to the destination. [6]. An interesting note is that BFS is a special example of A*star where b(x) = 0 for all x. Also, if b(x) has the never overestimates the cost from the current node to the destination, A*star would be optimal. If you only allow horizontal and vertical movements (Manhattan routing), b(x) could be set as the Manhattan distance to the destination, since it is the smallest distance from the current node to the destination b(x) would never overestimate the cost. A*star algorithms proposed in 1984 by and 1995 are used in modern routers. [5]. Since A*star use a heuristic function, its complexity depends on the used function. If one would choose that b(x) = 0 for all x, A*star would be the same as the BFS algorithm and share its complexity. In the worst case scenario, when the search space is unbounded, its complexity would be O(db) where d is the shortest path to the destination and b is the average number of childs from each visited node [16]. Like BFS, A*star guarantees that if there exists a path from the starting node to the destination then A*star will find one. A*star does not however guarantee that it finds a shortest path, it depends on the quality of the used heuristic. A*star only guarantees that the found path is a shortest one if b(x) is less or equal to the actual cost, the smaller the b(x) however, the longer time A*star will take [7]. Below is a picture illustrating A*star with a manhattan heuristic. The red tile is the starting node and the blue tile is the destination. The colored nodes give the b(x) value ranking from yellow to blue depending on the estimated cost where yellow is the largest cost and blue is the lowest.

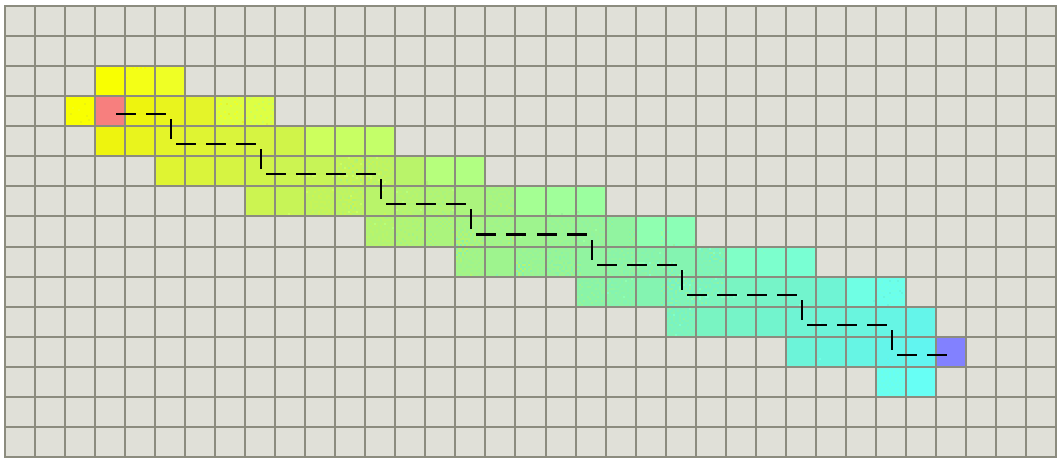


Figure 2.1: A*Star pathing example

Chapter 3

Method

This chapter describes how the Neural Network was constructed, how A*star and BFS was implemented and how the results were generated.

To generate a neural network that can solve the problem at hand supervised learning is not applicable because that required knowledge of an optimal circuit. Therefore a genetic algorithm was chosen to generate the network. NEAT was chosen as the genetic algorithm because it has been proven to be one of the most efficient neuroevolution algorithm.

3.1 Environment

Because of time constraints in this course and the fact that circuits are extremely complex, restrictions have to be set to the routing environment. First we assume that chip circuits are only two dimensional and the parameters for evaluating a circuit are only wire length and amount of wire turns.

The simplified circuit are represented in the form of a two dimensional bit matrix where the value of 1 represent a unpathable tile and the value of 0 represent pathable tile. This matrix will change during the wiring process, laying down a wire will switch a tile from pathable to unpathable.

3.2 Neural network setup

The input layer starts with the matrix described in section 3.1. As well as the position of each wire and the corresponding destination. For a circuit with size 10x10 with two wires to route the amount of inputs would be 10x10+4x2=108. The output layer consists of four actions for each wire, move east, move north, move south and move west. For the example above the amount of outputs would be 4x2=8.

The network is evaluated through the fitness function fitness = amountOfCompletedWires*sizeOfGrid - wireLength*2 - wireTurns - (sizeOfGrid if no wires were connected).

amountOfCompletedWires is the number of correctly connected wires.
sizeOfGrid is the size of the grid, in the case above 10x10=100.
wireTurns is the amount of turns in total all wires has made.
wireLength is the amount of wire used in total between all wires.

The higher fitness value the better the network has performed. The fitness function was chosen experimentally.

The output from the network is an array of prefered moves. Each wire has four directions it can move. The neural network outputs an array with these moves and the highest rated move is chosen. If this move is illegal the next on the list is chosen. If all moves are exhausted the evaluation of the network is done.

3.3 NEAT parameters

Mutation rate on weights is set to 0.8. A low mutation rate result in a stable network performing close to the networks parents. A high value results in a highly diverse network who can outperform its parents with high margin but also risk losing what made the parents successful.

Add new node rate is set to 0.2. A high rate can easily result in redundant nodes with no function. This rate was set low to strive after the minimal solution. This is important to keep the networks as small as possible to save time on evaluating them.

