Exploring Ms. Pac-Man Playing Agent [1]

CHENKAI KUANG, Carnegie Mellon University
HUI TU, Carnegie Mellon University

Pac-Man is a well-known, real-time computer game that provides an interesting platform for research and also requires non-trivial strategies for successful game play. This paper describes an approach to develop Pac-Man playing agent that dynamically chooses a target based on game state and uses A* search to find the best path with least cost. Cost model is specifically defined to ensure the path with least cost is the safest path. The experiment result shows the agent can achieve reasonable scores when compared to recent Pac-Man AI competition result. This paper also proposes an outline of using reinforcement learning to build the agent, and some discussion about future game AI as well.

Categories and Subject Descriptors: **I.2.8 [Artificial Intelligence]**: Problem Solving, Control Methods, and Search

Additional Key Words and Phrases: Pac-Man, A* Search, Cost Model, Reinforcement Learning

1. INTRODUCTION

During the last two decades, games have always been used as a test platform for Artificial Intelligence algorithm. In the same period, game industry becomes one of the fastest-growing industries in the world with sophisticated technology in game consoles and development of mobile device and social gaming as well. Moreover, applying AI to games has achieved some very notable successes such as Deep Blue, a chess playing machine that defeated world-champion Garry Kasparov in 1997. As more fields, no matter industry or academia, become aware of the importance of AI research on game playing strategy, the variety of discussion increases.

Ms. Pac-Man is a popular maze game in recent AI research. While it is no longer the latest or most advanced example of game development, it still provides a classical platform that is complex and challenge enough to require intelligent strategies for superb gameplay. In recent years, IEEE Computational Intelligence in Games Conference (CIG)[1] has hosted annual competitions that allow AI controller design for Ms. Pac-Man to complete for the highest score.

Looking back to the previous working with Ms. Pac-Man, several technologies, such as Genetic Algorithm and Neural Networks, are used in implementation of Ms. Pac-Man and Ghost. At the same time, the hand-coded algorithm from every year IEEE CIG enjoyed higher performance. Therefore, our work aimed at producing high-scoring agent to participate IEEE CIG this year.

This paper describes the application of A* search strategy with ghost behavior prediction to find best path in Ms. Pac-Man controller. The rest of this paper is organized as follows: we review the previous work on Ms. Pac-Man in Section II, in Section III we have a description on our framework and experimental environment, in Section IV we give the details on our implementation of Ms. Pac-Man controller, in Section V we present our experiment result along with recent result of the annual competition , Section VI is our ideas on utilizing reinforcement learning, and Section VII provides summary and conclusions.

2. PREVIOUSR WORK

Koza used Pac-Man as an example to study the effectiveness of genetic algorithm for task prioritization [2]. His work relied on a set of predefined control primitives for perception, action and program control. His version of game is that different items and mazes have different scores. In his version, the approximated score is about 5000 points.

This work is Carnegie Mellon University 15780 Grad AI course's final report.
Author's email: Chenkai Kuang, chenkaik@andrew.cmu.edu; Hui Tu, hui.tu@sv.cmu.edu

Kalyanpur and Simon [3] utilized a genetic algorithm to try to improve the strategy of the ghosts in a Pac-Man-like game. Here the solution produced is also a list of directions to be traversed. A neural network is used to determine suitable crossover and mutation rates from experimental date.

Gallagher and Ryan [4] used a Pac-Man agent based on a simple finite-state machine model with a set of rules to control the movement of Pac-Man. The rules contained weight parameters which were evolved using the Population- Based Incremental Learning (PBIL) algorithm. They ran a simplified version of Pac-Man with only one ghost and no power pills, which takes away scoring opportunities in the game and takes away most of the complexity of the game. This approach was able to achieve some degree of learning, however the representation used appeared to have a number of shortcomings.

Lucas [5] proposed evolving neural networks as move evaluators in a Ms. Pac-Man implementation. Lucas focused on Ms. Pac-Man because it is known that the ghosts in this game behave in a pseudo-random fashion, thus eliminating the possibility of developing path-following patterns to play the game effectively and presumably making the game harder and leading to more interesting game play. The neural networks evolved utilize a handcrafted input feature vector consisting of shortest path distances from the current location to each ghost, the nearest power pill and the nearest maze junction. A score is produced for each possible next location given Pac-Man's current location. Evolution strategies were used to evolve connection weights in networks of fixed topology. The results demonstrate that the networks were able to learn reasonably successful game play as well as highlighting some of the key issues of the task, such as the impact of a noisy fitness function providing coarse information on performance.

