9 Games: Adversarial Search

Until now, we considered the problems where there is a single agent focused on achieving the desired goal or optimizing utility. There are many other problems involving multiple agents, sometimes adversarial to one another. Let us discuss the problems with multiple agents which are adversarial to each other.

Let us consider the scenario of a two player game. In this game, there are 2 bags and there are 2 balls in each bag with a number on it. There are 2 players, one aiming to maximize the score, let us call him/her as MAX player, and the other one is aiming to minimize the score, again let us call him/her as MIN player. The MAX player has to chose a bag, while the MIN player has to choose one ball out of the 2 balls in the chosen bag. The score is determined by the number on the ball chosen.

Let the 2 bags be B^1 and B^2 and the 2 balls in bag B^1 be B_1^1 and B_1^2, and the 2 balls in bag B^2 be B_2^1 and B_2^2. The Figure 1 shows the possible outcomes of the game. The red terminal in the Figure 1 represents the balls, and the number on them.

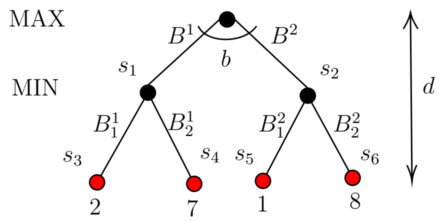

Figure 1: Example to Max-Min 2 player game.

As per Figure 1, s_0, the root is the start state of this game, and s_3, s_4, s_5, s_6 as indicated by the red nodes are the terminal states where the score of the game is calculated. In addition, we have the 2 parameters: depth d and branching factor b.

Let utility function be $Utility(s)$ where s is a terminal state, i.e., for example, $Utility(s_3) = 2$. The purpose of MAX player is to maximise the utility, while the purpose of MIN player is to minimize the utility.

An intuitive strategy that both players can adopt is to assume that the opponent would play optimally. That is, MAX player would always choose state with the maximum utility, and the MIN player would always choose the minimum utility. Let our transition model function be $\mathsf{Result}(s, a)$. $\mathsf{Result}(s, a)$ considers a state s and an action a taken as input to return the next state.

We can define the value of state, denoted by $\mathsf{MiniMax}(s)$, as follows:

$$\mathsf{MiniMax}(s) = \begin{cases} UTILITY(s) & \text{if s is terminal state} \\ \\ \max\limits_{a \in A} \mathsf{MiniMax}(\mathsf{Result}(s, a)) & \text{if MAX player} \\ \\ \min\limits_{a \in A} \mathsf{MiniMax}(\mathsf{Result}(s, a)) & \text{if MIN player} \end{cases}$$

We will now trace the $\mathsf{MiniMax}$ computation for the game showed in Figure 1. The trace is shown in Figure 2 and 2, in which we circle the explored nodes.

- Max player will first explore the s_3, and $Utility(s_3) = 2$, so at this point, we know that utility at state s_1 would be less than or equal to 2.

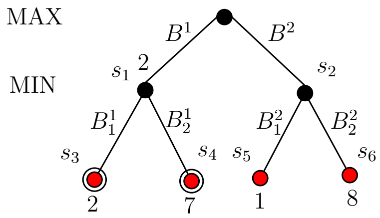

Figure 2: Example to Max-Min 2 player game; Nodes s_3 and s_4 explored.

- As, Max player does not know the utility at s_1, it will explore the node s_4. The utility of node s_4 is 7, i.e., $Utility(s_4) = 7$. As the utility at s_1 will be decided by the Min player and it will always optimally choose the state with minimum utility, Max player knows that the utility at s_1 is 2. Refer to Figure 2.
- Next, the Max player will now check for s_5. $Utility(s_5) = 1$, similarly the $Utility(s_6) = 8$. As the utility at s_2 will be decided by the Min player and it will always optimally choose the state with minimum utility, Max player knows that the utility at s_2 is 1.
- Finally Max player will choose s_1 over s_2 to maximize the utility at s_0.

Hence, Max player should choose B_1 in order to win the game.

9.1 Minimax Algorithm

As discussed above, the game can easily be solved by a recursive computation of the minimax values each node in the decision tree, working from the leaves to the root as the recursion unwinds. The algorithm is as follows assuming the MAX player starts first.

