Pac Man CTF Agent

Yen-Chun, Chiu 603410005

***Abstract*— This paper present methods and details for all aspects, including strategies, sensing methods, evolutionary algorithm, of how I build up a competitive agent for Pac Man Capture The Flag game. It's a multi-player, multi-objective game. There are two agents on each team. The goal is to get food in opponents' territory and bring the food back, while defending opponents to do the same. AI-related technique and knowledge from the paper I read will be adopted to this work.**

I. INTRODUCTION

Pac-Man is a challenging game which has been a game in everyone's childhood. Beside the standard version, there are quite a lot variants of it nowadays. Due to the outdated of the AI contest for Ms. Pacman version, I found a variant called Pac Man Capture the Flag Contest. It is originally a project of an AI class in Berkeley. After developing the required function for their agents, they even hold a contest to compete with each other, and the winner can get additional point added to their final grade!

The reason why I choose it as my final project is that, it includes some convenient API for you to get information for game states and agents. It also contains some example agents, which can be used to train and evaluate. And, it seems suitable and fun. I'll introduce the rule under the following paragraph. Figure 1 shows the overview of what the game looks like.

A. Basic Rules

The Pacman map is now divided into two halves: blue (right) and red (left). You can be at both sides. Red agents must defend the red food while trying to eat the blue food, and blue agents will do the same. When on your own side, your agents are ghosts, which is capable of defending food on your side. When crossing into enemy territory, the agent becomes a Pacman, which is able to get food and capsule in enemy territory.

B. Scoring

As a Pacman eats food dots, those food dots are stored up inside of that Pacman and removed from the board. When a Pacman returns to his side of the board, he deposits the food dots he is carrying, earning one point per food pellet delivered. Red team scores are positive, while Blue team scores are negative. If Pacman is eaten by a ghost before reaching his own side of the board, he will explode into a cloud of food dots that will be deposited back onto the board. In this situation, you will not earn any score.

C. Eating Pacman

When a Pacman is eaten by an opposing ghost, the Pacman returns to its starting position as a ghost. No points are awarded for eating an opponent, and no points will be deducted if a scared ghost is eaten by the opponent Pacman. Notice that only some of the layouts contain power Capsules.

D. Power Capsules

If Pacman eats a power capsule, agents on the opposing team become "scared" for the next several moves, or until they are eaten and respawn. Ghosts that are "scared" are susceptible while in the form of ghosts (i.e. while on their own team's side) to being eaten by Pacman. Specifically, if Pacman collides with a "scared" ghost, Pacman is unaffected and the ghost respawns at its starting position. The capsules and food are not relevant if you are in ghost state.

E. Observations

The definition of two kinds of distance used in this work are: Manhattan distance, the distance which equals to the sum of x and y difference of two coordinates; Maze distance, the actual shortest distance on the maze that we need to go from A to B coordinate. For later refer, I'll use these terms to describe which kind of distance is being used.

Agents can only observe an opponent's configuration (position and direction) if themselves or their teammate is within 5 Manhattan distance. In addition, an agent always gets a noisy distance Maze distance for each agent on the board, which can be used to approximately locate the unobserved opponents. The noisy distance is ±6 of the real distance.

F. Winning

A game ends when one team returns all but two of the opponents' dots. Games are also limited to 1200 agent moves (300 moves per each of the four agents), not moving is also counted as one move. If this move limit is reached, whichever team has returned the most food wins. If the score is zero, this is recorded as a tie game.

G. Computation Time Limit

Each agent has 1 second to return each action. Each move which does not return within one second will incur a warning. After three warnings, or any single move taking more than 3 seconds, the game is forfeit. And there will be an initial start-up allowance of 15 seconds.

