Realistic Agent Movement in Dynamic Game Environments

Ross Graham
Institute of Technology Blanchardstown
Dublin 15
Ireland
353 1 885 1119
grahamrp@gmail.com

Hugh McCabe
Institute of Technology Blanchardstown
Dublin 15
Ireland
353 1 885 1089
hugh.mccabe@itb.ie

Stephen Sheridan
Institute of Technology Blanchardstown
Dublin 15
Ireland
353 1 885 1094
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 paper examines pathfinding algorithms used presently in games and details their shortcomings. These shortcomings are particularly apparent when pathfinding must be carried out in real-time in dynamic environments. This paper proposes a strategy by which machine learning techniques such as Artificial Neural Networks and Genetic Algorithms can be used to enhance traditional pathfinding algorithms to solve the real-time aspect of this problem. We describe a test bed system, currently in development, that incorporates these machine learning techniques into a 3D game engine.

Keywords
AI, Pathfinding, Computer Games, Neural Network, Genetic Algorithm

1 INTRODUCTION

Proceedings of DiGRA 2005 Conference: Changing Views – Worlds in Play.

Realistic agent movement has always been difficult to achieve. This is particularly apparent in cases where we require the agents to carry out *pathfinding*; to navigate their way from a start position to a goal position through a complex game world. This difficulty is compounded in modern games that are becoming more dynamic in nature as a result of middleware engines such as Renderware [14] and Havok [6]. These middleware companies allow game developers to spend more time developing interesting dynamic games because they remove the need to build custom physics engines for each game. But these new dynamic games create a strain on the existing traditional pathfinding strategies as these strategies rely on a static representation of the game world. Therefore, since the game world can change in real-time, the pathfinding strategy also has to adapt in real-time to cope with this. There are two principal abilities required in basic real-time pathfinding. These are:

- the ability to head in the direction of a goal, and
- the ability to avoid obstacles that may arise along the path towards the goal.

This paper will highlight the need for real-time pathfinding and investigate how an agent can be controlled by a neural network (NN) in order to achieve the desired behaviour. We then discuss the steps necessary to train a NN to learn the two basic real-time path-finding components, and the results achieved. In the next section we describe the traditional pathfinding approach. We then discuss some related work on solving the real-time pathfinding problem. Section 4 details our proposed approach using Neural Networks and section 5 discusses the results we obtained. We finish by presenting our conclusions and outline plans for future work.

2 TRADITIONAL PATHFINDING

Typically the game world geometry is stored in a structure called a *map*. Maps contain all the polygons that make up the game environment. In most cases, in order to cut down the search space of the game world for the pathfinder, the game 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. These simplified representations correspond to *graphs* upon which search algorithms such as Dijkstra or A* [4][7][8][16] can be used to find paths between the nodes on these graphs. Common forms of simplified representations are *navigation meshes* and *waypoints* [1] [4] [21].

These simplified representations of a game map are too computationally expensive to be produced in real-time as game-play progresses and therefore they are pre-processed and loaded when the game begins. This means that throughout the course of the game, the AI agent can only search this static representation of the map that was created at load time. Unfortunately, the assumption that the geometry of the game remains static during the course of play is not necessarily valid anymore. This difficulty is then compounded by the fact that the agent typically has no real-time awareness of the environment around it. This situation results in a number of limitations for traditional pathfinding, some of which we now outline.

Firstly, middleware physics engines have the potential to allow completely *dynamic game geometry* where the players and agents can physically alter the structure of the game world as play progresses, by knocking over walls for example. Dynamic obstacles can therefore be introduced that block previously accessible nodes on the pathfinding graph. When this happens

the agent will still believe it can walk along this path due to its reliance on the pre-processed graph. Techniques have been developed that improve the agents' reactive abilities when dynamic objects obstruct a path. These work well in some situations but generally the agent will not react until it has collided with an obstacle as it has no sense of awareness until a trigger is set when a collision occurs.

Another problem is the *unrealistic movement* which arises when the agent walks in a straight line between nodes along the path. This is caused by the dilemma which occurs 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 [13][20].

