Adversarial Search

Table of Contents

[Game Playing 2](#_Toc138878448)

[Adversarial Games 2](#_Toc138878449)

[Deterministic Games 4](#_Toc138878450)

[Zero-Sum Games 4](#_Toc138878451)

[Single Agent Trees 5](#_Toc138878452)

[Adversarial Game Trees 7](#_Toc138878453)

[Minimax Properties 9](#_Toc138878454)

[Minimax Efficiency 9](#_Toc138878455)

[Game Tree Pruning 10](#_Toc138878456)

[Properties 11](#_Toc138878457)

[Resource Limitations 12](#_Toc138878458)

[Evaluation Function 13](#_Toc138878459)

## Game Playing

* Checkers
  + 1950 – First computer player
  + 1994 – First computer champion
  + 2007 – Solved (all possible terminal states reachable)
* Chess
  + 1997 – Deep Blue defeats human champion. Examined 200 million positions per second.
  + Current programs are even better
* Go
  + Branching factor of more than 300 makes this more difficult. Classic programs use pattern knowledge while recent ones use the Monte Carlo expansion method.
  + 2016 – Alpha GO defeats human champion. Uses Monte Carlo Tree Search and the Odd-Even evaluation function.

## Adversarial Games

So far, we have explored methods which calculated everything at the beginning and then followed a certain series of actions to reach a goal. This can help us solve things such as the shortest path required to eat all the dots in PACMAN. However, real games have opponents, and since we cannot know beforehand what moves the opponent will make, this requires real time computation. The **behaviour** of our agents are then modified based on the **computational results**.

**Adversarial Games** are ones which have players competing towards opposite goals. For example, PACMAN is an adversarial game since PACMAN’s goal is to eat all the dots as quickly as possible and the goal of the ghosts is to eat PACMAN as quickly as possible. Notice how it is not necessary for the other ‘player’ to even be an actual player for the game to be considered an adversarial game. We just need an opponent.

There are several ways in which we can categorize adversarial games:

* **Deterministic vs Stochastic** – A deterministic game is one in which we know all possible outcomes, such as Tic-Tac-Toe or Chess. A stochastic game is one in which the outcome is unknown, such as Monopoly.
* **Number of Players** – We can have single player games such as Solitaire, 2-Player games like Checkers or multi-player games such as Dungeons and Dragons.
* **Zero Sum** – Zero sum games are ones where the total results of all the players involved cancel out. For example, in Football, if one team wins, the other must lose. This makes it a zero-sum game. However, something like nuclear war (not really a game but okay) is non-zero-sum since there are no winners.
* **Perfect Information** – There are games in which we know everything that is happening, such as in Tic-Tac-Toe since the move the opponent has made is known, and ones where we do not, such as Poker since the cards the opponent has is unknown.

Our goal is to create an algorithm which calculates a strategy which will recommend a move for us to make at every state.

### Deterministic Games

There are many possible formalizations for deterministic games, but one which will look familiar goes like this:

* **States** – , starting with state
* **Players** – , which can be a number between and . Usually, the players take turns to make moves.
* **Actions** – , which might depend on the player and the state at which the action is being taken.
* **Transition Functions** - , which just takes the current state and the action to be made and provides the new state.
* **Terminal Tests** - , which checks if a terminal state has been reached.
* **Terminal Utilities** - . Each outcome has a utility attached to it. For example, the utility in Monopoly might be a positive or negative cash value.

The solution to be given for a specific player is a **policy**, .

