Teach the Pacman with Searching and Optimization

Zilong Liang
School of Mathematical Sciences
zlliang15@fudan.edu.cn

1 Introduction

How to teach a Pacman to find a path to his success? In the context of Artificial Intelligence, giving wisdom to the Pacman can be formulated as a search problem. Figure 1 shows a basic version of the game, where the map is a table of positions and the Pacman can move towards four directions. Obviously, it is intuitive to adapt graph search algorithms here.

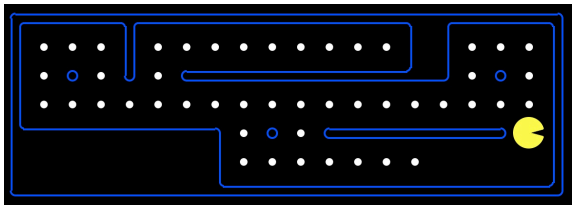

Figure 1: An example of Pacman game

However, the goal of the Pacman often varies, like looking out a shortest path to a fixed position, eating all the dots or getting rid of all the ghosts. As a result, we have to help the Pacman always keep his goal in mind. Then, solution of the problem can be divided into the following two steps:

- Construct an abstract graph (or state space), including the map itself and status of goals. Fortunately, in this project we can access the information of the Pacman world at any time, like the positions of walls and food dots.

- Adapt graph search algorithms on the abstract graph, to find a path with least cost to the goal. In this project, least cost often means fewest actions.

According to the project instruction, we should formulate several goals and implement several search algorithms. To make these two parts cooperate seamlessly, we define the problems according to a single interface in `searchAgents.py` and implement general versions of search algorithms in `search.py` which is able to solve any problem through the interface.

Consequently, this project consists of four working parts:

1. Read the existing codes and understand the framework that every formulation of search problems should follows.

2. Implement general versions of Depth First Search (DFS), Breath First Search (BFS), Uniform Cost Search (UCS) and A* Search.

3. Formulate two search problem that the Pacman should touch all four corners of the maze and the Pacman should eat all the food dots. Also, we should design appropriate heuristics for these problems.

4. Solution of suboptimal search, due to the heuristic-driven A* Search is hard to find the optimal path sometimes.

2 Framework of Search Problems

In line 22–62 of `search.py`, an abstract class gives methods that every search problem has to implement. And by reading the problem class `PositionSearchProblem` in `searchAgents.py` which defines a problem that the Pacman should find the shortest path to a fixed position, we can understand data structures used in the problem classes.

Firstly, the state space consists of tuples of states, every tuple including sufficient information like current position and positions of rest food dots. A start state is given by the method `getStartState`. Then, the method `getSuccessors` gives a list of successors of a state, which is a list of tuples in the format `(next state, move direction, move cost)`. Through this method, we can construct the state space recursively (lazy loading), avoiding compute the whole space at the beginning, which is often impossible in finite time. Finally, the method `getCostOfActions` returns total cost of a list of actions, and `isGoalState` determines the goal state, which is the terminal rule of a search algorithm.

3 General Implementations of Graph Search Algorithms

3.1 Structure of Search Algorithms

Graph search algorithms here follow similar structures, so we define a single primary framework in `search.py`, starting from line 121. According to the project instruction, every algorithm receives a problem object implementing the interface described in Section 2. Each of them iterates the state space recursively using data structures given in `util.py`, and returns a list of directions from the start state to the goal. Note that to make it *complete*, we should implement graph (instead of tree) search algorithms here, so every state of the state space would not be visited twice.

3.2 DFS, BFS and UCS

Graph search algorithms always simulate actions in the graph or maze, moving to a point, marking frontiers in a frontier set, and moving to another point in the frontier set. The only differences appears in the *order* of the frontier set.

Firstly we consider the situation that the Pacman should achieve his goal in least actions, which means the cost of every step equals. Depth First Search (DFS) uses a LIFO stack to store frontiers, which means always selecting the most recently marked frontier as the next step. As a result, it would try a single path to the goal, or get back to another path when meeting a dead end. In this way, the search algorithm would stop just when it meets the goal state. Consequently, the solution may not be optimal.

On the contrary, Breath First Search (BFS) uses a FIFO queue, which parallel tries the paths with same depth until meet the goal state. As a result, it would get the optimal solution.

