Artificial Intelligence
Final Project

Kevin Lin, Kong Huang, Samir Mohamed

December 4, 2015

Contents

Chapter 1

Algorithm Implementations

1.1 Main Evaluation Function

The main evaluation function gives a score depending on the given game state. First, if the game has ended with 0 lives being left, a value of -9999 is given. Otherwise, the evaluation will take into account the current level, the number of lives remaining, the distance from Pac-Man to the ghosts, the distance from Pac-Man to the nearest power pill, the number of active pills, the number of active power pills, the distance from Pac-Man to the nearest active pill, the score, and the total time. In particular, the highest weights are given to the level and number of lives remaining, while the lowest weights are given to the distance from Pac-Man to the nearest pill. All of these factors were chosen with the mindset of beating the most levels with as many points as possible.

1.2 Breadth-first Search

Breadth-first search is a simple tree-search algorithm that starts at a root node and explores all neighboring nodes first, before moving on to the next depth. The implementation used for Pac-Man, relied on a maximum depth (the value 50 was used) to determine which level to stop searching for possible moves.

To start, the current game state is used as the root node. For each possible move that Pac-Man can make from this node, the game state is copied, and advanced one step with Pac-Man using said move. Each of these new game states have a depth of 1, and are added to a queue. While the queue is not empty and time has not run out, one game state in the queue is popped out. Since a queue is used, the first game state put on the queue is chosen. First, this game state is evaluated using a evaluation function. If the evaluation is higher than all previous evaluations, the initial move that was made to create this game state is set as the best possible move. Next, if the depth has not reached the maximum depth, this game state is copied and subsequently advanced one time for each possible move that Pac-Man can make. However, for most situations it would not make sense for Pac-Man to move back and forth, so if the move was the opposite of the previously made move (e.g., moving left then right), the game state is not duplicated nor advanced. Also, this would greatly reduce the search space, so this condition is beneficial to the efficiency of searching each

subsequent level. Finally, the game state is added to the end of the queue with a depth of 1 greater than the previous game state.

At the end, the algorithm should find the ideal state within a specific depth and the initial move used to find the ideal state is returned.

1.3 Depth-first Search

Depth-first search is a simple tree-search algorithm that starts at a root node and explores as far as possible along each branch of the tree before backtracking back to the root node. The implementation used for Pac-Man relied on a maximum depth (the value 50 was used) to determine which level to stop searching for possible moves. If this was not used, only one path from the root node would be searched.

The implementation followed the same steps as implementing breadth-first search (see Section 1.1) with a minor difference of utilizing a stack instead of a queue. The difference is that a stack is LIFO (last in first out), so the game state that is popped from the stack will be the one that is put on the stack last.

1.4 Iterative Deepening

Iterative deepening is a tree-search algorithm that runs a depth-first search over and over with increasing depths. This ensures that you never end up exploring along infinitely long branches because the length of the path is capped at a certain depth each iteration. It combines the space-efficiency of depth-first search with the completeness of breadth-first search. The implementation used for Pac-Man relied on an iterative depth (the value 5 was used) and a maximum depth (the value 50 was used) to determine how many levels to iterate and when to finally stop searching.

The implementation followed the same steps as implementing depth-first search (see Section 1.2) using the iterative depth as a "reset" point. When the depth of a game state reaches the iterative depth, the new advanced game states are added to a new stack. Finally, when the original stack has run out, the original stack is replaced with the new stack.

1.5 A*

A* is a search algorithm that starts at a root node and uses a best-first search to explores nodes in a order based on the combination of a heuristic and the known cost of the path. The case of Pac-Man is interesting because the goal state is not a simple matter of reaching a destination on a map. If the goal state eating all of the pills on a stage were used, the algorithm would neglect being possibly eaten by the ghosts. If the goal state of just getting as high of a score as possible, the algorithm would have trouble if there were no simple actions nearby to increase points. To implement this for Pac-Man the main evaluation function (see Section 1.1) was used as the goal, while the heuristic to estimate the distance is simply the number of remaining pills multiplied by a factor.

