Adversarial Search

Sai Srinadhu Katta & Venkata Sainath Thota

1 Introduction

In this project we designed agents for the classic version of Pacman, including ghosts and along the way implemented minimax and expectimax search and tried hand at evaluation function design in the Multi-Agent Pacman. In the local search part we tried to maximize the Revenue obtained by the government(for bidding the coal blocks) by using Hill Climbing Algorithm with some modifications.

2 Multi-Agent Pacman

2.1 Reflex Agent

In this part evaluation function is written for Reflex Agent, it's written for evaluating state-action pairs. Initialize the state_score to 0, Given a state and action following features are taken:

1. If the new Pacman position has food increase the state_score.
2. Calculate iteratively minimum distance of food from Pacman position and keep adding inverse of it to state_score and replacing Pacman position with nearest food till exhausting food list.
3. If minimum ghost distance is 1 then return a large -ve value as state_score.
4. Add the score of game state to state_score.

```python

    def evaluationFunction(self, currentGameState, action):

        score = 0
        #If it's a food great which is really important here.
        if (currentGameState.getFood()[newPos[0]][newPos[1]]):
            score +=1
        #calculate all the distance to food less it is it's better.
        current_food = newPos
        for food in newFood:
            #return the nearest_food using the below line.
            nearest_food = min(newFood, key=lambda x: manhattanDistance(x, current_food))
            score += 1.0/manhattanDistance(nearest_food, current_food)
```

```
            newFood.remove(nearest_food)
            current_food = nearest_food
        #ghost distances if less than some basic return large negative value
        if ( min([manhattanDistance(newPos, ghost.getPosition()) for ghost in newGhostStates]) == 1):
            return -1000
        #add the Score as well.
        score += successorGameState.getScore()

        return score
```

Code Snippet for Evaluation Function of Reflex Agent.

It fared well with 1 ghost and with 2 ghosts it won some games and lost some.

2.2 Minimax

In this task Minimax was implemented with a small change to standard Minimax that here more than one min player/agents are there. Here depth 'D' search will involve Pacman and all Ghosts taking 'D' steps. The standard pseudocode with depth is implemented here.

```
def getAction(self, gameState):
        """Getting the action"""

        pacman_legal_actions = gameState.getLegalActions(0) #all the legal actions of pacman.
        max_value = float('-inf')
        max_action  = None #one to be returned at the end.

        for action in pacman_legal_actions:   #get the max value from all of it's successors.
            action_value = self.Min_Value(gameState.generateSuccessor(0, action), 1, 0)
            if ((action_value) > max_value ): #take the max of all the children.
                max_value = action_value
                max_action = action

        return max_action #Returns the final action .


    def Max_Value (self, gameState, depth):
        """For the Max Player here Pacman"""

        if ((depth == self.depth)  or (len(gameState.getLegalActions(0)) == 0)):
            return self.evaluationFunction(gameState)

```

```
        return max([self.Min_Value(gameState.generateSuccessor(0, action), 1, depth) for action in gameState.getLegalActions(0)])


    def Min_Value (self, gameState, agentIndex, depth):
        """ For the MIN Players or Agents  """

        if (len(gameState.getLegalActions(agentIndex)) == 0): #No Legal actions.
            return self.evaluationFunction(gameState)

        if (agentIndex < gameState.getNumAgents() - 1):
            return min([self.Min_Value(gameState.generateSuccessor(agentIndex, action), agentIndex + 1, depth) for action in gameState.getLegalActions(agentIndex)])

        else:  #the last ghost HERE
            return min([self.Max_Value(gameState.generateSuccessor(agentIndex, action), depth + 1) for action in gameState.getLegalActions(agentIndex)])
```

Code Snippet for Minimax

In trappedClassic and by using minimax search it's rushing to the ghost because minimax believes death is inevitable and will end the game as soon as possible by rushing towards ghost for depth=3. If the depth is changed to 2 instead of 3 it wins sometimes based on ghost's moves.

