Homework 4

Gurkirat Singh, Jessica Ng
gus011@ucsd.edu, jen030@ucsd.edu
February 18, 2016

1 Descriptions & Algorithms

1.1 Problem 1:

Description: Create an evaluation function for Reflex Agent such that actions obtained from the agent will reliably win in a Pacman game on the testClassic layout.

Algorithms:

Our evaluation function is $f(P) = \begin{cases} -\infty, & action = STOP \text{ or } ghost \text{ is on } pacman \\ -(minF - distSG), & ghosts \text{ are scared} \\ -(minF), & otherwise \end{cases}$

minF = smallest distance to the next food, which includes the capsule
distSG = distance to scared ghosts that are reachable in time and are closer than minF

We would like to minimize the number of actions that Pacman remains still because it is typically unproductive so we return negative infinity when a STOP action is one of the legal actions. We also want to minimize running into the ghost, hence the former and latter cases should be the least of our priority when choosing a priority action to perform. A state where the game has been lost is unfavorable, so the value is also set to negative infinity. We know that it is advantageous to the pacman to eat the capsule and attach the ghost when it is nearby. Therefore, we subtract the distance of the closest ghost so we give eating the ghost a priority when the closest food is further than the closest ghost. If the food is closest than the scared ghosts, then we make eating the food a priority. We negate all of these values so that the closest food is a higher priority than larger distant foods.

1.2 Problem 2:

Description: Implement the minimax algorithm for MinimaxAgent such that actions obtained from the agent's getAction() function will return the optimal action.

Algorithms: This problem utilizes the minimax algorithm. MinimaxAgent has a function named getAction() which should return Pacman's next action in the game. To determine which action that Pacman should take, we apply the minimax algorithm and have Pacman take the move with the maximum heuristic integer value at the end.

The algorithm takes in several parameters including the gameState, depth, and player type.
The gameState is utilized to generate the successor states from taking various actions.
The depth determines how large the search tree will be.
If the playertype is Pacman, then the function will find the action with the maximum value and return them is a tuple.
If the playertype is a ghost, then the function will find the action that minimizes the overall score and return that as a tuple.
In order to find the maximum and minimum values, the algorithm will recursively find the max/min values further down the tree and return those values upwards.

The final score returned is the maximal/optimal move for the player assigned a maximum value.

1.3 Problem 3:

Description: Implement the minimax algorithm for AlphaBetaAgent using alpha-beta pruning. The resulting runtime should be faster than the MinimaxAgent for the same test-case/map.

Algorithms: This problem utilizes the minimax algorithm with alpha-beta pruning. Once again, AlphaBetaAgent has function named getAction() which should return Pacman's next action in the game, and minimax is utilized to determine that action. However, alpha-beta pruning shortens the search process by pruning off branches that are unnecessary to search through.

The algorithm takes in the same parameters as minimax, but also has additional parameters alpha and beta.
The parameters that are the same as minimax are utilized for the same reasons.
The alpha and beta are used to determine the global minimum and maximum for the entire tree.

We know that if the value of a successor state is greater than beta for a maximum run-through of the algorithm, then it means that the min state one recursion above it will never choose any of the successors, so we may prune off the search there.

If the value of a successor state is less than alpha from a minimum run-through of the algorithm, then it means that the closest max state above it will never choose a value from this subtree, so we may prune off the search there.

1.4 Problem 4:

Description: Implement the expectimax algorithm for ExpectimaxAgent such that actions obtained from the agent's getAction() function will be determined by the algorithm.

Algorithms: This problem utilizes the expectimax algorithm. This algorithm is similar to minimax except that it includes a state for random chance.

This algorithm takes in the same parameters as minimax and utilizes them for the same purpose.
For pacman specifically, we assume that all ghost agents will randomly choose their actions, so we propagate the expected value upwards in such a recursive call.
For pacman agent, we want pacman to yield the best results possible, so we do the same as we did in minimax – return the maximum value move.

