



University of Isfahan
Pacman Project

Authors:

Melika Shirian

Kianoosh Vadaei

**Fundamentals and Applications of
Artificial Intelligence**

Dr. Hossein Karshenas

2024-01-06


Introduction

This project explores the realm of adversarial search in the context of the classic Pacman environment developed by UC Berkeley. Focused on strategic decision-making, the Minimax algorithm is employed, further enhanced by the efficiency of Alpha-Beta Pruning. In the competitive Pacman setting, where intelligent agents strive to optimize outcomes while outsmarting opponents, this project aims to unveil the nuanced dynamics of adversarial decision-making.

Code Explanation

The code of this project consists of two main sections: the Minimax algorithm and the **getAction** function, which is invoked by the **pacman.py** file during execution for each move of the Max agent.

Both of these functions are defined in the **AIAgent** class, which itself inherits from the **MultiAgentSearchAgent** class. The properties of the **AIAgent** class are as follows:




```
1 class MultiAgentSearchAgent(Agent):
2     def __init__(self, evalFn="scoreEvaluationFunction", depth="2", time_limit="6"):
3         self.index = 0 # Pacman is always agent index 0
4         self.evaluationFunction = util.lookup(evalFn, globals())
5         self.depth = int(depth)
6         self.time_limit = int(time_limit)
```

Figure 1 – MultiAgentSearchAgent Class Fields

First, we delve into the Minimax algorithm, implemented in the **minimax_alpha_beta** function.

This function is a recursive one, being called alternately for the Max agent and the Min agents. Essentially, this function constructs the Minimax game state tree, and scores are obtained from the leaf nodes.



```
1 def minimax_alpha_beta(self, game_state : GameState, depth, alpha, beta, agent_indx):
2
3     if depth == 0 or game_state.isWin() or game_state.isLose():
4         return self.evaluationFunction(game_state) , None
5
6     legal_actions = game_state.getLegalActions(agent_indx)
```

Figure 2 - minimax_alpha_beta Function - Part 1

At the initiation of the game, the arguments **game_state**, **depth**, **alpha**, **beta**, and the agent's index are passed to these functions.

The depth of exploration in the state space tree is determined by the user during program execution. At the beginning of the game, if the depth of the game reaches zero, indicating that the tree has been

fully traversed to its leaf nodes, or if the Max agent wins, or if the game state results in a loss, the corresponding score is returned. Please disregard the returned value for now; it will be explained further in the following explanation.

Afterward, for the agent currently positioned at the current level of the tree, all possible actions are stored in the variable **legal_actions**.



```

1  # index 0 is pacman and else are ghosts
2
3  #=====
4  #      max player
5  #=====
6
7
8  if agent_indx == 0: #pacman
9      action_scores = []
10     max_indices = []
11
12     max_eval = float('-inf')
13     for action in legal_actions:
14         successor_state = game_state.generateSuccessor(agent_indx, action)
15         eval, _ = self.minimax_alpha_beta(successor_state,
16                                           depth, alpha, beta,
17                                           (agent_indx + 1) % game_state.getNumAgents())
18         print(f'eval is: {eval}')
19         action_scores.append(eval)
20         max_eval = max(max_eval, eval)
21         alpha = max(alpha, max_eval)
22         if beta <= alpha:
23             break
24     for index in range(len(action_scores)):
25         if action_scores[index] == max_eval:
26             max_indices.append(index)
27     chosenIndex = random.choice(max_indices)
28     print(f'legal is : {legal_actions}\naction score is: {action_scores}')
29     return max_eval, legal_actions[chosenIndex]

```

Figure 3 - minimax_alpha_beta Function - Part 2

Here, index 0 corresponds to the Max agent, and indices greater than 0 belong to the Min agents.

Here, initially, two lists, **action_scores** and **max_indices**, are defined for the Max agent, and their explanations will follow. Then, for all possible actions for the current agent, the action is applied to the agent and the environment, and the result is stored in **successor_state**. Subsequently, the score obtained from applying

that action to the agent is passed to the **minimax_alpha_beta** function again through a recursive call in **eval**.

Note that in the recursive call to the **minimax_alpha_beta** function, the agent's index is incremented by one, and then the modulo operation is applied to get the new index within the total number of agents, including the Max agent. This is because, for example, if there were 3 ghosts in the game, given that the remainder of any number divided by 3 falls within the range [0, 2], the number of indices does not exceed 3. However, a different index is passed each time.

It's also noteworthy that when the **minimax_alpha_beta** function is recursively called in the Max agent's section, the depth of the tree does not decrease, and the same depth is passed again. The reason for this phenomenon will be explained later.

After the recursive call to the **minimax_alpha_beta** function, the final score obtained from the application of the new action on the agent is stored in the variable **eval**. Subsequently, this value is added to the **action_scores** list for later use. Additionally, the maximum score for each action is calculated and stored in the variable **max_eval**.

After that, attention turns to alpha-beta pruning. Here, for each action, alpha is set to the maximum of its current value and the maximum score obtained so far (**max_eval**). If beta is less than or equal to alpha, it means that further exploration of this node in the tree is not beneficial, and the loop should be terminated at this point. This is because we know that in the subsequent exploration of other actions, we will not find a higher score.

Finally, at this stage, the indices of all actions that lead to the maximum score are stored in the **max_indices** list. Afterward, one of these actions is randomly selected, along with the maximum score, and returned in the function.

The reason for only returning the selected action in this part of the function is that, ultimately, the last thing this function returns is this

value, and it doesn't create any issues in the algorithm. However, it's important to note that in other parts of the function where the function needs to return something, it should return a secondary value, like **None**, to avoid potential problems.

