A Monte-Carlo Approach for the Endgame of Ms. Pac-Man

Bruce Kwong-Bun Tong, Chun Man Ma, and Chi Wan Sung, *Member, IEEE*

***Abstract*—Ms. Pac-Man is a challenging video game which provides an interesting platform for artificial intelligence and computational intelligence research. This paper introduces the novel concept of path testing and reports an effective Monte-Carlo approach to develop an endgame module of an intelligent agent that plays the game. Our experimental results show that the proposed method often helps Ms. Pac-Man to eat pills effectively in the endgame. It enables the agent to advance to higher stages and earn more scores. Our agent with the endgame module has achieved a 20% increase in average score over the same agent without the module.**

I. Introduction

Computer games, which capture various aspects of real-life problems, provide a suitable platform for testing Artificial Intelligence (AI) or Computational Intelligence (CI) techniques. Although achieving high scores is the typical objective of many computer games, when designing a computer agent for them, it is not easy to directly use that as the performance measure to make control decisions, since a game play typically does not last for a short period and during the process, there may be many short-term conflicting objectives, which make the design of a computer agent difficult. Recently, there are many competitions for game agent design, including the competitions of Ms. Pac-Man [1], Mario [2] and simulated car racing [3]. These competitions provide a suitable test-bed for evaluating different AI or CI techniques.

Developed by the Namco Company in 1980, Pac-Man has now become a classic arcade game. It is a single-player game, in which the human player controls Pac-Man to traverse a maze, avoid the four ghosts (non-player characters) and eat all the pills to clear a stage. In the original version of Pac-Man, the ghosts have deterministic behaviors, allowing players to use a pre-determined fixed path to play the game. To increase the fun and difficulty of the game, a variation called Ms. Pac-Man introduces randomness to the movement of the ghosts. In this paper, we focus on this particular version.

In each stage of the Ms. Pac-Man game, a player needs to control Ms. Pac-Man to avoid being eaten by the four ghosts, namely, Blinky (red), Pinky (pink), Inky (light blue) and Sue (orange), and to eat all the pills in a maze. When all the pills are eaten, the stage is cleared and the game proceeds to the next stage. Totally, there are four mazes in the Ms. Pac-Man game. The first two stages use the first maze, the next three stages the second maze, the next three the third maze, and the next four the fourth maze. Fig. 1 shows the screen capture of the first maze and the second maze of the game.

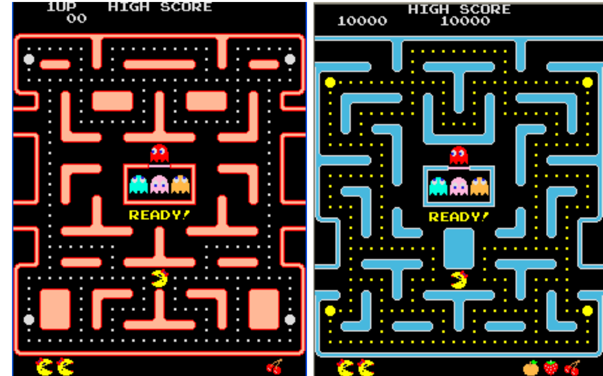

Fig. 1. The first maze (left) and second maze (right) of Ms. Pac-Man.

As mentioned, a computer game typically has many short-term conflicting objectives during a game play. In this paper, we focus on the endgame of each stage of the Ms. Pac-Man game. The objective of the endgame is relatively easy to specify. That is, Ms. Pac-Man should be controlled to eat all the remaining pills as soon as possible. This differs from the start of the game, where a player may prefer to make use of the power pills to eat the ghosts rather than to clear a stage.

Although the objective of the endgame is easier to define, the design of the endgame module is by no means trivial, since one has to balance the desire of eating pills and the assurance of the safety of Ms. Pac-Man. In this paper, we propose the use of Monte-Carlo simulations to tackle the problem. Our numerical experiments show that our approach is effective and enables Ms. Pac-Man to clear more stages.

The rest of this paper is organized as follows: In Section II we review the related research in Pac-Man and Ms. Pac-Man. Section III describes our proposed Monte-Carlo method for the endgame. Section IV presents the details and results of our experiments. Section V provides conclusions and discussions on future works.

II. Related Work

There were a number of researchers working on Pac-Man and Ms. Pac-Man agents since 1992 [4]. However, as most of their works were tested on their Pac-Man or Ms. Pac-Man simulators with modified game rules, it is not easy to compare their relative performance and results.