Algorithm 1 MinimaxDecision(s)

$bestAction \leftarrow null$
$max \leftarrow -\infty$
for all action a in Actions(s) **do**
  $util \leftarrow$MinValue(Result(s, a))
  if $util > max$ **then**
    $bestAction \leftarrow a$
    $max \leftarrow util$
return $bestAction$

Algorithm 2 MinValue(s)

if IsTerminal(s)=True **then return** UTILITY(s)
$v \leftarrow \infty$
for all action a in Actions(s) **do**
  $v \leftarrow \min(v,$ MaxValue(Result(s, a)))
return v

Algorithm 3 MaxValue(s)

```
if IsTerminal(s)=True then return UTILITY(s)
v ← −∞
for all action a in Actions(s) do
    v ←max(v, MinValue(Result(s,a)))
return v
```

The MinimaxDecision function can be modified to produce a set of optimal rational strategy for the game. A strategy is defined as a mapping of states in the game to an action or a series of actions. In other words, Strategy: $States \mapsto Actions$

9.1.1 Evaluation Time/Space Complexity of Minimax Algorithm

The *Minimax* algorithm implements a complete depth-first exploration of the game tree. The functions *MAXVALUE* and *MINVALUE* traverses the entire decision tree, down to each of the leaves to evaluate the backed-up value of a state represented as a branch node by backward induction. Assuming each game state represented by a node has a branching factor b and the game tree has a maximum depth of m, the number of terminal states to evaluate would be b^m, which means the *Minimax* algorithm has a time complexity of $O(b^m)$. This is inefficient in practice. The space complexity of the *Minimax* algorithm is $O(bm)$ if it generates the entire game tree at once, and $O(m)$ if it generates the actions of the game tree one at a time.

9.2 Alpha-Beta Pruning

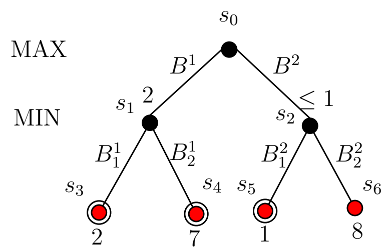

Figure 3: Example to Max-Min 2 player game; Nodes s_3, s_4, s_5 explored.

As shown in Figure 3, we do not need to compute the exact utility at the node s_2 in order to decide the strategy for Max player, just the information that utility at s_2 is less than or equal to 1 is sufficient to choose the optimal strategy — that is, we can prune off the branches that can not improve the outcome of the game for a player. Next, we move on to an alternative algorithm, Alpha-Beta Pruning, for the two player game that would optimize the *Minimax* algorithm, by 'pruning' off branches.

The alpha(α) and beta(β) values are to be interpreted as such:

- α : Best utility (highest value) from MAX Player's perspective
- β : Best utility (lowest value) from MIN Player's perspective

Algorithm 4 α-β-MinimaxDecision(S)

$bestAction \leftarrow null$
$max \leftarrow -\infty$
for all action a in Actions(s) **do**
  $util \leftarrow \alpha$-β-MinValue(Result(s, a),max, ∞)
  if $util > max$ **then**
    $bestAction \leftarrow a$
    $max \leftarrow util$
return $bestAction$

Algorithm 5 α-β-MinValue(s,α, β)

if IsTerminal(s)=True **then return** UTILITY(S)
$v \leftarrow \infty$
for all action a in Actions(s) **do**
  $v \leftarrow$min(v, α-β-MaxValue(Result(s, a),α, β))
  if $v \leq \alpha$ **then return** v
  $\beta \leftarrow$min(v, β)
return v

Algorithm 6 α-β-MaxValue(s,α, β)

if IsTerminal(s)=True **then return** UTILITY(s)
$v \leftarrow -\infty$
for all action a in Actions(s) **do**
  $v \leftarrow$max(v, α-β-MinValue(Result(s, a),α,β))
  if $v \geq \beta$ **then return** v ▷ Pruning of branch
  $\alpha \leftarrow$max(α, v)
return v

9.2.1 An Example for Alpha-Beta Pruning

Let us consider the game tree in Figure 4, where MAX player begins first. We will now trace the algorithm's logic as it processes the game tree, as well as how the algorithm avoids having to evaluate some terminal states.