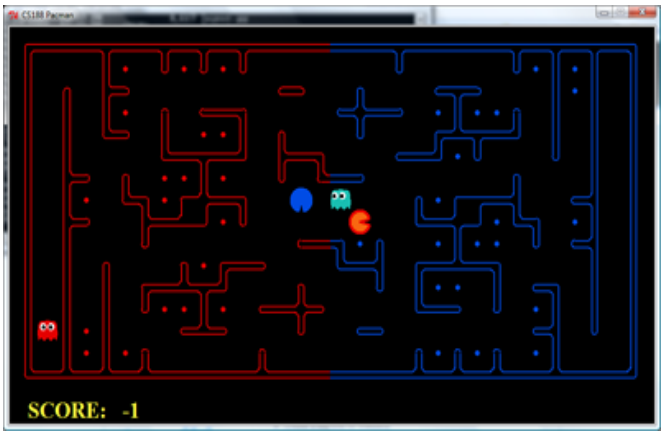


Figure 1: The graphic interface of Pac-Man Capture the Flag

II. Game-relate Strategy

There are some traits that the Pac-Man CTF game is quite similar to the original Pac-Man game: we have to pursue food and avoid getting caught by enemy ghosts. And the capsule works the same, too.

And of course, there are also lots of different. First, the map design is very different. Taking a clear look at Fig. 1, you may notice that, the path on the map forms a tree rather than a graph, which original game's map does. Secondly, you got two agent to operate, which makes cooperation possible. What's more, we get to define ghost agent. And aside from all these, I am sure there is still plenty of traits which are helpful for agent developing lie in this game.

In this section, I'll present the strategy I found and the thoughts behind it while exploring the game, but not all of them will be applied to my agents. As long as I don't change the API that is clearly stated that you shouldn't modify, taking advantage of all other usable API and map information are all allowed.

A. Map difference

The map makes the Pacman easier to get stuck and being caught, especially when you go into an alley. But as long as you are sure that you are able to escape after going in to an alley, it is definitely worth it because there are more food in more dangerous place, this is how all the maps are designed. So we must take that into consideration while pursuing food.

I construct a map which mark the "degree of dangerous" of each cell, as illustration shown in Figure 2. The deeper an alley is, the more dangerous it is.

Ideally, a Pacman will only go into a dangerous zone if the enemy agents are far away, and worth it. Hence, besides degree of dangerous, the distance of enemy is also a factor to decide whether the agent should go in or not. For example, before the agent want to eat the food on the grid with "dangerousness" 5, it must make sure that opponents ghost is at least 5 Maze distance away. And you can also use it as a feature.

B. Middle boundary

The middle boundary line, which decide your status (Pacman or ghost), is an important objective for both attacker and defender: Attackers should view it as an objective after collecting some food. Which means, as a defender, we can just hover around the boundary to prevent the attacker from getting into our territory or going back.

C. Enemy position

In my design, I'll try to get the actual distance of enemies. If it fails, which is the case that both enemy agents are out of observation scope, I'll then use the noisy distance one. And since the noisy distance list will always be the same as long as you are on the same position, we can't infer enemy position by using the noisy distance function for more than once, which is quite reasonable.

However, while poking around with the APIs, I found something interesting: Although you can only get non-precise distance if both you and your teammate are out of 5 Manhattan distance from opponents, just like I mentioned in E. of first section, you can have your opponents' precise position even when they are out of observation range at the very moment when they consume a food on your side because you can see the food disappear! It would be interesting to use this feature to infer opponents' position, as well as having some positive impact on our agent.

D. Time control

When time is about to run out, if you are wining, you can go into full-defensive mode - send back all agents to defend. If the time isn't enough for you to bring back any food, it's meaningless to continue the attack. If you are losing, you can try to make some desperate moves, which has higher chance to die and you normally wouldn't. Otherwise you are going to lose anyway.

E. Cooperating

After some survey, a lot of others' work let an agent defend and the other attacks. But in my opinion, things shouldn't always be that way.

For example, while attacking, you might want to send one agent to lure and attract the defending enemy, and the other to collect the food. It doesn't even matter if the bait agent dies, as long as it buy enough time for the other one to bring back the food. And as a defender, you can try to surround the attacker with two agents.

