Path-Finding in Pac-Man AI

Muhammad Ahsan Ninh Tran

January 11, 2022

Abstract

AI to play as Pac-Man, being one of the revolutionary games which developed the modern era of Video Games, have been created ever since the game released. However, each of these AI's have varied in their performance, regarding their end score. As such, we wanted to explore this performance furthermore, specifically tied in with the fundamental search algorithms. Particularly, in this project, we will explore a fundamental search algorithms such as DFS, BFS, UCS, A* and a machine learning approach called reinforcement learning then apply the various methods to the classical Pac-Man game. The goal of the Pac-Man agent will be to find paths through his maze world, both to stay alive and collect food as efficiently as possible while utilizing different path-finding algorithms.

1 Introduction

Path-finding in computer games may be conceptually easy, but for many game domains it is difficult to do well. Real-time constraints limit the resources—both time and space—that can be used for path-finding. One solution is to reduce the details of the grid, resulting in a smaller search space. Regardless, the demands for realism in games will always result in more detailed domain terrains, resulting in a finer grid and a larger search space. Watching how algorithms perform in video games also provides insight into how artificial intelligence behave. This is interesting because it provides better understanding of how an algorithm approaches problems and makes decisions. There is no doubt that real-world AI problems are challenging and Pac-Man provides a formidable problem environment that demands creative solutions.

The problem for this assignment is to find efficient pathfinding techniques to help aid in achieving the highest score possible in Pac-Man. Grid based maps have been shown as an effective technique for map creation in video game development due to their memory efficiency and ease of creation, so finding an efficient way of traversing grid-based map systems allows for the best chance of achieving the highest score in Pac-Man as the whole game occurs on a grid. We can translate all character positions to points on a grid and determine different heuristics for the player to determine efficient traversal patterns. Grid based map traversal can happen under a variety of differ circumstances, so to begin we must first determine the goal of the traversal, the agents present in the system(as many as the traversal agent can know about), and whether we have access to any heuristics that might aid in the grid search. Pac-Man presents

a challenge as the system is dynamically changing with the movement of the ghosts and random placements of the fruits on the map, so if any heuristics are involved they would have to be changed to reflect the new positions of the ghosts at every frame, so finding a fast traversal algorithm is crucial due to the massive calculation overhead.

2 Related Work and Literature Review

2.1 A Look at Existing Search Algorithms

2.1.1 Uninformed and Informed Search

Path-finding is a critical problem in a wide range of applications, including network traffic, robot planning, military simulations, and computer games. A grid is typically superimposed over a region, and a graph search is used to find the optimal path. The most common scenario is to use a tile grid and search with A*. The tradeoffs for various grid representations and grid search algorithms are discussed in this paper. For the purpose of this study we will only cover the 4-way tiles grid representation. There have been many attempts to develop a Pac-Man grid-based path-finding the utilizing of Breadth First Search, Depth First Search, Uniformed Cost Search, and A* as [6] implement these search algorithms on the Pac-Man map.

Search Algorithm Used	Small Maze	Medium Maze	Big Maze	Description
DFS	10	130	210	Cost
	15	146	390	Node Expanded
	500	380	300	Average Score
	500	380	300	Score
BFS	8	68	210	Cost
	15	269	620	Node Expanded
	502	442	300	Average Score
	502	442	300	Score
UCS	8	68	210	Cost
	15	269	620	Node Expanded
	502	442	300	Average Score
	502	442	300	Score
A*	8	68	210	Cost
	14	211	549	Node Expanded
	502	442	300	Average Score
	502	442	300	Score

Table 1: Search Algorithm Experiment Results on Small, Medium, Big Maze from [6].

From Table 1, we can observe that UCS outperforms the other two uninformed search techniques in terms of time complexity. However, UCS is only effective when the path costs are different; otherwise, it falls back to BFS. BFS has the best time complexity of $O(b^d)$ if all 10 references have the same path costs [6]. However, the spatial complexity is exponential. Although DFS has a linear space complexity, it has the same time complexity as Breadth First Search. DFS is also incomplete and inefficient,

whereas the other two are both complete and efficient (given certain conditions). The informed search strategy: If we apply admissible heuristics and a consistent path cost, A* search is optimal (in case of graph search). However, it has the same exponential space complexity as BFS. With this in mind, our team was unable to locate adequate proofs from published works to support the claim that BFS, DFS, and UCS are capable of competing with other novel algorithms.

With success of A* in multple areas, there are many attempts to improved and optimized the algorithm. In which case, [3] systematically reviews several popular A*-based algorithms and techniques according to the optimization of A*. When looking into larger maps, [1] presents two effective heuristics for estimating distances between locations in large and complex game maps. The former, the dead-end heuristic, eliminates from the search map areas that are provably irrelevant for the current query, whereas the second heuristic uses so-called gateways to improve its estimates. Observations of the actual game maps shows that both heuristics reduce the exploration and time complexity of A* search significantly over a standard octile distance (Euclidean distance).