Finally, another problem is the difficulty of implementing any kind of *tactical pathfinding*. This involves not just finding the shortest route but also the route that offers the most cover or avoids unnecessary encounters with undesirable game entities. One approach to this is to modify the cost heuristic of A* to take line of fire from other enemy agents into account [19]. This has the benefits of adding a more realistic touch to the game and also presents a less predictable opponent to the human player. The drawback is that due to the added cost, the search space becomes much larger for A* to process. This approach also assumes that the threat remains static during the paths duration, which is seldom the case.

In conclusion, computer games are now being built using middleware for key components of the game, including the physics engine. Middleware is software written by an external source that has hooks that allow it to be integrated into a game developer's code. Therefore game developers can spend much more time creating more flexible games with real-time dynamic scenes. This sounds exciting; however it is being impeded by traditional pathfinding AI that operates off a *static* representation of the games virtual environment. This limits the amount of dynamic objects that can be added to games, as the pathfinding strategy will have to be fine-tuned to handle them thus adding more time to the development of the game. Therefore new real-time pathfinding strategies are needed to complement the next generation of computer games and thereby give the user a more immersive experience.

3 REAL-TIME PATHFINDING

The two components for basic real-time pathfinding are (i) heading in the direction of a goal and (ii) avoiding any static and dynamic obstacles that may arise along the path to that goal. In order to avoid obstacles the agent need some method of detecting the presence of these obstacles and hence needs to be given some form of real-time awareness its surroundings so that is can react accordingly. There has been significant research into the implementation of real-time pathfinding, especially in the robotics field as similar problems are encountered here. These approaches will now be outlined with particular focus on their usefulness within the computer games domain.

3.1 Real-time A*

The Real-time A* algorithm [10] is designed to handle complex search spaces and in particular to ensure that results are returned within specified time limits in order to allow robots or agents to progress This algorithm works on the same principle as the A* algorithm except it is subject to a time variable i.e. a look ahead time. If the goal is found within the set time period it effectively

works in the same manner as A*. If the goal is not found within the time period the best path is chosen according to the information available to the agent. This information takes the form of a table of costs associated with each node which is then updated on an ongoing basis.

Real-time A* could be useful in computer games that require the navigation of large numbers of AI agents since the time interval can be set dynamically. This prevents the pathfinding taking longer than the graphics engine per frame which would result in jerky movement if not corrected. The potential flaw is if the time interval is too small the resulting paths will be far from optimal. The main drawback though is that the algorithm still relies on a pre-processed static representation of the map and therefore all of the dynamic problems highlighted in the previous section still apply. One option may be to use the costs associated with the nodes in order to deal with dynamic obstacles but the algorithm is not designed to cope with graphs where the structure changes dynamically.

3.2 D* Algorithm

There has been considerable research in the robotics field into pathfinding in a dynamic environment. One of the most suitable approaches with respect to computer games is the D* algorithm which stands for dynamic A* [17] [18]. The algorithm starts by computing a path from the start position to the end position in the normal manner. However if unforeseen obstacles are encountered while travelling along this path then the search algorithm is executed again in order to compute potential new paths. This is an interesting solution however it is may not be suitable for real-time games due to its requirement for extensive processing time in particular if there were more than one agent involved. Another potential problem is the generation of new waypoints in real-time as the agent encounters inconsistencies within the representation of the game map it possesses.

3.3 Steering Algorithms

Steering algorithms are sometimes referred to as *short-range pathfinding* algorithms and are intended to provide a means of carrying out tasks such as steering around obstacles and following paths. Any steering algorithm involves giving the agent real-time awareness using sensors in order to make decisions with regard to the local environment. Examples of steering include *force based methods* such as the flocking techniques popularised by Reynolds [15], and *ray casting methods* [20].

- **Force based steering:** Potential obstacles are regarded as emitting repulsive forces which grow stronger with proximity. As an agent travels around an environment containing such obstacles it computes the net result of these forces and uses it to adjust its velocity and acceleration accordingly. Different types of agent behaviour can be achieved depending on how the forces are deciphered.

- **Ray casting steering:** The agent employs sensors which test and resolves potential lines of movement against the environment, or against some pre-computed representation of that environment. Typically the agent will head in the direction with the longest sensor range.

