Implementation and Comparison of A.I. Agents in Game Play

Aman Rana, Katie Gandomi, Miguel Lasa, Rahul Krishnan

Abstract—This paper proposes the implementation and comparison of different A.I. agents that play the Super Mario game. The paper proposes to implement an A* Agent, a Genetic Algorithm Agent, a Forward Jumping Agent and a Weighted Reflex Agent; and compare their performance using certain predefined criteria. The advantages and disadvantages of the proposed algorithms in game play has been discussed.

Index Terms—AI, Game play, A* agent, Genetic Agent, Forward Jumping Agent, Weighted reflex Agent, Super Mario.

1 Introduction

ARTIFICIAL Intelligence, since it was invented, has been used in game play. Game play is an area in which people deal with behaviors that are generated completely by an AI algorithm. The market for video games is growing, with sales in 2007 of $17.94 billion marking a 43% increase over 2006 - *Microsoft Research* [9]. One of the first games that used AI came to market as early as 1951 - *Nim* (published in 1952) and was able to win against very skilled players. Since then AI has come a long way, from amateur programs to play Chess and Checkers to IBMs *Deep Blue*, which famously defeated Garry Kasparov in the game of Chess in 1997, and these algorithms are becoming even smarter. Recently in March 2015, Googles AI program *AlphaGo* defeated the world champion, Lee Sedol, in the game of *Go*, a game considered to be challenging for an artificial agent to play because of its boundless number of moves due to a huge branching factor.

- *Aman Rana, Department of Robotics Engineering, Worcester Polytechnic Institute, Worcester, MA, 01609.*
  E-mail: arana@wpi.edu
- *Katie Gandomi, Department of Robotics Engineering, Worcester Polytechnic Institute, Worcester, MA, 01609.*
  E-mail: kgamdomi@wpi.edu
- *Miguel Lasa, Department of Robotics Engineering, Worcester Polytechnic Institute, Worcester, MA, 01609.*
  E-mail: llasa@wpi.edu
- *Rahul Krishnan, Department of Robotics Engineering, Worcester Polytechnic Institute, Worcester, MA, 01609.*
  E-mail: rkrishnan@wpi.edu

In this project, we propose to implement and compare various AI agents to play Mario Bros. The code for Mario Bros has been taken from the MarioAI Benchmark competition held in 2009. This code provides us with the basic infrastructure that lets us focus on getting the state of Mario at a certain position and implementing the various AI agents.

Super Mario is a video game created by Nintendo with Mario as the main character. The game revolves around Marios adventures in the mushroom world where he must rescue Princess Peach from Bowser, the antagonist. The game requires the player to control the character - Mario, by moving right, left, down or jump, and navigate through all the obstacles and enemies to reach the end of each stage, upon which the player will be granted the entry to the next level.

The project proposes to implement various A.I. agents to play the Super Mario game and compare their performance. The following agents were used:

1) Forward Jumping Agent
2) Weighted Reflex Agent
3) A* agent
4) Neural Agent

Each agent was implemented separately one by one and their performance was compared.

```
Put node_start in the OPEN list with $f$(node_start) = $h$(node_start) (initialization)
while the OPEN list is not empty {
  Take from the open list the node node_current with the lowest
       $f$(node_current) = $g$(node_current) + $h$(node_current)
  if node_current is node_goal we have found the solution; break
  Generate each state node_successor that come after node_current
  for each node_successor of node_current {
    Set successor_current_cost = $g$(node_current) + $w$(node_current, node_successor)
    if node_successor is in the OPEN list {
      if $g$(node_successor) $\leq$ successor_current_cost continue (to line 20)
    } else if node_successor is in the CLOSED list {
      if $g$(node_successor) $\leq$ successor_current_cost continue (to line 20)
      Move node_successor from the CLOSED list to the OPEN list
    } else {
      Add node_successor to the OPEN list
      Set $h$(node_successor) to be the heuristic distance to node_goal
   }
    Set $g$(node_successor) = successor_current_cost
    Set the parent of node_successor to node_current
  }
  Add node_current to the CLOSED list
}
if(node_current != node_goal) exit with error (the OPEN list is empty)
```

