Bachelor Thesis

Game-theoretic Analysis of Optimal Blockchain Mining Strategies: The Computational Perspective

Department of Economics (Econ)
University of Zurich

Prof. Dr. Christian Ewerhart

Author	Kenan Öztürk
	kenan.oeztuerk@uzh.ch
	Albisstrasse 52
	6312 Steinhausen
Matr. Number	18-922-955
Field of Study	Banking and Finance
Submitted	24.07.2023

1 Abstract

This thesis studies finite blockchain games as introduced by Ewerhart (2020). Our analysis is restricted to games with two miners and seven or fewer stages. We draw the game tree for the two-stage game and show how quickly the game tree and the number of possible blockchains grow with each additional stage of the game. Furthermore, we explain how one can find the subgame-perfect equilibrium using backwards induction. Finally, we examine five conjectures using computer simulations and verify their validity for games with two miners and seven or fewer stages. We find that the unique subgame-perfect equilibrium results in a blockchain without any forks, that miners never switch to a shorter chain, that miners never switch away from the branch they won their first block on, that payoff-maximizing miners behave monotonically, and that coordination is never necessary to play a Nash equilibrium in pure strategies in any given stage.

2 Introduction

The idea of a blockchain has been around for a number of years. According to Sherman et al. (2019) a blockchain-like system was first proposed by Chaum (1979). However, it was Nakamoto (2008) whose influential Bitcoin whitepaper introduced blockchain technology to a wider (mainstream) audience.

There are a number of works related to the Bitcoin protocol and the games it gives rise to. To begin, we want to give a quick overview of the literature in this field that is most relevant to our work specifically. When we talk about equilibria in blockchain games we (and the game-theoretic literature as a whole) refer to the Nash equilibrium Nash (1950). In blockchain games, we have at least two miners who seek to extend the blockchain and earn block rewards by doing so. A set of actions where no miner wants to unilaterally deviate from their chosen action constitutes a Nash equilibrium. There is no consensus on what the equilibria in blockchain mining games look like. Depending on the model, there can be equilibria with forks of the blockchain, equilibria without forks, or no equilibria at all.

Carlsten et al. (2016) examine Bitcoin's protocol from a practical perspective. Their key insight is that in absence of block rewards, there exist situations where there is no equilibrium at all. This result is reinforced by Eyal and Sirer (2018) who come to a similar conclusion. According to them, Bitcoin's protocol does not constitute an equilibrium and its mining incentives do not incentivize honest behaviour by the miners.

Biais et al. (2019) offer a formal game-theoretic model of Nakamoto's Proof-of-Work blockchain protocol. They model the protocol as a stochastic game with infinite horizon and find that this blockchain game has multiple equilibria, and that there exist equilibria with forks. Additionally, they show that miners benefit from coordinating on a single chain without forks, but that coordination can also lead to abandoning portions of the blockchain.

Ewerhart (2020) approaches blockchains from a different angle. Inspired by Biais et al. (2019), he constructs a game-theoretic model of a blockchain game that makes a

number of simplifying assumptions, such as a finite horizon and the absence of mining effort, but which allows for straight-forward analysis. The proposed model can easily be understood and shows how a small set of rules can generate stable consensus formation in blockchains.

Kroll et al. (2013) study Bitcoin's mining mechanism and introduce a useful concept called monotonicity. They find that there is an equilibrium where all miners behave consistently with Bitcoin's reference implementation, but that there are also infinitely many equilibria where they behave otherwise. We will examine the concept of monotonicity closely, as it can be useful when applied to Ewerhart's model. If payoff-maximizing mining in Ewerhart's model is monotonic, then other statements about finite blockchain games can potentially be proven using that property.

The literature concerns itself mostly with blockchain games that have no defined end, i.e. they have an infinite time horizon. There exists little research related to finite blockchain games where there is a clearly defined end point. We seek to improve our understanding of this niche with this humble contribution.

We limit ourselves to Ewerhart's model and attempt to analyse it first theoretically and then computationally. The theoretical part mainly serves our understanding of the model, while the goal of the computational analysis is to discover new features of the game. We utilize the concept of subgame-perfection which was first introduced by Selten (1965). Our analysis is restricted to subgame-perfect equilibria in degenerate behavioural strategies of a game with $n = 2$ miners and fewer than $T = 7$ stages only. The reason for this is that the game expands rapidly in complexity, making a computational analysis infeasible for larger T or n. To see why, we present formulas that describe exactly how fast the number of intermediate nodes $\vartheta_T(n)$ of the game-tree and the number of possible blockchains $\xi_T(n)$ grow with the time horizon T and the number of miners n in a finite blockchain game. We answer the following questions: First, what does the equilibrium in a finite blockchain game look like? Second, will miners ever switch to a shorter chain? And third, how does a miner behave after winning their first block? These questions are all related, and as we will see, we stumble upon additional questions

during our analysis and answer them directly: Is payoff-maximizing mining monotonic in a game where all miners maximize their payoffs? And is coordination ever necessary to play a Nash equilibrium in any given stage?

In contrast to the model proposed by Biais et al. (2019), which produces equilibria with forks, we find that the subgame-perfect equilibrium does not lead to forks in Ewerhart's model. Additionally, we found that the payoff-maximizing strategy is monotonic, that a miner will continue mining on the same branch they won their first block on for the rest of the game, that a miner will never switch to a shorter chain than the one they are already mining on, and that coordination between the miners is never necessary to play a Nash equilibrium in pure strategies in any given stage. We were only able to confirm these conjectures for games with less than seven stages and cannot say anything about games with more than seven stages, because of the rise in complexity every additional stage brings. We used computer simulations to test our conjectures.

The rest of this document is structured as follows. We start with the theoretical part by talking about the game trees of finite blockchain games in Section 3 and subsequently show how we can find the subgame-perfect equilibrium of a finite blockchain game with two stages in Section 4.1. Then, we continue by looking at longer games where the help of a computer is useful. We look at three and four-stage games in detail in Section 4.2 and Section 4.3, and subsequently present the subgame-perfect equilibrium for longer games of up to seven stages in Section 5. Next, we provide an overview of our computer code in Section 6. And finally, we present our conjectures and discuss our findings in Section 7 and conclude in Section 8.

3 Game Trees

To begin, we assume that the reader is familiar with Ewerhart's work about finite blockchain games, since we make use of the same notation, definitions, and conventions laid out therein. Nonetheless, we want to provide the reader with a quick refresher of the material and its most important features.

Ewerhart (2020) defines a blockchain as follows: A blockchain $\mathbb{B}$ consists of three components. First, a sequence of blocks $B = \{b_0, b_1, \ldots, b_T\}$ where block b_t was created in stage $t \in \{0, 1, \ldots, T\}$. Second, a parent-child relation $\Leftarrow$ on B. And third, an assignment map $\iota : B \backslash \{b_0\} \to N$, which assigns every block to a miner, where $N = \{1, 2, \ldots, n\}$ is the set of miners. A blockchain where the most recent block is b_t is denoted by $\mathbb{B}_t$, we will use this notation later.

A finite blockchain game as proposed by Ewerhart (2020) has a finite horizon, i.e. it ends after a predetermined number of stages T. There are n players, also referred to as miners, who are rewarded for blocks they mine that are part of the longest chain after the game ends. They receive the payoff at the end of the game and each block yields a payoff of 1. Furthermore, we assume that the blockchain grows in a stochastic manner, i.e. each miner wins any given stage $t \in \{1, 2, \ldots, T\}$ with equal probability. The genesis block is generated in the beginning and has no parent block and no block reward associated with it. Every intermediate stage t follows the same pattern. First, all miners $i \in N$ simultaneously choose a block $\widehat{b}_{t-1}(i) \in B_{t-1}$, where $B_{t-1} = \{b_0, b_1, \ldots, b_{t-1}\}$ is the existing sequence of blocks. The block $\widehat{b}_{t-1}(i)$ is where miner i appends the next block b_t in the event that they win the current stage t. Next, Nature picks the winner i_t^* of stage t at random and with equal probability $\frac{1}{n}$. To conclude stage t, the new block b_t is appended to the winner's specified parent block $\widehat{b}_{t-1}(i_t^*)$.

Ewerhart's model assumes that mining comes at no cost and that mining does not require any computing power or mining effort. There are models that include mining effort and costs, a prime example of such a model was proposed by Dimitri (2017). Furthermore, it is important to note that Ewerhart's model describes a game with complete

information. All miners know what the Blockchain looks like and no miner can hide or refuse to publish a block they mined. This simplifies the analysis considerably. Kiayias et al. (2016) loosened this restriction partially by allowing miners to hide their block. However, miners still have to announce that they mined a block, they cannot keep it a secret.

Let us continue by looking at the game tree in order to better understand the game we are studying. Game trees are useful for our purposes because they allow us to visualize equilibrium paths. They aid in our understanding. We did not find any sources that draw the game tree explicitly like we do here.

We are looking at a multistage game, where every stage-game is a game of imperfect information (since all players move simultaneously). Each stage starts with simultaneous moves by miner 1 and miner 2 and ends with a move by Nature $\mathcal{N}$, who picks the winner of that particular stage. At the very end of the game, after the final stage's winner is chosen, Nature moves again, picking one of the longest chains at random and with equal probability.

In Figure 1 we can see the game-tree of a finite blockchain game with at least three stages. The first stage is trivial. Both miners only have one strategy, they have no choice but to mine on b_0. The stage ends after Nature picks a winner at random and with equal probability of $\frac{1}{2}$.

In the second stage miners have two actions to choose from. They can mine on b_0 or b_1. Both miners have two information sets in this stage. The second miner's information sets span two nodes each. Both miners know who won the previous stage (and what the winner's strategy was), but they do not know where the other miner mines in the current stage.

The tree in Figure 1 is incomplete. At every large dashed circular node, another subgame starts (not to be confused with a stage-game). They are omitted for simplicity.

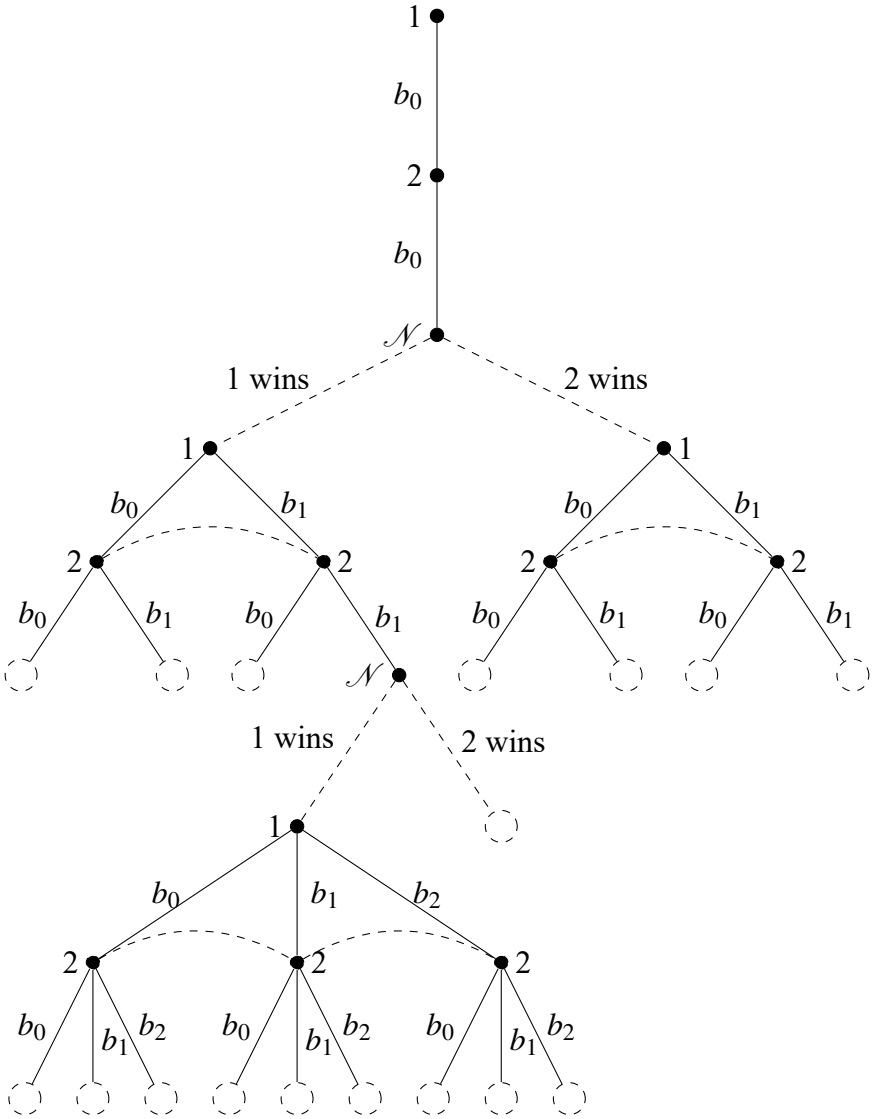

Figure 1 – A game-tree for a game with at least three stages.