Steering algorithms offer an effective real-time response to dynamic obstacles since the sensors are gathering information in real-time and thus can react to sudden changes within the immediate environment of the agent. Considerable research has been carried out on force based steering

since it offers the possibility of controlling group behaviour, known as *flocking* [15]. This flocking behaviour has been applied to monster characters in well known computer games such as Unreal and Half-Life [20] thereby offering low cost unscripted behaviour when groups of monsters appear together. Real-time strategy (RTS) games regularly use these flocking techniques as they typically have to deal with large amounts of active agents, all within the viewpoint of the player. Ray casting steering on the other hand, while being computationally less expensive, as it does not require computation of forces, is largely overlooked as deciding how to decipher the sensor data is much more abstract. However, if useful patterns could be derived from the sensor data, ray casting may prove to be a very useful tool for avoiding dynamic obstacles.

4 REAL-TIME PATHFINDING WITH LEARNING ALGORITHMS

Our objective is to provide the agent with a means of learning to navigate its own way around the game world rather than simply relying on routes provided by the game engine via a pathfinding algorithm such as A*. Providing an agent with this functionality means providing it with two important abilities. Firstly it needs the ability to examine its environment in some way in order to know what is in front of it and around it, thus giving it real-time awareness. Secondly it needs some way of processing this information to accomplish tasks such as steering around obstacles that have been placed in its path.

4.1 Sensors

The first ability is achieved by embedding sensors in the agent. This is a concept borrowed from the robotics literature where ultrasound or infrared sensors are commonly employed. We adapt this idea for our virtual agents by casting rays from the agent which test for intersections with the geometry of the game world. This scenario is illustrated in Figure 4.1 In this way information is provided to the agent pertaining to the proximity of objects within its field of vision.

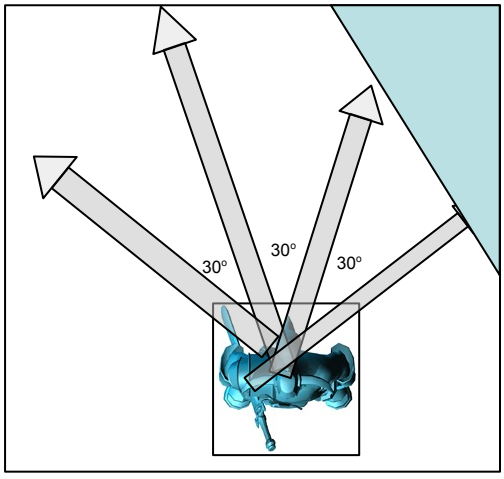

Figure 4.1: Shows an agent with four sensors

In our implementation the agents are equipped with four sensors each of which is separated by an angle of 30°. This gives an effective field of vision of 90° which should allow the agent to detect any obstacles which will significantly affect its path.

4.2 Interpreting Sensor Data

The second ability is to process this information in some way, and our solution to this problem is to furnish each agent with an Artificial Neural Network (ANN) [3] [5] which takes the sensor information as input. The ANN is a learning algorithm that is trained to exhibit the desired behaviour we want – namely that the agent has the ability to steer around objects. If trained correctly ANN's can generalise on situations that they have not encountered during training [3][16] and this should be useful when dealing with dynamic environments. Neural Networks once trained, should provide a very robust steering behaviour that is extremely tolerant of noisy data. Another advantage of this approach is that the amount of processing required is minimal and hence multiple agents can be imbued with this behaviour without causing a major strain on the CPU.

