

2003

Pathfinding in Computer Games

Ross Graham

Hugh McCabe
hugh.mccabe@tudublin.ie

Stephen Sheridan

Follow this and additional works at: <https://arrow.tudublin.ie/itbj>



Part of the [Computer and Systems Architecture Commons](#)

Recommended Citation

Graham, Ross; McCabe, Hugh; and Sheridan, Stephen (2003) "Pathfinding in Computer Games," *The ITB Journal*: Vol. 4: Iss. 2, Article 6.

doi:10.21427/D7ZQ9J

Available at: <https://arrow.tudublin.ie/itbj/vol4/iss2/6>

This Article is brought to you for free and open access by the Ceased publication at ARROW@TU Dublin. It has been accepted for inclusion in The ITB Journal by an authorized editor of ARROW@TU Dublin. For more information, please contact arrow.admin@tudublin.ie, aisling.coyne@tudublin.ie, vera.kilshaw@tudublin.ie.

Pathfinding in Computer Games

Ross Graham, Hugh McCabe, Stephen Sheridan
School of Informatics & Engineering,
Institute of Technology Blanchardstown

ross.graham@itb.ie, hugh.mccabe@itb.ie, stephen.sheridan@itb.ie

Abstract

One of the greatest challenges in the design of realistic Artificial Intelligence (AI) in computer games is agent movement. Pathfinding strategies are usually employed as the core of any AI movement system. This report will highlight pathfinding algorithms used presently in games and their shortcomings especially when dealing with real-time pathfinding. With the advances being made in other components, such as physics engines, it is AI that is impeding the next generation of computer games. This report will focus on how machine learning techniques such as Artificial Neural Networks and Genetic Algorithms can be used to enhance an agents ability to handle pathfinding in real-time.

1 Introduction

One of the greatest challenges in the design of realistic Artificial Intelligence (AI) in computer games is agent movement. Pathfinding strategies are usually employed as the core of any AI movement system. Pathfinding strategies have the responsibility of finding a path from any coordinate in the game world to another. Systems such as this take in a starting point and a destination; they then find a series of points that together comprise a path to the destination. A games' AI pathfinder usually employs some sort of precomputed data structure to guide the movement. At its simplest, this could be just a list of locations within the game that the agent is allowed move to. Pathfinding inevitably leads to a drain on CPU resources especially if the algorithm wastes valuable time searching for a path that turns out not to exist.

Section 2 will highlight what game maps are and how useful information is extracted from these maps for use in pathfinding. Section 3 will show how pathfinding algorithms use this extracted information to return paths through the map when given a start and a goal position. As the A* pathfinding algorithm is such a major player in the computer games it will be outlined in detail in Section 4. Section 5 will discuss the limitations of current pathfinding techniques particularly with their ability to handle dynamic obstacles. Sections 6 and 7 will introduce the concept of using learning algorithms to learn pathfinding behaviour. The report will then conclude, in Section 8, with how learning algorithms can overcome the limitations of traditional pathfinding.

2 Game World Geometry

Typically the world geometry in a game is stored in a structure called a map. Maps usually contain all the polygons that make up the game environment. In a lot of cases, in order to cut

down the search space of the game world for the pathfinder the games map is broken down and simplified. The pathfinder then uses this simplified representation of the map to determine the best path from the starting point to the desired destination in the map. The most common forms of simplified representations are (1) Navigation Meshes, and (2) Waypoints.

2.1 Navigation Meshes

A navigation mesh is a set of convex polygons that describe the “walkable” surface of a 3D environment [Board & Ducker02]. Algorithms have been developed to abstract the information required to generate Navigation Meshes for any given map. Navigation Meshes generated by such algorithms are composed of convex polygons which when assembled together represent the shape of the map analogous to a floor plan. The polygons in a mesh have to be convex since this guarantees that the AI agent can move in a single straight line from any point in one polygon to the center point of any adjacent polygon [WhiteChristensen02]. Each of the convex polygons can then be used as nodes for a pathfinding algorithm. A navigation mesh path consists of a list of adjacent nodes to travel on. Convexity guarantees that with a valid path the AI agent can simply walk in a straight line from one node to the next on the list. Navigation Meshes are useful when dealing with static worlds, but they are unable to cope with dynamic worlds (or worlds that change).

2.2 Waypoints

The waypoint system for navigation is a collection of nodes (points of visibility) with links between them. Travelling from one waypoint to another is a sub problem with a simple solution. All places reachable from waypoints should be reachable from any waypoint by travelling along one or more other waypoints, thus creating a grid or path that the AI agent can walk on. If an AI agent wants to get from A to B it walks to the closest waypoint seen from position A, then uses a pre-calculated route to walk to the waypoint closest to position B and then tries to find its path from there. Usually the designer manually places these waypoint nodes in a map to get the most efficient representation. This system has the benefit of representing the map with the least amount of nodes for the pathfinder to deal with. Like Navigation Meshes, Waypoints are useful for creating efficient obstacle free pathways through static maps but are unable to deal with dynamic worlds (or worlds that change).

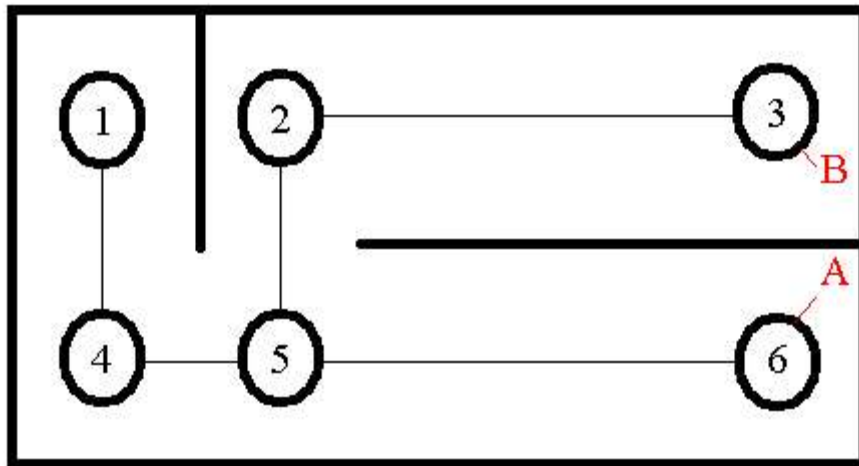


Figure 2.1

Figure 2.1 shows how a simple scene might be represented with waypoints. Table 2.1 shows the routing information contained within each waypoint. The path from (A) to (B) can be executed as follows. A straight-line path from (A) to the nearest waypoint is calculated (P1) then a straight-line path is calculated from (B) to the nearest waypoint (P2). These waypoints are 6 and 3 respectively. Then looking at the linking information, a pathfinding system will find the path as follows { P1, waypoint 6, waypoint 5, waypoint 2, waypoint 3, P2 }

Way Point Number	Link Information
1	4
2	3, 5
3	2
4	1, 5
5	2, 4
6	5

Table 2.1

2.3 Graph Theory

Pathfinding algorithms can be used once the geometry of a game world has been encoded as a map and pre-processed to produce either a Navigation Mesh or a set of Waypoints. Since the polygons in the navigation mesh and the points in the waypoint system are all connected in some way they are like points or nodes in a graph. So all the pathfinding algorithm has to do is transverse the graph until it finds the endpoint it is looking for. Conceptually, a graph G is composed of two sets, and can be written as $G = (V,E)$ where:

- **V – Vertices:** A set of discreet points in n-space, but this usually corresponds to a 3D map.