Add new connection rate is set to 0.5. A high rate can result in redundant connections between nodes but a low rate can result in new nodes never being used or experimented on.

The initial connection on a new network is set to empty with no connections. This is to save computing time and stay biased towards a minimal solution.

These parameters was chosen based on the NEAT paper [20]

3.4 A*Star and BFS

The A*star was implemented using the pseudo code in the appendixes. BFS was implemented with A*star using a heuristic that always return zero. When the input uses multiple start and end node, every permutation of the wire order is generated, each solution is generated and the one that creates the best fitness is chosen. In order for these algorithms to create the optimal solution, each shortest route must also be explored with each wire order. This is not something A*Star is able to do whilst remaining more efficient than BFS. Therefore, because of this and the time it would to take to test every solution on complicated inputs, only wire order permutation were implemented.

3.5 Evaluation

The algorithms were compared on a range of different circuits. Starting with small and easy circuits and working towards harder ones in order evaluate how the algorithms scale with complexity. Some tests were chosen in order to demonstrate the strengths and weaknesses of the algorithms.

Chapter 4

Result

This chapter presents the result of the testing of the algorithms. In each test the goal is to connect each unique pair of a color without pathing through another wire (represented with another color) or a wall (represented with black). In each graph the generation number is on the horizontal axis and fitness on the vertical axis. The red line represents the best fitness achieved in each generation and the blue line represents the average fitness of the generation. The green lines represents the standard deviation from the average in each generation.

4.1 Test 1

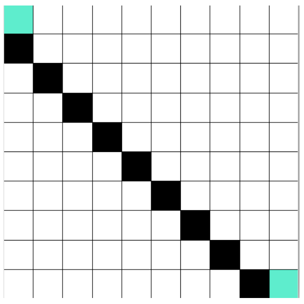


Figure 4.1: Base

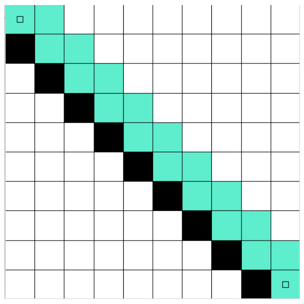


Figure 4.2: A*Star

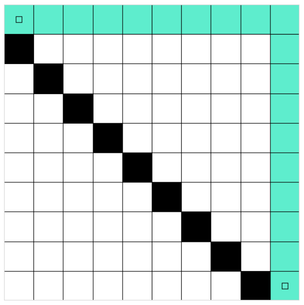


Figure 4.3: NEAT

Figure 4.4: Test 1 - NEAT generated 0 hidden nodes

This is the first test we used to compare the algorithms. Both BFS and A*star resulted in the same path with a fitness of 43. NEAT achieved a fitness of 77. This is a 79% better fitness than A*star and BFS.

4.2 Test 2

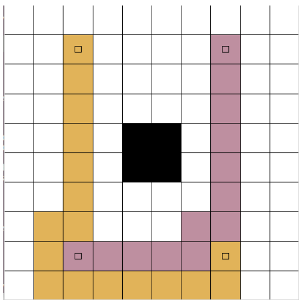


Figure 4.5: A*

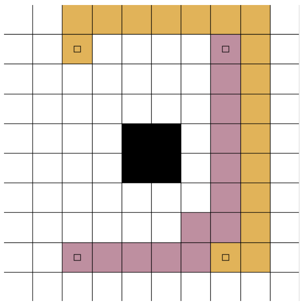


Figure 4.6: BFS

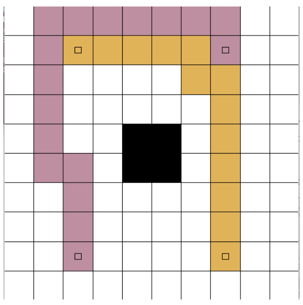


Figure 4.7: NEAT

Figure 4.8: Test 2 - NEAT generated 0 hidden nodes

This is the second test used to compare the algorithms. BFS, A*star and NEAT all gave different results, with A*star achieving a fitness of 142, BFS a fitness of 154 and NEAT a fitness of 142. NEAT its best solution after only five generations, but it was run for 150 generations.

4.3 Test 3

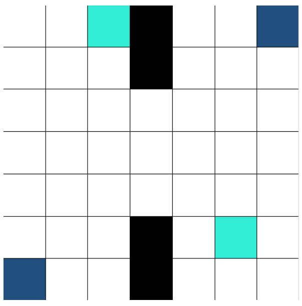


Figure 4.9: Base

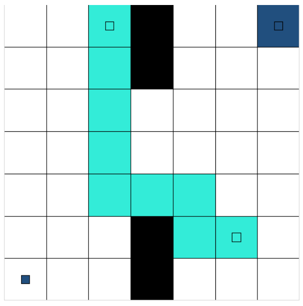


Figure 4.10: A*Star

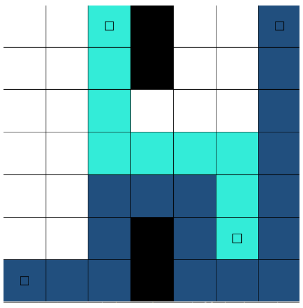


Figure 4.11: NEAT after one generation

Figure 4.12: Test 3 - NEAT generated 0 hidden nodes

In this test A*Star failed to complete the circuit but NEAT was able to correctly connect the wires.

