Lecture notes on Zero-Sum, Perfect-Information, Two-Player, Sequential, Finite Games

Jan Cervenan
MFF UK
jancervenan@gmail.com

April 8, 2025

Contents

Any comments on these notes are welcome. If there is an inconsistency or even a mistake, please let me know. Reach me at `jancervenan@gmail.com`. Questions are also welcome.

1. Introduction

Games are a fundamental part of human culture and society. They are a way to entertain, compete, and learn. Games have been studied in various fields, such as mathematics, computer science, economics, and psychology. As the title suggests, this talk will focus on zero-sum, perfect-information, two-player, sequential, finite games. Examples of such games include chess, checkers, tic-tac-toe, and go.

Note that the field of game theory is vast and complex, and this talk will merely scratch the surface. This is not a comprehensive overview of games as such but rather a focused discussion on a specific subset of games. It is not intended to be mathematically rigorous but rather serves as an introduction to the topic. We'll skip many details and subtleties and focus on the main ideas, providing some intuition and examples to help the reader understand the concepts. We will also present some basic algorithms and strategies for solving these games on a computer.

Also, many thanks to Jan Vendelin Tajcovsky and Zoe Cervenan for their assistance with language and technical refinements.

2. Formalities

2.1 Definition of a Game

Let us start with a definition of a game that we'll work with.

Definition 2.1.1: ***Game***

A game is defined as a tuple:

$$G = (S, T, P, A, \mathcal{P}, \mathcal{U})$$

where:

- S *is the set of all possible states.* ***Initial*** *state is denoted* s_0.
- $A(s)$ *is the set of available* ***actions*** *at state* s.
- $\mathcal{P} : S \to P$ *assigns to a each state all players from player set* $P = \{p_1,\ p_2,\ \dots\}$, *who are taking an action in that state.*
- $T \subset S$ *is the set of* ***terminal*** *states.*
- $\mathcal{U} : P \times T \to \mathbb{R}$ *is the* ***payoff*** *(or utility) function, while* $\mathcal{U}(p,\ t)$ *represents the outcome for player* $p \in \{p_1,\ p_2,\ \dots\}$ *at state* $t \in T$.

A game is called ***perfect-information*** *if every player knows the full history and state of the game.*
A game is called ***two-player*** *if there are exactly two players in set* P.
A game is called ***sequential*** *if players take turns making decisions.*
A game is called ***finite*** *if it has a finite number of states and terminal states, and if every possible sequence of actions that players can take before the game terminates (starting from any state) has a finite length.*

This definition is quite general and can be applied to many games. Chess is a classic example of a zero-sum, perfect-information, two-player, sequential, finite game. Examples of games that lack perfect-information include poker or bridge, where players have hidden information. Non-sequential games are games where players make decisions simultaneously, such as rock-paper-scissors. Infinite games are games with an infinite number of states, such as tic-tac-toe on an infinite grid.

Regarding the payoff function, game theory assumes that players are *rational*.

Assumption 2.1.1: ***Rationality***

Players act ***rationally****, meaning they try to* ***maximize*** *their payoff and their choices of actions are consistent with this goal.*

Definition 2.1.2: ***Zero-sum game***

Let G *be two-player finite game. A game is called* ***zero-sum*** *if the sum of the players' payoffs is zero for all terminal states:*

$$\forall t \in T : \sum_{p \in P} \mathcal{U}(p,\ t) = \mathcal{U}(p_1,\ t) + \mathcal{U}(p_2,\ t) = 0.$$

Chess is also a zero-sum game, as both players cannot win the same game. If we consider payoff of the winning player to be 1, the drawing player to be 0 and the losing player to be -1, then the sum of payoffs is always zero.

In the following, we will shorten "zero-sum, perfect-information, two-player, sequential, finite games" to "ZPTSF games" for brevity[1].

For ZPTSF games, we can make a trivial observation.

[1] If at any point we use, for example, "PSF game," it is to be understood as "Perfect-information, Sequential, Finite game"

Lemmish 2.1.1: ***About payoffs***

In zero-sum two-player games, the payoff of player p_1 is the negative of the payoff of player p_2.

$$\mathcal{U}(p_1,\ t) = -\mathcal{U}(p_2,\ t).$$

Thanks to this it is possible to introduce a single payoff function $u(t)$ as follows.

Definition 2.1.3: ***Two-player payoff function***

Let $G = (S, T, P, A, \mathcal{P}, \mathcal{U})$ be a ZPTSF game. We define a function $u : T \to \mathbb{R}$ as

$$u(t) = \mathcal{U}(p_1,\ t)\ \forall t \in T.$$

This function is called the ***two-player payoff function*** *of the game G.*

We will refer to both $\mathcal{U}(p,\ t)$ and $u(t)$ as the payoff functions of the game G, as it will be clear from the context which one is meant. This redefinition of the payoff function allows for simpler reasoning about the game states and their payoffs. We can now introduce and talk about maximizing and minimizing players, which will prove very useful later.

Lemma 2.1.1: ***Maximizing and minimizing players***

Let $G = (S, T, P, A, \mathcal{P}, \mathcal{U})$ be a ZPTSF game. Let u be the two-player payoff function of game G as above. Then player p_1 is trying to maximize the payoff function $u(t)$, while player p_2 is trying to minimize it.

Proof: From Assumption 2.1.1, we know that players are both rational and trying to maximize their payoff $\mathcal{U}(p,\ t)$. The definition of $u(s)$ implies that player p_1 is trying to maximize $u(t) = \mathcal{U}(p_1,\ t)$. Thanks to Lemma 2.1.1 we see that $\mathcal{U}(p_2,\ t) = -u(t)$, so player p_2 is trying to minimize $u(t)$, as it leads to maximizing $\mathcal{U}(p_2,\ t)$. □

We call player p_1 the *maximizing player* and player p_2 the *minimizing player*. From the above lemma, it should be clear why we refer to them in this way.