An intermediate node in our context is a node in the game tree of a finite blockchain game that sits at the end of an intermediate stage t. Every stage-game ends in an intermediate node. When an intermediate node is reached, the next stage-game begins and all players make their simultaneous moves. After intermediate stage T (the very last stage), Nature moves once more and picks one of the longest chains. This step is always omitted in the game trees, but must be considered in the payoff calculations. There are two intermediate nodes after the first stage of a finite blockchain game with T stages and $n = 2$ miners.

It is evident from looking at Figure 1 that the trees explode in size very quickly. But how quickly exactly? First consider the fact that the number of actions a miner can choose from in any given intermediate stage t is equal to t. In the first stage there is only one action (mine on b_0), in the second stage there are two actions (mine on b_0 or b_1) et cetera.

Consider a finite blockchain game with T stages and n miners and denote the number of intermediate nodes in the game tree after the last intermediate stage T by a function $\vartheta_T(n)$. By looking at the general game tree in Figure 1 we can see that the number of intermediate nodes after the final stage $\vartheta_T(n)$ must be equal to the number of intermediate nodes after the previous stage $\vartheta_{T-1}(n)$ multiplied by the factor $T^n \cdot n$. Why? Recall that all n miners can choose from T actions in stage T, which means that there are T^n possible strategy profiles for this stage. Furthermore, Nature picks one winner amongst the n miners, which means we must also multiply by n. Note that $\vartheta_T(n)$ is the number of nodes before Nature picks one of the longest chains as the winner.

After the very fist stage there are n intermediate nodes because miners only have one strategy they can play and Nature picks one of them as the winner. This allows us to define $\vartheta_T(n)$ recursively.

$$\vartheta_1(n) = n \qquad \text{and} \qquad \vartheta_t(n) = \vartheta_{t-1}(n) \cdot t^n \cdot n$$

We can rewrite this as follows:

$$\vartheta_T(n) = \prod_{t=1}^{T} (t^n \cdot n) = n^T \cdot (T!)^n$$

In Table I below we count the terminal nodes $\frac{1}{n}\vartheta_T(n)$ of the game tree. We cut the game tree before Nature picks the winner of the final stage. That number is half the number of nodes after Nature moves, because we have two miners. This explains the factor $\frac{1}{2}$. Why do we omit Nature's last moves? Nature's last moves represent simple lotteries and can be condensed into their expected values for both miners. These are the value the miners base their decisions on and the values which appear in the payoff-matrices in later sections.

Stages	$\frac{1}{2}\vartheta_T(2)$	$\xi_T(2)$
$T=1$	1	2
$T=2$	8	8
$T=3$	144	48
$T=4$	4'608	384
$T=5$	230'400	3'840
$T=6$	16'588'800	46'080
$T=7$	1'625'702'400	645'120
$\vdots$	$\vdots$	$\vdots$

Table I – Number of terminal nodes after T stages $\frac{1}{n}\vartheta_T(n)$ and the number of unique blockchains $\xi_T(n)$ for a blockchain after a game with T stages and $n=2$ miners.

Note that $\vartheta_T(n)$ refers to the number of intermediate nodes in the game tree after T intermediate stages, not to the number of possible blockchains at the end of the game. Let us take a look how we arrive at that number, $\xi_T(n)$, which behaves tamer than $\vartheta_T(n)$. We proceed as before and consider a game with T stages and n miners. The number of unique blockchains at the end of the game $\xi_T(n)$ is equal to the number of unique blockchains after the previous stage $\xi_{T-1}(n)$ multiplied by the factor $T\cdot n$. Why? For every unique blockchain after the previous stage $T-1$ we can append a child block at T different parent blocks and have n possible winners.

After the very first stage there are n possible blockchains, which all have two blocks (b_0 and b_1) and only differ in the winner of b_1. We can now define $\xi_T(n)$ recursively.

$$\xi_1(n)=n \qquad \text{and} \qquad \xi_t(n)=\xi_{t-1}(n)\ \cdot t\cdot n$$

This can be rewritten as:

$$\xi_T(n)=\prod_{t=1}^{T}(t\cdot n)\ =\ n^T\cdot T!$$

Why are we interested in ϑ_T and ξ_T? We want to know how far we can go with our computational analysis, how many stages the game can have before the calculation becomes prohibitively expensive. The purpose of Table I is to show that the game explodes extraordinarily rapidly in complexity. We are especially interested in $\frac{1}{2}\vartheta_T$ because this is equivalent to the number of paths a computer program has to check to test the conjectures in Section 7. Our limiting factor for the computational analysis is computing power. More about this can be found in Section 7.6.

The functions ϑ_T and ξ_T are related. It holds that $\vartheta_T \geq \xi_T$ for any number of miners n and any number of stages T. Why is ϑ_T larger? There exist some pair(s) of intermediate nodes which result in the same blockchain. To see this, please take a look at Example 1.

Before we turn our attention to the example, we first have to explain what we mean when we talk about strategy profiles in finite blockchain games. Let us consider a finite blockchain game with T stages and $n = 2$ miners. A strategy profile s_t in pure strategies for any given intermediate stage t has the following form:

$$s_t = (\widehat{b}_{t-1}(1), \widehat{b}_{t-1}(2))$$

Example 1. *Consider a two-stage game with two miners, where miner* 1 *won the first stage. In the second stage, the strategy profiles* (b_1, b_0) *and* (b_1, b_1) *result in the same blockchain* $\mathbb{B}_2$ *when miner* 1 *wins the second stage. In both cases* $\mathbb{B}_2$ *consists of a sequence of blocks* $\{b_0, b_1, b_2\}$ *with* $b_0 \Leftarrow b_1$ *and* $b_1 \Leftarrow b_2$ *and* $\iota(b_1) = \iota(b_2) = 1$. *This is possible because miner* 2*'s strategy has no effect on the blockchain when they lose the stage. They do not get to append a block of their own. We have two different intermediate nodes after* 2 *stages, each with their unique path through the game tree, that result in the same blockchain that does not have any forks and where miner* 1 *won both blocks* b_1 *and* b_2. *Looking at Figure 2 can be helpful when trying to understand this example, but be aware that we omit Nature's decision in the second stage.*

4 Short Games

4.1 Two-Stage Game

When we say T-stage game, we mean a finite blockchain game with T stages. A two-stage game ends after $T = 2$ stages.

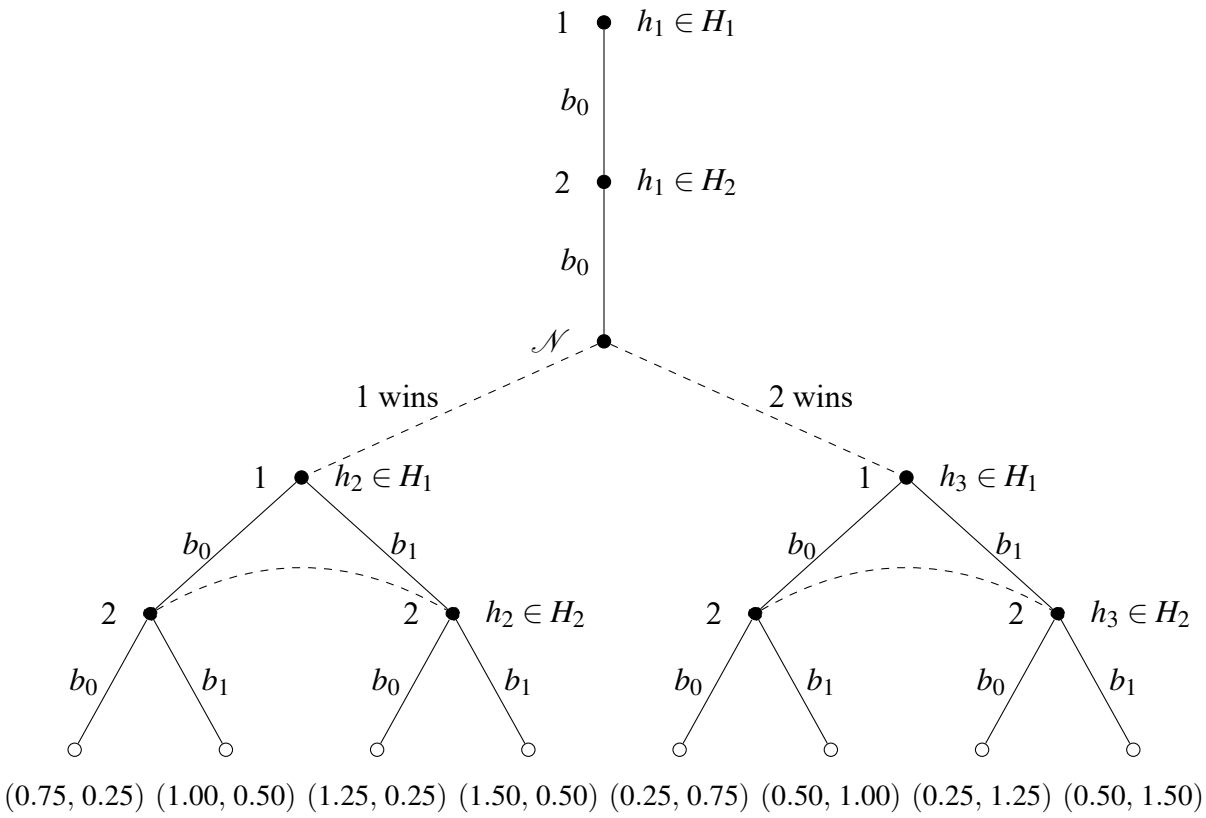

Figure 2 – The game-tree for the two-stage game.

4.1.1 Payoffs

We often talk about how the miners maximize their payoffs, this means that they are maximizing their *expected* payoff. Finite blockchain games are inherently uncertain and payoffs therefore only exist in expectation (unless the game has ended). We consider payoff-maximizing behaviour to be optimal. It is important to note that this payoff-maximizing behaviour automatically results in a subgame-perfect equilibrium, as we will see later.

Consider a finite blockchain game with T stages and $n = 2$ miners. For any T, there exists only one strategy in the first stage $t = 1$ (mine on b_0). Hence, there exists only one strategy profile that can be played, which is (b_0, b_0).

We can assume w.l.o.g. that miner 1 wins the first stage. If miner 1 does not win the first stage, we may swap the indices of the miners. Hence, there is only one possible blockchain in stage $t = 2$ (Figure 3). This is equivalent to only looking at the left side of the game trees in Figure 1 or Figure 2.

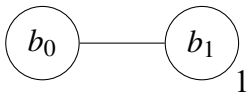

Figure 3 – The only possible the blockchain after the first stage of a finite blockchain game with $n = 2$ miners. We assume w.l.o.g. that miner 1 wins the first stage.

Now consider the final stage of the two-stage game. Therefore, let $T = 2$ and $t = 2$. The Nash equilibrium strategy profile in the last stage of the game is given by (b_1, b_1). This is a strictly dominant strategy equilibrium. To show how this Nash equilibrium is found, we proceed by explaining how the payoff-matrix (Table II) is constructed. All payoffs were mathematically rounded to two decimal places and the Nash equilibria are marked with round brackets, this is true for all payoff-matrices herein.

