Comparing State-Space and Multi-Agent Search Algorithms for Pac-Man

M. Kubilay Kahveci
Department of Computer Engineering
METU

Mert Gencturk
Department of Computer Engineering
METU

Abstract

Pac-Man is a well-known, real-time computer game that provides an interesting platform for AI researches. In this study, we reviewed state-space and multi-agent search algorithms in the Pac-Man environment. Experimental results are presented that explain the drawbacks and advantages of each algorithm.

Introduction

In 2003s, the advances in computer at both software and hardware levels enabled game developers to create visually immersive game environments (Bourg & Seemann, 2004). After that the technical focus in game design started to be turned towards Artificial Intelligence (AI) in order to create even more immersive games (Mateas, 2003). The aim was to create a level of abstraction, read player's actions and provide intelligent behaviour towards the actions. As a result the "game AI" term appeared consisting of broad range of AI concepts, such as pathfinding, neural networks, rule systems, decision-tree learning etc.

Pac-Man is an arcade game developed by Namco and first released in Japan on May 21, 1980 (*Pacman*, n.d.). It was one of the most popular games in the 1980s and 1990s and is still played as of today. In Pac-Man, there exist a maze filled with pac-dots, Pac-Man and four enemies. The player controls the Pac-Man to eat all the pac-dots. When all pac-dots are eaten, Pac-Man is taken to the next stage. Enemies chase Pac-Man during the game and if an enemy catches the Pac-Man, a life is lost and level restarts.

Pac-Man is tought to be an interesting platform for AI research because it not only provides deterministic, stochastic, partially informed and adversarial problem setting; but it can also be considered as a non-planar traveling salesman problem since eating all the pac-dots in as few steps as possible is one of the aims of the game (Russel & Norvig, 2002).

In this study, we have developed different kinds of state-space search and multi-agent search algorithms and injected them to the Pac-Man environment. The aim is, however, not to come up with a new algorithm or solution. Since it is the first time that the authors of the paper have stepped in Artificial Intelligence domain, the aim is to observe the effects of different approaches, compare them and propose the best solution. The rest of this paper is organized as follows. First we give a summary of the survey in this area and then explain the algorithms that we developed. After that we present the evaluation results of the different algorithms, comparisons among them and conclude the paper.

Related Work

The studies on applying AI techniques to Pac-Man have been done for many years. We are actually not in position to state the weak points in current AI implementations of Pac-Man but considering its game environment Pac-Man is a good choice to apply a new technique or approach on it in order to see whether it provides an enhancement or not. Therefore, it has been in interest of researchers for many years. Common implementations of Pac-Man mainly consists of state-space search, adversarial search, Markov decision processes, reinforcement learning and probabilistic tracking. Recent studies have been focused on changing an approach in one step aiming to receive more effective results. For example, Gallagher and Ryan implemented a dynamic strategy where agent learns the maze adaptively, rather than by a pre-programmed maze-solving method (Gallager & Ryan, 2003). In another study Wirth and Gallagher developed an agent based on an influence map model using random and systematic global exploration and a greedy algorithm (Wirth & Gallagher, 2008). It is possible to find a large number of studies in the literature in this regard.

In addition to these studies, Pac-Man is also used as a way to teach Artificial Intelligence in universities. For example, Computer Science Division of University of California, Berkeley implemented Pac-Man projects to teach foundational concepts of AI in CS188 course(DeNero & Klein, 2010). The projects are made available to the computer science community for educational use and have already been adopted by several universities. As noted earlier this is the reason that this study is focusing on Pac-Man.

Algorithm Definition

We carried out the work of implementing a multi-agent system for playing Pac-Man in two steps: State-space search and multi-agent search. State-space search algorithms have been used to solve navigation problem of Pac-Man, while

multi-agent search algorithms have been used to model Pac-Man as a multiplayer game. Details of each step is explained in following subsections.

