# Adversarial Search

Adversarial Search Problems are Search Problems in a COMPETITIVE EVIRONMENT that is a MULTIAGENT ENVIRONMENT where the agents’ goals are in conflict. Adversarial Search Problems are also known as GAMES.

In AI the most common games have the following properties:

1. **Deterministic**
2. **Turn-Taking** => fully observable environment
3. **Two-player** => two players alternate to play
4. **Zero-Sum** => the sum of the utilities of the players is always 0 for each instance of the game

An example of a Zero-Sum game is Chess where the utility associated to a player can be the following:

1. -1 => LOSS
2. 1 => WIN
3. 0 => TIE

For decades, researchers in Adversarial Search Problems have tried to make the best possible use of time. Indeed, in real games these algorithms must be able to do two things:

1. Make some decision even when it is infeasible to compute the OPTIMAL MOVE
2. Make some decision before the end of their turns

Therefore, an essential characteristic that ADVERSARIAL SEARCH ALGORITHMS must have is TIME EFFICIENCY.

The two TOOLS through which it is possible to make ADVERSARIAL SEARCH algorithm efficient are the following:

1. **Pruning** => cutting out portions of the STATE SPACE of the problem which are irrelevant
2. **Heuristic Function** => a function that estimates the true UTILITY of a state without doing a COMPLETE SEARCH.

A Game can be easily defined as a SEARCH PROBLEM as follows:

A picture containing text, newspaper

Description automatically generated

The Initial State, the Actions Functions and the Result Function define the GAME SPACE.

A close up of text on a white background

Description automatically generatedMany real-world GAME SPACES cannot be represented in a computer either because of memory or time inefficiencies, this is why in most cases the whole GAME SPACE is not traversed by an ADVERSARIAL SEARCH algorithm but only a portion of it which is called the **SEARCH SPACE.**

It is possible to estimate the utility value of a state of the SEARCH SPACE without a complete GAME SPACE search by using the HEURISTIC FUNCTION.

## MiniMax Algorithm

The MiniMax Algorithm is an ADVERSARIAL SEARCH ALGORITHM which computes the best move to make assuming that both players MAX and MIN play OPTIMALLY.

The algorithm takes the point of view of the MAX player and computes the utility of each state from the point of view of MAX. Thus, the higher the utility value of a state is, the better it is for MAX. On the other, the MIN player prefers low utility values.

The algorithm works as the following:

Given that it is Max turn

* It computes the utility value of all states that Max can reach with its move by assuming that from that state on both MIN and MAX will play optimally

Therefore, given that the algorithm states that the best action to perform for Max is action a which leads to a state with utility n, we know that n is reached if Min plays optimally and thus if Min does not play optimally Max can obtain an utility value which is higher than n.

Thus, the utility value associated with a node in a MINIMAX GAME SPACE is the maximisation of the worst case utility that MAX can obtain from that node down to the end of the game.

The whole algorithm MAXIMISES the worst case utility of Max because it assumes that MIN will play OPTIMALLY.

The MINIMAX function that computes the utility value of a node is the following:

A picture containing knife

Description automatically generated

A screenshot of a cell phone screen with text

Description automatically generatedThe MINIMAX algorithm pseudocode is illustrated below:

A screenshot of a cell phone

Description automatically generated

## Analysis of the MINIMAX algorithm

The time complexity of the MINIMAX Algorithm is O(bm) given that b is the branching factor and m is the maximum depth of the GAME SPACE.

As regards SPACE COMPLEXITY, in the case of the pseudocode provided above, since it generates one action at a time then space complexity is O(m). If it generated all actions from a node simultaneously then the space complexity would be O(bm).

The minimax algorithm proposed above uses DEPTH-FIRST SEARCH.

## MiniMax for more than 2 players

Naturally, it is possible to use the MiniMax Algorithm for games that involve more than 2 players.

The algorithms stays conceptually the same but every state in the Game Space instead of being associated with just one utility value must be matched to a vector of utility values each of them referring to one player.