The game tree in Figure 5 is annotated to reflect the logic of the Alpha-Beta Pruning.

The algorithm evaluates the nodes from left to right along the game tree recursively. The logic flows in the following order, denoted by the increasing numbers on the white boxes in Figure 5:

- **Box 1:** The algorithm recursively traverses down from root node S_0 to S_{15} to evaluate the first terminal state, which results in a utility of 8
- **Box 1.1:** The MIN player at S_7, and they have the information that the terminal state at S_{15} will have the utility of 8, that is, the utility at (S_7) is ≤ 8, because they could still potentially choose S_{16} which might have a lower utility.
- **Box 2:** The algorithm evaluates the next terminal state S_{16} to find a utility of 7.

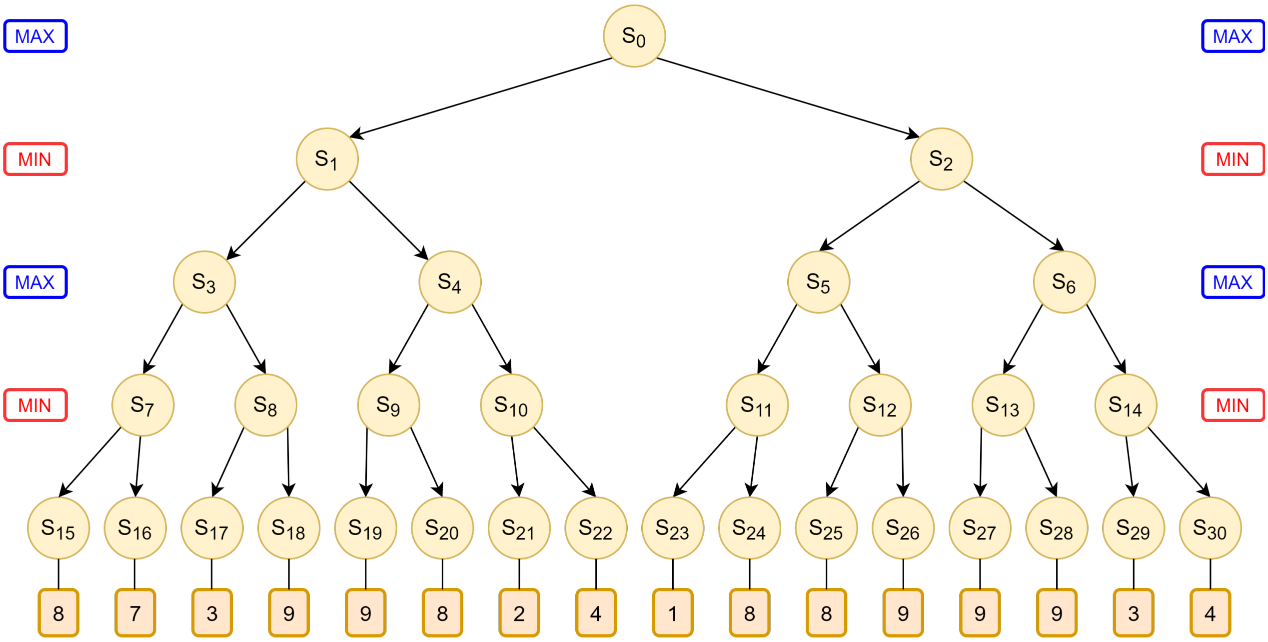

Figure 4: Game tree where MAX Player starts first

- **Box 2.1:** For the MIN player at S_7. As utility of S_{16} is less than the utility of S_{15}, the player will definitely choose S_{16}, and thus, the utility at S_7 would be 7.
- **Box 2.2:** The MAX player at S_3, knowing that choosing S_7 will result in a utility of 7, that is, they know that utility at s_3 is ≥ 7. Still, they can potentially choose S_8 that could have a higher utility value.
- **Box 3:** The algorithm evaluates the next terminal state S_{17} to find a utility of 3.
- **Box 3.1:** The MIN player at S_8, now knows that utility at s_8 is ≤ 3. Still, they can potentially choose S_{18} which could have a lower utility.
- **Box 3.2:** The MAX player at S_3 now have two information: (i) utility at S_7 is 7, (ii) utility at S_8 is ≤ 3. Thus, the player will choose S_7, and the entire branch to S_8 is pruned off, without having to evaluate the remaining nodes. We do not need the exact utility of S_8 in order to find the utility at node S_3.
- **Box 3.3:** With only S_7 remaining, utility at S_3 is 7.
- **Box 3.4:** The MIN player at S_1 has an information that choosing S_3 will result in a utility of 7. That is, utility at S_1 is ≤ 7, still they can potentially choose S_4 which could have a lower utility.
- **Box 4:** The algorithm evaluates the next terminal state S_{19} to find a utility of 9.
- **Box 4.1:** The MIN player at S_9, and utility at S_9 is ≤ 9, they can potentially choose S_{20} which could have a lower utility.
- **Box 5:** The algorithm evaluates the next terminal state S_{20} to find a utility of 8.

