Pac-Man AI: Competition Write-up and Technical Report

Qiyuan Zhuang[1]

***Abstract*—This report documents the design and implementation of classical search-based AI agents for the Pac-Man environment as part of a course assignment and in-class competition. The project addresses two core problems: single-agent pathfinding and adversarial multi-agent planning. For single-agent navigation, an A* search framework is developed with domain-specific heuristics — including BFS-based maze distance estimation — achieving significant reductions in node expansions under strict runtime constraints. For adversarial planning, a Minimax agent with Alpha-Beta pruning is constructed, augmented by a custom evaluation function integrating game-specific features such as food proximity, ghost dynamics, and capsule locations. Empirical reward shaping and parameter tuning are employed to balance solution quality and computational efficiency. Results demonstrate that effective search performance depends critically on domain-aware heuristic design and evaluation strategy, beyond algorithmic structure alone. The system ranked 1st among 150+ participants in the competition and received the Best in Class Award. Source code is available at: https://github.com/waterEand/fit5047-pacman-ai.**

***Index Terms*—Search, A* search, Minimax, Alpha-Beta pruning, Pac-Man**

I. Introduction

PAC-MAN is one of the most classic and widely studied video games in artificial intelligence education. In this maze-based environment, the player controls Pac-Man to collect food dots while avoiding ghosts, with the goal of maximizing score and surviving as long as possible. Despite its visual simplicity, Pac-Man presents a rich decision-making landscape that involves planning, search, and adversarial reasoning [1]. The interface of the Pac-Man environment used in this project is shown in Fig. 1.

This assignment focuses on designing intelligent search-based agents to solve a series of Pac-Man planning tasks. These tasks are categorized into two main parts: **single-agent search** and **adversarial search**. In the single-agent scenarios, Pac-Man operates in a ghost-free maze and must plan paths to either a single food dot, one among multiple food dots, or as many food dots as possible. Each of these problems is modeled as a classical search problem with deterministic transitions and uniform action cost [2], allowing the use of graph search algorithms.

To solve these tasks efficiently, we implemented and analyzed the A* algorithm [3] with various heuristic functions. For instance, in the one-goal setting, performance bottlenecks were identified and eliminated through state representation optimization. For the multi-goal setting, a BFS-based heuristic significantly reduced node expansions by providing accurate maze-aware distance estimates. When collecting all food dots under a runtime constraint, we adopted a greedy planning approach using repeated BFS with early termination to form long-term plans as chains of locally optimal segments.

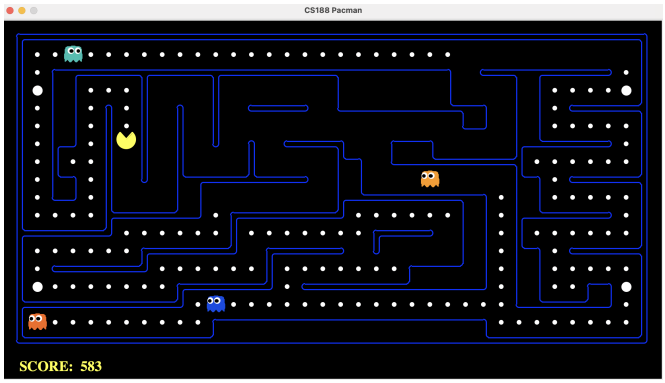

Fig. 1. The interface of the Pac-Man game environment.

In the second part of the project, the problem is extended to an adversarial setting where Pac-Man must act strategically in the presence of ghost agents. This introduces the need for multi-agent search. We implemented a Minimax agent with Alpha-Beta pruning **knuth1975alpha**, and developed a custom evaluation function that considers proximity to food, capsules, ghost dynamics, and game score. Additional design decisions, such as adjusting search depth and incorporating reward shaping [4], were crucial in balancing performance with computational efficiency.

Overall, the project explores how classical AI search algorithms, when carefully adapted with domain-specific heuristics and evaluation strategies, can be applied to solve both deterministic and adversarial planning problems in dynamic environments.

II. Single-Agent Search

The first part of the assignment focuses on the design of search algorithms. The problem is based on Pac-Man game tasked with collecting food dots in a variety of mazes without the presence of ghosts. The problem includes three progressively challenging sub-tasks ranging from single dot navigation to multiple dots seeking. All three tasks must be solved under a strict runtime constraint. While Q1(a) and Q1(b) are required to be solved using the A* search algorithm, Q1(c) allows the use of auxiliary search strategies

1st Place Winner (150+ participants), FIT5047 Pacman AI Competition, Monash University.