Fig. 1. The A* Algorithm

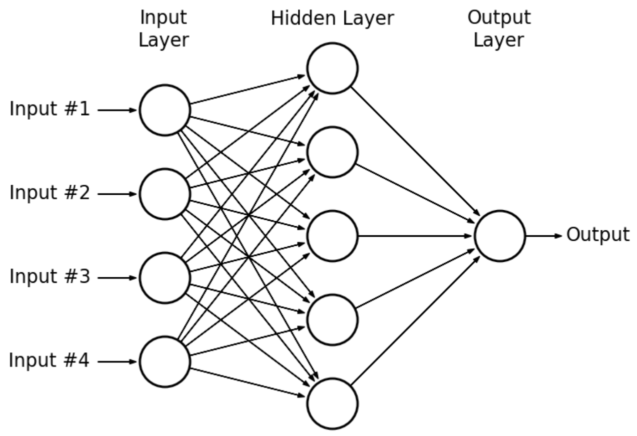

Fig. 2. A three layer Neural Network

2 Literature Review

2.1 A* Algorithm

A* is an informed search algorithm which is used to find the solution to a number of problems, finding the shortest and most profitable path to the destination being one of them. A* is an improved form of the Dijkstra algorithm. It uses heuristics for guidance rather than looking blindly.

Here OPEN list refers to the modes that have been visited but not expanded and CLOSE list refers to the nodes that have been visited and expanded. Node_start and node_goal are the starting and the end nodes.

The A* algorithm opens those nodes that lead to the goal node via heuristic function. This way the program is more efficient and fast. Calculation of the heuristic: f(current node) = g(current node) + h(current node)

Here f(current node) is the total weight of the current node, g(current node) is the cost of coming to this node and h(current node) is the heuristic value that tells the program if it is going closer to the goal node or farther away. Using this method, the A algorithm is able to open those nodes that lead to the goal node.

2.2 Neural Network

Neural Networks are one of the many scientific algorithms that try to replicate nature. It is a form of unsupervised learning that is inspired by the way biological nervous systems, such as the brain, process information. The neural network is composed of a high number of interconnected neurons (nodes) that work together to solve a specific problem which can range from finding the sum of any 2 numbers to finding the solution of a nonlinear problem to character and object recognition. Neural networks learn to solve a certain problem through experience and training, which can lead to very accurate results.

The input layer receives information from the environment, which it processes, recognizes and learns to interpret using the hidden layers. The Neural Network consists of one input layer, n number of hidden layers and one output layer. There can be any number of neurons in the input layers. The hidden layers connect the input layer and the output layer. They are the main processing plant of the Neural Network. The number of hidden layers can vary and depends on the complexity of the requirement. There can be more than one output node in the output layer depending on the application.

2.3 Genetic Algorithm

A genetic algorithm (GA) is an algorithm that mimics biological evolution and is used for solving optimization problems based on a natural selection process. The algorithm repeatedly modifies a population of individual nodes. At each step, the genetic algorithm randomly selects individuals from the current population

```
for all members of population
    sum += fitness of this individual
end for

for all members of population
    probability = sum of probabilities + (fitness / sum)
    sum of probabilities += probability
end for

loop until new population is full
     do this twice
         number = Random between 0 and 1
       for all members of population
           if number > probability but less than next probability
                then you have been selected
       end for
     end
     create offspring
end loop
```

Fig. 3. Genetic Algorithm pseudo code

and uses the ones with the highest fitness as parents to breed children for the next generation. Over successive generations, the population ”evolves” towards an optimal solution.

The genetic algorithm works like this: a population is created with a group of individuals created randomly. The individuals in the population are then evaluated. The evaluation function gives the individuals a score based on how well they perform at the given task. Two individuals are then selected based on their fitness, the higher the fitness, the higher the chance of being selected. These individuals then ”reproduce” to create one or more offspring, after which the offspring are mutated randomly. This continues until a suitable solution has been found or a certain number of generations have passed, depending on the requirement.

