UC Berkeley Pac-Man Projects

Introduction:

The UC Berkeley Pac-Man Project aims to build an AI for Pac-Man with a variety of goals. The project is divided into two main parts, each presenting unique challenges and requiring different strategies.

Part 1 focuses on environments with no ghosts, where Pacman's objective is solely to collect dots scattered throughout the maze. This part is further subdivided into three tasks:

a) Guiding Pacman to a single dot using the A* algorithm with the Manhattan heuristic.
b) Modifying the algorithm to target the closest dot among multiple dots
c) Designing an approach that allows Pacman to efficiently collect all the dots in the maze.

Part 2 introduces a more complex scenario by incorporating ghosts that Pacman must avoid while maximizing his score. Here, the alpha-beta pruning algorithm is implemented to enable Pacman to make optimal decisions in the presence of these dynamic adversaries.

This report is designed to show the setbacks, growth and my reasoning for my design choices. The algorithms and heuristic will also be analyzed based on their performance.

This project lasted 3-4 weeks hence I will also share what I wish I changed in my design if I had more time.

General remarks:

Euclid's algorithm

In the context of the Pacman AI, while the Euclidean distance, which calculates the straight-line distance between two points, might seem like an intuitive choice for guiding Pacman, however it proves to be less effective in a grid-based environment. The reason lies in the nature of Pacman's movement, which is restricted to horizontal and vertical directions. Euclidean distance can underestimate the actual path length by ignoring the grid's layout, leading to suboptimal routes.

This is clearly evident in the following figure

√2	1	√2
1	P	1
√2	1	√2

2	1	2
1	P	1
2	1	2

Figure 1. Shows the Euclidean (Left) and Manhattan (Right) distance to travel from P to an adjacent or diagonal square

The true distance, h^*, to travel diagonally is the Manhattan distance, whereas the Euclidean distance underestimates it. This difference increases the further away you have to predict moving. This makes it a superior choice for the Pacman AI.

This fact will be further demonstrated with figures and examples in the following sections of the report.

A* with the Manhattan heuristic is optimal if the heuristic is consistent, to prove this show:

for any node n, and its successor n' of n, the following condition holds:

$$h(n) \leq c(n, n') + h(n')$$

Proof:

Note: c(n, n') will always be 1 or infinity (in case of wall being there)

goal coordinate = (x2, y2)

n = (x1, y1)

$h(n)=|x2-x1|+|y2-y1|$

n' = (x1±1, y1)

$h(n')=|x2-(x1\pm1)|+|y2-y1|$

This means that |x2 – x1| increases or decreases by 1, and |y2 – y1| stays the same.

Therefore, h(n) – h(n') ≤ ±1

⇨ h(n) ≤ h(n') ±1 ≤ h(n') +1 ≤ h(n') + c(n, n').

The proof is identical in the case of moving North or South.

Background into "Inconsistent heuristics in theory and practice" 2011

One of the resources I wish I had explored earlier in my project is the 2011 paper "Inconsistent Heuristics in Theory and Practice" by Nathan Sturtevant and Ariel Felner. If I had read this earlier I would have utilised their insights in Part 1 of my project.

I will go into specific details later, however for now I will introduce their conclusions:

- In reference to path finding heuristics (like in Pac-Man) they said "In general, for this application the best heuristic depends on properties of the domain". This encourages the reader to stop chasing a purely consistent heuristic and aim for considering the particulars of this map, such as number of ghosts in each section.
- Normally most h-values are alike, however with this springs more "diversity of heuristic values into a search which can lead to a reduction in the number of node expansions."
- This eventually results in "a very large reduction (more than an order of magnitude for many cases) in the number of generated nodes (and CPU time)"

Although it's too late to integrate these ideas fully into my current implementation, I will include examples throughout this report that demonstrate how a heuristic could be designed with these principles in mind.

Part 1.

General remarks

Improvements of state and it's time complexity.