[1]Southeast University-Monash University Joint Graduate School, Southeast University, Suzhou 215123, China

to accommodate its larger state space and more complex objective.

A. Implementation of A*

According to the requirement, I need to implement a basic version of the A* search algorithm, a widely known form of best-first search. The algorithm evaluates each state using a function defined as $f(n) = g(n) + h(n)$. $g(n)$ represents the cost from the start state to state n, and $h(n)$ is an estimated cost of the optimal path from n to the goal. Thus $f(n)$ means the estimate of cheapest path passing through n.

In my implementation, a **priority queue** is employed to manage the frontier, with states prioritized based on their $f(n)$ values. Upon expanding a state, the algorithm checks whether a lower-cost path to that state has already been discovered; if so, the current path is discarded. In addition, to prevent unnecessary exploration, the minimal cost at which each state has been reached is recorded in a **visited dictionary**. When the search reaches a goal state, a series of actions will be returned as the answer, and the search ends.

Such implementation serves as the foundational framework for Q1(a) and Q1(b) with heuristics detailed in the following subsections. Although deprecated at last, A* was also attempted in Q1(c). To provide a clearer understanding of the search process, the following pseudocode outlines the structure of A* algorithm:

Algorithm 1 A* Search

```
Initialize priority queue Q, visited map, food position
Push start state into Q with priority g + h
while Q not empty do
    Pop (state, actions, cost) with lowest priority
    if state is goal then
        return actions
    end if
    for all successors of state do
        Compute new_cost
        if better path found then
            Update visited and queue
        end if
    end for
end while
```

B. Q1(a): Collect a Single Dot

In this task, the objective is to guide Pac-Man to collect **a single food dot** in the maze using the A* search algorithm. While the goal definition is relatively straightforward, achieving full marks proved to be unexpectedly time-consuming, as I will elaborate later.

To begin, the file `q1a_problem.py` was modified to define the problem representation. Leveraging the `GameState` class from `pacman.py`, I constructed the problem by directly utilizing `GameState` within each function. At first glance, the task appeared deceptively simple, requiring only basic function calls from `pacman.py`. Encouraged by this, I proceeded to implement the search logic in `q1a_solver.py`. It took me less than ten minutes to produce a functionally correct version of the algorithm 1, thanks to the simplicity and effectiveness of A* search.

However, I was shocked to see that I only got 1 out of 4 marks when I submitted my response to the autograder. Most test cases failed with the error: `Pacman ran out of time in registerInitialState Pacman crashed`. This led me to a round of debugging that was aimed at improving the algorithm's efficiency.

My first optimization attempt involved replacing the visited set with a dictionary to store the cost at which each state was reached:

```
visited = {state: cost}
```

In A* search, it's possible to encounter a state via a more efficient (lower-cost) path after it has already been visited. By using a dictionary, the algorithm can compare and update the cost dynamically, allowing for more flexible path selection. This change made some sense, whereas it only slightly increased performance (less than 0.1 seconds).

In an effort to further reduce runtime, I aimed to decrease the number of node expansions during the search. The heuristic function might be a bottleneck. In order to improve heuristic efficiency, I tried using the real distance between Pac-Man's location and the food dot as the $h(n)$ function,taking maze walls and layout into account. This distance was precomputed during initialization using a Breadth-First Search (BFS) traversal.

The result was remarkably effective: In a big maze layout, this version expanded only 211 nodes, while the standard A* implementation required approximately 550 node expansions for a total path cost of 210. With this version, I received a score of 3 out of 4. Upon discussing with classmates, I learned that full marks for Q1(a) were not difficult to obtain. So I continued to investigate the discrepancy in my implementation.

After a period of debugging and careful review of my code architecture, I was able to determine the root cause of the inefficiency. The issue arose from `q1a_problem.py`, where I had used the `GameState` object to fully represent each search state(node). Such design led to repeated and computationally expensive calls to `generatePacmanSuccessor` and `getPacmanPosition`. These processes were likely to introduce significant overhead, especially in large mazes, and were the primary causes of the timeout errors encountered during grading.

To fix this issue, I changed the state representation by replacing `GameState` with **positional coordinates** of Pac-Man. In the `getSuccessor` function, I computed legal successor positions directly using the wall layout information, thereby eliminating the need to call `generatePacmanSuccessor`. Runtime was significantly decreased by this modification. In big mazes, it saved nearly 0.3 seconds, which was critical for passing the autograder's time constraints. I was finally able to receive full credit for this effort after implementing this change.