3 About Mario Environment

The Mario environment used in this project is a modified version of Marcus Persson’s *Infinite Mario Bros*. *(see figure 4)*

The Mario environment is a 22x22 array of bytes that contain information about the various graphics seen on the screen which include: sky, clouds, ground, gaps, pipes, coins, and the various types of bricks.

Mario is always at the center of the environment 2D array at position (11, 11).

The game play involves moving Mario in the 2D world using a combination of the following actions: left, right, jump, duck and speed. There are gaps, elevated surfaces and enemies in the way which Mario must jump over in order to avoid getting hurt and reach the end of the level. Mario has three states: Fire *(Large with the special ability to shoot fireballs)*, Large *(can break bricks)* and Small. In our code, Mario is in Fire state at the beginning of the game. He can shoot fireballs at enemies to kill them. *An explanation of Mario’s states when he gets hurt can be seen in Figure 5.* Mario can get hurt when he comes in contact with an enemy.

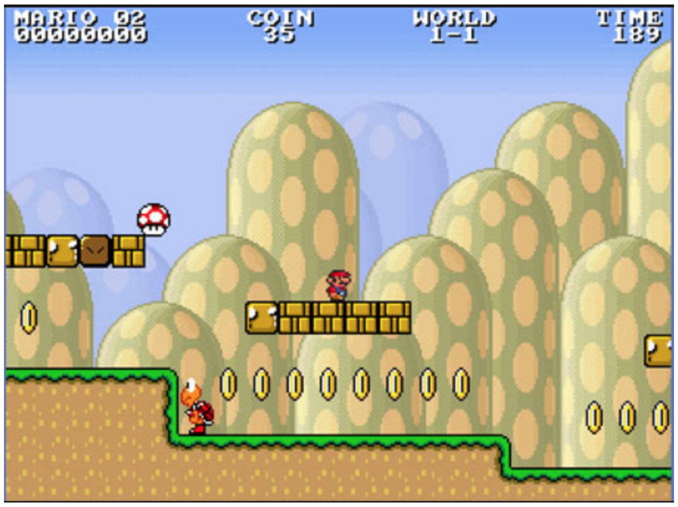

Fig. 4. Infinite Mario Bros

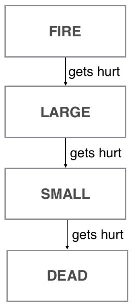

Fig. 5. Mario States when he gets hurt

The purpose of the game is to traverse from starting point to the destination place.

Mushrooms hidden inside some of the bricks can help Mario go up to the large state. For example, if he is in small state, finds a mushroom, and comes in contact with it, he will go up one state to Large. However, if he is large and comes in contact with another mushroom he will simply remain in the large state. Similarly, the flower item can be used to elevate Mario from the large state into the fire state.

4 Implementation

The team started by downloading and getting to know the MarioAI code and its working. For any algorithm we need an input and in our case the inputs were: Mario's state, state of environment and the position of enemies. Mario's state includes properties like: x and y position, distance covered, enemies killed, coins collected and damage to Mario's health. The environment observation is a 22x22 array of bytes that represent the environment. This array includes information about the various elements like: ground, gap position, position of pipes and elevated surfaces. The enemies observation is also a 22x22 array of bytes that provides information about the various kinds of enemies and their position. This is helpful when calculating costs of a particular child in a tree. Moreover, the information about the environment can be found using the environment observation variable: *Observation* which is an Environment object.

4.1 Forward Jumping Agent

The team was curious: what if Mario just jumped all through the level ? Would it still be able to complete the level ? If yes, then what score will he be able to achieve ? Will it perform better then any of the above agents ?

These questions led the team to implement the *Forward Jumping Agent*.

Mario has the following actions that are constantly activated, regardless of the level environment or enemy locations:

1) Right
2) Jump
3) Speed

After playing this agent several times, we found that Mario was able to complete the level 38% of the times when the difficulty was set to 1 and 25% of the times when the difficulty was set to 2. The levels were generated randomly for all agents*(with the exception of Neural Agent)*. This was then used as a baseline for which to compare the performance of the other agents.