- **E – Edges:** A set of connections between the vertices, which can be either directed or not

Together with this structural definition, pathfinding algorithms also generally need to know about the properties of these elements. For example, the length, travel-time or general cost of every edge needs to be known. (From this point on cost will refer to the distance between two nodes)

3 Pathfinding

In many game designs AI is about moving agents/bots around in a virtual world. It is of no use to develop complex systems for high-level decision making if an agent cannot find its way around a set of obstacles to implement that decision. On the other hand if an AI agent can understand how to move around the obstacles in the virtual world even simple decision-making structures can look impressive. Thus the pathfinding system has the responsibility of understanding the possibilities for movement within the virtual world. A pathfinder will define a path through a virtual world to solve a given set of constraints. An example of a set of constraints might be to find the shortest path to take an agent from its current position to the target position. Pathfinding systems typically use the pre-processed representations of the virtual world as their search space.

3.1 Approaches to Pathfinding

There are many different approaches to pathfinding and for our purposes it is not necessary to detail each one. Pathfinding can be divided into two main categories, *undirected* and *directed*. The main features of each type will be outlined in the next section.

3.1.1 Undirected

This approach is analogous to a rat in a maze running around blindly trying to find a way out. The rat spends no time planning a way out and puts all its energy into moving around. Thus the rat might never find a way out and so uses most of the time going down dead ends. Thus, a design based completely on this concept would not be useful in creating a believable behaviour for an AI agent. It does however prove useful in getting an agent to move quickly while in the background a more sophisticated algorithm finds a better path.

There are two main undirected approaches that improve efficiency. These are *Breadth-first search* and *Depth-first search* respectively, they are well known search algorithms as detailed for example in [RusselNorvig95]. *Breadth-first search* treats the virtual world as a large connected graph of nodes. It expands all nodes that are connected to the current node and then in turn expands all the nodes connected to these new nodes. Therefore if there is a

path, the *breadth-first* approach will find it. In addition if there are several paths it will return the shallowest solution first. The *depth-first* approach is the opposite of *breadth-first* searching in that it looks at all the children of each node before it looks at the rest, thus creating a linear path to the goal. Only when the search hits a dead end does it go back and expand nodes at shallower levels. For problems that have many solutions the *depth-first* method is usually better as it has a good chance of finding a solution after exploring only a small portion of the search space.

For clarity the two approaches will be explained using a simple map shown in Figure 3.1.

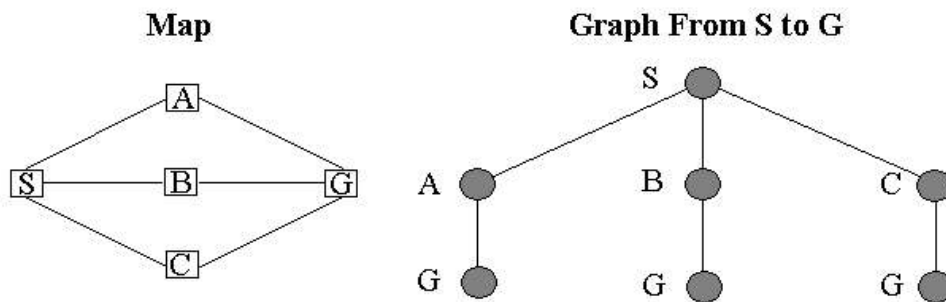


Figure 3.1

Figure 3.1 shows a waypoint representation of a simple map and its corresponding complete search tree from the start (*S*) to the goal (*G*).

Figure 3.2 shows how the two approaches would search the tree to find a path. In this example the *breadth-first* took four iterations while the *depth-first* search finds a path in two. This is because the problem has many solutions, which the *depth-first* approach is best, suited to. The main drawback in these two approaches is that they do not consider the cost of the path but are effective if no cost variables are involved.

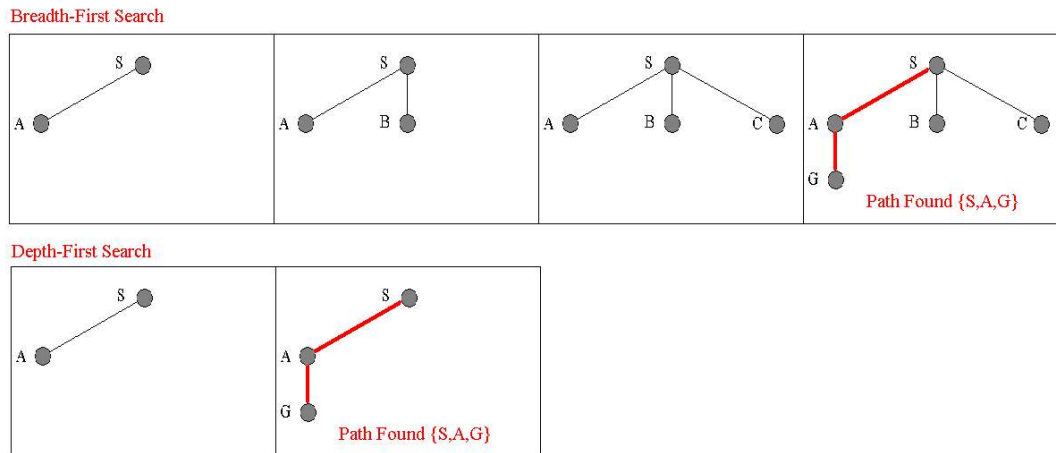


Figure 3.2

3.1.2 Directed

Directed approaches to pathfinding all have one thing in common in that they do not go blindly through the maze. In other words they all have some method of assessing their progress from all the adjacent nodes before picking one of them. This is referred to as assessing the cost of getting to the adjacent node. Typically the cost in game maps is measured by the distance between the nodes. Most of the algorithms used will find a solution to the problem but not always the most efficient solution i.e. the shortest path. The main strategies for directed pathfinding algorithms are:

- *Uniform cost search* $g(n)$ modifies the search to always choose the lowest cost next node. This minimises the cost of the path so far, it is optimal and complete, but can be very inefficient.
- *Heuristic search* $h(n)$ estimates the cost from the next node to the goal. This cuts the search cost considerably but it is neither optimal nor complete.