More recently, ICE Pambush won in the IEEE Congress on Evolutionary Competition (CEC)2009. He moves Ms. Pac-Man based on the path cost and lures the Ghost near the power pill. With this strategy, he got a maximum score of 24,640 and an average of 13,059[6].

3. EXPERIMENT SETUP

While our project is focused on researching AI approach to build our Pac-Man agent, we didn't spend time programming the game engine and the graphics interface. Instead, we used a well-built and documented Pac-Man source package [7] which is used for every year's competition.

3.1 Basic Game Engine

The game is played asynchronously in real time: every discrete game tick, the game uses the moves supplied by the controllers to update the game state. Then, the game waits for 40ms for new actions to come in, after which the game updates again. At each game tick, the controllers have 40 ms to respond. The controllers are given a copy of the current game and the time the move is due. Each controller can then query the game using its many methods to compute an appropriate response. It is the responsibility of the controller to respond in a timely manner (this is one of the challenges of the competition).

If a controller returns a move on time, the game will use that move to advance the game. If the move happens to be illegal, the game tries to replay the previous move or chooses a new legal move randomly. If a controller does not replay on time, the game tries to play the previous move or, if that is not possible, chooses a legal move randomly.

The controller will receive a new copy of the game state for the remaining time of that game tick to compute a new move. In other words, if a controller takes 60ms to respond, it will have roughly (2x40)-60=20ms in the next game tick. If a controller does not reply over several game ticks, the game keeps replaying previous actions.

The game has four different mazes. The mazes are modeled as graph, with each node in the graph being connected to its immediate neighbors. Each node has two, three or four neighbors depending on whether it is in a corridor, a T-junction, or a crossroads. Each node occupied two screen pixels. After the maze has been loaded, a simple bread first search algorithm will run to compute the shortest path distance between every pair of nodes in the maze. Theses distances are stored as a look-up table, which speeds up our implementation of Pac-Man agent.

Game play is defined by the ghosts behaviors. The ghosts we use are starter ghost: if edible or Pac-Man is close to the power pill, run away from Pac-Man; if non-edible, attack Pac-Man with 0.9 probability, else choose random direction. Ghosts do reversals with a very low probability. The path ghosts choose to chase Pac-Man is the path with shortest distance.

3.2 Game APIs

The software package provides us with plenty of APIs to directly query the game state. Some useful APIs are:

getPacManCurrentNodeIndex(): current node index of Pac-Man

getGhostCurrentNodeIndex(Ghost ghost): current node index of specified ghost

getGhostEdibleTime(GHOST ghost): edible time left of specified ghost

getPillIndices() / getPowerPillIndices(): the indices to all nodes that have pills / power pills.

getShortestPath(int src, int dest): the shortest path from src to dest specified by their indices. We also have another version of this function that doesn't allow reversals.

isPillStillAvailable(int index) / isPowerPillStillAvailable(int index): whether the pill/powerpill specified is still there or has been eaten.

