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

1.9 Genetic Algorithm

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

1.13 ID3 Decision Tree

1.14 Q-learning

Chapter 2

Algorithm Analysis and Comparisons

2.1 Data

2.2 Analysis

Chapter 3

Custom Algorithm

3.1 Idea

3.2 Implementation

3.3 Performance

References

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