2.2 Reinforcement Learning

As acknowledged in [12], early solutions to the problem of pathfinding in computer games, such as depth first search, iterative deepening, breadth first search, Dijkstra's algorithm, best first search, A* algorithm, and iterative deepening A*, were soon overwhelmed by the sheer exponential growth in the complexity of the game. Along with these search algorithms we should consider reinforced learning[2]. This takes the results of different maneuvers generated via the search algorithms, and learns what works and what doesn't by applying different weights to different actions and extrapolating the highest earning actions. This data can then be used to prune non-deal decisions and should be heavily considered as it can provide a optimal outcome faster than the bare search algorithms can accomplish. The results of this method are astoundingly positive:

	Single network		**Action networks**	
Trial	Level completion	Wins	Level completion	Wins
1	90.1% (SE 0.2)	46.3%	93.8% (SE 0.2)	65.5%
2	89.2% (SE 0.2)	55.2%	88.7% (SE 0.2)	53.5%
3	86.9% (SE 0.3)	53.5%	86.7% (SE 0.2)	46.3%
4	85.7% (SE 0.3)	50.1%	84.8% (SE 0.2)	40.2%
5	85.6% (SE 0.3)	49.9%	81.7% (SE 0.3)	52.2%
6	82.1% (SE 0.4)	47.8%	77.5% (SE 0.4)	43.8%
Avg.	86.6% (SE 1.2)	50.5%	85.6% (SE 2.3)	50.3%

Table 2: Average percentage of level completion (and standard error) along with percentage of successful games (out of 5.000 total games) during testing on the first two mazes. the bottom row contains results averaged over the various trials. [2]

From Table 2 we can easily observe that the single action network and multiple actions networks appear to perform equally well when tested on the first two mazes.

	Single network		Action networks	
Trial	Level completion	Wins	Level completion	Wins
1	90.5% (SE 0.2)	46.1%	84.9% (SE 0.3)	50.0%
2	89.1% (SE 0.2)	49.0%	80.8% (SE 0.3)	34.7%
3	88.8% (SE 0.2)	47.6%	79.8% (SE 0.4)	42.3%
4	86.3% (SE 0.3)	54.6%	70.5% (SE 0.4)	31.3%
5	83.0% (SE 0.3)	47.2%	68.4% (SE 0.5)	29.1%
6	82.3% (SE 0.3)	47.3%	61.8% (SE 0.5)	28.0%
Avg.	86.7% (SE 1.4)	48.6%	74.4% (SE 3.6)	35.9%

Table 3: Average percentage of level completion (and standard error) along with percentage of successful games (out of 5.000 total games) during testing on the third maze. the bottom row contains results averaged over the various trials. [2]

In this case, the single action network outperforms multiple action networks when tested on the unknown maze. Table 3 shows that in 5 of the 6 runs the performance of the single action networks is better than the performance of the multiple action networks. Data from [2] shows that not all networks are capable of forming a generalized policy.

2.3 Deep Dive Into Grid-Based Search

Each paper provides different methods of traversing grid based systems. When considering which to use to create the best Pac-Man agent two main factors need to be considered. Firstly, the speed at which the traversal occurs. Since Pac-Man is a dynamic environment where there are adversaries, decisions must be made with haste in order to correctly avoid the ghosts. Rectangle Expansion provides the slowest of the three in terms of speed, since its path finding grows out from the original rectangular grid[13]. Plainly implemented this system wouldn't be able to make correct decisions as the environment might've changed by the time it expands its search. This is especially faulty when considering it uses heuristics generated at the beginning which might not be useful after expansion due to the dynamic nature of the game. The graph pruning algorithms falls into the same issue of the previous algorithm where it relies on previous heuristics to create jump points to direct the agent[4], but the analysis happens holistically, so if new heuristic are generated in-between jump point generation, the agent could quickly determine the most optimal pathing to avoid being consumed by a ghost. The final paper provides a robust methodology for calculating the optimal moves at a given moment in the dynamic environment, but there is a lot of overhead involved with each step calculating the tactic data applying it to the nodes and making the best decision in that given moment which slows the program down[9]. As the complexity of the program increases, the algorithm could potentially find itself slowing down which would not be ideal in a winning situation. We also need to consider the space complexity of the algorithms. The game finds itself adding more ghosts to the game as the agent progresses, so the space complexity needs to be considered as the data needed to be processed for decision making grows. The first two algorithms on their own don't have much space complexity overhead, since their heuristic data is generated at the beginning of their execution, and further calculations are based around exploring optimal pathways and pruning irrelevant ones[4][13]. The third paper which is a practical application of Pac-Man does have a lot of space overhead, calculating the reward costs

at every node traversal[9]. They try to fix this with tree search reuse[9], but there is a better solution. When looking at the best possible application for pathfinding whistle playing Pac-Man, the best solution comes from using of Graph Pruning for Pathfinding on Grid Maps. This method provides jump points which are safe points that can be expanded to very quickly. The downside of this approach is that the heuristics are not dynamically generated. However, if we take some of the teaching from the Monte Carlo approach and generate heuristics based on the tactics they generated, this algorithm becomes very efficient. Not only is this a fast algorithm, but once the heuristic data is generated for a given moment we don't need to worry about storing previous heuristic data. This is a largely greedy approach, since the agent makes the best decision at a given point, but when considering a game where the environment is changing in every frame this methodology becomes very promising. Furthermore, [7] proposed the Fast Approximate Max-n MCTS where they performing MCTS on a five player $\max^n$ tree representation of the game with limited tree search depth. They performed a number of experiments using both the MCTS for Pac-Man but interestingly [7] could also be used on the ghost agent in catching Pac-Man to showcase its effectiveness. Another related work that utilize the idea of implementing ghost agent's path-finding called Navigation Mesh (NavMesh) [14], in which the map is represent as polygons and the A* algorithm was implemented. Both [7][14] introduce the novel idea of optimizing the ghost agents traversal though the grid. So for the purpose of this research - score the most points in Pac-Man - we will not put the ghosts' path-finding implementation into consideration. With that in mind, won't be implementing MCTS algorithms for our project.

