

Homework 3: Multi-Agent Search

Part I. Implementation (5%):

Minimax:

```
109 class MinimaxAgent(MultiAgentSearchAgent):
110     """
111     Your minimax agent (Part 1)
112     """
113
114     def getAction(self, gameState):
115         """
116         Returns the minimax action from the current gameState using self.depth
117         and self.evaluationFunction.
118
119         Here are some method calls that might be useful when implementing minimax.
120
121         gameState.getLegalActions(agentIndex):
122             Returns a list of legal actions for an agent
123             agentIndex=0 means Pacman, ghosts are ≥ 1
124
125         gameState.getNextState(agentIndex, action):
126             Returns the child game state after an agent takes an action
127
128         gameState.getNumAgents():
129             Returns the total number of agents in the game
130
131         gameState.isWin():
132             Returns whether or not the game state is a winning state
133
134         gameState.isLose():
135             Returns whether or not the game state is a losing state
136         """
137         # Begin your code (Part 1)
138         def minimax(gameState, agent, depth):
139             if gameState.isWin() or gameState.isLose() or depth == self.depth:
140                 return self.evaluationFunction(gameState), Directions.STOP
141             ...
142             Check if the game is over.
143             ...
144             if agent == 0: # pacman (max)
145                 return max_value(gameState, agent, depth)
146             else: # ghosts (min)
147                 return min_value(gameState, agent, depth)
```

```

148     """
149     Pacman's index is always 0, which means that we need to evaluate maximum value. On the
150     contrary, indexes that are larger than 0 represent ghosts, so we need to evaluate minimum
151     value instead.
152     """
153     """
154     General:
155     initialize v =  $-\infty$ 
156     for each a in Actions(s):
157         v = max(v, value(T(s,a)))
158     return v
159     """
160     def max_value(gameState, agent, depth):
161         maxScore = -1e9
162         bestAction = NULL
163         actions = gameState.getLegalActions(agent)
164         """
165         Find all possible moves
166         """
167         for action in actions:
168             nextState = gameState.getNextState(agent, action)
169             nextAgent = agent + 1
170             nextDepth = depth
171             if nextAgent == gameState.getNumAgents(): # last ghost
172                 nextAgent = 0
173                 nextDepth += 1
174             """
175             Multiagents, so we need to take turns to take action. When the successor's index equals
176             to the number of agents, it means that the successor is the last ghost, and a whole
177             round is finished. As a result, set the successor's index to 0, and add 1 to the depth.
178             """
179             curScore = minimax(nextState, nextAgent, nextDepth)[0] # score
180             if curScore > maxScore:
181                 maxScore = curScore
182                 bestAction = action
183             """
184             Save the max value for every loop, and also update the best decision move for current
185             agent
186             """
187
188         return maxScore, bestAction

```

```

190     """
191     General:
192     initialize v =  $+\infty$ 
193     for each a in Actions(s):
194         v = min(v, value(T(s,a)))
195     return v
196     """
197     def min_value(gameState, agent, depth):
198         minScore = 1e9
199         bestAction = NULL
200         actions = gameState.getLegalActions(agent)
201         for action in actions:
202             nextState = gameState.getNextState(agent, action)
203             nextAgent = agent + 1
204             nextDepth = depth
205             if nextAgent == gameState.getNumAgents(): # last ghost
206                 nextAgent = 0
207                 nextDepth += 1
208
209             curScore = minimax(nextState, nextAgent, nextDepth)[0] # score
210             if curScore < minScore:
211                 minScore = curScore
212                 bestAction = action
213             """
214             Almost the same as the max_value(), the only difference is that we save the min value
215             for every loop. (and the initial value is +inf, while the initial value of max_value()
216             is -inf
217             """
218
219         return minScore, bestAction

```

```

221     """
222     Return the best move decided. As minimax() would return a tuple of [score, action], we only
223     need action, which is minimax()[1] for this function's return.
224     """
225     move = minimax(gameState, 0, 0)[1]
226     # [score, action]
227     return move
228
229     raise NotImplementedError("To be implemented")
230     # End your code (Part 1)

```

AlphaBeta:

```

234 class AlphaBetaAgent(MultiAgentSearchAgent):
235     """
236     Your minimax agent with alpha-beta pruning (Part 2)
237     """
238
239     def getAction(self, gameState):
240         """
241         Returns the minimax action using self.depth and self.evaluationFunction
242         """
243         # Begin your code (Part 2)
244         """
245         Same as minimax
246         """
247         def AlphaBeta(gameState, agent, depth, alpha, beta):
248             if gameState.isWin() or gameState.isLose() or depth == self.depth:
249                 return self.evaluationFunction(gameState), Directions.STOP
250
251             if agent == 0: # pacman (max)
252                 return max_value(gameState, agent, depth, alpha, beta)
253             else: # ghosts (min)
254                 return min_value(gameState, agent, depth, alpha, beta)
255
256         """