1.5 Problem 5:

Description: Implement a better evaluation than in problem 1!

Algorithms: For this problem, we create a new evaluation function.

Our evaluation function is $f(P) = \begin{cases} -\infty, & action =' STOP' or\ ghostPos = pacmanPos \\ \frac{1}{minFoodDist} - \frac{1}{minGhostDist} + score, & no\ nearby\ scared\ ghost \\ \frac{1}{minFoodDist} + \frac{1}{minScaredGhostDist} + score, & there\ are\ nearby\ scared\ ghosts \end{cases}$

In the case that the move is STOP or the ghostPos = pacmanPos, we want to return the minimum value possible. This is because these are the moves we consider least optimal. We want to keep Pacman moving and if ghostPos = pacmanPos, then the game has been lost.

In the case that there are no nearby scared ghosts, we want the action that is closest to the nearest food. In order to achieve this we use 1/minFoodDist. Large distances will yield smaller results as we use the distance as a fraction, and the smallest food distance will yield the highest score in this case. In this case, we subtract the 1/minGhostDist because we wish to maximize the distance between Pacman and the ghost when the ghost is not edible.

In the case that there are nearby scared ghosts, we prioritize this state over other states. We add the distance of the first scared ghost that is closer than the minimum food distance. This places its priority over the actions that move toward the smallest food distance as there is an additional factor being added in.