Open sets are held in min-heap priority queues because of their fast retrieval and appending of the minimum element into the PQ.

In astar_initialize I iterate over the whole grid and create my graph and find the food coordinate(s). This simplifies the GameState greatly, even though it takes O(width*height), this complexity is dwarfed to the expontential complexity of A*. The nodes are stored in a dictionary for O(1) retrieval.

In the future this can be further improved by generating the graph as Pac-Man naturally explores the map.

The state I use holds:

- Visited coordinates array: To make sure Pac-Man doesn't move on the same coordinate again.
- Directions array: An array of directions Pac-Man has travelled
- Rolling weight: The distance Pac-Man has travelled
- Food coordinate(s): Coordinate of food. Is a an array if multiple food
- `__eq__` method: Two states are equal if the current coordinate is equal in both. This is important because it allows for the minheap priority queue of properly use the `update` method.

In hindsight, I regret creating separate arrays for directions travelled and rolling weight as they can be derived directly from the visited coordinates in O(V) and O(1) time respectively. This does slow the algorithm, but the big-O complexity would stay the same.

Space complexity:

The space complexity for all of Part 1 the space complexity is the same, which is O(closed) = $O(4^d)$.

For Part 1c) the average space complexity is less than $O(4^d)$ since the closed set is reset, however the worst case complexity is still $O(4^d)$.

Part 1a.

Approach – Pseudocode and clear explanation

The generic A* algorithm is implemented with the state and open set given above.

Also as shown before, the Manhattan heuristic is consistent in this situation hence A* is optimal.

Time complexity:

Branching factor = 4.

Each iteration of `astar_loop_body` the time complexity is

O(open.pop) + O(get successors) + O(iterate over new successors * (|closed| + open.update))

= O(1) + O(4*visited) + O(4 *(4^d + log |open|))

= O(visited) + O(4^d + log|4^{d+1}|) (|open| < 4^{d+1} because for each iteration)

= O(4^d) (|visited| <= d)

To find the overall complexity, let there be depth = n so the big-O complexity of A* is

$$\sum_{i=0}^{n} 4^i < O(4^n)$$

Part 1b.

Approach

Commented [RB1]: Give pseudocode for a bit of it

Everything is the same as part a) except now I have an array of food coordinates in my state and my heuristic is calculated differently.

$$Let\ goals\ be\ s_1, s_2, \dots, s_n$$

$h(s) = min_{1 \le i \le n}\{h_{manhattan\_i}(s)\}$ where $h_{manhattan_i}(s)$ is Manhattan heuristic from s to goal s_i

The intuition behind this is that while Pac-Man traverses there are vectors pointing him towards goals with magnitude of the Manhattan distance, and he tries to follow the one which is the smallest, let this be goal s_k. This is consistent, I will examine the different cases below.

Case 1: There are no obstacles between Pac-Man and s_k. Then it will travel to it and s_k will be the true closest goal since h_{man} is consistent.

Case 2: There are obstacles between Pac-Man and s_k. Then it will take a detour, probably around some walls. This will give opportunity to other vectors of Pac-Man to become smaller if while taking this detour another goal comes closer than s_k then it will begin to move towards it. This flexibility makes this heuristic useful, as can be showed in the following example.

	A	B	C	D	E	F
1					G	
2		P	->	->	v	
3					v	
4		G	<-	<-	<-	

Figure 2. Shows the original (temporary) path implicitly decided by h when B4 was the minimum.

From the current position Pac-Man would follow a path to move towards B4, as drawn. However, when it walks to D2 it will be attracted to the real closest goal, E1.

Case 3: However, what if the real closest goal isn't on the path being walked? Then this heuristic will make unnecessary node expansions. This is shown in the example below.

	Z	A	B	C	D	E	F
1							
2							
3					^		
4	G				P		G

Figure 3. Shows the initial node expansions with h

Node expansions will first take P to D3 to get to Goal F4 even though Z4 is closer. Despite this it should be noted h is still consistent.