```

```

256     """
257     def max_value(state,  $\alpha$ ,  $\beta$ ):
258         initialize v =  $-\infty$ 
259         for each successor of state:
260             v = max(v, value(successor,  $\alpha$ ,  $\beta$ ))
261             if v  $\geq \beta$ :
262                 return v
263              $\alpha$  = max( $\alpha$ , v)
264         return v
265     """
266     def max_value(gameState, agent, depth, alpha, beta):
267         maxScore = -1e9
268         bestAction = NULL
269         actions = gameState.getLegalActions(agent)
270         for action in actions:
271             nextState = gameState.getNextState(agent, action)
272             nextAgent = agent + 1
273             nextDepth = depth
274             if nextAgent == gameState.getNumAgents(): # last ghost
275                 nextAgent = 0
276                 nextDepth += 1
277
278             curScore = AlphaBeta(nextState, nextAgent, nextDepth, alpha, beta)[0] # score
279             if curScore > maxScore:
280                 maxScore = curScore
281                 bestAction = action
282             alpha = max(alpha, maxScore)
283             if maxScore > beta:
284                 break
285             ...
286
287         Decide whether we should prune it or not. If the current value(score) is greater than(or equal to) beta, then it means
288         that there is no need to keep traversing the remaining successors. Because if the condition holds, the next minimum value
289         (min node's turn) has already been determined, and no matter how large (or small) the next successor's value is, that
290         value won't be chosen as the final value, which allows us to prune it.
291         ...
292         return maxScore, bestAction

```

```

294     ...
295     def min_value(state,  $\alpha$ ,  $\beta$ ):
296         initialize v =  $+\infty$ 
297         for each successor of state:
298             v = min(v, value(successor,  $\alpha$ ,  $\beta$ ))
299             if v  $\leq \alpha$ :
300                 return v
301              $\beta$  = min( $\beta$ , v)
302         return v
303     ...
304     def min_value(gameState, agent, depth, alpha, beta):
305         minScore = 1e9
306         bestAction = NULL
307         actions = gameState.getLegalActions(agent)
308         for action in actions:
309             nextState = gameState.getNextState(agent, action)
310             nextAgent = agent + 1
311             nextDepth = depth
312             if nextAgent == gameState.getNumAgents(): # last ghost
313                 nextAgent = 0
314                 nextDepth += 1
315
316             curScore = AlphaBeta(nextState, nextAgent, nextDepth, alpha, beta)[0] # score
317             if curScore < minScore:
318                 minScore = curScore
319                 bestAction = action
320             beta = min(beta, minScore)
321             if minScore < alpha:
322                 break
323             ...
324             Similar to max_value(), but this time we use alpha to check if we can prune the nodes. Since the next successor of the
325             min node is max node, any value that is smaller than alpha would not be chosen as the final value, so the remaining part
326             can be pruned.
327             ...
328         return minScore, bestAction

```

```

330     ...
331     Same as minimax, but need to assign an initial value of alpha and beta
332     ...
333     move = AlphaBeta(gameState, 0, 0, -1e10, 1e10)[1]
334     # alpha beta
335     # [score, action]
336     return move
337     raise NotImplementedError("To be implemented")
338     # End your code (Part 2)

```

Expectimax:

```

341     class ExpectimaxAgent(MultiAgentSearchAgent):
342         """
343         Your expectimax agent (Part 3)
344         """
345
346         def getAction(self, gameState):
347             """
348             Returns the expectimax action using self.depth and self.evaluationFunction
349
350             All ghosts should be modeled as choosing uniformly at random from their
351             legal moves.
352             """
353             # Begin your code (Part 3)
354             ...
355             Same as minimax
356             ... (parameter) gameState: Any
357             def expectimax(gameState, agent, depth):
358                 if gameState.isWin() or gameState.isLose() or depth == self.depth:
359                     return self.evaluationFunction(gameState), None
360
361                 if agent == 0: # pacman
362                     return max_value(gameState, agent, depth)
363                 else: # ghosts
364                     return expected_value(gameState, agent, depth)