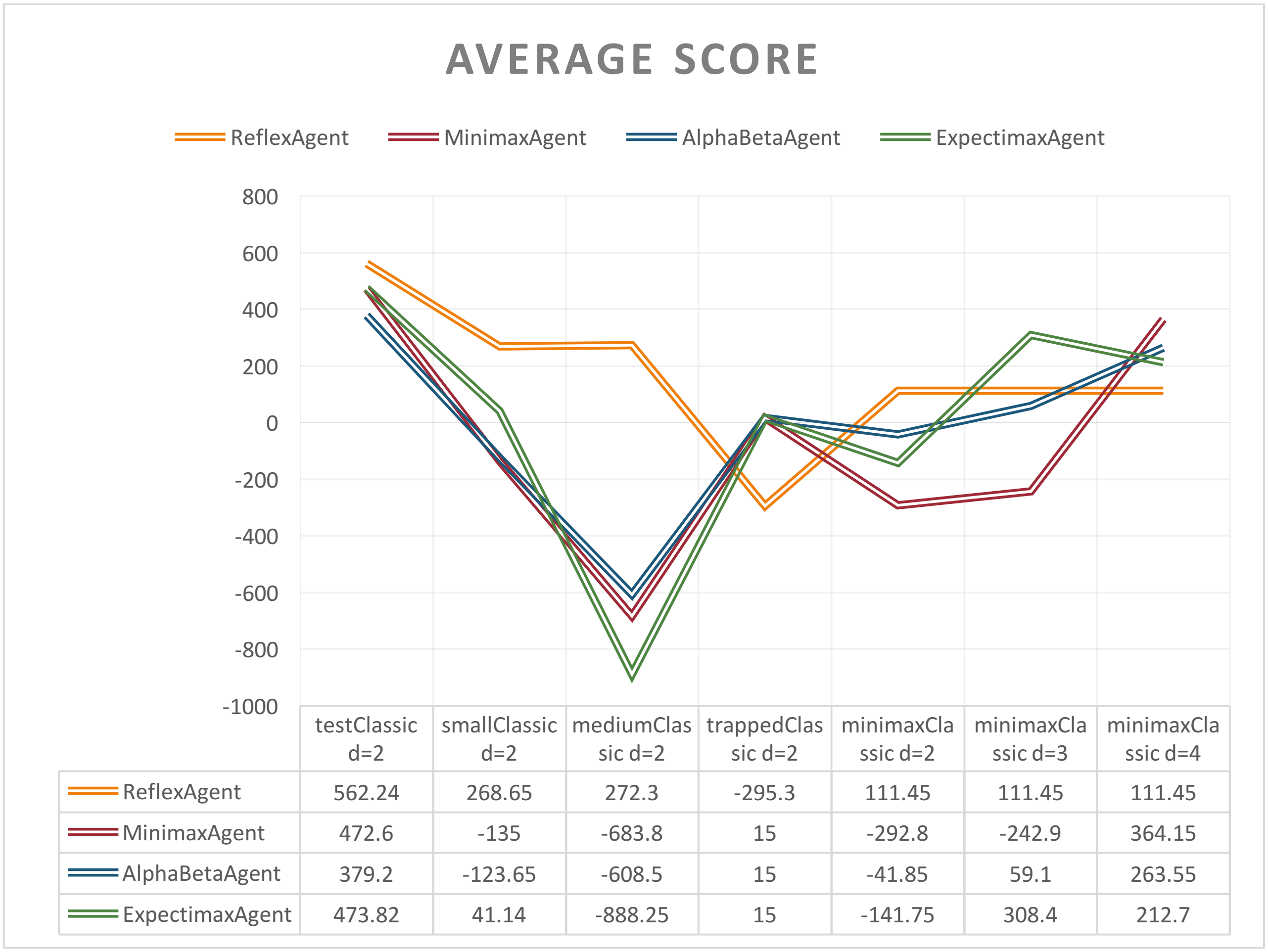
2 ANALYSIS

2.1 RANGE OF OUTCOME VALUES

	ReflexAgent	MinimaxAgent	AlphaBetaAgent	ExpectimaxAgent
testClassic d=2	560 - 564	170 - 554	-86 - 538	254 - 552
smallClassic d=2	-410 – 1563	-311- 840	-524 – 962	-332 – 1607
mediumClassic d=2	-347 – 429	-2027 – 52	1452 – 312	-1119 – 2094
trappedClassic d=2	-502 – 532	-502 – 532	-502 – 532	-502 – 532
minimaxClassic d=2	-493 – 516	-499 – 516	-500 – 516	-498 – 516
minimaxClassic d=3	-493 – 516	-500 – 514	-496 – 513	-510 – 513
minimaxClassic d=4	-493 – 516	-496 – 516	-492 – 516	-493 – 516

We sampled across multiple layouts and created a chart listing the range of values that resulted from each of the Agents. What we can see is that the range of values for every agent ranged a significant amount, and this range increases for more complex maps. i.e. MediumClassic is the most complex map sampled and its range is far larger than other maps. We can see even with increasing depth, the range of values for the same map will remain the same. The range of values varies for any type of agent.