### Zero-Sum Games

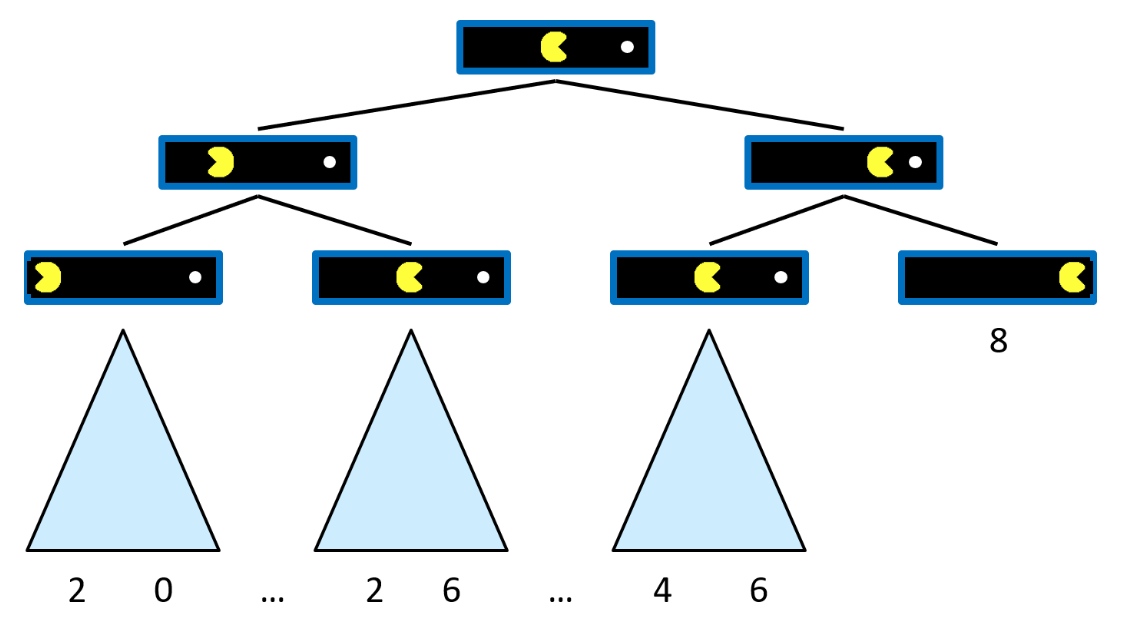
In **Zero-Sum Games**, the agents have opposing utilities. Consider this a single value that one agent attempts to maximize, and the other agent attempts to minimize. This is truly adversarial.

This is opposed to **general games**, which the agents have independent utilities. For example, we could have one agent trying to find gems of one colour from a pile while the other agent is trying to find gems of a different colour. Here, the agents could cooperate, be indifferent to each other, compete and more. For example, one agent removing a gem of their colour reduces the search space for the other agent, which makes it a form of cooperation (albeit perhaps unwilling cooperation).

## Single Agent Trees

**Adversarial Search** works like this: For our current state, we think about what move we will make, and then what move the opponent will make if we make that move, and then what move we will make when the opponent makes that move, and then what move the opponent will make when we make that move, and so on.

To get into adversarial search, let’s first start with a simple situation. Suppose we are the only agent in the PACMAN game. From a given simple starting state, there are several branches we can explore.



Let’s assign a cost of -1 to making any move and a reward of +10 for eating a food pellet. Thus, the above diagram shows that the right-most branch is the optimal path, with the total points earnt being 8. This value is assigned to the root and is called the **utility**.

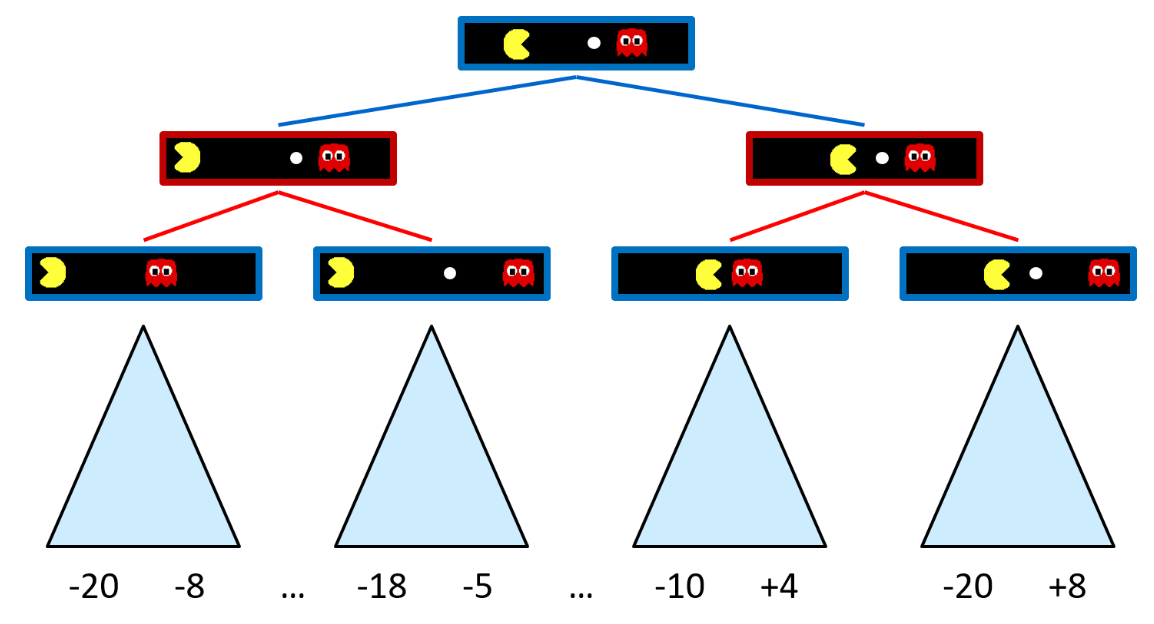
The only places where we actually know a value are the **terminal states**, since we can calculate the value at that point based on how many moves we took to get there. Let the value at a terminal state be .