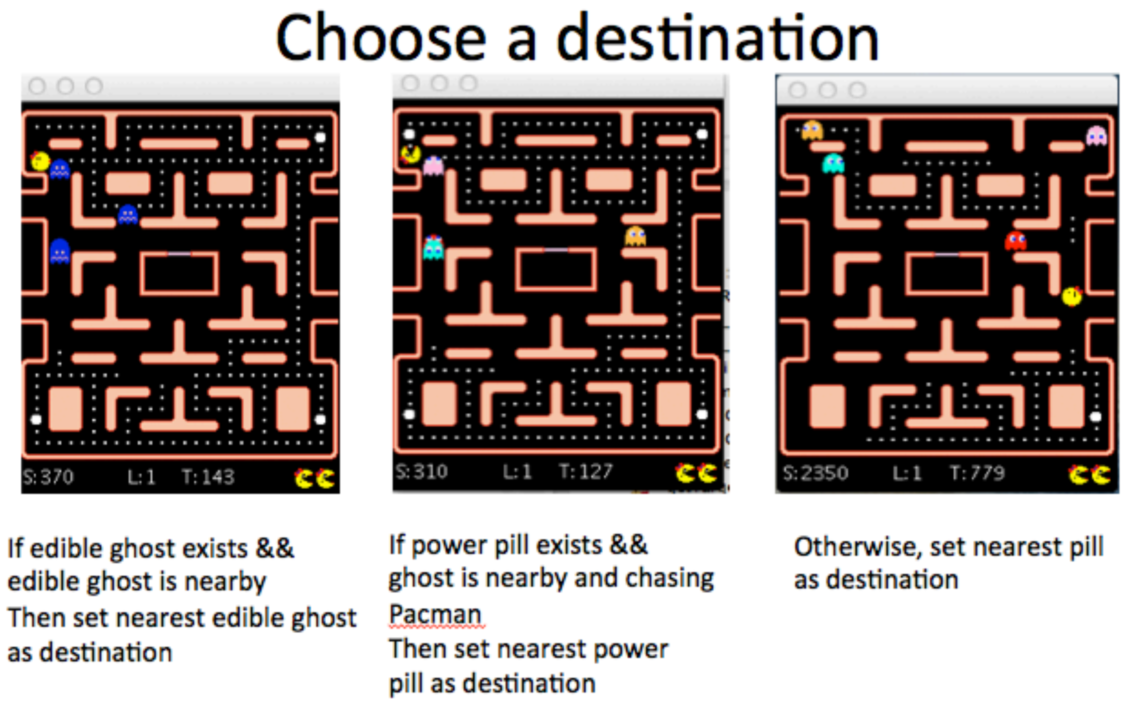
4. PAC-MAN AGENT DESIGN

In our approach, we mainly solve four problems:

- Pac-Man doesn't have a fixed target. We defined its target according to current game state. Destination candidates are the nearest pill, power pill and edible ghost.
- Given a destination, Pac-Man should choose a best path with least possibility of being eaten and maximize pills count in the path. We carefully create a cost model so that the path with least cost will be our best path. And we use A* to search the best path.
- Game state is changing in every game tick. So we need to recalculate the path at every move of Pac-Man.
- There are some rules for the four ghosts. We study these rules to predict ghosts' movement. Simple Markov Model is used for anticipation.

4.1 Choose a target

There are three targets: edible ghost, power pill and the nearest pill. Our Pac-Man makes this decision based on the game state. The details is as Figure 1:



Choose a destination

If edible ghost exists &&
edible ghost is nearby
Then set nearest edible ghost as destination

If power pill exists &&
ghost is nearby and chasing Pacman
Then set nearest power pill as destination

Otherwise, set nearest pill as destination

Fig. 1. How to choose a destination given game state

4.2 Cost Model

The basic idea of our Pac-Man AI is to define cost for every path that Pac-Man can move along when given current state. And then it will choose the path with least cost. A good cost model should satisfy three requirements:

- The path that will put Pac-Man in a dangerous state should have larger cost than safe path. Dangerous state means Pac-Man will be easily caught by ghosts or trapped by several ghosts in the near future. This is also the most crucial part of our cost model.
- The path that has more pills should cost less when dangerous costs are the same. So Pac-Man will prefer the path with more pills.
- If Pac-Man's target is not power pill, power pill cost should be large so that Pac-Man will leave power pill uneaten when it is safe.

We use A* to search for the path with least cost, which uses formula f(n) = g(n) + h(n). Each node in the path has an accumulative cost and a heuristic cost. For accumulative cost, we consider node cost and ghost cost. Node cost is based on what node contains. Therefore, cost = heuristicCost + nodeCost + ghostCost. We define some cost constants in the table below. The values of these constants are determined based on lots of experiments and observations. The description of these constants will be covered later.

Table I. Constant values in cost model

EMPTY_COST	4
PILL_COST	3
POWRPILL_COST	1
GHOST_IN_PATH	2000
GHOST_OUT_PATH	1000

4.2.1 Heuristic cost and nodeCost

For game like Pac-Man that is on a square that allows 4 directions of movement, heuristic cost is usually calculated using Manhattan distance. However, in Pac-Man game, we can adopt an exact precomputed heuristic between every pair of nodes. As mentioned before, the shortest path look-up table is loaded with each maze. Since ghost will not do reversal freely (actually in our game ghost has 0.05 possibility to turn around), so the distance which regards ghost as source node takes no-reversal into consideration.