In the continuation, we see a visual representation of what happens in the alpha-beta pruning.

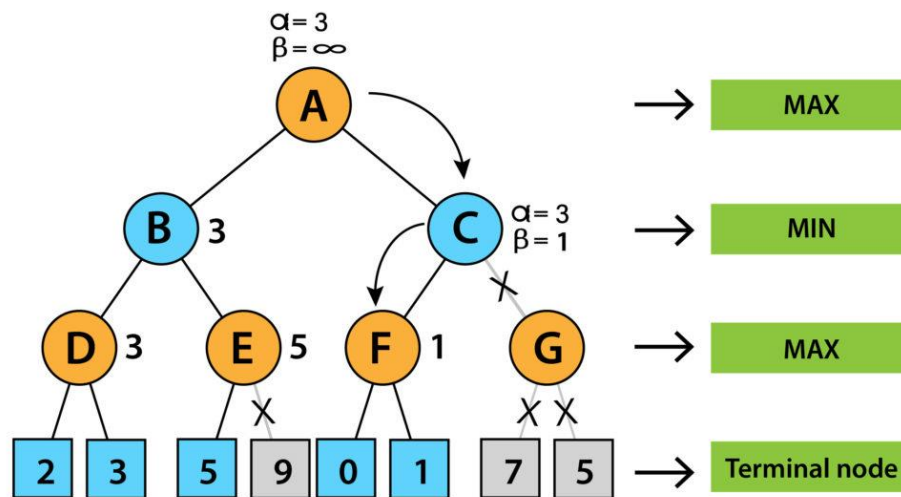


Figure 4 - MiniMax with Alpha-Beta Pruning

In the final section of the **minimax_alpha_beta** function, we move on to the Min agent. Here, the Max agent assumes that the Min agent or agents are choosing their best actions, effectively attempting to minimize the utility value.



```

1      #-----
2      #      min player
3      #-----
4
5      else:
6          minEval = float('inf')
7          if (agent_indx % game_state.getNumAgents()) == game_state.getNumAgents() - 1:
8              depth -= 1
9
10         for action in legal_actions:
11             successor_state = game_state.generateSuccessor(agent_indx, action)
12             eval , _ = self.minimax_alpha_beta(successor_state,
13                                                depth, alpha, beta,
14                                                (agent_indx + 1) % game_state.getNumAgents())
15             minEval = min(minEval, eval)
16             beta = min(beta, minEval)
17             if beta <= alpha:
18                 break # Alpha cut-off
19
20         return minEval , None

```

Figure 5 - minimax_alpha_beta Function - Part 3

This section of the code bears a significant resemblance to the Max section. Let's first address when the depth of the tree should be reduced.

In the minimax algorithm, the depth should only be decreased once for each Max agent and Min agents. To ensure this happens correctly throughout the program, we reduce the depth only for the last Min agent present. To achieve this, we calculate the remainder of the current agent's index divided by the total number of agents, including the Max agent. If this value is one less than the total number of agents, it means we are examining the last Min agent, and here is where the depth should be reduced by one.

Then, similar to the Max agent, for each possible action in the actions of the Min agent, the effect of applying that action on the Min agent and the environment is stored in **successor_state**. The **minimax_alpha_beta** function is then called with this new game state for the next agent (depending on whether the current agent's index, the next agent can be either Max or Min). Additionally, the

depth change is propagated based on the same current agent's index.

Then, for each action, the minimum scores are calculated, and alpha-beta pruning is applied. Here, instead of updating alpha, we update beta with the minimum score. If beta becomes less than alpha, the node is pruned, and there is no need to explore other actions.

Finally, the minimum score is returned.

Here, our minimax algorithm comes to an end. Now, it's sufficient to call this algorithm for action selection. The returned action can then be applied to the Pacman agent in the environment.

```
1 def getAction(self, gameState: GameState):
2
3     initialAlpha = float('-inf')
4     initialBeta = float('inf')
5
6     _, action = self.minimax_alpha_beta(gameState, self.depth, initialAlpha, initialBeta, 0)
7     print(f'action: {action}')
8     return action
```

Figure 6 – getAction Function

The initial values for alpha and beta should indeed be negative infinity and positive infinity, respectively. Then, the **minimax_alpha_beta** function is called for the Max agent (index 0), and the selected action is returned.



Usage

To use the environment, first clone the GitHub repository of this project from [this link](#).

Script Usage:

1. Navigate to the directory **multi-agent-search-pandas\Code\multiagent** using the terminal:

```
cd path/to/downloaded/package
```

2. Run the following script to play the game:

```
python pacman.py -p AIAgent -k 1 -a depth=4
```

3. Run the following script to play against intelligent ghosts:

```
python pacman.py -p AIAgent -k 1 -a depth=4 -g DirectionalGhost
```

4. Run the following script to increase the number of ghosts:

```
python pacman.py -p AIAgent -k 2 -a depth=4
```

5. Run the following script for manual play mode:

```
python pacman.py -k 1
```

References

- Game Playing 1 - Minimax, Alpha-beta Pruning | Stanford CS221: AI (Autumn 2019). Retrieved from [this link](#).
- Multi-agent Pac-Man Stanford CS221 Spring 2018." Retrieved from [this link](#).
- OpenAI. "ChatGPT." <https://www.openai.com/>
- Russell, Stuart, and Norvig, Peter. "Artificial Intelligence: A Modern Approach." (Book)