2.2 Strategies

A strategy is a plan of action for a player. In ZPTSF games, we can define a strategy as a function that maps states to actions.

Definition 2.2.1: ***Strategy***

Let $G = (S, T, P, A, \mathcal{P}, \mathcal{U})$ be a ZPTSF game. A ***strategy*** *for player p is a function $\sigma_p : S \to A$ that assigns a set of actions to each state. We denote the set of all strategies for player p as Σ_p.*

For the maximizing player (recall we denote them p_1), we can say that at each state $s \in S$, they aim to choose the action $A \ni a = \sigma_{p_1}(s)$ that leads to the terminal state with the highest payoff. Whether they can do this or not is currently unclear. We will return to maximizing and minimizing strategy formulations later, when we discuss the minimax theorem. For proper formulation we need to know, how to extend the definition domain of the payoff function to the entire set S.

There are also other strategies, like the *random strategy* which chooses an action with some probability distribution, but these lie outside the scope of this talk.

It is also natural to define the concept of an optimal strategy.

Definition 2.2.2: ***Optimal strategy***

Let $G = (S, T, P, A, \mathcal{P}, \mathcal{U})$ be a ZPTSF game. A strategy σ is an ***optimal strategy*** *for player p if it always selects the action that leads to the highest possible payoff for player p in the given state.*

2.3 Representation of Games

Game theory works with various representations of games and matches. Studying these representations can help us understand the structure of the game and develop strategies for playing it. Some representations are more suitable for theoretical analysis, while others are more practical for implementation in computer programs. The formal definitions of different game representations are quite complex and would require much more time and space to be introduced. There are two main representations of games: *Normal form* and *extensive form*. For introductory purposes, it will be sufficient to say that normal form is a game matrix and extensive form is a game tree.

2.3.1 Game matrix

This is best demonstrated on a simple example. Let us consider a ZTF game of rock-paper-scissors. The game matrix for this game is shown in Table 2.1. From the game matrix it is clear that the game is zero-sum, as the sum of payoffs is always zero. The game is also finite, as after one round, the game ends. The game is obviously not sequential, players make their choices simultaneously. It is a *simultaneous* game.

What about perfect-information games? Perfect-information game is a game where players know the full history and state of the game. This is not the case here, as players do not know the opponent's choice while making theirs. However, both players are aware of the possible strategies, possible actions of the opponent and the payoff matrix of the game, allowing them to make a rational decision. Similar games include e.g. poker, where players do not know the hands of others but are aware of possible combinations and strategies. Games like this are reffered to as *complete-information* games and we will not discuss them in depth here.

$p_2 \rightarrow$ $p_1 \downarrow$	**Rock**	**Paper**	**Scissors**
Rock	p_1 gets 0 p_2 gets 0	p_1 gets -1 p_2 gets 1	p_1 gets 1 p_2 gets -1
Paper	p_1 gets 1 p_2 gets -1	p_1 gets 0 p_2 gets 0	p_1 gets -1 p_2 gets 1
Scissors	p_1 gets -1 p_2 gets 1	p_1 gets 1 p_2 gets -1	p_1 gets 0 p_2 gets 0

Table 2.1: Game matrix for rock-paper-scissors.

The game matrix is a simple way to represent a game, but it is not always the most suitable. For example, in multi-player games, the game matrix can become very large and difficult to read. Also when the game lasts more than one move, the game table grows... A LOT... The number of rows and columns in a game matrix of rock-paper-scissors with n rounds is 3^n. For example, for $n = 3$ rounds, there are $3^3 = 27$ possible combinations of actions. One possible game of rock-paper-scissors with 3 rounds could be played out

Round	p_1	p_2	p_1 payoff	p_2 payoff
1	Rock	Paper	-1	1
2	Rock	Rock	0	0
3	Scissors	Rock	-1	1
Total payoff			-2	2

Table 2.2: Example of a game of rock-paper-scissors with 3 rounds.

for example like in Table 2.2. Note please that repeated zero-sum games preserve this property, even with multiple rounds. The full game matrix for two rounds of rock-paper-scissors is shown in Table A.1.

2.3.2 Game tree

Another way to represent a game is as a *directed tree*, where:

- Nodes correspond to game states.

- Edges represent actions that lead to new states.
- Leaves correspond to terminal states with payoffs.

This representation is meant for sequential games, where players take turns making moves until the game ends. A sample game tree is on the figure 2.1. The game starts in the initial state, and players take turns making actions. Each action leads to a new state, until we reach a terminal state with a payoff. In the game tree, we can see all the possible sequences of actions and their outcomes.

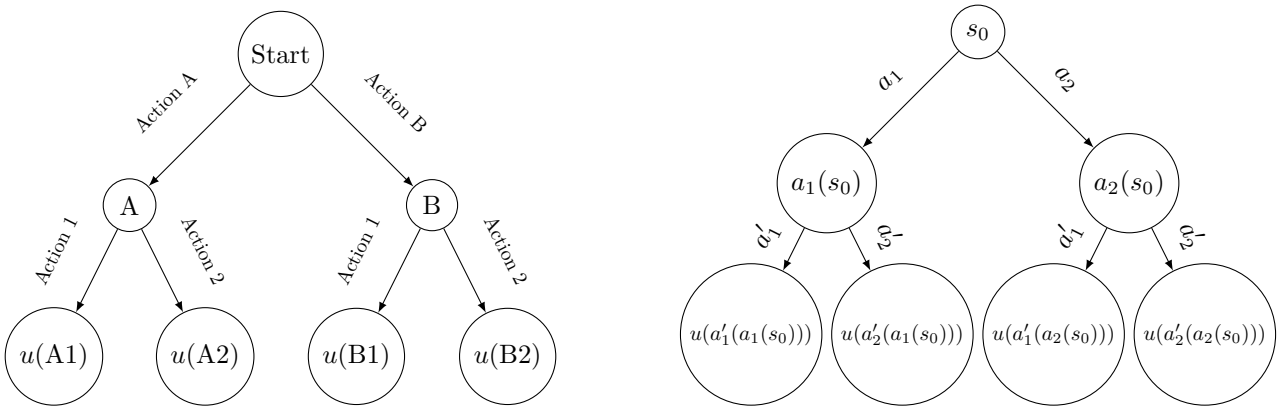

Figure 2.1: A sample game tree. In the left figure, a simple game tree with two levels is shown. The last level is terminal. In the right figure is the same tree with the notation used in the text.