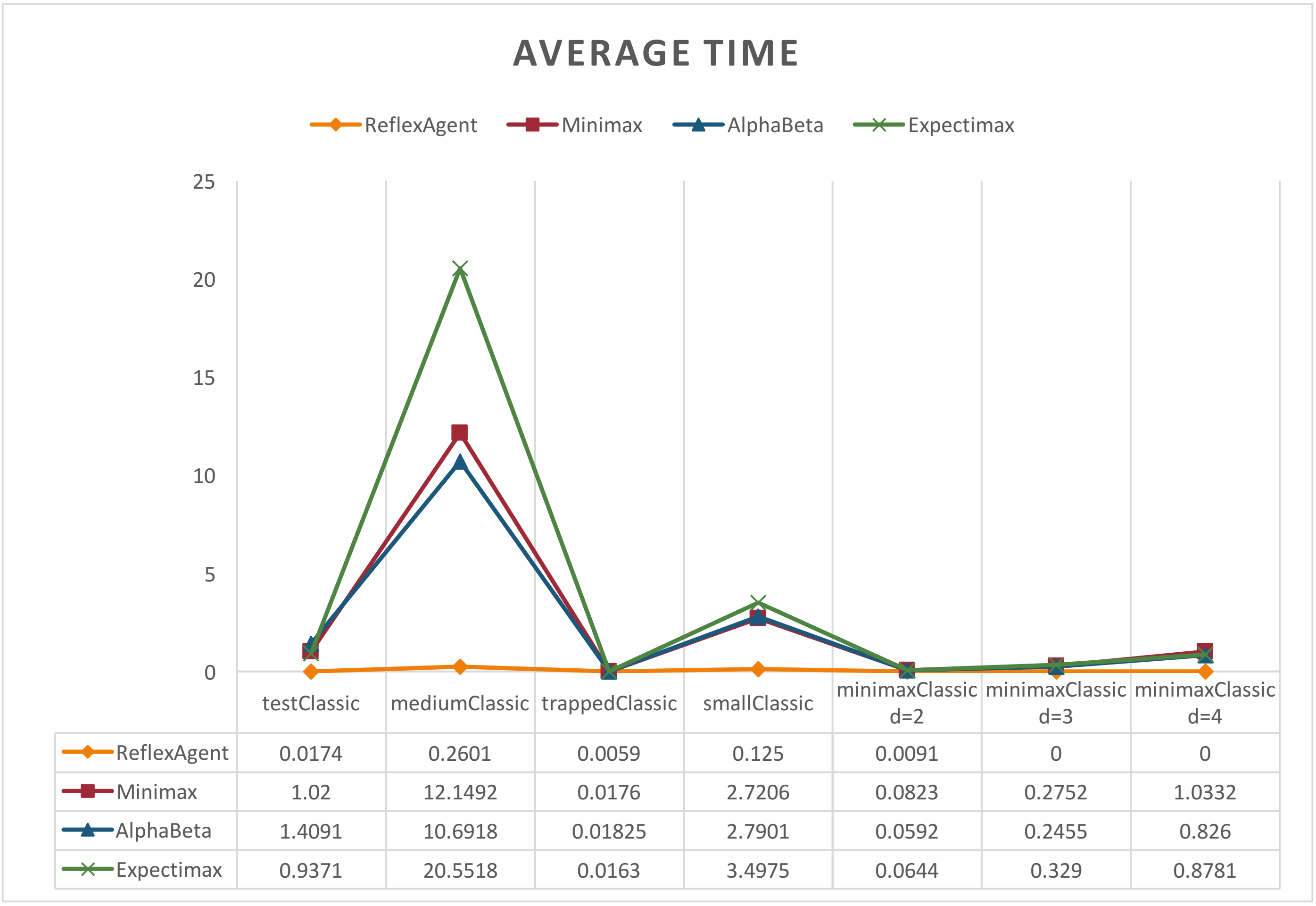
2.2 AVERAGE SCORE



	testClassic d=2	smallClassic d=2	mediumClassic d=2	trappedClassic d=2	minimaxClassic d=2	minimaxClassic d=3	minimaxClassic d=4
ReflexAgent	562.24	268.65	272.3	-295.3	111.45	111.45	111.45
MinimaxAgent	472.6	-135	-683.8	15	-292.8	-242.9	364.15
AlphaBetaAgent	379.2	-123.65	-608.5	15	-41.85	59.1	263.55
ExpectimaxAgent	473.82	41.14	-888.25	15	-141.75	308.4	212.7

We sampled the average score of multiple layouts and depths to compare differences and similarities. We can see from the results that the performance of all agents suffer increasingly as the complexity of the layout increases (seen from testClassic, smallClassic, and mediumClassic). ReflexAgent fares the best with increasing complexity.
We see that in the case that the pacman is trapped, agents other than ReflexAgent perform much better.
The other observation that we can conclude is that the higher the depth, the more accurate the result as can be seen in the last 3 tests performed.