Then, when we consider the cost of each step and want to find a path with least cost, it is straightforward to adapt BFS to Uniform Cost Search (UCS), which uses a priority queue to choose and pop the frontier with least cost. Here we use `getCostOfActions` in the problem objects to implement this data structure. As a result, our algorithms works well with agents using special cost models in `searchAgents.py`.

3.3 A* Search

A* Search is a variant of UCS that introduces a term of *heuristic*, which is to say, the priority queue of frontiers sort items according to the evaluation function

$$f(k) = g(k) + h(k), \tag{1}$$

where $g(k)$ is the cost of the path from start state to the node k and $h(k)$ heuristic estimate of the cost of achieving the goal from k.

Briefly, a heuristic is an estimate of *future* cost. Giving this term to the search algorithm can inform it with a direction to the goal in some ways, and decrease frontier expanding numbers. To guarantee the completeness and optimality, we have

Theorem *If the heuristic function is non-negative and consistent, which means for all states A and B, we have*

$$h(A) - h(B) \leqslant \text{cost}(A \to B),$$

then the A graph search is complete and optimal.*

In the A* Search, designing a heuristic that is consistent and estimates the real cost well is significant. For hand-crafted heuristic design, we can begin with problem relaxations, or divided our problem into subproblems. Also, heuristic should be easy to compute because we would refer to it frequently. In the following part of discussion, we will discuss our heuristic designing in detail, and prove their consistency.

4 Formulation of the *Corners Problem*

4.1 Design of the State Space

According to the project instruction, in this part of problem, the goal of the Pacman is find a shortest way to touch all the four corners of the map. Firstly we should formulate the problem in `CornersProblem` of `searchAgents.py`.

To inform the Pacman where he is, our state space should consists of the position of the Pacman and whether each of the corners has been touched. So intuitively, we can store the state in a tuple: `(position, isTouched)`, where `isTouched` is a tuple of four boolean values, representing whether the certain corner has been touched. We implement this straightforward thought in the methods `getStartState` and `getSuccessors` to build the state space.

Then, to determine the goal state, we just check the tuple `isTouched`. If all of the elements are `True`, it is judged as a goal state.

4.2 Design of the Heuristic

For the first time, we now consider the designing of heuristic functions. How can we make a good estimate of future costs? We study our state space and begin with relaxations.

To solve the *Corners Problem*, the Pacman only needs to touch all the four corners. It is obvious that if there are no walls, the cost of achieving the goal is the manhattan distance bwtween the Pacman and the nearest corner, and the distance from the corner to the others (like Figure 2).

This heuristic is a simple estimate according to the relaxation of the original problem. Now we have to prove its consistency.

Proof. We only need to prove the situation that after a single step the Pacman moves from A to B, then $h(A) - h(B) \leqslant 1$. The other situations can be proven inductively. Moreover, if the number of untouched corners is fixed, the manhattan distance from the nearest corner to the goal state is also fixed. As a result, we only need to prove the manhattan distances from the Pacman to the nearset corner $m(\cdot)$ satisfies $m(A) - m(B) \leqslant 1$.

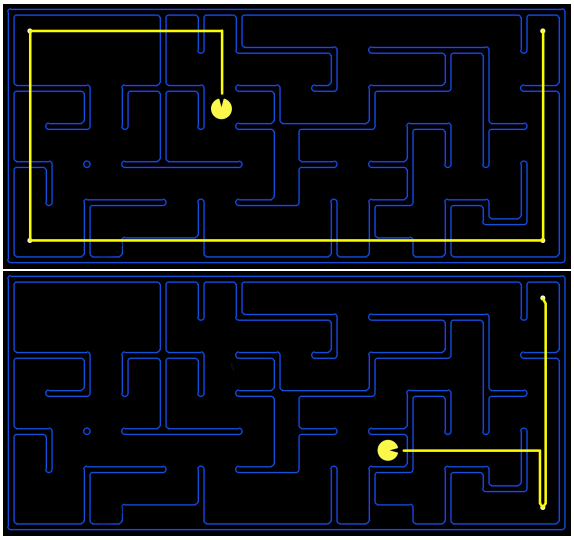

Figure 2: Heuristic of *Corners Problem*

If the moving from A to B reduces or increases the m-value without changing the nearest corner, the inequality satisfies obviously. Now we consider the moving changes the nearest corner, P denoting the former nearset corner from A and $\tilde{P}$ the latter nearest corner from B, then we can conclude that

$$m(A, P) \leqslant m(B, P),$$

then according to the definition of the heuristic function, we can get that $m(A) - m(B) \leqslant 1$. Then we can prove the consistency inductively. □

4.3 Experiment and result

We take an experiment between UCS and A* algorithms in the *Corner Problem*. The result is shown in Table 1.

