CS 217: Artificial Intelligence and Machine Learning **Jan-Apr 2024**

Lecture 20: Sequential and Simultaneous Move Games

Lecturer: Swaprava Nath *Scribe(s): Kavya, Mahak, Mayank, Kanishk (SG39)*

Disclaimer: *These notes aggregate content from several texts and have not been subjected to the usual scrutiny deserved by formal publications. If you find errors, please bring to the notice of the Instructor.*

This lecture's notes illustrate some uses of various LaTeX macros. Take a look at this and imitate.

20.1 Introduction

Previous lecture we came across Backward Induction to help the agent decide making the optimal move. Although effective, it's not efficient. It requires a lot of computation for large games, e.g. Chess ($\sim 10^{40}$ nodes), Go ($\sim 10^{170}$) etc.

Hence this lecture we came across two Speedup Techniques to overcome this limitation:-

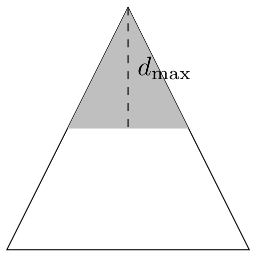
- Depth-Limited Search
- Pruning

20.2 Depth-Limited Search

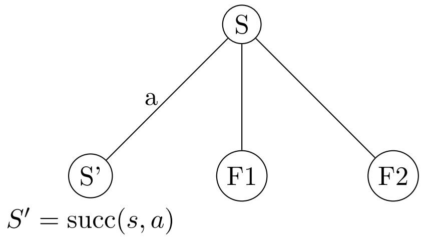
Depth-limited search (DLS) is an uninformed search algorithm similar to Depth-First Search. The objective of DLS is to traverse a search tree efficiently, constrained by a maximum depth parameter, $d_{\max}$. This parameter controls the exploration depth from the root before backtracking to explore other branches.

The algorithm starts from the root of the search tree. It then explores all the nodes up to a maximum depth $d_{\max}$ from the root nodes, after which it backtracks to previously unexplored nodes, continuing the search process. The choice of $d_{\max}$ depends largely on computational resources.

At every iteration of DLS, the algorithm checks if the current node is the goal node. The algorithm terminates successfully if the current node corresponds to the goal state. Otherwise, the algorithm backtracks to explore other branches within the search tree.



(a) d_{max} for a tree



(b) S' is the goal node

When the goal node is located on the right side of a deep tree, normal DFS may be inefficient as it will traverse deep left-side branches before exploring other branches, increasing the running time. In such cases, Depth-limited search helps reduce computational time by limiting the exploration depth. DLS can efficiently navigate towards the goal node without exhaustively traversing irrelevant branches.

The utility function in Game Search Tree when using Depth-limited search is defined as follows:

$$u_{\text{agent}}(s,d) = \begin{cases} \text{utility}(s) & \text{if isEnd}(s) \\ \text{eval}(s) & \text{if } d = 0 \\ \max\limits_{a \in \text{action}(s)} u_{\text{agent}}(\text{succ}(s,a), d-1) & \text{if player}(s) = \text{agent} \\ \min\limits_{a \in \text{action}(s)} u_{\text{agent}}(\text{succ}(s,a), d-1) & \text{if player}(s) = \text{opponent} \end{cases}$$

If the depth limit d is reached, the utility function evaluates the state using a domain-specific function eval(s), representing the potential utility to the agent at that state.

Consider the game of Chess, where the utility evaluation function eval(s) is defined as follows:

$$\text{eval}(s) = \text{army} + \text{mobility} + \text{king-safety}$$

The army component represents the overall strength of a player's position based on the relative worth of their pieces compared to the opponent's. Each piece is given a weight value.

$$\text{army} = 10^{100}(K - K') + 9(Q - Q') + 5(R - R') + \dots$$

Here, K, Q, R, etc., denote the number of kings, queens, rooks, and so on that the player has, while K', Q', R', etc., denote the respective counts for the opponent. The weights assigned to each piece signify the **strategic importance** in the game. For instance, the King has the most significance, denoted by the weight (10^{100}), showing its almost infinite worth due to the game's objective of protecting the king at all costs.

Mobility quantifies the player's ability to place their pieces effectively across the board. It is given as:

$$\text{mobility} = C \times \text{ number of (legal moves - legal moves}')$$

Here, C is a scaling factor.

Therefore, in DLS, the agent evaluates potential moves by exploring the game tree up to a specified depth $d_{\max}$. At any intermediate step, the agent chooses the move that maximizes its utility within the depth limit, similar to the strategic foresight techniques of grandmasters in chess.