Consider tic-tac-toe. The game tree is too large to be fully shown, as already in the second layer there are 9 nodes. In the second layer, there are 72 nodes. Sketch of the first two layers of this tree is shown in figure 2.2. This factorial tendency is diminishing as the game progresses, as some moves lead to terminal states and therefore we do not get to make a choice from then onwards.

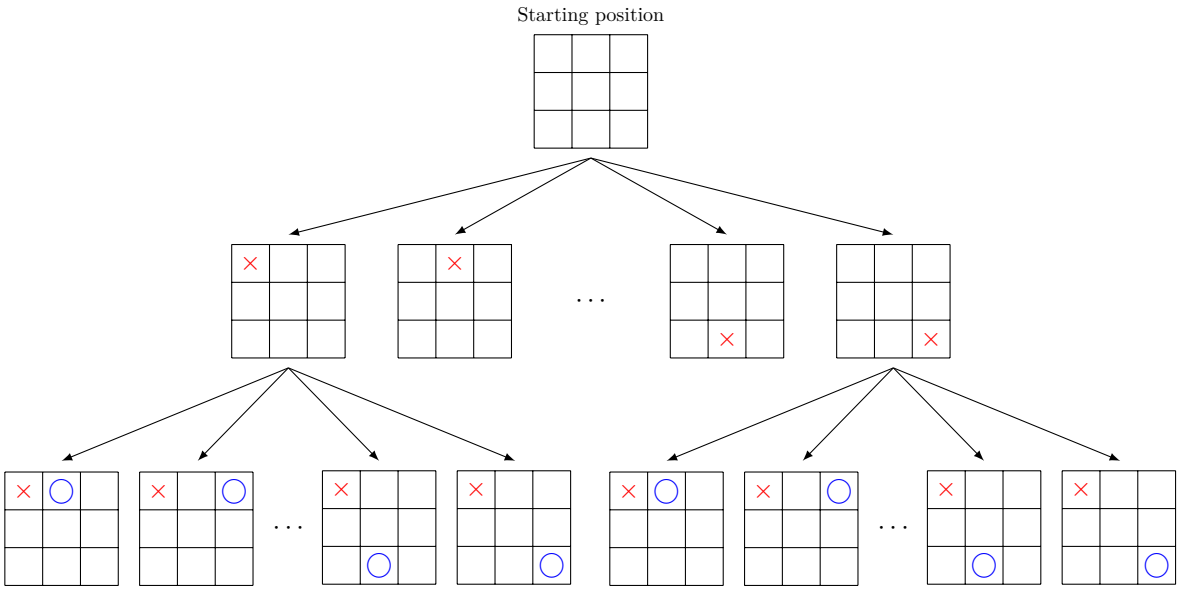

Figure 2.2: First two layers of game tree for tic-tac-toe. It's a simple game, but the tree can grow quickly.

The game tree is a powerful tool for analyzing games, but it can be computationally expensive to work with. Later we will discuss algorithms that can be used to analyze game trees and find optimal strategies.

2.4 Game match

By *match* M_G of a ZPTSF game G we understand a series of states connected by actions that ends at a terminal state. Let us the following game of tic-tac-toe as an example:

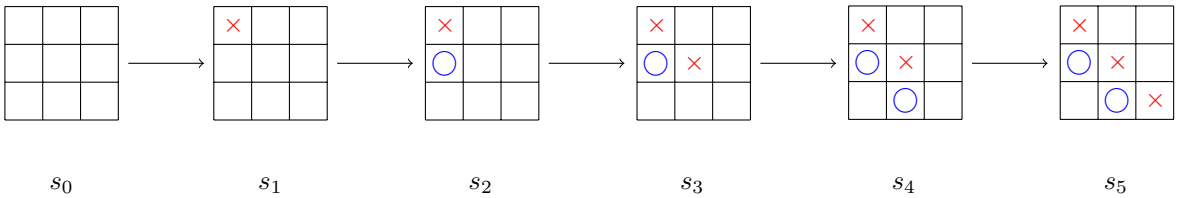

Figure 2.3: A match of tic-tac-toe.

The above representation is called *representation through a sequence of states*. This representtaion is the most explicit but also the most space-consuming. Saving this in computer memory or drawing it on paper would require storing all game states, which can be highly inefficient for larger games.

If we were to explicitely add the actions to the representation above, the game would be represented as:

$$s_0 \xrightarrow{p_1 \to (1,1)} s_1 \xrightarrow{p_2 \to (1,2)} s_2 \xrightarrow{p_1 \to (2,2)} s_3 \xrightarrow{p_2 \to (3,1)} s_4 \xrightarrow{p_1 \to (3,3)} s_5 \in T.$$

Since the actions are deterministic, the match is uniquely determined by the initial state and the actions taken by the players. It is therefore possible to represent the same match in a simplified manner as

$$M_G = (s_0,\ \{(1,1),\ (1,2),\ (2,2),\ (3,1),\ (3,3)\}),$$

or, if the initial state is known, it could be represented even more simply:

$$M_G = \{(1,1),(1,2),(2,2),(3,1),(3,3)\}.$$

This is a sequence of actions taken by players in the match. We call this the *representation through a sequence of actions*. Since the last move is terminal, we can use the payoff function to determine the outcome of the match

$$u(s_5) = \mathcal{U}(p_1,\ s_5) = 1, \quad \mathcal{U}(p_2,\ s_5) = -1.$$

The value $u(s_5) = 1$ means player 1 wins in the state s_5. Sometimes, an alternative notation

$$u(M_G) = 1$$

can be seen. This is a shorthand for the above, where the payoff function is applied to the terminal state of the match. The value of $u(M_G)$ is called the *payoff of the match* M_G.