Bruce K. B. Tong is with Department of Electronic Engineering, City University of Hong Kong (e-mail: kbtong@cityu.edu.hk).

Chun Man Ma is with Department of Electronic Engineering, City University of Hong Kong (e-mail: chunmanma3@student.cityu.edu.hk).

Chi Wan Sung is with Department of Electrical Engineering, City University of Hong Kong (e-mail: albert.sung@cityu.edu.hk).

978-1-4577-0011-8/11/$26.00 ©2011 IEEE

Since 2007, the Ms. Pac-Man competition [1] has been held in various IEEE conferences. It provides a good platform to compare the performance of different AI and CI techniques.

A rule-based Ms. Pac-Man agent developed by Fitzgerald, Kemeraitis and Congdon called RAMP [5] won the competition held in the IEEE World Congress on Computational Intelligence (WCCI) 2008. It achieved a high score of 15,970 points and an average of 11,166. It is the first agent which obtained an average score higher than 10,000 in the competition.

A simple tree search method was proposed by Robles and Lucas [6]. The best path for Ms. Pac-Man is searched from a simplified game tree. The tree is reduced by ignoring the movements of ghosts and/or changing of the ghost state. It achieved a high score of 15,640 and an average of 7,664 in the IEEE Conference on Computational Intelligence and Games (CIG) 2009 and was the runner-up in the competition. This work demonstrated the importance of look-ahead strategy of agent design.

The rule-based agents, ICE Pambush 2 [7], 3 [8], and 4 [9] developed by Thawonmas et al, were the champion in the IEEE Congress on Evolutionary Computation (CEC) 2009, the champion in CIG 2009, and the first runner-up in CIG 2010, respectively. It achieved a maximum score of 24,640 and an average of 13,059 in CEC 2009, a maximum of 30,010 and an average of 17,102 in CIG 2009, and a maximum of 20,580 and an average of 15,353 in CIG 2010. These three versions are similar. They use A* algorithm to search for the pill with lowest cost, which is defined mainly according to the distances between Ms. Pac-Man and the ghosts. Besides, in order to further increase the score, the agents aggressively lure the ghosts to the power pills followed by ambushing them. Their works were further improved by optimizing the parameters using an evolutionary algorithm.

Pac-mAnt [10] developed by Martin, Martinez, Recio and Saez has won the champion in CIG 2010. It achieved a high score of 21,250 and an average of 9,343 in the competition. It uses ant colonies to control Ms. Pac-Man and the parameters of the ants were optimized by genetic algorithm.

Tong and Sung proposed a Monte-Carlo method to select a move when Ms. Pac-Man is in the dangerous state [11]. Their agent achieved a maximum score of 10,820 and an average of 5,702 in CIG 2010. Although the agent did not obtain a score closed to the champion, it often escapes from a dangerous situation successfully and prevents Ms. Pac-Man being trapped by the ghosts. It demonstrated the importance of look-ahead in searching and evaluating a move in a game tree. In this paper, we improve the agent by adding a new endgame module, also based on Monte-Carlo methods.

III. Monte-Carlo Approach for The Endgame

Instead of considering the entire agent design, in this paper, we focus on the endgame of Ms. Pac-Man. For each stage of the game, we say that the endgame of that stage has been reached whenever the number of remaining pills is less than or equal to a pre-specified threshold, denoted by η.

```
function isNearby(arguments: g0, g1, g2, g3, t) {
    //g0, g1, g2, g3: the locations of the four ghosts when
Ms. Pac-Man reaches the destination
    //t: the destination of the path

    d_gt ← min(d(g0, t), d(g1, t), d(g2, t), d(g3, t))
    if (d_gt ≤ d_min) return true /* there is a ghost nearby if
the min. shortest distance between g_i and t is less than the
threshold value d_min */
    for each adjacent vertex v_i of t {
        d_tv ← d(t, v_i)
        d_gv ← min(d(g0, v_i), d(g1, v_i), d(g2, v_i), d(g3, v_i))
        if (d_tv < d_gv) return false /* if there exists an
adjacent vertex of t which Ms. Pac-Man can reaches
before the ghosts, the situation is considered as no ghosts
are nearby Ms. Pac-Man */
    }
    return true
}
```

Fig. 2. The rule of determining if there is any ghost nearby Ms. Pac-Man after she reaches the destination

Just like in our previous work [11], we divide a maze into 28 × 30 square cells, each of which contains 8 × 8 pixels. A cell is either passable or impassable. Examples of impassable cells include the walls and the blocked area in a maze. Clearly, a pill must reside in a passable cell. Given any two passable cells A and B, we define a *path* from A to B as a sequence of neighboring passable cells that starts from A and ends at B. A path that does not visit any cell twice is said to be a *simple path*.