Some disadvantages of Depth-Limited Search are:

- DLS is based on a heuristic approach to decision-making. Consequently, the solution obtained may not always be optimal and depends on the depth limit.
- The $d_{\max}$ needs to be recalculated for every sub-tree encountered during the search process. This introduces additional computational overhead.

20.3 Pruning (Alpha-Beta Pruning)

20.3.1 Intuition

Say $y < x$ in the game tree (Fig. 20.2). We read the tree from left to right, hence it reads x first then among children of "min" node, y is read first. The min-player will choose the minimum (say m) out of y, z_1 and z_2

and m should be $\leq y$. Hence the utility of the "min" node ($= m$) will also be $\leq y$, so the max-player will guaranteedly **not** take the "min" node decision. Therefore, we don't need to evaluate the z_1 and z_2 nodes and can directly backtrack to the parent. In other words, we *prune* the edges going to z_1 and z_2 (Fig. 20.3).

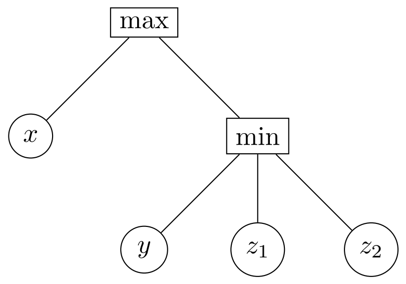


Figure 20.2: Before Pruning

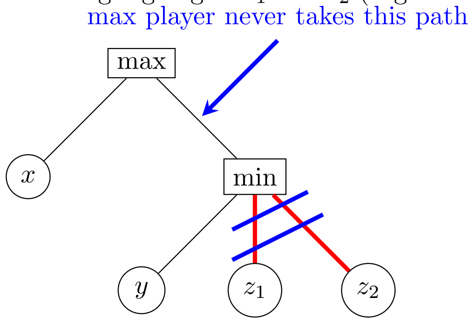


Figure 20.3: After Pruning

20.3.2 Algorithm

Let's see how the Pruning Algorithm goes. Here is a simple implementation of it. Later, we'll see an example, applying it.

Algorithm 1: Alpha-Beta Algorithm

```
Function alphabeta(node, depth, $\alpha$, $\beta$, maximizingPlayer):
    if depth == 0 or node is terminal then
        return heuristic value of node
    end
    if maximizingPlayer then
        value := $-\infty$
        for each child of node do
            value := $\max(value, \texttt{alphabeta}(child, depth - 1, \alpha, \beta, \text{FALSE}))$
            if value > $\beta$ then
                break /* $\beta$ cutoff */
            end
            $\alpha := \max(\alpha, \text{value})$
        end
        return value
    end
    else
        value := $+\infty$
        for each child of node do
            value := $\min(value, \texttt{alphabeta}(child, depth - 1, \alpha, \beta, \text{TRUE}))$
            if value < $\alpha$ then
                break /* $\alpha$ cutoff */
            end
            $\beta := \min(\beta, \text{value})$
        end
        return value
    end
```

Here α signifies lower-bound on the max-player's pick at each node and β signifies upper-bound on the min-player's pick at each node.

α, β are also updated by the following equations:

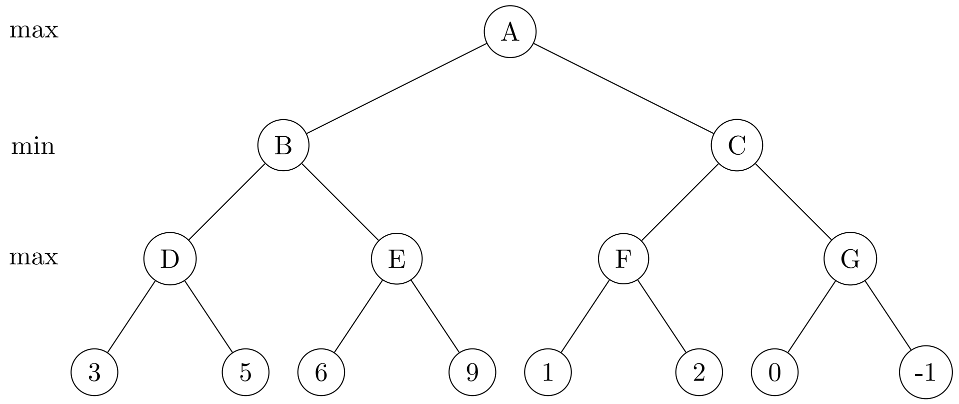
- $\alpha = \max(\alpha, \text{value\_encountered\_in\_max\_node})$
- $\beta = \min(\beta, \text{value\_encountered\_in\_min\_node})$

It stops evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. Such moves need not be evaluated further. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.