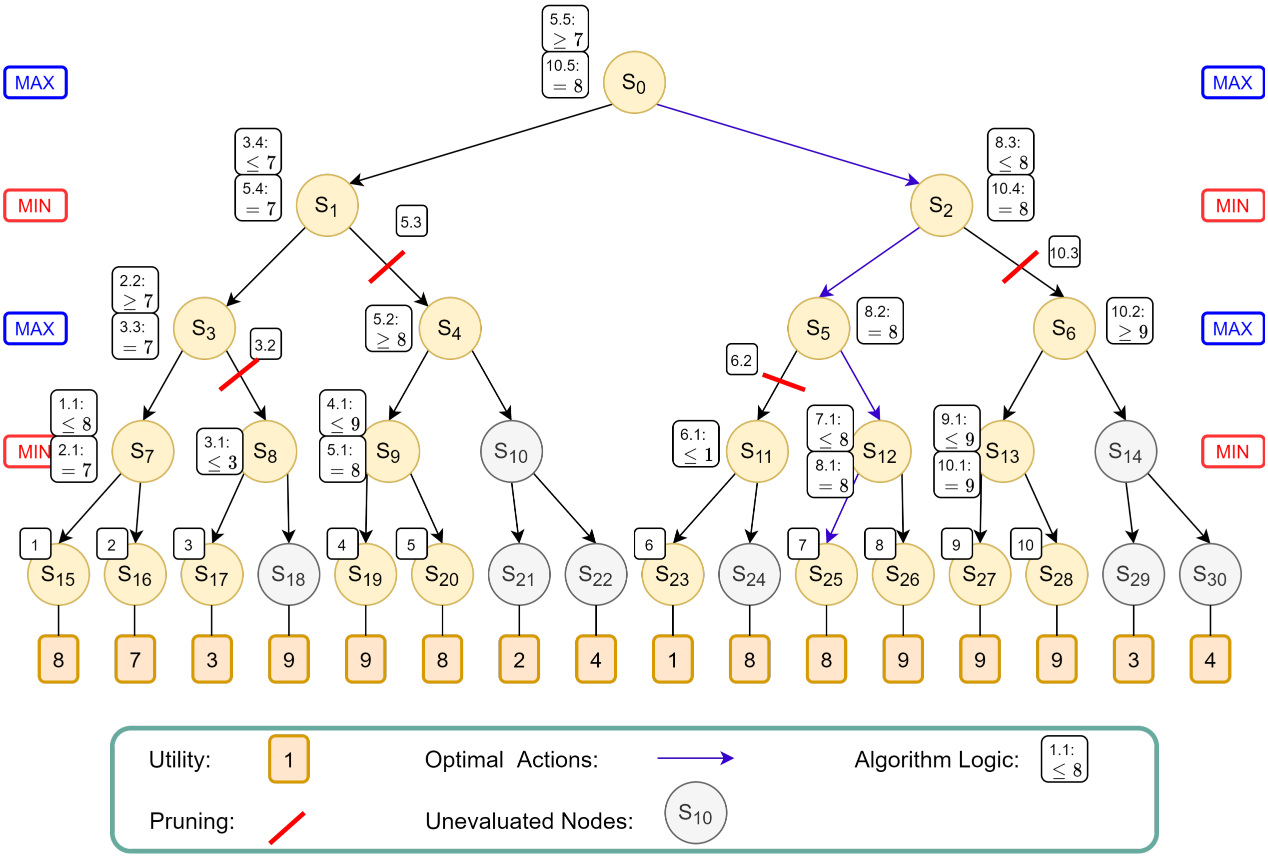

Figure 5: Game tree annotated with Alpha-Beta Pruning Logic

- **Box 5.1:** The MIN player at S_9 would choose S_{20} which gives a lower utility of 8 compared to S_{19} which gives a utility of 9. Utility at S_9 is 8.
- **Box 5.2:** The MAX player at S_4, knowing that choosing S_9 results in a utility of 8. As utility at S_4 is ≥ 8, they could potentially choose S_{10} which might have a higher utility.
- **Box 5.3:** The MIN player at S_1 have two information: (i) utility at S_3 is 7 (ii) utility at S_4 is ≥ 8. Therefore, MIN player would pick S_3 for utility 7, which is better that picking S_4 which grants the utility of at least 8. Therefore, S_4 will not be picked and the entire branch of S_4 is pruned.
- **Box 5.4:** The MIN player at S_1 will therefore pick S_3. The utility of S_1 is 7.
- **Box 5.5:** The MAX player at S_0 knows that picking S_1 results in a utility of 7, that is, the utility at S_0 is ≥ 7. They could potentially pick S_2 which might have a higher utility.
- **Box 6:** The algorithm evaluates the next terminal state S_{23} to find a utility of 1.
- **Box 6.1:** The MIN player at S_{11} knows that picking S_{23} results in utility 1, i.e, the utility at S_{11} is ≤ 1, they could potentially choose S_{24} which might yield a lower utility.
- **Box 6.2:** The MAX player at S_5 knows that choosing S_{11} will result in a utility at most 1. The MAX player will never not choose S_{11} because that would result in utility at (S_2) is ≤ 1 and in

that case the MAX player at S_0 will always choose S_1 with utility 7 over S_2. Thus, the entire branch of S_{11} is pruned. The utility at is S_5 still uncertain as choosing S_{12} could still potentially yield a higher utility than 7.

- **Box 7:** The algorithm evaluates the next terminal state S_{25} to find a utility of 8.
- **Box 7.1:** The MIN player at $S_1 2$ knows that the utility at (S_{12}) is ≤ 8, they could potentially choose S_{26} which might have a lower utility.