Maybe we can use one-attack-one-defend strategy as a start, since it seems unreasonable if you go full aggressive or full defensive, but if it is always fixed that way, many other cooperation possibilities will be ignored.

Hence, instead of implementing one attacker agent and one defender agent, which is what most of the others do, two agents will run on only one agent strategy. According to my strategy, in the beginning, one will be the attacker while the

other be defender. But both of them has the capability to act as both attacker and defender. According to the situation, one might change from attacker to defender or vice versa. D. in this section is a good example.

I. Agent Design

Basically, in my agent design, I use an evaluation function to evaluate every available directions, and takes the one with the highest evaluation score. If there are more than one highest scores, select a direction randomly. The concept of the evaluation function formula is the following:

$$F(x, w) = \sum_{n=0}^{k} x_n * w_n \quad (1)$$

, where x is feature and w is weight for the corresponding feature. Features are all fixed, and most of them are a distance like, distance to opponent agent, distance to food, distance to the middle line, etc. We just need to evaluate all possible next step we can take, and they provide a distance object which allow us to get the actual distance between two coordinates on the maze, we shouldn't have to worry about the path, since the distance is the total step of that shortest path.

However, weights should vary as the game state change and should be re-evaluate before each decision of direction. For example, the weight for the feature "distance to middle line" will be influenced by the number of food you are carrying and the time steps left. Because if you are not carrying any food, you wouldn't want to go back to your territory for sure. And the more you carry, or the less time left, the urgency for you to go back increases. So actually the weight for this feature should looks like the following formula:

$$W = w_1 * x_1 + w_2 * x_2 \quad (2)$$

, where W is the variable weight for this feature, and it is the combination of some factor multiply other two features, And those factors are left for us to design.

Above is just an example. Practically, the interaction and scenario in game is much more complicated. The biggest problem will be, how are we going to decide the value of so many factors?

In Jacob and Risto's work [1], they use neural evolution method to optimize the Pacman's behavior. By combining neurons (something similar to "feature" in this paper) with weights and evolving the neuro-network to get a set of weight that is good to use. However, all of them end up to be linear combination. That seems strange, the combination function should have some other forms and may varies from time to time, especially in Capture the Flag version of Pacman.

If not so, it might cost some problems. Take the result of their work, for example: after eating a power pill, their Pacman agent will keep on chasing the vulnerable ghost even if there isn't enough time left for it to reach the ghost, which is a time wasting behavior. Normally if we want to deal with this problem, we judge if there is enough time. If there is, we continue to pursue the ghost. If there isn't, the pursuing force should immediately drop to 0.

In their case, they do have a sensor that tells them how much time left will the ghost be vulnerable. Nevertheless, the force will drop gradually according to the time goes by because that sensor is only controlled by a weight, and that's not what we want. The concept between these two is shown in Figure 3, considering the situation that the time left won't be enough for Pacman to get some vulnerable ghost.

That's the main reason for me to implement my architecture. In my design, the thought of combining

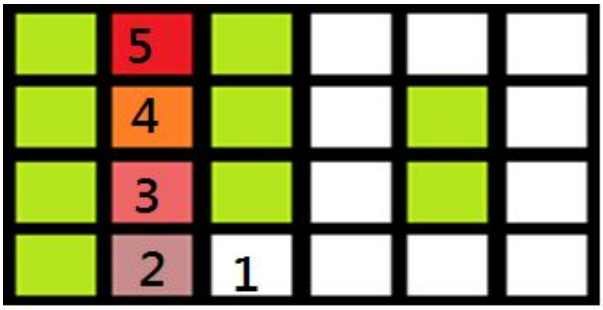


Figure 2. Illustrate the "degree of dangerous". The green cells represent walls. The redder it is, the higher its "degree of dangerous" will be, which means it's more dangerous to go in that deep. A usage example: Suppose my Pacman agent is now at 2. To decide whether I should go in further, I check the enemy distance, and only go in if it's larger than 3, since I need 3 steps to get out of the alley without getting sealed.

weight to feature is quite similar to the original work. But in contrast of their unbiased learning method, I try to make up the interaction between agents myself, and assigned them with uncertain factors. Then I try to use GA to deal with the uncertainty – evolve all uncertain factors until the score of the individuals in the population are saturated. The detail of GA implementation will be describe in the next section.