To make our heuristic cost admissible, we assume among all the grid points along the shortest path, there is one power pill and others are pills. It is admissible because it uses minimum node cost along that path as heuristic cost, therefore, the real node cost will never below the heuristic cost. It uses minimum node cost because the cost relation is EMPTY > PILL > POWER_PILL. And when we limit search depth, there is at most one power pill in a path.

Heuristic cost's equation is:

$$(distance_{current\_to\_destination} - 1) \times PILL\_COST + POWER\_PILL\_COST \quad (1)$$

Where $distance_{current\_to\_destination}$ is the shortest path between current node and destination, $PILL\_COST$ and $POWER\_PILL\_COST$ is the cost constants. $PILL\_COST$ represents the cost of a node with pill, $POWER\_PILL\_COST$ represents the cost of a node with power pill. The following figure demonstrates the way to compute heuristic cost and node cost.

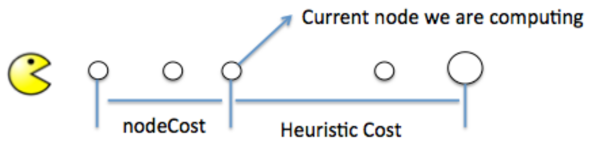


Fig. 2. Node Cost and Heuristic Cost. Node cost is the real cost so far. Heuristic cost is the admissible estimated cost to the goal.

4.2.2 Ghost Cost

It is intuitive that if there is a ghost residing in the path, then this path's cost should be very large. Meanwhile, the ghost cost should be inversely proportional to the distance between ghost and current node we are computing. Therefore, we simply calculate the ghost by the following logic:

```
If(ghost is edible)  ghostCost = 0
if(current index == ghost index)  ghostCost = GHOST_IN_COST
else  ghostCost = GHOST_OUT_COST / distance(ghost to current)
```

Since we are determining the whole path's dangerous level by ghost cost of the last node expanded in this path. The last node should be aware of the previous ghost cost. So when thinking about single node's ghost cost, it's reasonable that the node's ghost cost should inherit from previous node in the same path. And current node's ghostCost will only be updated when new ghostCost > prev_ghostCost.

Total ghost cost is equal to the sum of four ghosts' costs.

Another problem comes is how to anticipate ghosts' movement, Sometimes we want to expand node in A*, however, it's inappropriate to use ghost's current index, because at the time when Pac-Man reaches the expanded node, ghost's index may change. Therefore, we predict ghosts' position at the time when expanded node is reached. The anticipation is based on ghosts' fixed rules, which is, if edible, run away from Pac-Man, if non-edible, attack Pac-Man with 0.9 probability. Simple Markov model is used to compute the expectation of ghost cost according to the probability distribution of ghost's position.

4.3 A* Search for least cost path

We use a min-heap to store all the nodes waiting to be expanded in A*. So the branch operation is of $O(\log(n))$running time and select min-cost node operation is $O(1)$. The search begins at current node of Pac-Man, then expand to its neighbors that are not wall. Compute all the neighbors' cost and add them into heap. Then select the node with minimum cost to expand. The search will terminate when it reaches destination or exceeds maximum search depth. We set search depth to 20, which is a sufficient length for our agent to look ahead and also guarantee to make a decision in every game tick time limited (40ms). The flow chart of A* is as following:

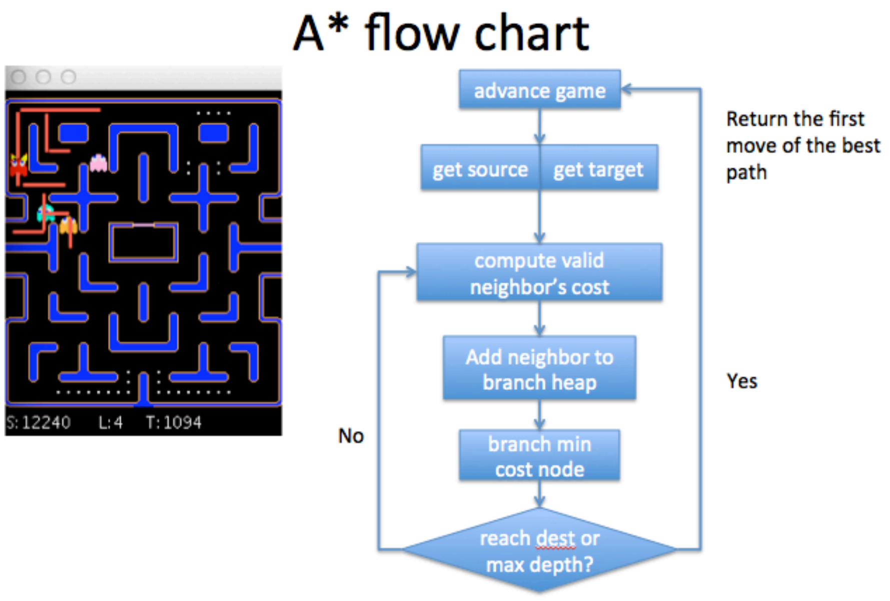


Fig. 3. A* search flow chart. It uses a min-heap to store nodes waiting to be expanded. It will not expand the node backwards. The maximum depth of the search is 20, which is a reasonable depth guaranteeing good performance and computational speed.

5. EXPERIMENT RESULT

Figure 4 shows the scores our agent got in consecutive 300 games.

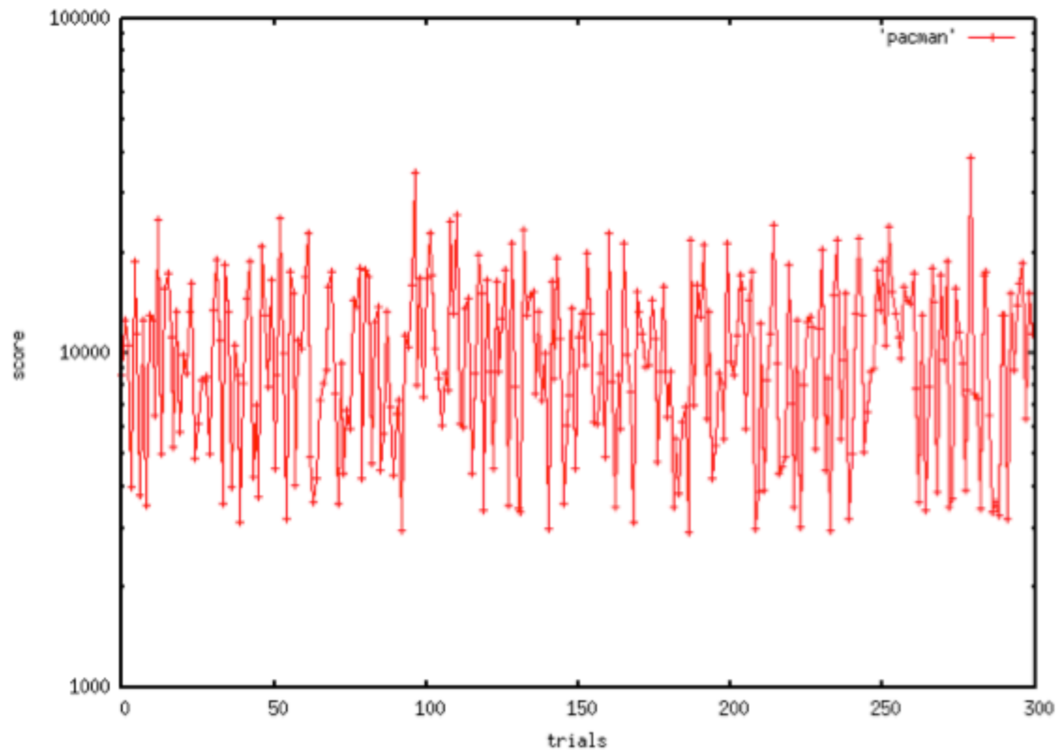
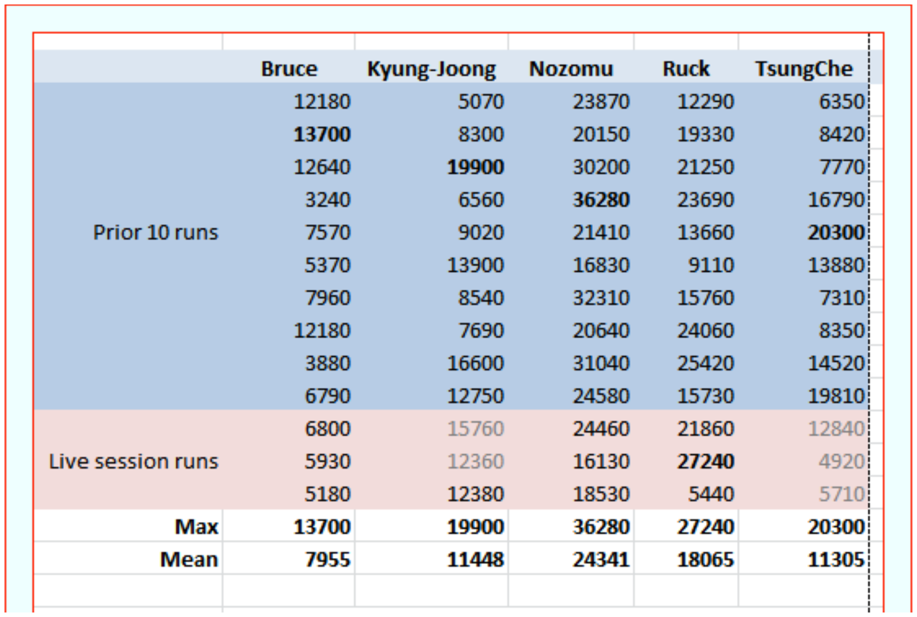