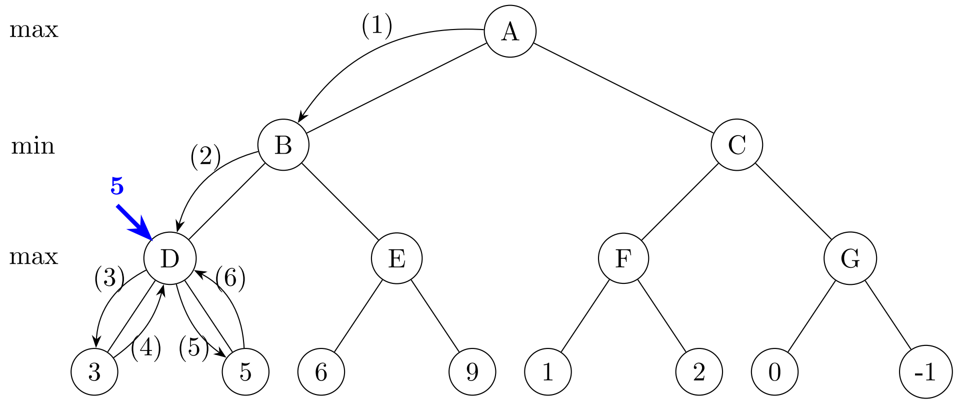
$\alpha \geq \beta$ is checked at every step. If found true, then it means that the **parent of current node won't choose this node** at all and hence checking other child nodes is useless and we should immediately backtrack to the parent node. This is what is meant by *pruning*.

20.3.3 Example

Let's work out an example on Alpha-Beta algorithm that we just discussed above. Consider this game tree:-



Let's see it step-by-step:-

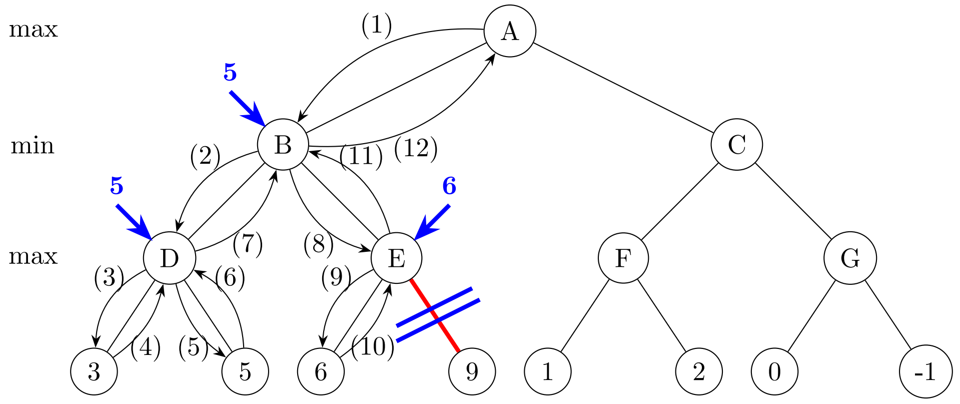


Step (1): Goes from A to B.

Step (2): Goes from B to D.

Steps (3) and (4): Fetches value 3 to D. α value of D becomes 3.

Steps (5) and (6): Fetches value 5 to D. α value of D becomes 5 and D node's value becomes max(3, 5) = 5.



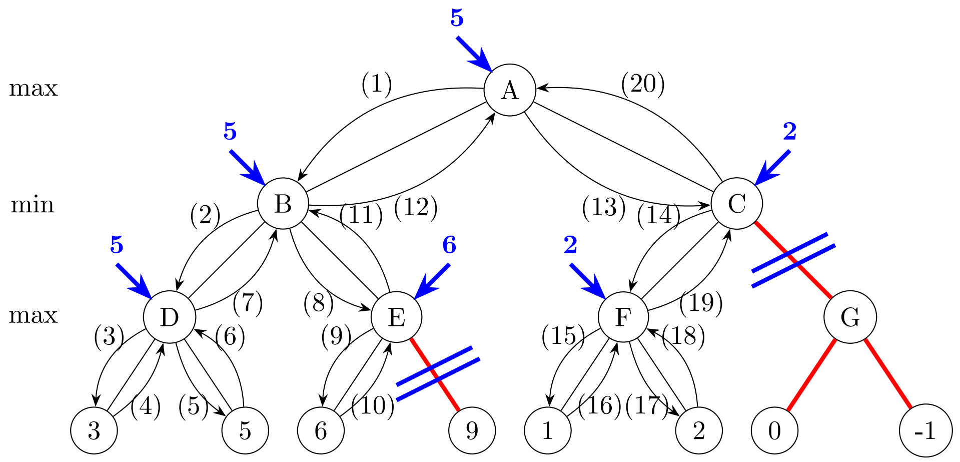
Step (7): Fetches value of D node (= 5) to B. β value of B becomes 5.

