Using Genetic Programming to Evolve Heuristics for a Monte Carlo Tree Search Ms Pac-Man Agent

Atif M. Alhejali, Simon M. Lucas
School of Computer Science and Electronic Engineering
University of Essex
Colchester, UK
amalhe@essex.ac.uk, sml@essex.ac.uk

***Abstract*— Ms Pac-Man is one of the most challenging test beds in game artificial intelligence (AI). Genetic programming and Monte Carlo Tree Search (MCTS) have already been successful applied to several games including Pac-Man. In this paper, we use Monte Carlo Tree Search to create a Ms Pac-Man playing agent before using genetic programming to enhance its performance by evolving a new default policy to replace the random agent used in the simulations. The new agent with the evolved default policy was able to achieve an 18% increase on its average score over the agent with random default policy.**

Keywords—genetic programming, Monte Carlo Tree Search, Pac-Man.

I. INTRODUCTION

Monte Carlo Tree Search (MCTS) is a random sampling tree search method that creates a partial game tree and then searches for the best move based on its approximated game-theoretic value, which can be obtained from random simulations carried out during the search[1]. MCTS came to the attention of researchers after its success in computer Go [1-4] in 2006 [1]. Thereafter, MCTS became one of the favorite methods for creating game-playing agents due to the nature of many games, which were easily represented as trees, and due to the ability of MCTS to provide competitive solutions. One of the games previously examined using MCTS was Pac-Man. We used MCTS because of its significant success in computer games and in Pac-Man as illustrated by the work of Samothrakis et al [5]. Genetic programming (GP), on the other hand, is an evolutionary, computational algorithm that uses the theory of natural selection to evolve solutions to the problem at hand in the form of trees [6, 7].

In this paper we use MCTS to build a Ms Pac-Man playing agent before attempting to enhance it with a better heuristic created with genetic programming. We aim to investigate how GP can be used to strengthen MCTS using a challenging real-time game as a benchmark

The structure of the experiment was to build the hand-coded MCTS Ms Pac-Man agent and then use GP to evolve a playing agent that can be used to run the simulations before testing the final outcome and then compare the results.

II. BACKGROUND INFORMATION

A. Ms Pac-Man

Ms Pac-Man is a predator-prey arcade game that was released as a second version of Pac-Man in the early 1980s. The game consists of a maze with paths and corridors that the Pac-Man moves through collecting food pills that fill some of these paths. The aim of the game is to control the Pac-Man in order to clear all the pills in the current maze and then advance to the next one. During the game, the Pac-Man is chased by four ghosts any of whom will kill the Pac-Man if they are able to catch him. The ghosts behave in a non-deterministic way, which makes it hard to predict their next move although their general behavior varies from random to very aggressive. Near the corners of the maze lie four power pills or energizers. If the Pac-Man eats any of these power pills, the four ghosts will become edible for a short period of time and will change their behavior from chasing the Pac-Man to escaping from him. Eating these edible ghosts will increase the Pac-Man's score dramatically because eating the four ghosts after consuming a single power pill will be awarded with 3,000 points which means that the agent can score up to12,000 points if the Pac-Man eats all of them four times. In contrast the food pills score only 2,000 to 2,500 points depending on the current maze.

B. Genetic Programming

GP is a traditional, evolutionary computation algorithm, which means that it evaluates a randomly generated population of trees. It then uses the best programs in this population as parents to create the next generation by using crossover, mutation, and reproduction before starting the cycle again by evaluating this new generation.

Algorithm 1 shows how a default GP system works. The first population is created randomly from the list of functions and terminals provided by the programmer, and it follows the programmer's specified grammars. These grammars can place any constraints on the creation process, by, for example, limiting the tree's size or depth in order to prevent the tree from growing in an unlimited manner and reaching a level that the computer cannot handle. The function set serves as the genes the system will use to generate the programs, and they are usually hand coded. Each one of them is a small program in itself that return a value that can be used by another function and may take arguments. These arguments can be returned by another function or terminal. The difference between functions and terminals is that functions must have children that can be

978-1-4673-5311-3/13/$31.00 ©2013 IEEE

either other functions or terminals, while terminals are the trees' leaves and hence they cannot be connected to children. Each function has an arity value that is the number of children it should have. Hence, the arity of the terminals is always 0.

1. Create the initial population.
2. Evaluate the initial population.
3. Repeat for the required number of generations:
   a. Select the parents.
   b. Create the offspring, using:
      i. Crossover.
      ii. Mutation.
      iii. Reproduction
   c. Evaluate the offspring.
   d. Select the survivors.
   e. Create the new population from the survivors.
4. Present the final solution.

Algorithm 1: Genetic Programming Algorithm

C. Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a random sampling method that uses a best-first search technique to find the optimal decision for a domain that can be represented as a tree [1, 8, 9]. This tree contains all the possible future states of game (or problem) at hand in the form of nodes, and a reward value is assigned to each node that is based on random simulations that originated with this node and that have included all of its descendants.

An MCTS algorithm starts by building a game tree in a step-by-step progression, and whenever a leaf node is reached, a random simulation will run to find one of the possible scenarios starting from this point. The algorithm is usually summarized in four steps: expansion, selection, simulation, and back-propagation [1, 8, 10].