To facilitate path finding in a maze, we represent each game maze by a weighted graph. As Ms. Pac-Man is not allowed to enter impassable cells, they are ignored in the graph representation. Regarding the passable cells, we represent those with three or more adjacent cells, i.e., intersections in the maze, as vertices of the corresponding graph. Two vertices are connected by an edge if they can reach each other without passing through any other vertices. Each edge has a weight, which measures the distance in cells between the two corresponding vertices. The shortest distance between any pair of vertices and the corresponding path are pre-computed using Dijkstra's algorithm. The shortest distance between any two cells c_1 and c_2, denoted by $d(c_1, c_2)$, is computed on demand, based on some pre-computed information as in our previous work [11]. The results are cached for future lookup; re-calculation is therefore unnecessary. This allows us to design a fast shortest-path algorithm, which plays an important role in the Monte-Carlo simulation to be described later.

Our proposed endgame module consists of two major components, namely *path generation* and *path testing*. The first component generates a certain number of paths that connect two given cells, typically with the current location of Ms. Pac-Man as one cell while the location of one of the

remaining pills as the other cell. Due to the structure of the mazes, we can always find more than one path connecting the two cells.

After suitable paths are found, the second component will then be invoked to test whether each of the generated paths is safe if Ms. Pac-Man chooses to move according to that path. To perform such a testing, we apply the Monte-Carlo technique, as described below.

A. Monte-Carlo Path Testing

We use Monte-Carlo simulation to test whether a given path is safe or not. The simulation is performed in discrete time and involves the movement of five objects, namely Ms. Pac-Man and the four ghosts. We make the following assumptions in their movement:

1. At any given time, Ms. Pac-Man and the four ghosts reside in one of the passable cells.
2. After one time unit, Ms. Pac-Man moves to the next cell according to the given path, and the four ghosts move to their neighboring cells according to a probabilistic model, to be described shortly.

The accuracy of the simulation depends highly on how well we model the behavior of the ghosts. In [11], we have constructed a behavior model for the ghosts and have successfully applied it to the ghost avoidance module. Here we adopt the same model. We assume that a ghost may change its current direction only when it reaches an intersection. (In the real game, the ghosts often behave like that, though there are exceptions.) It then selects a new direction according to the following rules:

- With probability 0.9, Blinky moves towards Ms. Pac-Man's current coordinate according to the shortest-path algorithm.
- With probability 0.8, Pinky moves, according to the shortest-path algorithm, towards the first intersection to be reached by Ms. Pac-Man. But if d(Pinky, Ms. Pac-Man) ≤ 10, it moves towards Ms. Pac-Man's current location instead.
- With probability 0.7, Inky moves, according to the shortest-path algorithm, towards the intersection that has just been passed by Ms. Pac-Man. But if d(Inky, Ms. Pac-Man) ≤ 10, it moves towards Ms. Pac-Man's current position instead.
- With probability 0.5, Sue moves towards Ms. Pac-Man's current position according to the shortest-path algorithm.

In each of the above cases, the given rule will be enforced only with a certain probability p. With probability $1 - p$, a ghost chooses a direction uniformly at random instead.

```
function genKSimplePaths
  (arguments: s, t, k, d_max, d_current, path_current, k_paths) {
  //s: the source cell
  //t: the destination cell
  //k: the number of shortest simple paths to find
  //d_max: the maximum allowed distance
  //d_current: the length of the current path
  //path_current: the current path
  //k_paths: store the k shortest simple paths

  if (s = t) {
    clone and insert path_current to k_paths (maintain in
  ascending order of path length)
    if (size(k_paths) > k) { //i.e. size(k_paths) = k + 1
    remove the (k+1)-th path from k_paths
      d_max ← the length of the k-th path
    }
    return
  }
  for each adjacent vertex v_i of s AND v_i is not visited {
    d_i ← d_current + d(s, v_i) + d(v_i, t)
  }
  sort v_i's in ascending order of d_i
  for each i AND (d_i < d_max) {
    append v_i to path_current
    mark cell v_i as visited
    genKSimplePaths(v_i, t, k, d_max, d_i, path_current,
k_paths)
    remove v_i from path_current
    mark cell v_i as unvisited
  }
}
//Note: all cells are marked as unvisited initially
```