Fig. 4. Scores of 300 games. Average score is 10782.36. Maximum score is 35890.27.

The average score is 10782.36. The maximum score is 35890.27. Figure 5 shows the result of IEEE CIG 2011[8] competition result. From that we can tell our agent's performance is at medium level.



	Bruce	Kyung-Joong	Nozomu	Ruck	TsungChe
	12180	5070	23870	12290	6350
	13700	8300	20150	19330	8420
	12640	**19900**	30200	21250	7770
	3240	6560	**36280**	23690	16790
Prior 10 runs	7570	9020	21410	13660	**20300**
	5370	13900	16830	9110	13880
	7960	8540	32310	15760	7310
	12180	7690	20640	24060	8350
	3880	16600	31040	25420	14520
	6790	12750	24580	15730	19810
	6800	15760	24460	21860	12840
Live session runs	5930	12360	16130	**27240**	4920
	5180	12380	18530	5440	5710
Max	**13700**	**19900**	**36280**	**27240**	**20300**
Mean	**7955**	**11448**	**24341**	**18065**	**11305**

Fig. 5. IEEE CIG 2011 Pac-Man vs. Ghost competition result

The score distribution of our agent is not stable. We predict ghosts' move so the agent can forecast dangerous states before it happens and try to avoid them. However, the prediction is not 100% correct due to the randomness of ghosts' movement. Once the ghosts move in a path out of our expectation, even though our agent can realize it can try to save its life as soon as possible, however, sometimes it is too late to do any rescue. That is the main cause of the instability.

We also found if we changed the some constants in our A*, the game would advance differently. For example, if we increase the minimum distance between Pac-Man and nearest ghost that triggers Pac-Man to go to eat power pill, Pac-Man would eat the power pills too early. The final constants we choose are based on many experiments with different constants' values.

6. REINFORCEMENT LEARNING FOR PAC-MAN

Although most game AI used finite state machine (just as our A* approach). Using reinforcement Learning to build a learning agent is also an attractive topic. This type of agent uses a reward function to determine how good a movement is when it explores the environment and updated its decision set incrementally. In Pac-Man game, the world is fully observable and agent's movement is deterministic.

For Pac-Man game, State-Action-Reward-State-Action(SARSA) is a better choice than Q Learning. The difference between them is Q Learning update function takes into consideration the action the agent actually picks rather than the best action in next state. The other difference is that SARSA doesn't cut of the eligibility trace when an action is performed, this allowed updates to be passed back to earlier states. This ability allows the stochastic nature of the game to be considered as the rewards will be gained stochastically.