- **Box 8:** The algorithm evaluates the next terminal state S_{26} to find a utility of 9.
- **Box 8.1:** The MIN player at S_{12} will choose S_{25} which provides a lower utility of 8. Thus, the utility at S_{12} is 8.
- **Box 8.2:** The MAX player at S_5 now knows that choosing S_{12} returns a utility of 8, which is higher than the the utility at S_1. Thus, the utility at S_5 is 8.
- **Box 8.3:** The MIN player at S_2 knows that choosing S_5 results a utility of 8, thus they know that the utility at S_2 is at most 8, the could potentially choose S_6 which might potential yield a lower utility.
- **Box 9:** The algorithm evaluates the next terminal state S_{27} to find a utility of 9.
- **Box 9.1:** The MIN player at S_{13} knows that choosing S_{27} results in a utility of 9, and they know that the utility at S_{13} is at most 9.
- **Box 10:** The algorithm evaluates the next terminal state S_{28} to find a utility of 9.
- **Box 10.1:** The MIN player at S_{13} knows that choosing either S_{27} or S_{28} results in a utility of 9, thus assigning a utility of 9 at S_{13}.
- **Box 10.2:** The MAX player at S_6 knows that choosing S_{13} results in a utility of 9, thus the utility at S_6 is at least 9, because they could potentially choose $S_1 4$ which might potentially yield a higher utility.
- **Box 10.3:** The MIN player at S_2 knows that choosing S_6 results in a utility of at least 9, which is worse than choosing S_5 which guarantees a lower utility of 8. Thus, the MIN player will not choose S_6, and the entire branch of S_2 is pruned.
- **Box 10.4:** The MIN player at S_2 will therefore choose S_5 to get a utility of 8. Thus S_2 is assigned an utility of 8.
- **Box 10.5:** The MAX player at S_0 knows that choosing S_2 will result in a utility of 8, which is higher than the utility of 7 by choosing S_1. Thus, the MAX player will choose S_2 to get a utility of 8 at S_0.