State-space Search

In this step we have implemented depth-first search (DFS), breadth-first search (BFS), uniform cost search, and A* search algorithms for Pac-Man agent to find paths through the maze world to eat the pac-dot in a particular location. We have tested the results of each algorithm in the same maze where a single pac-dot is placed in a fixed position and traced the path Pac-Man agent followed. The results are presented in the next section. Each algorithm is actually very similar. They differ only in the details of how fringe is managed. For instance, fringe is managed with $Stack$ data structure in DFS, $Queue$ data structure in BFS, and $PriorityQueue$ data structure in both uniform cost search and A* search. In uniform cost search, the cost of each step is assumed the same. However, it is possible to encourage Pac-Man to find different paths by changing the cost function. In A* search, we have used Manhattan distance as heuristic function. The sample code segment is presented in Figure 1.

```
def depthFirstSearch(problem):
    fringe = util.Stack()
    visited = set()
    start = problem.getStartState()
    fringe.push((start, []))
    visited.add(start)

    while not fringe.isEmpty():
        state, path = fringe.pop()
        visited.add(state)
        if problem.isGoalState(state):
            return path
        moves = problem.getSuccessors(state)
        for move in moves:
            position, direction, cost = move
            if position not in visited:
                newPath = path[:]
                newPath.append(direction)
                fringe.push((position, newPath))

    return []
```

Figure 1: Sample code segment for state-space search

Multi-agent Search

In this step we have implemented reflex agent, minimax, alpha-beta pruning and expectimax algorithms as well as designed proper evaluation functions to let Pac-Man agent to play in a maze including ghosts that move randomly. In reflex agent algorithm, the Pac-Man agent has 5 options to move in a single state: left, right, up, down, or none. We first calculate the value of each move according to an evaluation function, and then move the agent to the best position. While calculating the value and deciding on which position is best, we consider the distance to pac-dots and ghosts. The idea is to go towards the pac-dots, run away from the ghosts. Therefore, proximity to a pac-dot means positive value whereas proximity to a ghost means negative value if it is not scared. In Pac-Man, ghosts are said to be scared (for a short time) when the Pac-Man agent eats bonus pac-dot. In this case ghosts cannot kill the Pac-Man, hence Pac-Man does not need to run away from them, which means no negative value. In minimax algorithm we have constructed a tree where MAX nodes represent the Pac-Man agent and MIN nodes represent the ghosts. The algorithm is generic enough to work with any number of ghosts. In addition, it is also possible to give an arbitrary depth value which specifies how many times the Pac-Man and ghosts will be moved while calculating the node values, hence the best action. Evaluation of smaller and bigger depth values will be explained in next section. In alpha-beta pruning, we improved the agent to explore the minimax tree more efficiently. The purpose is to speed-up the Pac-Man agent so that it eats all the pac-dots in a shorter time. In our implementation we pruned the nodes on greatness but not on equality. The minimax and alpha-beta pruning algorithms works well but they both assume that ghosts (adversary) make the optimal decisions, but this is not always the case in real life. In expectimax, we make probabilistic assumption. We assume that ghosts do not always take the min value to move, instead they take the average value. We first calculate all successor states and then take the average of them. There is no change in max calculation though.

Experimental Evaluation

Initial experiments for search algorithms have been conducted on simple maze with a single goal square, one agent and no opponents. The aim is to analyze the agent behaviour in the Pac-Man world. Figure 2, 3, 4 and 5 shows the states explored, and the order in which they were visited for BFS, DFS, uniform cost search and A* search navigation respectively. Brighter red means earlier exploration. In all settings, there is a single goal state which is very deep in the state space. When using breadth-first approach, the agent explores almost all squares before reaching the goal state. However, when it uses depth-first approach, since it can go deeper towards the goal state quickly, agent finds the goal state without exploring many squares in the maze. This shows the fundamental difference between those two approaches given the solution is in deeper nodes. Since the action cost is constant in these setups, uniform cost search behaves exactly like BFS. At each turn, the agent explores the node with the lowest cost, which happens to be the one in breadth first order. Figure 5 shows how A* can be an improvement on uniform cost search even with a very naive heuristic function such as Manhattan Distance. Although agents starts moving very similarly at the beginning (this is due to selected heuristic function), after some point A* does a good job of preventing non-promising paths.