2.3 Alpha-Beta Pruning

In this task Alpha-Beta Pruning was implemented with a small change that here more than one min player/agents are there. Here depth 'D' search will involve Pacman and all Ghosts taking 'D' steps. The standard pseudocode with depth is implemented here.

```
def getAction(self, gameState):
        """
          Returns the minimax action using self.depth and self.evaluationFunction
        """

        alpha = float('-inf') #max best option on path to root
        beta = float('inf') #min best option on path to root
        action_value = float ('-inf')
        max_action = None
        for action in gameState.getLegalActions(0):
            action_value = self.Min_Value(gameState.generateSuccessor(0, action), 1, 0, alpha, beta)
            if (alpha < action_value):
```

```
                alpha = action_value
                max_action = action
        return max_action

    def Min_Value (self, gameState, agentIndex, depth, alpha, beta):
        """ For Min agents best move """

        if (len(gameState.getLegalActions(agentIndex)) == 0): #No Legal actions.
            return self.evaluationFunction(gameState)

        action_value = float('inf')
        for action in gameState.getLegalActions(agentIndex):
            if (agentIndex < gameState.getNumAgents() - 1):
                action_value = min(action_value, self.Min_Value(gameState.generateSuccessor(agentIndex,action), agentIndex + 1, depth, alpha, beta))
            else:  #the last ghost HERE
                action_value = min(action_value, self.Max_Value(gameState.generateSuccessor(agentIndex, action), depth + 1, alpha, beta))

            if (action_value < alpha):
                return action_value
            beta = min(beta, action_value)

        return action_value

    def Max_Value (self, gameState, depth, alpha, beta):
        """For Max agents best move"""

        if (depth == self.depth or len(gameState.getLegalActions(0)) == 0):
            return self.evaluationFunction(gameState)

        action_value = float('-inf')
        for action in gameState.getLegalActions(0):
            action_value = max(action_value, self.Min_Value(gameState.generateSuccessor(0, action), 1, depth, alpha, beta))

            if (action_value > beta):
                return action_value
            alpha = max(alpha, action_value)

        return action_value
```

Code Snippet of Alpha-Beta Pruning.

2.4 Expectimax

In this task Expectimax was implemented with a small change that here more than one min player/agents are there. Here depth 'D' search will involve Pacman and all Ghosts taking 'D' steps. The standard pseudocode with depth is implemented here. There is not much change from minimax except in Min_Value function

```
    def Min_Value (self, gameState, agentIndex, depth):
        """ For the MIN Players or Agents """

        num_actions = len(gameState.getLegalActions(agentIndex))

        if (num_actions == 0): #No Legal actions.
            return self.evaluationFunction(gameState)

        if (agentIndex < gameState.getNumAgents() - 1):
            return sum([self.Min_Value(gameState.generateSuccessor(agentIndex, action), agentIndex + 1, depth) for action in gameState.getLegalActions(agentIndex)]) / float(num_actions)

        else:  #the last ghost HERE
            return sum([self.Max_Value(gameState.generateSuccessor(agentIndex, action), depth + 1) for action in gameState.getLegalActions(agentIndex)]) / float(num_actions)


```

Min_Value Function of Expectimax

In trappedClassic and for both searches fixing depth as 3, AlphaBeta agent always looses since it assumes worst always and Expectiminimax gives more options and if ghost moves go our way we can win.

2.5 Evaluation Function

In this task implemented better evaluation function which evaluates states as a whole. Initialize the state_score to 0, Given a state and action features are taken and in the below mentioned manner score is calculated for a state:

1. Calculate iteratively minimum distance of food from Pacman position and keep adding inverse of it to state_score and replacing Pacman position with nearest food till exhausting food list.
2. Do the above same for capsules as well. 3. If minimum ghost distance is 1 or less then return a large -ve value as state_score, otherwise subtract it's inverse from score.
4. Add eight times score of game state to state_score.
5. Subtract six times total food plus total capsules remaining.

```
def betterEvaluationFunction(currentGameState):
    """
      Better evaluation function for a state
    """

    state_score = 0 #initializing to zero.

    #Feature 1: distances from ghosts if exists
    if currentGameState.getNumAgents() > 1:
        ghost_dis = min( [manhattanDistance(Pacman_Pos, ghost.getPosition()) for ghost in GhostStates])
        if (ghost_dis <= 1):
            return -10000
        state_score -= 1.0/ghost_dis

    #Feature 2: food positions
    current_food = Pacman_Pos
    for food in food_list:
        closestFood = min(food_list, key=lambda x: manhattanDistance(x, current_food))
        state_score += 1.0/(manhattanDistance(current_food, closestFood))
        current_food = closestFood
        food_list.remove(closestFood)

    #Feature 3: capsule positions
    current_capsule = Pacman_Pos
    for capsule in capsule_list:
        closest_capsule = min(capsule_list, key=lambda x: manhattanDistance(x, current_capsule))
        state_score += 1.0/(manhattanDistance(current_capsule, closest_capsule))
        current_capsule = closest_capsule
        capsule_list.remove(closest_capsule)

    #Feature 4: Score of the game
    state_score += 8*(currentGameState.getScore())

    #Feature 5: remaining food and capsule
    state_score -= 6*(no_food + no_capsule)

    return state_score

```

Code Snippet from Evaluation Function.