Table 1: Search nodes expanded number of *Corner Problem* via UCS and A*

Layout	tinyCorners	medium	big
UCS	252	1966	7949
A*	154	**692**	1725
Proportion	61.1%	35.2%	21.7%

From the result, we can see that giving heuristic to search algorithms can inform it the directions to the goal, and decrease the number of expanded nodes. The more complex the map is, the larger proportion of the expanded nodes the A* Search can avoid.

5 Formulation of the *Food Search Problem*

In the *Food Search Problem*, the goal of the Pacman is to find a shortest path to eat all the food on the map. The formulated problem class `FoodSearchProblem` has been implemented for us in `searchAgents.py`. The state space is setted as tuples in the format `(position, food grid)`, where `food grid` can be converted to a list of coordinates of the rest food dots.

Note that it is proven that this problem is NP-hard, which means it is hard to find an algorithm with polynomial time complexity. Now we try to use A* and to find a relatively more efficient algorithms to solve the `mediumSearch` problem in short time.

5.1 Greedy Search

Since this problem is NP-hard level, so an appropriate suboptimal solution should be raised up as an approximation. According to the project instruction, we implement a greedy search agent in `searchAgents.py`, as the first step to solve the *Food Search Problem*. And in the following parts, we discuss some optimal algorithms for small-scale problems and better suboptimal algorithms for medium or large problems.

5.2 A* Search

Firstly, we want to have a try on A* Search to solve the *Food Search Problem*, and the core mission is to design an appropriate heuristic function.

5.2.1 Heuristic 1: H_{max}

One of the intuitive thoughts is to take the maximum manhattan distance between the current position and the food dots (shown in Figure 3). Here we take this heuristic function H_{max} as a baseline, and leave out the consistency proof.

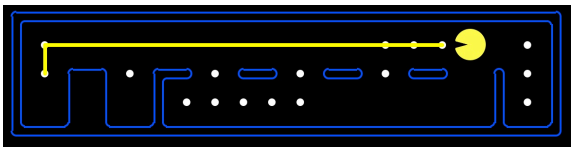

Figure 3: Max heuristic of *Food Search Problem*

5.2.2 Heuristic 2: $H_{\text{min-max}}$

Another idea sources from that we can approximate the problem better if we take the walls into consideration. In `searchAgents.py`, we implement a helper function `mazeDistance` that computes and store the actual distances between food dots, thanks to BFS. Then, since we cannot compute the maze-distance every step using BFS, we can compute the minimum maze-distance between current position and the food dots plus the maximum maze-distance betwwen that dots and the rest dots. Apparently, this heuristic function $H_{\text{min-max}}$ is also a relaxation of the future problem. Now let's prove its consistency.

Proof. Also, we only need to prove the situation that after a single step the Pacman moves from A to B, then $h(A) - h(B) \leqslant 1$. Now we consider $H_{\text{min-max}}$ as two separate parts discussed above:

$$H_{\text{min-max}} = m(A, F_{\text{min}}) + m(F_{\text{min}}, F_{\text{max}}),$$

where

$$F_{\text{min}} = \operatorname*{argmin}_{f \in D} m(A, f),$$
$$F_{\text{max}} = \operatorname*{argmax}_{f \in D - F_{\text{min}}} m(F_{\text{min}}, f),$$

Here D is the set of food dots, and $m(\cdot)$ represents maze-distance. Note the fact that if F_{min} does not change from A to B, then $m(A, F_{\text{max}})$ would not change, either. So $H_{\text{min-max}}$ is consistent in this situation.

