C.L.Y.D.E
A.I. Final Project

Whitney Mulhern, Giorgio Pizzorni, Kunal Relia

December 4, 2015

1 Part 1 - Algorithm Implementations

1.1 Uninformed Search

None of the Uninformed search algorithms performed well within the PacMan framework even with some modifications to improve performance. This, however, is to be expected as these are not necessarily "smart" algorithms. All of them are inefficient because they spent a lot of resources blindly search adjacent state spaces, only to figure out the one move ahead of the current game state.

1.1.1 Breadth First Search (BFS)

Breadth first search is one of the worst-performing algorithms in our submission, but that is mostly due to the nature of the naive algorithm. Breadth first search looks at all possible adjacent states before then searching each of those states. You can see this happening in the visual of the game when running BFS, which is interesting. The termination condition we used was a limited number of iterations for BFS. This prevents taxing the framework and the controller timing out but still does a significant BFS.

BFS uses a wrapper object TreeNode (which holds some useful information including a game instance and the move taken from parent to get there) to make the implementation a little easier. We also use a queue to store the visited nodes so that we keep track of all of the children in a given level, which creates the breadth-ness of the BFS. We then look at the deepest node we found in the BFS and traverse back up the tree to get the MOVE that would bring our game towards that game. Normally in a BFS algorithm you are looking for a specific condition (value of a node perhaps). In this case we are just trying to demonstrate how traversing the nodes works in a BFS. We then return the most recently found node and try to get there by choosing the move needed to get there.

1.1.2 Depth First Search (DFS)

Depth first search is another one of the worst-performing algorithms. It has a lot of the same issues and limitations as BFS. However, the behavior is exactly the opposite of BFS, yet it is still as inefficient. DFS searches the first child node of each node it reaches. Since the first child will always advance with the same move DFS just results in PacMan trying to go in only one direction (LEFT), thus it has horrible performance. We used the same termination condition for DFS as BFS. There are some other termination conditions that would probably lead to better performance like number of pills eaten, score increase, etc.

The implementation of DFS also uses the TreeNode object and stack. The nodes are pushed onto the stack as they are evaluated for so many iterations. We then look at the deepest node we found in the DFS and traverse back up the tree to get the MOVE that would bring our game towards that game.

1.1.3 Iterative Deepening

Iterative Deepening is similar to DFS, however it was implemented a little bit differently than the first two uninformed search algorithms. There are two termination conditions: the first is that the score has increased by SCORE points, in which case we traverse back up from that node to find the move that brings us towards that game instance. The second is a number of iterations for the algorithm. If the maximum number of iterations is reached the controller returns a random move; this prevents the controller from timing out. This algorithm performs slightly better than both DFS and BFS. That is partially because of the different implementation, but also because iterative deepening is essentially a BFS of a limited DFS, so it doesn't get stuck in any strange behaviors. However, this is still not a ”smart” or well-performing algorithm.

1.2 Informed Search

1.2.1 A*

A* performed relatively well in comparison to the other algorithms. The usage of a heuristic function allowed us to express a wider set of strategies more easily, and less verbosely. However, coming up with a good heuristic was challenging and a lot of time was spent fine tuning the weights to gain better performance. Selecting the weights for the heuristic proved to be more challenging than the actual implementation. Our difficulties here actually inspired part 2 of our project, which largely revolves around evolving the A* heuristic.

Our A* implementation uses a simple Node class which is a small wrapper over a Game object, some logistical information for building the search tree, and some additional information about the game state. We then chose to use a PriorityQueue sorted by move cost allowing us to easily find the lowest cost unexplored node. Because of our choice of data structure, we had to be careful to actually return a move that would've corresponded to a leaf in the tree, instead of a low cost move closer to the root that would be ahead in the Queue.

1.2.2 Hill Climber

Hill Climber performed surprisingly well given how short sighted and simple it is. It was actually, on average, one of the most consistent and well performing algorithms. However, it hits a hard wall. It performs very well early on, but once it reaches an area in the maze with no pills, usually after dying once, it gets suck. At first, we thought this was due to the short sighted nature of the Hill Climber, but its actually the framework. Given a very simple heuristic revolving entirely around score, a Hill Climber would actually not be particularly short sighted in PacMan if properly implemented. Since score doesn't decrease, if we are careful to avoid constantly re-exploring the same nodes, then climber would eventually find his way back to where there are pills. However, there isn't enough time, generally, to simulate far enough because of the time limit on move computations.