Definition 2.4.1: ***Length of a match and maximal length of a game***

The ***length*** *of a match M_G is the number of actions taken in the match. The* ***maximal length*** *of a game G is the length of the longest possible match of a game G.*

We can refine the notion of a "finite game" by requiring only that its maximal length is finite. This condition alone does not force the game itself to have finitely many states or terminal positions. Indeed, it permits constructions like infinite-board tic-tac-toe, provided we impose rules which limit the total number of moves.

However, as soon as we allow for inifinite state spaces, we must be careful about the assumptions of the *Zermelo's theorem*. We will formulate the theorem in the next chapter and it will be crucial for our further discussion. Most ways this theorem is usually proven are based on the assumption that the game has a finite number of states. For an infinite state space, one must appeal to well-foundedness of the move relation and deploy more advanced set-theoretic arguments (e. g. transfinite induction) to extend the Zermelo's theorem. This works whenever every match is guaranteed to terminate in finite time, but the proof's details become more involved, especially if one allows uncountably many states. Games sticking to our original definition, that is finite in terms of maximal length and number of possible states, are called *strictly finite games*.

We take a wild guess that your local supermarket does not sell inifinite papers for tic-tac-toe, and therefore we will stick to the concept of strictly finite games.

3. Solving a Game

A solution of a game is, vaguely put, a strategy that guarantees the best possible outcome for all players, given their optimal play. In this chapter, we will discuss the concept of solving a game and how to find such solutions.

3.1 Zermelo's Theorem

Definition 3.1.1: ***Preceding state***

Let $G = (S, T, P, A, \mathcal{P}, \mathcal{U})$ be a ZPTSF game. We say that a state s' is a ***preceding state*** *of the state s if there exists an action $a \in A(s')$ such that $s = a(s')$.*
The set $\mathbb{S}(s)$ of all the preceding states of s is defined as

$$\mathbb{S} = \{s' \in S \mid \exists a \in A(s') : s = a(s')\}.$$

We define $\mathbb{S}(s_0) := \{s_0\}$ for all initial states s_0.

Definition 3.1.2: ***Successor state***

Let $G = (S, T, P, A, \mathcal{P}, \mathcal{U})$ be a ZPTSF game. We say that a state s' is a ***successor state*** *of the state s if there exists an action $a \in A(s)$ such that $s' = a(s)$.*
The set $\mathbb{F}(s)$ of all the successor states of s is defined as

$$\mathbb{F} = \{s' \in S \mid \exists a \in A(s) : s' = a(s)\}.$$

We define $\mathbb{F}(t) := \{t\}$ for all terminal states t.

Definition 3.1.3: ***Isolated state***

Let $G = (S, T, P, A, \mathcal{P}, \mathcal{U})$ be a ZPTSF game. We say that a state $t \in T$ is an ***isolated state*** *of the game G if $\mathbb{S}(t) = \{t\}$ and $\mathbb{F}(t) = \{t\}$.*

It is evident that under assumption of ZPTSF game have all states in $\mathbb{S}(s)$ the same value of $\mathcal{P}$. We will therefore write just $\mathcal{P}(\mathbb{S}(s))$ to denote wich player it taking an action in all states from $\mathbb{S}(s)$ (similarly for $\mathbb{F}$). Now we are ready to formulate and prove the Zermelo's theorem.

Theorem 3.1.1: ***Zermelo's Theorem***

Let $G = (S, T, P, A, \mathcal{P}, \mathcal{U})$ be a ZPTSF game. Then in each game state either

(i) player p_1 has a winning strategy no matter what p_2 does,

(ii) player p_2 has a winning strategy no matter what p_1 does, or

(iii) the game is a draw if both players play optimally.

Proof: Recall that player p_1 is the maximizing player and player p_2 is the minimizing player. Let us also restrict the payoff function u to the values $\{-1, 0, 1\}$ for win, draw or loss of player p_1 respectively. Let us have a look at T and assign a value to each terminal state using u. Since the isolated states are by definition also terminal, we can assign values to them as well. We then consider the set of states

$$\mathbb{S}_1 = \{s \in \bigcup_{t \in T} \mathbb{S}(t);\ \mathbb{F}(s) \subset T\}.$$

Since all terminal states of the game take values assigned to them via function u, we can propagate the values to each state $s \in \mathbb{S}_1$ using the following simple logic.

- If $\mathcal{P}(\mathbb{S}(s)) = p_1$, then the value we assign to each $s' \in \mathbb{S}(s)$ is $\max_{s' \in \mathbb{F}(s)} u(s')$.
- If $\mathcal{P}(\mathbb{S}(s)) = p_2$, then the value we assign to each $s' \in \mathbb{S}(s)$ is $\min_{s' \in \mathbb{F}(s)} u(s')$.

Now, given that all the states from $\mathbb{S}_1$ and T have assigned some value, we assign values to all states s from the set

$$\mathbb{S}_2 = \left\{ s \in \mathbb{S}_1 \cup \left(\bigcup_{t \in T} \mathbb{S}(t) \right) ; \ \mathbb{F}(s) \subset \mathbb{S}_1 \cup T \right\},$$

using the same logic as above. By repeating this process, we can assign values to all states in the system of sets

$$\mathbb{S}_{n+1} = \left\{ s \in \left(\bigcup_{i=1}^{n} \mathbb{S}_i \right) \cup \left(\bigcup_{t \in T} \mathbb{S}(t) \right) ; \ \mathbb{F}(s) \subset \left(\bigcup_{i=1}^{n} \mathbb{S}_i \right) \cup T \right\},$$

until we cover the whole set S by the system $\{\mathbb{S}_i\}_{i=1}^{N}$ where N is the maximal length of the game G. We were able to assign a value from $\{-1, 0, 1\}$ to each state in the game, especially to the initial state s_0.