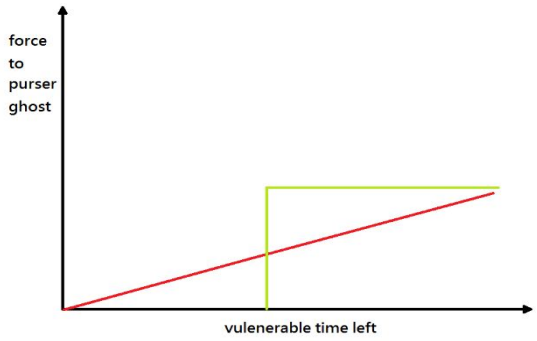


Figure 3. The red one stands for the method in Jacob and Risto's work. The green one is an example what an ideal function might look like.

III. Genetic Algorithm evolution

This section is for the elements of GA. The experimental detail will be described in the next section. The value for each variable will be restricted from -200 to +200 in order to limit the search space.

A. Initialization

We can determine whether the factor is positive of negative, since for example, when you are an attacker, it is very obvious that the weight for the feature "opponents distance" should be positive because the closer you get to the opponent ghost, the higher chance you'll get caught. Using this technique to generate initial population makes the average score much higher.

At the beginning, I manually choose several factors which seems okay to make the agent work properly, and put some with random number with the same sign. This is done by the script "pacman_contest/generate_data.py", which is written by myself.

B. Representation

Just like I had mentioned, I put the uncertain factors into the individuals of GA. It will contain a series of numbers, which are factors for agents. The position order represent a factor for an object. For example, in an individual, the first number is for the factor of distance to enemy ghost, the second number is the factor for distance to food, and so on. Figure 4 illustrates how the representation look like.

C. Parent Selection

Randomly choose two parents in the population to reproduce an offspring.

D. Evaluation

Send the individuals as factor, and use them to play some games. Then use the result as evaluation function. For example, we have an individual "3, -4, 2, 6", representing factors for enemy distance, food distance, dangerousness factor and middle line respectively, and by using it to play the game, we got a score. I use contest01~05capture.lay as evaluation, run an individual on those 5 layout, if we win by x point, we add x to the score. But if we lose by x point, then x point will be deducted from the score. A Tie game has 0 point. We will use the score sum as evaluation.

E. Mutation

First, randomly choose a factor in the generated offspring, and revise its value. Notice that we should keep it the same sign as the one before mutation, otherwise there are a lot more chance to get a bad offspring.

Secondly, add a random number ranged from -5 to +5 to every factor. The creativity is from the thought that, although a child may resemble his parents in many way, the thing that looks quite alike does have some subtle difference in most of the time. It is also reasonable to do so because such a small change one the factors will remain the main behavior. It is also helpful to diverse the individuals, otherwise the whole population would look very alike (only different in a factor), which is quite strange.

F. Termination

Assuming my strategy of behavior function if fixed, the ideal termination will be the saturation of the population, i.e. the average score of each individual nearly fixed after a certain amount of generations. However, due to the limitation of time, it is still not saturated when the deadline comes.