At **non-terminal states**, the value is found as maximum value from all the children of that state. The value of each child is in turn found from its children and so on. Thus, the value for every state can only be calculated by going to the terminal states and working our way backwards. We can only find a root value if we manage to get to a terminal state.

A game is only considered ‘solved’ once we know the root value, which in turn means that we need to have reached every possible terminal state (how else would we know that the current value we have is the best possible one?). This is the reason why some games, such as Tic-Tac-Toe, which have a low branching factor, are considered solved, while others, such as Chess, which have a high branching factor are not considered solved. Chess has its terminal states very deep in the tree. So deep in fact, that current computers are unable to reach them.

## Adversarial Game Trees

Moving on to our actual game. In adversarial game tress, each player takes a turn. For simplicity, we will consider that we make the first move. For a simple scenario, we could have a tree like this:

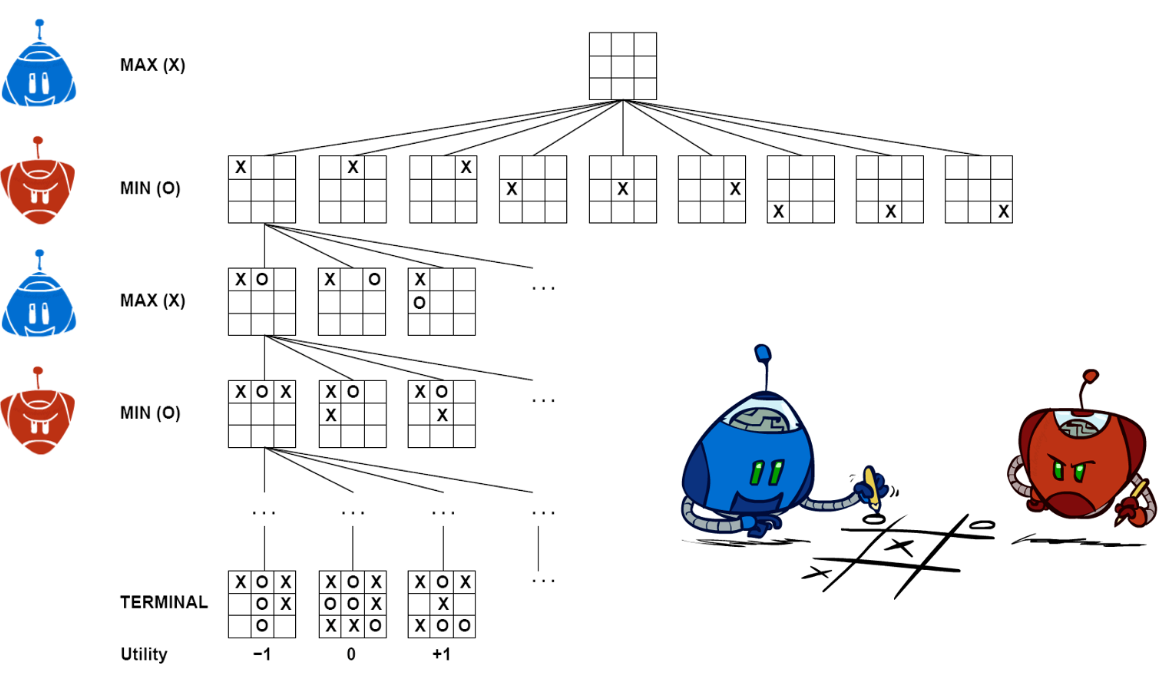


Thus, we are making one of two moves, which takes us to one of two new states. From each of those states, the opponent might make one of two moves.