4.4 Test 4

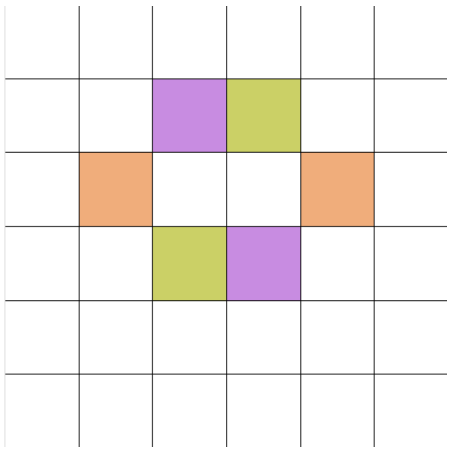


Figure 4.13: Base

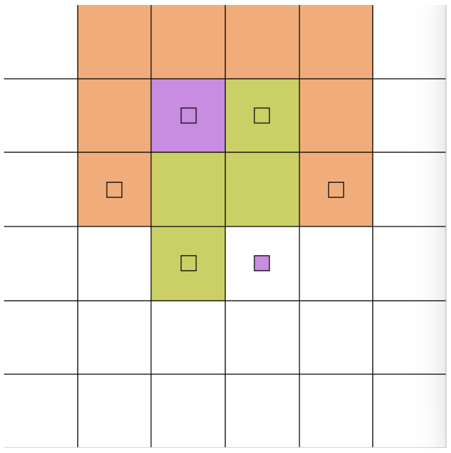


Figure 4.14: A*, BFS and NEAT

Figure 4.15: Test 4 - NEAT generated 0 hidden nodes

The fourth test we used is impossible to complete all the wires. In this test, every algorithm constructed the same paths and was able to find one of the best solutions with a fitness of 176.

4.5 Test 5

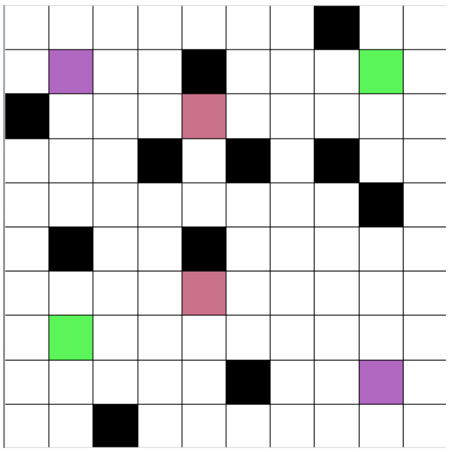


Figure 4.16: Base

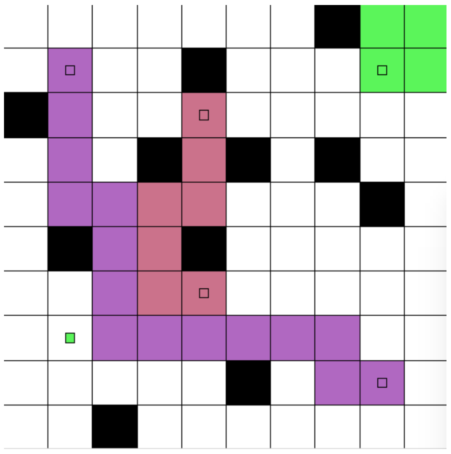


Figure 4.17: NEAT after 173 generation

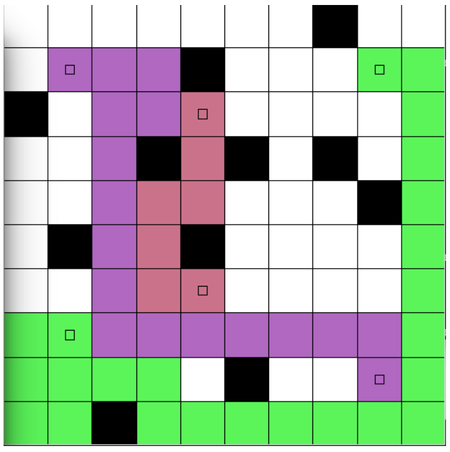


Figure 4.18: NEAT after 578 generations

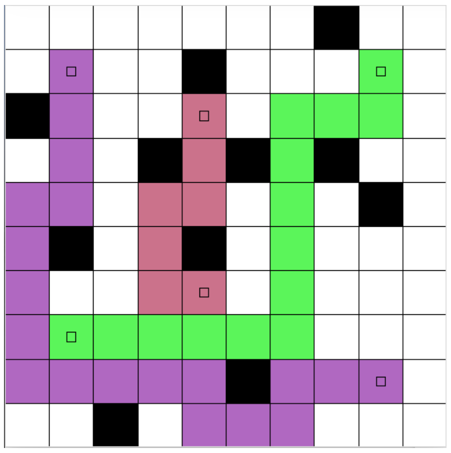


Figure 4.19: A*

Figure 4.20: Test 5 - NEAT generated 5 hidden nodes

NEAT after 173 generation acquired a fitness score of 148. NEAT after 578 generation acquired a fitness score of 185. A*Star scored a fitness of 204.