Expansion. The algorithm starts by building the tree from its root, which is the current state of play. The first step is to expand the root by adding children that represent all the states that can be reached in the next step. The expansion of the tree will continue any time a leaf node is selected. A leaf node is the node that does not have any children although it is not terminal.

Selection. When the search reaches any non-leaf node, one if its children must be selected to continue the search. MCTS is a first-best search method, which means it always favors the most promising node unless there is a node that has not been visited before. The most popular MCTS algorithm is UCT [11] (Upper Confidence Bounds in Trees) [5], which treats the tree as though it is a multi-armed bandit problem [1]. This method depends on the reward value attached to each child and how many times it was visited before. Traditionally, the UCT value that determines which child is selected is calculated using UCB1.

$$UCB1 = x_j + \frac{\sqrt{\ln(n_p)}}{n_j} + \mathrm{r}\ c \quad (1)$$

In this equation, x_j represents the average reward that this node j gained from previous visits to it, while n_j is its number of visits. n_p is its parent number of visits, and c is a very small constant, while r is a random value that will break a tie if two nodes have the same value[1, 5].

guarantees that the unvisited nodes are chosen first because n_j will be 0 and hence the value will be ∞. Otherwise, the nodes with the most promising reward will be favored.

These previous two steps are known as the tree policy [1].

Simulation. When a leaf node is visited for the first time, a random agent will continue solving the problem. In a computer game, a random player will start from this point until the either the game has finished or the playing budget has reached its limit (e.g., a time limit). This simulation will return a value that can be 0 for losing or 1 for winning, or a continuous number such as the score obtained. This value will be added to the node as its reward from the simulation. This step is generally known as a rollout or playout and the playing policy during the simulation is the default policy according to the terminology in [1] .

Back-Propagation. The value that has been obtained through simulation is a reward that should be added to the current node and all its ancestors up to the root of the tree. Back-propagation means that the algorithm will follow the line of parents from the root to the current node and update their reward value as a result of this simulation. This means that the value of any given node in the tree is the aggregated value (sum of values) of all the nodes in the tree that originate from it.

After finishing all of the simulations, the next decision or move is chosen from the children of the root ,and the choice is the node, or child, with the best total value, the most number of visits, both, or the best UCB value [1].

1. repeat N times 2. select 3. if not a leaf 4. go to one 5. expand 6. select 7. rollout 8. back-propagate 9. choose	**Select**: select one of the available children of the current node using its UCT value, which can be calculated from its total value. **Leaf**: a node with no children. **Expand**: create the children of the current node which are the possible next states arising from the current state (node). **Rollout**: use the MC simulation to find a possible value that can be reached from the current state. **Back-propagate (update)**: add the value found by the simulator to the current node and all its ancestors. **Choose**: choose the best next move from the moves available to the root according to the total values (or average values) of its children

Algorithm 2: Monte Carlo Tree Search Algorithm

III. PREVIOUS WORK

Many attempts have been made to create Pac-Man controllers using genetic programming, though as with many Pac-Man experiments, most have used different simulations of the game, so most of the scores quoted below cannot be directly compared with each other. John Koza was the first to try, using his own simulator of the game [6]. He was able to score 9,220 points out of a possible 18,220 in one trial. More recently, Brandstetter and Ahmadi [12] conducted an experiment in which they used genetic programming to evolve a Ms Pac-Man agent. Their approach was based on using basic commands such as up, down, left, and right to direct the Pac-Man in the generated tree instead of using complex terminals that would direct it to a variable target (such as a ghost). Their

agent was able to score an average of over 19,000 points in a simulated version of the game. Rosca [13] studied generality versus size in genetic programming using Pac-Man as a test bed.

The authors of this paper (Alhejali and Lucas) performed two previous experiments to evolve a Pac-Man agent using GP. In the first one [14], they tested how changing the evolution environment can affect the final outcome. They performed several tests with different numbers of maximum levels that the Pac-Man can play (1, 4, and unlimited). Their final findings proved that choosing the training environment is very critical, especially when the real environment cannot be used. In the second experiment [15], the authors proposed a new technique that is based on problem decomposition. In this technique, the problem is divided into several smaller problems that can be solved using independent GP runs before combining the evolved solutions into a final run in order to create the final agent. This technique proved its superiority over standard GP by generating better solutions in less time.

On the other hand, tree search has also been popular with Pac-Man. Robles and Lucas [16] used a simple tree search method to create a Ms Pac-Man agent that achieved an average score of 9,630 in the original screen-capture game, with a maximum score of 15,280. Their agent was able to score over 43,000 points, with an average score of 14,757 on a simulator. Samothrakis, Robles, and Lucas [5] used MCTS to play Ms Pac-Man. After a set of experiments, they tested their controller with three setups. The first one was the known model, in which the agent knows exactly which ghost team is against him, which means the moves of 3 out of the 4 ghosts are known all the time. The second and third tests were on the unknown model, in which the agent did not have any information about the strategy of the ghosts. The difference between these tests was that the first ran in real time mode, which meant that the agent had a time limit assigned to responding to a move (about 30-40 milliseconds). In the other test, the agent could take all the time it needed to explore the tree in greater depth. In the known model, the players achieved a maximum score of 2.8 million with an average score of over 500,000. In the unknown, real-time model, they scored a maximum of over 200,000 points with an average of over 45,000 and nearly the double of that in the final model. Ya'nan et al. [17] used dynamic difficulty adjustment (DDA) with MCTS to create "more-interesting non-player characters (NPCs)" – in this case, a team of ghosts. Their work focused on using DDA on a team of ghosts created by MCTS in order to adjust their intelligence level so they were not too stupid, which could make the game boring, or too smart, which could make the game too difficult. Xiao et al. [18] used MCTS to create a team of ghosts. They focused on the computational time required to create the agents rather than focusing on the agents' performance. The Monte Carlo controllers provided better agents, while the compared technique of using a neural network performed better in terms of resource consumption.