Our implementation is very straight forward. For every neighbor, meaning every possible MOVE, we simulate on copies of games, evaluate the score, and move towards the one with the highest score, keeping track of which MOVE this was. We managed to perform roughly five iterations before timing out. Once we are done simulating, we return the MOVE that got us the highest score originally.

1.2.3 Simulated Annealing

We initially struggled to get good performance out of simulated annealing, but after some quick tuning, it quickly become one of our best performing algorithms. With a small enough rate of change, simulated annealing is likely to find a global maximum. But, the underlying controller, Hill Climber, tends to lose way before the temperature ever approaches cold. Keeping this in mind, we spent time tuning the temperature and rate of change to allow for a Climber to usually make it towards cold temperatures near the end of their life. With a moderate rate of change, and a moderately large starting temperature, it consistently performs very well.

Simulated Annealing did provide a slight implementation challenge, we were unable to implement it as a controller. The performance was abysmal, and it was very difficult to simulate for a long enough amount of time for the temperature change to really have an impact before timing out. Because of this, we wrote it as a function in the Executor class, and it simply loops while using a Hill Climber as a controller to generate moves. This allowed us to completely remove the timeout bottle neck as the Hill Climber is now the only one subject to it, and its a very fast algorithm.

1.3 Evolutionary Algorithms

1.3.1 Evolutionary Strategy

Originally, we struggled implementing an evolutionary strategy since we immediately jumped into attempting to evolve our A* heuristic. However, we eventually settled on a simple mutation driven strategy where we randomly mutate moves in an action sequence. It performed relatively well given we could only run it with small populations and small number of generations otherwise it would time out.

The implementation is pretty straightforward. The algorithm predominantly relies on using PriorityQueues to keep the population sorted. We wrote an Evolutionary wrapper object called Candidate that keeps track of the relevant information like the action sequence, fitness, and corresponding game. We chose to implement the comparable interface and return an inverse sorting based off fitness, so the fittest will be at the front of the PriorityQueues. We generate an initial population with random action sequences and then simulate games with their initially random move sequences. We then choose to keep the 25% fittest members of the population and discard the rest. From these 25%, we randomly select and mutate their action sequences generating new Candidates until the population is replenished. There is no crossover step. The mutation step itself iterates through the action sequence and flips a coin on every element. If it's heads, we select a new move at random, other wise, we don't do anything. After we finish simulating the generations, we take the fittest candidate and return the first move of their action sequence.

1.3.2 Genetic Algorithm

Much like the evolutionary strategy, this was originally a struggle for identical reasons. We ended up with a nearly identical set up to our evolutionary strategy. Like before, we evolve a sequence of actions, but it is now driven by a crossover step. After simulating a generation, we automatically pass the 25% fittest members into the survivor population. However, as we are not mutating, and we are reproducing, we need to make an effort to maintain genetic diversity. To do so, we draw another 25% of our survivor population randomly from the non-survivor population. Then, we discard the old population, and randomly select pairs of parents for reproduction. We must ensure the parents are distinct however, because our crossover function doesn't make sense if we were to allow asexual reproduction. In the reproduction step, we iterate over both parents' move sequences and flip a coin at every move. If it's heads, we take the first parent's move and add it to our new action sequence. If it's tails, we take the second parent's move and add it to our new action sequence. We then create a child, give it this action sequence, and continue. At the end, we remove the fittest member of the population and return the first move in their action sequence.

1.4 Adversarial Search

1.4.1 Alpha-Beta Pruning

A quick note about Alpha-Beta Pruning. This algorithm is not really applicable to PacMan because it is for an adversarial turn based game (like chess). Whereas in PacMan both PacMan and the ghosts operate and perform actions simultaneously so it is difficult to react to ghosts as they are reacting to you.

1.5 Supervised Learning

1.5.1 k-Nearest Neighbor

K-Nearest Neighbor was a fun algorithm to implement. We creates some training data somewhat arbitrarily considering two attributes: nearest ghost and nearest pill (either power or normal). In the training data if PacMan is closer to a pill than a ghost, he will go towards the pill. But if he is equidistant or closer to a ghost he runs from the ghost. In the actual play of the game we get the distances of the nearest ghosts and pill then calculate the distance between the current game state and game states recorded in the training data storing the K nearest. We take the mode of the decisions of the K nearest training data instances (run from ghost or run towards pill) and use that as the decision for this move.