For the time being, suppose we somehow know the values of the terminal states and have a situation such as the one shown above. Remember how we said that the value at each state is the maximum value amongst all its children? We choose which move is the optimal one based on this information. Well, that is only true for us. This is an adversarial game, so the opposite is true for our opponents. The ghost will pick the child which gives the **minimum value**.

Since the states at each row use the maximum and the minimum values in turn, these are called **minimax values**. This is used in all deterministic zero-sum games.

Coming back to Tic-Tac-Toe, a part of the possible branches is shown below.



Thus, for us, the best possible outcome is +1, while the best possible outcome for the opponent is -1. The middle ground is 0. If both players are playing optimally (taking the best move at every state), we will end up with a score of 0. This is why it is a zero-sum game.

A similar scenario can be created for checkers, which is also solved. However, it cannot be done for chess, which is why it is not solved.

The implementation for **minimax search** is provided below:



### Minimax Properties

**Minimax Search** is optimal against a perfect player. However, real players make mistakes, so we should also prepare for that scenario. In addition, if we are the ones designing a game (meaning we are the opponent), we should also design our agents to make mistakes on purpose sometimes. Otherwise, if we always play optimally, it becomes impossible for the player to beat us. No one wants to play an unbeatable game. On the other hand, if we are the player and we somehow know that the opponent is going to make a suboptimal move, we need to make sure to catch that and exploit it to get better results.

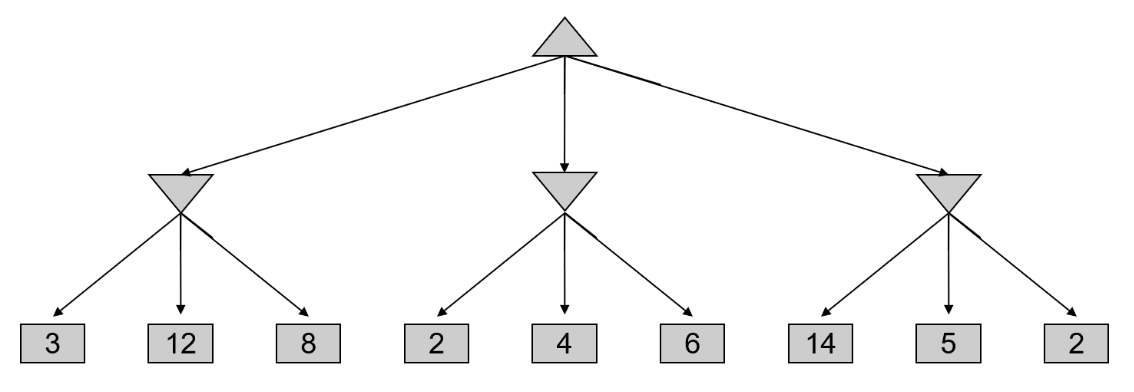
We will be exploring all of these properties in more depth soon. For the time being, just keep these things in mind.

### Minimax Efficiency

Minimax search is literally just an exhaustive DFS. This has a time complexity of and a space complexity of . For a game like chess with a branching factor and depth of , this becomes impossible to solve.

Next, we will be exploring ways in which we can exploit certain scenarios to avoid having to exhaustively search the entire tree and still solve the game.