The equation below shows the update δ function.

$$\delta = r + \mu Q(s_{t+1}, a^{t+1}) - Q(s, a) \tag{2}$$

SARSA Algorithm is defined as following:

ALGORITHM 1: SARSA Algorithm

```
1. Initialize Q(s,a) arbitrarily and e(s,a) = 0 for all s,a
2. For each episode
   (a) Initialize s, a
   (b) Repeat until s is a final state
       i.   Take action a in s
       ii.  Observe r, $s_{t+1}$
       iii. Choose $a_{t+1}$ from $s_{t+1}$ using policy derived from Q
       iv.  $\delta = r + \mu Q(s_{t+1}, a_{t+1}) - Q(s, a)$
       v.   e(s,a) = e(s,a) + 1
       vi.  For all s,a
                Q(s,a) = Q(s,a) + $\alpha\delta e(s,a)$
                e(s, a) = $\mu\lambda e(s,a)$
       vii. s = $s_{t+1}$ and $a = a_{t+1}$
```

The agent is provided with all the information of the game. So if all the related information is used in state space, our agent would perform very well. However, Pac-Man game's state space is too large, including available pill position, four mazes, four ghost's positions, four possible actions, edible ghost exists or not, power pill exists or not. Agent would need to explore all the state and get the reward for all actions in that state. It's impossible to explore a so large state space, so reducing the state space wisely is where the challenge lies.

The state space we thought would works well is as the following table:

Table 2. State Space for Reinforcement Learning

d1..d4	distance between pacman and four ghosts
dp	distance between pacman and nearest pill
dpp	distance between pacman and nearest power pill
a1…a4	possible actions
jn	junctions count in the shortest path between nearest ghost and pacman

The reward function can be the same as score function, except that we can make losing a life to be -10000.

It's also important to balance exploration and exploitation. By choosing the best action with p = 1- e we can set how often it explores and how often it exploits. We use the following equation, which decreases e to 0 over time. So we do more exploration in the early time and choose the best action when e tends to be 0.

$$e = e_0 - (e_0 * EpisodesNum \ / \ TotalEpisodes) \quad (3)$$

However we didn't have time to implement the proposed reinforcement learning. We may want to spend some time working on it later and to see if this approach is better than A* or not.

7. CONCLUSION

In this project we applied artificial intelligence technology we learned from class to Ms Pac-Man game, including A* search and reinforcement learning. The simulation of the game retains most of the features of the origin game. The Pac-Man package is convenient for us to implement our algorithm and saved us a lot of time in building up the simulator and querying game state.

We implemented the handed-coded finite state machine approach based on our cost model and A* search. It is proved that with well-defined cost model, the agent can have an impressive performance. We recalculate the best path at every game tick, which may be a waste of computational power as some information can be cached and used later. A possible improvement of our approach is to adopt incremental search algorithm like D*.

In fact, most game AI is designed using finite state machine. It is easy to implement and computational speed is fast in real time game. Moreover, the world to game AI is fully observable, making it pretty convenient to apply handed-coded strategy.

We also proposed reinforcement learning approach for Pac-Man game. However, we didn't fully implement it due to the limited time this semester. Actually some articles have proved methodology in machine learning such as neutral network / generic algorithm and reinforcement learning can be used to build game AI, especially the game with stochastic nature.

The future of game AI is still unknown. Pac-Man game is a small test-bed for implementing AI technology. There is much work remains to be done in this area. It would also be interesting to investigate how to combine the hand-coded rule based AI and AI that uses machine learning methodology for a better performance instead of using either one of them.

REFERENCE

[1] Ms. Pac-Man competition official website, http://www.pacman-vs-ghosts.net

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

[3] A. Kalyanpur and M. Simon, "Pacman using genetic algorithms and neural networks," Retrieved from http://www.ece.umd.edu/~adityak/Pacman.pdf (19/06/03), 2001.

[4] M. Gallagher and A. Ryan, "Learning to play pac-man: An evolutionary, rule-based approach," in IEEE Symposium on Computational Intelligence and Games, 2003, pp. 2462–2469.

[5] S. M. Lucas, "Evolving a neural network location evaluator to play ms. pac-man," IEEE Symposium on Computational

Intelligence and Games, pp. 203–210, 2005.
[6] R. T. Hiroshi Matsumoto, Chota Tokuyama, Ritsumeikan University, “Ice pambush 2,” in http://cswww.essex.ac.uk/staff/sml/pacman/cec2009/ICEPambush2.pdf, 2008.
[7] Pac-Man software package, https://s3.amazonaws.com/wcci12/documentation/javadoc/index.html
[8] IEEE CIG 2011 Results, http://dces.essex.ac.uk/staff/sml/pacman/CIG2011Results.html