This concludes the proof of the theorem, as we can now say that the game

- is a win for player p_1 if $u(s_0) = 1$,
- is a win for player p_2 if $u(s_0) = -1$,
- is a draw if $u(s_0) = 0$.

□

When the game satisfies the assumptions of the Zermelo's theorem, we can say that the game is *determined*. This means that the outcome of the game is – in principle – known from the beginning. Our proof is different than the original proof by Ernst Zermelo from 1913 [2], as well as many other proofs that can be found in the literature. Typically, the proofs are based on the idea of induction on the length of the game. We approached the proof in this way to introduce the idea we will now generalize.

Definition 3.1.4: ***Minimax state payoff function***

Let $G = (S, T, P, A, \mathcal{P}, \mathcal{U})$ be a ZPTSF game. We define a function $v : S \to \mathbb{R}$ as

$$v(s) = \begin{cases} \max_{a \in A(s)} v(a(s)) & \text{if } s \in S \setminus T, \ \mathcal{P}(s) = p_1, \\ \min_{a \in A(s)} v(a(s)) & \text{if } s \in S \setminus T, \ \mathcal{P}(s) = p_2, \\ u(s) & \text{if } s \in T. \end{cases}$$

This function is called the ***state payoff function*** *of the game G. It assigns to each state the best possible payoff that can be achieved from that state.*

A biproduct of our proof of the Zermelo's theorem is that the state payoff function is well-defined. We could say that this function returns values assigned to states by the procedure we have shown. The above definition is simply a recursive and more transparent way to define the function. We can now state and prove the following claim.

Claim 3.1.1: ***Optimality of the minimax state payoff function***

Let $G = (S, T, P, A, \mathcal{P}, \mathcal{U})$ be a ZPTSF game. Then for each state $s \in S$ the value of the state payoff function $v(s)$ is the best possible payoff that can be achieved from that state, if both players play optimally in sense of Definition 2.2.2.

Proof: We will prove the claim by induction on the maximal length n of the game.

Base case: Let $n = 1$. Then $\forall a \in A(s) : a(s) \in T$.

- If it is the maximizing player's turn, then their choice is $\max_{a \in A(s)} u(a(s))$.
- If it is the minimizing player's turn, then their choice is $\min_{a \in A(s)} u(a(s))$.

In both cases it holds that the choice of the player is in accordance with the result of the state payoff function defined as in Definition 3.1.4. Thus the claim holds for $n = 1$.

Inductive step: Let the claim hold for all games with maximal length $< n$. Let $s \in S$ be an initial state of a game with maximal length n. Then all games starting in states from $\mathbb{S}(s)$ can have maximal length $\leq n - 1 < n$. According to the induction hypothesis, the value of the state payoff function $v(s')$ for such states is the best possible payoff that can be achieved from the state $s' \in \mathbb{S}(s)$, meaning all actions from $A(s)$ have optimal values given by $v(a(s))$.

- If it is the maximizing player's turn, then their choice is $\max_{a \in A(s)} v(a(s))$.
- If it is the minimizing player's turn, then their choice is $\min_{a \in A(s)} v(a(s))$.

In both cases it holds that the player's optimal choice is in accordance with the result of the state payoff function defined as in Definition 3.1.4. □

3.2 Choosing the optimal strategy for ZPTSF game

Of course, the players want to win the game. For this, they need to choose the optimal strategy. Although the optimal strategy is not always easy to find, if we're given the minimax state payoff function (Definition 3.1.4), we can easily determine the optimal strategy (Definition 2.2.2) for both players.

We can now formulate the optimal strategies for both players.

Lemmish 3.2.1: ***Optimal strategy for the maximizing player***

*Let $G = (S, T, P, A, \mathcal{P}, \mathcal{U})$ be a ZPTSF game. The strategy $\sigma = \sigma_{max}$ is an **optimal strategy** for the maximizing player p_1 if*

$$\forall s' \in \{s \in S;\ \mathcal{P}(s) = p_1\} \forall a' \in \sigma(s') : v(a'(s')) = \max_{a \in A(s')} v(a(s')).$$

Lemmish 3.2.2: ***Optimal strategy for the minimizing player***

*Let $G = (S, T, P, A, \mathcal{P}, \mathcal{U})$ be a ZPTSF game. The strategy $\sigma = \sigma_{min}$ is an **optimal strategy** for the minimizing player p_2 if*

$$\forall s' \in \{s \in S;\ \mathcal{P}(s) = p_2\} \forall a' \in \sigma(s') : v(a'(s')) = \min_{a \in A(s')} v(a(s')).$$

Both results can be proven using the minimax algorithm. The minimax algorithm works as follows:

(i) For each state $s \in S$, we calculate the minimax state payoff function $v(s)$.

(ii) We then find an action $a \in A(s)$ such that $v(s) = v(a(s))$.

(iii) We repeat this process until we reach the terminal state.

Since we have proven that our minimax state payoff function is well-defined and gives optimal payoffs for both players, we can be sure that the strategy found using this algorithm is indeed the optimal one.

But what if we do not have the luxury of the minimax state payoff function? Is it still possible to find the optimal strategy? Is it at least possible to find a strategy that can get close enough? That is the main topic of the next section.

4. Engines

By a game engine, we mean a program that can play a game. The engine can be used to play against a human player, against another engine, or to analyze the game. Many of us have already played against a chess engine, such as Stockfish, Komodo, or Houdini. These engines are very powerful and can beat even the best human players by a longshot. But how do these engines work? How do they decide which move to play next? And how come they do not need to think for hours to make a move?