The implementation followed the same steps as implementing breadth-first search (see Section 1.2) with a minor difference of utilizing a priority queue

instead of a queue. The difference is that the priority queue can sort the queue based on preference (the combination of the evaluation of the state and the number of remaining pills were used).

1.6 Hill Climber

The most basic of the optimization algorithms, the hill climbing algorithm generates neighboring solutions until there are no better neighboring solutions. In the case of our Pacman implementation, it looks at all the possible moves around where the player currently is. Afterwards, it evaluates each move and simply picks the one with the best score. This will sometimes result in situations where the algorithm gets stuck in a local maxima.

1.7 Simulated Annealing

Simulated annealing is similar to the hill climbing algorithm with one small difference. If the next solution is not better than the previous solution, generate an acceptance probability and *maybe* move to it depending on that probability's comparison to a random number. This ensures that the algorithm will sometimes elect to keep the worse solution, allowing it to break out of local maxima. Our acceptance probability is based on the current level time, and goes down as the game level timer goes up.

1.8 Evolution Strategy

Samir didn't come through.

1.9 Genetic Algorithm

Samir didn't come through.

1.10 Alpha-beta Pruning

Alpha-beta pruning is away of finding an optimal solution to the minimax algorithm while *pruning* the subtrees of moves that won't be selected. The algorithm gets its name from two bounds that are passed during the calculation. These bounds limit the solution set based on what's already been seen in the tree. The beta, also referred to as the MinValue in our code, refers to the minimum upper bound of possible solutions. The alpha, also referred to as the MaxValue in our code, refers to the maximum lower bound of the possible solutions. The alpha and beta can be referenced as the best and worst move for Pacman respectively.

Our implementation of the algorithm ignores searches in the opposite direction since they are already covered elsewhere in the tree. The MIN and MAX functions call each other to get a good reference alpha and beta value while the main function is used to decide a proper decision on what move to make depending on those values.

1.11 k-Nearest Neighbor

k-Nearest Neighbor is a method that utilizes training data to assist in classifying instances to the ones known from the data. To implement this for Pac-Man, replay data of a person playing Pac-Man was recorded and stored as a text file. Using the replay data, many different types of classifications could have been utilized to determine the moves for Pac-Man. For instance, a straight forward classification would examine every instance of the replay data where Pac-Man was located within n spaces of each position of the stage to determine the ideal move at every position. However, this would be the same as telling Pac-Man to move a specific away at every spot on the stage and nothing concerning the pills nor the ghosts would change the movement patten.

The implementation chosen for Pac-Man involved observing the behavior of Pac-Man in the replay, and mimicking it. At every game state, the average distance from all of the ghosts are calculated and the$k$ (set to 10) instances where the difference between that and the average distance from all of the ghosts in the replay are the closest to 0 are used. The k instances are then observed to see if Pac-Man took the behavior of running away from ghosts, or going to take the nearest pill. The behavior chosen the most is then chosen as the optimal behavior for Pac-Man. After finding the proper state to act upon the behavior, the proper move is returned.

1.12 Perceptron

Samir didn't come through.

1.13 ID3 Decision Tree

ID3 decision tree is a method that utilizes data to assist in generating a decision tree. The decision tree will then be used to decide the optimal decision depending on the given situation. For Pac-Man, replay data was already recorded and stored as a text file. Going through the data, different characteristics were used to create the decision tree: the total number of moves Pac-Man can make (left, right, up, and down), what kind of moves Pac-Man can make, and the move of the closest ghost. These moves were chosen because they are some of the important factors when controlling the movements of Pac-Man. If the decision tree was further expanded to include other attributes, the location of the nearest pill and the location of the nearest edible ghost can also be used.

The tree that is generated is in the form of a multidimensional array, where the first index corresponds to the types of moves, the second index corresponds to the move of the nearby ghost, and the third index corresponds to the optimal move for Pac-Man. Using this, if the algorithm simply needs to check the maximum value corresponding to the first and second indices, and return the third index.