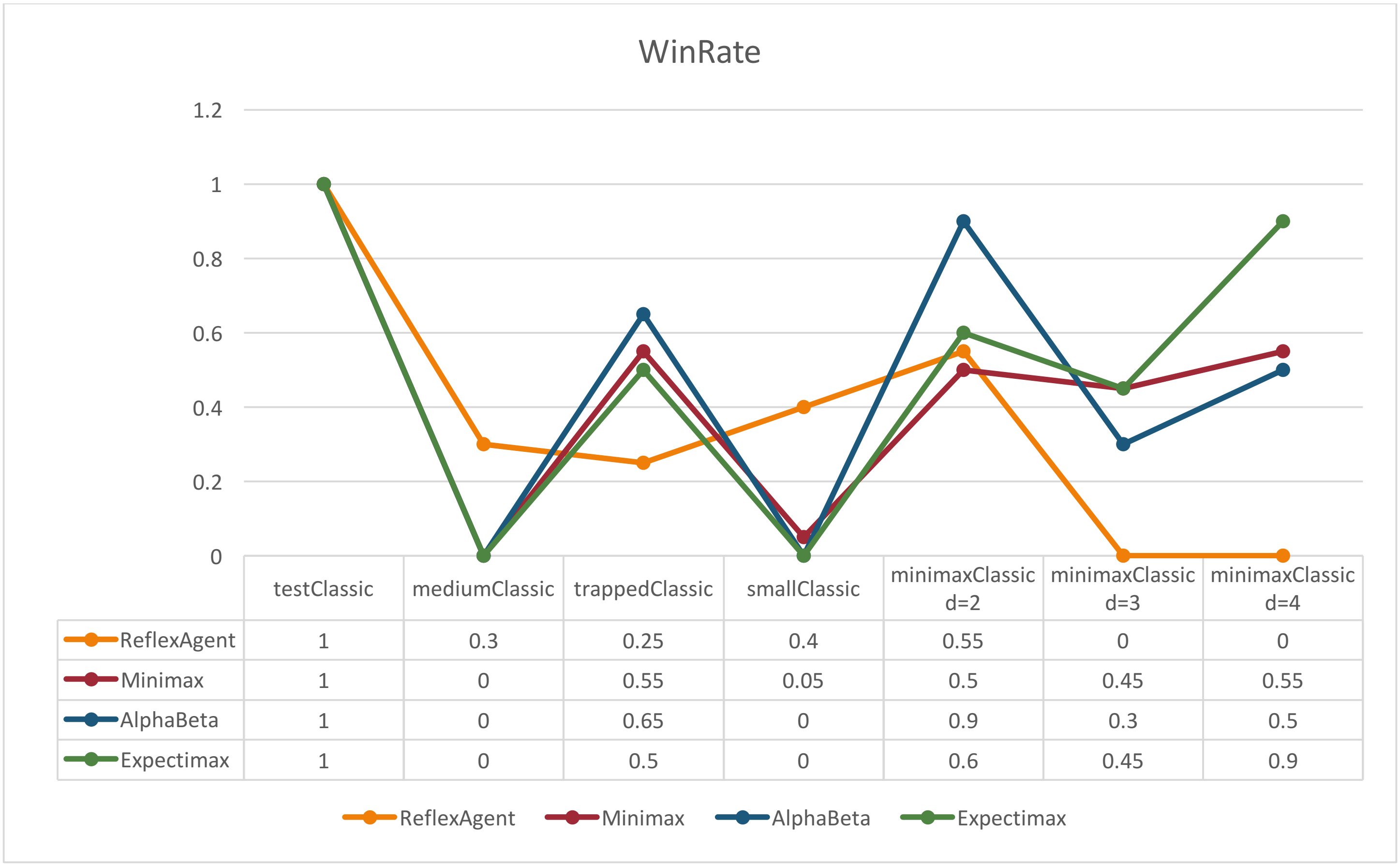
2.3 AVERAGE TIME



	testClassic	mediumClassic	trappedClassic	smallClassic	minimaxClassic d=2	minimaxClassic d=3	minimaxClassic d=4
◆ ReflexAgent	0.0174	0.2601	0.0059	0.125	0.0091	0	0
■ Minimax	1.02	12.1492	0.0176	2.7206	0.0823	0.2752	1.0332
▲ AlphaBeta	1.4091	10.6918	0.01825	2.7901	0.0592	0.2455	0.826
✕ Expectimax	0.9371	20.5518	0.0163	3.4975	0.0644	0.329	0.8781