C. Q1(b): Collect One Dot among Many

In Q1(b), Pac-Man is placed in a maze containing **multiple food dots**, and the objective is to find and collect any one of them while minimizing the number of node expansions. Unlike Q1(a) where the goal state was fixed and known in advance, this task allows the goal to be dynamically chosen based on the agent's heuristic guidance. Consequently, the algorithm must strike a balance between solution quality and search efficiency, requiring careful heuristic design to reduce the number of expanded nodes.

My implementation adopts the same A* framework described previously, extended to handle multiple potential goal states. The start state is defined as the current `GameState`, which includes Pac-Man's position and the positions of all remaining food dots. The goal condition is satisfied when Pac-Man reaches any position that originally contained a food dot—that is, when his position matches any in the set of food positions. Although using `GameState` as the state representation caused timeout issues in Q1(a), it worked flawlessly in Q1(b), likely due to the reduced depth and branching factor in the search space. As such, it was unnecessary to simplify the state representation to mere positional coordinates in this case.

To efficiently direct the agent toward the most accessible food dot, I designed an improved heuristic based on the minimal maze distance to all food dots. Specifically, drawing inspiration from the BFS-based optimizations explored during Q1(a), I employed a **multi-source BFS traversal** to precompute the shortest distances from all food positions to every reachable point in the maze. During the A* search, the heuristic value $h(n)$ for a given state is computed as the BFS distance from the current position to the nearest accessible food dot. This heuristic is both admissible and consistent, as it never overestimates the true path cost and automatically ignores unreachable food dots.

The following pseudocode illustrates the BFS routine, which can be seamlessly integrated into the heuristic computation within the A* algorithm presented in Algorithm 1:

Algorithm 2 Precompute BFS Distances

```
Get wall grid, initialize empty map D, queue Q
for all f in food positions do
    D[f] ← 0,   push f into Q
end for
while Q not empty do
    Pop (x, y) from Q
    for all legal neighbor (nx, ny) of (x, y) do
        if (nx, ny) ∉ D then
            D[(nx, ny)] ← D[(x, y)] + 1
            Push (nx, ny) into Q
        end if
    end for
end while
return D
```

To explore potential improvements in search efficiency, I experimented with a weighted variant of the A* algorithm, where the cost function is modified to $f(n) = w \cdot g(n) + h(n)$ with a tunable weight parameter w. Setting $w = 1$ recovers the standard A* formulation, while using $w < 1$ increases the influence of the heuristic, effectively making the search more greedy. In theory, decreasing w can reduce the number of node expansions at the cost of possibly suboptimal paths.

However, empirical results revealed a more nuanced picture. As shown in Table I, the BFS-based heuristic consistently and substantially reduced the number of node expansions compared to simpler heuristics such as Manhattan distance. Notably, once the BFS-based heuristic was applied, varying the weight parameter w had virtually no impact on search efficiency. This suggests that the heuristic provides cost-to-go estimates that are sufficiently accurate to dominate the evaluation function $f(n)$, thereby stabilizing the node expansion order.

More remarkably, I observed that when using the BFS-based heuristic with any $w < 1$, the number of expanded nodes exactly matched the solution cost plus one. This implies that the search process expands only the optimal path and the goal node, with no unnecessary exploration. Such behavior strongly indicates that the heuristic is not only *admissible*—never overestimating the true cost to the goal—but also *consistent*, ensuring that the estimated cost is always less than or equal to the cost of reaching a successor plus the estimated cost from that successor. In this setting, the heuristic approaches a **near-perfect** estimate of the true cost-to-go, effectively transforming A* into an optimal greedy search. These findings highlight that the informativeness and theoretical soundness of the heuristic function play a far more critical role in optimizing search performance than merely tuning the weight parameter.