		Miner 2	
		b_0	b_1
Miner 1	b_0	0.75, 0.25	1.00, 0.50
	b_1	1.25, 0.25	(1.50, 0.50)

Table II – Payoff matrix in the final stage of a two-stage game. The Nash equilibrium is marked with round brackets. (Generated in Appendix B.1.6)

Assume that miner 1 decides to mine on b_1 while miner 2 decides to mine on b_0. We

are looking at the bottom left quadrant of the payoff-matrix (Table II). In this case we are faced with two possibilities. Either miner 1 wins, resulting in the blockchain (a), or miner 2 wins, resulting in the blockchain (d) at the end of the game. Figure 4 shows the blockchains (a) and (d). Miner 1 receives a payoff of 2.00 in blockchain (a), and a payoff of

$$\frac{1}{l} \cdot 1.00 + \frac{1}{l} \cdot 0.00 = 0.50$$

in blockchain (d), where $l = 2$ is the number of the longest chains in that blockchain. Recall that in the case where there are multiple longest chains at the end of the game, one is chosen with equal probability $\frac{1}{l}$. At the end of the game, miner 1's expected payoff is

$$\frac{1}{n} \cdot 2.00 + \frac{1}{n} \cdot 0.50 = 1.25$$

Recall that $\frac{1}{n}$ is the probability of winning the stage, since each miner wins with equal probability. Miner 2 receives a payoff of 0 in blockchain (a), and a payoff of 0.50 in blockchain (d). At the end of the game, miner 2's expected payoff therefore is 0.25. The other entries of the payoff-matrix at the end of the two-stage game (Table II) are computed completely analogously. Their derivation is hence omitted.

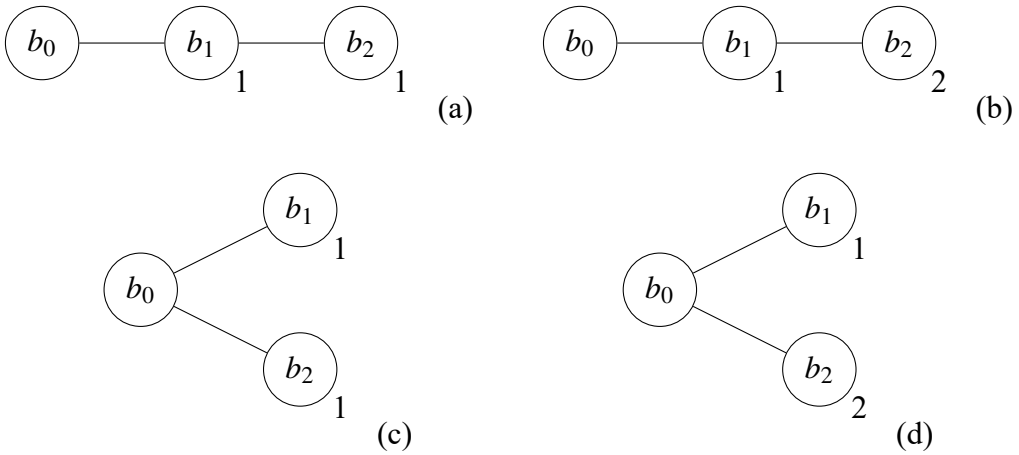

Figure 4 – Four possible situations after the second stage of a finite blockchain game with $n = 2$ miners.

4.1.2 Subgame-Perfect Equilibrium

The concept of subgame-perfection as defined by Selten (1965) builds on top of the concept of a Nash equilibrium and is invaluable for our analysis. In short, a Nash equilibrium is subgame-perfect if it constitutes a Nash equilibrium in every subgame.

After the last nodes in the game tree, Nature makes at least one more move, deciding the winner of the second stage as well as deciding which chain is chosen if there is more than one longest chain. However, these steps are omitted from the tree because they only make the tree (Figure 2) unnecessarily complicated. The payoffs at the terminal nodes represents the expected value of Nature's subsequent decisions. In this way the payoffs in the tree correspond to the entries of the payoff-matrix (Table II). We have $\frac{1}{2}\vartheta_2(2) = 8$ terminal nodes in this tree.

Note how the left and the right side of the tree only differ in the reversal of the payoff tuples. That is the reason we can assume w.l.o.g. that miner 1 wins the first stage.

Now we have to talk about information sets and behavioural strategies. In the two-stage game there are six information sets, three for each player. We denote an information set by $h_j \in H_i$ where H_i is the set of all information sets for miner i. All information sets are visible in Figure 2. Please note that we abuse the notation slightly and take advantage of the fact that the game is symmetrical. The information sets for both miners are equivalent, because they move simultaneously and only differ in their index. This is why we do not differentiate between $h_1 \in H_1$ and $h_1 \in H_2$, they are equivalent.

Next, we want to characterize the subgame-perfect equilibrium by writing down the subgame-perfect behavioural strategy profiles s_i^* for both players $i = 1, 2$. A behavioural strategy profile specifies an independent probability distribution over a player's actions in every information set $h_j \in H_i$. In our case these probability distributions are always degenerate along the subgame-perfect equilibrium path, i.e. players never mix between actions in any information set (at least for games with less than seven stages, this follows from Conjecture 1). Consequently, to keep things tidy, we only write down the strategy which is chosen with a probability of 1, e.g. we write $s_i^*(h_j) = b_t$ where b_t is miner i's

chosen strategy in information set h_j. The subgame-perfect Nash equilibrium is

$$\begin{aligned} s^* &= \{s_1^*(H_1),\, s_2^*(H_2)\} \\ &= \{\big(s_1^*(h_1),\, s_1^*(h_2),\, s_1^*(h_3)\big),\ \big(s_2^*(h_1),\, s_2^*(h_2),\, s_2^*(h_3)\big)\} \\ &= \{(b_0,\, b_1,\, b_1),\ (b_0,\, b_1,\, b_1)\} \end{aligned}$$

where both players play b_0 in the first stage and b_1 in the second (no matter who wins the first stage). Remember that the individual stages are games of imperfect information. In the second stage this means that miner 2 cannot make their strategy dependent on miner 1's strategy *in the same stage* (and vice versa), because both players move simultaneously.

The payoff-maximizing strategy is straight forward in the two-stage game. All miners mine on b_0 in the first stage and on b_1 in the second stage. Nonetheless, the path through the game-tree is stochastic and depends on Nature's decisions.

Since the game explodes in complexity very quickly, writing down the subgame-perfect Nash equilibrium like this becomes impractical for the four-stage game at the latest. Conveniently, saying that the strategy profile (b_{t-1}, b_{t-1}) is played in every stage t (regardless of Nature's decisions) is equivalent. It is equivalent because s^* says exactly that, that (b_0, b_0) is played in the first stage and that (b_1, b_1) is played in the second stage, no matter what.

Hence, for the sake of simplicity, we characterize the subgame-perfect Nash equilibrium by saying that each player mines on the most recent block, i.e. that the subgame-perfect strategy profiles for every stage t are given by (b_{t-1}, b_{t-1}). As we will see, this subgame-perfect equilibrium is *unique* for games with less than $T = 7$ stages. We do not know about games with $T \geq 7$ stages yet, although we strongly suspect that the pattern will continue forever.

To end this section, we would like to address a concern some astute readers may have. In games of imperfect information, backwards induction may not always result in a subgame-perfect equilibrium in *pure strategies*. To see this, consider Table XIII in Sec-

tion 7.5. In such a situation, coordination is necessary to play a Nash equilibrium in pure strategies. Consequently, the existence of a Nash equilibrium in mixed strategies cannot be ruled out. That such a situation does not exist for the finite blockchain games we study is shown by Conjecture 5. Furthermore, there always exists a Nash equilibrium in pure strategies in any given (on-path) situation we look at. Not only that, but the equilibrium in any given (on-path) situation is also always one in strictly dominant strategies, i.e. it is unique. This is shown by Conjecture 1. The game is well-behaved in that way. However, off-path there exist situations where there is no strictly dominant strategy. Please have a look at the payoff-matrix in Table V for such an off-path situation in the three-stage game. We will inspect that table more closely in the next section, Section 4.2.

4.2 Three-Stage Game

Next, we turn our attention to a three-stage game. We show how to derive the payoff-matrix in intermediate stage $t = 2$ by the means of backwards induction. Assume again w.l.o.g. that miner 1 wins the first stage.

4.2.1 Final Stage

Let $T = 3$ and $t = 3$. Figure 4 shows some possible blockchains at the beginning of stage $t = 3$. Some, as we will see, can never be reached if both players maximize their payoffs. When the Nash equilibrium was played in stage $t = 2$, only blockchains (a) and (b) are possible.

Table III, Table IV, Table V and Table VI show the payoff-matrices for the cases (a), (b), (c) and (d) respectively. The entries of the payoff-matrices are calculated analogously to the previous chapter. The only difference is, that we now have three instead of two strategies for each miner.

		Miner 2		
		b_0	b_1	b_2
Miner 1	b_0	2.00, 0.00	1.75, 0.25	2.00, 0.50
	b_1	2.00, 0.00	1.75, 0.25	2.00, 0.50
	b_2	2.50, 0.00	2.25, 0.25	(2.50, 0.50)

Table III – Payoff-matrix for blockchain (a) in Figure 4 in the three-stage game. (Generated in Appendix B.1.6)

		Miner 2		
		b_0	b_1	b_2
Miner 1	b_0	1.00, 1.00	1.00, 1.00	1.00, 1.50
	b_1	1.25, 0.75	1.25, 0.75	1.25, 1.25
	b_2	1.50, 1.00	1.50, 1.00	(1.50, 1.50)

Table IV – Payoff-matrix for blockchain (b) in Figure 4 in the three-stage game. (Generated in Appendix B.1.6)

		Miner 2		
		b_0	b_1	b_2
Miner 1	b_0	0.83, 0.17	1.00, 0.50	1.00, 0.50
	b_1	1.33, 0.17	(1.50, 0.50)	(1.50, 0.50)
	b_2	1.33, 0.17	(1.50, 0.50)	(1.50, 0.50)

Table V – Payoff-matrix for blockchain (c) in Figure 4 in the three-stage game. (Generated in Appendix B.1.6)

		Miner 2		
		b_0	b_1	b_2
Miner 1	b_0	0.50, 0.50	0.83, 0.67	0.33, 1.17
	b_1	1.17, 0.33	1.50, 0.50	(1.00, 1.00)
	b_2	0.67, 0.83	1.00, 1.00	0.50, 1.50

Table VI – Payoff-matrix for blockchain (d) in Figure 4 in the three-stage game. (Generated in Appendix B.1.6)

At this point we want to take a closer look at Table V. What is interesting about this table is the fact that there is no Nash equilibrium in strictly dominant strategies. Once we consider blockchain (c) in Figure 4 however, we can see why. Case (c) represents a symmetrical blockchain where miner 1 won both the first and second stage, and appended both blocks to the genesis block b_0. Both miners have to be indifferent between mining on b_1 or b_2, because the situation is symmetrical.

4.2.2 Second Stage

Let $T = 3$ and $t = 2$. Much like the two-stage game, the unique Nash equilibrium is given by (b_1, b_1). To show how this Nash equilibrium is found, we proceed by explaining how to use backwards induction to calculate the entries in the payoff-matrix in the second stage (Table VII).

		Miner 2	
		b_0	b_1
Miner 1	b_0	1.25, 0.75	1.50, 1.00
	b_1	1.75, 0.75	(2.00, 1.00)

Table VII – Payoff-matrix in intermediate stage $t = 2$ in the three-stage game. (Generated in Appendix B.1.6)

Assume that miner 1 decides to mine on b_0 while miner 2 decides to mine on b_1. We

are looking at the top right quadrant of the payoff-matrix (Table VII). In this case we are faced with two possibilities. Either miner 1 wins, resulting in the blockchain (c), or miner 2 wins, resulting in the blockchain (b) for the last stage of the game (stage $t = 3$).

From our analysis of stage $t = 3$ we know what the payoffs will be when the game reaches that stage. We can assume that one of the Nash equilibria will be played. If miner 1 wins the second stage (they append a block to b_0), then both miners' payoffs in the third stage are given by one of the Nash equilibrium payoffs in Table V. Conveniently, they are all the same, which means that no matter which Nash equilibrium is played, miner 1 and miner 2 receive expected payoffs of 1.50 and 0.50 respectively. If miner 2 wins the second stage (they append a block to b_1), then the miner's payoffs in the third stage are given by the unique Nash equilibrium payoffs in Table IV. In this case miner 1 and miner 2 both receive an expected payoff of 1.50.