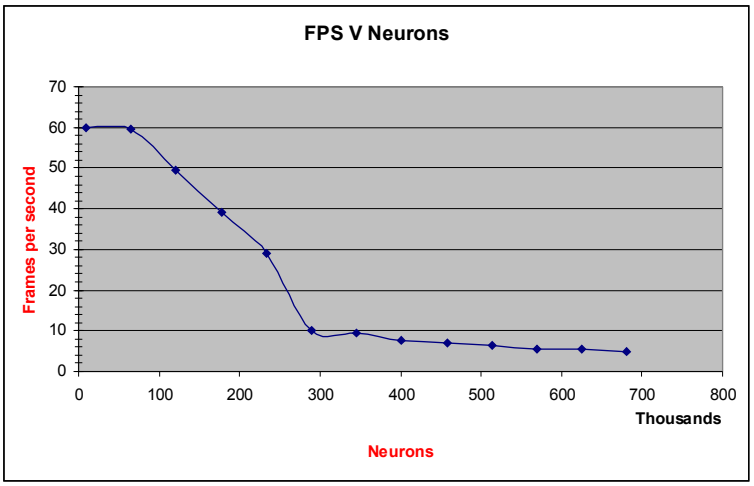

Figure 4.2: Shows the relationship between the amount of neurons in the Quake 2 engine and the Frames per Second (FPS)

The definitive measure of success in real-time games is the frames per second (FPS) rate. For a real-time game frame rates below 25 - 30 FPS are generally deemed unacceptable. As shown in figure 3.1 our system can have well over two hundred thousand neurons active in the Quake 2 engine at 30 FPS. This translates depending on how many neurons a NN is composed of to thousands of AI agents being able to use a trained NN at the same time.

The data from the sensor is fed as input to the NN and then in turn the output from the NN will govern how the AI agent will move. This is a relatively straightforward scenario to set up but the main problem arises in training the NN to learn useful reactions to the inputs it receives. The training is accomplished by evolving the weights of the NN through a genetic algorithm (GA). So essentially our approach involves getting an ANN to learn meaningful patterns from the sensors i.e. ray casting steering.

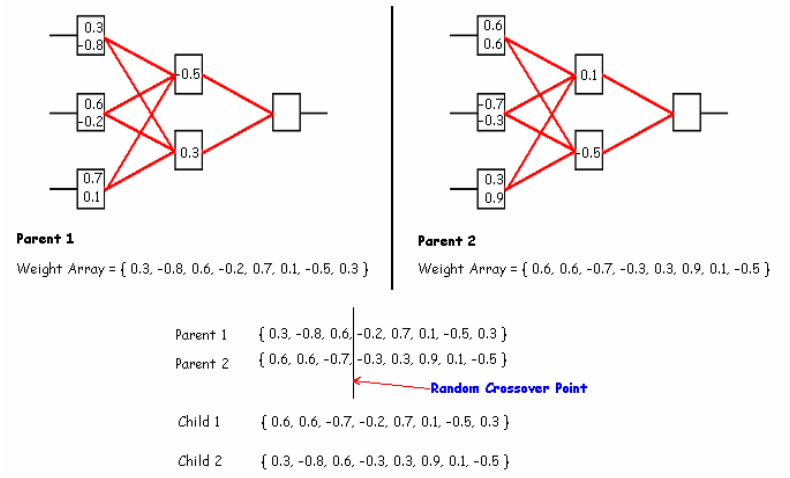

Figure 4.3: Illustrates the evolution of a NN's weights

The encoding of a neural network which is to be evolved by a genetic algorithm is fairly 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 (AI agent) 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 as shown in figure 4.3. Our system is implemented using the Quake 2 game engine by id software.

4.3 Training

Reinforcement learning [2] [16] is used to evolve the NN's weights through a genetic algorithm (GA). This is achieved by rewarding AI agents for following various rules that the user specifies at different time intervals. Then the AI agents are then ranked according to their respective scores, with the top ranking agents putting a mixture of their weights into a lower ranking agent. This is analogous to the evolutionary *survival of the fittest* model.

The NN and the GA were implemented in C++ and compiled into a standalone library named the AI Library. The AI Library gives any programme linking to it access to NN, GA and traditional pathfinding functionality through high-level commands. Therefore to train the AI agents within the Quake2 engine the AI Library was linked to the engine's source code. Once linked a number of graphical user interfaces (GUI) were implemented that allow the user to integrate a NN into the AI agents and evolve them through a GA.

4.3.1 GA Options GUI

The user is given real-time control over all the GA parameters thus giving the user huge scope to dynamically change each of them throughout a simulation. These parameters are the selection function, the crossover function, mutation probability, evolution time and all the elements concerned with the rank function. This facilitates evolution in stages of difficulty, by introducing more elements as the AI agent learns previous ones, thus gradually evolving to a more complex behaviour.