Fig. 3. The algorithm of k shortest simple paths generation

Given any path, the above simulation is executed a certain number of times. Each time, either "safe" or "unsafe" is returned, depending on the simulation outcome. The returned value is "safe" if and only if Ms. Pac-Man reaches the destination without being eaten by a ghost and after reaching the destination, there are no ghosts nearby. Whether there is any ghost nearby is defined according to a simple rule shown in Fig. 2.

Given any path, we obtain the *alive rate* by repeating the simulations n_{sim} times. The alive rate is defined as the proportion of times that the outcome is "safe".

Note that the last remaining pill is treated in a slightly different way. Since Ms. Pac-Man can clear the stage after successfully eating the last pill, it is not necessary to check if there is any ghost nearby the destination (i.e., the location of the last pill) in the path testing simulation.

B. Path Generation

Now we describe how paths are generated for Monte-Carlo testing. Suppose there are m remaining pills in the endgame. Let cell a be the current position of Ms. Pac-Man and cell b_i be the position of pill i, for $i = 1, 2, \dots, m$. For each pair (a, b_i), we find the k shortest simple paths that connect cell a and cell b_i, and test them using the Monte-Carlo path testing method.

Recall that our shortest-path algorithm applies to the special case where k is equal to one. For general values of k, the algorithm in [12], for example, can be used. In our implementation, as the number of vertices of the graph corresponding to a game maze is not large, we simply use depth-first search to find them. The pseudo code is shown in Fig. 3.

The algorithm searches from the start cell, a. It finds all its adjacent vertices, which are denoted by $v_1, v_2, \dots, v_{n(a)}$, where $n(a)$ is the number of adjacent vertices of cell a. For $i = 1, 2, \dots, n(a)$, if $d(a, v_i) + d(v_i, b)$ is less than a threshold value d_{max}, the algorithm recursively searches paths from v_i to b. A path is obtained if b is already adjacent to v_i. The path and its length are then stored. If more than k paths have been found, only the k shortest paths will be kept. Those longer paths will be discarded and the threshold d_{max} will be updated to the length of the current k-th path.

Since d_{max} is monotonically decreasing as the search proceeds, the path generation process can be sped up by sorting the adjacent vertices v_i's of cell a in ascending order of $d(v_i, b)$, so that d_{max} decreases faster and hence longer paths can be removed from our considerations earlier.

In order to avoid generating loops in a path, visited vertices are marked and will not be visited again. A vertex will be unmarked if the algorithm backtracks to it for generation of another simple path.

We remark that this algorithm may return fewer than k paths, which happens when the k value is too large or the initial d_{max} value is too small so that the graph does not contain k paths between a and b with length smaller than d_{max}.

C. Endgame Algorithm

Now we summarize the endgame algorithm in Fig. 4. When the number of remaining pills, m, is less than or equal to a pre-specified threshold, η, the endgame module is invoked. The algorithm first sorts the pills according to increasing distance between the current location of Ms. Pac-Man, a, and the location of each remaining pills, b_i. Starting from (a, b_1), it generates k shortest simple paths between them using the previous described path generation algorithm. Next it tests each of the generated paths, starting from the shortest one, using the Monte-Carlo path testing method. It then iterates to the next pair, (a, b_2), and repeats the same procedure. The algorithm stops when all pairs have been done. The path that has the highest *alive rate* is selected and returned.

However, if the highest *alive rate* is less than another threshold α, it indicates that no paths for pill capturing are safe enough for Ms. Pac-Man to use. Another module is invoked to determine the move of Ms. Pac-Man. In our implementation, we simply let Ms. Pac-Man to choose a direction that maximizes the minimum distance to the nearest ghost. She then waits until there is a good chance to eat the remaining pills so as to clear the stage.