We can now compute the expected payoffs in the second stage. Miner 1 receives 1.50 in any case and miner 2 receives 1.50 or 0.50 each with probability $\frac{1}{n}$. We yield payoffs of 1.50 for miner 1 and 1.00 for miner 2 in stage $t = 2$ when miner 1 mines on b_0 and miner 2 mines on b_1. All other entries of the payoff-matrix are calculated completely analogously and their calculation is hence omitted.

Finding the subgame-perfect equilibrium path is straight forward. In every stage we know what each miner will do. Hence, we can simulate a playthrough stage by stage. In stage $t = 1$ both miners mine on b_0 and in stage $t = 2$ both miners mine on b_1 (which can be seen in Table VII). In the third stage we are either in blockchains (a) or (b). In both cases, both miners will mine on b_2 (which can be seen in Table III and Table IV). Therefore, there is only one subgame-perfect equilibrium path resulting in one chain without any forks. The strategy profiles in every stage are given by (b_{t-1}, b_{t-1}) along the subgame-perfect equilibrium path.

4.3 Four-Stage Game

The analysis of the four-stage game is done analogously to the analysis of the two- and three-stage games. Once again, we assume w.l.o.g. that miner 1 wins the first stage.

4.3.1 Final Stage

There are $\frac{1}{2}\xi_3(2) = 24$ different possible blockchains after stage $t = 3$, six for each of the four possible blockchains (a), (b), (c) and (d). In total there are $\xi_3(2) = 48$, but we assumed that miner 1 wins the first stage, hence we only look at half of them (the left side of the game tree).

Please be aware that some blockchains are strategically equivalent and only differ in the order the blocks were mined in, i.e. they are the same except for the block indices. In Figure 5 below and in Figure A.1 in Appendix A we can see some blockchains in the final stage of the four-stage game.

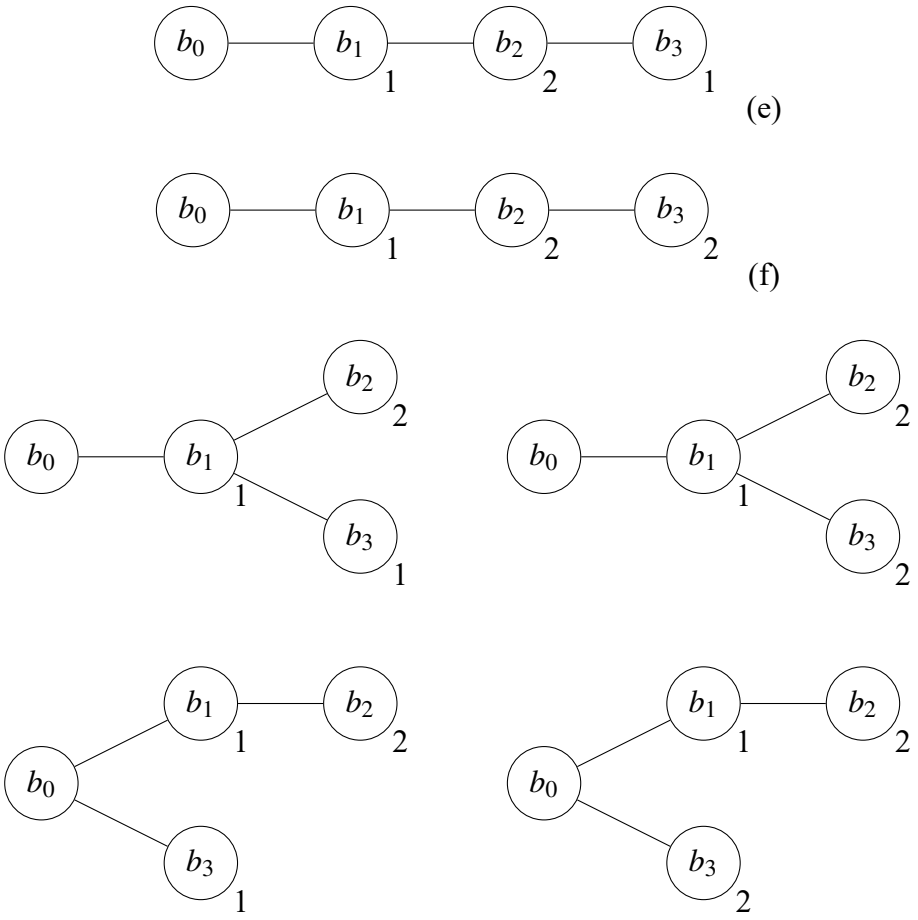

Figure 5 – Six possible situations after the third stage, when case (b) materialized after the second stage.

At this point we want to show one example of a payoff-matrix in the last stage of the four-stage game. The payoff matrix below (Table VIII) is on the subgame-perfect equilibrium path, as we will see later. There is only one Nash equilibrium and the entries are calculated in the same way as in previous sections. The matrix was generated using a computer program and then checked manually for errors.

Note that such a payoff-matrix exists for all 24 blockchains in stage $t = 4$. Doing the analysis by hand, while possible, is impractical. The game explodes rapidly in complexity.

		Miner 2			
		b_0	b_1	b_2	b_3
Miner 1	b_0	2.00, 1.00	2.00, 1.00	1.75, 1.25	2.00, 1.50
	b_1	2.00, 1.00	2.00, 1.00	1.75, 1.25	2.00, 1.50
	b_2	2.00, 1.00	2.00, 1.00	1.75, 1.25	2.00, 1.50
	b_3	2.50, 1.00	2.50, 1.00	2.25, 1.25	(2.50, 1.50)

Table VIII – Payoff-matrix for blockchain (e) in the four-stage game. (Generated in Appendix B.1.6)

		Miner 2			
		b_0	b_1	b_2	b_3
Miner 1	b_0	1.00, 2.00	1.00, 2.00	1.00, 2.00	1.00, 2.50
	b_1	1.00, 2.00	1.00, 2.00	1.00, 2.00	1.00, 2.50
	b_2	1.25, 1.75	1.25, 1.75	1.25, 1.75	1.25, 2.25
	b_3	1.50, 2.00	1.50, 2.00	1.50, 2.00	(1.50, 2.50)

Table IX – Payoff-matrix for blockchain (f) in the four-stage game. (Generated in Appendix B.1.6)

4.3.2 Third Stage

The payoff-matrix below (Table X) was generated using a computer program. Let's take a closer look at the Nash equilibrium (b_2, b_2). Both miners mine on the most recent block, resulting in either blockchain (e) or blockchain (f). In case (e) the expected payoffs in the next stage will be 2.50 for miner 1 and 1.50 for miner 2 (as seen in the Nash equilibrium in Table VIII). In case (f) the payoffs are reversed, 1.50 for miner 1 and 2.50 for miner 2 (Table IX). Recall that (e) and (f) both materialize with probability $\frac{1}{n}$, hence the expected payoff in the third stage when mining at b_2 is 2.00 for both players.

		Miner 2		
		b_0	b_1	b_2
Miner 1	b_0	1.50, 1.50	1.50, 1.50	1.50, 2.00
	b_1	1.75, 1.25	1.75, 1.25	1.75, 1.75
	b_2	2.00, 1.50	2.00, 1.50	(2.00, 2.00)

Table X – Payoff-matrix for blockchain (b) in the four-stage game. (Generated in Appendix B.1.6)

Alternatively, blockchain (a) may materialize instead of (b). In that case we have two different chains (g) and (h) in the final stage. The analysis is done in the same way. All the payoff-matrices are listed in the Appendix (Table A.1, Table A.2, and Table A.3).

4.3.3 Second Stage

The payoff-matrix below (Table XI) was generated using a computer program. We arrive at it by applying backwards induction multiple times until we arrive at the payoff-matrix for the second stage. As we can see, there is a Nash equilibrium in strictly dominant strategies (b_1, b_1).

After the second stage we have a situation with either blockchain (a) or (b). Then the game may end up in (e) or (f) when (b) materializes, or (g) or (h) when (a) materializes. The payoffs will be different depending on which blockchains the game goes through.

As explained in previous sections, we take the expected value, which can be easily computed because all probabilities are fixed and known. The payoffs for miner 1 are either 3.00 in (a) or 2.00 in (b), and the payoffs for miner 2 are either 1.00 in (a) or 2.00 in (b). Hence, we end up with expected payoffs of 2.50 and 1.50 in the second stage Nash equilibrium (Table XI). If the reader wishes to go through the calculations themselves, the payoff-matrices for blockchains (a), (g), and (h) in the four-stage game are listed in Appendix A (Figure A.1, Table A.1, Table A.2 and Table A.3).

		Miner 2	
		b_0	b_1
Miner 1	b_0	1.69, 1.19	2.00, 1.50
	b_1	2.19, 1.19	(2.50, 1.50)

Table XI – Payoff-matrix in intermediate stage $t = 2$ in the four-stage game. (Generated in Appendix B.1.6)

The subgame-perfect equilibrium path through the four-stage game is found completely analogously to the path in the three-stage game. We once again start in the first stage and simulate a playthrough while assuming that the Nash equilibrium is played in every stage. Conveniently, no situations with multiple Nash equilibria will be reached along the path of play, meaning that we once again end up with only one chain without any forks.

5 Long Games

To start, let us recapitulate. The subgame-perfect equilibrium paths through games with $T \leq 4$ stages are all similar. In the first stage of the two-stage game, the players play (b_0, b_0) and in the second stage they play (b_1, b_1). The three- and four-stage games behave in the same way. In every intermediate stage t the strategy profile (b_{t-1}, b_{t-1}) is played. Miners always mine on the most recent block.

We used a computer program to verify that games with $T = 5, 6$ and 7 stages follow the same pattern. The time it took for the program to terminate give some clue about how computationally intensive the task can get. Refer to Table XII for an overview. An average desktop PC with a 9th generation Intel i7 Processor was used for the computation.

Stages	Time to Termination
$T \leq 4$	$<$ 1 second
$T = 5$	$\approx$ 11 seconds
$T = 6$	$\approx$ 13 minutes
$T = 7$	$\approx$ 24 hours

Table XII – The time it took to verify that the equilibrium path in the $n = 2$ player game never results in forks. (Conjecture 1)

We proved computationally that all games with $T \leq 7$ stages result in a single chain without any forks. This contrasts the findings of Biais et al. (2019), who prove that there exist equilibria that exhibit forks (in their specific model).

If Ewerhart's model can exhibit equilibria with forks for games with $T > 7$ stages remains to be seen. The aid of a more powerful computer or a faster programming language is unlikely to help. It may enable us to consider two or three additional stages, but that is not useful.

Please refer to Section 6 and Appendix B for more details about our code. It is freely available on GitHub, the project's link can be found in the Appendix.

6 Computer Code

In this section we attempt to provide a broad overview of our Python code. This section is intentionally kept short. For more information please refer to the Jupyter Notebooks in Appendix B. Almost every line of code is commented, and we are confident that the reader can understand how the code works from the notebooks alone. Nonetheless, we want to go over the most important aspects of our Python code. Our code is split into six Jupyter Notebooks, each implements different aspects of the code. This split was made to bring some order to the inevitable chaos that comes with coding. We will now go over the features each notebook implements, for more details please refer to each notebook's respective section in the appendix.

First, we have the notebook `main.ipynb` (Appendix B.1), which serves as a hub for all the simulation outputs. Readers who are only interested in our final results can ignore all other notebooks and only look at this one. This notebook only runs the simulations and returns their results. Additionally, every payoff-matrix found in this document is generated in this notebook.

The notebook `blockchain.ipynb` (Appendix B.2) is the heart of the code. It implements three classes: `Block`, `Blockchain` and `ExtendedBlockchain`. The `Blockchain` class is the most important class, and we encourage the reader to pay particular attention to its attributes, as they get referenced frequently in the other notebooks.

The notebook `payoff_matrix.ipynb` (Appendix B.3) houses the game-theoretic aspects of the code. The function `intermediatePayoffMatrix()` in this notebook is crucial for all simulations. It makes use of recursion to calculate the payoff-matrix at any given intermediate node in the game tree.

Next, we have the notebook `conjectures.ipynb` (Appendix B.4). It implements the simulations we use to try and find a counter example for each conjecture. There is a lot of recursion, but we argue this solution is elegant and makes the code easier to understand.