4.2 Weighted Reflex Agent

The first approach we took was somewhat similar to that of a Reflex Agent with some differences. Since each level has many variables and elements, it was very inefficient to implement a "pure" reflex agent. This meaning that just the appearance of a single element on the screen is not enough to make an accurate decision.

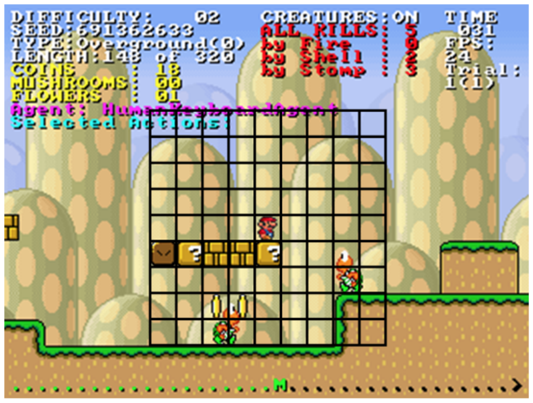

Fig. 6. Observation Grid around Mario's position

Basically, the idea is to analyze Mario's immediate surroundings and increase or decrease each possible action's cost depending on the observed data. First, we generate a Node that contains each possible action Mario could take at the given moment and initialize each action's cost to a default value. We generate these Nodes every time since there are some states where Mario is not able to take all actions.. After evaluating all observed elements of interest that are currently on the screen, all Nodes costs have been modified accordingly and we just command Mario to execute the action corresponding to the Node with the highest cost (the use of the word "cost" is a little counter-intuitive since in this context the higher the cost the better the action).

We considered "Mario's immediate surroundings" is a 22x22 matrix centered around Mario, being Mario's position (11,11).

The events that affect Mario's decisions are:

1) A gap in front of him.
2) Enemies above him.
3) Enemies straight in front of him
4) The presence of walls.
5) Combination of some of these.

When presented with any of these, Mario's action's costs get modified. In some cases, the preferred action is clear, in which case that particular action would get an increment in cost. In other cases, the preferred action could be any except for one or two, so in that case the undesirable actions get a penalty. For instance: if there is a gap, it is clear that in order to keep progressing in the level the next action should be to jump that gap, so jumping to the right gets its cost incremented. However, if there is an enemy detected somewhere above you then it is definitely bad to jump. In that case, any action involving jumping would get a penalty. It would also be bad to stay in the same place, but not as bad as jumping and directly hitting the enemy, so doing nothing would get a penalty, but smaller than the action jumping. Sometimes multiple events can happen at the same time: so if there is a wall, jumping gets rewarded; independently from that evaluation if there is an enemy detected above Mario, jumping gets a significant penalty as we specified before. Furthermore, since enemies can fall off walls, when the event ”wall and enemy above” happens, the action of doing nothing gets a penalty as well.

By default, only two actions start with a different initial cost. The actions of doing nothing or moving to the left (either running or jumping) suffer a slight penalty since the idea is to keep Mario moving forward when nothing else is happening around him.

Although the success rate of this agent is 52,6% (more than acceptable), and we can see it guide Mario through some complex situations, like having multiple enemies around him. There are certainly some improvements that could be done to make it more efficient. First, after detecting an enemy in front or above Mario, we could read what type of enemy we are dealing with and evaluate the action's cost again. For example, if we detect an enemy right in front, normally, the best evaluated action would be to jump. However, if the enemy is a flying turtle, what would probably happen next is that Mario jumps and meets the flying turtle mid-air. Another possible improvement would be dealing with hidden carnivore plants. When carnivore plants are hidden in their pipes, the enemy observation shows an enemy (even though there really isn't one there) and the independent level observation shows a wall. In order to make the agent better, if the enemy we detect is a carnivore plant, we could also check if there is a wall present in that same spot. If the check is true, the agent would just discard the enemy observation.