RelfexAgent takes the shortest amount of time as it is just a linear function and it doesn't look ahead unlike the other agents. AlphaBeta is the second fastest and this makes sense because it is pruning branches, as it should be doing. Minimax takes bronze and runs slightly longer than AlphaBeta because it is running through branches that were not taken in AlphaBeta. Expectimax takes almost the same time for most tests but it spikes up during mediumClassic, presumably because it couldn't find a solution. All tests were conducted on depth 2, except for minimaxClassic layout which also takes into account depth 3 and 4. Another thing that is apparent in this graph is that the runtime takes longer as the depth increases.

2.4 Win Rate



	testClassic	mediumClassic	trappedClassic	smallClassic	minimaxClassic d=2	minimaxClassic d=3	minimaxClassic d=4
ReflexAgent	1	0.3	0.25	0.4	0.55	0	0
Minimax	1	0	0.55	0.05	0.5	0.45	0.55
AlphaBeta	1	0	0.65	0	0.9	0.3	0.5
Expectimax	1	0	0.5	0	0.6	0.45	0.9

The win-rate for testClassic for all agents is 100% due to the nature of the layout (extremely small and designed to have a 100% win-rate to test the agents). From other tests, we can see that the AI algorithms do poorly against larger maps such as mediumClassic. In smaller, more confined maps such as minimaxClassic, the agents are able to win more frequently. We see, particularly from minimaxClassic, that increasing the depth of the search increases the win-rate of the agents as this improves the accuracy of the values returned by minimax and expectimax. As a note, testClassic – smallClassic tests were conducted on the default depth.

To computer the win rate we ran 20 test cases per layout/depth. As visualized by the graph above, ReflexAgent, which is our linear evaluation function, was good enough to have the pacman win in simple layouts, even in mediumClassic where none of the other algorithms were able to make the pacman win. As we increased the complexity of the layouts, the depth became more and more important as it was necessary to look ahead what moves the pacman could make. The caveat of Minimax, AlphaBeta, and Expectimax algorithms is that with increasing depth the time complexity also increases. We saw a tremendous increase in win rate for Expectimax in the minmaxClassic layout due to the fact that that algorithms plays by the rules of how the ghosts attack pacman. Since we know that the ghosts are not always going to make the most optimal move, Expectimax takes this fact into account and uses probabilities to choose a path that perhaps is not the most optimal.

3 Contribution And Conclusion

Jessica: There are two contributors to this assignment: Jessica Ng and Gurkirat Singh. Responsibilities were split evenly across team members through paired programming. The write-up and testing were done via similar methods.

Some interesting things we learned about were AlphaBetaAgent's tendency to lose in contrast to ExpectimaxAgent's winning capability in a trapped map because the 'optimal move' based on the opponents' 'optimal move' may not result in the best choice of moves. ExpectimaxAgent's moves were more 'risky' but yielded better results in general. Additionally, trying to create an evaluation function was a good learning experience as it was rather difficult and time-consuming to create a function that was logical and produced good scores.

Gurkirat: I contributed in writing the evaluation function, which was already good enough for us to modify a little and make it better for Question 5. I further wrote few of the algorithms, tested for their accuracy, and as well helped make the graphs and write their analysis for this report. What I learned from this assignment is that Minimax, AlphaBeta, and Expectimax are all good algorithms that increase pacman's performace, however they are not good enough for the pacman to win every single time despite its environment. You have to make a tradeoff between time complexity of how good your win algorithm is and actually winning the game. It just goes to show how complex it is to program a machine to handle 100% success rate.