4.3.2 NN Options GUI

The NN options GUI allows the user real-time control over the inputs to each AI agents NN and its activation function. It also offers the user the facility to bias certain inputs thus decreasing the search space for the NN initially, and then gradually removing the bias values at later stages of the evolution thus gradually increasing the search space. This again facilitates evolution through different stages of difficulty. A set of custom maps were also created to facilitate training the AI agents to learn the basic components of real-time pathfinding.

5 RESULTS

The first thing that the NN was tested on was its ability to go towards a goal. The idea here is to have an agent relentlessly pursues a dynamic object around an obstacle free space. Therefore the agent will decide which way to move via a NN that takes the relative position of the goal as its input. The NN has three outputs which are *turn left*, *move forward* and *turn right* respectively. The output with the strongest signal will be selected for the next move. This was learned with ease by the AI agents by scoring them for moving towards the goal. An interesting result however is the variety in the solutions the GA produces. This is shown in figure 5.1 where three agents x and y coordinates were recorded as they moved from the same initial position to the same goal.

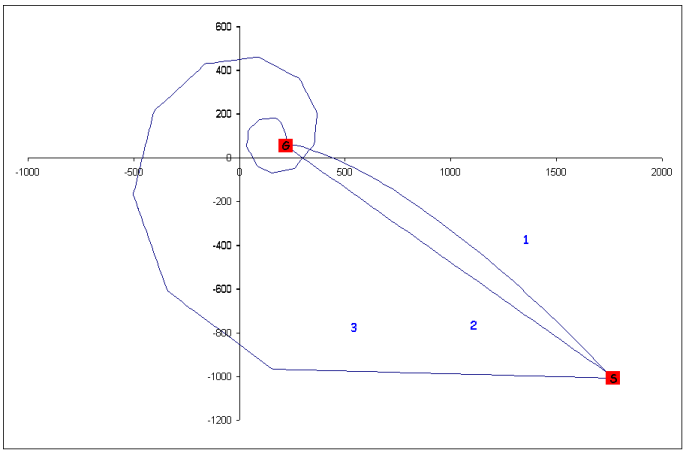

Figure 5.1: Trace of three AI agents as they move from the same starting position (S) to the same goal position (G)

The next test was to supply the AI agents with sensors and insert them into a map with obstacles and evolve them to use the sensor information to steer around obstacles. Once again the NN had no trouble learning this behaviour once scored on valid moves and turning in the correct direction once the sensors detected an obstacle. This time the inputs were the sensors and the output was the same as before.

The next test was to see if a NN could learn to head in the direction of a goal and avoid obstacles that may litter the path. The AI agent also has no prior knowledge of the map and reacts purely on what it senses in real-time. The inputs provided to the NN were relative position to the goal and the data received from each of the sensors. This proved to be very difficult for the NN to learn so much so that a complete rethink on the training procedures had to be done. It was also evident that a NN with one hidden layer was not capable to learning this behaviour. Another

major change that was integrated into the system was the ability to run the simulation in discreet intervals. This meant at the end of each interval the agents were reset to there original position and orientation.

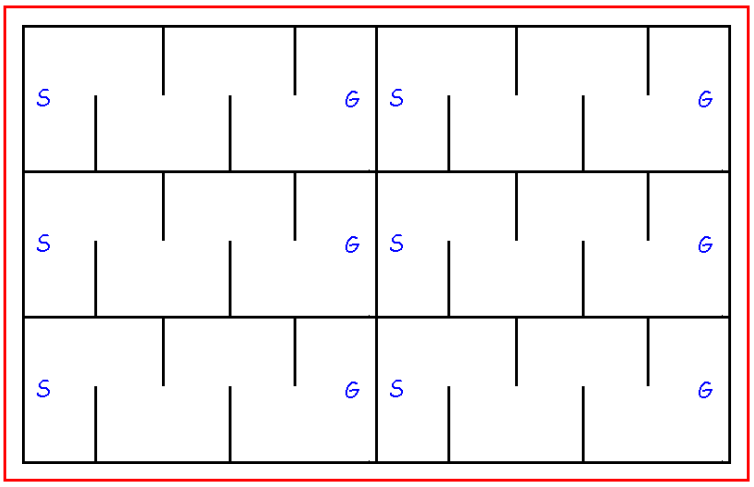

Figure 5.2: Outline of one of the bot training maps where bots have to move from (S) to (G) to score points