2.4 Practically Applying Search Algorithms

Moreover, once we consider the implications of the search algorithms we must also consider how it all comes together. For this we can look towards [8]. The paper describes how once we extrapolate data from search algorithms we can practically apply this to Pac-Man. While being built with foundation of path-finding techniques previously discussed, the programmer needs to consider a methodology to direct the actual character. The paper emphasizes the use of conditional logic to make this happen, so the space complexity of these decisions remains O(1) at any given moment since the maneuvers aren't stored. Note that this is not discussing the data needed to make the maneuvering decisions, rather the maneuvering itself. The distinction is necessary since Pac-Man is a dynamic game where these decisions will need to be made on the fly. This is further exemplified in the paper regarding "Ms. Pac-Man Versus Ghost Team CIG 2016 Competition"[10]. The paper shows how it makes the decisions regarding the character and using their path finding data the Ms.Pac-Man character make the best given move at any juncture in the game.

3 Approach

For our approach to test the search algorithms on the Pac-Man game with the motivation to cleverly avoiding the ghosts and eating the food and scared ghosts as much as possible, we will employ rules and rewards table to determine the optimum of the search algorithms as follows:

Event	Reward	Description
Win	+500	Pac-Man has eaten all the food and power-ups
Lose	-500	Pac-Man and a non-scared ghost have collided
Ghost	+200	Pac-Man and a scared ghost have collided
Food	+10	Pac-Man ate a piece of food
Power-up	+100	Pac-Man ate a power-up
Step	-1	Pac-Man performed a move
Reverse	-2	Pac-Man reversed on his path

Table 4: In-game Events and The Corresponding Rewards, As Used In Our Experiments

With that in mind, the reward table will be more significant for reinforcement learning for the penalty system. Furthermore, we will be using developed code for search algorithms such as BFS, DFS identify these as uninformed state-space search and UCS, A* are informed state-space search. Another algorithm we will implement the proposed reinforcement learning algorithm using higher-order action-relative inputs [2]. For these algorithms, we will use implementations from [11] and modify their code to fit our experimentation needs. For uniformed search algorithm, we will have the initial state as the maze cell that Pac-Man starts in; successor function as possible actions that he can take (e.g., move East), and the cell that he wants to be in (i.e. cell with food dot or power-up) is the goal state. We will expand cells until we reach the goal state while treating ghosts like walls or unexpandable regions.

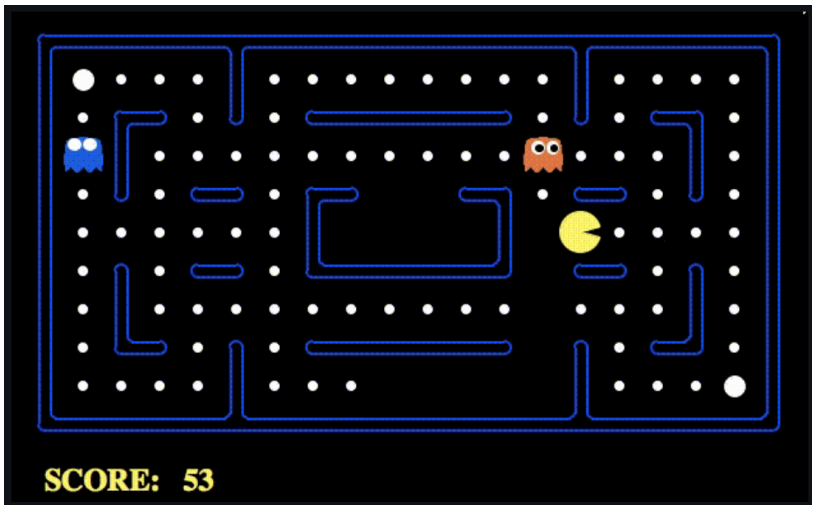


Figure 1: Pac-Man Map Representation

3.1 Breadth-First Search

BFS's frontier is a first-in-first-out (FIFO) queue and expands successors in the order they were added. Consider Figure 1 above, we starting at initial state, BFS explores the path heading south, north and the path heading east concurrently. BFS explores all the neighbouring cells on each path that are 1 step away. Since there is a goal cell at the east, BFS will reach the cell and before the same search algorithm. Otherwise, BFS tries the cells 2 steps away, and still no goal cell, so BFS tries looking 3 steps away,

and so on. By the nth iteration, BFS finds that the path heading south yields food or power-ups, so that is the path that Pac-Man takes. Once found food, the algorithm will also look head into adjacent nodes for goal states. If the goal states are located next to each other, Pac-Man will be program to move in the same direction.

3.2 Depth-First Search

Another search algorithm that DFS uses a last-in-first-out (LIFO) stack to construct the frontier. DFS adds a successor to the frontier and immediately expands it; in other words, it builds a path by exploring a neighbouring cell, then exploring the cell next to that, then the cell next to that, and so on. For instance, referring to Figure 1, it determines Pac-Man can go either east, north or south from the initial state, but chooses to explore the path heading east to the end before considering the path heading south. The path that DFS explores first is indicated in white. At the end of this path, DFS has no new un-explored cells, and has found no food. Therefore DFS moves to the next un-explored state on the frontier: it "backtracks" to the last cross-roads and continues down the green path. Similarly, DFS finds food and reaches a goal state to score the most points as points as possible.