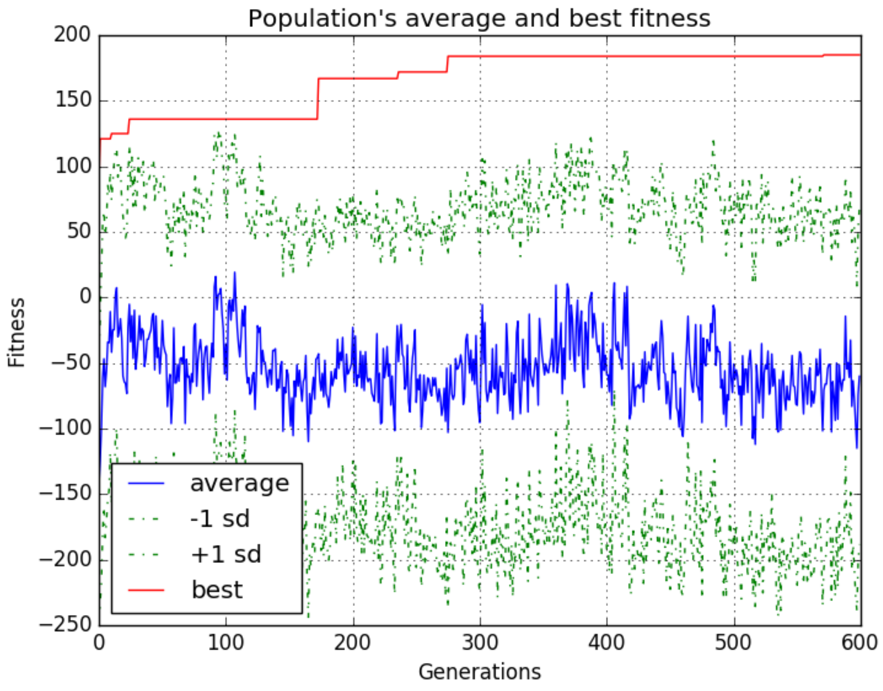


Figure 4.21: Test 5 - NEAT Fitness graph

The best fitness was found at generation 578. The best fit curve steadily increases early and begins to converge at 270 generations while the average does not increase during the session.

4.6 Test 6

This is a test with a grid of size 35x35 with five wires to path. NEAT had 1245 inputs and 20 outputs.