To cope with the tight time constraint of the game, we need to ensure that the endgame module returns a move within a certain time limit. To this end, several refinements to the algorithm have been made. Firstly, there is a timer t to control the running time of the endgame module. When t remains smaller than a pre-specified constant, t_{max}, the endgame module continues to test the k paths for each of the remaining pills. When timeout occurs, even though not all pills have been fully tested, the endgame module returns the path with the highest *alive rate* among its tested paths. Secondly, to speed up the algorithm, once a path that has alive rate higher than a pre-specified threshold, β, has been found, the

```
function endgame(arguments: a, b, g, n_sim, t_max, α, β) {
   //a: the current location of Ms. Pac-Man
   //b: the locations of the remaining pills
   //g: the locations of the four ghosts
   //n_sim: the no. of simulations used for testing a path
   //t_max: the max. allowed time for this module
   //α: the min. required alive rate for a safe path
   //β: the max. required alive rate for a safe path

   m ← size(b)
   sort m pills by ascending order of d(a, b_i)
   k, d_max ← the value depends on m (see Table I)
   path_best ← null
   rate_max ← 0
   reset timer t
   for each pill pill_i at b_i AND timer t ≤ t_max {
      d_current ← 0
      path_current ← a
      k_paths ← null
      mark all cells as unvisited
      genKSimplePaths(a, b_i, k, d_max, d_current, path_current,
k_paths)
      for each path path_i in k_paths AND t ≤ t_max {
         rate_alive ← MonteCarloPathTesting(path_i, g,
n_sim)
         if (rate_alive > β) {
            path_best ← path_i
            return path_best
         } else if (rate_alive > rate_max) {
            rate_max ← rate_alive
            path_best ← path_i
         }
      }
   }
   if (rate_max < α) path_best ← null
   return path_best
}
```

Fig. 4. The algorithm of the proposed Monte-Carlo endgame module

algorithm halts and returns that path immediately.

TABLE I
THE PARAMETERS OF THE ENDGAME MODULE

η	35
n_{sim}	100
β	0.95
α	0.60
d_{min}	2
t_{max}	30ms

m	[1, 5)	[5, 15)	[15, 35]
k	15	12	8
d_{max}	60	50	50

IV. PERFORMANCE EVALUATION

We test our proposed endgame algorithm by implementing it in our previous Ms. Pac-Man agent [11], which was submitted to the competition held in CIG 2010. The original agent does not have any special handling in the endgame; it simply moves towards the nearest pill using the shortest path, provided that there are no ghosts around the current position of Ms. Pac-Man. In the new agent, the Monte-Carlo endgame module is invoked when the endgame of each stage is reached. Except that, there are no differences between the two agents.

The parameters of the endgame module, listed in Table I, were selected manually based on our direct observation. The threshold of number of remaining pills, η, below which our proposed endgame module is invoked, is chosen to allow the agent to have enough time to generate paths for each remaining pills and test the safeness of these generated paths. The number of simulation runs for testing the safeness of a path is denoted by n_{sim}. The larger its value, the more accurate the result is but the longer the computation time is required. We tried different values of n_{sim} and find a small value that can produce reliable simulation result. Note that the number of shortest simple paths, k, and the maximum allowed distance, d_{max}, are increased when the number of remaining pills decreases. In other words, we test more paths for each source-destination pair when there are fewer remaining pills in the maze. The reason is that more simulation time for each source-destination pair is available if there are fewer pills left.

We tested our new agent in the real Ms. Pac-Man game via the screen capture mechanism, which has been used in Ms. Pac-Man competitions [1]. The experiments were conducted on a PC with Windows XP Professional SP3, Intel Core 2 Quad Q9550 2.83 GHz CPU and 3 GB memory. Each agent played the Ms. Pac-Man game 70 times, and the statistics were collected.

A. Performance in the Real Ms. Pac-Man Game

We observed that our original agent does not eat pills effectively when it enters the endgame. It often moves towards a pill which is "unsafe" or not likely to be eaten by Ms. Pac-Man. The reason is that the agent uses a greedy method to solve a look-ahead problem. The ghosts may be far away from the target pill at the beginning of a search but they may be very near to it when Ms. Pac-Man reaches the target. The agent easily loses a life of Ms. Pac-Man after eating a pill.

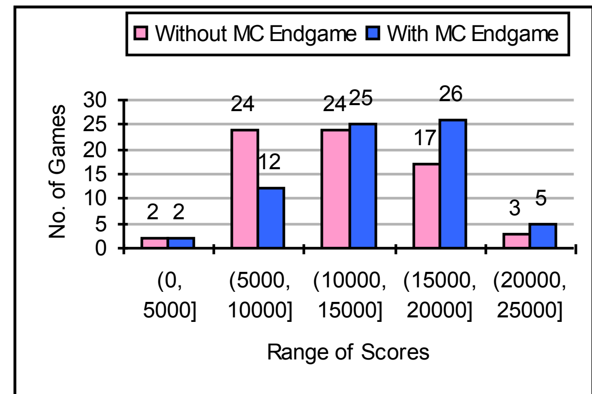

Fig. 5. The score distribution of the two agents for 70 games.