3.3 A-Star Search

As for A*, we define a heuristic (foodHeuristic) that takes into account the positions of the food currently in the maze. By calculating estimated cost using the Manhattan distance from Pac-Man's current position to the food dots, we can guide Pac-Man to eat up all the food in the optimal time while maintaining admissibility and consistency for A* search. The heuristic function can be calculated as follows:

$$foodHeuristic = (cell(Pac - Man) - cell(closestFood)) + estimatedCost$$

Utilizing a new type of heuristic function which utilizes the priority in-game of gathering food as goal nodes. Alongside with the traditional A* algorithm, our heuristic greedily prioritizes cells with the closes food to create a local maximum for game score in the shortest amount of time.

3.4 Uniform Cost Search

On the other hand, we also implement UCS, which will also utilize a priority queue but without the foodHeuristic. In this algorithm from the starting cell we will visit the adjacent cells and will choose the least costly cell then we will choose the next least costly cell from the all un-visited and adjacent cells of the visited cells, in this way we will try to reach the goal cell, even if we reach the goal cell we will continue searching for other possible paths because there will be multiple as expected for the Pac-Man game. We will keep a priority queue which will give the least costliest next cell from all the adjacent cells of visited cells.

3.5 Reinforcement Learning

Additionally, we also implement the reinforcement learning algorithm from [2] and to extract information from the map such as placement of ghosts, food and power-ups

to produce higher-order inputs from the game-state. These inputs are then given to a neural network that is trained using Q-learning. With that, we will develop the following general Q-learning rule:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t))$$

where a state is referred to as s_t and an action as a_t, at a certain time t. The reward emitted at time t after action a_t is represented by the value r_t the constants α and γ respectively refer to the learning rate and discount factor.

The Q-value of a state-action pair is updated using the immediate reward plus the value of the best next state-action pair. The discount factor decides how distant rewards should be valued, when adjusting the Q-values. The learning rate influences how strongly the Q-values are altered after each action [2]. Then we specify the probability of the next state and reward based on the complete history to have Markov property:

$$Pr\{s_{t+1} = s', r_t = r | s_t, a_t, r_{t-1}, \ldots, r_0, s_0, a_0\}$$

Referring to Table 1, the target output is calculated based on the reward, discount factor and the Q-value of the best next state-action pair. If the last action ended the game, we use this equation to compute the target output value:

$$Q^{\text{target}}(s_t, a_t) \leftarrow r_t$$

Otherwise this equation is used to compute the target output:

$$Q^{\text{target}}(s_t, a_t) \leftarrow r_t + \gamma \max_a Q(s_{t+1}, a)$$

Furthermore, we'll also modify the implemented approximate Q-learning code [11] to fit our testing points. In approximate Q-learning, $Q(s, a)$ is obtained from a linear combination of features $f_i(s, a)$ as explained by [5]:

$$Q(s, a; w) = \sum_{i=1}^{n} f_i(s, a) w_i$$

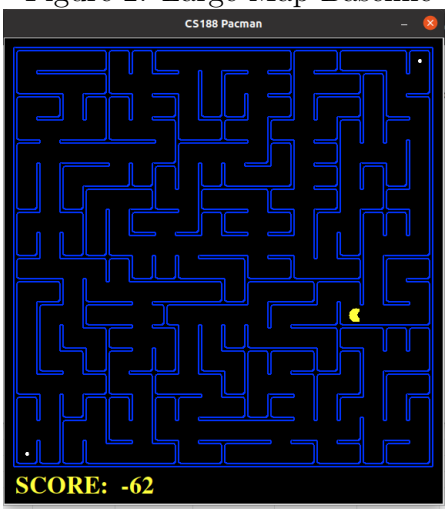
Pac-Man will be able learn the weights for the features extracted from the game states. As described in [5], a feature $f(s, a)$ is defined over state and action pairs, which yields a vector $(f_0(s, a), {}_1(s, a), \cdots, f_i(s, a), \cdots f_n(s, a))$ of feature values, being $f_0(s, a) = 1$ the bias term. The weights are updated according to the following rule:

$$w_i \leftarrow w_i + \alpha \cdot (r + \gamma \max_{a'} Q(s', a') - Q(s, a)) \cdot f_i(s, a)$$