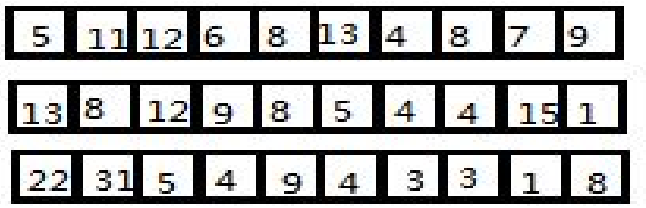


Figure 4. Illustration of GA representation. The numbers are the factors for weights and features in an agent. Each individual represent a factor set for an agent.

IV. Experimental setups

I actually have a lot more thoughts than the experiment and the source code I present. And there are also a lot of place that can be tuned better. However, due to the limitation of time, I've simplify some parts, which will all be listed in this section.

A. Experiment reproduce

The whole project is written in python. Some important functions will be listed in this stage. As to further information, please read the manual or use the help option in the program.

To execute the main program, use: python capture.py, and add options listed here:

1) Option "-l": Assign layout, ex: -l contest01Capture
2) Option "-b,-r": Assign Team for blue/red team, ex: -b ShuaiTeam
3) Option "-h": Show the help page with all options

4) Option "--redOpts,blueOpts": Assign specific agent in the team, ex: --redOpts first=blaagent,second=blaagent
5) –keys0, 1, 2, 3: Control agent with index 0,1,2,3 with keyboard

B. Included feature

You may notice there are some feature with a trailing '2' in its name. It is the square of the original value. For example, feature 'A2' is actually the square of feature 'A'. The inspiration for this implementation is "Newton's law of universal gravitation" [5], which the universal gravity between two object is the square of the distance between them. And this make me think that maybe is force to attract Pacman to the food might also be the same. So I implement this idea to some of the distance based feature to test out if my thoughts is correct.

I classify the features into three types: Distance-based, one-step-to-trigger, and others. The distance based are just the distance to objects like food, opponent. The one-step-to-trigger is for the fact that, if you step on a food to eat it, the food on that grid will disappear, making the distance to the closest food suddenly become far away, letting the Pacman not want to eat the food. This behavior also appears on defenders, it won't eat enemy ghost without additional feature. The others type are something like time left, dangerousness of the position and food distribution on the map, which help agent to accomplish complicate behaviors. Below are the features I implemented with detail described, each of them will be bounded with a correspondent weight.

(1) Distance-based features: Food distance, opponent distance, distance to middle line
(2) One-step-to-trigger: One step to death, one step to food, one step to enemy ghost
(3) Others: Dangerousness, time step left, consecutive food

C. Layouts

All the layouts are inside the "layouts" directory. There are from layouts from "contest01Capture.lay" to "contest20Capture.lay", which are the layouts they use for the competition. In my experiments, I'll use the first 5 to train, and 5~10 to evaluate the result of training. In these layouts, the maze are different from one another, of course. Although there are some other maps with "power pill", notice that all of the contest maps have no "Power pill" on it.

D. Training

To train with GA, I write another program named "trainer.py". It will call the "python capture.py" with the option we desire. It reads the trained data from a file in the directory "data", and after using finishing some games with those data, it save the result – the trained data to another file after this go.

In those files, numbers, which represent the uncertain factor mentioned earlier, will be separated by space. Notice that in the data, there is an additional number is the end of the series, which representing the score and is not part of the factors in an agent. This design is for the fact that one evaluation takes so long and we must record the value down once we had it. For training, I use contest01 to 05capture.lay.

In my read-and-continue structure of the experiment, it is easy for us to see how the population varies, and we are able to stop and continue easily if we find that some tuning is needed. I train our agent against the baseline agents until the deadline comes. The flow of the training process is shown in figure 5.

E. Evaluating the agent

When the training is completed, I run it on the contest layout other than the first 5, which the Pacman had never seen in the training.

Originally, I try to find some agents from other students who previously attain the Pacman CTF contest. I did find some, but they cannot be used since their version of the game is very old and even the game rules are different, and so are the APIs. And unfortunately I really cannot find any agent from other students that is usable, it seems they have strict rule not to spread their agents, making it so hard to find.