Now that I know about Sturtevant and Felner's paper, in the future I would have made my heuristic also add the average number of walls if Pac-Man took a straight path to the estimated closest goal.

$$I = number\ of\ intercepts\ with\ wall\ from\ P\ to\ estimated\ closest\ goal$$

$$If\ I = 0\ then\ hasWalls = 0, else\ hasWalls = 1$$

$h = min_{1 \le i \le n}\{h_{manhattan}(s_i)\} + a_i I + hasWalls * 2$ where a_i is some constant

The motivation behind hasWalls is that if there is a single wall it will take between Pac-Man and the goal then it will take 2 more steps than without it.

This is in accordance with Felner's paper which wants heuristics which considers the domain thus creating more diversity in the h values. Furthermore, this would reduce tiebreakers and will also make solving Figure 3 more efficient.

Different Algorithm & Comparison

Euclid's algorithm figures:

To illustrate the effectiveness of different heuristics, the following figures compare the Euclidean and Manhattan heuristics in guiding Pacman through the maze. Although both are optimal with the same scores the number of node expansions are different:

Map	Euclid (node expansions)	Manhattan (node expansions)
q1b_bigCorners.lay	234	59
q1b_mediumCorners.lay	58	21
q1b_closed.lay	55	55
q1b_openCorners.lay	191	156
q1b_tinyCorners.lay	8	6
q1b_trickyCorners.lay	34	19

Figure 4. Gives number of node expansions for the Euclid and Manhattan heuristic.

This further proves our theoretical conclusion that Manhattan distance is more accurate than the Euclidean for Pac-Man's movements.

Time Complexity

F, V = food array, visited array.

Branching factor = 4.

Each iteration of `astar_loop_body` the time complexity is

O(open.pop) + O(goal state) + O(get successors) + O(iterate over new successors and see which should be in open)

= O(1) + O(F) + O(4*(V + F)) + O(4*(closed + F + log|open|))

= O(4*(V+F)) + O(4*(4^d + F + log|open|)) (closed < 4^d)

= O(4^d + log|open| + F) (V <= depth)

= O(4^d) (open < 4^d & F < 4^d for a large enough depth)

So overall the time complexity is the same as in Part 1a. O(4^d)

Part 1c.

Approach

This approach is largely based off part b since Pac-Man moves towards the closest estimated food based of the same heuristic. Then when it is reached then the closed & open set is reset, the state's foods removes the reached food and records the weight to reach this point. Then A* runs again. This continues until all the foods are reached, then it is checked until which food should the algorithm stop by? Because sometimes ending early can be more favourable.

To avoid an infinite loop for when foods aren't reachable, in astar_intitialise perform a BFS/DFS to see which foods are reachable, these are the only foods considered.

Limitations

Consider initially you are in the scenario which resembles Figure 3, then the mistakes of the heuristic from Part 1b. is ***inherited***. So the heuristic based on Felner's paper in part 1b) can be used for h-values which are more tailored to the specific map.

Different Algorithm & Comparison

Euclid's heuristic

1. Map	Euclid (node expansions)	Manhattan (node expansions)
q1b_bigSearch.lay	712	420
q1b_greedySearch.lay	24	23
q1b_closed.lay	123	78

Figure 5. Gives number of node expansions for the Euclid and Manhattan heuristic.

This matches the theoretical conclusion drawn. That the Euclidean heuristic is not applicable to Pac-Man

High commitment approach:

A glaring limitation of this algorithm is also the fact that it is greedy. Although the next food you reach will be the closest, it may be more beneficial to go for the further away one for now. This is clear in the Figure 6. Scenario it would first go top-right and then reset go to the top-left then bottom-left and bottom-right.