4 Experiment Design and Results

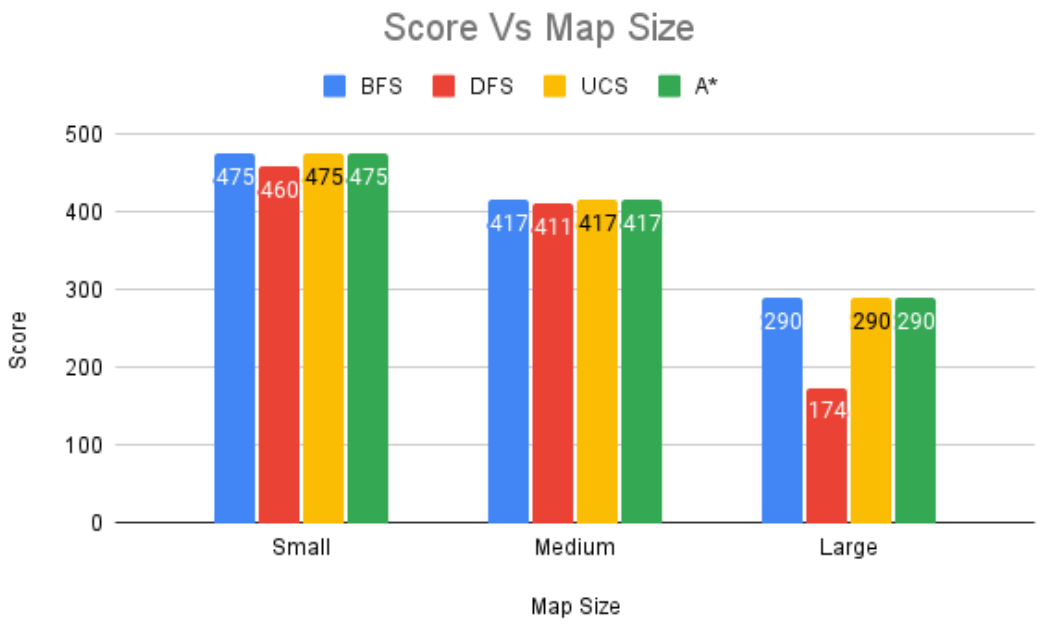
When considering the design of the experiment there needed to be consideration of how to compare algorithm performance in practical Pac-Man settings. In lieu of this, the design is centered around the Pac-Man character as the primary agent. The environment is fully observable, the system is deterministic as actions lead the agent to a determined position. Actions are sequential as the map traversal must happen sequentially. Now, for the environment tests were run in static environments to better understand the efficiency of the algorithms, but once ghosts were brought into the mix the environment became both dynamic and a multi-agent one. The environment is continuas to account for the possibility of changing positions at any moment. The actions for the agent were to move up, down, left, and right whenever the agent encountered a crossroad in the map, making the action system known. The base code for the experimentation was retrieved from Jason Wu[11], and modified to account for our specific use cases. As discussed, four algorithms were deemed to be the most optimal for testing: BFS, DFS, UCS, and A*. As a baseline these algorithms were tested under three map sizes customized with two goals located on opposite ends of the map using the point system described in the previous section.

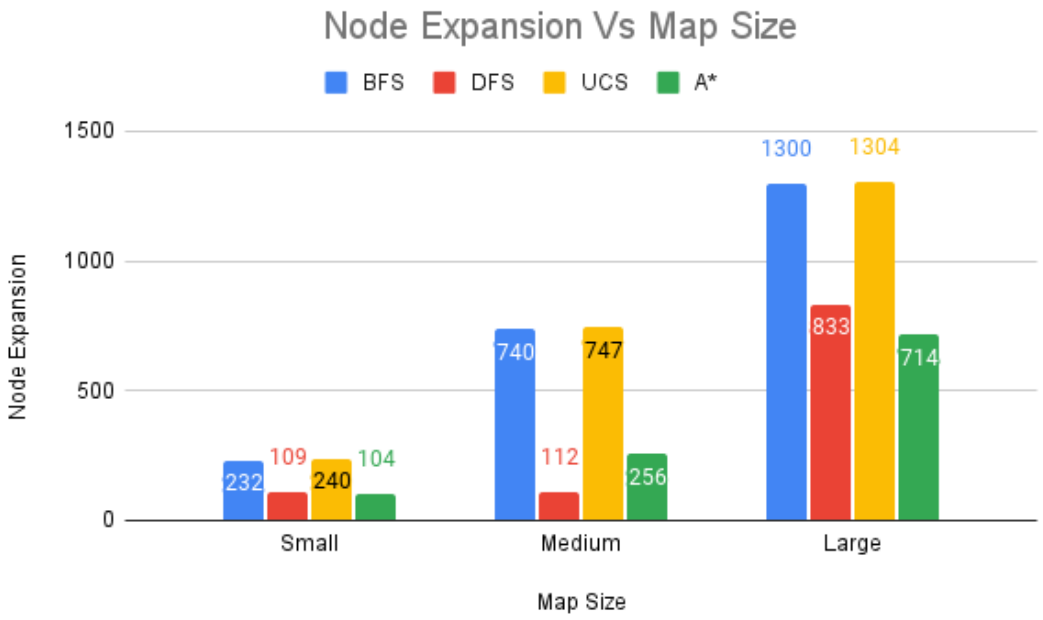
Figure 2: Large Map Baseline



The following tables and graphs depict the search time, score, and node expansion of each algorithm, respectively.

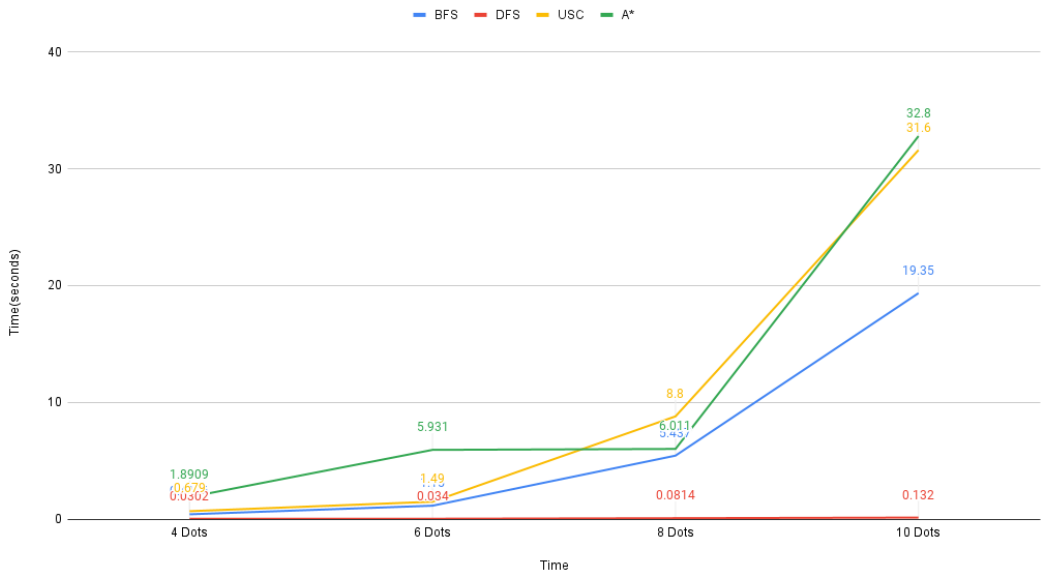
Map Size	BFS Time(sec)	DFS Time(sec)	UCS Time(sec)	A* Time(sec)
Small	.00971	.00462	.0155	.0293
Medium	.0652	.00871	.108	.1786
Large	.196	.119	.321	1.51