A close up of a piece of paper

Description automatically generated

However, there is much more involved in MULTIPLAYER games than in two-player games. Indeed, in multiplayer games there are ALLIANCES that can be either implicit or explicit.

For example, suppose A and B are in weak positions and C is in a stronger position. Then it is often optimal for both A and B to attack C rather than each other, lest C destroy each of them individually. In this way, collaboration emerges from purely selfish behaviour. Of course, as soon as C weakens under the joint onslaught, the alliance loses its value, and either A or B could violate the agreement.

If the game is not zero-sum, then collaboration can also occur with just two players. Suppose, for example, that there is a terminal state with utilities < vA = 1000, vB = 1000 > and that 1000 is the highest possible utility for each player. Then the optimal strategy is for both players to do everything possible to reach this state—that is, the players will automatically cooperate to achieve a mutually desirable goal.

## Alpha-Beta Pruning

The time complexity of the MiniMax Algorithm is O(bm) which means that the number of nodes generated is exponential in relation to the maximum depth of the GAME SPACE. It turns out that we cannot reduce the EXPONENTIAL COMPLEXITY but we can cut the exponent in half by using PRUNING.

That is, it is possible to cut portions of the GAME SPACE safely and still obtain correct minimax values for the portion of the game space being analysed.

The technique that lets us do it is called **ALPHA-BETA PRUNING**.

## The general idea of ALPHA-BETA PRUNING



A close up of a map

Description automatically generated

In other words, if PLAYER can get in the WORST CASE a utility of value m if opting for a move that leads to the state a and the value n is smaller than m then since we assume that the OPPONENT plays optimally then the utility that the PLAYER gets going through state b is for sure less than m. In other words, we do not need to expand anymore all subtrees rooted at the state b, that is we **PRUNE** the subtrees rooted at b.

Another explanation of why we can prune nodes safely getting the right minimax values is explained by the following illustration:

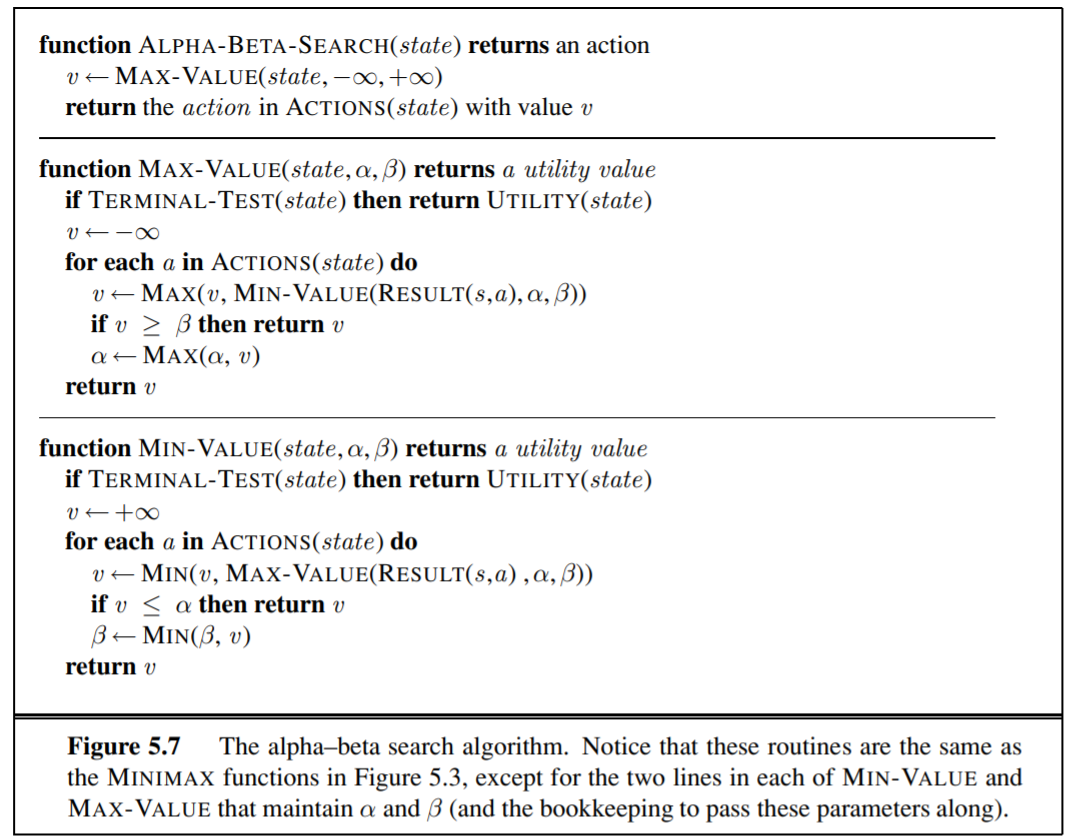
A close up of a map

Description automatically generated



## Pseudocode of Alpha-Beta Pruning



The pseudocode works in the following way:

1. The MAX-VALUE function is called at nodes where it is the turn of the MAX Player. On the other hand, the MIN-VALUE function is called at nodes where it is the turn of the MIN Player.
2. At each node it is passed from the parent node the values α and β where α is the value of the best choice found so far for the MAX Player and β indicates the value of the best choice found so far the MIN Player
3. The lines in the code that make it possible to prune parts of the GAME SPACE are indicated with a **X**. When we are located at a node where it is the turn of the MAX Player and an action has been tried which leads to a utility of n which is greater than the β value then the minimax value of the current node must be greater than n assuming that the Max Player will play optimally. Since the current node whose turn is of the Max Player will definitely lead to a utility value which is greater than the current best found path for the Min Player than the Min Player has no reason to reach the current state and so we can prune all not expanded subtrees rooted at the current node by returning n as the utility of the current node. Same thing holds for the α value.

What I said is illustrated in the following illustrations.

## Killer Move Heuristic

The effectiveness of alpha–beta pruning is highly dependent on the order in which the states are examined. For example, in Figure 5.5(e) and (f), we could not prune any successors of D at all because the worst successors (from the point of view of MIN) were generated first. If the third successor of D had been generated first, we would have been able to prune the other two. This suggests that it might be worthwhile to try to examine first the successors that are likely to be best.

The killer move heuristic tries to make alpha-beta pruning as efficient as possible and it consists of trying for each node the best action for the player that must make a move in that node. Naturally, it is not possible to know for sure which action will be the best move for the player that must make the move and this is why it is a HEURISTIC.

It has been computed that using Alpha-Beta Pruning with the killer move heuristic (assuming the best move is always guessed) gives as result a time complexity of O(bm/2) in other words the branching factor passes from b to

Alpha-Beta pruning with a random choice of action will achieve on average O(b3m/4)

## Transposition Table

In some Game Spaces there may be repeated states. We can use a Transposition Table in case there are repeated states to save the utility value associated to a state. By doing so, the time complexity to analyse the Game Space can reduce dramatically. However, if too many nodes are present in the Game Space it may happen that the computer runs out of space because too many states are saved in the TRANSPOSITION TABLE.

The name of this table comes from the fact that in some games it is possible to reach the same state by using different permutation of a move sequence and such permutations are called **TRANSPOSITIONS.**

## Imperfect Real time Decisions

Still, Alpha-Beta Pruning may involve to look for a huge SEARCH SPACE and this is IMPRACTICAL in many Real-Time Decisions Problems.

The solution to this is to cut even more the GAME SPACE by using a cutoff limit. In other words, the GAME SPACE is generated until a certain depth value. The nodes that are in this cutoff value are treated as LEAF NODES and their utility value is computed by using an **EVALUATION FUNCTION** (heuristic function).