Obviously the performance of this algorithm is going to rely heavily on the training data. We kept the training data small using a simple strategy because a complicated training set is not necessary to demonstrate our understanding of the algorithm. Perhaps a more complicated but, ultimately, better performing training set would be extracted from a human played game as suggested.

1.5.2 Perceptrons

The Perceptron was one of the more difficult algorithms to implement. We used similar training data as in k-Nearest neighbors but tweaked it a little bit to make the strategy a little more simple and easier for the perceptron to model. We implemented an eager learning perceptron. So given the training data we train a randomly initialized wight vector to model the hyperplane. Then during game play the necessary attributes are calculated and evaluated against the hyperplane to determine the best choice.

1.5.3 ID3

We think this was the most difficult algorithm to implement. We again went for an eager learning approach where we trained a model, in this case a decision tree, from some training data then during game play we evaluated the game state against the static model to make the "best" decision.

Given a large training data set concerned with 3 attributes (nearest ghost, nearest power pill, nearest normal pill) we then created a recursive function that is called when the controller is initialized. At each node, as long as all the data in the resulting set didn't evaluate to the same decision (that is the base case), we calculate the information gain for each of the three attributes. We then branch that node based on the attribute with the most gain and create children based on the values of that attribute in that set. We then evaluate each of those children in the same way and store this whole tree in the controller. Then during game play, we calculate the attribute values for the given game state and evaluated those against the tree model until we reach a leaf node that yields a decision.

Unfortunately, there are still some major bugs in our code so we have not been able to finish debugging or test it. We are fairly certain the high-level logic of the code is correct.

1.6 Reinforcement Learning

1.6.1 Q-Learning

Probably this turned out to be a more difficult algorithm to implement than the most difficult one too. Though we were able to code the algorithm and make it run independently, we were not able to map the requirements of PacMan for this algorithm. Though we managed to update Q(s,a) for a depth up to n actions, we weren't able to fetch the actual rewards (Pills, Ghosts, PowerPill,etc.) from the game. Unfortunately, this means that we can't record the scores and draw a conclusion from this.

2 Statistics and Comparisons

Most of the algorithms perform decently well and all of them are consistent across a large number of trials. The averages of 20 trials and the averages of 100 trials are relatively close. The worst performing algorithms are the uninformed search algorithms, unsurprisingly. The best are Perceptron, simulated annealing, and hill climber. This makes sense that the smarter, more complicated, and better trained algorithms generally perform better. One surprise the the drastic difference between the performance of the evolutionary strategy and that of the genetic algorithm. We think that is due to the fact that, at least in the PacMan framework, it is more useful mutate successful heuristic weights as opposed to combining several successful ones. Since there are a lot of attributes and a lot of vectors at play that are correlated in complicated ways, mutation seem better because they don't drastically change multiple vectors in different ways, whereas crossover does. Overall, the algorithms performed as expected.

Algorithm	Score
DFS	120.0
BFS	225.5
Iterative Deepening	251.0
A*	919.0
Hill Climber	2806.5
Simulated Annealing	2403.0
Evolutionary Strategy	2120.0
Genetic Algorithm	666.5
k-Nearest Neighbors	2262.5
Perceptron	2731.5
ID3	??
Q-Learning	??

Table 1: Avg score of algorithms with 20 trials.

Algorithm	Score
DFS	120.0
BFS	240.9
Iterative Deepening	272.9
A*	1017.8
Hill Climber	2395.7
Simulated Annealing	2464.7
Evolutionary Strategy	2423.0
Genetic Algorithm	916.4
k-Nearest Neighbors	2412.8
Perceptron	3028.7
ID3	??
Q-Learning	??

Table 2: Avg score of algorithms with 100 trials.

3 Part 2 - C.L.Y.D.E

From the moment we heard the words "evolutionary computation", we were hooked. As we began to implement the evolutionary algorithms, we realized we really wanted to use them in some way for part 2 of this project. Our original attempts at implementing the algorithms were largely unsuccessful as we tried to evolve a different controller from within a controller. We settled for more boring evolutionary algorithms, and decided to focus on this for part 2. We named our project C.L.Y.D.E. in honor of our favorite PacMan ghost, Clyde. It stands for Cambrian Life Yerking Dazzling Explosion.