The two most commonly employed algorithms for directed pathfinding in games use one or more of these strategies. These directed algorithms are known as Dijkstra and A* respectively [RusselNorvig95]. Dijkstra's algorithm uses the *uniform cost strategy* to find the optimal path while the A* algorithm combines both strategies thereby minimizing the total path cost. Thus A* returns an optimal path and is generally much more efficient than Dijkstra making it the backbone behind almost all pathfinding designs in computer games. Since A* is the most commonly used algorithm in the pathfinding arena it will be outlined in more detail later in this report.

The following example in Figure 3.3 compares the effectiveness of Dijkstra with A*. This uses the same map from Figure 3.1 and its corresponding search tree from start (S) to the goal (G). However this time the diagram shows the cost of travelling along a particular path.

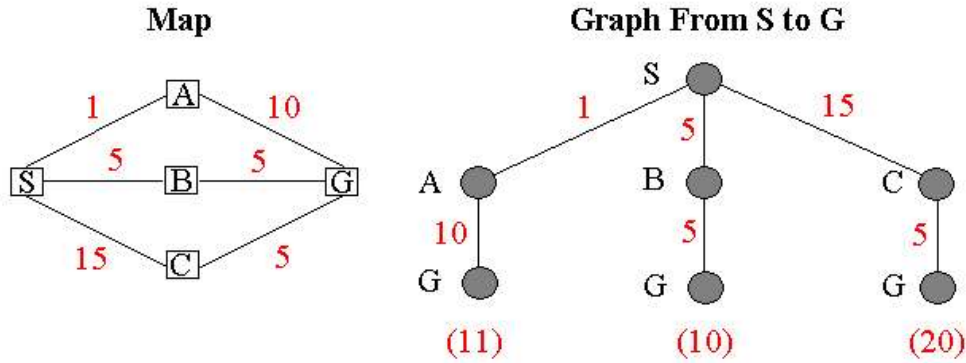


Figure 3.3

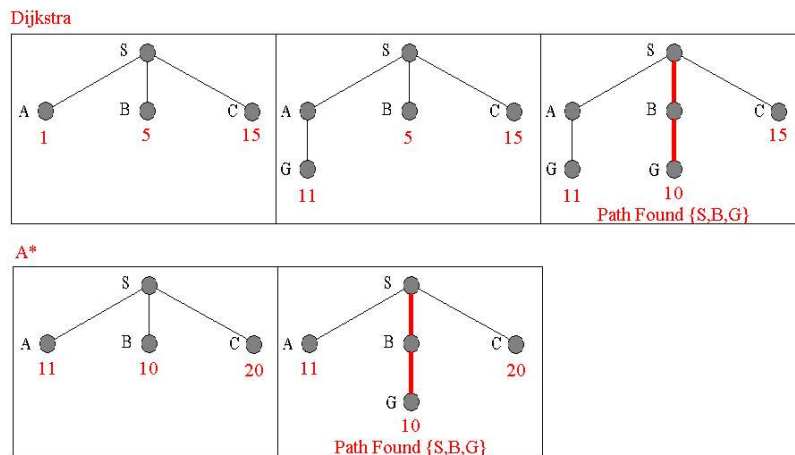


Figure 3.4

Figure 3.4 illustrates how Dijkstra and A* would search the tree to find a path given the costs indicated in Figure 3.3. In this example Dijkstra took three iterations while A* search finds a path in two and finds the shortest path i.e. the optimal solution. Given that the first stage shown in Figure 3.4 for both Dijkstra and A* actually represents three iterations, as each node connected to the start node (S) would take one iteration to expand, the total iterations for Dijkstra and A* are six and five respectively. When compared to the Breadth-first and Depth-first algorithms, which took five and two iterations respectively to find a path, Dijkstra and A* took more iterations but they both returned optimal paths while breadth-first and depth-first did not. In most cases it is desirable to have agents that finds optimal pathways as following sub-optimal pathways may be perceived as a lack of intelligence by a human player.

Many directed pathfinding designs use a feature known as *Quick Paths*. This is an undirected algorithm that gets the agent moving while in the background a more complicated directed pathfinder assesses the optimal path to the destination. Once the optimal path is found a “slice path” is computed which connects the quick path to the full optimal path. Thus creating the illusion that the agent computed the full path from the start. [Higgins02].

4 A* Pathfinding Algorithm

A* (pronounced a-star) is a *directed* algorithm, meaning that it does not blindly search for a path (like a rat in a maze) [Matthews02]. Instead it assesses the best direction to explore, sometimes backtracking to try alternatives. This means that A* will not only find a path between two points (if one exists!) but it will find the shortest path if one exists and do so relatively quickly.

4.1 How It Works

The game map has to be prepared or pre-processed before the A* algorithm can work. This involves breaking the map into different points or locations, which are called nodes. These can be waypoints, the polygons of a navigation mesh or the polygons of an area awareness system. These nodes are used to record the progress of the search. In addition to holding the map location each node has three other attributes. These are fitness, goal and heuristic commonly known as f , g , and h respectively. Different values can be assigned to paths between the nodes. Typically these values would represent the distances between the nodes. The attributes g , h , and f are defined as follows:

- g is the cost of getting from the start node to the current node i.e. the sum of all the values in the path between the start and the current node
- h stands for heuristic which is an estimated cost from the current node to the goal node (usually the straight line distance from this node to the goal)
- f is the sum of g and h and is the best estimate of the cost of the path going through the current node. In essence the lower the value of f the more efficient the path

The purpose of f , g , and h is to quantify how promising a path is up to the present node. Additionally A* maintains two lists, an *Open* and a *Closed* list. The Open list contains all the nodes in the map that have not been fully explored yet, whereas the Closed list consists of all the nodes that have been fully explored. A node is considered fully explored when the algorithm has looked at every node linked to it. Nodes therefore simply mark the state and progress of the search.

4.2 The A* Algorithm

The pseudo-code for the A* Algorithm is as follows:

1. Let P = starting point.
2. Assign f , g and h values to P .
3. Add P to the Open list. At this point, P is the only node on the Open list.
4. Let B = the best node from the Open list (i.e. the node that has the lowest f -value).
 - a. If B is the goal node, then quit – a path has been found.
 - b. If the Open list is empty, then quit – a path cannot be found
5. Let C = a valid node connected to B .
 - a. Assign f , g , and h values to C .
 - b. Check whether C is on the Open or Closed list.
 - i. If so, check whether the new path is more efficient (i.e. has a lower f -value).
 1. If so update the path.
 - ii. Else, add C to the Open list.
 - c. Repeat step 5 for all valid children of B .
6. Repeat from step 4.

A Simple example to illustrate the pseudo code outlined in section 4.2. The following step through example should help to clarify how the A* algorithm works (see Figure 4.1).