Tong and Sung [19] used Monte Carlo simulations with Ms Pac-Man to escape a deadly situation. Their agent used Monte Carlo simulations to look ahead and find the best path to escape when it was surrounded by one or more ghosts and could be killed within a few time steps. They were able to score an average and maximum score that was nearly double what a greedy controller scored with 12,872 as an average and 22,670 as a maximum score. Their agent played in the original game's screen capture mode, which reduced the amount of time allowed per move. To overcome this problem, the agent simulated the next move, starting from its current position.

Tong, Ma, and Sung [20] also used Monte Carlo simulators to find the safest path to clear the final few remaining pills in a maze in the game of Ms Pac-Man. In their work, they compared two agents with a single difference. Near the end of each maze, the agent usually faces the problem of having to eat the remaining pills that can be fragmented around the maze while being chased by the ghosts. Their first agent used an algorithm that determined the shortest path from the current Pac-Man location to each remaining pill according to how safe it was. The second technique included using the Monte Carlo simulators to determine the safest path to the nearby pills. In their final finding, the Monte Carlo agent proved to perform better, with an average score that was 20% higher than that of the other agent.

Nguyen and Thawonmas [21] used MCTS to create a ghost team. Each ghost used its own game tree with a specified, limited depth. In the simulations, the ghosts moved randomly while the Pac-Man moved according to simple rules that made it choose a random exit at a crossroad and never reverse course. If there was a ghost at a certain distance in front of the Pac-Man it reversed, while if there was a power pill it went straight to it. This ghost team won the CEC 2011 Pac-Man vs. Ghosts Team competition [22] with an average score of 11,407 obtained from all of its matches against all of the Pac-Man entries. (The second and third teams scored 13,025 and 13,594 respectively). Also, Ikehata and Ito used MCTS to create a Pac-Man agent that outperformed the CIG 2009 [23] winner, ICEPambush3 [24]. The game tree in this agent was different from what is usually used. Instead of having the game states and actions be the components of the tree, they created a tree in which the cross nodes in the maze were the nodes of the tree and the straight paths between them were the links. In their tree, the root was the current crossroad for the Pac-Man, which was the next crossroad the Pac-Man would have reached when following its current direction (their Pac-Man was not allowed to reverse). Starting from this point, the next level of the tree was all of the neighbors' cross points to the current one, and each one of them was connected by a straight road that could contain a corner but could not contain any exits. Using this tree and running simulations in each leaf, they were able to achieve an average score over 24,000 with a maximum score of 37,000 while ICE pAmbush achieved around 30,000 as a maximum score and 20,000 on average in the researchers' tests.

In the CIG 2009 Pac-Man vs. Ghosts competition [23], the controller ICE pAmbush was created using a combination of a rule-based system and MCTS. The controller had a set of 4 rules that were tested in order. The first three rules focused on creating an ambush for the ghosts near a power pill. The fourth rule could be reached only if the first three failed, and it would run MCTS to determine the next action [25]. This approach scored an average of 20,009, granting it third place in the competition. Pepels [26] also used MCTS to create a Pac-Man playing agent that ranked second in the WCCI 2012

competition [27], with an overall average score of 87,431. In another experiment, Nguyen and Thawonmas [28] used a combination of Monte Carlo Tree Search and a rule-based approach to create a ghost team that won first place in the CEC 2011 Pac-Man vs. Ghosts competition. Their team, named ICE gUCT, had a score of 16,436 against the winning Pac-Man agent which was able to score over 21,000 against all the other ghost teams in the competition.

IV. EXPERIMENTAL SETUP

A. The MCTS Controller

The idea of the controller was very simple. At every time step the agent was called and was supposed to respond by indicating the direction of the Pac-Man's next move. When called, the agent ran the MCTS algorithm to build a partial game tree and then ran the required simulations according to the MCTS method in order to find the ideal next move.

As mentioned before, the MCTS algorithm uses four steps: expansion, selection, simulation and back-propagation. Completing these four steps was considered a full cycle at the tree. Next, we will discuss how each of these steps was implemented in our controller.