If the moving from A to B changes F_{min}. According to properties of the distance, if there is a dot C, then

$$m(F_{\text{min}}^{(A)}, F_{\text{max}}^{(A)}) \leqslant m(F_{\text{min}}^{(A)}, C) + m(C, F_{\text{max}}^{(A)}).$$

Here we assume that F_{max} would not change, which is the most common situation, then we take the new $F_{\text{min}}^{(B)}$ as the C, and note that $m(A, B) = 1$

$$m(F_{\text{min}}^{(A)}, F_{\text{max}}) \leqslant m(F_{\text{min}}^{(A)}, F_{\text{min}}^{(B)}) + m(F_{\text{min}}^{(B)}, F_{\text{max}}).$$

Then according to he definition of $H_{\text{min-max}}$, the consistency exists. □

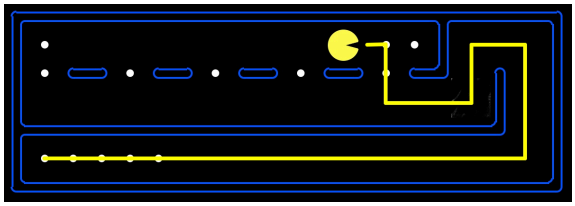

Figure 4: Min-max heuristic of *Food Search Problem*

5.3 Ant Colony Optimization (ACO): Try to solve the `mediumSearch`

After several attempts, it is found that search algorithms are hard to solve the medium-scaled *Food Search Problem*, because dimension of the state space increses exponentially with the food dots. So, several approximation algorithms are held to solve this class of problems, using iteration approaches. Here, we have a try on Ant Colony Optimization (ACO).

ACO is a probabilistic technique for graph search problems like Pacman, especially large-scaled problems like `mediumSearch`. It is based on the behavior of ants seeking a path between their colony and a source of food (it is found that ants always choose the shortest path through the pheromone mechanism). Using the iteration stategy, theorically it can approximate the optimal solution in polynomial time.

We implement this approach in the bottom of `search.py` and create a helper search agent in `searchAgents.py`. Hyperparameters of this program are defined in line 156–162. To run the ACO process, we can use the command: `python pacman.py -p ACOSearchAgent -l mediumSearch`.

5.4 Experiment and result

We firstly carried out a simple comparison among UCS and two A* Search on `tinySearch` and `trickySearch`. Table 2 shows that the appropriate heuristics can also hugely decrease the nodes expanded number in *Food Search Problem*. Disappointingly, none of the approaches can solve `mediumSearch` in 60 seconds.

Table 2: Search nodes expanded number of A* algorithms with different kind of heuristic function

Layout	tinySearch	trickySearch
UCS	5057	16688
A* (H_{max})	2468	9551
A* ($H_{\text{min-max}}$)	1309	**2030**

So, finally we experiment ACO on `mediumSearch` and get a path cost **152**, after 25 iterations (57 seconds on my PC), which is the optimal solution for this specific problem. Figure 5 shows the decrease of the cost along with iterations on `mediumSearch`. We can see that this algorithm can converge to the optimal solution using iteration strategies, and arrive this sub-optimal solution in polynomial time.

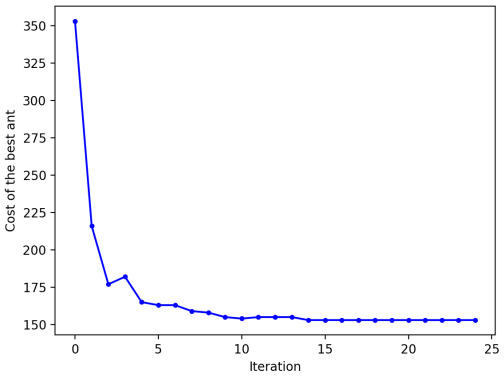

Figure 5: Ant Colony Iterations

6 Conclusion

In this project, we implement DFS, BFS, UCS and A* Search for several Pacman problems, and find that appropriate heuristic functions can inform the agent to arrive its goal easily. Although design of the heuristic counts a lot, it is hard for the classical search algorithms to solve medium-scaled *Food Search Problem* problems, which is the basic demand of the Pacman game. As a result, Greedy Search, Dynamic Programming, Evolutionary Algorithms such as ACO discussed above, etc., could be carried out to approximate the optimal solution, and among them, Ant Colony Optimization may be a good choice for medium-scaled *Food Search Problem*.