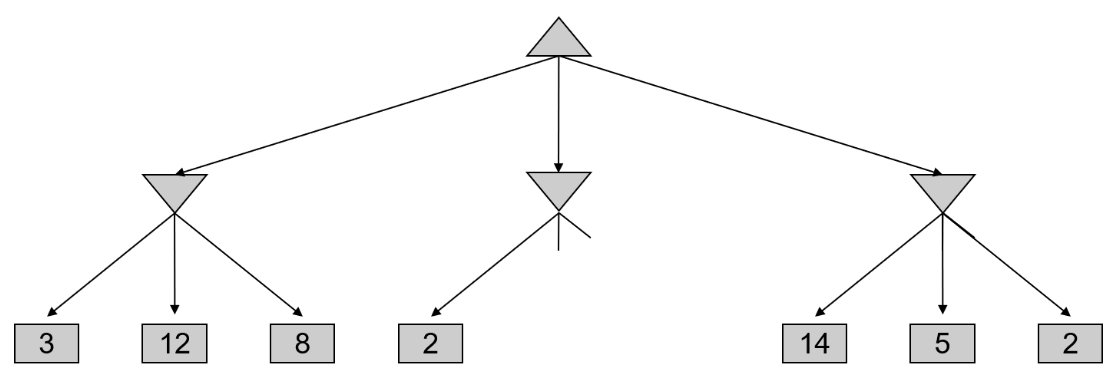
## Game Tree Pruning



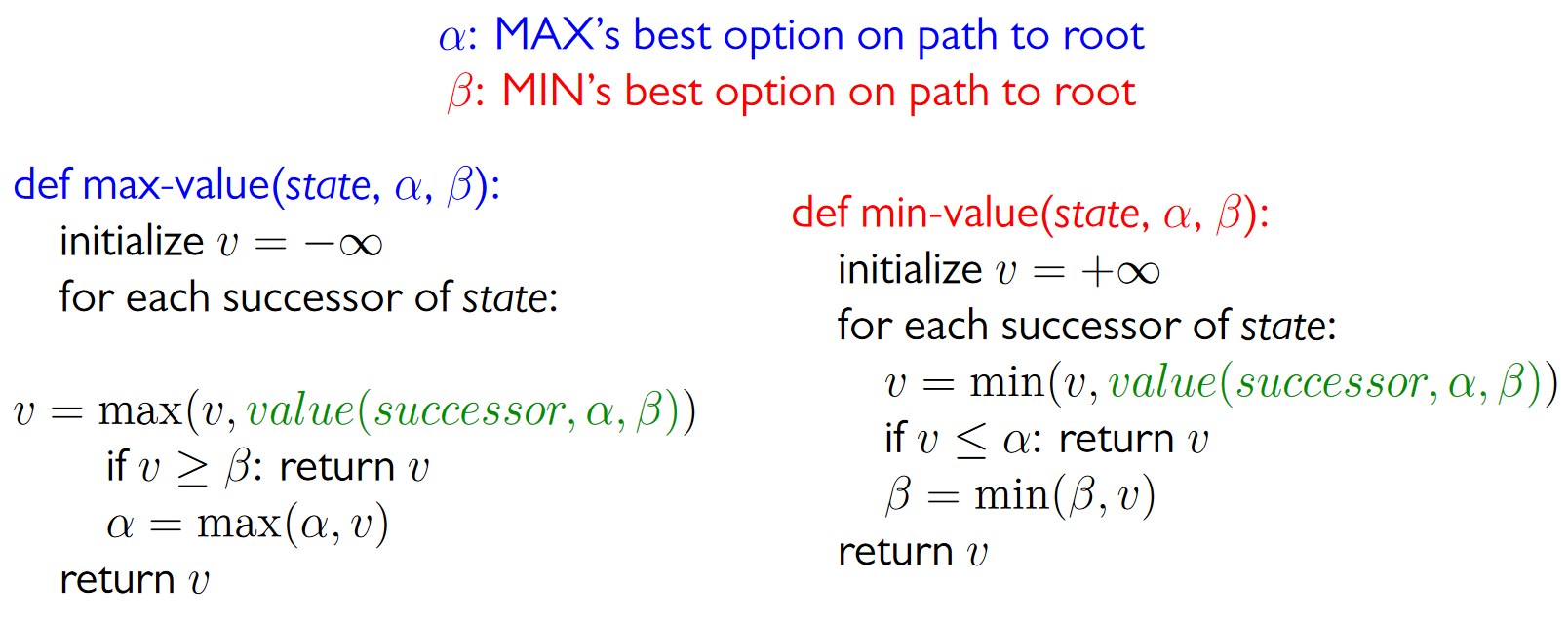
Suppose we have an adversarial search tree like the one shown. We make the first move, and the opponent makes the second move. After the first move, we will be at the second layer, which has three nodes. We do not know which node to pick, so we explore its children.