Expansion:

The first thing that needs to be decided when building a game tree is which rules will be followed when expanding the next level in the game tree and how to select the next node, which is the tree policy[1]. In our implementation, the tree could explore all of the possible moves only when it was creating the first level because the Pac-Man can move in all directions. At the next level in the tree, the reversed direction was not investigated. This meant that the Pac-Man and the ghosts had the same policy starting from the second level in the tree. This restriction was made to prevent the tree search from wasting time moving the Pac-Man back and forth in the same place, which would increase the probability of moving it the same way in the real game and require far more computational time. Another decision that needed to be made arose from the fact that Pac-Man is a two-player game, although the opponent is a team of NPCs (non-player characters). In MCTS, each level should contain all of the possible game states that can be reached next and all the possible moves that could be made by the player. Hence, the movement of the ghosts needed to be considered even if the focus was only on the Pac-Man. In the real game, these moves happen simultaneously, which is different from the board games that MCTS succeeded with where a player makes a move and then the next player makes his move. This problem has been solved with several versions of MCTS for simultaneous-move games such as the work of Shafiei et al. [29]. However, the best and simplest solution is to build the game tree as though the Pac-Man and ghost moves are sequential and not simultaneous, creating the Pac-Man moves on one level followed by the ghost moves on the next. Samothrakis, Lucas, and Robles proposed a similar technique in Tron [30], while Xiao et al. used the same method in Pac-Man [18] but in reverse ,since they considered the move of the ghosts first and then the moves of the Pac-Man.

Finally, when building any game tree, the nodes of the tree will be either terminal or non-terminal. In our version of MCTS for Pac-Man, the node was terminal if the Pac-Man was killed in it, regardless of whether this was his last life or not, or if he cleared the current level and advanced to the next. The non-terminal nodes were nodes where the Pac-Man could still move on to the next step and they must have children which were the next possible states.

Selection:

MCTS is a best-first search technique that focuses the search on the part of the tree that shows the most potential. This means that in a game like Pac-Man, the best move is the one that will gain the Pac-Man the best score without his being eaten by a ghost. In order to do that, the search should consider the best move for the Pac-Man against all the possible moves of the ghosts and especially against the best move (or set of moves) made by the ghosts. This important since the most powerful ghosts have an aggressive behavior and hence it is beneficial if the Pac-Man assume that it is facing the best possible ghosts' team. This can be done easily with our implementation of the game tree where, at the first level (Pac-Man moves level), the search will choose the best node for Pac-Man while at the next level, the search will go towards the worst node for Pac-Man.

For selection, MCTS uses the UCB (Upper Confidence Bounds) to calculate a UCB value for each child, and the child with the highest value will be chosen. The main equation used is UCB1, which has worked with Pac-Man successfully before [5, 24, 28].

$$UCB1 = x_j + \frac{\sqrt{\ln(n_p)}}{n_j} + \mathrm{r}\ c \quad (2)$$

In this equation, x_j represents the average reward that this node j gained from previous visits to it, while n_j is its number of visits. n_p is its parent number of visits, and c is a very small constant, while r is a random value that will break a tie if two nodes have the same value[1, 5].

This method maximizes the reward collected by nodes, which means that it is perfect for determining which move is best for the Pac-Man. In order to choose the best moves for the ghosts, the method needed to be reversed. The first way to reverse it is to minimize it by choosing the child with the lowest UCB value. This idea will not work because UCB1 gives the children that have not been visited before extremely large values to ensure that they will be visited first. If the algorithm minimizes the UCB1 value, then the first child will be selected randomly the first time the parent is expanded, and then the same child will be selected every time. A simpler and more efficient way is to reverse the average reward x_j. The reward given to any node after simulation is always between 0 and 1, which means that the average x_j is also between 0 and 1. In this case, if x_j in the last equation is subtracted from 1, then maximizing the UCB value will mean choosing the worst node for Pac-Man and as a result the best node for the ghosts.

$$reversed\ UCB1 = (1 - x_j) + \frac{\sqrt{\ln(n_p)}}{n_j} + \mathrm{r}\ c \quad (3)$$

Simulations:

In games, simulations usually run randomly from the game state represented by the node that started the simulation until

the end of the game. The reward for this random game will be 1 for winning, 0 for a draw and -1 for losing [14], or 1 for winning, 0.5 for a draw and 0 for losing [14, 165]. This rewarding system is not valid for Pac-Man because the death of Pac-Man is the final outcome of any game regardless of its success, which is measured by the score. In order to remedy this, we determined that the reward would be the total score the agent might gain from the root of the tree until the end of the simulation. This score was divided by a large number such as 5,000 to guarantee that it would be smaller than 1, because UCB is usually used with average rewards if between 0 and 1 (although in can be tuned to handle larger values) [14]. In addition to that, after several tests another modification was made to the reward system by dividing it into two parts. The first part was the score as explained here and it represented 50% of the total reward, while the other 50% was added if the agent survived to the end of the simulation. It was not added if the agent died before the end.

$$R_i = \frac{S_i}{c} * 0.5 + 0.5 \times d_i$$

In this equation, R_i is the reward for rollout i while S_i is the score obtained in the current cycle on the tree i from the root to the end of the simulation. c is a large constant (e.g., 5,000) to ensure that the outcome is less than 1. d_i is the death status for Pac-Man in the current iteration i, which is a boolean value of 1 if the Pac-Man survives and 0 if it dies before the end of the simulation.

The last part in the simulation is the default policy, which is the way the agent behaves during the simulated game. In our main tests, we used the random non-reverse agent, which is a random agent that always moves forward and chooses a new random direction only at a junction and that can never reverse back. As for the ghosts in these simulations, we used the random ghosts' team so the Pac-Man would not have any idea how the ghosts would behave in the real game regardless of what ghosts it was facing.