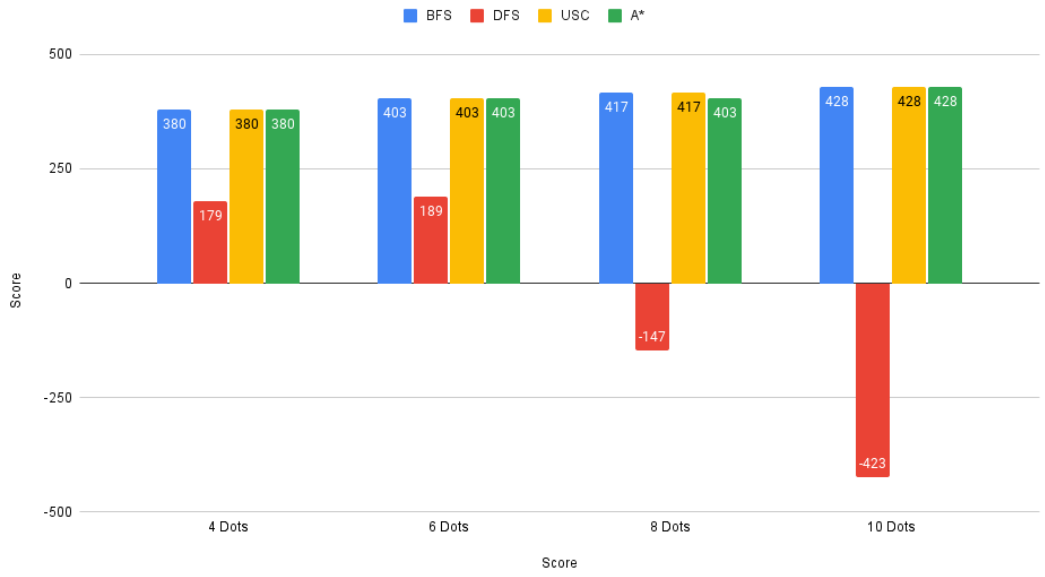


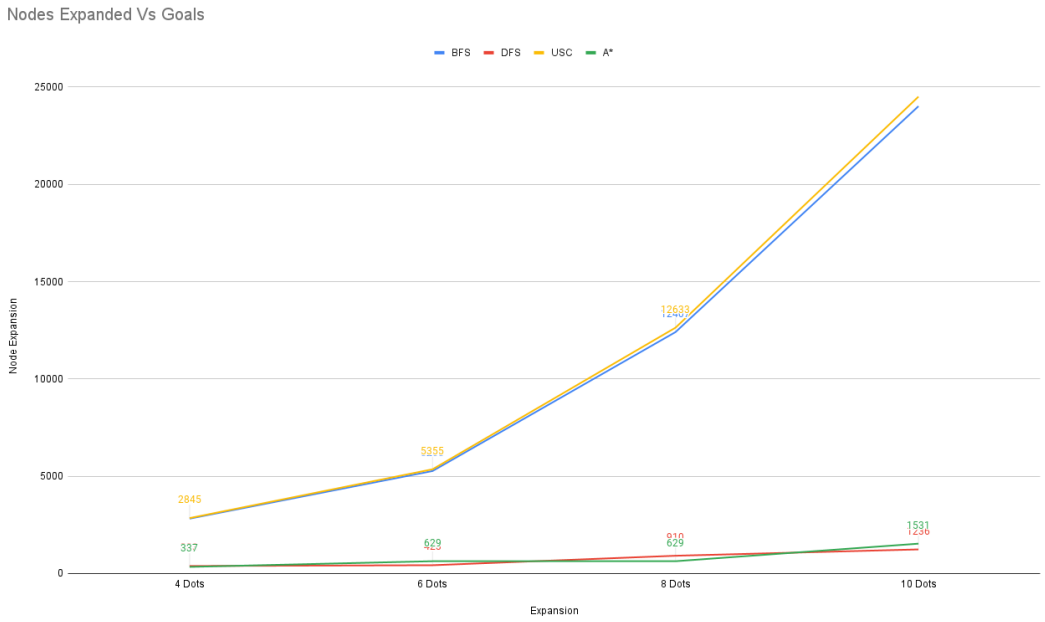
Originally, the continued analysis included applying the search algorithms in practical Pac-Man scenarios to avoid ghosts, however it will become evident in the analysis why this wasn't feasible. So to further analyze the data, the algorithms were applied to the medium sized map under 4 different tests. Each test includes adding two more goals at random points and analyzing run time, score, and node expansion for each algorithm. The results are seen here:

Time VS Goals



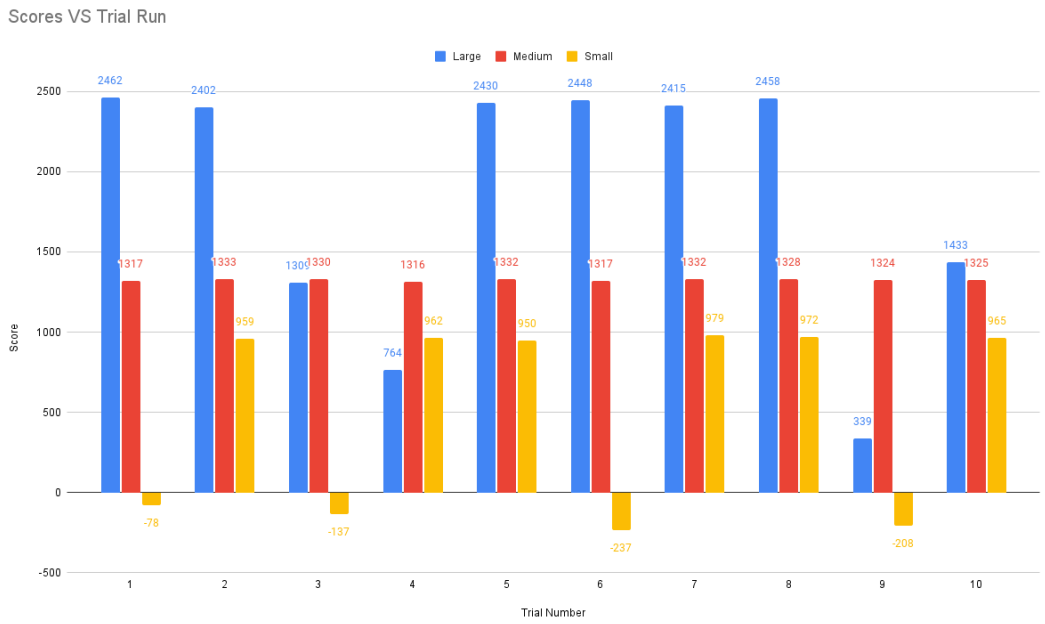
Scores VS Goals





While the algorithm analysis provides quite a bit of information on efficient methodology for environment traversal of the Pac-Man agent there must also be consideration of ghosts. While the algorithms did not lend themselves for efficient dynamic analysis of the environment there was respite found in reinforced learning, particularly with approximate Q-learning. Straight-forward Q-learning was only effective on a small grid as attempts to train the agent the with larger systems crashed our test computer, but due to the symmetrical nature of the game board, approximate Q-learning could be applied on larger board sizes. For this section the tests included running 50 training episodes on small, medium, and large classic boards and then having the Pac-Man agent play through 10 normal games on each board. Results are seen below:

Map Size	Training Time(Sec)	Win Rate
Small	1.54	6/10
Medium	6.24	10/10
Large	57.01	7/10



5 Analysis

5.1 Baseline Analysis

Now that the data is laid out, sound conclusions about which algorithms are efficient and practical for Pac-Man can be made. To start the baseline experimentation provides data that lines up with our previous understanding of the algorithms which we can use to build a more complex understanding. Inherently, when the map size grows it can be seen that algorithms must do more work as seen in computational time for each algorithm. This is expected as there are more nodes that must be searched in order to reach the goal state, as seen in the node expansion graph. BFS has steady growth throughout it's time and node expansion second only to DFS. This tracks known knowledge as BFS needs needs to expand the amount of nodes it traverses as the map size increases,but does not fall victim to the huertisc generation and cost computations of A* and UCS while still being able to result in an optimal solution and retrieve the highest score possible. UCS has the 2nd largest time growth rate which follows what is expected as UCS needs to compute and keep track of path costs. This results in the largest node expansion as more paths are considered in the cost analysis as seen in the node chart. Still this algorithm does find the optimal solution, so there is potential here. DFS has the fastest time of the algorithms, since it's not looking for an optimal solution rather just the first solution. As seen in the score chart, DFS sacrifices score for a faster search time which might not lend itself to practical application when considering fractions of seconds. A* has the largest time costs and rate of time increase. This is due to the increase time it takes for hueristic generation, but this seems to pay off as A* has an average node expansion of 358 2nd to DFS's average expansion of about 351.66. A difference of about 7 nodes while also finding the optimal solution shows that A* has quite a bit of potential for practical application.

5.2 Expanded Algorithm Analysis

Now that there is a baseline of data, complex analysis can be carried on the expanded data. The expanded search was carried out on the medium sized map with two randomly assigned dots were places on the map and the the algorithms were carried on the maps. The algorithms showed expected results with their growth rates in terms of time complexity. DFS had the smallest rate of time growth, as it's searches were non-optimal. This is seen in the Scores vs Goals graph where DFS actually ended up with a negative score due to it's poor performance. It should not be considered when creating a Pac-Man AI. The time and node expansion benefits are not justified when a losing score is retrieved. The BFS and UCS both showed exponential growth with each trial in the form of $O(b^d)$ and $O(b^{1+c/e})$, respectively. We were able to see UCS, A*, and BFS all maintained optimal searches by looking at the score chart. When considering the time A* and UCS had larger time growth than BFS by about 60 percent when reaching 10 dots on the map. However, even BFS doesn't lend itself to being very applicable in a dynamic game settings. This was an issue with our initial design. We tried to implement the search with the ghosts, but the searches were not fast enough to keep up with the dynamic environment. That being said while the search time for BFS and UCS were taken up with searching through the nodes when looking at the Node Expansion vs Goals graph it can be observed that A* has node expansion more akin to DFS than the other two. This shows that the search time was heavily delayed by the heuristic generation seen in the approach section. Now, Pac-Man is a well-known game, so it goes to reason that path finding heuristics could be generated beforehand, making A* a prominent algorithm for Pac-Man. There is a caveat, though. While A* provides a great search algorithm for a static environment when the ghosts are introduces it would fail. This is where we can look to reinforced learning to make up for these gaps.