The remaining notebooks `helper_functions.ipynb` (Appendix B.5) and `tests.ipynb` (Appendix B.6) are not directly used in the simulations. The functions in `helper_functions.ipynb` are quality-of-life functions that make the rest of the code more easily readable and the printed output cleaner. The notebook `tests.ipynb` was used a lot during development, and is therefore not central to our analysis, but it can provide some more insight into what the code is doing.

To end this section, let us consider two ways to speed up the code. Example 1 at the end of Section 3 represents an obvious optimization opportunity. Since there are multiple paths that result in the same intermediate blockchains, we do not have to spend computing power to calculate the payoff-matrices multiple times. However, we decided against this because the code is already complicated as-is, and further complications are likely to introduce errors. Another optimization opportunity is to calculate all payoff-matrices once, and storing them in memory instead of calculating them separately for each conjecture. This represents a trade-off between computing time and computer memory. And given how fast the game-tree grows, this approach will quickly result in a prohibitively large dataset.

7 Conjectures

We tried, but failed to construct formal proofs of the conjectures in this section. However, we verified all conjectures to be true for finite blockchain games with $T \leq 7$ stages and $n = 2$ miners.

7.1 Subgame-Perfect Equilibrium

Conjecture 1. *The strategy profile (b_{t-1}, b_{t-1}) is played in every stage in the subgame-perfect equilibrium of a finite blockchain game with any number of stages T and $n = 2$ miners. Consequently, equilibria in randomized actions need not be considered along the subgame-perfect equilibrium path, and the subgame-perfect equilibrium is unique.*

If Conjecture 1 is true, then the subgame-perfect equilibrium will result in a blockchain without forks. To see this, please consider Proposition 1.

Proposition 1. *If the strategy profile (b_{t-1}, b_{t-1}) is played in every stage of a finite blockchain game with T stages and $n = 2$ miners, then the resulting blockchain $\mathbb{B}_T$ at the end of the game does not exhibit any forks.*

Proof. Assume that the players play (b_{t-1}, b_{t-1}) in every stage t. After the first stage we will have a sequence of blocks $B_1 = \{b_0, b_1\}$ with $b_0 \Lsh b_1$, because (b_0, b_0) was played in stage $t = 1$. After the second stage we will have a sequence of blocks $B_2 = \{b_0, b_1, b_2\}$ with $b_0 \Lsh b_1$ and $b_1 \Lsh b_2$, because (b_1, b_1) was played in the stage $t = 2$. We can repeat this argument until we reach stage $t = T$, the end of the game. Therefore, this pattern must continue for all remaining stages $t \in \{3, \ldots, T\}$. This concludes the proof. □

This conjecture is tested in Appendix B.1.1.

7.2 Monotonicity

Monotonicity as defined by Kroll et al. (2013) is used to describe the behaviour of blockchain mining strategies. Consider an arbitrary situation where miner i's strategy in

stage t is to mine on block $\widehat{b}_{t-1}(i)$. Miner i's strategy is monotonic if and only if miner i always mines on b_t in the next stage $t+1$ as long as $\widehat{b}_{t-1}(i) \Leftarrow b_t$. In other words, if a monotonic strategy tries to extend the blockchain at some block $\widehat{b}_{t-1}(i)$, then the addition of a new block b_t on $\widehat{b}_{t-1}(i)$ will cause it to mine on b_t instead.

Conjecture 2. *The payoff-maximizing strategy in a finite blockchain game with $n = 2$ payoff-maximizing miners is monotonic for any horizon T.*

Proving Conjecture 2 would be very useful because it is a strong result. We argue that some valuable insights follow from it. We know that Conjecture 2 implies Conjecture 1. To see this, please consider Proposition 2.

Proposition 2. *Conjecture 2 implies Conjecture 1. In other words, if all players behave consistent with monotonicity, then the strategy profile (b_{t-1}, b_{t-1}) will be played in every stage t.*

Proof. This follows directly from the definition of monotonicity. Assume that all players behave monotonically. We know that the strategy profile (b_0, b_0) is always played in the stage $t = 1$, regardless of player's strategies. Now we have a situation where both miners sought to extend the blockchain at b_0 in stage $t = 1$, and b_1 was appended to b_0 in that stage, i.e. $b_0 \Leftarrow b_1$. Because all miners behave monotonically, their strategy is now to mine on b_1, i.e. the strategy profile (b_1, b_1) is played in stage $t = 2$. This argument can be repeated for all remaining stage $t \in \{3, \ldots, T\}$. Therefore, the strategy profile (b_{t-1}, b_{t-1}) is played in every stage. □

Conjecture 2 sounds intuitive, but writing a formal proof is hard. The complexity stems from the fact that strategies in the subgame-perfect equilibrium are interdependent. Writing a proof for monotonicity thus requires showing that behaving monotonically is mutually optimal for all miners in any situation. We suspect that we do not know enough about the game to prove that yet.

This conjecture is tested in Appendix B.1.2.

7.3 A Miner's Behaviour after Winning their First Block

Conjecture 3. *Consider a finite blockchain game with T stages and $n = 2$ payoff-maximizing miners. Assume that miner i has not won any blocks before stage $t \in \{1, 2, \ldots, T-1\}$. Under the condition that miner i maximizes their payoff and wins their first block in stage t, it is optimal for miner i to continue mining on the same branch for the rest of the game.*

Note that Conjecture 3 does not apply if miner i does not maximize their payoff in stage t. One can construct an example where miner i wins their first block in stage t, in which it is not optimal for them to keep mining on the same branch in the subsequent stage $t+1$ when they do not maximize their payoffs in stage t (see Example 2). Additionally, Conjecture 3 may follow from Conjecture 2 (monotonicity), but we were unable to construct a formal proof. Conjecture 3 is only interesting in off-path situations. In the subgame-perfect equilibrium the situation is clear and Conjecture 3 will never be violated. Why? We know from Conjecture 1 and Proposition 1 that there are no forks along the subgame-perfect equilibrium path, which means that there will never be any other branches to switch to.

This conjecture is tested in Appendix B.1.3.

Example 2. *Consider a five-stage game with two miners. Let $t = 4$ and assume that miner 1 won all previous stages, which resulted in the sequence of blocks $B_3 = \{b_0, b_1, b_2, b_3\}$ with $\iota(b_1) = \iota(b_2) = \iota(b_3) = 1$ and $b_0 \Leftarrow b_1 \Leftarrow b_2 \Leftarrow b_3$. This is blockchain $\mathbb{B}_3$. We therefore have a situation where there are no forks yet. Now assume that miner 2 mines on the genesis block b_0 in stage $t = 4$, and that miner 2 wins this stage. Mining on b_0 cannot be payoff-maximizing, because it is not possible for a fork starting at b_0 to overtake the existing longest blockchain $\mathbb{B}_3$, because there are only two stages left in the game. Similarly, it cannot be optimal for miner 2 to mine on b_4 in the final stage $t = 5$, because the second blockchain $\{b_0, b_4\}$ with $b_0 \Leftarrow b_4$ cannot become the longest blockchain, even if b_5 is appended to b_4.*

7.4 Switching to Shorter Chains

Conjecture 4. *Consider a finite blockchain game with T stages and $n = 2$ payoff-maximizing miners. In any intermediate stage $t \in \{1, 2, \ldots, T-1\}$ miner i mines on some branch C, seeking to extend it. If miner i switches to a different branch C' in the next stage $t+1$, then that branch will never be shorter than the one they are switching from, i.e. C' will never be shorter than C.*

This conjecture is tested in Appendix B.1.4.

7.5 Coordination

Conjecture 5. *In a finite blockchain game with T stages and $n = 2$ payoff-maximizing miners, coordination between the miners is never necessary to play a Nash equilibrium in pure strategies in any stage (on and off the subgame-perfect equilibrium path).*

		Miner 2	
		b_0	b_1
Miner 1	b_0	$(\pi_1,\ \pi_2)$	$\pi_1',\ \pi_2'$
	b_1	$\pi_1',\ \pi_2'$	$(\pi_1,\ \pi_2)$

Table XIII – A payoff-matrix where coordination between the players is necessary to play one of the Nash equilibria. A situation like this can never happen as long as Conjecture 5 holds.

Consider Table XIII and assume that $\pi_1 > \pi_1'$ and $\pi_2 > \pi_2'$. This represents a situation akin to the classic game known as *Battle of the Sexes*, which was introduced by Luce and Raiffa (1957). There are two Nash equilibria in pure strategies, (b_0, b_0) and (b_1, b_1), which can only be played if the players coordinate their actions (consciously or by random chance). Ewerhart's model assumes coordination to be impossible because the miners move simultaneously. This can lead to a problem: What if the players play a strategy profile that is not a Nash equilibrium? To avoid this problem, we always

assumed that players only ever play a Nash equilibrium in any given stage, but this assumption is unnecessary as long as Conjecture 5 holds. Indeed, in practice we never found any situation where coordination was necessary to play a Nash equilibrium in pure strategies. When we encounter a situation with multiple Nash equilibria, the miners are always indifferent between each equilibrium. An example of such a situation is shown in Table V. All equilibria in that payoff-matrix yield identical payoffs for both miners. The game is well-behaved in this way.

This conjecture is tested in Appendix B.1.5.

7.6 Insights from Computer Simulations

Using computer simulations we were able to verify that all conjectures shown above (Conjecture 1, Conjecture 2, Conjecture 3, Conjecture 4 and Conjecture 5) are true for finite blockchain games with $T \leq 7$ stages.

On an average desktop computer with a 9th generation Intel i7 processor it took on the order of 24 hours to run each test. Significant speed-ups are possible, but unlikely to be of much use. Our Python implementation makes use of recursion, which is slow, but elegant. We require a speed-up of approximately 128 times to analyse an additional stage (this requirement gets larger with every additional stage). One arrives at this value by calculating the ratio between the number of terminal nodes in the game tree of a $T+1$ stage game and a T stage game.

$$\frac{\vartheta_8(2)}{\vartheta_7(2)} = 128$$

This value only serves as a rough estimate of how much more computational power we need. This means that if we were somehow able to achieve a speed-up of $100'000$ times (by using more powerful computers or a faster programming language), we would only be able to look at two or three additional stages, i.e. to look at games with $T \leq 9$ stages. More details about our Python code can be found in Appendix B.

8 Conclusion

We limited our analysis to finite blockchain games with $n = 2$ miners and investigated five conjectures, which proved to be true for games with $T \leq 7$ stages. In short, we found that the strategy profile (b_{t-1}, b_{t-1}) is played in every stage in the subgame-perfect equilibrium, which leads to only one chain without forks at the end of the game. This symmetric equilibrium is stable (i.e. no player has an incentive to deviate at any stage). It even represents an equilibrium in strictly dominant strategies and is therefore unique. Additionally, we found that the payoff-maximizing strategy is monotonic, that a miner will continue mining on the same branch they won their first block on for the rest of the game, that a miner will never switch to a shorter chain than the one they are already mining on, and that coordination between the miners is never necessary to play a Nash equilibrium in pure strategies any given stage.

We were only able to confirm these conjectures for games with less than seven stages because of the complexity every additional stage adds. To quantify this increase in complexity, we introduced the functions $\vartheta_T(n)$ and $\xi_T(n)$, which count the number of intermediate nodes in the game tree and the number of possible blockchains respectively.

Even tough we are looking at a complex multistage game, the subgame-perfect equilibrium strategies for both players are straight forward: In the subgame-perfect equilibrium they always mine on the most recent block. In this way, Ewerhart's model can show how a small set of rules can generate stable consensus in blockchains. Additionally, the subgame-perfect equilibrium is welfare-maximizing. No payoffs are lost because of orphaned blocks or branches.

Given that the computer simulations confirm all our conjectures for $T \leq 7$, it might be worth investigating them formally, i.e. to attempt to come up with formal proofs. The value of our computer based analysis lies in the fact that we now know that there are no "simple" counter-examples for the conjectures to be found. The conjectures are not easily disproven.