Back-Propagation:

After the end of the simulation and the calculation of the reward, the final step was to update the relevant nodes with this reward. The relevant nodes were the one that triggered the simulation and all its ancestors up to the root of the tree. The algorithm started with the current node and moved up level by level, adding this reward to each nod's total reward value and increasing its number of visits by one.

At the end of the final cycle, the algorithm chose the best next move from the children of the root. In general, the way to choose the next move in MCTS is by selecting the child with the greatest total value, or number of visits, or both, in addition to choosing the child that maximizes a lower confidence bound [1, 8]. In this agent, and all the MCTS controllers we created, we used the first method, with the highest total value, although testing the second and fourth methods did not produce any significant change in the results.

B. The Evolution in MCTS Pac-Man

The previous section described a complete working MCTS playing agent controller. We tried to enhance it by using genetic programming to evolve a new default policy to be used instead of the random agent that was used in the simulations. This new policy took the form of an agent that could play the game using information from the current game state. An agent that uses information from the game and moves according to this information creates new simulations that Drake and Uurtamo called heavy playouts [1, 31]. This evolved agent was expected to be extremely fast and to have a performance time that was similar to a random controller in order to keep the computational time, which was already high, at its minimum. Apart from that, the agent could be in any form and length and could follow any technique and policy with or without a random component and use any information available about the current game state.

The GP system used in this experiment was a newer version of the system used in earlier studies [14, 15]. At the beginning of the GP run, a complete new population is generated randomly and each one of them is passed to the agent to be evaluated. In the evaluation, the agent runs on the simulator using the evolved tree at each rollout to run the simulator. After evaluating of all the individuals, the new generation created using crossover and mutation as well as reproduction before starting the whole cycle again by evolving the new population.

The GP Setup

The function set:

The GP system is a strongly-typed GP with a function set divided into the seven categories listed below.

The non-terminal functions:

IFTE. This is the main IF ELSE statement that is used as the base of the tree. This function requires three children. The first one receives TRUE or FALSE to direct the search to either the second or the third branch, which should return a positive integer of between 0 and 4 as the next direction for Pac-Man.

Numerical Operators. This category consists of the main mathematical operators such as + and −. They take two numerical values and return the result of the operation they perform as a numerical value as well.

Comparison Operators. This includes >, <, ≥, ≤, ≠, and =. These functions compare two numerical values from either numerical terminals or operators and return a logical value.

Logical Operators. This category contains two functions, AND, and OR. They take any two children with logical returned values and return a logical value.

Terminals:

Action Terminals. These terminals direct the Pac-Man towards its target. When called, each one of them returns a single integer value that represents the direction to the next node along the shortest path leading to the target. These terminals can only be parented by IFTE. The terminal list includes the original random, non-reverse agent. The following is a list of the action terminals used in this experiment.

(*To1stEdibleGhost, To2ndEdibleGhost, To3rdEdibleGhost, To4thEdibleGhost, RandomNonReverse, ToEnergizer, ToPill*)

The Logical Terminals. The logical terminals answer a simple question about the current state of the game with TRUE or FALSE. They can be children of either IFTE or logical operators. The category consisted of these three terminals.

(IsEdible, IsEnergizersCleared, IsInDanger)

The Numerical Terminals. These terminals return numerical data about a single object in the game from the current game state such as the distance between the Pac-Man and one of the ghosts or the remaining edibility time. This category consists of 27 terminals, such as:

(*DIS1stEnergizer, DIS1stGhost, DIS1stInedibleGhost, DIS1stEdibleGhost, Constant, DISPill, EdibleTime*)

The experiment was performed in two stages primarily because of the extremely long time that was required to complete a single GP run. In the first run, the GP had 100 individuals and 50 generations to determine whether there was any potential for evolving better agents. In the second stage, it had 500 individuals and 100 generations. In addition to the normal parameters in this experiment, we had to decide on the maximum number of rollouts allowed at each MCTS run as well as the maximum number of time steps allowed in the simulation, which were set to relatively small values in order to reduce the evolution time as much as possible. The following is a summary of all of the GP parameters.

- Population size: 100 , 500
- Number of generations: 50 , 100
- Mutation probability: 0.2
- Initial population creation: rumbled-half-and-half
- Parent selection method: tournament (size 3)
- Maximum tree depth: 8
- Pac-Man given lives: 1
- MCTS number of rollouts: 30
- MCTS maximum steps in simulation: 20

V. Results and Discussion

All the controllers can be tested in several modes and with several settings. Changing the total number of rollouts, the maximum depth of the tree, and the number of steps allowed in the simulations can all change the results. In addition, any change in the rewards calculation method will have its impact as will changes in the Pac-Man and ghosts controllers used in the simulations. In the following sections, we will discuss the results of various settings and compare the results.

All of the tests were made on the same simulator from the CIG 2011 competition [23] with the original setup. In these games, the Pac-Man had 3 lives from the start of the game, and an additional life was earned if the score reached 10,000. The ghost team was the Legacy Team with 4 ghosts and edibility time beginning at 200 time steps and decreasing at every level.