As shown in the figures above DFS has given better result than the others while reaching the goal state. However one can ask the question that why is DFS is more effective than A* because most of time A* works better than DFS. The reason might be the fact that the pac-dot that the agent

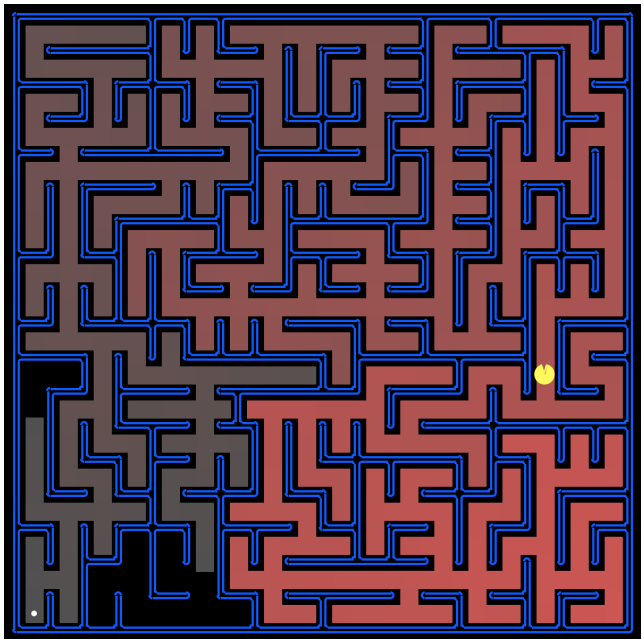

Figure 2: Breadth-first Search

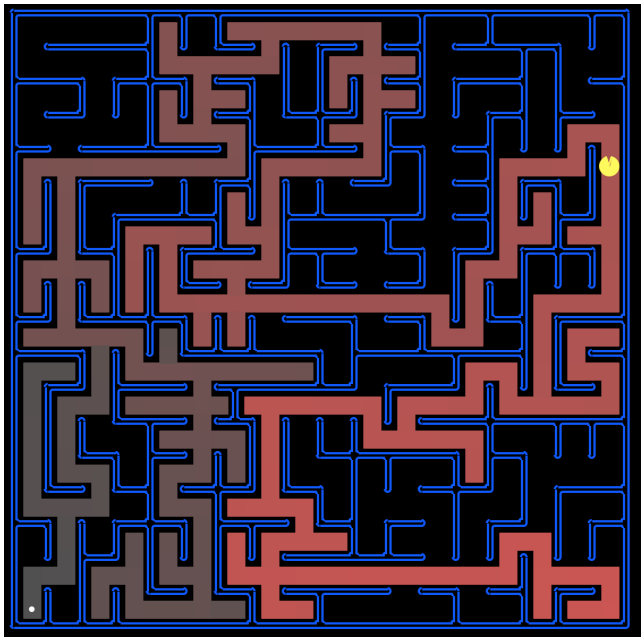

Figure 3: Depth-first Search

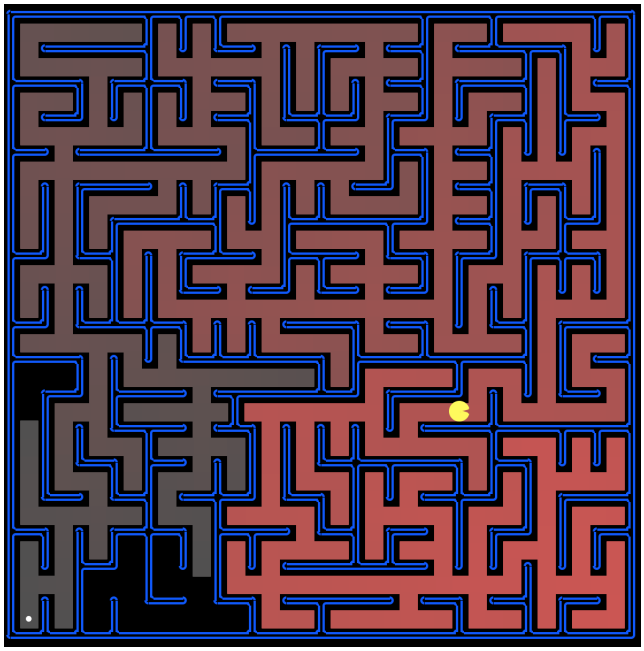

Figure 4: Uniform cost Search

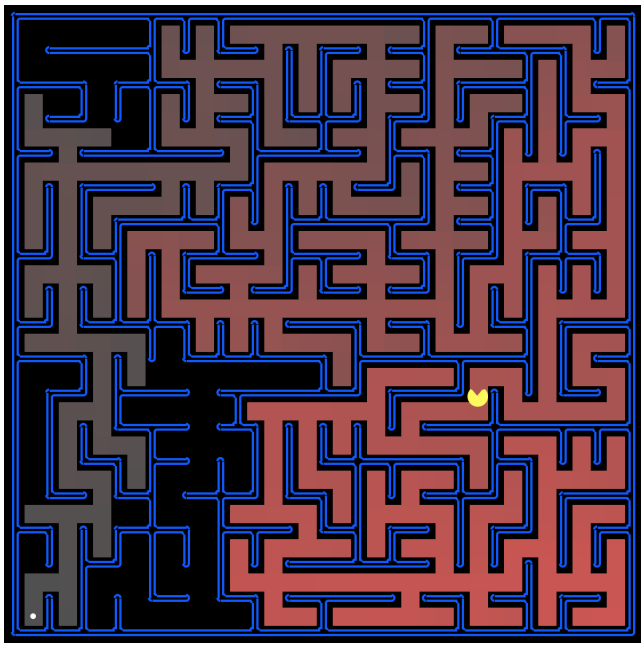

Figure 5: A* Search

is trying to reach in the figures above is the left most leaf node of the tree. What if it is not in the left most node? Will DFS still give better result than A*? In order to see this, we have run A* search and DFS algorithms one more time on a different maze where the goal pac-dot is located in the middle of the tree. As shown in Figure 6 and 7, the agent has explored less squares in A* search than DFS before reaching the goal state. Since the probability of a goal pac-dot to be located in the middle part of the tree is much more higher than left most part of the tree, we can conclude that A* is the most effective search algorithm to reach the goal state.

By running tests among state-space search algorithms we were able to observe agent behavior against a goal state. However, in real Pac-Man world the goal state is not so simple and there are more than one agent in the environment. Pac-Man shall consider reaching the goal state by eating