9 References

Biais, B., Bisière, C., Bouvard, M., & Casamatta, C. (2019). The blockchain folk theorem. *The Review of Financial Studies*, *32*(5), 1662–1715. https://doi.org/10.1093/rfs/hhy095

Carlsten, M., Kalodner, H., Weinberg, S. M., & Narayanan, A. (2016). On the instability of bitcoin without the block reward. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 154–67. https://doi.org/10.1145/2976749.2978408

Chaum, D. L. (1979). *Computer systems established, maintained and trusted by mutually suspicious groups*. Electronics Research Laboratory University of California.

Dimitri, N. (2017). Bitcoin mining as a contest. *Ledger*, *2*, 31–37. https://doi.org/10.5195/LEDGER.2017.96

Ewerhart, C. (2020). Finite blockchain games. *Economics Letters*, *197*, 109614. https://doi.org/10.1016/j.econlet.2020.109614

Eyal, I., & Sirer, E. G. (2018). Majority is not enough: Bitcoin mining is vulnerable. *Communications of the ACM*, *61*(7), 95–102. https://doi.org/10.1145/3212998

Kiayias, A., Koutsoupias, E., Kyropoulou, M., & Tselekounis, Y. (2016). Blockchain mining games. *Proceedings of the 2016 ACM Conference on Economics and Computation*, 365–382. https://doi.org/10.1145/2940716.2940773

Kroll, J. A., Davey, I. C., & Felten, E. W. (2013). The economics of bitcoin mining, or bitcoin in the presence of adversaries. *Proceedings of WEIS*, *2013*.

Luce, R. D., & Raiffa, H. (1957). *Games and decisions: Introduction and critical survey*. John Wiley & Sons.

Nakamoto, S. (2008). Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, 21260.

Nash, J. F. (1950). Equilibrium points in n-person games. *Proceedings of the National Academy of Sciences*, *36*(1), 48–49. https://doi.org/10.1073/pnas.36.1.48

Selten, R. (1965). Spieltheoretische Behandlung eines Oligopolmodells mit Nachfrageträgheit: Teil I: Bestimmung des dynamischen Preisgleichgewichts. *Zeitschrift für die gesamte Staatswissenschaft / Journal of Institutional and Theoretical Economics*, *121*(2), 301–324. https://www.jstor.org/stable/40748884

Sherman, A. T., Javani, F., Zhang, H., & Golaszewski, E. (2019). On the origins and variations of blockchain technologies. *IEEE Security & Privacy*, *17*(1), 72–77. https://doi.org/10.1109/MSEC.2019.2893730

A Appendix

A.1 More About the Four-Stage Game

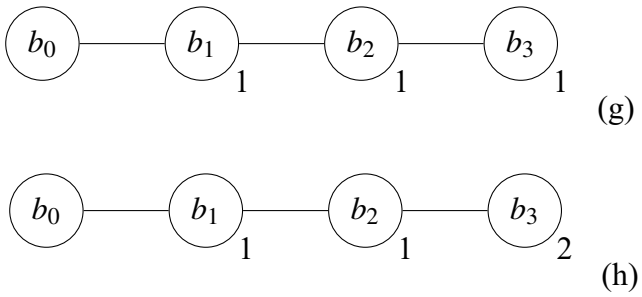

Figure A.1 – Two possible situations after the third stage, when case (a) in Figure 4 materialized after the second stage.

		Miner 2		
		b_0	b_1	b_2
Miner 1	b_0	2.38, 0.50	2.25, 0.75	2.50, 1.00
	b_1	2.38, 0.50	2.25, 0.75	2.50, 1.00
	b_2	2.88, 0.50	2.75, 0.75	(3.00, 1.00)

Table A.1 – Payoff-matrix for blockchain (a) in Figure 4 in the four-stage game. (Generated in Appendix B.1.6)

		Miner 2			
		b_0	b_1	b_2	b_3
Miner 1	b_0	3.00, 0.00	3.00, 0.00	2.75, 0.25	3.00, 0.50
	b_1	3.00, 0.00	3.00, 0.00	2.75, 0.25	3.00, 0.50
	b_2	3.00, 0.00	3.00, 0.00	2.75, 0.25	3.00, 0.50
	b_3	3.50, 0.00	3.50, 0.00	3.25, 0.25	(3.50, 0.50)

Table A.2 – Payoff-matrix for blockchain (g) in Figure A.1 in the four-stage game. (Generated in Appendix B.1.6)

		Miner 2			
		b_0	b_1	b_2	b_3
Miner 1	b_0	2.00, 1.00	2.00, 1.00	2.00, 1.00	2.00, 1.50
	b_1	2.00, 1.00	2.00, 1.00	2.00, 1.00	2.00, 1.50
	b_2	2.25, 0.75	2.25, 0.75	2.25, 0.75	2.25, 1.25
	b_3	2.50, 1.00	2.50, 1.00	2.50, 1.00	(2.50, 1.50)

Table A.3 – Payoff-matrix for blockchain (h) in Figure A.1 in the four-stage game. (Generated in Appendix B.1.6)

A.2 Payoff Matrices in the Second Stage

For the particularly interested reader we include a collection of payoff-matrices in the second stage. The time it took to generate each matrix is roughly equal to the entries in Table XII (same order of magnitude).

		Miner 2	
		b_0	b_1
Miner 1	b_0	2.19, 1.69	2.50, 2.00
	b_1	2.69, 1.69	(3.00, 2.00)

Table A.4 – Payoff-matrix in intermediate stage $t = 2$ in the five-stage game. (Generated in Appendix B.1.6)

		Miner 2	
		b_0	b_1
Miner 1	b_0	2.75, 2.25	3.00, 2.50
	b_1	3.25, 2.25	(3.50, 2.50)

Table A.5 – Payoff-matrix in intermediate stage $t = 2$ in the six-stage game. (Generated in Appendix B.1.6)

		Miner 2	
		b_0	b_1
Miner 1	b_0	3.25, 2.75	3.50, 3.00
	b_1	3.75, 2.75	(4.00, 3.00)

Table A.6 – Payoff-matrix in intermediate stage $t = 2$ in the seven-stage game. (Generated in Appendix B.1.6)

B Codebase

As mentioned in Section 6, we used Python to conduct some of our analysis. The code was used to verify the non-existence of forks in the subgame-perfect equilibrium in finite blockchain games with $T \leq 7$ stages and $n = 2$ miners. It was also used to test the other conjectures in Section 7 and to generate the payoff-matrices throughout this thesis. Jupyter Notebooks enabled us to add further explanations to our code, in an effort to make it more comprehensible and accessible to all readers. The notebooks are freely available on GitHub at https://github.com/koeztu/blockchain-game-analysis.

In all the Notebooks we follow the same conventions to make the code more readable.

- Function arguments start with and underline (e.g. `_x`), while function internal variables do not (e.g. `x`).
- Often we simply write 'array', by this we mean Numpy arrays specifically.
- It is important to know when we use lists and when we use arrays. The distinction will be made clear in the code's comments.
- In the code, we refer to the miners as miner 0 and miner 1 instead of as miner 1 and 2. This is because we start to count at 0 in Python.
- We make use of recursion. This is slow, but it is elegant and improves readability as well as comprehensibility.

B.1 Main

This notebook serves as a hub for all the simulation output. The reader may refer to the other notebooks for the inner-workings of the simulations.

Notes specific to this notebook:

- When the output says that it checked all situations for height h, it means that for all possible situations at the beginning of stage $t = h+1$, the conjecture was not violated in any of the subsequent intermediate stages $t, \ldots, T$. A block's height is equal to its index, which is equal to the stage it was mined in.
- We start checking the conjectures in stage $t = 2$. The first stage ($t = 1$) can be

skipped because we assume w.l.o.g. that miner 0 wins that stage. This also means that we can skip the two-stage game, because we know what the Nash equilibrium in the second stage of the two-stage game looks like, and that it does not violate any conjectures. Please refer to Section 4.1 about the two-stage game for more information.

[1]:
```
#IMPORTS
#libraries
import numpy as np
import time
import platform
import psutil
import cpuinfo #the package is called 'py-cpuinfo'
import import_ipynb

#notebooks
import blockchain as bc
import helper_functions as helfun
import payoff_matrix as pm
import conjectures

#SYSTEM SPECIFICATIONS
print("\n### System Specifications ###")
print("System:", platform.system())
print("Architecture:", cpuinfo.get_cpu_info()["arch"])
print("CPU:", cpuinfo.get_cpu_info()["brand_raw"])
print("CPU Cores:", psutil.cpu_count(logical=True))
print("Memory:", round(psutil.virtual_memory().total / (1024 ** 3), 2),␣
                        ↪"Gigabytes")

#VERSIONS
print("\n### Versions ###")
print("Python", platform.python_version()) #3.10.6
print("Numpy", np.__version__) #1.25.0
```

```
### System Specifications ###
System: Windows
Architecture: X86_64
CPU: Intel(R) Core(TM) i7-9700K CPU @ 3.60GHz
CPU Cores: 8
Memory: 31.92 Gigabytes

### Versions ###
Python 3.10.6
Numpy 1.25.0
```

B.1.1 Testing Conjecture 1

Forks. The relevant function `Forks()` is defined in `conjectures.ipynb` (Appendix B.4.1).

[2]:
```
n = 2
max_horizon = 7

#test all equilibrium paths for forks
for T in range(3, max_horizon+1):
    start = time.process_time()
    print(f'found a fork for T={T}? ', conjectures.Forks(T, n))
    end = time.process_time()
    print(helfun.timeElapsed(start, end))
    print("")

print("done")
```

```
found a fork for T=3?  False
0.0 seconds

found a fork for T=4?  False
0.2 seconds

found a fork for T=5?  False
11.0 seconds

found a fork for T=6?  False
13 minutes

found a fork for T=7?  False
24 hours

done
```

B.1.2 Testing Conjecture 2

Monotonicity. The relevant function `Monotonicity()` is defined in `conjectures.ipynb` (Appendix B.4.2).

[3]:
```
for T in range(3, max_horizon+1):
    start = time.process_time()
    conjectures.Monotonicity(T, n)
    end = time.process_time()
    print(helfun.timeElapsed(start, end))
    print("")

print("done")
```

```
checked height 1 in games with T = 3 stages
0.0 seconds

checked height 1 in games with T = 4 stages
checked height 2 in games with T = 4 stages
0.3 seconds

checked height 1 in games with T = 5 stages
checked height 2 in games with T = 5 stages
checked height 3 in games with T = 5 stages
17.8 seconds

checked height 1 in games with T = 6 stages
checked height 2 in games with T = 6 stages
checked height 3 in games with T = 6 stages
checked height 4 in games with T = 6 stages
22 minutes
```

```
checked height 1 in games with T = 7 stages
checked height 2 in games with T = 7 stages
checked height 3 in games with T = 7 stages
checked height 4 in games with T = 7 stages
checked height 5 in games with T = 7 stages
39 hours

done
```

B.1.3 Testing Conjecture 3

A miner's behaviour after winning their first block. The relevant function `FirstBlock()` is defined in `conjectures.ipynb` (Appendix B.4.3).

[4]:
```
for T in range(3, max_horizon+1):
    start = time.process_time()
    conjectures.FirstBlock(T, n)
    end = time.process_time()
    print(helfun.timeElapsed(start, end))
    print("")

print("done")
```

```
checked height 1 in games with T = 3 stages
0.0 seconds

checked height 1 in games with T = 4 stages
checked height 2 in games with T = 4 stages
0.2 seconds

checked height 1 in games with T = 5 stages
checked height 2 in games with T = 5 stages
checked height 3 in games with T = 5 stages
12.8 seconds

checked height 1 in games with T = 6 stages
checked height 2 in games with T = 6 stages
checked height 3 in games with T = 6 stages
checked height 4 in games with T = 6 stages
16 minutes

checked height 1 in games with T = 7 stages
checked height 2 in games with T = 7 stages
checked height 3 in games with T = 7 stages
checked height 4 in games with T = 7 stages
checked height 5 in games with T = 7 stages
29 hours

done
```

B.1.4 Testing Conjecture 4

Switching to shorter chains. The relevant function `Switching()` is defined in `conjectures.ipynb` (Appendix B.4.4).