TABLE I
NODE EXPANSIONS ACROSS MAZES AND HEURISTIC SETTINGS

Maze	**Algorithm**	**Weight** w	Nodes Expanded	**Cost**
openCorners	A* (standard)	1.0	212	23
		0.0	212	
	Our A*	1.0	156	
		0.5	24	
		0.0	24	
mediumCorners	A* (standard)	1.0	70	18
		0.0	70	
	Our A*	1.0	21	
		0.5	19	
		0.0	19	
bigCorners	A* (standard)	1.0	293	30
		0.0	293	
	Our A*	1.0	38	
		0.5	31	
		0.0	31	

D. Q1(c): Collect As Many As Possible

The objective of Q1(c) is to collect as many food dots as possible while balancing solution quality with computational efficiency. Unlike Q1(a) and Q1(b), which target a fixed or single optimal food dot, Q1(c) requires dynamically selecting a subset of reachable food dots and generating a complete

action plan within a strict time constraint of 10 seconds. Thanks to the algorithmic groundwork developed in Q1(a) and Q1(b), this problem becomes more approachable despite its increased complexity. Formally, the task can be viewed as a grid-based variation of the Traveling Salesman Problem (TSP), which is NP-hard. Consequently, the focus shifts from exact optimization to designing a fast and effective approximation strategy.

My initial attempt used the standard A* algorithm to plan a full path, but it quickly proved computationally expensive, taking over 10 seconds even on relatively small mazes. I then experimented with Depth-First Search (DFS), which was much faster, but performed poorly: Pac-Man frequently revisited the same locations unnecessarily, resulting in inefficient paths and redundant node expansions.

Ultimately, I implemented a **Greedy Breadth-first Search** strategy that repeatedly selects and navigates to the nearest food dot until either all food is collected or the time budget is exceeded. Similar to the approach in Q1(b), I employed a BFS subroutine to find the shortest path to the closest dot, execute that path, and then remove the dot from the remaining set. This process is repeated iteratively, effectively building a complete action plan by concatenating multiple short, locally optimal paths. The BFS is implemented from scratch without using any generic search agents or heuristics, and includes early termination upon locating the first reachable food dot—thereby minimizing unnecessary expansions and runtime. The overall strategy is summarized in the following pseudocode 3, which outlines the Greedy BFS solver used to sequentially navigate to the nearest food dot.

Algorithm 3 Greedy BFS Solver

Get start state $(pos, food\_set)$
Initialize empty $actions$
while $food\_set$ not empty **do**
    if time exceeded **then return** $actions$
    end if
    Find shortest path and target dot from pos using BFS
    if no path or target **then return** $actions$
    end if
    Append $path$ to $actions$
    $pos \leftarrow target$, remove $target$ from $food\_set$
end while
return $actions$

Thanks to this design, the algorithm remains simple and interpretable while achieving strong performance. In all test cases, including large mazes, Pac-Man successfully collected all food dots within the time limit, completing the task in under one second.

III. Adversarial Search

A. Q2: Adversarial Search with Alpha-Beta Pruning

In the final task, Pac-Man must navigate mazes populated with ghost adversaries, transforming the problem into a classical multi-agent, adversarial search scenario. Unlike the single-agent settings in Q1, this task requires reasoning over the joint decision space of Pac-Man and multiple ghost agents, whose behaviors are governed by the game environment. To address this, I implemented a Minimax agent with **Alpha-Beta pruning**, as specified, to optimize Pac-Man's actions in the presence of adversaries.

As a starting point, I implemented a basic depth-limited Alpha-Beta Search. In this setting, Pac-Man is modeled as the maximizing agent (index 0), and each ghost is treated as a minimizing agent (indices ≥ 1). The search tree is expanded recursively in a round-robin fashion across all agents. When the algorithm reaches a terminal state (win/loss) or the predefined depth limit (default: 3), the leaf node is evaluated using the default evaluation function provided by the environment. The core implementation is illustrated in Algorithm 4. Using only this baseline version, the agent only achieved a score of 1 out of 35.

Algorithm 4 Alpha-Beta Pruning

function AlphaBeta($state$, $depth$, $agent$, $alpha$, $beta$)
    if terminal or $depth = 0$ **then**
        return evaluation of $state$
    end if
    Get legal actions (excluding `STOP`)
    Set best to $-\infty$ if Pacman, $+\infty$ if Ghost
    for all actions **do**
        Recursively evaluate successor
        if Pacman: value $>$ best **then**
            Update best and α; break if $\alpha \geq \beta$
        else
            Update best and β; break if $\beta \leq \alpha$
        end if
    end for
    return best
end function

Upon closely observing the visualizations of the Pac-Man game during multiple runs, I noticed that Pac-Man often became stuck in an oscillatory loop between adjacent positions. Specifically, when Pac-Man moved to position 1, it would prefer to go to position 2; however, upon reaching position 2, it would then choose to return to position 1, resulting in an endless back-and-forth cycle. After reviewing my code, I hypothesized that the evaluation scores of these neighboring states were nearly identical, making it likely that some meaningful actions were pruned during the Alpha-Beta search.