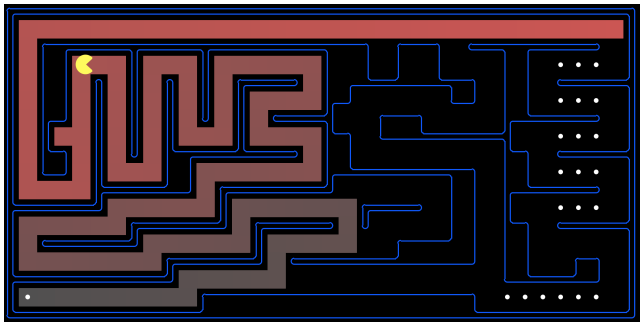

Figure 6: Depth-first Search, pac-dot in the middle

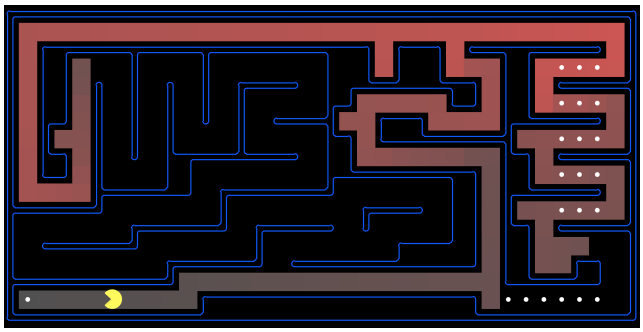

Figure 7: A* Search, pac-dot in the middle

pac-dots in the maze while also preventing ghost agents. We have also compared different multi-agent search algorithms to simulate a real life setup.

Reflex Agent

As it select actions based on the current perception, this agent can only be considered as a baseline comparison. The main problem with reflex agent is infinite loops. With simple evaluation functions they are often unavoidable. We were able to observe that in our tests. We have evaluated reflex agent in two different mazes with single ghost. In open mazes (Figure 8), which contain no obstacle at all, Pac-Man were able to win 100 out of 100 games. However, in a more complex field with obstacles (Figure 9, 10, 11) the agent suffered from infinite loops very often resulting in timeouts. This is quite reasonable considering that evaluation function pays equal importance to pac-dots and ghost. The agent usually got stuck in a field that is away from pac-dots. It couldn't get closer to eat them due to ghost threat.

Look-ahead Agents

Since look-ahead agents evaluate future states as opposed to reflex agent evaluating actions from the current state, algorithms in this part performed better than reflex agent on complex mazes with a few exceptions.

As seen in Table 1, look-ahead agents performed significantly worse than reflex agent in an open maze. In open classic maze, Pac-Man were good at not dying but quite bad at winning. Pac-Man often moved around without making any progress. Sometimes, even though it is right next to a pac-dot, it didn't eat it because it didn't know where to got after eating that pac-dot. This case were observed more with Minimax and AlphaBeta agents, which due to the fact that these algorithms assume ghosts (adversary) making optimal decisions. However, this is not the case for our tests; ghost agents move random. Expectimax algorithm cleaned up these issues by taking an average of possible decisions to be made by adversary. This assumption is closer to real life.

Table 1: Winning percentages of multi-agent search

Maze/Agent	Reflex	Minimax	AlphaBeta	Expectimax
Open	1.00	0.4	0.4	0.7
Small	0.2	0.5	0.4	0.4
Medium	0.4	0.6	0.6	0.6
Tricky	0.2	0.4	0.3	0.7
Minimax	0.1	0.6	0.6	0.8
Trapped	0.2	0.0	0.0	0.8

In the trapped classic maze, the evaluation results were interesting. In this setup, Pac-Man is trapped between two ghosts. Since ghosts are not that intelligent (random movement), there still is a chance of winning the game. However, Minimax algorithm believes that the death is unavoidable. Because an intelligent adversary would easily make the optimal decision. Since there is a constant penalty for living, Minimax algorithm tries to end the game as soon as possible and directly moves towards ghosts. That's why Minimax and AlphaBeta agents were not able to win a single game in trapped maze. Even a reflex agent performed better thanks to randomness. Again, by making more proper assumptions about the adversary Expectimax agent performs considerably well in that case. It is aware of the fact that ghosts will not make the optimal decisions which would end the game right away.

On boards such as small and medium classic, look-ahead agents were not very successful compared to reflex agent. This is mostly due to not taking any risks. In these mazes, games were much longer for look-ahead agents as they calculate future moves and prevent ghosts well. However, they were not able to clean up pac-dots and eventually timed out. Because reflex agent didn't consider future moves, it made much more risky moves, which sometimes brought good results. This issue can be overcome with a better evaluation (scoring) function for Minimax algorithms.

Minimax classic layout is specifically designed for look-ahead algorithms to perform better. The results were not surprise.