This spawned a series of new custom maps which we call the *bot boot camps*. These maps contain sets of parallel obstacle courses, each of which takes a single AI agent for discreet evolution. Figure 5.2 shows an outline of one of the custom bot boot camp maps. Each agent starts at the left side of the map (S) and has to make its way to the goal on the right (G).This finally produced AI agents that would head towards a goal and avoid obstacles on the way.

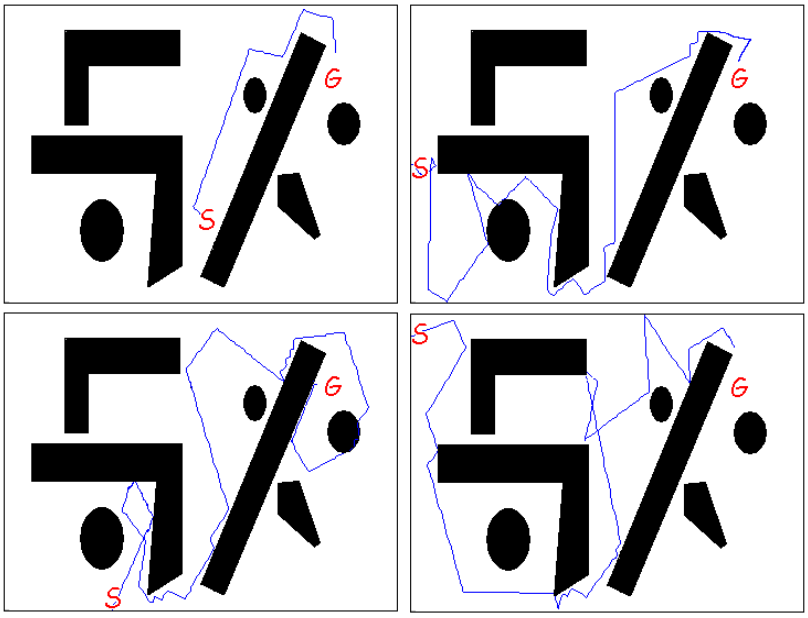

Figure 5.3: Trace of four AI agents as they move from their starting position (S) to the goal position (G) on the same map

As shown in figure 5.3 the path the AI agent takes is not the smoothest of paths but illustrates that the agent has learned to head towards the goal position and avoid obstacles on route with no prior knowledge of the map.

6 CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

While NN's seemed an obvious choice for our implementation of real-time pathfinding, due to their speed at deciphering real-time data and their ability to generalise, they proved very difficult to train. However, the results that have been achieved so far demonstrate that NN can learn the basic components of real-time pathfinding. This is an exciting prospect as it could become the basis of a real-time pathfinding Application Programming Interface (API) that could be used by game developers for low level pathfinding in a dynamic game map. The only element the game engine would have to provide would be a ray casting function which is a basic component of any physics engine.