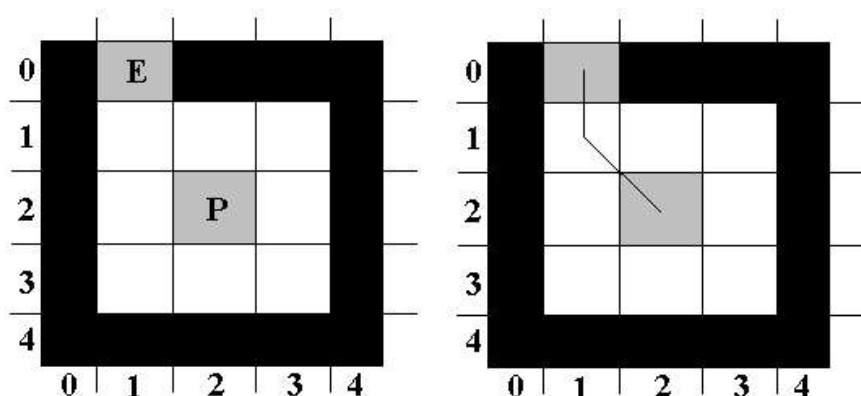


Figure 4.1

Let the center (2,2) node be the starting point (P), and the offset grey node (0,1) the end position (E). The h -value is calculated differently depending on the application. However for

this example, h will be the combined cost of the vertical and horizontal distances from the present node to (E). Therefore $h = |dx-cx| + |dy-cy|$ where (dx,dy) is the destination node and (cx,cy) is the current node.

At the start, since P(2,2) is the only node that the algorithm knows, it places it in the Open list as shown in Table 4.1.

Open List	Closed List
{ P(2,2) }	{ Empty }

Table 4.1

There are eight neighbouring nodes to p(2,2). These are (1,1), (2,1), (3,1), (1,2), (3,2), (1,3), (2,3), (3,3) respectively. If any of these nodes is not already in the Open list it is added to it. Then each node in the Open list is checked to see if it is the end node E(1,0) and if not, then its f -value is calculated ($f = g + h$).

Node	g -value	h -value	f -value
(1,1)	0 (Nodes to travel through)	1	1
(2,1)	0	2	2
(3,1)	0	3	3
(1,2)	0	2	2
(3,2)	0	4	4
(1,3)	0	3	3
(2,3)	0	4	4
(3,3)	0	5	5

Table 4.2

As can be seen from Table 4.2 Node (1,1) has the lowest f -value and is therefore the next node to be selected by the A* algorithm. Since all the neighbouring nodes to P(2,2) have been looked at, P(2,2) is added to the Closed list (as shown in Table 4.3).

Open List	Closed List
{ (1,1), (2,1), (3,1), (1,2), (3,2), (1,3), (2,3), (3,3) }	{ P(2,2) }

Table 4.3

There are four neighbouring nodes to (1,1) which are E(1,0), (2,1), (1,2), (2,2) respectively. Since E(1,0) is the only node, which is not on either of the lists, it is now looked at. Given that all the neighbours of (1,1) have been looked at, it is added to the Closed list. Since E(1,0) is the end node, a path has therefore been found and it is added to the Closed list. This path is found by back-tracking through the nodes in the Closed list from the goal node to the start node { P (2,2), (1,1), E(1,0) }. This algorithm will always find the shortest path if one exists [Matthews02].

5 **Limitations of Traditional Pathfinding**

Ironically the main problems that arise in pathfinding are due to pre-processing, which makes complex pathfinding in real-time possible. These problems include the inability of most pathfinding engines to handle dynamic worlds and produce realistic (believable) movement. This is due primarily to the pre-processing stages that produce the nodes for the pathfinder to travel along based on a static representation of the map. However if a dynamic obstacle subsequently covers a node along the predetermined path, the agent will still believe it can walk where the object is. This is one of the main factors that is holding back the next generation of computer games that are based on complex physics engines similar to that produced by middleware companies such as Havok (www.havok.com) and Renderware (www.renderware.com). Another problem is the unrealistic movement which arises when the agent walks in a straight line between nodes in the path. This is caused by the dilemma which arises in the trade off between speed (the less number of nodes to search the better) and realistic movement (the more nodes the more realistic the movement). This has been improved in some games by applying splines (curve of best fit) between the different nodes for smoothing out the path.

The problems listed above, are mainly due to the introduction of dynamic objects into static maps, are one of the focuses of research in the games industry at present. Considerable effort is going into improving the AI agent's reactive abilities when dynamic objects litter its path. One of the solutions focuses on giving the agent a method of taking into account its surroundings. A simple way to achieve this is to give the agent a few simple sensors so that it is guided by the pathfinder but not completely controlled by it. However this method will not be effective if the sensors used are unable to deal with noisy data.

5.1 **Limitations Of A***

A* requires a large amount of CPU resources, if there are many nodes to search through as is the case in large maps which are becoming popular in the newer games. In sequential programs this may cause a slight delay in the game. This delay is compounded if A* is

searching for paths for multiple AI agents and/or when the agent has to move from one side of the map to the other. This drain on CPU resources may cause the game to freeze until the optimal path is found. Game designers overcome these problems by tweaking the game so as to avoid these situations [Cain02].

The inclusion of dynamic objects to the map is also a major problem when using A*. For example once a path has been calculated, if a dynamic object then blocks the path the agent would have no knowledge of this and would continue on as normal and walk straight into the object. Simply reapplying the A* algorithm every time a node is blocked would cause excessive drain on the CPU. Research has been conducted to extend the A* algorithm to deal with this problem most notably the D* algorithm (which is short for dynamic A*) [Stentz94]. This allows for the fact that node costs may change as the AI agent moves across the map and presents an approach for modifying the cost estimates in real time. However the drawback to this approach is that it adds further to the drain to the CPU and forces a limit on the dynamic objects than can be introduces to the game.

A key issue constraining the advancement of the games industry is its over reliance on A* for pathfinding. This has resulted in game designers getting around the associated dynamic limitations by tweaking their designs rather than developing new concepts and approaches to address the issues of a dynamic environment [Higgins02]. This tweaking often results in removing/reducing the number of dynamic objects in the environment and so limits the dynamic potential of the game. A potential solution to this is to use neural networks or other machine learning techniques to learn pathfinding behaviours which would be applicable to real-time pathfinding.

5.2 Machine Learning

A possible solution to the problems mentioned in section 5.1 is to use machine learning to assimilate pathfinding behaviour. From a production point of view, machine learning could bypass the need for the thousand lines or so of brittle, special case, AI logic that is used in many games today. Machine learning if done correctly, allows generalisation for situations that did not crop up in the training process. It should also allow the game development team to develop the AI component concurrently with the other components of the game.

Machine learning techniques can be broken into the following three groups:

- **Optimisation** – Learning by optimisation involves parameterising the desired behaviour of the AI agent and presenting a performance measure for this desired

behaviour. It is then possible to assign an optimisation algorithm to search for sets of parameters that make the AI agent perform well in the game. *Genetic Algorithms* are the most commonly used technique for optimisation.