TABLE II
THE SCORES OF DIFFERENT AGENTS FOR 70 GAMES

Agent	Without MC Endgame	With MC Endgame
Minimum	2,930	3,650
The 25th percentile	7,700	12,100
Median	12,450	14,710
The 75th percentile	15,720	17,650
Maximum	22,100	23,130
Mean	11,870	14,269
Std. Dev.	4554	4428

Another problem is that the shortest path to a pill may not always be the best; a longer, detour path may be better in some occasions. One example is when a ghost is already on or near to the shortest path to the target pill. The original agent does not know that Ms. Pac-Man would be caught by the ghost on her way to the target when searching for the target pills. This situation often happens when the target pill is quite far away from the current location of Ms. Pac-Man. Under the control of the original agent, Ms. Pac-Man moves towards the nearest pill and on the midway she moves backward since there are ghosts nearby. She often keeps travelling around the maze without eating pills effectively in the endgame.

Now with the Monte-Carlo endgame module, the new agent attempts to search and test different paths for the remaining pills. From our observation, it can often find a "safe" path to reach a nearby remaining pill. Under the control of the new agent, Ms. Pac-Man aggressively eats the remaining pills by travelling along the "safe" paths. She is usually able to clear the stage and advance to the next one after entering the endgame. As the new agent is able to play more stages, it earns more scores. The scores obtained and the number of stages cleared are described in the below paragraphs.

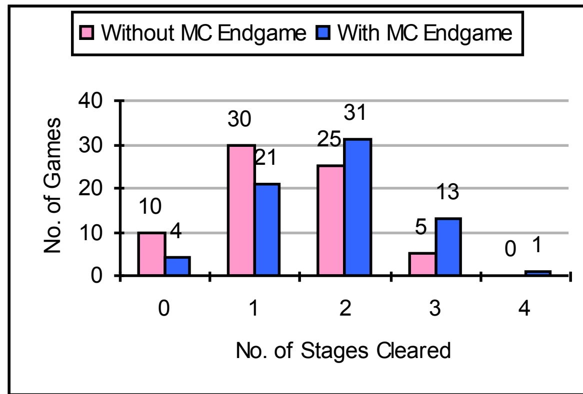

Fig. 6. The distribution of number of stages cleared by the two agents for 70 games.

TABLE III
THE NUMBER OF STAGES CLEARED BY DIFFERENT STRATEGIES FOR 70 GAMES

Agent	Without MC Endgame	With MC Endgame
Minimum	0	0
The 25th percentile	1	1
Median	1	2
The 75th percentile	2	2
Maximum	3	4
Mean	1.36	1.80
Std. Dev.	0.82	0.86

B. Scores obtained

The scores of the two agents are shown in Fig. 5 and Table II. The agent with the endgame module achieved a high score of 23,130 and an average score of 14,269 in 70 games. The original agent achieved a maximum score of 22,100 and an average score of 11,870 only. The endgame module raised the average score by around 20.2%. Besides, the minimum score, the 25th percentile, the median and the 75th percentile of the new agent are all higher.

C. The Number of Stages Cleared

The statistics of number of stages cleared by the two agents is shown in Fig. 6 and Table III. It is clearly shown that the agent with the endgame module cleared more stages than the original one. With the aid of the endgame module, Ms. Pac-Man is able to find an appropriate path to eat the remaining pills and thus easier to clear a stage and ultimately has a higher chance to earn more points.

It should be emphasized that among these 70 games there were two games in which the new agent played badly. It obtained scores below 5,000 and both games were over in the first stage. In fact, the new agent lost all lives of Ms. Pac-Man before entering the endgame of Stage 1. It means that for those particular plays, the endgame module was not called.