Remember that the second layer is the opponents move, so each node will have the minimum value. The leftmost node thus has a value of 3. At the central node, the first value is 2. We can in fact skip the rest of the children.

The leftmost node has a value of 3 so that is our best possible value so far. The first child of the central node has a value of 2. At this point we know that the central node will have a value of at most 2 (since the opponent will try to pick the lowest value). Thus, we will not be using the central node and there is no point in exploring further.

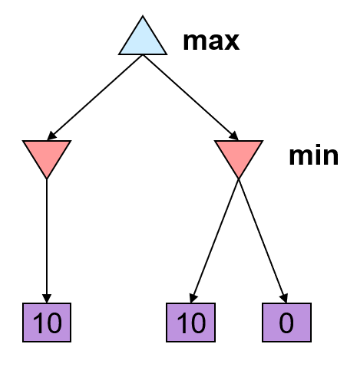


This process is called **game tree pruning** or **alpha-beta pruning**. The implementation for this is provided below:



### Properties

Alpha-beta pruning has no effect on the final root value. However, the **intermediate values** might be incorrect. In a situation where two sub-branches have equal values (due to incorrect calculations), it is possible to make a wrong move.



In the above tree, the branch with 0 at the end would be pruned, so we will not see it. Both of the nodes at the second level have the same value, so we might pick the right node (randomly). Unfortunately, this will take us to the 0 outcome (since the opponent will pick that node).

There are two possible solutions to this. The first is to prune only on **inequalities**. The second is to keep track of which node was calculated first, since we know for sure that that node has the correct calculation.

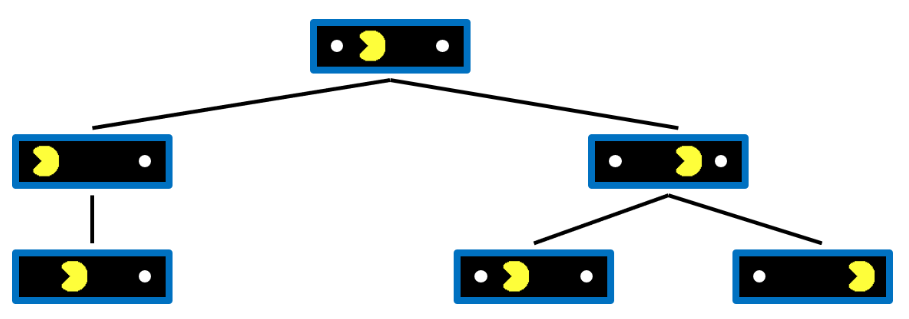
The **order of the children** is also important. A good ordering will allow us to prune more branches. With ‘perfect’ ordering, the time complexity can be reduced to , which doubles the solvable depth.

Alpha-beta pruning is an example of **meta-reasoning**, which is computing things to decide what to compute.

## Resource Limitations

In realistic games, it is usually not possible to search the tree to the leaves. The solution is to perform **depth-limited search**. At each state, we search the tree up to a limited depth. Instead of using the terminal utilities at the ‘leaves’, we use an **evaluation function** which gives us an estimate for the value at that state. By doing this, our agent becomes a **replanning agent**.

Unfortunately, by doing this, we lose our guarantee of **optimality**. In fact, it can lead to lots of problematic situations, such as the one below.



In the above situation, the PACMAN can only see up two moves ahead. Thus, going west (to eat a dot) and then east seems just as good as going east and then west. This leads to PACMAN going back and forth without moving forward. This situation is called **thrashing**.

## Evaluation Function

Ideally, the evaluation function would return the true value at that state, but of course, this is not possible practically. In reality, it is usually the **weighted linear sum** of the features, such as the number of each type of piece left in a game of chess.

There is no standard way of measure how good an evaluation function is. Instead, we use different example scenarios and check if the evaluation function is returning lower scores for worse situations. For example, in chess, a state with fewer important pieces left should return a lower value than a state with more important pieces left.

Evaluation functions will always be imperfect. The **further ahead** we can see without having to use the evaluation function, the **better our results** are going to be.