Depth and Complexity

All results for look-ahead agents presented in Table 1 are run with a depth value of 3. This parameter defines how further in the future should the algorithm consider when making a

decision. Naturally, if you increase the depth values algorithms start to make better and better decisions. However it is a trade-off between runtime complexity and precision. With larger depth values, agents spend more time before making a decision. In our experiments, we observed that algorithms perform better until a certain point at which winning percentage settles. This point is around depth of 7 moves for Minimax and AlphaBeta agents, 8 for Expectimax agent.

For a real-time agent, it is not convenient to have a depth value greater than 4 due to time constraints. Agent spends approximately 1 second for each move at that level. After that point, it becomes a bottleneck.

We also observed that AlphaBeta pruning provides a significant gain in terms of time. At depth 3, it runs as fast as Minimax algorithm at depth 2. By implementing a simple pruning logic, we were able to cover the cost of one more move to consider in the future.

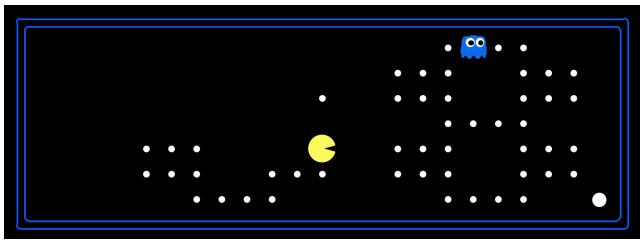

Figure 8: Open classic layout

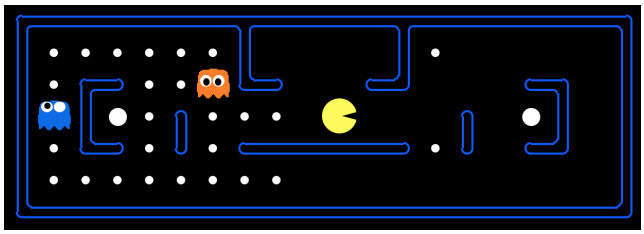

Figure 9: Small classic layout

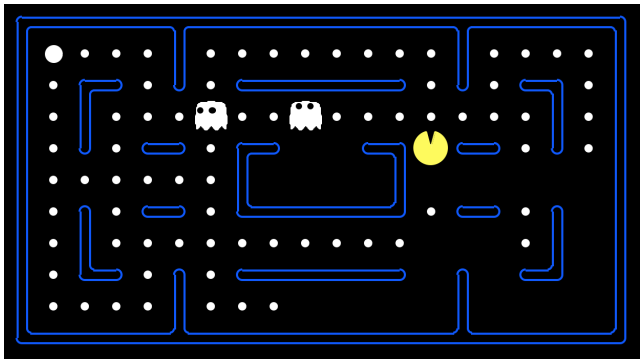

Figure 10: Medium classic layout

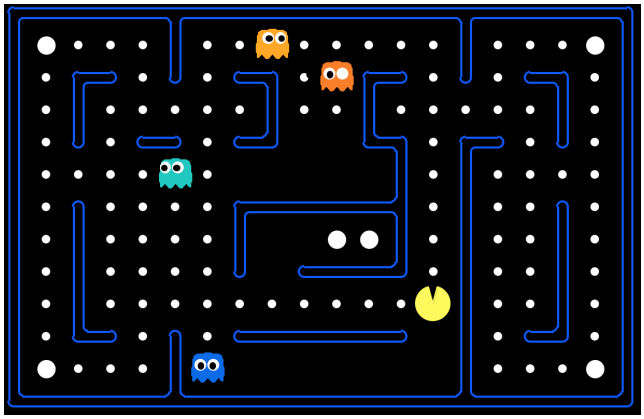

Figure 11: Tricky classic layout

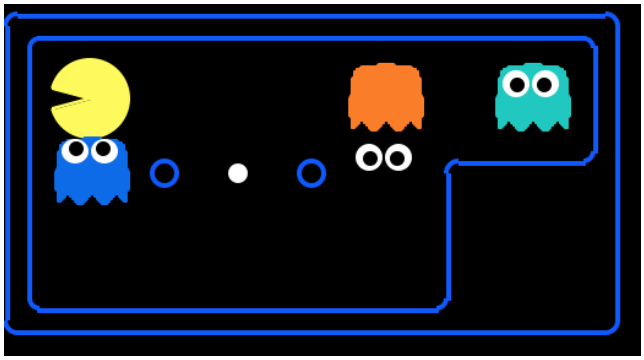

Figure 12: Minimax classic layout

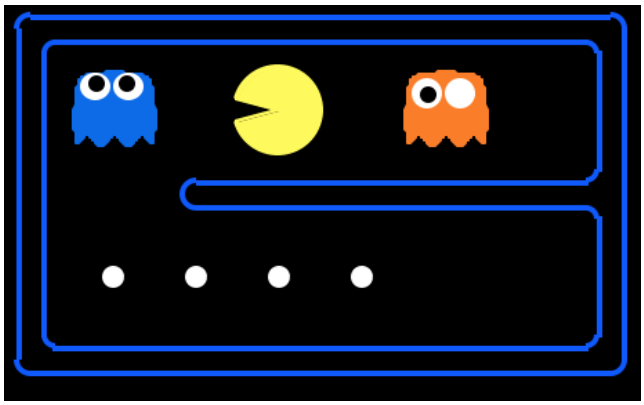

Figure 13: Trapped classic layout

Conclusion

In this project, we conducted a review of state-space and multi-agent search methods in the Pac-Man environment. State-space search algorithms are considered for decision making for navigation of the Pac-Man. Multi-agent search algorithms, on the other hand, are considered for a real life game playing experience, where the environment contains multiple agents and adversaries. By implementing different methods, we were able to grasp the details of each one.