5.3 Reinforced Learning

Reinforced learning helped fill in the glaring holes the previous algorithms couldn't fill. Particularly interacting with dynamic environments. Q-learning allowed the Pac-Man agent to learn how to play the game to earn as many points as possible while avoiding the ghosts. The first Q-learning implementation which was straight forward training the agent to play the game had to much processing overhead and once the training was complete on any map except the smallest grid caused the host computer to crash. However, since the Pac-Man map is symmetric approximate Q-learning can be applied. Where the Largest Map wouldn't even finish training under the base Q-learning algorithm with approximate Q-learning the agent was able to be trained in just under one minute and win 70 percent of the games played with consistent scoring across all won matches. With this it's determined that the most efficient system for a dynamic game wouldn't be via a search algorithm rather one that is trained and learns what optimal pathing is. This doesn't mean that there isn't a place for search algorithms in games. There are many places they can be used to improve the efficiency of search algorithms. These notions are expanded upon in the future work discussion.

6 Conclusion and Future Work

The goal of this project was to find the most efficient algorithms to create Pac-Man playing agents. While in the end, the data shows that when creating a Pac-Man agent search algorithms can be relatively efficient in finding optimal paths in order to get the highest score, they lack the time and space complexity to properly traverse a dynamic environment. There was hope, however, in reinforced learning where the agents could be trained in optimal actions to score the highest possible points. In terms of the search algorithms, if the map and heuristic data was known previous to playing the game then A* would be the best bet as it had node expansion akin to DFS, but with the optimal search of BFS and UCS. However, due to the time needed to generate Heuristic the next best bet would have been BFS as it was faster than UCS and lead to an optimal outcome. Beyond optimal paths, in order to deal with ghosts, reinforced learning is ideal. Specifically approximate Q-learning such that it would be easier to take advantage of the symmetric maps. In the future it would be good to apply the heuristic search to approximate Q-learning. As it seems that approximate Q-learning learns via optimal actions by applying A* search data to the Q-learning algorithm it could further optimize the learning process as the agent would skew towards optimal paths increasing the score. It should be noted that it might be worth looking into the heuristic generation as it is the only factor slowing the A* search down.

7 Contributions by Group Members

Our work was divided as follows:
Muhammad Ahsan

- Wrote multiple sections which include: Related Work and Literature Review, Experiment Design and Results, Analysis, Conclusion and Future Work.
- Modified code for search algorithms, ran tests and generated data visualisations.
- Revised document.

Ninh Tran

- Wrote multiple sections which include: Abstract, Introduction, Related Work and Literature Review, Approach, Conclusion and Future Work.
- Revised document.

References

[1] Y. Björnsson and K. Halldórsson. Improved heuristics for optimal path-finding on game maps. *AIIDE*, 6:9–14, 2006.

[2] L. Bom, R. Henken, and M. Wiering. Reinforcement learning to train ms. pac-man using higher-order action-relative inputs. In *2013 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*, pages 156–163. IEEE, 2013.

[3] X. Cui and H. Shi. A*-based pathfinding in modern computer games. *International Journal of Computer Science and Network Security*, 11(1):125–130, 2011.

[4] A. G. Daniel Harabor. Online graph pruning for pathfinding on grid maps. *Proceedings of the AAAI Conference on Artificial Intelligence*, 25(1):1114–1119, 2011.

[5] A. Gnanasekaran, J. F. Faba, and J. An. Reinforcement learning in pacman. *See nalso URL http://cs229. stanford. edu/proj2017/final-reports/5241109. pdf*, 2017.

[6] J. B. Pal, S. Modak, and D. Chatterjee. Designing of search agents using pacman. 2019.

[7] S. Samothrakis, D. Robles, and S. Lucas. Fast approximate max-n monte carlo tree search for ms pac-man. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(2):142–154, 2011.

[8] J. Smith and S. Cayzer. Teaching problem solving and ai with pacman. 2010.

[9] M. L. Tom Pepels, Mark H. M. Winands. Real-time monte carlo tree search in ms pac-man. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(1):245–257, 2014.

[10] P. R. Williams, D. Perez-Liebana, and S. M. Lucas. Ms. pac-man versus ghost team cig 2016 competition. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8. IEEE, 2016.

[11] C.-S. Wu. Ai-pacman. https://github.com/jasonwu0731/AI-Pacman, 2016.

[12] P. Yap. Grid-based path-finding. In *Conference of the canadian society for computational studies of intelligence*, pages 44–55. Springer, 2002.

[13] B. W. Zhang An, Li Chong. Rectangle expansion a pathfinding for grid maps. *Chinese Journal of Aeronautics*, 29(5):1385–1396, 2016.

[14] M. Zikky. Review of a*(a star) navigation mesh pathfinding as the alternative of artificial intelligent for ghosts agent on the pacman game. *EMITTER International journal of engineering technology*, 4(1):141–149, 2016.