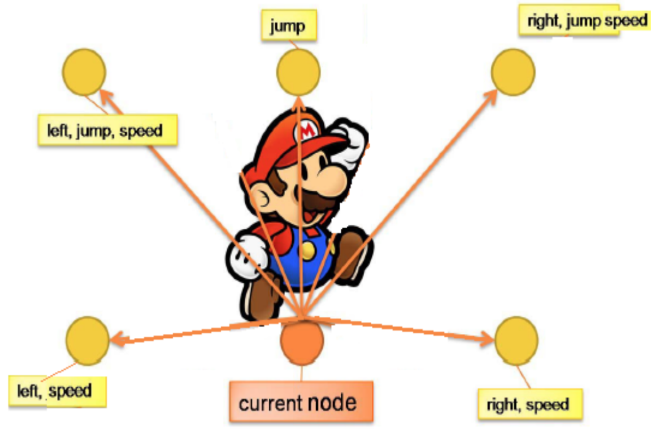

Fig. 7. Mario's Actions

4.3 A* Agent

A* search is a simple and fast algorithm for finding the shortest path and seemed like a perfect match for the Mario environment. It is also flexible enough to find routes through the generated levels in real-time [7].Our implementation of the A* search algorithm had a few key features: (a) Simulating future world states, (b) Planning till end of screen, i.e. goal node, (c) Re-planning path based on API constraints.

4.3.1 *Simulating Future World States*

Since the Mario API was open source, the physics engine was directly accessible and could be used to simulate future world states which were similar to the actual future world states. This simulation was used to find the effects of executing an action and finding its associated cost. The node with the lowest cost was added to the path. Simulating the states of enemies using the coordinates received from the API was particularly difficult. This was because the API only provides location and the speed of motion of the enemies. Since the simulation of the behavior of the enemies were not very accurate, there were times when our algorithm failed, especially in situations where there were multiple enemies nearby as well as a gap ahead. In such scenarios, the end result was either Mario receiving damage or jumping into a gap. Future work would include finding a way to more accurately simulate enemy behavior.

4.3.2 Planning till Edge of Screen

The A* search algorithm finds a path with the lowest cost between a pre-defined start node and one of many possible goal nodes [8]. Our algorithm uses a heuristic that determines the horizontal distance remaining to the goal. It also adds a penalty cost in case the associated action resulted in Mario receiving any damage. This heuristic has to be admissible (should not overestimate the distance) for optimality to be guaranteed [7]. However, in our case, this constraint is relaxed and a near-optimal solution also showed promising results.

To implement the A* search algorithm in the game, we define a node, its neighbors, goal node and how the heuristic works. A node is the entire world state at a given moment in time, including Marios state, position and environment information. We interact with this environment through actions associated with each node. These actions include, left, right, jump and speed. However, to speed up the algorithm we set the speed action by default to always true.

The Neighbors of a node are the world state after one simulation step, applying the action to the previous node. One possible area that causes an error in our cost estimation is that the speed of Mario at the current node can distort the position of all following nodes, thereby adding error in our estimated cost (using our current heuristic)[7]. It is also interesting to note that to create an admissible heuristic, we would have to take Marios current speed into account as well. Horizontal distance alone does not lead to a very accurate heuristic. This is because with our current heuristic, we assume that Mario is always at maximum speed, which is often not the case (even though the action *speed* is permanently set). To incorporate Marios current speed, we would have to determine the time left to reach the right border of the screen. Finally, our algorithm is slightly different from the traditional A* search algorithm. In our case, since we only plan till the edge of the current screen, our goal node is any node present on the right border of the screen. Since we know the width of the screen, we use the difference between Mario's position at the start of the search, i.e., the left border of the screen and Mario's current horizontal position. When that difference becomes greater than the width of the screen, we know that we have reached the goal node and we select the node with the lowest cost. Using this node and its corresponding parent nodes, we trace the path with the lowest cost back to the first node and send the associated actions with each of these nodes for execution.