Step (8): Goes from B to E. $\beta = 5$ is copied to E.

Steps (9) and (10): Fetches value 6 to E. α value of E becomes 6. But now, $\alpha = 6 \geq 5 = \beta$ or $\alpha \geq \beta$. Hence we prune the edge from E to 9.

Step (11): Backtracks to B. B node's value becomes min(5, 6) = 5.

Step (12): Fetches value of B node (= 5) to A. α value of A becomes 5.



Step (13): Goes from A to C. $\alpha = 5$ is copied to C.

Step (14): Goes from C to F.

Steps (15) to (18): Value of F node comes out to be max(1, 2) = 2.

Step (19): Fetches value of F (= 2) to C. β value of C becomes 2. But now, $\alpha = 5 \geq 2 = \beta$ or $\alpha \geq \beta$. Hence we prune the edge from C to G and discard the subtree rooted at G.

Step (20): Backtracks to A. A's node value becomes max(5, 2) = 5.

20.4 Reconnaissance Blind Games

Reconnaissance Blind Games are a category of games characterized by imperfect information. In games of imperfect information, they cannot access the complete set of their opponent's moves. Instead, they have limited information about their opponents' past moves within a specified area.

- **Reconnaissance Blind Tic-Tac-Toe:** Consider a variant of Tic-Tac-Toe played on a 3x3 grid, where the player has limited visibility of their opponent's moves. At each turn, the player can only observe their opponent's past moves within a 2x2 region of their choice. This restricted visibility adds the element of imperfect information in the game, and the player must make decisions based on this partial information.
- **Reconnaissance Blind Chess (RBC):** RBC is a recently proposed imperfect-information variant of Chess. You can learn more about RBC from Reconnaissance Blind Chess website and Reconnaissance Blind Chess user profile page

20.5 Simultaneous Move Games

Let's explore a new realm of game theory: simultaneous move games. Unlike sequential or turn-based games, where players make decisions one after the other, simultaneous move games require players to choose their strategies simultaneously, without knowledge of their opponent's choices. This dynamic adds a layer of complexity, as each player must anticipate their opponents' moves while simultaneously formulating their strategies. We will now explore the strategic intricacies and analytical frameworks essential for understanding such games.

20.5.1 Penalty Shootout

We will begin our analysis with an example of a penalty shootout in a football game, featuring two adversarial agents: the shooter and the goalkeeper. The game setup is as follows:

- The shooter has three options: shooting toward the Left, Right, or Center in an attempt to score a goal
- Similarly, the goalkeeper can choose to dive toward the Left, Right, or Center to prevent the shooter from scoring
- If the shooter and goalkeeper both aim in the same direction, the goalkeeper can successfully block the shot, with one exception: the shooter is an exceptionally skilled left-side shooter. Hence, even if the goalkeeper dives to the left, the shooter possesses the ability to score by aiming in that direction.
- This constitutes a two-player zero-sum simultaneous move game, wherein the utilities gained by the players are inverse of each other
- For the shooter, successfully scoring a goal yields a utility of +1, while an unsuccessful attempt results in a utility of -1. The goalkeeper's utilities can be deduced accordingly from the aforementioned rule

20.5.2 Matrix Games

- It turns out that the game described above, along with the corresponding utilities/payoffs gained by the shooter in different scenarios, can be conveniently represented as a matrix
- In this matrix, each entry corresponds to the utility gained by the shooter based on the combination of strategies chosen by both players. Entries containing -1 represent goals blocked by the goalkeeper, while entries containing +1 signify goals successfully scored by the shooter
- Hence, these types of games, known as two-player zero-sum simultaneous move games, are also referred to as matrix games.