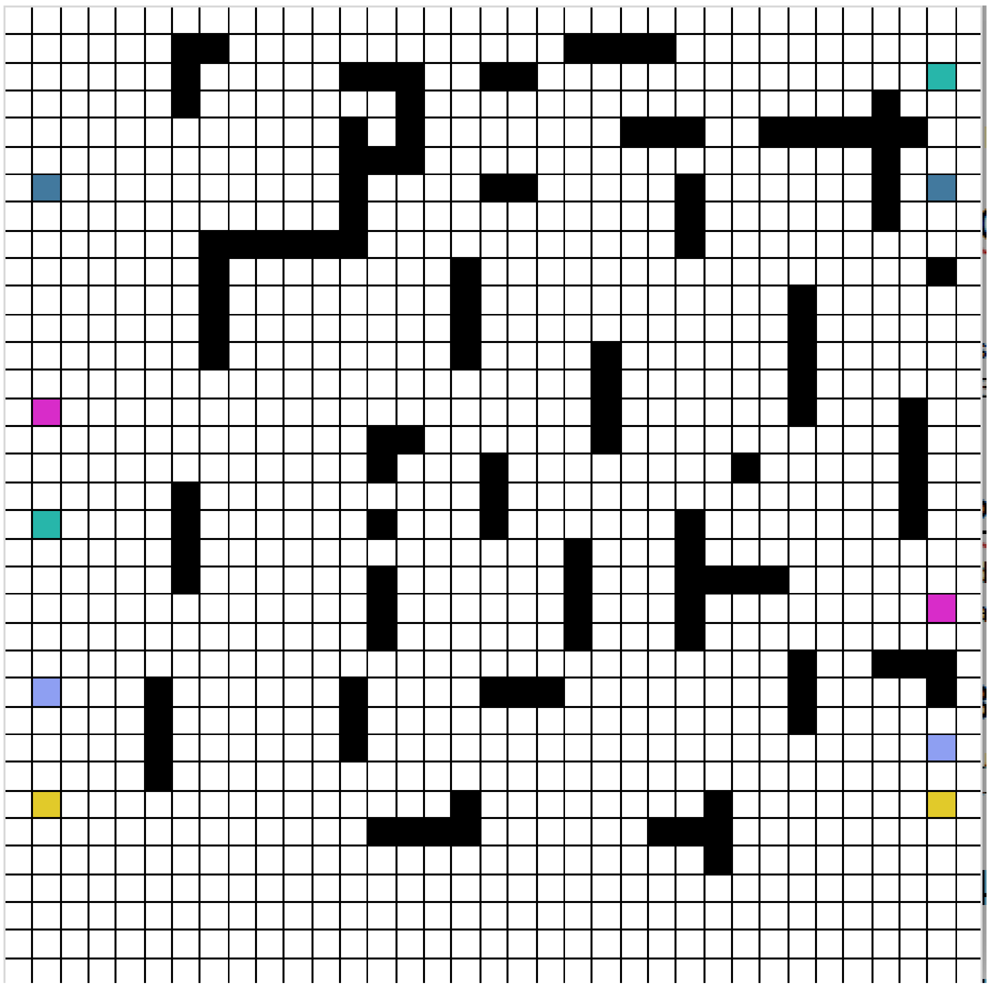


Figure 4.22: Test 6 - Base. Grid of size 35x35 with 5 wires

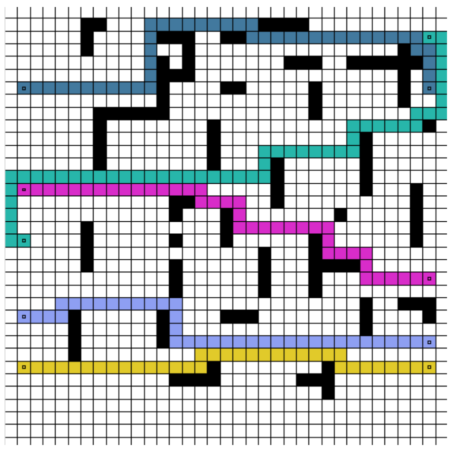


Figure 4.23: A*

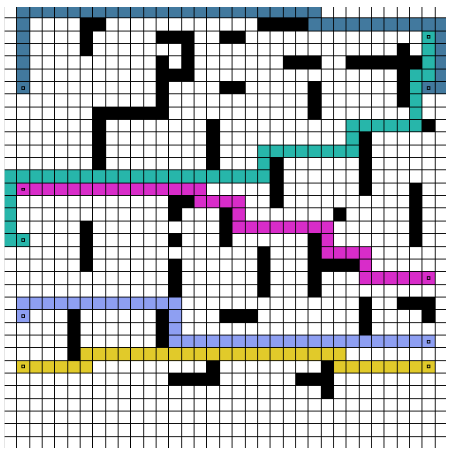


Figure 4.24: BFS

Figure 4.25: Test 6 - A* and BFS

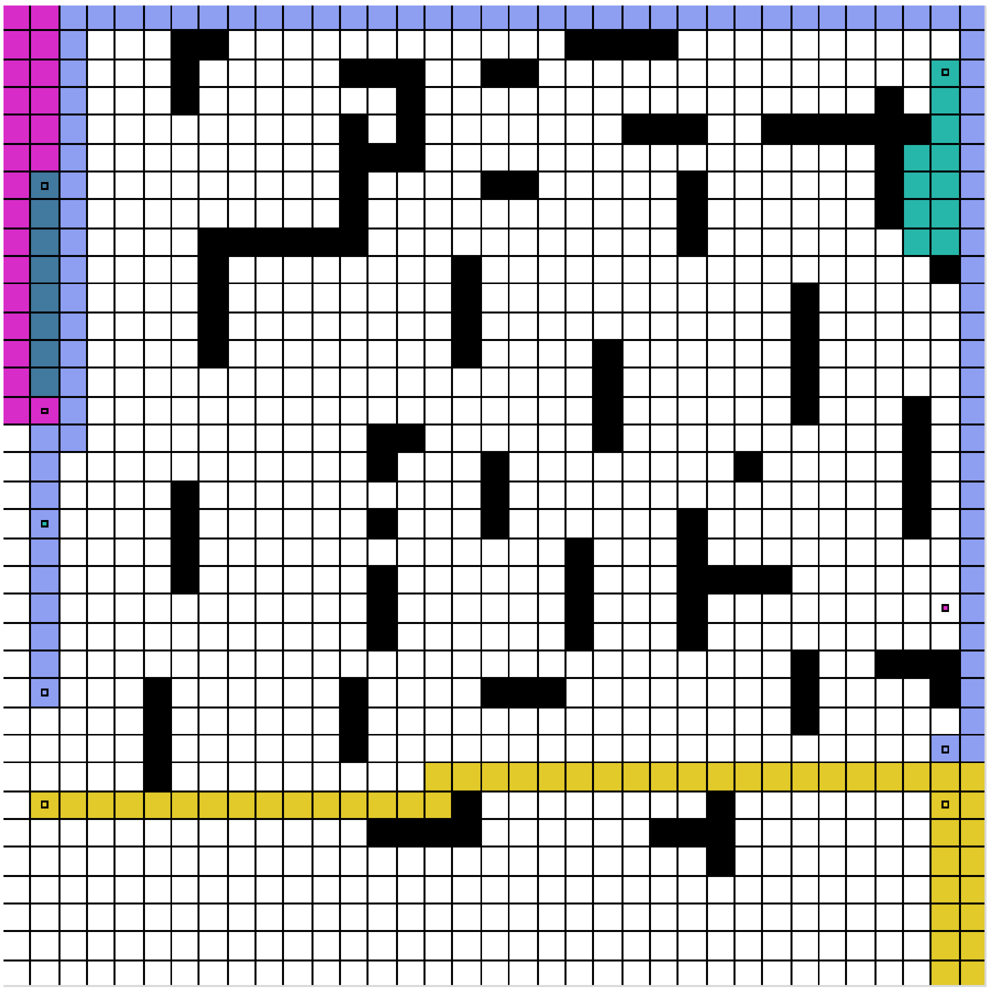


Figure 4.26: Test 6 - NEAT After 26 generations

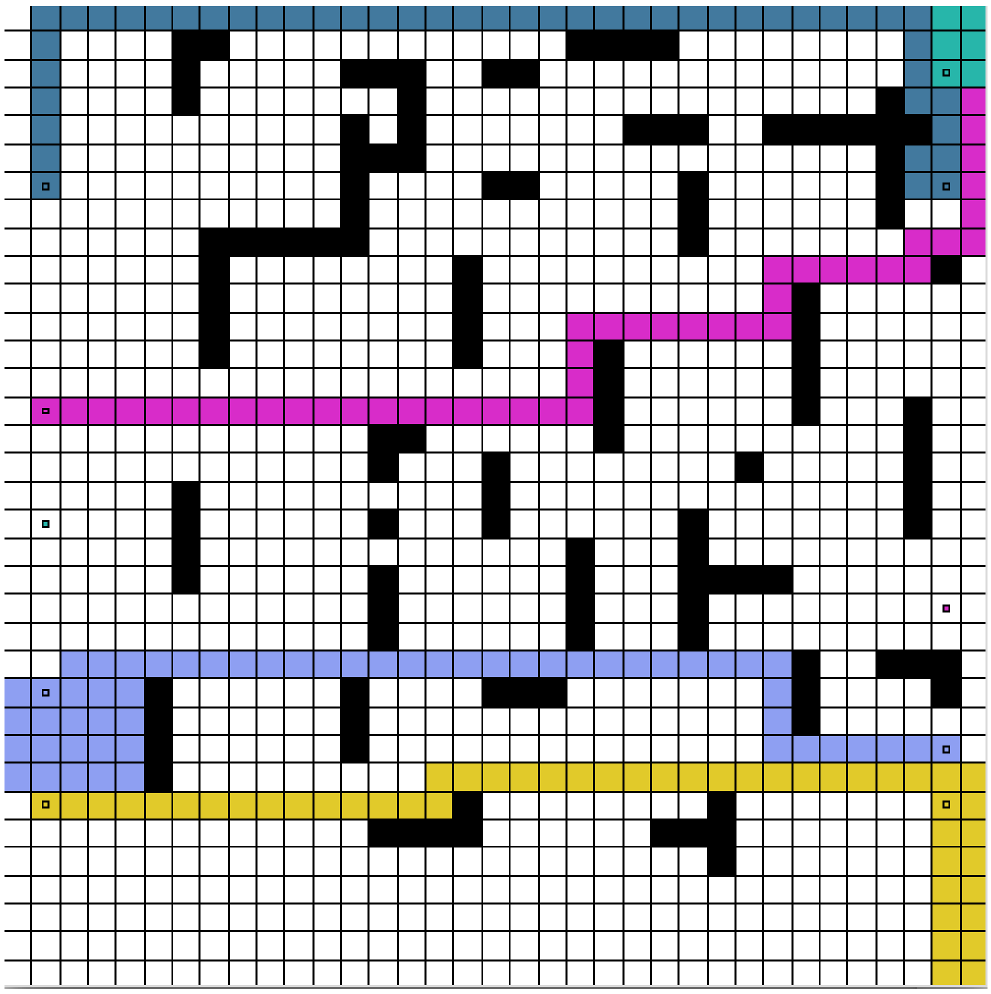


Figure 4.27: Test 6 - NEAT After 300 generations