In the end, the TA and professor in Berkeley of that AI class didn't reply my e-mail, either. So I just use the one I can get in the original pack. Otherwise I could have compete and train against the best agents in the competition.

The agent I used in my experiment is baseline agent. It acts reasonable, but is too simple. For example, it just try to get the closest food instead of finding a large pack of food, which is more efficient. Well.. that's better than nothing.

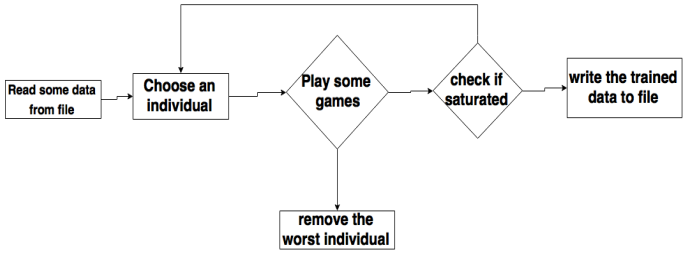


Figure 5 shows the flow of the training process. As you can see, it's actually steps in GA. The whole training system just use file I/O to keep record of the data and help use to train our agents. It is also good to use when you try to tune the interaction function, since you can test the recorded data manually and make some improvement.

V. RESULT

In this thesis, I present my own GA training framework,

combining with the Pacman. The result data is inside the "pacman_contest/data/" directory. I saved the training data once in every two-hundred generations in order to track the effectiveness of my GA. You can see how the population evolve from the data sets. The result is from approximately 2000 generations, which takes around 50 hours of running time. And the result you see below are the individual with the highest score among the population.

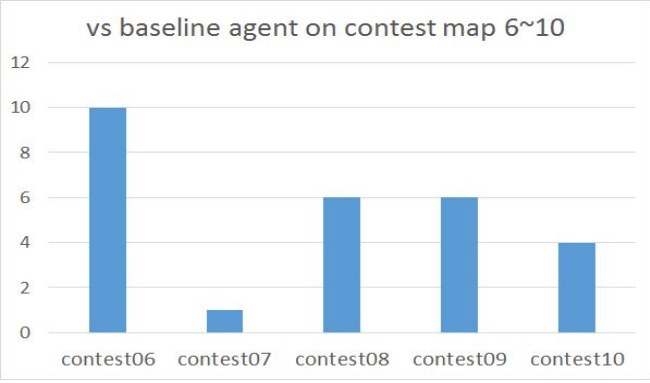
Figure 6 is the training result of GA. Although the training time is not enough for it to fully converge as we can notice from the average score, we can see it is really doing some work.

Figure 7, 8, 9 is the result of my agent vs baseline agents on contest 06~10, 11~15 and 16~20 capture. To my surprise, all of the game scores is positive, indicating that our agent wins all games and does a decent job even on other maps. It's such a pity that I couldn't find anyone to compete with. I really want to see how it does against others' agents.

There are some snapshots of my experiment result in directory "result_snapshot". They are in text because the GUI version will not halt and just disappeared after the game finish. And I have a copy of all my script and agent in directory "my_works".

Figure 6. The orange line is the maximum score in the population, and the blue line is the average.