A. The Hand-coded MCTS controller

In order to have a clear view of the performance of this controller, we performed a series of tests using the same game setup and various MCTS parameters. All of the changes made in the number of rollouts and the maximum number of steps allowed in each simulation and in each category were tested for 100 trials. In these tests, we did not make any changes in the maximum depth of the tree, which was unlimited given the tree's ability to expand vertically and accommodate the maximum size the rollouts could reach in order to explore the game as much as possible.

At first, we tested two different numbers of rollouts, being 100 and 500. The maximum number of steps allowed was 50. Table I clearly reveals that 500 rollouts outperform 100 with an average score of over 28,000 compared to 19,000 achieved with 100 simulations. An unpaired t-test performed between the two experiments showed that the results are significant with a p value of less than 0.0001. This test was important in proving that the controller was working correctly since it is known that more simulations mean better results in MCTS [1].

Table I. Result of testing the agent with different number of rollouts. Test results over 100 games played in each category

Number of rollouts	Average score	Minimum score	Maximum score	Standard deviation
100	19,154.70	4,360	45,710	9,850.10
500	28,116.6	3,990	62,630	12,628.86

The next test was on the maximum number of steps, in which the agent was allowed 10, 30, 50, 70, 100, and 150 steps in each simulation, while the number of rollouts was fixed at 100. Table II shows the results of this test.

Table II. Results of testing different maximum number of steps allowed in the simulations. Test results with 100 rollouts and over 100 games

Number of steps	Average score	Minimum score	Maximum score	Standard deviation
10	18,135	4,440	42,170	9,554.86
30	19,001.70	4,830	56,570	9,051.83
50	19,154.70	4,360	45,710	9,850.10
70	17,963.50	3,650	41,230	9,141.46
100	17,266.20	2,770	50,540	8,444.25
150	15,613.50	2,290	32,960	7,586.27

It is clear from Table II that 30 and 50 steps produced the best average and maximum score, although 10, 70, and 100 did not fall far behind. In fact, an unpaired t-test showed that the results of the first 5 categories in the table were not significantly different from each other. The only statistical significant was the result of testing the final row in the table, which is 150 steps. This category was proved to be statistically significant when compared to 10, 30, 50, and 70 steps. These results indicate that a longer time in the simulator does not always provide better results, with 50 steps providing the best average score and 150 providing the worst. Similar findings were reported by Xie and Liu [32], who found that "shorter simulations tend to be more accurate than longer ones"[1]. However, it was decided after seeing these results to repeat the test on a larger scale. The reason for this was to determine whether allowing more steps on the simulator was always worse or whether it depended on the number of simulations performed. This was because the agent used was random and it was possible that it required more iterations for the longer simulations. In the next test, we allowed the simulations to run for 50, 100, 150, and 200 steps in each simulation, with 500 rollouts.

Table III. Results of testing different maximum number of steps allowed in the simulations. Test results with 500 rollouts and over 100 games

Number of steps	Average score	Minimum score	Maximum score	Standard deviation
50	30,822.3	5,580	76,720	13,103.79
100	28,633.80	4,930	58,250	12,068.33
150	26,276.80	4,100	65,100	11,662.49
200	23,492.50	2,980	50,920	10,136.77

The results in Table III clearly indicate that 50 steps are still the best choice. As in the previous test, 50 and 100 steps results did not reveal any significant difference, while 150 and 200 were statistically significant compared to the 50 steps and using *t*-test. This proves Xie and Liu's theory regarding the superiority of shorter simulations [33].

B. The Controller with Evolved Default Policy

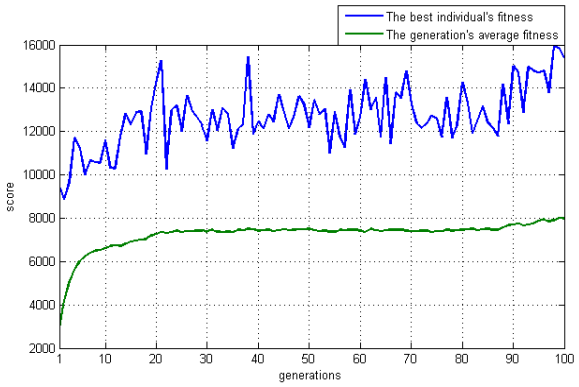

Figure 1. The evolution of a Controller for the MCTS Default Policy with 500 Individuals and 100 Generations

In order to evolve a controller that could be used in the MCTS simulation instead of the random agent that was originally used, we performed 7 different GP runs. In the first 5, the population size was set to 100 with 50 generations. After examining the results, two extra runs were performed with 500 individuals and 100 generations. The best controller from each run was added to the MCTS controller that was previously built and tested.

Although MCTS were given only 30 rollouts when tested on a GP run, to reduce the computational time, a single run with 100 individuals and 50 generations required an average of over 58 hours to complete. The 500 individuals and 100 generations version required approximately 18 days to finish a single run. Figure 1 shows an example of the evolutionary process.

After creating all the controllers using the trees evolved with GP, a series of experiments was performed that was identical to the one described in the previous section. The purpose of the first experiment was to find out whether the controllers with evolved default policy could outperform the controller we hand-coded using the random, non-reverse Pac-Man as a default policy.