9.2.2 Evaluation of Time/Space Complexity of Alpha-Beta Pruning

One weakness of the Alpha-beta pruning algorithm is that it is sensitive to the order at which the terminal states are evaluated. If the algorithm happens to evaluate the worse successors first in all nodes, there would be no chance for optimization by pruning. Therefore in the worse case, the algorithm runs the same as the Minimax Search at $O(b^m)$ in time complexity. However, if the algorithm knows where to priorities the search of the best successor nodes first, it only need to examine $b^{m/2}$ nodes, and therefore improves its time complexity to $O(b^{m/2})$ in the best case. If the successors are evaluated randomly, the algorithm's time complexity becomes $O(b^{3m/4})$ on average. The space complexity is similar to that of the minimax algorithm.

9.3 Heuristic Function for Imperfect Decision-Making (Optional)

Because Alpha-beta pruning still has to explore the game tree until the terminal state, this makes its time complexity impractical in games with large depth such as chess. One strategy is to cut off the search earlier, and apply a heuristic evaluation function to each state at the cutoff, thus allowing for non-terminal nodes to become leaves of the smaller search tree.

A good heuristic evaluation function should have the following properties:

1. The evaluation function should evaluate terminal states in a similar preference order as the true utility function would on the terminal states.
2. Computation of the evaluation function should not take too long (ideally close to constant time complexity)
3. For non-terminal states, the evaluation function should choose states highly correlated with the chances of winning. The uncertainty is induced by computational limits.

For example in chess, an evaluation function the expected probability of a win based on a weighted linear function of certain features of the game. For example, the evaluation function can be,

$$EVAL(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s)$$

where w_i refers to the weights, while $f_i(s)$ refers to a computation of features in the game state, such as the number of a type of chess piece on the board. The evaluation function need not calculate the actual probability of winning, but the ordinal relation between states via the evaluation function should be similar to that of the expected probability of winning.

The Alpha-beta pruning algorithm could implement the cut-off and heuristic by modifying the terminal condition of the algorithm.

Algorithm 7 α-β-Cutoff-MinimaxDecision(s)

$bestAction \leftarrow null$
$max \leftarrow -\infty$
for all action a in Actions(s) **do**
  $util \leftarrow$ α-β-Cutoff-MinValue(Result(s, a), 1, max, ∞)
  if $util > max$ **then**
    $bestAction \leftarrow a$
    $max \leftarrow util$
return $bestAction$

Algorithm 8 MinValue(s, $depth$, α, β)

if ShouldCutOff($s, depth$)=True **then return** EVAL(s)
$v \leftarrow \infty$
for all action a in Actions(s) **do**
  $v \leftarrow$min(v, α-β-Cutoff-MaxValue(Result(s, a), $depth + 1$, α, β))
  if $v \leq \alpha$ **then return** v
  $\beta \leftarrow$min(v, β)
return v

Algorithm 9 MaxValue(s, $depth$, α, β)

if ShouldCutOff$(s, depth)$=True **then return** EVAL(s)
$v \leftarrow -\infty$
for all action a in Actions(s) **do**
    $v \leftarrow$max(v, α-β-Cutoff-MinValue(Result(s, a), $depth + 1$, α, β))
    if $v \geq \beta$ **then return** v ▷ Pruning of branch
    $\alpha \leftarrow$max(α, v)
return v

One way to control the amount of search is to set a depth limit d so that ShouldCutOff$(s, depth)$ returns true when depth reaches limit d or a terminal state is reached whichever is earlier. The depth limit can be determined by the deepest depth a computer can explore while just keeping to the time limit required to make a decision. With a cut-off, the time complexity improves to $O(b^{3d/4})$ on average where $d < m$.