NEAT 300 generations Fitness: 8, NEAT 150 generations Fitness: -72, A*star fitness 296, BFS Fitness 296. A*star and BFS got a fitness 3700% higher than the best NEAT neural net could generate within its session.

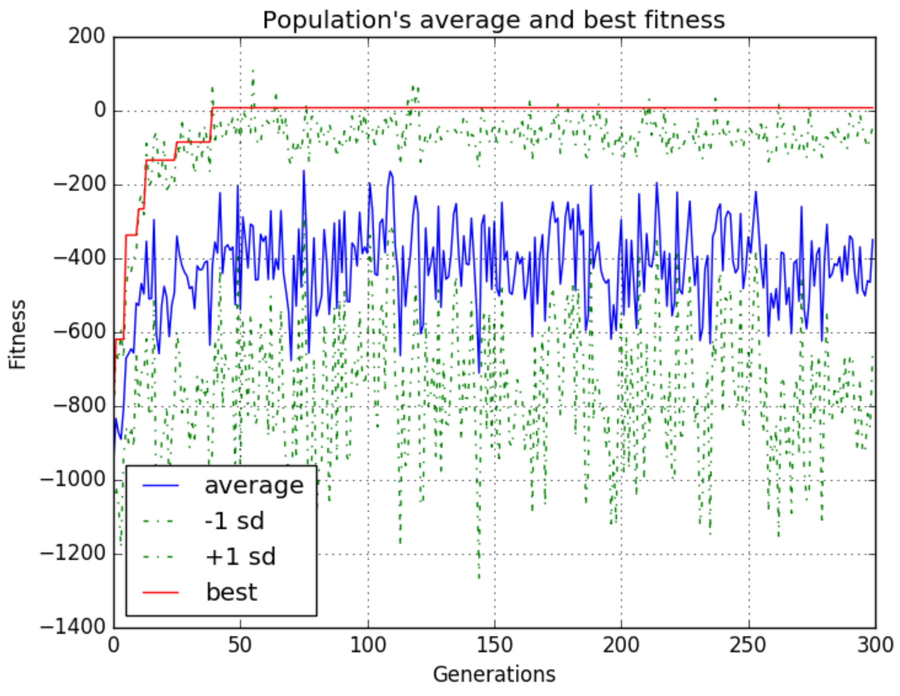


Figure 4.28: Test 6 - NEAT Fitness graph
The best solution was found in generation 31. A step climb in best fitness early along with average fitness. But 250 generations later still no improvement.