[5]:
```
for T in range(3, max_horizon+1):
    start = time.process_time()
    conjectures.Switching(T, n)
    end = time.process_time()
    print(helfun.timeElapsed(start, end))
    print("")

print("done")
```

```
checked height 1 in games with T = 3 stages
0.0 seconds

checked height 1 in games with T = 4 stages
checked height 2 in games with T = 4 stages
0.3 seconds

checked height 1 in games with T = 5 stages
checked height 2 in games with T = 5 stages
checked height 3 in games with T = 5 stages
18.3 seconds

checked height 1 in games with T = 6 stages
checked height 2 in games with T = 6 stages
checked height 3 in games with T = 6 stages
checked height 4 in games with T = 6 stages
23 minutes

checked height 1 in games with T = 7 stages
checked height 2 in games with T = 7 stages
checked height 3 in games with T = 7 stages
checked height 4 in games with T = 7 stages
checked height 5 in games with T = 7 stages
41 hours

done
```

B.1.5 Testing Conjecture 5

Coordination. The relevant function `Coordination()` is defined in `conjectures.ipynb` (Appendix B.4.5).

[6]:
```
for T in range(3, max_horizon+1):
    start = time.process_time()
    conjectures.Coordination(T, n)
    end = time.process_time()
    print(helfun.timeElapsed(start, end))
    print("")

print("done")
```

```
checked stage t = 2 in games with T = 3 stages
checked stage t = 3 in games with T = 3 stages
0.0 seconds

checked stage t = 2 in games with T = 4 stages
checked stage t = 3 in games with T = 4 stages
checked stage t = 4 in games with T = 4 stages
0.3 seconds

checked stage t = 2 in games with T = 5 stages
checked stage t = 3 in games with T = 5 stages
```

```
checked stage t = 4 in games with T = 5 stages
checked stage t = 5 in games with T = 5 stages
15.2 seconds

checked stage t = 2 in games with T = 6 stages
checked stage t = 3 in games with T = 6 stages
checked stage t = 4 in games with T = 6 stages
checked stage t = 5 in games with T = 6 stages
checked stage t = 6 in games with T = 6 stages
19 minutes

checked stage t = 2 in games with T = 7 stages
checked stage t = 3 in games with T = 7 stages
checked stage t = 4 in games with T = 7 stages
checked stage t = 5 in games with T = 7 stages
checked stage t = 6 in games with T = 7 stages
checked stage t = 7 in games with T = 7 stages
33 hours

done
```

B.1.6 Payoff Matrices

The code below generates all the payoff-matrices shown in the thesis.

[7]:
```
#PRINT PAYOFF MATRICES IN SECOND STAGE FOR GAMES OF HORIZON 'T'
for T in range(2, max_horizon+1):

    #BLOCKCHAIN WITH TWO BLOCKS, b0 and b1
    parents = np.array([0])
    winners = np.array([0]) #w.l.o.g. miner 0 wins the first stage

    B = bc.Blockchain(n, parents, winners)

    start = time.process_time()
    print(f"\npayoff-matrix for t = 2 and T = {T}:\n\n", pm.
                        ↪intermediatePayoffMatrix(B, T, n))
    end = time.process_time()
    print("\n", helfun.timeElapsed(start, end), "\n")
```

```
payoff-matrix for t = 2 and T = 2:

 [[[0.75 0.25]
  [1.   0.5 ]]

 [[1.25 0.25]
  [1.5  0.5 ]]]

 (shown here: Table II)
 0.0 seconds


payoff-matrix for t = 2 and T = 3:

 [[[1.25 0.75]
  [1.5  1.  ]]

 [[1.75 0.75]
  [2.   1.  ]]]

 (shown here: Table VII)
 0.0 seconds
```

```
payoff-matrix for t = 2 and T = 4:

 [[[1.6875 1.1875]
  [2.     1.5   ]]

 [[2.1875 1.1875]
  [2.5    1.5   ]]]

 (shown here: Table XI)
 0.2 seconds


payoff-matrix for t = 2 and T = 5:

 [[[2.1875 1.6875]
  [2.5    2.    ]]

 [[2.6875 1.6875]
  [3.     2.    ]]]

 (shown here: Table A.4)
 8.6 seconds


payoff-matrix for t = 2 and T = 6:

 [[[2.75 2.25]
  [3.   2.5 ]]

 [[3.25 2.25]
  [3.5  2.5 ]]]

 (shown here: Table A.5)
 10 minutes


payoff-matrix for t = 2 and T = 7:

 [[[3.25 2.75]
  [3.5  3.  ]]

 [[3.75 2.75]
  [4.   3.  ]]]

 (shown here: Table A.6)
 18 hours
```

Next we generate payoff-matrices in Section 4.2.1 for specific blockchains (a), (b), (c) and (d) in the three-stage game (Figure 4).

[4]:
```
#BLOCKCHAIN (a) IN THREE-STAGE GAME
parents = np.array([0, 1]) #block b1 is appended to the genesis block, block b2␣
                           ↪is appended to b1
winners = np.array([0, 0]) #miner 0 wins the first and second stage
helfun.printPayoffMatrix(n, parents, winners, "(a)", 3)


#BLOCKCHAIN (b) IN THREE-STAGE GAME
parents = np.array([0, 1]) #block b1 is appended to the genesis block, block b2␣
                           ↪is appended to b1
winners = np.array([0, 1]) #miner 0 wins the first stage, miner 1 wins the second␣
                           ↪stage
```

```
helfun.printPayoffMatrix(n, parents, winners, "(b)", 3)


#BLOCKCHAIN (c) IN THREE-STAGE GAME
parents = np.array([0, 0]) #both blocks b1 and b2 are appended to the genesis␣
                           ↪block
winners = np.array([0, 0]) #miner 0 wins the first and second stage
helfun.printPayoffMatrix(n, parents, winners, "(c)", 3)


#BLOCKCHAIN (d) IN THREE-STAGE GAME
parents = np.array([0, 0]) #both blocks b1 and b2 are appended to the genesis␣
                           ↪block
winners = np.array([0, 1]) #miner 0 wins the first stage, miner 1 wins the second␣
                           ↪stage
helfun.printPayoffMatrix(n, parents, winners, "(d)", 3)
```

```
blockchain (a):

b0
└── b1
    └── b2

miner 0 wins stages [1 2]
miner 1 wins stages []

payoff-matrix for blockchain (a) in 3-stage game:

 [[[2.   0.  ]
  [1.75 0.25]
  [2.   0.5 ]]

 [[2.   0.  ]
  [1.75 0.25]
  [2.   0.5 ]]

 [[2.5  0.  ]
  [2.25 0.25]
  [2.5  0.5 ]]]

 (shown here: Table III)
 0.0 seconds


blockchain (b):

b0
└── b1
    └── b2

miner 0 wins stages [1]
miner 1 wins stages [2]

payoff-matrix for blockchain (b) in 3-stage game:

 [[[1.   1.  ]
  [1.   1.  ]
  [1.   1.5 ]]

 [[1.25 0.75]
  [1.25 0.75]
  [1.25 1.25]]

 [[1.5  1.  ]
  [1.5  1.  ]
  [1.5  1.5 ]]]
```

```
 (shown here: Table IV)
 0.0 seconds


blockchain (c):

b0
├── b1
└── b2

miner 0 wins stages [1 2]
miner 1 wins stages []

payoff-matrix for blockchain (c) in 3-stage game:

 [[[0.83333333 0.16666667]
  [1.         0.5       ]
  [1.         0.5       ]]

 [[1.33333333 0.16666667]
  [1.5        0.5       ]
  [1.5        0.5       ]]

 [[1.33333333 0.16666667]
  [1.5        0.5       ]
  [1.5        0.5       ]]]

 (shown here: Table V)
 0.0 seconds


blockchain (d):

b0
├── b1
└── b2

miner 0 wins stages [1]
miner 1 wins stages [2]

payoff-matrix for blockchain (d) in 3-stage game:

 [[[0.5        0.5       ]
  [0.83333333 0.66666667]
  [0.33333333 1.16666667]]

 [[1.16666667 0.33333333]
  [1.5        0.5       ]
  [1.         1.        ]]

 [[0.66666667 0.83333333]
  [1.         1.        ]
  [0.5        1.5       ]]]

 (shown here: Table VI)
 0.0 seconds
```

Next we generate payoff-matrices in Section 4.3.1 and Section 4.3.2, as well as Appendix A.1 for specific blockchains (a), (b), (e), (f), (g) and (h) in the four-stage game (Figure 4, Figure 5 and Figure A.1).

[5]:
```
#BLOCKCHAIN (a) IN FOUR-STAGE GAME
parents = np.array([0, 1]) #block b1 is appended to the genesis block, block b2␣
                           ↪is appended to b1
winners = np.array([0, 0]) #miner 0 wins the first and second stage
helfun.printPayoffMatrix(n, parents, winners, "(a)", 4)


#BLOCKCHAIN (b) IN FOUR-STAGE GAME
parents = np.array([0, 1]) #block b1 is appended to the genesis block, block b2␣
                           ↪is appended to b1
winners = np.array([0, 1]) #miner 0 wins the first stage, miner 1 wins the second␣
                           ↪stage
helfun.printPayoffMatrix(n, parents, winners, "(b)", 4)

#BLOCKCHAIN (e) IN FOUR-STAGE GAME
parents = np.array([0, 1, 2]) #block b1 is appended to the genesis block, block␣
                           ↪b2 is appended to b1, block b3 is appended to b2
winners = np.array([0, 1, 0]) #miner 0 wins the first and third stages, miner 2␣
                           ↪wins the second stage
helfun.printPayoffMatrix(n, parents, winners, "(e)", 4)


#BLOCKCHAIN (f) IN FOUR-STAGE GAME
parents = np.array([0, 1, 2]) #block b1 is appended to the genesis block, block␣
                           ↪b2 is appended to b1, block b3 is appended to b2
winners = np.array([0, 1, 1]) #miner 0 wins the first stage, miner 2 wins the␣
                           ↪second and third stages
helfun.printPayoffMatrix(n, parents, winners, "(f)", 4)

#BLOCKCHAIN (g) IN FOUR-STAGE GAME
parents = np.array([0, 1, 2]) #block b1 is appended to the genesis block, block␣
                           ↪b2 is appended to b1, block b3 is appended to b2
winners = np.array([0, 0, 0]) #miner 0 wins the first, second and third stages
helfun.printPayoffMatrix(n, parents, winners, "(g)", 4)


#BLOCKCHAIN (h) IN FOUR-STAGE GAME
parents = np.array([0, 1, 2]) #block b1 is appended to the genesis block, block␣
                           ↪b2 is appended to b1, block b3 is appended to b2
winners = np.array([0, 0, 1]) #miner 0 wins the first and second stages, miner 1␣
                           ↪wins the third stage
helfun.printPayoffMatrix(n, parents, winners, "(h)", 4)
```

```
blockchain (a):

b0
└── b1
    └── b2

miner 0 wins stages [1 2]
miner 1 wins stages []

payoff-matrix for blockchain (a) in 4-stage game:

 [[[2.375 0.5  ]
  [2.25  0.75 ]
  [2.5   1.   ]]

 [[2.375 0.5  ]
  [2.25  0.75 ]
  [2.5   1.   ]]

 [[2.875 0.5  ]
  [2.75  0.75 ]
  [3.    1.   ]]]
```

```
 (shown here: Table A.1)
 0.0 seconds


blockchain (b):

b0
└── b1
    └── b2

miner 0 wins stages [1]
miner 1 wins stages [2]

payoff-matrix for blockchain (b) in 4-stage game:

 [[[1.5  1.5 ]
  [1.5  1.5 ]
  [1.5  2.  ]]

 [[1.75 1.25]
  [1.75 1.25]
  [1.75 1.75]]

 [[2.   1.5 ]
  [2.   1.5 ]
  [2.   2.  ]]]

 (shown here: Table X)
 0.0 seconds


blockchain (e):

b0
└── b1
    └── b2
        └── b3

miner 0 wins stages [1 3]
miner 1 wins stages [2]

payoff-matrix for blockchain (e) in 4-stage game:

 [[[2.   1.  ]
  [2.   1.  ]
  [1.75 1.25]
  [2.   1.5 ]]

 [[2.   1.  ]
  [2.   1.  ]
  [1.75 1.25]
  [2.   1.5 ]]

 [[2.   1.  ]
  [2.   1.  ]
  [1.75 1.25]
  [2.   1.5 ]]

 [[2.5  1.  ]
  [2.5  1.  ]
  [2.25 1.25]
  [2.5  1.5 ]]]

 (shown here: Table VIII)
 0.0 seconds


blockchain (f):
```

```
b0
└── b1
    └── b2
        └── b3

miner 0 wins stages [1]
miner 1 wins stages [2 3]

payoff-matrix for blockchain (f) in 4-stage game:

 [[[1.   2.  ]
  [1.   2.  ]
  [1.   2.  ]
  [1.   2.5 ]]

 [[1.   2.  ]
  [1.   2.  ]
  [1.   2.  ]
  [1.   2.5 ]]

 [[1.25 1.75]
  [1.25 1.75]
  [1.25 1.75]
  [1.25 2.25]]

 [[1.5  2.  ]
  [1.5  2.  ]
  [1.5  2.  ]
  [1.5  2.5 ]]]

 (shown here: Table IX)
 0.0 seconds


blockchain (g):

b0
└── b1
    └── b2
        └── b3

miner 0 wins stages [1 2 3]
miner 1 wins stages []

payoff-matrix for blockchain (g) in 4-stage game:

 [[[3.   0.  ]
  [3.   0.  ]
  [2.75 0.25]
  [3.   0.5 ]]

 [[3.   0.  ]
  [3.   0.  ]
  [2.75 0.25]
  [3.   0.5 ]]

 [[3.   0.  ]
  [3.   0.  ]
  [2.75 0.25]
  [3.   0.5 ]]

 [[3.5  0.  ]
  [3.5  0.  ]
  [3.25 0.25]
  [3.5  0.5 ]]]

 (shown here: Table A.2)
 0.0 seconds
```

```
blockchain (h):

b0
└── b1
    └── b2
        └── b3

miner 0 wins stages [1 2]
miner 1 wins stages [3]

payoff-matrix for blockchain (h) in 4-stage game:

 [[[2.   1.  ]
  [2.   1.  ]
  [2.   1.  ]
  [2.   1.5 ]]

 [[2.   1.  ]
  [2.   1.  ]
  [2.   1.  ]
  [2.   1.5 ]]

 [[2.25 0.75]
  [2.25 0.75]
  [2.25 0.75]
  [2.25 1.25]]

 [[2.5  1.  ]
  [2.5  1.  ]
  [2.5  1.  ]
  [2.5  1.5 ]]]

 (shown here: Table A.3)
 0.0 seconds
```