This is clearly not optimal, however the impact of this greed can be dulled if Pac-Man commits to a direction, for example, going top-right then bottom-right first. In the future, this could be implemented by dividing the food coordinates array into quadrants/halves of the map, and then completing the map 1 quadrant at a time, but Pac-Man leave that quadrant if the h-value becomes too large. This is also how humans intuitively solve Pac-Man.

Figure 6. A scenario of Pac-Man

Time Complexity

The time complexity can be derived similarly like in Part 1a & b. The big-O time complexity is $O(4^d)$ where d is the total depth.

More specifically, let the order be of food found be $s_1, s_2, ..., s_n$ and the d_i = depth to go from s_{i-1} to s_i.

So $d = d_0 + d_1 + ... + d_n$ hence the time complexity is $O(4^{d0 + d1 + ... + dn})$

Part 2.

Approach

Pseudocode

```
def max_score(state, depth, alpha, beta):
    … This is the same as a normal max_score

def min_score(state, depth, agent_id, alpha, beta):
'''
Since each ghost is indexed we keep repeating
'''
    is gameOver: return state.getScore()
    next_agent = (agent_id + 1) % state.getNumAgents()
    best_score, score = np.inf, np.inf
    scores = []
    Iterate over agent_id's possible actions:
            If this next_agent is pacman:
                    If max depth is reached:
                            score = evaluation function
                    else: [score, _] = max_score(gameState after action, depth,
                                    next_agent, alpha, beta)
            else: score min_score(gameState after action, depth, next_agent, alpha,beta)
            scores.append(score)
            if candidate_value < best_score:
                    best_score = candidate_value,
                    beta = max(beta, best_score)
            if best_score <= alpha: break
            return average(scores) # return best_score if you don't want expectimax.
```

Give motivation and how Pacman moves around.

The algorithm uses alpha-beta pruning to efficiently search through game states, with Pacman acting as the maximizer and ghosts as minimizers. In the `min_score` function, a for loop iterates over each ghost (min-agent), updating the score for each possible action, and recursively calling either `max_score` (for Pacman's turn) or `min_score` (for the next ghost) depending on the next agent.

The pruning occurs when the current score cannot improve upon previously evaluated states, and the average of all scores, this is expectimax.

I used expectimax in this scenario because it better models the uncertainty of ghost movements, which may not always act optimally or predictably. By averaging the possible outcomes, expectimax accounts for the probabilistic nature of ghost's behaviour, leading to more robust decision-making for Pacman compared to purely minimizing the worst-case scenario.

My evaluation function:

Returns current game score if game is over

Else: $h(s) = current\_score + h_{ghost} + h_{food}$

where h_food = 0 if closest_food_dist = 0, else: 5 * 1/closest_food_dist

where h_ghost = 0 if closest_ghost_dist = 0, else: ghost_chase * 1/closest_ghost_dist

where ghost_chase = 10 if closest ghost is scared, else: -10.

Limitation:

Although the current heuristic isn't consistent it does however slightly follow the

The problems with the minimum heuristic from Part 1b & Part 1c still hold in this scenario, all the problems are inherited. Hence the possible solutions listed prior are also relevant to this, such as: counting number of wall intercepts if a straight line went from Pac-Man to the goal; creating a high commitment Pac-Man.

Another limitation is not considering food capsules it would have created more diversity of thought inside my heuristic since more things would be considered. However, I didn't have enough time to fine tune its' coefficient, however the distance to the closest capsule is present in my code (but left unused).

The final, inevitable limitation was the depth limit. With n ghosts ($0 < n < 5$) there would be $O((n+1)*4^d) < O(n4^d)$ possible moves to consider where d is the depth. Hence I decided to limit depth to 3, or 2 when there a 4 ghosts. I will later explain how this turned out to be in my favour.

Personifying Pacman

If one of these coefficients are positive, then it is desirable to Pac-Man and undesirable if negative.