		Goalkeeper		
		Left	**Center**	**Right**
Shooter	**Left**	1	1	1
	center	1	-1	1
	Right	1	1	-1

Table 20.1: Game Matrix for Shooter vs. Goalkeeper

20.5.3 Equilibrium

Now that the game has been setup, we can proceed to analyze the equilibrium conditions for the game. First, let's define what equilibrium means in the context of such games.

Equilibrium: A tuple of actions from which no player gains by a unilateral deviation, where gain implies a strict increase in utility. A unilateral deviation refers to a situation where only the concerned player changes their move while the other players maintain their original moves.

In other words, an equilibrium is a state where no player has an incentive to deviate from their current strategy, as doing so would not result in a higher utility. We will show that in the above described game (L,L) is a simultaneous move equilibrium.

In this specific game, it is not difficult to see that (L,L) is a simultaneous move equilibrium i.e. none of the players have any reason to unilaterally deviate. This fact can be observed by considering various cases: we will see three cases to determine the equilibrium action for the shooter and similarly three cases to determine the goalkeeper's equilibrium action.

		Goalkeeper		
		Left	**Center**	**Right**
Shooter	**Left**	1	1	1
	Center	1	-1	1
	Right	1	1	-1

Table 20.2: Shooter's perspective keeping Goalkeeper's action constant

Let's approach the scenario from the shooter's viewpoint, while holding the goalkeeper's actions constant at a specific moment in time. By maintaining the goalkeeper's position fixed, we can analyze various advantageous positions for the shooter relative to the goalkeeper's movement. If the goalkeeper shifts towards the left, center, or right, favourable movement options for the shooter are (left, center, right), (left, right), and (left, center) respectively. By considering possible scenarios from the viewpoint of the shooter, we observe that shooting towards left is a favourable option for the shooter in all cases.

		Goalkeeper		
		Left	**Center**	**Right**
Shooter	**Left**	1	1	1
	Center	1	-1	1
	Right	1	1	-1

Table 20.3: Goalkeeper's perspective keeping Shooter's action constant

Let's approach the scenario from the goalkeeper's viewpoint, while holding the shooter's shoot direction constant at a specific moment in time. We can analyze various advantageous positions for the goalkeeper relative to the shooter's movement. If the shooter aims towards the left, center, or right, favorable movement options for the goalkeeper are (left, center, right), (center), and (right) respectively.

By considering possible scenarios from the viewpoint of both the shooter as well as the goalkeeper, we observe that taking intersection of the favourable cases for the shooter and the goalkeeper yields (L,L), (L,C) and (L,R). These are equilibrium states.

We see that the above analysis is cumbersome, isn't systematic, and requires taking cases. Since this isn't favourable for us, we try to search for a more structured and systematic way to determine the existence of an equilibrium.

20.5.4 The MinMax-MaxMin Method

The MinMax-MaxMin method is a useful technique for checking the existence of equilibrium in games. The procedure of proving (or disproving) existence of an equilibrium using this method is as follows:

1. **MinMax:**
   (a) Consider the move-set of the opponent(or the minimizing player)
   (b) For every possible move of the opponent, calculate the maximum utility that can be gained by our agent (or the maximizing player) using its move-set to respond to the opponent's move
   (c) The MinMax value is the minimum of the these maximum utilities
2. **MaxMin:**
   (a) Consider the move-set of our agent(or the maximizing player)
   (b) For every possible move of our agent, calculate the minimum utility our agent can be forced to gain by the opponent's actions (whose motive is to minimize the utility gained by our agent)
   (c) The MaxMin value is the maximum of the these minimum utilities

We will later prove that if MaxMin value = MinMax value, then an equilibrium exists, else it does not. This method provides a systematic approach to analyzing games and determining the presence of equilibrium. Let's see an example by analyzing the shooter-goalkeeper problem using this method.

		Goalkeeper			
		Left	**Center**	**Right**	**Minimum Utilities**
Shooter	**Left**	1	1	1	1
	center	1	-1	1	-1
	Right	1	1	-1	-1
	Maximum Utilities	1	1	1	**MaxMin = 1**

Table 20.4: Minimum Utilities

		Goalkeeper			
		Left	**Center**	**Right**	**Minimum Utilities**
Shooter	**Left**	1	1	1	1
	center	1	-1	1	-1
	Right	1	1	-1	-1
	Maximum Utilities	1	1	1	**MinMax = 1**

Table 20.5: Maximum Utilities

		Goalkeeper			
		Left	**Center**	**Right**	**Minimum Utilities**
Shooter	**Left**	1	1	1	1
	center	1	-1	1	-1
	Right	1	1	-1	-1
	Maximum Utilities	1	1	1	**MaxMin = MinMax = 1**

Table 20.6: MinMax = MaxMin

Since MaxMin value = MinMax value in this case, the element formed by the corresponding row and column, i.e. (L,L), represents a simultaneous move equilibrium.