- **Training** – Learning by training involves presenting the AI agent with a set of input vectors and then comparing the output from the AI agent to the desired output. The difference between the two outputs is known as the error. The training involves modifying the internal state of the agent to minimise this error. *Neural Networks* [Fausett94] are generally used when training is required.
- **Imitation** – Learning by imitation involves letting the AI agent observe how a human player plays the game. The AI agent then attempts to imitate what it has observed. The Game observation Capture (GoCap) technique [Alexander 02] is an example of learning through imitation.

6 Learning Algorithms

There are two machine learning approaches that have been used in commercial computer games with some degree of success. These are Artificial Neural Networks (ANNs) and Genetic Algorithms (GAs). This section will outline both of these approaches in detail followed by a discussion of the practical uses for them in learning pathfinding for computer games.

6.1 Neural Networks

An artificial neural network is an information-processing system that has certain performance characteristics in common with biological neural networks [Fausett94]. Artificial Neural networks have been developed as generalizations of mathematical models of human cognition or neural biology, based on the following assumptions:

1. Information processing occurs in simple elements called neurons
2. Signals are passed between neurons over connection links
3. Each of these connections has an associated weight which alters the signal
4. Each neuron has an activation function to determine its output signal.

An artificial neural network is characterised by (a) the pattern of connections between neurons i.e. the architecture, (b) the method of determining the weights on the connections (Training and Learning algorithm) (c) the activation function.

6.2 The Biological Neuron

A biological neuron has three types of components that are of particular interest in understanding an artificial neuron: its *dendrites*, *soma*, and *axon*, all of which are shown in Figure 5.1.

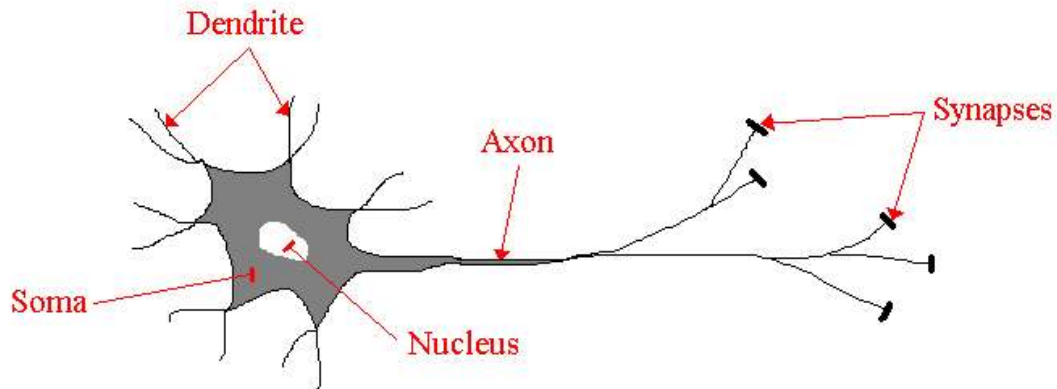


Figure 5.1

- **Dendrites** receive signals from other neurons (synapses). These signals are electric impulses that are transmitted across a synaptic gap by means of a chemical process. This chemical process modifies the incoming signal.
- **Soma** is the cell body. Its main function is to sum the incoming signals that it receives from the many dendrites connected to it. When sufficient input is received the cell fires sending a signal up the *axon*.
- **The Axon** propagates the signal, if the cell fires, to the many synapses that are connected to the dendrites of other neurons.

6.3 The Artificial Neuron

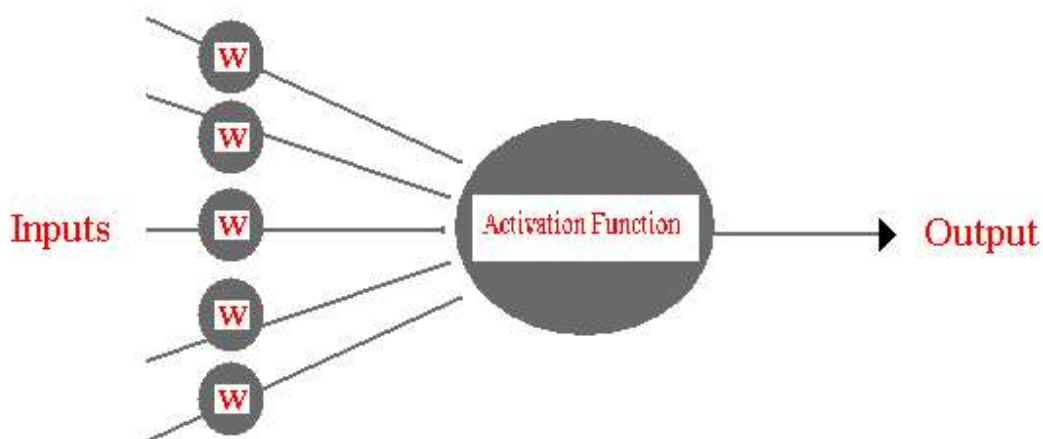


Figure 5.2

The artificial neuron structure is composed of (i) n *inputs*, where n is a real number, (ii) an *activation function* and (iii) an *output*. Each one of the inputs has a weight value associated with it and it is these weight values that determine the overall activity of the neural network. Thus when the inputs enter the neuron, their values are multiplied by their respective weights. Then the activation function sums all these weight-adjusted inputs to give an activation value (usually a floating point number). If this value is above a certain threshold the neuron outputs this value, otherwise the neuron outputs a zero value. The neurons that *receive* inputs from or *give* outputs to an external source are called *input* and *output* neurons respectively.

Thus the artificial neuron resembles the biological neuron in that (i) the inputs represent the dendrites and the weights represent the chemical process that occurs when transferring the signal across the synaptic gap, (ii) the activation function represents the soma and (iii) the output represents the axon.

6.4 Layers

It is often convenient to visualise neurons as arranged in layers with the neurons in the same layer behaving in the same manner. The key factor determining the behaviour of a neuron is its activation function. Within each layer all the neurons typically have the same activation function and the same pattern of connections to other neurons. Typically there are three categories of layers, which are *Input Layer*, *Hidden Layer* and *Output layer* respectively.

6.4.1 Input Layer

The neurons in the input layer do not have neurons attached to their inputs. Instead these neurons each have only one input from an external source. In addition the inputs are not weighted and so are not acted upon by the activation function. In essence each neuron receives one input from an external source and passes this value directly to the nodes in the next layer.

6.4.2 Hidden Layer

The neurons in the hidden layer receive inputs from the neurons in the previous input/hidden layer. These inputs are multiplied by their respective weights, summed together and then presented to the activation function which decides if the neuron should fire or not. There can be many hidden layers present in a neural network although for most problems one hidden layer is sufficient.

6.4.3 Output Layer

The neurons in the output layer are similar to the neurons in a hidden layer except that their outputs do not act as inputs to other neurons. Their outputs however represent the output of the entire network.