Table IV. Test Results for All Controllers in Default Policy Evolution for 100 Games, 100 Rollouts, and 50 Steps

Controller no	default policy	Average score	Minimum score	Maximum score	Standard deviation
1	Evolved with 100 individuals and 50 generations	20,129	5,060	54,990	11,409.77
2		22,625.10	4,990	45,590	9,765.60
3		20,015.60	3,820	45,740	9,615.46
4		17,011.70	3,320	48,110	8,611.05
5		18,307.70	4,540	47,660	9,684.82
6	500 inds 100 gens	22,119.70	3,530	53,400	10,034.04
7		19,836	4,700	45,270	9,195.08
8	H-Coded	19,110	4,680	43,490	9,699.28

Table IV shows the results of testing the 7 evolved controllers and the random, hand-coded controller from the previous section. Each controller was tested for 100 games and with 100 rollouts and 50 maximum steps in the simulations. First, there is no clear difference between the controllers evolved with 100 and 500 individuals because they all performed at the same level. On the other hand, GP was able to evolve better agents in 2 out of the 7 runs. Controllers 2 and 6 both proved to be statistically significant compared to the hand-coded agent using an unpaired t-test with $p = 0.0114$ and $p = 0.0322$, respectively. To confirm these results, the most successful evolved controller (number 2) was retested against the random, hand-coded controller with 500 rollouts. The results in Table V illustrate that the evolved agent still outperformed the random agent with a statistically significant result for the t-test with $p = 0.0130$.

Table V. Test Results for the Most Evolved and Hand-coded Controllers with 500 Rollouts, 50 steps, and 100 Games

Controller no	Average score	Minimum score	Maximum score	Standard deviation
2	32,641.20	4,350	69,010	12,909.56
8	28,116.60	3,990	62,630	12,628.86

Figure 2 shows the developed tree for controller 2, which achieved better results than the random controller. The tree is very simple as it uses the same random, non-reverse controller used in the hand-coded version in which the Pac-Man is not in a dangerous position or in which it directs the Pac-Man to the nearest pill if it is in danger. This means that the only difference between this tree and the random non-reverse agent is that this tree will try to collect as many points as possible if it detects a danger.

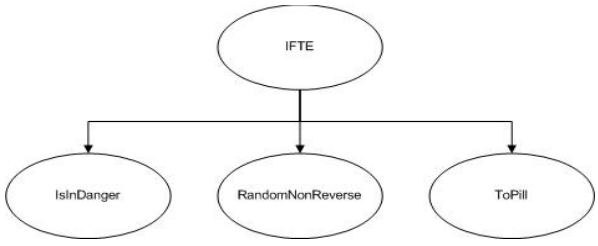

Figure 2. Controller 2

VI. Conclusion

This experiment consisted of two tasks; to build the MCTS controller and to enhance it using GP. The tests demonstrated that the hand-coded MCTS agent was able to perform at the level of a well-evolved agent if it had a large enough number of rollouts. The evolution process, on the other hand, faced several problems but its major setback was the time required to complete a single run, which forced us to reduce the number of evaluations to 5,000 compared to minimum of 25,000 recommended by Poli [7]. Nonetheless, the evolution was successful and a new controller was developed that outperformed the original MCTS controller.