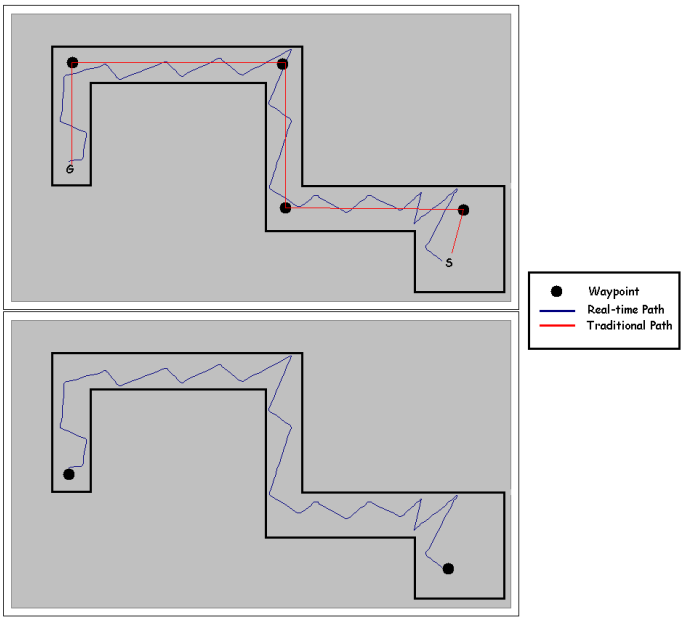

Figure 6.1: Shows how real-time pathfinding reduces waypoints in a simple map

6.2 Future Work

Future work will involve refining the training procedures further so as to obtain better results. Experiments that have been carried out using the combinations of sensors and Neural Nets to emulate smooth steering behaviour have been successful so we anticipate that refinement of training procedures should result in smoother paths. We will also investigate how the use of hybrid neural networks [11] might compliment our results. These would be capable of breaking up the problem into its two components thus reducing the search space for the full problem. Since there will constantly be situations where a higher planning algorithm will be needed to guide the AI agent in complex maps, we will investigate the concept of using a trained NN to cut down the number of waypoints required to represent these game maps. Figure 6.1 illustrates how

the simple map requires four waypoints to represent it. Whereas by using a trained NN with sensors the map can be represented by two waypoints with the added benefit of being able to avoid any obstacle that may litter the map during runtime.

REFERENCES

1. Board, B. and M. Ducker, *Area Navigation: Expanding the Path-Finding Paradigm*, in *Game Programming Gems 3*. 2002, Charles River Media.
2. Champandard, A.J., *AI Game Development*. 2004: New Riders Publishing.
3. Fausett, L., *Fundamentals of Neural Network Architectures, Algorithms, and Applications*. 1994: Prentice-Hall Inc.
4. Graham, R., *Pathfinding in Computer Games.* in proceedings of ITB Journal, 2004.
5. Graham, R., *Neural Networks for Real-time Pathfinding in Computer Games.* ITB Journal, 2004(Issue 9).
6. Havok: www.havok.com
7. Higgins, D., *Generic A* Pathfinding*, in *AI Game Programming Wisdom*. 2002, Charles River Media.
8. Higgins, D., *How to Achieve Lightning-Fast A**, in *AI Game Programming Wisdom*. 2002, Charles River Media.
9. Higgins, D., *Generic Pathfinding*, in *AI Game Programming Wisdom*. 2002, Charles River Media.
10. Korf, R.E., *Real-Time Heuristic Search.*, Artificial Intelligence, Vol. 42, No. 2-3, March 1990, pp. 189-211.
11. Masters, T., *Practical Neural Network Recipes in C++*. 1993: Boston: Academic Press.
12. Matthews, J., *Basic A* Pathfinding Made Simple*, in *AI Game Programming Wisdom*. 2002, Charles River Media.
13. Rabin, S., *A* Aesthetic Optimizations*, in *Game Programming Gems*. 2000, Charles River Media.
14. Renderware: www.renderware.com
15. Reynolds, C.W. *Steering Behaviours For Autonomous Characters*. in *Game Developers Conference*. 1999. San Jose California.
16. Russel, S. and P. Norvig, *Artificial Intelligence A Modern Approach*. 1995: Prentice-Hall, Inc.
17. Stentz, A. *Optimal and Efficient Path Planning for Partially-known Environments*. in *IEEE International Conference on Robotics and Automation*. 1994.
18. Stentz, A. *Map-Based Strategies for Robot Navigation in Unknown Environments*. in *AAAI Spring Symposium on Planning with Incomplete Information for Robot Problems*. 1996.
19. Sterren, W.V.d., *Tactical Path-Finding with A**, in *Game Programming Gems 3*. 2002, Charles River Media.
20. Tomlinson, S.L., *The Long and Short of Steering In Computer Games*. 2004.
21. White, S. and C. Christensen, *A Fast Approach to Navigation Meshes*, in *Game Programming Gems 3*. 2002, Charles River Media.