6.5 Activation Function

The same activation function is typically used by all the neurons in any particular layer of the network. However this condition is not required. In multi-layer neural networks the activation used is usually non-linear, in comparison with the step or binary activation function functions used in single layer networks. This is because feeding a signal through two or more layers using linear functions is the same as feeding it through one layer. The two functions that are mainly used in neural networks are the *Step* function and the *Sigmoid* function (S-shaped curves) which represent linear and non-linear functions respectively.

Figure 5.3 shows the three most common activation functions, which are binary step, binary sigmoid, and bipolar sigmoid functions respectively.

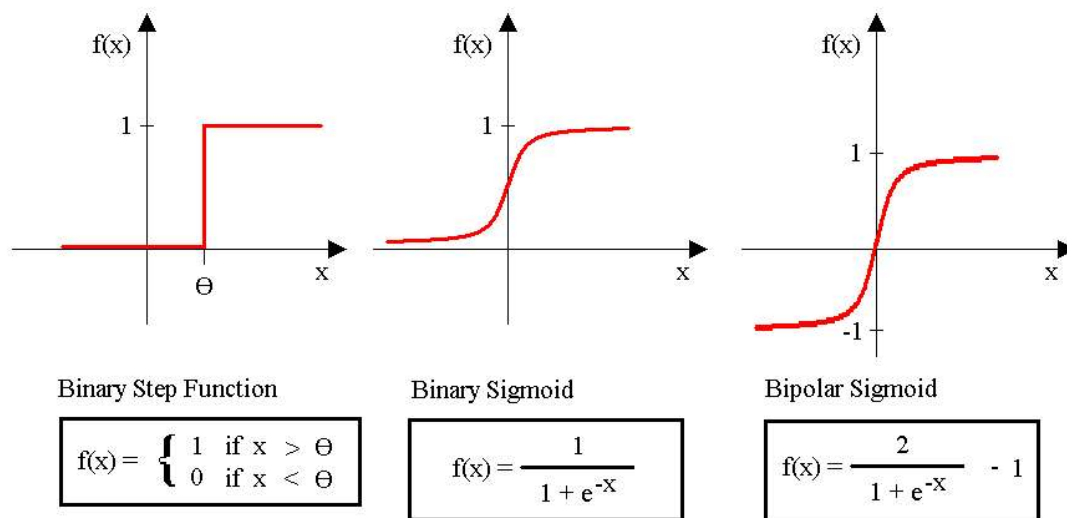


Figure 5.3

6.6 Learning

The weights associated with the inputs to each neuron are the primary means of storage for neural networks. Learning takes place by changing these weights. There are many different techniques that allow neural networks to learn by changing their weights. These broadly fall into two main categories, which are *supervised learning* and *unsupervised learning* respectively.

6.6.1 Supervised Learning

The techniques in the supervised category involve mapping a given set of inputs to a specified set of target outputs. This means that for every input pattern presented to the network the corresponding expected output pattern must be known. The main approach in this category is backpropagation, which relies on error signals from the output nodes. This requires guidance from an external source i.e. a supervisor to help (monitor) with the learning through feedback.

- **Backpropagation** – Training a neural network by backpropagation involves three stages: the feed forward of the input training pattern, the backpropagation of the associated output error, and the adjustments of the weights to minimise this error. The associated output error is calculated by subtracting the networks output pattern from the expected pattern for that input training pattern.

6.6.2 Unsupervised Learning

The techniques that fall into the unsupervised learning category have no knowledge of the correct outputs. Therefore, only a sequence of input vectors is provided. However the appropriate output vector for each input is unknown.

- **Reinforcement** – In reinforcement learning the feedback is simply a scalar value, which may be delayed in time. This reinforcement signal reflects the success or failure of the entire system after it has performed some sequence of actions. Hence the reinforcement-learning signal does not assign credit or blame to any one action. This method of learning is often referred to as the “*Slap and Tickle approach*”. Reinforcement learning techniques are appropriate when the system is required to learn on-line, or a teacher is not available to furnish error signals or target outputs.

6.7 Generalization

When the learning procedure is carried out in the correct manner, the neural network will be able to generalise. This means that it will be able to handle scenarios that it did not encounter during the training process. This is due to the way the knowledge is internalised by the neural network. Since the internal representation is neuro-fuzzy, practically no cases will be handled perfectly and so there will be small errors in the values outputted from the neural network. However it is these small errors that enable the neural network handle different situations in that it will be able to abstract what it has learned and apply it to these new situations. Thus it will be able to handle scenarios that it did not encounter during the training process. This is known as a “Generalisation”. This is opposite to what is known as “Overfitting”.

6.8 Overfitting

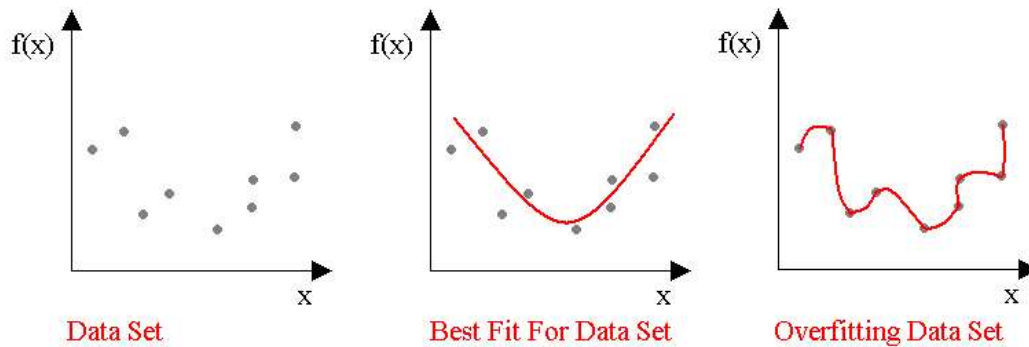


Figure 5.4

Overfitting describes when a neural network has adapted its behaviour to a very specific set of states and performs badly when presented with similar states and so therefore has lost its ability to generalise. This situation arises when the neural network is over trained i.e. the learning is halted only when the neural networks output exactly matches the expected output.

6.9 Topology

The topology of a NN refers to the layout of its nodes and how they are connected. There are many different topologies for a fixed neuron count, but most structures are either obsolete or practically useless. The following are examples of well-documented topologies that are best suited for game AI.

6.9.1 Feed forward

This is where the information flows directly from the input to the outputs. Evolution and training with back propagation are both possibilities. There is a connection restriction as the information can only flow forwards hence the name feed forward.

6.9.2 Recurrent

These networks have no restrictions and so the information can flow backward thus allowing feedback. This provides the network with a sense of state due to the internal variables needed for the simulation. The training process is however more complex than the feed forward because the information is flowing both ways.