Figure 7. Competing with baseline agent on map 6~10



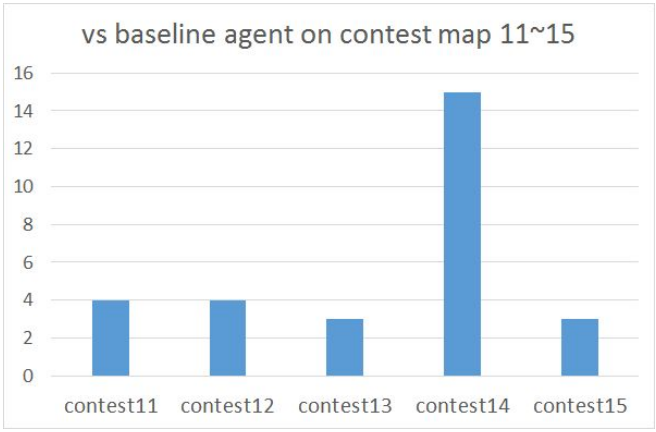


Figure 8. Competing with baseline agent on map 11~15

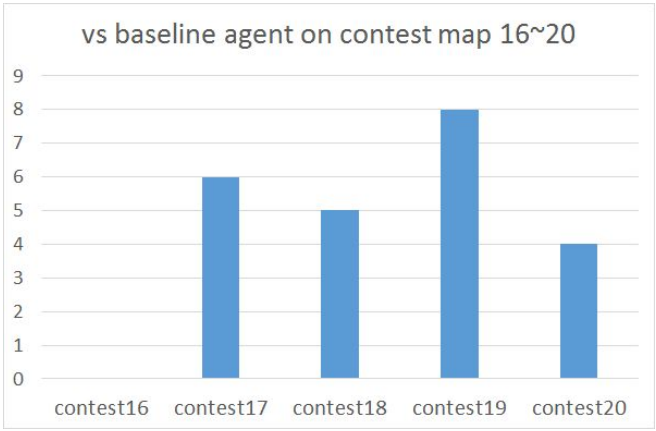


Figure 9. Competing with baseline agent on map 16~20

VI. Conclusion

Honestly, there are still a lot of place that can be improve in my work. By observing the evolution process and debugging, I found some problems.

At the beginning of GA, the initial population in the populations are so bad that makes it almost impossible to generate a good solution by cross over.

Furthermore, due to the randomness behavior in opponent agent used for training, it is not very fair to judge only by the result of one game, since it might beat it by luck. Maybe by running through more games per evaluation may solve this problem, but this also leads to another problem: the evaluation takes too long. I tried my best to reach a balance between fairness and time consumption. Fortunately, the problems are relieved after some adjustment toward the methods.

It now takes roughly 15 second for a game finished, and it has 5 games to run for one evaluation. The result still leaves something to be desired. However, I don't think that there is any way to evaluate an individual without running it on different map. I did start the experiment as soon as possible, but due to the long evaluation time, I still think that more time is needed for the experiment to be more complete.

I also found that in my method, how the human-defined feature functions work influence a lot on the result. If there are some flaws with the feature functions, then it is almost impossible to get a good result, no matter how you implement the GA. And sometimes, due to the constriction on range of sight, it is not that straight forward to realize the desired function.

And it would be better to get some methods to help me with constructing the feature functions. It does need a lot of human work, but the advantage is, one can tune these feature functions if they think something is not quite right. With a structural method like neuro-network, it might be difficult to solve some subtle problems, for example, the one I mentioned in 3rd section.

As to the future work, I am thinking about adding the concept of Pareto front to GA evaluation, which might help. And how to optimize the feature functions might also be an issue. And although there is a little cooperation between my two agents, I still find the cooperation too weak. There must be some way to enforce the cooperation between the two.

This is the first time I have done something like this project. From studying the paper, finding what I should do, organizing the method, conducting the experiment, and presenting my own work as an IEEE format paper, I have really learned a lot. Overall, to me, this final project was quite a fun and challenging experience.

VII. References

[1] Jacob Schrum and Risto Miikkulainen. Evolving Multimodal Behavior With Modular Neural Networks in Ms. PacMan

[2] Brandstetter and S. Ahmadi. Reactive control of Ms. Pac Man using information retrieval based on Genetic Programming

[3] A. M. Alhejali and S. M. Lucas. Using a Training Camp with Genetic Programming to evolve Ms Pac-Man agents

[4] J. Schrum and R. Miikkulainen. Evolving agent behavior in multi-objective domains using fitness-based shaping. In GECCO, 2010.

[5] .wikipedia.org/wiki/Newton%27s_law_of_universal_gravitation