Experiments on various setups with variable parameters made the shortcomings and advantages of each one very clear. We inferred that if the adversary acts random, a simple reflex agent may end up in better states by making risky moves. However, if the field is large and complex enough you have to have more intelligent algorithms that can simulate adversary behavior with randomness. That's why Expectimax agent performed better in many cases. If ghosts were making optimal decisions, Minimax and AlphaBeta agents may have had closer winning percentages.

The evaluation also shows that the decision making depends on scoring function, too. With a better evaluation function, reflexive agent may prevent infinite loops and Minimax variants may be more attacking and better at winning.

Future Work

The first thing that can be done to improve the current implementation is to implement some reinforcement learning algorithms so that Pac-Man agent can learn from experience. Q-learning is a good choice for the agent to learn by trial and error from interactions with the environment. In addition, current implementation can be enhanced by changing some parts such as heuristic function in state-space search and evaluation function in multi-agent search. For example, although Manhattan Distance is not a bad choice and results well, it is still possible to create a better heuristic function so that Pac-Man agent can eat all the pac-dots in a more effective way.

References

Bourg, D. M., & Seemann, G. (2004). Ai for game developers. O'Reilly Media, Inc.

DeNero, J., & Klein, D. (2010). Teaching introductory artificial intelligence with pac-man. In *Proceedings of the symposium on educational advances in artificial intelligence.*

Gallager, M., & Ryan, A. (2003). Learning to play pac-man: An evolutionary, rule-based approach. In IEEE (Ed.), *Evolutionary computation* (Vol. 4).

Mateas, M. (2003). Expressive ai: Games and artificial intelligence. In *Proceedings of level up: Digital games research conferenc.*

Pacman. (n.d.). `https://en.wikipedia.org/wiki/Pac-Man`. (Last visited on May 2016)

Russel, S., & Norvig, P. (2002). Artificial intelligence: A modern approach. Prentice Hall Series in Artificial Intelligence.

Wirth, N., & Gallagher, M. (2008). An influence map model for playing ms. pac-man. In IEEE (Ed.), *Computational intelligence and games.*