Such evaluation function is called EVAL and the function to compute the minimax value of each state is called H-MiniMax.

A picture containing knife

Description automatically generated

The characteristics that the EVALUATION FUNCTION must have are the following:

1. It must not take too long
2. It must reflect the actual chances of winning of a state

A close up of a newspaper

Description automatically generated

A screenshot of a cell phone screen with text

Description automatically generated

Implementing this CUTOFF methodology to cut the GAME SPACE can be easily implemented in the Alpha-Beta Pruning Algorithm by simply replacing the TERMINAL-TEST with the CUT-OFF TEST.

However, establishing a fixed depth level can be a very simple approach that can lead to erratic results.

A better approach is to use ITERATIVE DEEPENING that is going a level deeper at each iteration and when the time runs our returning the best found move that has been found during the last iteration of ITERATIVE DEEPENING.

Consider again the simple evaluation function for chess based on material advantage. Suppose the program searches to the depth limit, reaching the position in Figure 5.8(b), where Black is ahead by a knight and two pawns. It would report this as the heuristic value of the state, thereby declaring that the state is a probable win by Black. But White’s next move captures Black’s queen with no compensation. Hence, the position is really won for White, but this can be seen only by looking ahead one more ply. Obviously, a more sophisticated cutoff test is needed. The evaluation function should be applied only to positions that are quiescent—that is, unlikely to exhibit wild swings in value in the near future. In chess, for example, positions in which favorable captures can be made are not quiescent for an evaluation function that just counts material. Nonquiescent positions can be expanded further until quiescent positions are reached. This extra search is called a quiescence search; sometimes it is restricted to consider only certain types of moves, such as capture moves, that will quickly resolve the uncertainties in the position.

Thus, we have given birth to a better version (efficient in real time problems) of Alpha-Beta Pruning consisting of the CUTOFF METHOD WITH ITERATIVE DEEPENING + QUIESCENT SEARCH.

## The Horizon Effect

The Horizon Effect is another problem that the CUTOFF METHOD can introduce. It consists of the fact that when an opponent makes a move which causes serious damage and cannot be avoided since the algorithm won’t search the whole GAME SPACE it may think to stop such a move. However such a move cannot be avoided and the algorithm does not know it because it is over its HORIZON (the algorithm should go deeper in the SEARCH TREE). In order to stop such a move, the algorithm may damage itself making the move even more damaging.

Consider the chess game in Figure 5.9. It is clear that there is no way for the black bishop to escape. For example, the white rook can capture it by moving to h1, then a1, then a2; a capture at depth 6 ply. But Black does have a sequence of moves that pushes the capture of the bishop “over the horizon.” Suppose Black searches to depth 8 ply. Most moves by Black will lead to the eventual capture of the bishop, and thus will be marked as “bad” moves. But Black will consider checking the white king with the pawn at e4. This will lead to the king capturing the pawn. Now Black will consider checking again, with the pawn at f5, leading to another pawn capture. That takes up 4 ply, and from there the remaining 4 ply is not enough to capture the bishop. Black thinks that the line of play has saved the bishop at the price of two pawns, when actually all it has done is push the inevitable capture of the bishop beyond the horizon that Black can see.

One way to avoid the Horizon Effect is to use the technique called SINGULAR EXTENSION.

## Forward Pruning

An additional technique to reduce the SEARCH SPACE is FORWARD PRUNING. FORWARD PRUNING consists of pruning a node immediately without further consideration.

One way to implement FORWARD PRUNING is to use the EVALUATION FUNCTION so that to consider only the n best actions. This approach is called BEAM SEARCH.

Furthermore, if the algorithm being implemented is ALPHA/BETA Pruning then out of the n best actions it is possible to prune those ones that lead to a node whose utility (according to the EVALUATION FUNCTION) is greater than β or smaller than α.

## Search vs Lookup

At the beginning of a game or at the end of a game when there are fewer moves to do it is possible to use a lookup table in order to map a state to the best move.