4.3.3 Re-planning path based on API constraints

Certain rules of the API mean that some restrictions need to be put on the execution time of our algorithm, to stay within the allowed 40ms for each game update. Another factor is that only a certain window of information is visible at a time. Hence, every time new information is obtained, like presence of enemies or obstacles, recalculation is required. Through research and experimentation, we only plan for two game updates. Planning longer leads to actions based on incorrect information, leading to damage and/or loss of life. In short, we use the A* search algorithm to plan all actions from the start of the screen till the end of the screen, with accurate environment information available, but only execute two actions before re-planning is required because the window has now shifted and new environment information is available that deems the only path inaccurate.

The A* algorithm has a success rate of 67% and can be greatly improved. By modifying the heuristic to incorporate Marios current speed, as used by [7], can make the heuristic admissible (underestimate the distance) and possibly improve performance. By incorporating speed, we check the time remaining to goal node as opposed to distance remaining. This would guarantee an optimal solution and also remove any chance of falling into gaps or running into enemies.

4.4 Neural Agent

The team explored developing a learning agent that made use of neuro-evolution, a combination of neural networks and genetic algorithms, to help Mario complete levels of the game. In this case, many agents will be created with their own feed-forward neural networks whose weights are trained with a genetic algorithm.

4.4.1 Neural Networks

Neural networks are built using a number of interconnected simple building blocks called neurons or nodes. These neurons are built in multiple layers where : the first layer is used for receiving inputs to the system, the last layer is used for relaying outputs from the system, and any intermediate layers, referred to as hidden layers, are used to manipulate/propagate the input through the system and into the output. The goal of the neural network is to find an association between the inputs and the outputs based on patterns in the data, this is done by modifying the weights of the synapses in the hidden layer.

In this project there exists a NeuralAgent Class that contains a NeuralNetwork Object. The NeuralAgent uses its NeuralNetwork to determine what action Mario should perform based on a set of inputs. The inputs considered in this project and passed into the neural network were based on three main criteria : layout of the current view-able scene, nearby enemies, and Marios current state, i.e.: fire, large, or small, and ability to jump. These values were inputted one by one as binary variables, 1 meaning yes or occupied and 0 representing no or empty. The network had a total of 33 inputs. These inputs were then propagated through the system and a series of outputs corresponding to each of Marios core actions were outputted : Left, Right, Jump, Duck, Speed, Fire. In this way, if Mario received a 1 for traveling left and zero for all other options, he would travel to the left, similarly if a 1 was outputted for both right and jump, Mario would move to the right and jump simultaneously. Due to the nature of the neuroevolution algorithm implemented here, only a feed forward neural network was needed.

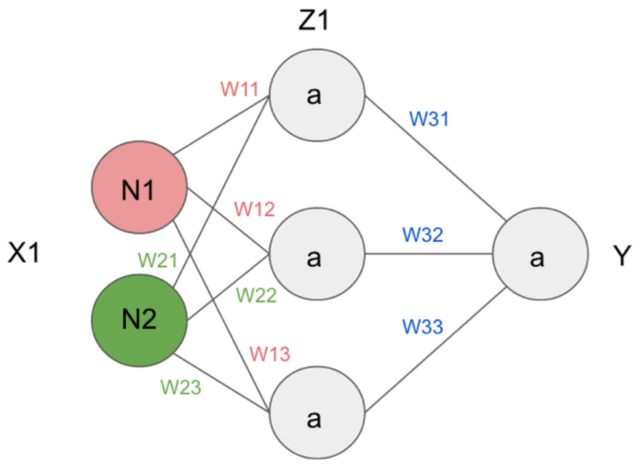

Fig. 8. Forward Propagation

Feed-forward neural networks are neural networks that only allow signals to travel in one direction. The values are propagated through the network from input all the way to the output. Lets consider a simple case shown in the Fig. 6.