7 Genetic Algorithms

Nature has a robust way of evolving successful organisms. The organisms that are ill suited for an environment die off, whereas the ones that are fit live to reproduce passing down their good genes to their offspring. Each new generation has organisms, which are similar to the fit members of the previous generation. If the environment changes slowly the organisms can gradually evolve along with it. Occasionally random mutations occur, and although this usually

means a quick death for the mutated individual, some mutations lead to a new successful species.

It transpires that what is good for nature is also good for artificial systems, especially if the artificial system includes a lot of non-linear elements. The genetic algorithm, described in [RusselNorvig95] works by filling a system with organisms each with randomly selected genes that control how the organism behaves in the system. Then a fitness function is applied to each organism to find the two fittest organisms for this system. These two organisms then each contribute some of their genes to a new organism, their offspring, which is then added to the population. The fitness function depends on the problem, but in any case, it is a function that takes an individual as an input and returns a real number as an output.

The genetic algorithm technique attempts to imitate the process of evolution directly, performing selection and interbreeding with randomised crossover and mutation operations on populations of programs, algorithms or sets of parameters. Genetic algorithms and genetic programming have achieved some truly remarkable results in recent years [Koza99], beautifully disproving the public misconception that a computer “can only do what we program it to do”.

7.1 Selection

The selection process involves selecting two or more organisms to pass on their genes to the next generation. There are many different methods used for selection. These range from randomly picking two organisms with no weight on their fitness score to sorting the organisms based on their fitness scores and then picking the top two as the parents. The main selection methods used by the majority of genetic algorithms are: *Roulette Wheel selection*, *Tournament selection*, and *Steady State selection*. Another important factor in the selection process is how the fitness of each organism is interpreted. If the fitness is not adjusted in any way it is referred to as the *raw fitness value* of the organism otherwise it is called the *adjusted fitness value*. The reason for adjusting the fitness values of the organisms is to give them a better chance of being selected when there are large deviations in the fitness values of the entire population.

7.1.1 Tournament Selection

In tournament selection n (where n is a real number) organisms are selected at random and then the fittest of these organisms is chosen to add to the next generation. This process is repeated as many times as is required to create a new population of organisms. The organisms that are selected are not removed from the population and therefore can be chosen any number of times.

This selection method is very efficient to implement as it does not require any adjustment to the fitness value of each organism. The drawback with this method is that because it can converge quickly on a solution it can get stuck in local minima.

7.1.2 Roulette Wheel Selection

Roulette wheel selection is a method of choosing organisms from the population in a way that is proportional to their fitness value. This means that the fitter the organism, the higher the probability it has of being selected. This method does not guarantee that the fittest organisms will be selected, merely that they have a high probability of being selected. It is called roulette wheel selection because the implementation of it involves representing the populations total fitness score as a pie chart or roulette wheel. Each organism is assigned a slice of the wheel where the size of each slice is proportional to that respective organisms fitness value. Therefore the fitter the organism the bigger the slice of the wheel it will be allocated. The organism is then selected by spinning the roulette wheel as in the game of roulette.

This roulette selection method is not as efficient as the tournament selection method because there is an adjustment to each organism's fitness value in order to represent it as a slice in the wheel. Another drawback with this approach is that it is possible that the fittest organism will not get selected for the next generation, although this method benefits from not getting stuck in as many local minima.

7.1.3 Steady State Selection

Steady state selection always selects the fittest organism in the population. This method retains all but a few of the worst performers from the current population. This is a form of *elitism selection* as only the fittest organisms have a chance of being selected. This method usually converges quickly on a solution but often this is just a local minima of the complete solution. The main drawback with this method is that it tends to always select the same parents every generation this results in a dramatic reduction in the gene pool used to create the child. This is why the population tends to get stuck in local minima as it is over relying on mutation to create a better organism. This method is ideal for tackling problems that have no local minima or for initially getting the population to converge on a solution in conjunction with one of the other selection methods.

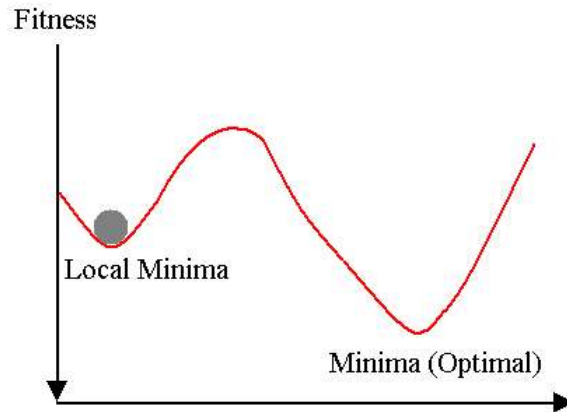
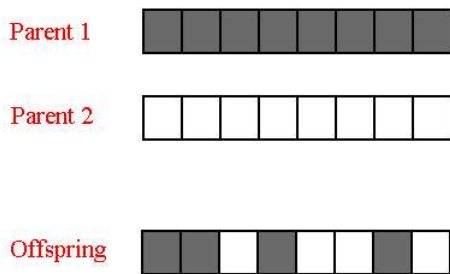


Figure 6.1

7.2 Crossover

The crossover process is where a mixture of the parent’s genes is passed onto the new child organism. There are three main approaches to this *random crossover*, *single-point crossover* and *two-point crossover*.

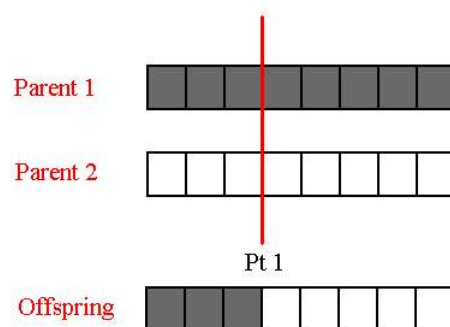
7.2.1 Random Crossover



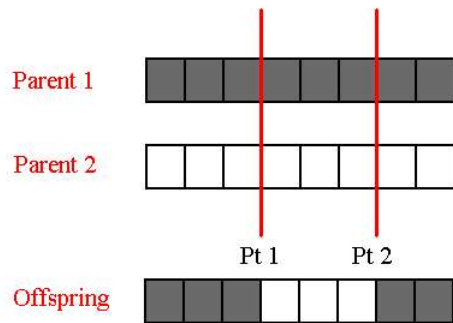
For each of the genes in the offsprings chromosome a random number of either zero or one is generated. If the number is zero the offspring will inherit the same gene in Patent 1 otherwise the offspring will inherit the appropriate gene from Parent 2. This results in the offspring inheriting a random distribution of genes from both parents.

7.2.2 Single-Point Crossover

A random crossover point is generated in the range ($0 < Pt\ 1 < \text{length of Offspring chromosome}$). The offspring inherits all the genes that occur before Pt1 from Parent 1 and all the genes that occur after Pt1 from Parent 2.