4.7 Summary

	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6
A*Star	43	142	31	204	296	176
BFS	43	154	31	204	296	176
NEAT	77	142	50	185	8	176

Above is a table that summarises the results of the tests. The NEAT row is from the solution that generated the most fitness in each test. The value in the cells are the fitness from each test that we algorithm got.

Chapter 5

Discussion

The first test (Figure: 4.4) was the only test that resulted in NEAT vastly outperforming both BFS and A*star. NEAT is able to take into account amount of turns, where A*star used manhattan distance as a heuristic which makes a vertical step and a horizontal step equal in distance. The shortest path generated by BFS and A*star created a stair-shape which resulted in a lot of turns, getting less fitness, even tho the path is equally long.

In the second test (Figure: 4.8) all algorithms got very similar fitness even though they have distinct solutions. The NEAT solution was found very quickly, but was unable to correct its error of having a unnecessary turn within the session. This could be a result of a poorly designed fitness function that does not punish turns enough.

In the third test (Figure: 4.12) A*Star was not able to solve the circuit. This is because regardless of which order of wires is chosen the first wire will block the other wires path and thereby fail to complete the circuit correctly. NEAT however can with ease path around this problem.

In the fourth test (Figure: 4.15) all algorithms produced the same, optimal, result. This is interesting because there are multiple optimal solutions to this problem. BFS and A*Star both goes through all possible orderings of wires and uses the one that generated the best fitness, it is pure chance that NEAT created the exact same solution.

In the fifth test (Figure: 4.20), NEAT was not able to perform at A*star and BFS level. For many generations, NEAT refused the path the green wire to its destination. Instead it insisted of shortening the wire length. This is unwanted behaviour that could be a result of a poorly designed fitness function. The extra bits of wire seen in figure 10 on the green and purple wires could be a result of NEAT sets up a priority order to move and without further analysis of the input these bits are required.

In the sixth test (Figure: 4.22 - 4.28), NEAT was not able to compete with A*star or BFS. The size of the grid in this test was larger than previous test and wires had long way to path. Since pathing is punished by the fitness function NEAT opted to minimize wire length instead. This resulted in NEAT having a hard time solving the problem.

5.1 Pathing choice

Since the neural network do not have the option of doing no move at all a wire the can move always will. Because wire length is punished by our fitness function NEAT seem to prefer to minimize wire length by moving itself into a corner to cut out possible moves as can be seen in figure 4.20. Figure 4.27 it seems like another wire cuts of another to minimize wire length. This could be the result of a poorly written fitness function.

The low number of hidden inputs generated via NEAT indicates that not much analysis of the input are being made. Input are usually connected directly to output with no in between nodes. Since the highest rated action is not always chosen (in the case that it is illegal) so the network wish to go one way but it goes another, if that happens to be the right way it gets rewarded for attempting to go the wrong way. This indicates that NEAT creates an internal priority list of moves for each wire. A priority list is enough to wire small grids but more analysis is required for larger grids.

5.2 Fitness function

In the previous section it is described briefly that the behavior where wires collide and kill each other could be a result of a poorly designed fitness function. The behavior could be because a wire is only rewarded once it reaches its destination. Adding so that wires also get rewarded based on its distance to the destination could remove this behavior, but also lead to other unwanted behaviors. Such as wires going in straight lines until they get close to their destination before killing themselves. Improving the fitness function is something future research could look upon.

5.3 Positive result

Pathfinding is not the only merit in designing circuits. There are a lot of other variables involved in the process of creating an efficient circuit such as temperature distribution. In the scope of this rapport a simple model of reality is used to focus on only pathfinding. If NEAT would perform better on complex circuits and prove that it is an efficient algorithm for pathfinding other variables could be introduced to the fitness function to create optimal circuits. However since NEAT has problem solving the base problem, which is pathfinding, this algorithm is not suited for further implementations. The competitive edge NEAT has over BFS and A*Star is the external factors it can take into account.

Chapter 6

Conclusion

The conclusion of the study is that the neural network approach is not able to compete with the standard industry methods of the routing process in the simplified model used in this study. The result could be due to the fitness function, which is something future research should study. Improving the fitness function could lead to result similar to that of BFS and A*Star in the simplified model, and even better result if a more realistic model was used due to the fact that a neural network could take more parameters into consideration.

Bibliography

[1] John E. Ayers. "Digital Integrated Circuits". In: CRC PRESS, 2005, pp. 17–19.

[2] Lluís A. Belanche. *Some applications of MLPs trained with backpropagation*. http://www.cs.upc.edu/~belanche/Docencia/apren/2009-10/Excursiones/Some\%20Applications\%20of\%20Bprop.pdf. Accessed:2017-03-18. 2010/11.

[3] Kristin Bennett, Michael C Ferris, and Yannis E Ioannidis. *A genetic algorithm for database query optimization*. Computer Sciences Department, University of Wisconsin, Center for Parallel Optimization, 1991.