References

1. Browne, C., E. Powley, D. Whitehouse, S. Lucas, P. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, *A Survey of Monte Carlo Tree Search Methods.* Computational Intelligence and AI in Games, IEEE Transactions on, 2012. **4**(99): p. 1-43.
2. Coulom, R., *Efficient selectivity and backup operators in Monte-Carlo tree search.* Computers and Games, 2007: p. 72-83.
3. Bouzy, B. and T. Cazenave, *Computer Go: an AI oriented survey.* Artificial Intelligence, 2001. **132**(1): p. 39-103.
4. Bouzy, B. *Move Pruning Techniques for Monte-Carlo Go*. in *Advances in Computer Games*. 2005. Taipei, Taiwan: Springer
5. Samothrakis, S., D. Robles, and S. Lucas, *Fast Approximate Max-n Monte Carlo Tree Search for Ms Pac-Man.* Computational Intelligence and AI in Games, IEEE Transactions on, 2011. **3**(2): p. 142-154.
6. Koza, J., *Genetic programming: on the programming of computers by means of natural selection*1992, Cambridge, MA: The MIT press.
7. Poli, R., W. Langdon, and N. Mcphee, *A field guide to genetic programming*2008, UK: Lulu Enterprises Uk Ltd.
8. Chaslot, G.M.J.B., *Monte-Carlo Tree Search.* 2010.
9. Winands, M., Y. Björnsson, and J.T. Saito, *Monte-carlo tree search solver.* Computers and Games, 2008: p. 25-36.
10. Winands, M.H.M., Y. Bjornsson, and J. Saito, *Monte Carlo Tree Search in Lines of Action.* Computational Intelligence and AI in Games, IEEE Transactions on, 2010. **2**(4): p. 239-250.
11. Kocsis, L., C. Szepesvári, and J. Willemson, *Improved monte-carlo search.* Univ. Tartu, Estonia, Tech. Rep, 2006. **1**.
12. Brandstetter, M.F. and S. Ahmadi, *Reactive Control of Ms. Pac Man using Information Retrieval based on Genetic Programming*, in *Computational Intelligence and Games (CIG)*2012, IEEE: Granada, Spain. p. 250-256.
13. Rosca, J. *Generality versus size in genetic programming*. in *the First Annual Conference on Genetic Programming*. 1996. MIT Press.
14. Alhejali, A.M. and S.M. Lucas. *Evolving diverse Ms. Pac-Man playing agents using genetic programming*. in *Computational Intelligence (UKCI), Workshop on, .* 2010. Essex, UK: IEEE.
15. Alhejali, A.M. and S.M. Lucas. *Using a training camp with Genetic Programming to evolve Ms Pac-Man agents*. in *Computational Intelligence and Games (CIG), IEEE Conference on*. 2011. Seol, South Korea: IEEE.
16. Robles, D. and S. Lucas. *A Simple Tree Search Method for Playing Ms. Pac-Man*. in *The IEEE Symposium on Computational Intelligence and Games (cig'09)*. 2009. IEEE.
17. Ya'nan, H., H. Suoju, W. Junping, L. Xiao, Y. Jiajian, and H. Wan, *Dynamic Difficulty Adjustment of Game AI by MCTS for the game Pac-Man*, in *Natural Computation (ICNC), Sixth International Conference on*2010 IEEE: Yantai, China. p. 3918-3922.
18. Xiao, L., L. Yao, H. Suoju, F. Yiwen, Y. Jiajian, J. Donglin, and C. Yang. *To Create Intelligent Adaptive Game Opponent by Using Monte-Carlo for the Game of Pac-Man*. in *Natural Computation, 2009. ICNC '09. Fifth International Conference on*. 2009.
19. Tong, B.K.B. and C.W. Sung. *A Monte-Carlo approach for ghost avoidance in the Ms. Pac-Man game*. in *Games Innovations Conference (ICE-GIC), 2010 International IEEE Consumer Electronics Society's*. 2010.
20. Tong, B.K.B., C.M. Ma, and C.W. Sung, *A Monte-Carlo Approach for the Endgame of Ms. Pac-Man*, in *Computational Intelligence and Games (CIG), IEEE Conference on*2011, IEEE: Seol, Korea. p. 9-15.
21. Nguyen, K.Q. and R. Thawonmas. *Applying Monte-Carlo Tree Search to collaboratively controlling of a Ghost Team in Ms Pac-Man*. in *Games Innovation Conference (IGIC), IEEE International*. 2011. IEEE.
22. Rohlfshagen, P. and Lucas, S.M.. *Ms Pac-Man versus Ghost Team CEC 2011 competition*. in *Evolutionary Computation (CEC), 2011 IEEE Congress on*. 2011.
23. Rohlfshagen, P. *IEEE Conference on Computational Intelligence and Ganes (CIG 2011) Ms Pac-Man vs Ghost Team Competition*. 2011; Available from: http://www.pacman-vs-ghosts.net.
24. Ikehata, N. and T. Ito. *Monte-Carlo Tree Search in Ms. Pac-Man*. in *IEEE Conference on Computational Intelligence and Games (CIG)*. 2011. seol, Suth Korea: IEEE.
25. Nakamura, M., K.Q. Nguyen, and R. Thawonmas, *ICE pAmbush CIG11, Entry report*, in *IEEE Conference on Computational Intelligence and Ganes (CIG 2011) Ms Pac-Man vs Ghost Team Competition*2011: Seoul.
26. Pepels, T. and M.H.M. Winands. *Enhancements for Monte-Carlo Tree Search in Ms Pac-Man*. in *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*. 2012. Granada, Spain: IEEE explore.
27. Philipp Rohlfshagen, Simon M. Lucas. *WCCI 2012 Pacman vs Ghosts Competetion*. 2012 [cited 2012 19/11/2012]; Available from: http://www.pacman-vs-ghosts.net/competitions/3.
28. Nguyen, K. and R. Thawonmas, *Monte-Carlo Tree Search for Collaboration Control of Ghosts in Ms. Pac-Man.* Computational Intelligence and AI in Games, IEEE Transactions on, 2013. **5**(1): p. 11.
29. Shafiei, M., N. Sturtevant, and J. Schaeffer. *Comparing UCT versus CFR in simultaneous games*. in *IJCAI Workshop on General Game Playing*. 2009. Pasadena, CA, USA.
30. Samothrakis, S., D. Robles, and S.M. Lucas. *A UCT agent for Tron: Initial investigations*. in *Proc. IEEE Symp. Comput. Intell. Games, Dublin, Ireland*. 2010.
31. Drake, P. and S. Uurtamo. *Move ordering vs heavy playouts: Where should heuristics be applied in Monte Carlo Go*. in *Proceedings of the 3rd North American Game-On Conference*. 2007.
32. Xie, F. and Z. Liu. *Backpropagation modification in Monte-Carlo game tree search*. in *Intelligent Information Technology Application, 2009. IITA 2009. Third International Symposium on*. 2009. IEEE.
33. Soule, T. and J.A. Foster, *Effects of code growth and parsimony pressure on populations in genetic programming.* Evolutionary computation, 1998. **6**(4): p. 293-309.