7.2.3 Two-Point Crossover



This is the same as the single-point crossover except this time two random crossover points are generated. The offspring inherits all the genes before Pt1 and after Pt2 from Parent 1 while all the genes between Pt1 and Pt2 are inherited from Parent 2.

7.3 Mutation

Mutation is another key feature of the crossover phase in genetic algorithms as it results in creating new genes that are added to the population. This technique is used to enable the genetic algorithm to get out of (resolve) local minima. The mutation is controlled by the mutation rate of the genetic algorithm and is the probability that a gene in the offspring will be mutated. Mutation occurs during the crossover stage; when the child is inheriting its genes from the parent organisms each gene is checked against the probability of a mutation. If this condition is met then that gene is mutated although this usually means a quick death for the mutated individual, some mutations lead to a new successful species.

8 How Learning Algorithms Can Solve Pathfinding Problems

The main problems associated with real-time pathfinding are:

- Handling dynamic objects
- Using up too many resources especially on game consoles, which have limited memory
- Leave the AI until the end of the development process

Learning algorithms offer the possibility of a general pathfinding Application Programming Interface (API) that would allow an agent to learn how to find its way around the game world. An API is a collection of specific methods prescribed by an application program by which a programmer writing another program can make requests to. This would allow game developers experiment with training agents in more complex situations and then simply reuse this behaviour in future games even if the new game was completely different, as long as it can present the agent with the inputs it requires (as defined by the API).

8.1 Evolving the Weights of a Neural Network

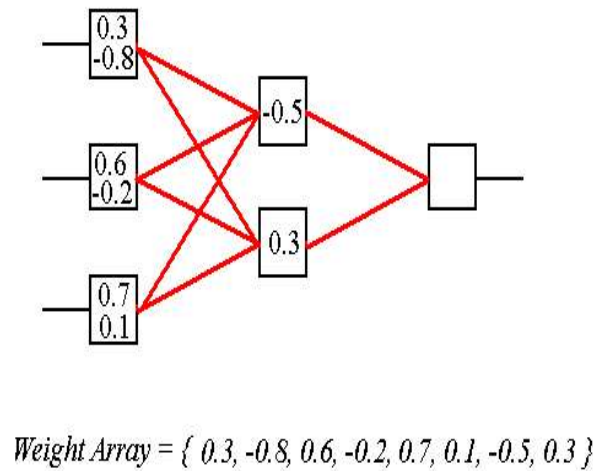


Figure 8.1

The encoding of a neural network which is to be evolved by a genetic algorithm is very straightforward. This is achieved by reading all the weights from its respective layers and storing them in an array. This weight array represents the chromosome of the organism with each individual weight representing a gene. During crossover the arrays for both parents are lined up side by side. Then depending on the crossover method, the genetic algorithm chooses the respective parents weights to be passed on to the offspring.

A discussion on how learning algorithms can overcome the problems listed at the start of this section will now be looked at:

- **Dynamic problem** – Having continuous real-time sensors the AI agent should be able to learn to use this information via a neural network to steer around obstacles and adjust to a changing environment. Also because neural networks can generalise, the agent should be able to make reasonable decisions when it encounters a situation that did not arise during training. Genetic algorithms can be used to train the neural network online as new elements are added to the game.
- **Resource problem** – Neural networks do not require large amounts of memory and can handle continuous inputs in real-time as the data processing mainly involves addition and multiplication which are among the fastest processes a computer can perform

- **Speeding up the development process** – Giving the agents the ability to learn how to navigate around the map allows developers to start developing the AI at an earlier stage. If a new element is added to the game, then it should be a simple matter of allowing the agent learn the game with this new element.

8.2 Conclusion

The reason that games developers have not researched machine learning for pathfinding is that it would take too much time to do so and time is money! Games developers are also very reluctant to experiment with machine learning, as it could be unpredictable. One game that did use unsupervised machine learning was Black & White (www.lionhead.com) where the human player was able to train their own creature. A neural network was used to control the creature but while the human player was able to train the creature through reinforcement learning, it was tightly controlled to avoid totally unpredictable/unrealistic behaviour. So it seems that until the game developers are shown proof that machine learning can overcome the limitations of standard approaches they will avoid it.

Future work will involve setting up a test bed to test the practicality of using machine learning to perform pathfinding. Pacman is the game chosen for the test bed, as it is a real-time game that uses pathfinding algorithms to navigate around a 2D maze. Learning algorithms will be used to train a neural ghost that can be compared to standard ghosts in terms of speed, believability, and ability to play dynamic maps. The ghosts will use a neural network, which will decipher real-time data inputted by a number of sensors attached to the ghost, to decide what path to follow. The neural network will be trained using reinforcement learning with a genetic algorithm to evolve the weights as described in section 8.1.

References

- [Alexander02] Alexander, Thor., “GoCap: Game Observation Capture”, AI Game Programming Wisdom, Charles River Media, 2002
- [Alexander02a] Alexander, Thor., “Optimized Machine Learning with GoCap”, Game Programming Gems 3, Charles River Media, 2002
- [Board & Ducker02] Board, Ben., Ducker, Mike., “Area Navigation: Expanding the Path-Finding Paradigm”, Game Programming Gems 3, Charles River Media, 2002
- [Cain02] Cain, Timothy, “Practical Optimizations for A*”, AI Game Programming Wisdom, Charles River Media, 2002
- [Fausett94] Fausett, Laurene, “Fundamentals of Neural Networks Architectures, Algorithms, and Applications”, Prentice-Hall, Inc, 1994.
- [Higgins02] Higgins, Dan, “Generic A* Pathfinding”, AI Game Programming Wisdom, Charles

River Media, 2002.

[Higgins02a] Higgins, Dan, "How to Achieve Lightning-Fast A*", AI Game Programming Wisdom, Charles River Media, 2002

[Higgins02b] Higgins, Dan., "Pathfinding Design Architecture", AI Game Programming Wisdom, Charles River Media, 2002

[Higgins02c] Higgins, Dan., "Generic Pathfinding", AI Game Programming Wisdom, Charles River Media, 2002

[Matthews02] Matthews, James, "Basic A* Pathfinding Made Simple", AI Game Programming Wisdom, Charles River Media, 2002.

[RusselNorvig95] Russel, Stuart., Norvig, Peter., "Artificial Intelligence A Modern Approach", Prentice-Hall, Inc, 1995

[Stentz94] Stentz, Anthony., "Optimal and Efficient Path Planning for Partially-known Environments." In proceedings of the IEEE International Conference on Robotics and Automation, May 1994

[Stentz96] Stentz, Anthony., "Map-Based Strategies for Robot Navigation in Unknown Environments". In proceedings of the AAAI Spring Symposium on Planning with Incomplete Information for Robot Problems, 1996

[WhiteChristensen02] White, Stephen., Christensen, Christopher., "A Fast Approach to Navigation Meshes", Game Programming Gems 3, Charles River Media, 2002