[4] Tao B. Schardl Charles E. Leiserson. *A Work-Efficient Parallel Breadth-First Search Algorithm (or How to Cope with the Nondeterminism of Reducers)*. MIT Computer Science and Artificial Intelligence Laboratory, 2010.

[5] Huang-Yu Chen and Yao-Wen Chang. "Global and detailed routing". In: chap. 12, pp. 687–740.

[6] Red Blog Games. *Global routing*. http://www.redblobgames.com/pathfinding/tower-defense/. Accessed:2017-03-20. 2016.

[7] Red Blog Games. *Heuristics*. http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html. Accessed:2017-03-19. 2016.

[8] Indranil Sen Gupta. *Global routing*. http://www.facweb.iitkgp.ernet.in/~isg/CAD/SLIDES/11-global-routing.pdf. Accessed:2017-03-19.

[9] National Instruments. *Best practices in PCB Design - National Instruments*. http://www.ni.com/tutorial/6880/en/\#toc8. Accessed:2017-03-20. 2017.

[10] C. Y. Lee. "An Algorithm for Path Connections and Its Applications". In: *IRE Transactions on Electronic Computers* EC-10.3 (1961), pp. 346–365. ISSN: 0367-9950. DOI: `10.1109/TEC.1961.5219222`.

[11] GPCDB Mosetti, Carlo Poloni, and B Diviacco. "Optimization of wind turbine positioning in large windfarms by means of a genetic algorithm". In: *Journal of Wind Engineering and Industrial Aerodynamics* 51.1 (1994), pp. 105–116.

[12] David C. Plaut and Geoffrey E. Hinton. *Learning sets of filters using back-propagation*. `http://www.cs.toronto.edu/~fritz/absps/plautfilters.pdf`. Accessed:2017-03-18. 1987.

[13] LEE W. RITCHEY. *PCBROUTERS*. `http://www.speedingedge.com/PDF-Files/pcbrouters.pdf`. Accessed:2017-03-20. 1999.

[14] Eric Roberts. *History: The 1940's to the 1970's*. `https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/History/history1.html`. Accessed:2017-03-18.

[15] Eric Roberts. *History: The 1940's to the 1970's*. `https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/Applications/index.html`. Accessed:2017-03-18.

[16] Stuart Jonathan Russell and Peter Norvig. *Artificial intelligence: a modern approach (3rd edition)*. pages 97-104. 2009.

[17] Kenneth O. Stanley and Risto Miikkulainen. *evolving neural networks through augmented topologies*. Accessed: 2017-03-21. 2002.

[18] Kenneth O. Stanley and Risto Miikkulainen. *Evolving Neural Networks through Augmenting Topologies*. `http://nn.cs.utexas.edu/downloads/papers/stanley.ec02.pdf`. Accessed: 2017-03-21, pages: 115-117. 2002.

[19] Kenneth O. Stanley and Risto Miikkulainen. *Evolving Neural Networks through Augmenting Topologies*. `http://nn.cs.utexas.edu/downloads/papers/stanley.ec02.pdf`. Accessed: 2017-03-21, pages: 108. 2002.

[20] Kenneth O. Stanley and Risto Miikkulainen. *Evolving Neural Networks through Augmenting Topologies*. `http://nn.cs.utexas.edu/downloads/papers/stanley.ec02.pdf`. Accessed: 2017-03-21, pages: 111. 2002.

.1 A*Star pseudocode

```
function A*(start, goal)
    // The set of nodes already evaluated.
    closedSet := {}
    // The set of currently discovered nodes
    // that are not evaluated yet.
    // Initially, only the start node is known.
    openSet := {start}
    // For each node, which node it can most
    // efficiently be reached from.
    // If a node can be reached from many nodes,
    // cameFrom will eventually contain the
    // most efficient previous step.
    cameFrom := the empty map

    // For each node, the cost of getting from
    // the start node to that node.
    gScore := map with default value of Infinity
    // The cost of going from start to start is zero.
    gScore[start] := 0
    // For each node, the total cost of getting
    // from the start node to the goal
    // by passing by that node. That value is
    // partly known, partly heuristic.
    fScore := map with default value of Infinity
    // For the first node, that value is completely heuristic.
    fScore[start] := heuristic_cost_estimate(start, goal)

    while openSet is not empty
        current := the node in openSet having the
        lowest fScore[] value
        if current = goal
            return reconstruct_path(cameFrom, current)

        openSet.Remove(current)
        closedSet.Add(current)
        for each neighbor of current
            if neighbor in closedSet
```

```
                continue        // Ignore the neighbor which
                                // is already evaluated.
            // The distance from start to a neighbor
            tentative_gScore := gScore[current] +
             dist_between(current, neighbor)
            if neighbor not in openSet    // Discover a new node
                openSet.Add(neighbor)
            else if tentative_gScore >= gScore[neighbor]
                continue        // This is not a better path.

            // This path is the best until now. Record it!
            cameFrom[neighbor] := current
            gScore[neighbor] := tentative_gScore
            fScore[neighbor] := gScore[neighbor] +
             heuristic_cost_estimate(neighbor, goal)

    return failure

function reconstruct_path(cameFrom, current)
    total_path := [current]
    while current in cameFrom.Keys:
        current := cameFrom[current]
        total_path.append(current)
    return total_path
```