Now, let's analyze a regular shooter (without the exceptional left-side shooting abilities). Below is the corresponding game matrix. The only alteration is that when shooter aims toward the left and when goalkeeper dives in the same direction, the goalkeeper catches the football, resulting in a utility gain of -1 for the shooter.

		Goalkeeper		
		Left	**Center**	**Right**
Shooter	**Left**	-1	1	1
	center	1	-1	1
	Right	1	1	-1

Table 20.7: Game Matrix for Shooter vs. Goalkeeper

		Goalkeeper			
		Left	**Center**	**Right**	**Minimum Utilities**
Shooter	**Left**	-1	1	1	-1
	center	1	-1	1	-1
	Right	1	1	-1	-1
	Maximum Utilities	1	1	1	**MaxMin $\neq$ MinMax**

Table 20.8: MinMax $\neq$ MaxMin

If we analyze the **MaxMin-MinMax** condition in this game matrix, we find that **MaxMin $\neq$ MinMax**, and therefore there exists no equilibrium

20.5.5 Building up towards the proof of MaxMin-MinMax condition

To establish the MinMax-MaxMin condition, we first define a term pertaining to the game matrix which is the **Saddle Point**.

- **Saddle Point:** an element in the game matrix that is both the largest in its column and the smallest in its row.

As it turns out, proving the **MinMax-MaxMin** condition is equivalent to the existence of a Saddle Point in the game matrix. First, we prove a lemma regarding **MinMax** and **MaxMin**. Formally, for any matrix game, we can express these as:

$$\max_{s_1} \min_{s_2} u(s_1, s_2) = \underline{v}$$
$$\min_{s_2} \max_{s_1} u(s_1, s_2) = \overline{v}$$

Let us try to compare the above values: is $\underline{v} \geq \overline{v}$ or $\underline{v} \leq \overline{v}$ or are they incomparable? Through a proof provided below, it's revealed that $\underline{v} \leq \overline{v}$.

Proof:

$$u(s_1, s_2) \geq \min_{s_2} u(s_1, s_2) \forall s_1, s_2$$
$$\implies \exists s_1^* \text{ s.t. } \min_{s_2} u(s_1^*, s_2) = \max_{s_1} \min_{s_2} u(s_1, s_2) \because \text{ actions } s_1 \text{ are finite}$$
$$\implies \text{ but } u(s_1^*, s_2) \geq \min_{s_2} u(s_1^*, s_2) = \max_{s_1} \min_{s_2} u(s_1, s_2)$$
$$\implies u(s_1^*, s_2) \geq \max_{s_1} \min_{s_2} u(s_1, s_2) \; \forall s_2$$
$$\text{also } \max_{s_1} u(s_1, s_2) \geq u(s_1^*, s_2) \; \forall s_2$$
$$\implies \exists s_2^* \text{ s.t. } \max_{s_1} u(s_1, s_2^*) = \min_{s_2} \max_{s_1} u(s_1, s_2)$$
$$\implies \min_{s_2} \max_{s_1} u(s_1, s_2) = \max_{s_1} u(s_1, s_2^*) \geq u(s_1^*, s_2^*)$$
$$\implies \min_{s_2} \max_{s_1} u(s_1, s_2) \geq u(s_1^*, s_2^*)$$
$$\text{Consider } u(s_1^*, s_2) \geq \max_{s_1} \min_{s_2} u(s_1, s_2) \; \forall s_2 \text{ and } \min_{s_2} \max_{s_1} u(s_1, s_2) \geq u(s_1^*, s_2^*)$$
$$\implies \min_{s_2} \max_{s_1} u(s_1, s_2) \geq u(s_1^*, s_2^*) \geq \max_{s_1} \min_{s_2} u(s_1, s_2)$$
$$\implies \min_{s_2} \max_{s_1} u(s_1, s_2) \geq \max_{s_1} \min_{s_2} u(s_1, s_2)$$
$$\text{Hence } \overline{v} \geq \underline{v}$$

■ See [scarvalone2008] for more details.

References

[scarvalone2008] Mira Scarvalone, "GAME THEORY AND THE MINIMAX THEOREM", *https://www.math.uchicago.edu/m̃ay/VIGRE/VIGRE2008/REUPapers/Scarvalone.pdf*