To mitigate this issue, I first introduced a random shuffling of the available actions before the `for` loop in the `Alpha-Beta` function. This reduced the frequency of oscillations to some extent, as the search explored different branches, but also introduced new problems: Pac-Man's paths became more erratic, occasionally visiting unnecessary positions due to randomness.

To improve stability, I designed a custom `rankActions` function for Pac-Man, incorporating handcrafted heuristics based on distances to food, capsules, ghosts, and other environmental features. This guided the agent toward more meaningful moves and further reduced oscillations. Although

this modification improved the agent's behavior qualitatively, oscillations still occasionally occurred, and the overall performance on the autograder remained limited, yielding only 2 out of 35 marks.

To find a more principled and effective solution to the oscillation problem and overall performance limitations, I sought additional insights from both ChatGPT-4o and course TAs. Through these discussions, I gained a deeper understanding of the Minimax algorithm's structure. As a recursive decision-making framework, Minimax is largely insensitive to the order in which actions are evaluated. Instead, its effectiveness is fundamentally tied to the quality of the evaluation function used at the leaf nodes.

Since the default evaluation function could not be directly modified, I chose to build a custom evaluation function that builds upon the default by incorporating additional reward and penalty terms. This new evaluation function was carefully designed to reflect key game dynamics, including:

- Proximity to remaining food dots, measured using actual maze distances(BFS);
- Proximity to power capsules, which allow Pac-Man to eat ghosts;
- Ghost behavior, including scared timers and potential collision risks;
- The current game score, serving as a baseline indicator of progress.

To address the timeout issues encountered on the online autograder, I reduced the Alpha-Beta search depth from 3 to 2. While this reduced computation time, it also limited the agent's foresight—causing Pac-Man to occasionally overlook nearby food that could have been consumed in subsequent steps. This behavior stems from the fact that the default evaluation function does not anticipate short-term gains unless they are realized within the limited search horizon. To compensate, I introduced a local adjustment within the Alpha-Beta routine: if a successor state's score exceeds that of the current state, I added an immediate bonus to the successor's evaluation value. This encourages Pac-Man to prioritize high-reward moves such as eating nearby food, capsules, or scared ghosts more aggressively.

The remainder of the development process involved extensive empirical tuning. By repeatedly observing Pac-Man's behavior within the maze visualizations and monitoring the local evaluation scores, I adjusted the relative weights of various reward and penalty terms to strike a balance between safety, efficiency, and goal-oriented decision making. Ultimately, the final agent achieved a score of 32.6 out of 35 on the autograder, demonstrating both fruitful planning and robust decision-making under adversarial conditions.

IV. Conclusion

This report presented a series of search-based solutions for both single-agent and adversarial variants of the Pac-Man game. Through the implementation and analysis of A* search with task-specific heuristics, as well as Minimax with Alpha-Beta pruning, the project highlighted the critical role of state representation, heuristic design, and evaluation functions in practical AI planning.

For single-agent tasks, informed heuristics—particularly those based on maze-aware BFS—led to significant reductions in node expansions and improved runtime without compromising solution quality. In adversarial settings, carefully crafted evaluation functions proved essential for overcoming search limitations such as oscillatory behavior and shallow depth constraints.

One key observation is that optimizing search behavior is not solely about algorithmic structure, but about integrating domain knowledge into heuristics and evaluation. While A* guarantees optimality under admissible heuristics, the effectiveness in real-time settings hinges more on practical constraints and tailored design. Similarly, in adversarial environments, rule-based rewards and penalties can help bridge the gap between limited lookahead and strategic foresight.

Overall, the results reinforce a core principle of AI search: strong performance emerges not just from general algorithms, but from their adaptation to the structure and dynamics of the domain at hand.

References

[1] UC Berkeley CS188, *The pac-man projects*, Accessed: 2024-04-01, 2024. [Online]. Available: https://inst.eecs.berkeley.edu/~cs188/fa24/projects/.

[2] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. Pearson, 2021.

[3] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[4] A. Y. Ng, D. Harada, and S. J. Russell, "Policy invariance under reward transformations: Theory and application to reward shaping," in *Proceedings of the International Conference on Machine Learning*, 1999, pp. 278–287.