In this chapter, we will discuss the basics of game engines. More specifically, we will focus on algorithms that aid the engine in searching through the tree for optimal strategies. Sometimes, it is easy. Games such as tic-tac-toe or connect four are simple enough to be solved completely. But what about more complex games such as chess or go? These games require more sophisticated approach. We cannot just go ahead and bruteforce calculation of the minimax state payoff function for all states. We need to be smarter. A natural idea is to approximate the minimax state payoff function in a clever way, that allows us to search the game tree more efficiently, without the need to find the terminal states explicitly. If such an approximation is found, one can proceed with a partial search of the game tree. At a certain depth we can take the reached states as terminal states and evaluate them using the approximation. Then via a minimax algorithm, one can propagate the values back to the root of the tree and decide which move to play.

Definition 4.0.1: ***Approximation of the state payoff function***

Let $G = (S, T, P, A, \mathcal{P}, \mathcal{U})$ be a ZPTSF game. Let u be the two-player payoff function of the game and $\sigma_{p_1} = \sigma_{max}$, $\sigma_{p_2} = \sigma_{min}$ be optimal strategies for the maximizing and minimizing player respectively. Let $\tilde{v}$ be a function that we use to evaluate the states. We call $\tilde{v}$ an ***approximation of the state payoff function*** *with error $I_{\tilde{v}}$, if it satisfies the properties*

(i) $\tilde{v}(s)$ *is defined for all states* $s \in S$.

(ii) $\forall t \in T : \tilde{v}(t) = u(t)$.

(iii) $\exists I_{\tilde{v}} > 0 \; \forall s \in S \; \forall a \in \sigma_{\mathcal{P}(s)}(s) : \; |\tilde{v}(s) - \tilde{v}(a(s))| \leq I_{\tilde{v}}$

A short remark is in order. The first property is necessary, because we need to be able to evaluate all states. It is also usually the only one that can be guaranteed. The second property is necessary, because we need to be able to evaluate terminal states correctly. This is in practice most of the time achieved by a different set of rules, that are built to recognize and evaluate terminal states correctly in the sense of the u function, so it is not included directly in the implementation of $\tilde{v}$. When a terminal state is reached, we replace the value of the approximative state payoff function with the value of the actual payoff function, since we know it for terminal states. The third property of the approximation of the state payoff function is probably the most interesting one. It effectively states that if $\tilde{v}$ in state $s \in S$, $\mathcal{P}(s) = p_2$ takes a value that shows a big advantage for player 1, then player 2 should not be able to make a move that shifts this value to their advantage by a big margin (it has to be smaller than $I_{\tilde{v}}$). If we were to put $I_{\tilde{v}} = 0$, we would arrive at the minimax state payoff function.

A careful reader might have noticed that when defining the approximative state payoff function, we have used knowledge of both optimal strategies. When defining these strategies, we assumed full knowledge of the minimax state payoff function. This renders the approximative state payoff function useless, as in practice we do not have this knowledge. How can we resolve this? Is there any way to approximate the minimax state payoff function without knowing it a priori? The answer is yes and no.

This is achieved by literally playing the game. Vaguely said, the recipe is this:

- Build multiple distinct guesses for the approximative state payoff function.
- Build multiple engines that use these guesses to find the optimal strategy.
- Orchestrate a tournament between the engines.
- The engine that wins the tournament has the highest chance of carrying the best approximative state payoff function in terms of error $I_{\tilde{v}}$.
- Repeat the process with the winning engine and new, more refined guesses for the approximative state payoff function.

This is a very expensive process, as it requires a lot of computational power and time. Moreover, we have no measure of the error $I_{\tilde{v}}$ of our approximative state payoff function. Thus the above holds only as long as somebody does not come up with a new strategy and destroys all of our efforts in one go. Then there is just a few options. Namely: cry and start over, cry and leave the field, or resort to violence and destroy the new engine and its authors[1].

4.1 Tree search algorithms

How exactly do we let the engine decide based on the approximate state payoff functions we have designed? The answer is the tree search. We have to search the game tree, assign values to states and then propagate them back to the root, based on which we can decide which move to play. The previously mentioned minimax algorithm is the most straightforward way to do this, but it is unfortunately not very efficient, as it requires evaluating all states of the game tree. This is where the tree search algorithms come in. There are tons of sophisticated ways to search through the game tree. Some reduce the number of states that need to be evaluated, and thus allow for faster search of the game tree. Others remember already evaluated states and use them to search the game tree more efficiently. Some use heuristics to guide the search in a more promising direction. In the following sections, we provide a very brief overview of some of these algorithms. For a deeper understanding, we refer the reader to [1].

4.1.1 Alpha-Beta pruning

Pseudocode for this algorithm is given in Algorithm B.1. It cuts off the search of the game tree branch when it is clear that the values in the branch will not affect the final decision. It is based on the idea that if we know that the value of the state is not going to be used, we do not need to evaluate it, allowing us to prune big chunks of the tree.

4.1.2 Null-move heuristic

Pseudocode for this algorithm is given in Algorithm B.2. The *null-move heuristic* is a technique used in game tree search algorithms, like Alpha-Beta pruning, to improve efficiency. It works by temporarily skipping (or "nullifying") the current player's turn to see if the opponent can gain a significant advantage. This helps to quickly identify unpromising moves and prune parts of the game tree.

The idea is that if skipping a turn still results in a good position for the current player, then exploring that branch further is unnecessary. This saves computation time by focusing only on more critical moves. However, it has some limitations. Null-move heuristic can fail in situations where skipping a turn misrepresents the true value of a position, such as in zugzwang (a chess term where any move worsens the player's position).

4.1.3 Alpha-Beta pruning with transposition tables

Pseudocode for this algorithm is given in Algorithm B.4. Transposition tables are used to store the results of previously evaluated positions in the game tree. This allows the algorithm to avoid re-evaluating the same position multiple times, saving computation time. The idea is to store the position's hash value and the associated value in a table. When the algorithm encounters the same position again, it can look up the value in the table instead of re-evaluating it. If it finds a value, it can use it to prune the search tree more effectively. If the value is not found, the algorithm proceeds with the evaluation as usual.