1.14 Q-learning

Q-leaning is a reinforcement learning strategy that does not need to rely on a prior dataset to make actions. We did not implement this algorithm because we implemented alpha-beta pruning.

Chapter 2

Algorithm Analysis and Comparisons

2.1 Data

Algorithm Score Comparison Part 1						
Trials	BFS	DFS	Iterative	A*	Hill Climber	Simulated Anneal
1	20560	32300	28270	16520	5840	4000
2	17540	5950	23330	47990	6970	1330
3	25540	9090	17050	12790	7640	23220
4	35980	47900	11060	11050	5480	2970
5	10200	19710	27970	41920	3500	2640
6	31030	32410	28860	37320	5450	4030
7	14890	19360	10160	34530	8480	4590
8	23200	11860	17570	24870	4610	3190
9	48940	12570	29870	29190	1350	3050
10	29920	22380	29700	41850	4480	710
11	28790	24740	18000	57890	6040	2030
12	18100	27360	18950	17590	1350	3040
13	31520	37420	35300	12690	6700	3990
14	40900	60620	7610	25680	7240	6370
15	8240	7210	18080	19960	2790	3820
16	23250	9050	25520	31320	3850	1180
17	19960	45050	27080	11090	4610	9220
18	18420	38640	19950	8460	4610	810
19	18580	20050	36380	19250	7040	5770
20	29700	25590	39300	12760	2090	3420
Average	24763	25463	23500.5	25736	5006	4469

Algorithm Score Comparison Part 2						
Trials	Evolution	Genetic	AB Pruning	kNN	Perceptron	ID3 Decision
1	3040	280	4080	3860	S	220
2	2350	400	6200	3510	A	220
3	6150	290	2660	3860	M	220
4	3360	400	5970	3910	I	220
5	1630	200	4670	3860	R	240
6	2710	200	3780	3600	F	220
7	5580	200	5690	3460	L	250
8	2720	200	6340	3860	A	220
9	4940	290	2420	4070	K	220
10	960	280	20980	3860	E	240
11	1120	580	6590	2670	D	220
12	2860	280	2430	3860	O	220
13	2710	1060	7550	3860	N	220
14	1650	200	3290	3800	H	220
15	4250	200	5940	3860	I	220
16	5020	380	8190	3860	S	220
17	2570	200	4670	2680	S	250
18	710	200	28110	4590	T	260
19	2340	200	12760	5480	U	220
20	1610	200	11260	3600	F	240
Average	2914	312	7679	3805.5	F	5006

2.2 Analysis

As expected, the best performing algorithm out of our implementations was the A* algorithm. Breadth-first search comes close to A* but it looks at a lot more nodes, which makes it slower in comparison. We limited the depth to 50 for BFS to deal with that slow speed. The other tree searches also achieved an average score around 23,000-25,000, possibly due to their similarity in implementation. One thing to note about these search algorithms is the absurdly high standard deviation on the twenty trial runs. The algorithms had a standard deviation ranging from 8800 all the way to 15000. This means that the algorithm either does really well or really poorly.

A seemingly surprising result is that the simulated annealing performed worse on average compared to the normal hill climber algorithm. However, closer analysis shows that simulated annealing has a much higher best score in comparison. The highest score for hill climber tops off at 8480 but simulated annealing tops off at 23220. This is possibly because hill climber can reach situations where Pac-man just gets stuck and moves back and forth while simulated annealing as a way to break out of that loop sometimes.

The evolution and genetic algorithms performed very poorly. One thing to note is that since there are only a small amount of lives for Pac-man, it can't iterate enough times to find an optimum solution through a process similar to

natural selection. It is also possible that our heuristics were messed up, as even with that issue, the score should not be below 1000 in the majority of the cases.

Alpha-Beta pruning performed admirably compared to the rest of the algorithms but performed poorly in comparison to the tree search algorithms. One issue that might be bottlenecking the performance is our calculations of the alpha and betas. If the alphas and betas are not accurate, the algorithm ends up pruning a branch that contains an optimal solution and picks a worse solution.

kNN didn't perform the worst out of the algorithms but it performed bad enough that it never got past level 1. The implementation involved using a replay to attempt to mimic, and the performance issue might stem from the fact that the replay data is not optimal data. Nevertheless, the standard deviation for the twenty trial runs for this algorithm resulted in one of the lowest amongst the algorithms. This means that while the algorithm might not perform as well as expected, it is consistent.