Initially, we began by writing an evolutionary strategy, in the Executor class, that would evolve our A* heuristic through mutation of weights. This worked pretty well, and we managed to get some decent results. Average scores would consistently increase across generations. We then wrote a genetic programming algorithm that worked horrendously. This occurred because we didn't have enough variance in our initial population, and so we would converge to a sub-par solution. We wrote a small hack to make the initial population five times bigger than requested, and to seed it much more randomly than anything else in the algorithms. From there we would pick and maintain the specified population size, and it worked much better. At this point, we wrote a third evolutionary algorithm that combined both a crossover step and a mutation step. It's performance was largely mediocre, it would oscillate because the two strategies were at odds. So, we set out to do two things: write an evolutionary algorithm that combined both a mutation phase and a crossover phase that performed well, and to systematically test different evolutionary strategies against each other.

We then came up with what is now our final idea. We create a randomly seeded population and make six copies of it. This actually proved to be more than slightly annoying from a technical perspective as we would get a lot of strange subtle bugs because we didn't realize we were only making shallow copies of the populations, and so the algorithms actually interfered with each other. Once we fixed this issue by learning about copy constructors, we were ready to begin.

We decided to combine our evolutionary algorithms with some concepts derived from simulated annealing to see what would happen. We ran our three evolutionary algorithms, mutation, crossover, and combination with and without concepts drawn from simulated annealing. Our evolutionary algorithms have randomness in the mutation and crossover phases, so we used a temperature measure to provide a little heat and spice things up initially. The heat measure proportionately increased the maximum random values used in the mutation and crossover functions and it cooled down a bit every generation.

So, with our identical populations we ran the combination of algorithms drawn from {(Mutation, Crossover, Combination) , (Normal, Annealing)}. However, it took somewhere in the ball park of 20 minutes to run every combination for even very very small populations and number of generations. We wanted to be able to gather meaningful statistics in our lifetime, so we decided to multi-thread it. We created an EvolThread object that implemented runnable, and spawned a thread for each of the six algorithms, and it worked. It was still slow, but it was quite literally six times faster now, and we managed to run a handful of trials. We were, and are, still limited heavily though and are unable to run jobs with large generations or populations. We, unfortunately, have not had enough time to gather meaningful data. Moreover, the algorithms don't perform well. They are either too random, or not random enough, and don't have enough time to overcome this because of our run time limitations.

Our idea had potential, but we didn't have the time or computational resources to fully realize it. From the few trials with tiny sample sizes that we did manage to run, we managed to reach absolutely no conclusions. It was largely random since our algorithms don't have enough time to converge. No one algorithmic combination stood out as the best or worst, and we didn't notice any sort of convergence on the chosen weights either. Once we managed to run a slightly larger trial, a population of 20 for 20 generations, we noticed that there was a slight mistake in our crossover function that made it many times more random than we wanted. Here are some results. G:5 P:10 means that column corresponds to results for a trial consisting of 5 generations and a population of size 10. Algorithms ending in S are the simulated annealing variants. M stands for Max Score, and A stands for Average Score.

	G: 5 P: 10	G: 5 P: 20	G: 10 P: 10	G: 10 P: 20	G: 20 P: 20
Mut	M: 1760 A: 897	M: 1090 A: 548	M: 1310 A: 801	M: 2130 A: 807	M: 2400 A: 963
Cross	M: 1860 A: 703	M: 1970 A: 539	M: 2260 A: 639	M: 2030 A: 551	M: 4350 A: 827
Comb	M: 1290 A: 743	M: 1980 A: 556	M: 1610 A: 732	M: 2630 A: 747	M: 2240 A: 846
MutS	M: 1700 A: 843	M: 1730 A: 578	M: 1630 A: 828	M: 1970 A: 784	M: 4150 A: 925
CrossS	M: 2620 A: 591	M: 3910 A: 515	M: 2130 A: 721	M: 1630 A: 602	M: 2140 A: 487
CombS	M: 1940 A: 770	M: 2560 A: 624	M: 1720 A: 871	M: 2310 A: 808	M: 6480 A: 1143

Table 3: Cambrian Explosin Data