It is important to mention that one should also account for the depth of the search reached after the saved move in the transposition table. The value of the move in the transposition table is only valid for the depth at which it was calculated. If the depth of the current search exceeds the depth of the saved move, the value may no longer be valid, and the search should continue as usual. However, we can still use the data in the transposition table to order moves before proceeding with the standard search, increasing our chances of achieving cutoffs.

4.1.4 Negamax

All three of the previous algorithms can be simplified to improve efficiency if we exploit the two-player payoffs. The key idea is to use the fact, that the two player payoffs are related by multiplication by -1. Speedup is

[1] This is not recommended, as it is illegal and can lead to a serious consequences. It is also not very effective, as the engine will probably be repaired and improved afterwards by someone else.

then achieved by makig the search always behave as if it was the maximizing player's turn, while inverting the evaluation sign between turns. Again, the pseudocode is given in Algorithm B.3.

5. Conclusion

Using the knowledge gained from this text, one should be able to understand the basic principles of engines for ZPTSF games. We have shown how to find the optimal strategy for both players in a ZPTSF game, and how to approximate it if the direct calculation is not feasible.

One part we did not cover is the evaluation function itself. This is a crucial part of the game engine, as it determines the quality of the moves the engine makes. Unfortunately, there is no unified way to create an evaluation function, as it depends on the game itself. Nowadays, the evaluation function is often created using machine learning techniques, such as neural networks. Back in the day the evaluation function used to be created by hand, using expert knowledge of the game, empirical data, and some heuristics, which is still used in some engines today.

Of course, there is much more to consider beyond the theory. In real-world scenarios, we need to consider the computational complexity of the algorithms and memory requirements. We also need to consider the trade-offs between the quality of the evaluation function and the time it takes to calculate it, as well as how to implement all the mentioned algorithms in a programming language of our choice. The choice of the programming language itself is also important, as some languages are better suited for certain tasks than others. For example, C++ is often used for game engines, as it is fast and has low-level access to the hardware. Equally important is the choice of data structures and algorithms, as they can significantly impact the performance of the engine. If the reader finds themselves interested in this topic, I recommend visiting the website [1].

Please feel free to contact me with any questions or to discuss these topics further – perhaps with an eye toward building your own game engine. Thank you for reading this text. I hope you have enjoyed reading it at least half as much as I enjoyed writing it.

Jan Cervenan

A. 2-round Rock-Paper-Scissors

The game matrix is a simple way to represent a game, but it is not always the most suitable. For example, in multi-round games, the game matrix can become very large and difficult to read. Following is the fully expanded two-round Rock-Paper-Scissors game matrix with all possible combinations of actions. The payoffs are 1 and -1 for the winning and losing player respectively. Zero stands for a draw. Results of both rounds are summed up.

$p_2 \rightarrow$ $p_1 \downarrow$	**(R,R)**	**(R,P)**	**(R,S)**	**(P,R)**	**(P,P)**	**(P,S)**	**(S,R)**	**(S,P)**	**(S,S)**
(R,R)	(0,0)	(-1,1)	(1,-1)	(-1,1)	(-2,2)	(0,0)	(1,-1)	(0,0)	(2,-2)
(R,P)	(1,-1)	(0,0)	(-1,1)	(0,0)	(-1,1)	(-2,2)	(2,-2)	(1,-1)	(0,0)
(R,S)	(-1,1)	(1,-1)	(0,0)	(-2,2)	(0,0)	(-1,1)	(0,0)	(2,-2)	(1,-1)
(P,R)	(1,-1)	(0,0)	(2,-2)	(0,0)	(-1,1)	(1,-1)	(-1,1)	(-2,2)	(0,0)
(P,P)	(2,-2)	(1,-1)	(0,0)	(1,-1)	(0,0)	(-1,1)	(0,0)	(-1,1)	(-2,2)
(P,S)	(0,0)	(2,-2)	(1,-1)	(-1,1)	(1,-1)	(0,0)	(-2,2)	(0,0)	(-1,1)
(S,R)	(-1,1)	(-2,2)	(0,0)	(1,-1)	(0,0)	(2,-2)	(0,0)	(-1,1)	(1,-1)
(S,P)	(0,0)	(-1,1)	(-2,2)	(2,-2)	(1,-1)	(0,0)	(1,-1)	(0,0)	(-1,1)
(S,S)	(-2,2)	(0,0)	(-1,1)	(0,0)	(2,-2)	(1,-1)	(-1,1)	(1,-1)	(0,0)

Table A.1: Two-round Rock-Paper-Scissors game matrix. Payoffs are in form (p_1, p_2).

B. Pseudocodes

Algorithm B.1 Alpha-beta pruning

```
1:  function AlphaBeta(node, depth, alpha, beta, maximizingPlayer)
2:      if depth = 0 or node is terminal then          ▷ Base case: leaf node or max depth
3:          return Evaluate(node)                      ▷ Return evaluation of the node
4:      end if
5:      if maximizingPlayer then                       ▷ Maximizing player's turn
6:          value ← −∞                                 ▷ Initialize value
7:          for all child in Children(node) do         ▷ Iterate over child nodes
8:              temp ← AlphaBeta(child, depth-1, alpha, beta, False)   ▷ Recursive call for other player
9:              value ← max(value, temp)               ▷ Update the best value
10:             alpha ← max(alpha, value)              ▷ Update alpha (best value so far)
11:             if alpha ≥ beta then
12:                 break                              ▷ Beta cutoff
13:             end if
14:         end for
15:         return value                               ▷ Return the best value found
16:     else                                           ▷ Minimizing player's turn
17:         value ← ∞                                  ▷ Initialize nullValue
18:         for all child in Children(node) do         ▷ Iterate over child nodes
19:             temp ← AlphaBeta(child, depth-1, alpha, beta, True)    ▷ Recursive call for other player
20:             value ← min(value, temp)               ▷ Update the best value
21:             beta ← min(beta, value)                ▷ Update beta (best value so far)
22:             if alpha ≥ beta then
23:                 break                              ▷ Alpha cutoff
24:             end if
25:         end for
26:         return value                               ▷ Return the best value found
27:     end if
28: end function
```