Samir insert Perceptron algorithm analysis here.

ID3 decision tree's implementation is logically sound, however there are some factor that are working against it. The main factor is that there are not enough conditions in the decision tree, which means that

Chapter 3

Custom Algorithm

3.1 Idea

The twelve algorithms discussed in Section 1 were initially examined, and it was clear that the best performing algorithms were the tree-search algorithms. They relied on examining all the game states within reach and it seemed that expanding on these ideas would possibly generate even better scores.

Coming across the Monte Carlo tree search algorithm, it seemed like a combination of it and breadth first search would be possible. The Monte Carlo algorithm relies on reaching decisions by analyzing the moves with most potential and then expanding the search trees on those moves. Essentially the most successful moves will continue to be expanded upon, while the moves that simply fail will never be expanded. At the end, the initial move that resulted in the greatest ratio of successes to failures will be chosen as the ideal decision.

Using the same goal of reaching the furthest level possible with the best score, a new algorithm was developed using the ideas of the breadth-first search algorithm, the Monte Carlo tree search algorithm, and a variety of tweaks in creating game states.

3.2 Implementation

To start, the algorithm takes into account the fact that each position on the Pac-Man board has either 2, 3, or 4 possible moves. For a position with only 2 possible moves, Pac-Man can only either progress (continue moving forward or make a 90 degree turn), or make a full 180 degree turn. The only situation where Pac-Man would willingly turn 180 degrees around is to bait the ghosts to get away from them or get them near a power pill. However, because of this, additional nodes must be created on the search tree to check if moving backwards is a viable move. And as the tree grows, this greatly impacts the efficiency of the algorithm. Additionally, if these 2-move spots were ignored (Pac-Man only has the option to progress forwards), more than half of the Pac-Man board can be ignored because the minority of the board are junctions (spots with 3 or 4 possible moves). Thus, the first part of the algorithm checks to see if the position is a junction. If it is not, the move that correlates to progressing is returned.

If the position is a junction, the current game state is evaluated and set as the original evaluation. Then, each possible move is copied and advanced upon, until a junction is reached. The new game state is then evaluated and is given a depth of 1. If the new evaluation is better than the original evaluation, 1 "success" point is given to the corresponding initial move. If it is worse, 1 "failure" point is given to the corresponding initial move. Also, if the evaluation is worse than the original evaluation by more than n (set to 3000), the game state is going to be completely discarded. The game states that still remain are copied and advanced upon until another junction is reached (depth is set to the previous depth + 1), and the same process occurs. These expansions are done following the same order as the breadth-first search algorithm (see Section 1.2).

As in the implementation of the tree-search algorithms, when the game state node reaches a maximum depth (set to 5), the searching is completed. Looking at the ratio of successes to failures of the initial moves, the initial move corresponding to the greatest ratio is chosen and returned by the algorithm.

3.3 Performance

The performance was unexpected and did not perform anywhere as well as the original tree search algorithms. It is likely that the algorithm is finding good paths (in terms of score and reaching the next levels), but they are being discarded as a result of a poor ratio due to the other possible paths in that general direction. And as good paths are being skipped, poor situations occur. In general, forcing Pac-Man to be obligated to move forward on 2-way paths could be a poor decision as well.

Trials	**Score**	**Level**
1	3060	1
2	4080	1
3	8230	1
4	5640	1
5	8500	1
6	1720	1
7	3950	1
8	3320	1
9	4210	1
10	2360	1
11	2290	1
12	1990	1
13	3570	1
14	5680	1
15	1690	1
16	3550	1
17	1490	1
18	4750	1
19	7020	1
20	510	1
Average	**3880.5**	**1**

References

[1] "Hill climbing" *Wikipedia.* Wikimedia Foundation, n.d. Web. 03 May 2015.