V. CONCLUSIONS

The aim of this paper is to design an effective method to play the endgame of Ms. Pac-Man. Existing methods such as Minimax Tree Search (including its variants), and Monte-Carlo Tree Search (MCTS) are not suitable for two reasons. First, Ms. Pac-Man is a real-time game. As the game tree is relatively large, Minimax Tree Search and MCTS do not have enough time to produce reliable or reasonable results. Second, for these approaches to work properly, expert knowledge is generally required. For Minimax Tree Search, a reasonable evaluation function for intermediate nodes is needed, which is not easy to design. For MCTS, its performance depends on whether the move selection in each of its simulation is properly guided. For example, the technique of pattern matching in Monte-Carlo Go does not exist in the Ms. Pac-Man game.

In this paper, we introduce the novel concept of path testing. Instead of searching a promising path (or move) from a huge game tree, we select reasonable paths from the game tree using heuristic techniques and use Monte-Carlo simulation to evaluate their values. In our proposed endgame module, the paths are selected by a *k* shortest simple paths algorithm. The paths are then tested by Monte-Carlo simulation, which is a fast and easy-to-implement method. It simplifies the evaluation of a path. Experimental results show that our endgame module can help Ms. Pac-Man clear stages effectively in the endgame and ultimately play more stages and achieve a higher score.

The path testing idea may also be applied to other games that have similar properties. For example, in many games, the opponent behavior is probabilistic so that the branching factor in the corresponding game tree is large. Besides, good strategies in some games are formed by long sequences of moves, which can be found only by deep search of the game tree. Common approaches such as Minimax Tree Search and MCTS do not handle these problems well.

We are currently targeting to develop a high-scoring agent to participate in the Ms. Pac-Man Screen Capture Competition. We have already designed a Monte-Carlo ghost avoidance module and a Monte-Carlo endgame module. We plan to have a pill capturing module, which is able to plan how to eat pills efficiently. Currently, our agent uses the greedy method; it just moves towards the nearest pill.

REFERENCES

[1] S. M. Lucas, "Ms. Pac-Man competition," SIGEVOlution, vol. 2, no. 4, pp. 37-38, 2007.

[2] J. Togelius, S. Karakovskiy and R. Baumgarten, "The 2009 Mario AI Competition," IEEE Congress on Evolutionary Computation, pp. 1-8, 2010

[3] D. Loiacono, J. Togelius, P. L. Lanzi, L. Kinnaird-Heether, S. M. Lucas, M. Simmerson, D. Perez, R. G. Reynolds and Y. Saez "The WCCI 2008 Simulated Car Racing Competition," IEEE Symposium on Computational Intelligence and Games, pp. 119-126, 2008

[4] J. R. Koza, Genetic Programming: on the Programing of Computers by Means of Natural Selection. MIT Press, 1992.

[5] A. Fitzgerald and C. B. Congdon, "RAMP: A rule-based agent for Ms. Pac-Man," IEEE Congress on Evolutionary Computation, pp. 2646-2653, 2009.
[6] D. Robles and S. M. Lucas, "A simple tree search method for playing Ms. Pac-Man," IEEE Symposium on Computational Intelligence and Games, pp. 249-255, 2009.
[7] R. Thawonmas and H. Matsumoto, "Automatic controller of Ms. Pac-Man and its performance: Winner of the IEEE CEC 2009 software agent Ms. Pac-Man competition," Proc. of Asia Simulation Conference 2009 (JSST 2009), 2009.
[8] R. Thawonmas and T. Ashida, "Evolution strategy for optimizing parameters in Ms Pac-Man controller ICE Pambush 3," IEEE Symposium on Computational Intelligence and Games, pp. 235-240, 2010.
[9] T. Ashida, T. Miyama, H. Matsumoto and R. Thawonmas. ICE Pambush 4 controller description paper [Online]. Available: http://cswww.essex.ac.uk/staff/sml/pacman/cig2010/ICE%20Pambush%204.pdf
[10] E. Martin, M. Martinez, G. Recio and Y. Saez, "Pac-mAnt: Optimization Based on Ant Colonies Applied to Developing an Agent for Ms. Pac-Man," IEEE Symposium on Computational Intelligence and Games, pp. 458-464, 2010.
[11] B. K. B. Tong and C. W. Sung, "A Monte-Carlo Approach for Ghost Avoidance in the Ms. Pac-Man Game," International IEEE Consumer Electronics Society's Games Innovations Conference (ICE-GIC), pp. 1-8, 2010.
[12] J. Y. Yen, "Finding the K Shortest Loopless Paths in a Network," Management Science, Vol. 17, No. 11, pp. 712-716, 1971