Using this evaluation function it wins all games with an average score of 1072.6

3 Coal Block Auction

In this question we are asked to experiment with different different local search algorithms to find the maximum revenue obtained by the government in the auction of coal bids.

3.1 State Representation

The state of the problem is modelled as the allotment of bids to each company with their respective blocks.The data structure used is dynamic pointer array.

```
 for (int i=0;i<C;i++){ //for loop for all the companies

    int cid,ncid;
    fscanf(file1,"%d %d\n",&cid,&ncid);

    for(int j=0;j<ncid;j++){  //for loop for all the bids of that company

        int cid1,nbid,revenue;
        fscanf(file1,"\n%d %d %d ",&cid1,&nbid,&revenue);

        REV_TABLE[BID_COUNT][0]=cid1;
        REV_TABLE[BID_COUNT][1]=nbid;
        REV_TABLE[BID_COUNT][2]=revenue;
        BID_ARRAY[BID_COUNT]=(int *)malloc(nbid * sizeof(int));

        for(int k=0;k<nbid;k++){
            fscanf(file1,"%d ",&BID_ARRAY[BID_COUNT][k]);
        }

    BID_COUNT++;
   }
}
```

Code Snippet from inputing the bids and putting them in a datastructure.

3.2 Successor

The successor state is generated by removing the current allotted bid and then trying to allot a new valid bid i.e with different company and different blocks.The successor of each state is defined exactly as the states which is different from the current bid company and which asked for blocks that are different from the current blocks.

```
int CHILD[B][B];
for (int i=0;i<B;i++){
    int BLOCK[N];

    for(int l=0;l<N;l++){
```

```
        BLOCK[l]=0;
    }

    int c=REV_TABLE[i][1];

    for(int l=0;l<c;l++){
        int h=BID_ARRAY[i][l];
        BLOCK[h]=1;
    }

    for(int j=0;j<B;j++){
        int c=REV_TABLE[j][1];
        int a=1;
        for(int l=0;l<c;l++){
            int h=BID_ARRAY[j][l];
            if(BLOCK[h]==1){
                a=0;
                break;
            }
        }

        if(a==1 && REV_TABLE[i][0]!=REV_TABLE[j][0]){
            CHILD[i][j]=1;
        }
        else{
            CHILD[i][j]=0;
        }
    }
}
```

Code Snippet for creating the child.

3.3 Local Search

We have implemented stochastic hill climbing with some modifications and random restarts as our solution.

Stochastic hill climbing with some modifications chooses the best child(with highest revenue) with a probability if 0.5 and chooses some random successor state with a probability 0.5 when ever its depth is increased.If it has reached a dead end i.e no possible bids to add then its saves the result and restarts with different state.

```
while(p!=(B*B)){
        int prob=rand();
        if(prob%2==0 && S==0){
            state=MAX_CHILD[STATE];
            S=1;
```

```
            }
            else{
                if(K!=B){
                do{
                      state=rand();
                      state=state%B;
                }while(CHILD[STATE][state]==0);
                }
            }
             int COM_ID,BLOCK_COUNT,REVENUE1;
             COM_ID=REV_TABLE[state][0];
             BLOCK_COUNT=REV_TABLE[state][1];
             REVENUE1=REV_TABLE[state][2];

             if(BID_NUMBER[state]==0){

    if(COMPANY[COM_ID]==0){
      int check=1;
                         for(int s=0;s<BLOCK_COUNT;s++){
                            int l=BID_ARRAY[state][s];
                            if(BLOCK_TABLE[l]==1){
                                  check=0;
                                  break;
                               }
                          }
                         if(check==1){
                          REVENUE=REVENUE+REVENUE1;
                          for(int s=0;s<BLOCK_COUNT;s++){
                            int l=BID_ARRAY[state][s];
                           BLOCK_TABLE[l]=1;
                            }
                          COMPANY[COM_ID]=1;
                          BID_NUMBER[state]=1;
                          p=0;
                          STATE=state;
                          K=0;
                          S=0;
                         }
                         else{
                         K++;
                         }

                     }
             }

 p++;
 }             //end of while2
```

The main part of the hill climbing.

References

[1] UC Berkeley CS 188 Intro to AI – Course Materials,
http://ai.berkeley.edu/multiagent.html

[2] LaTeX Templates for Laboratory Reports,
https://github.com/mgius/calpoly_csc300_templates