```

```

366     ...
367     The max_value() of expectimax is same as minimax, since the pacman would always do the best choice
368     ...
369     def max_value(gameState, agent, depth):
370         maxScore = -1e9
371         bestAction = NULL
372         actions = gameState.getLegalActions(agent)
373         for action in actions:
374             nextState = gameState.getNextState(agent, action)
375             nextAgent = agent + 1
376             nextDepth = depth
377             if nextAgent == gameState.getNumAgents(): # last ghost
378                 nextAgent = 0
379                 nextDepth += 1
380
381             curScore = expectimax(nextState, nextAgent, nextDepth)[0] # score
382             if curScore > maxScore:
383                 maxScore = curScore
384                 bestAction = action
385
386         return maxScore, bestAction

```

```

388     ...
389     This part is similar to min_value() in minimax. However, since the ghosts' actions are unknown, instead of assuming that they
390     would always do the worst (in terms of pacman) choice, we evaluate their expectation to predict the moves and deal with the
391     result.
392     ...
393     def expected_value(gameState, agent, depth):
394         expectedScore = 0
395         expectedAction = NULL
396         actions = gameState.getLegalActions(agent)
397         for action in actions:
398             nextState = gameState.getNextState(agent, action)
399             nextAgent = agent + 1
400             nextDepth = depth
401             if nextAgent == gameState.getNumAgents(): # last ghost
402                 nextAgent = 0
403                 nextDepth += 1
404
405             ...
406             Calculate the expectation of every move. In the fuction, I assume it is uniformly distributed.
407             ...
408             curScore = expectimax(nextState, nextAgent, nextDepth)[0] # score
409             expectedScore += curScore * (1 / len(actions))
410
411         return expectedScore, expectedAction
412
413     ...
414     Same as minimax
415     ...
416     move = expectimax(gameState, 0, 0)[1]
417     # [score, action]
418     return move
419
420     raise NotImplementedError("To be implemented")
421     # End your code (Part 3)

```

betterEvaluationFunction:

```
424 def betterEvaluationFunction(currentGameState):
425     """
426     Your extreme ghost-hunting, pellet-nabbing, food-gobbling, unstoppable
427     evaluation function (Part 4).
428     """
429     # Begin your code (Part 4)
430     """
431     Get all the remaining food's distance from the pacman, and find the smallest one.
432     """
433     curPos = currentGameState.getPacmanPosition()
434     foods = currentGameState.getFood()
435     foodList = foods.asList()
436     foodNum = len(foodList)
437     minDist = 1e9
438     for foodPos in foodList:
439         dist = util.manhattanDistance(curPos, foodPos)
440         if minDist > dist:
441             minDist = dist
442     """
443     """
444     Get the ghost's distance from the pacman, and if the distance is too close, escape.
445     """
446     ghostPos = currentGameState.getGhostPositions()
447     ghostDist = 0
448     for ghost in ghostPos:
449         dist = util.manhattanDistance(curPos, ghost)
450         ghostDist += dist
451
452     """
453     Since capsules' score is very high, eating them all will lead to a higher score. Meanwhile, eating the ghost also
454     gives us high score, so I add a 'hunt mode'. After the pacman eat the capsule, the pacman would turn into hunting
455     mode. It would try to chase the ghost and eat it.
456     """
457     capsule = currentGameState.getCapsules()
458     capsuleNum = len(capsule)
459     GhostStates = currentGameState.getGhostStates()
460     hunt = False
461     for state in GhostStates:
462         if state.scaredTimer > 5:
463             hunt = True
464     """
465     """
466     I separate the final sum into two conditions: one is hunt mode, the other is normal mode. First, in normal mode,
467     I put the score, minimum food distance, ghost's distance, number of capsules and the number of remaining food
468     into consideration and calculate the total value. On the other hand, in hunt mode, the closer the ghost is, the
469     higher chance the pacman can eat it, so the weight of 1/ghostDist is +10 compared to the one(-1) in normal mode.
470     Also, in this mode, I lower the weight of the remaining food from 100 to 10, which means that the priority of
471     eating the ghost is higher than eating food.
472     """
473     score = currentGameState.getScore()
474     if hunt:
475         total = 1000 * score + 10 * (1 / minDist) + 10 * (1 / ghostDist) - 500 * capsuleNum - 200 * foodNum
476     else:
477         total = 1000 * score + 100 * (1 / minDist) - 1 * (1 / ghostDist) - 500 * capsuleNum - 200 * foodNum
478     return total
479
480     raise NotImplementedError("To be implemented")
481     # End your code (Part 4)
```

Part II. Results & Analysis (5%):

```
Provisional grades
=====
Question part1: 20/20
Question part2: 25/25
Question part3: 25/25
Question part4: 10/10
-----
Total: 80/80
```

ALL HAIL GRANDPAC.
LONG LIVE THE GHOSTBUSTING KING.

[illegible]