The initial values put into the neural network are represented by the matrix X, such that the first column consists of entries to node 1 and the second column consists of entries to node 2,

$$X1 = \begin{bmatrix} x_{11} & x_{21} \\ x_{12} & x_{22} \\ x_{13} & x_{23} \end{bmatrix}$$

These inputs are then multiplied by the weight of each synapse represented by the matrix W,

$$W1 = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ x_{21} & x_{22} & w_{23} \end{bmatrix}$$

Using the following formula, the value for each hidden layer Z is computed as a sum of weighted inputs,

$$Z1 = X1 \times W1$$

$$\begin{bmatrix} x_{11}w_{11} + x_{12}w_{21} & x_{11}w_{12} + x_{12}w_{22} & x_{11}w_{13} + x_{12}w_{23} \\ x_{12}w_{11} + x_{22}w_{21} & x_{12}w_{12} + x_{22}w_{22} & x_{12}w_{13} + x_{22}w_{23} \\ x_{13}w_{11} + x_{23}w_{21} & x_{13}w_{12} + x_{23}w_{22} & x_{13}w_{13} + x_{23}w_{23} \end{bmatrix}$$

And then finally an activation function is applied,

$$a1 = f(Z1)$$

The most common activation function are threshold, sigmoid, piecewise linear, and Gaussian. In this project, a *Gaussian* activation function was used as they are more often used when finer control is needed over the activation range.

This process is then repeated until the output as follows,

$$Z2 = a1 \times W2$$

$$W2 = \begin{bmatrix} w_{31} \\ w_{32} \\ w_{33} \end{bmatrix}$$

After this step back propagation techniques are commonly applied such that the weights of the neural network synapses are tweaked to better match the training data. In this case, the team will not be doing back propagation, but will instead explore the use of a genetic algorithms for tuning weights.

4.4.2 Genetic Algorithm

In this project there is a GeneticEvolver Class, where the genetic algorithm that is used to train the weights of the NeuralAgents NeuralNetwork is performed. Genetic Algorithms operate under the principle of survival of the fittest, and the following section will walk through how this technique was applied to help Mario complete levels of the game.

At the start of the genetic algorithm an initial population of NeuralAgents is created each with their own NeuralNetwork, which is assigned random weights to start. Each member of the population is then run through a given Mario level and receives a fitness score based on their performance. The score is assigned to be the actual score the agent receives from the Mario game. After all agents are given a fitness, the agents are split up into pairs and a tournament style deselection process is done.

The stronger of each pair, the one with the higher fitness, is kept and allowed to breed children, while the weaker is removed from the population pool. Each remaining agent is then paired up with another agent and these two agents move on to the breeding stage. To keep the size of the population constant, each pair of parent agents must produce two offspring, lets call them Child A and Child B. These two children then have their weights assigned based on their parents weights as follows: Child A - Gets 1st half of Parent As weights and the 2nd half of Parent Bs weights, and Child B - Gets 2nd half of Parent As weights and 1st half of Parent B's weights. Immediately after each child is created, the weights of their neural network is mutated using the following procedure : each weight is multiplied by a random value pulled from a Gaussian, with mean zero and standard deviation 1.0, with a mutation value that controls how large the change in value is.

Finally after all children are created, this mix of both parent and children agents becomes the next population to run through the given level. A level is run until one agent completes the level, i.e. receives a score of 4000, or 1000 genetic cycles have passed. In this way the weights of agents who perform better are kept and those who perform poorly are removed, and Mario eventually learns how to play the level.

4.4.3 NeuralAgent Results

The NeuralAgent was trained over a series of training levels that ranged in difficulty from 0 to 5. After being trained, the NeuralAgent had a perfect success rate for completing levels it had already seen before and a 60% success rate for completing a level it had never seen before. The agent does very well in overcoming obstacles it had previously seen during the training phase, like jumping over a pipe or squishing an enemy, but it had a difficult time adapting to situations it has never seen, such as being ambushed by multiple enemies or the orientation of certain gaps. The average training time for tuning the NeuralAgent weights was between 8-10 hours. Some improvements to this Agent could be: to add more inputs to the Neural Network, use a different activation function, change the way the deselection process for removing agents is performed, edit the way the children agents receive weights from their parents, or change the mutation process done to the children agents.