Algorithm B.2 Alpha-beta pruning with Null-Move Heuristic

```
1: function AlphaBetaNullMove(node, depth, alpha, beta, maximizingPlayer)
2:     if depth = 0 or node is terminal then ▷ Base case: leaf node or max depth reached
3:         return Evaluate(node) ▷ Return evaluation of the node
4:     end if
5:     if NullMoveAllowed(node, depth) then ▷ Check if null move is allowed
6:         nullBeta ← beta - 1 ▷ Set a reduced beta value
7:         reduction ← R(depth) ▷ Calculate depth reduction
8:         newDepth ← depth - 1 - reduction ▷ Reduce depth
9:         nullValue ← AlphaBetaNullMove(node, newDepth, alpha, nullBeta, not maximizingPlayer)
   ▷ Perform null move search
10:        if nullValue ≥ beta then
11:            return beta ▷ Null move cutoff
12:        end if
13:    end if
14:    if maximizingPlayer then ▷ Maximizing player's turn
15:        value ← −∞ ▷ Initialize value
16:        for all child in Children(node) do ▷ Iterate over child nodes
17:            temp ← AlphaBetaNullMove(child, depth-1, alpha, beta, False) ▷ Call for other player
18:            value ← max(value, temp) ▷ Update the best value
19:            alpha ← max(alpha, value) ▷ Update alpha (best value so far)
20:            if alpha ≥ beta then ▷ Beta cutoff condition
21:                break ▷ Prune the branch
22:            end if
23:        end for
24:        return value ▷ Return the best value found
25:    else ▷ Minimizing player's turn
26:        value ← ∞ ▷ Initialize value
27:        for all child in Children(node) do ▷ Iterate over child nodes
28:            temp ← AlphaBetaNullMove(child, depth-1, alpha, beta, True) ▷ Call for other player
29:            value ← min(value, temp) ▷ Update the best value
30:            beta ← min(beta, value) ▷ Update beta (best value so far)
31:            if alpha ≥ beta then ▷ Alpha cutoff condition
32:                break ▷ Prune the branch
33:            end if
34:        end for
35:        return value ▷ Return the best value found
36:    end if
37: end function
```

Algorithm B.3 Negamax Algorithm

```
1: function Negamax(node, depth, alpha, beta, color)
2:     if depth = 0 or node is terminal then ▷ Base case: leaf node or max depth reached
3:         return color × Evaluate(node) ▷ Return evaluation adjusted by perspective
4:     end if ▷ Tree search
5:     value ← −∞ ▷ Initialize value
6:     for all child in Children(node) do ▷ Iterate over children
7:         temp ← -Negamax(child, depth-1, -beta, -alpha, -color) ▷ Recursive call with inverted
   parameters
8:         value ← max(value, temp) ▷ Select best value
9:         alpha ← max(alpha, value) ▷ Update alpha
10:        if alpha ≥ beta then
11:            break ▷ Beta cutoff
12:        end if
13:    end for
14:    return value ▷ Return best value found
15: end function
```

Algorithm B.4 Alpha-beta pruning with Transposition Tables

```
1: function AlphaBetaTT(node, depth, alpha, beta, maximizingPlayer, transpositionTable)
2:     hash ← Hash(node)                                              ▷ Generate unique hash
3:     if TranspositionTableContains(transpositionTable, hash) then   ▷ Check if node exists
4:         entry ← GetEntry(transpositionTable, hash)                 ▷ Retrieve entry
5:         if entry.depth ≥ depth then                                ▷ Sufficient depth?
6:             if entry.flag = EXACT then                             ▷ Exact value
7:                 return entry.value                                 ▷ Return stored value
8:             else if entry.flag = LOWERBOUND then                   ▷ Lower bound
9:                 alpha ← max(alpha, entry.value)                    ▷ Update alpha
10:            else if entry.flag = UPPERBOUND then                   ▷ Upper bound
11:                beta ← min(beta, entry.value)                      ▷ Update beta
12:            end if
13:            if alpha ≥ beta then                                   ▷ Cutoff
14:                return entry.value
15:            end if
16:        end if
17:    end if
18:    if depth = 0 or node is terminal then                          ▷ Base case
19:        value ← Evaluate(node)                                     ▷ Evaluate node
20:        StoreInTranspositionTable(transpositionTable, hash, value, depth, EXACT) ▷ Store value
21:        return value
22:    end if
23:    if maximizingPlayer then                                       ▷ Maximizing turn
24:        value ← −∞
25:        for all child in Children(node) do
26:            temp ← AlphaBetaTT(child, depth-1, alpha, beta, False, transpositionTable)
27:            value ← max(value, temp)
28:            alpha ← max(alpha, value)
29:            if alpha ≥ beta then                                   ▷ Beta cutoff
30:                break
31:            end if
32:        end for
33:    else                                                           ▷ Minimizing turn
34:        value ← ∞
35:        for all child in Children(node) do
36:            temp ← AlphaBetaTT(child, depth-1, alpha, beta, True, transpositionTable)
37:            value ← min(value, temp)
38:            beta ← min(beta, value)
39:            if alpha ≥ beta then                                   ▷ Alpha cutoff
40:                break
41:            end if
42:        end for
43:    end if
44:    if value ≤ alpha then                                          ▷ Set flag
45:        flag ← UPPERBOUND
46:    else if value ≥ beta then
47:        flag ← LOWERBOUND
48:    else
49:        flag ← EXACT
50:    end if
51:    StoreInTranspositionTable(transpositionTable, hash, value, depth, flag)  ▷ Store value
52:    return value
53: end function
```

Bibliography

[1] Chess Programming Contributors. Chess programming wiki. https://www.chessprogramming.org/Main_Page, 2023. Accessed: 2023-10-01.

[2] Ernst Zermelo. Über eine anwendung der mengenlehre auf die theorie des schachspiels. *Proceedings of the Fifth International Congress of Mathematicians*, 2:501–504, 1913.