Give justification for each

- h_{food}: Encourages Pacman to prioritize food collection. When the distance to the nearest food is smaller, the heuristic value increases, guiding Pacman to the closest dot efficiently.
- h_{ghost}: Balances the risk of encountering ghosts. If ghosts are scared, the heuristic rewards Pacman for moving toward them. If not, it penalizes proximity to dangerous ghosts, promoting safer paths.

		g = h_ghost coefficient		
	|coefficient|	1	5	10
f = h_food coefficient	1	731.9	641.1	694.85
	5	449.8	772	750.1
	10	430.5	620.9	604

Figure 7. 40 games are tested on q2_capsuleClassic layout then averaged. The (h_ghost's coefficient) = g then ghost_chase = g if ghost is scared, else: -g. Similarly for h_food's coefficient.

If a non-scared ghost is on a food coordinate then Pac-Man should not try to travel there, this would be indicated with h_foods < h_ghost, and this happens when g > f. This is proven in Figure 7 where when the average score for when g > f is greater than when f > g.

So I chose f = 7 and g = 10.

Common habits & what I learnt from them:

Pac-Man would move back and forth in a corner of the map after all nearby food were eaten and the ghosts were far away. It only left the corner when a ghost tried to come close to it. This wasted a lot of time and left the algorithm too much on the chance of a ghost coming close Pac-Man. This happened Pac-Man was too scared of the ghosts.

To fix this I tried:

- Hungry Pac-Man: Making the heuristic's reward for getting food be higher, but since the food coefficient is always bounded below the ghost coefficient it would still fear the ghosts (argument for this given below Figure 7.2)

- Adventurous Pac-Man: I tried making it so if the ghost is faraway it would try to chase it, but runaway if it was close. However, this led to it sometimes trying to micro-manage how far away from ghosts it should be rather than getting the bullets.

- Removing outliers: I realized that when the game was lost it would skew the expecti-max's average so I decided to remove the outliers by removing values which were < 0.1 and > 0.9 of the quantile of the data set. However, the scores across all maps dropped drastically. This was because in some cases it would be less weary of ghosts and losing thus would get cornered.

- Finally, the best solution to be less scared of ghosts was to make him more naïve. This was done by reducing the depth of the algorithm to 3. Also, since removing outliers partially worked I multiplied losing score by 0.75.

Expecti-max vs without expecti-max

2. Map	Expecti-max (Average score)	Normal mini-max (Average score)
q2_minimaxClassic.lay	452.02	411.46
q2_capsuleClassic.lay	753.46	700.64
q2_openClassic.lay	1314.8	1128.0

Figure 8. 10 games in each of these maps. Comparing average scores from expecti-max and mini-max, both with alpha-beta pruning.

This aligns with the theoretical explanation for why expecti-max is better than mini-max in the scenario of ghosts with random movements.

Time complexity:

Note, depth, d, is counted as the number of turns Pac-Man has had.

Let G = Complexity to generate game state

Let N = number of ghosts

Let E = Space/Time complexity of Evaluation function = O(capsule + ghost + food)

Let S = Space/Time complexity of number of scores collected. This is bounded by the branching factor so let S = b.

The recurrence relations are the following:

Max(d) = b*G*Min(d)

Min(d) = N*E*b*G*Max(d-1) + b

And Max(0) = |scores|

So, to begin to simplify...

So Max(d) = b*G*(N*E*b*G*(Max(d-1)) + b)= $N*G^2*b^2*Max(d-1) + b^2G$.

In our case, for d = 2 (where there are 4 ghosts)...

Max(2) = $N*E*G^2*b^2*Max(1) + b^2G$

= $N*E*G^2*b^2*(N*G^2*b^2*Max(0) + b^2G) + b^2G$.

< $O(E*N^2*G^4*b^5) = O(E*N^2*G^4*4^5)$

Space Complexity:

In each min_score: O(b*G)

In each max_score: O(N*b*E*G)

Since there will be d calls to each the space complexity is O(d*N*b*E*G) = O(N*E*G) since b and d are constant.