5 Conclusions

Since the only objective of our approach was to get Mario to the end of the level, the criteria to classify the agents was one: their success rate.

From the results we see that the agents, from better to worse, were: A*, NNs, Weighted Reflex Agent and finally the Forward Jumping Agent.

We can attribute the high success rate of the A* algorithm to the fact that it plans two moves ahead, making ”wiser” decisions. However, the heuristic used is not optimal, still causing Mario to take non-ideal decisions. Working on the heuristic would definitely improve its success rate. The NeuralAgent did very well, almost perfectly, on levels it had seen before. However, its performance considerably dropped when presented with new situations and needed a significant amount of time to work its way around them. The best way to improve this agent's performance is to increase the number of inputs to the input layer. The Weighted Reflex Agent proved to be very robust, even in complex levels. The main weakness of this agent comes from the limited amount of elements that takes into consideration to make decisions. The more elements and situations described in the agent, the more robust it becomes. As expected, the A* and NeuralAgent did better than the Reflex Agent. Which, in itself, did better than the Jumping Forward Agent.

Evaluation Results

Agent Type	**No. of Trials Done**	**Trials Completed Successfully**
Forward Jumping	100	25
Weighted Reflex	100	52
A*	100	67
Neural Network	100	60

6 Scope for Future Work

For the project four A.I. agents were developed and compared. The following work can be done in future to improve the efficiency of the agents:

- In the Weighted Reflex Agent, more cases can be included so that Mario can make better decisions and avoid damage. This would take time and effort as there are innumerable such combination of cases, but if implemented, it would improve the success rate considerably.
- Better optimization techniques can be implemented for the Neural Network agent so that the training is more efficient.
- The number of hidden layers can be increased to implement deep learning for better success rate.
- The code for the A* agent can be improved to achieve higher accuracy.

7 Appendix

Code can be found at:

1) The Git Hub repository: $https://github.com/Kygandomi/MarioAi_2009/tree/master/src/a/cs534$

2) The submission package for CS 534 Artificial Intelligence course.

References

[1] T. Schaul and J. Schmidhuber. Scalable neural networks for board games.” in Proceedings of the International Conference on Artificial Neural Networks (ICANN). 2008.

[2] Ng Chee Hou, Niew Soon Hong, Chin Kim On, Jason Teo. Infinite Mario Bros AI using Genetic Algorithm. Evolutionary Comput. Lab., Univ. Malaysia Sabah, Kota Kinabalu, Malaysia. 2011.

[3] Leung, F., Lam, H., Ling, S., & Tam, P. (2003). Tuning of the structure and parameters of a neural network using an improved genetic algorithm. IEEE Trans. Neural Netw. IEEE Transactions on Neural Networks, 14(1), 79-88. doi:10.1109/tnn.2002.804317

[4] Liao, Y., Yi, K., Yang, Z. (2012). Cs229 final report reinforcement learning to play Mario (Doctoral dissertation, PhD thesis, Stanford University, USA).

[5] Taylor, M. E. (2011, August). Teaching reinforcement learning with Mario: An argument and case study. In Proceedings of the Second Symposium on Educational Advances in Artificial Intelligence (pp. 1737-1742).

[6] Stergiou, Christos, and Dimitrios Siganos. ”Neural Networks.” Neural Networks. N.p., n.d. Web.

[7] Julian Togelius, Sergey Karakovskiy and Robin Baumgarten The 2009 Mario AI Competition , Mario AI Competition, CIG 2009

[8] A formal basis for the heuristic determination of minimum cost paths, IEEE transactions on Systems Science and Cybernetics, vol. 4, no. 2, pp. 100107, 1968.

[9] Li Deng, Li, and Dong Yu. Foundations and Trends R in Signal Processing. 3-4 ed. Vol. 7. Richmond, VA: Presbyterian Committee of Publication, 1861. 197387. Foundations and Trends R in Signal Processing, 4 Mar. 2013. Web. 4 Mar. 2013.