B.2 Blockchain Module

The heart of it all.

[1]:
```
#IMPORTS
import numpy as np
```

B.2.1 Block Class

A block has three attributes:

- Its height in the blockchain.
- The height of its parent.
- The player who mined it. This is equivalent to the value of the assignment map ι for this block.

Additionally, we have a method to print the attributes in a convenient form.

[2]:
```
class Block:
    def __init__(self, _h, _p, _i):
        self.height = _h #equivalent to the stage in which the block was mined
        self.parent = _p #height of parent block
        self.iota = _i #index of the miner who mined the block

    def printBlock(self):
        print("block", self.height, "mined by miner", f"{self.iota},", "parent is␣
                        ↪block", self.parent)
```

B.2.2 Blockchain Class

This class builds on top of the `Block` class. An object of class `Blockchain` has the following attributes:

- The number of players. In our case this is always equal to 2.
- An array that keeps track of the parent-child relation $\Leftarrow$ on the sequence of blocks.
- An array that keeps track of the winners of each block. This is equivalent to the assignment map ι.
- The number of stages T the game has.
- An array that contains T objects of class `Block`. The method `__generateChain()` is responsible for the creation of this array.
- Lastly, we have a list of the longest chains. This is done because it is efficient. We construct the list when the `Blockchain` object is created. Later we can simply look up what the longest chains are, instead of computing them each time we need that information. The method `__getLongestChains()` is responsible for the creation of this list.

Some methods start with double underscores `__`, this is supposed to signify that these methods should only ever be called by the class itself. Python lacks the private / public functionality we know from other languages.

The remaining methods not yet mentioned are straight forward. The method `printMiner()` prints the blocks which miner i won and the method `getPayoff()` calculates the number of blocks a miner i mined in a chain ending in block b_t. The last method `chainLength()` is used to calculate the length of the chain ending in the block

b_t.

We make use of the fact that a block can never have more than one parent. This allows us to iterate backwards all the way from a given block b_t to the genesis block b_0. This is a nice and convenient property of blockchains. The path from block b_t to b_0 would not necessarily be unique if blocks were allowed multiple parents.

[3]:
```
class Blockchain:
    def __init__(self, _n, _parents, _winners):
        #
        #'_parents' is an array where '_parents[t]' is the parent of block 't+1',␣
                        ↪i.e. the fifth element is the index of the parent of␣
                        ↪the block in t=5
        #'_winners' is an array where '_winners[t]' is the index of the winner of␣
                        ↪stage 't+1'
        #
        #'self.sequence' is an array with objects of class 'Block'
        #'self.horizon' is the time horizon T of the game and is implicitly given␣
                        ↪by the shape of the '_parents' array.
        #'self.winners' is an array with the index of the winning miner of every␣
                        ↪stage.
        #

        self.n = _n
        self.parents = _parents
        self.winners = _winners
        self.horizon = self.parents.shape[0]

        self.sequence = self.__generateChain()

        #pre-calculate a list of the longest chains for more speed
        self.longestchains = self.__getLongestChains() #a list of lists


    def __generateChain(self):
        #
        #generates the chain according to the miners strategies and the rounds␣
                        ↪they win
        #

        #CREATE ARRAY
        sequence = [] #use a list because appending to lists is approximately 5x␣
                        ↪faster than appending to numpy arrays
        for t in range(self.horizon + 1): #fill a chain with default blocks, we␣
                        ↪have 'T+1' blocks: that is 'T' blocks mined plus the␣
                        ↪genesis block
            if t == 0:
                sequence.append(Block(0, np.nan, np.nan)) #genesis block
            else:
                sequence.append(Block(t, self.parents[t-1], self.winners[t-1]))

        sequence = np.array(sequence) #convert to array

        return sequence


    def __getLongestChains(self):
        #
        #returns a list of lists filled with the intermediate stages 't' where␣
                        ↪the blocks in the longest chains were mined
```

```
        #example: if the longest chain is '{b0, b1, b2, b4, b6, b9}' it returns a␣
                        ↪list '[[0, 1, 2, 4, 6, 9]]'
        #we only ever need to know the longest chain at the end of the game, not␣
                        ↪in intermediate stages. we take advantage of this fact.
        #

        T = self.horizon
        longest_chains = []
        longest_chain_length = 0

        for t in range(T+1):
            #test if the block 't' has a child, if not, then it is the last block␣
                        ↪in a chain
            #we take advantage of the fact that chains that end at a block with a␣
                        ↪child can never be the longest

            childless = True

            for s in range(t+1, T+1): #for each block that could be child of␣
                        ↪block 't'
                if self.sequence[s].parent == t: #if this is true, then block 't'␣
                        ↪has a child
                    childless = False
                    break

            if childless == True: #if block 't' is childless
                candidate_chain = [0] #the longest chain always starts with the␣
                        ↪genesis block at height 0
                u = t #'u' is the index of the block we are currently looking at,␣
                        ↪we start with the last block in the chain: block 't'

                #ITERATE BACKWARDS
                #keep appending blocks to the candidate chain as long as we have␣
                        ↪not reached the genesis block with height 0
                while self.sequence[u].height != 0:
                    candidate_chain.append(u)
                    u = self.sequence[u].parent

                length_candidate_chain = len(candidate_chain)

                #TEST LENGTH
                if length_candidate_chain >= longest_chain_length: #if the chain␣
                        ↪is one of the longest we've seen so far
                    candidate_chain.sort()

                    #ADD
                    #add the candidate chain to the list of longest chains if it␣
                        ↪is not strictly the longest
                    if length_candidate_chain == longest_chain_length:
                        longest_chains.append(candidate_chain)

                    #OR REPLACE
                    #replace the list of longest chains with the candidate chain␣
                        ↪if it is strictly longer than all chains we saw before
                    if length_candidate_chain > longest_chain_length:
                        longest_chain_length = length_candidate_chain
                        longest_chains = [candidate_chain]

        return longest_chains


    def printMiner(self, _i):
        #
        #prints a lits of stages miner '_i' won
        #

        stages_won = np.where(self.winners == _i)[0]
        stages_won = stages_won + 1
        print("miner", _i, "wins stages", stages_won)
```

```
    def getPayoff(self, _i, _t):
        #
        #sums up all the blocks miner '_i' mined in the chain going from block 0␣
                        ↪to block '_t'
        #returns an integer corresponding to the payoff if the game were to end␣
                        ↪in '_t'
        #

        payoff = 0
        t = _t

        #ITERATE BACKWARDS
        while self.sequence[t].height != 0:
            if self.winners[t-1] == _i:
                payoff += 1
            t = self.sequence[t].parent

        return payoff


    def chainLength(self, _t):
        #
        #returns the length of the chain ending in block mined in stage '_t'
        #

        length = 1 #we must start at 1 because the genesis block is not counted␣
                        ↪in the while loop below
        t = _t

        #ITERATE BACKWARDS
        while self.sequence[t].height != 0:
            length += 1
            t = self.sequence[t].parent

        return length
```

B.2.3 Extended Blockchain Class

The blockchain is extended often. The simplest solution we found is to create a new class for this purpose.

[4]:
```
class ExtendedBlockchain(Blockchain):
    def __init__(self, _B, _i, _t):
        #
        #an extended copy of the base blockchain '_B', extended by one block,␣
                        ↪which is appended at block at height '_t'
        #miner '_i' wins the appended block
        #

        #EXTEND
        parents = np.append(_B.parents, _t)
        winners = np.append(_B.winners, _i)

        #BUILD
        Blockchain.__init__(self, _B.n, parents, winners)
```

B.3 Payoff Matrix Module

This notebook implements the game-theoretic aspects of the code.

As mentioned in `tests.ipynb`, we checked that the code returns the correct results for $T \leq 3$. We also did some checks for $T = 4$, but not for all possibilities, just the ones that could likely contain errors (cases where there are multiple Nash equilibria off-path).

[1]:
```
#IMPORTS
#libraries
import numpy as np
import import_ipynb

#notebooks
import blockchain as bc
```

The function `finalExpectedPayoff()` makes use of the blockchain class method `getPayoff()` to calculate the payoff for miner i over the longest chains. That is, it computes the expected value for miner i when Nature chooses one of the longest chains at random.

The function `finalPayoffMatrix()` builds on top of the previously mentioned `finalExpectedPayoff()` function. It returns the payoff matrix of the normal-form simultaneous game in the final stage of the game.

Next, we have the function `matrixNashEq()`. It finds the Nash equilibria of a payoff matrix. That payoff matrix may be generated by `finalPayoffMatrix()` or `intermediatePayoffMatrix()`, both work. Finally, the function `getStrategies()` returns the strategies of all miners in every Nash equilibrium.

These functions only work for $n = 2$ and are written with that assumption in mind. Going higher than $n = 2$ is not feasible anyway.

[2]:
```
def finalExpectedPayoff(_B, _i):
    #
    #'_B' is a blockchain object
    #'_i' is the index of the miner we want the compute the expected payoff for
    #
    #this function simply computes the expected payoff for miner 'i' over all the␣
                        ↪longest chains in '_B'
    #returns the payoff 'p' of the game if it were to end now (i.e. if '_B'␣
                        ↪defines the situation at the end of the game)
    #

    longest_chains = _B.longestchains #this is a list of lists!
    l = len(longest_chains) #number of longest chains
    payoffs = []

    #FILL THE LIST
    for chain in longest_chains:
```

```
        payoffs.append(_B.getPayoff(_i, chain[-1]))

    p = np.sum(payoffs) / l #each chain is chosen at random with equal probability

    return p


def finalPayoffMatrix(_B, _n, _T):
    #
    #calculates the payoff matrix of the normal-form simultaneous game with '_n' players in the final stage '_T' with blockchain '_B',
    #before nature chooses the winning chain!
    #

    M = np.zeros((_T, _T, _n))

    for r in range(_T): #rows, blocks m0 can mine at
        for c in range(_T): #cols, blocks m1 can mine at

            #EXTEND THE CHAIN
            B_ext0 = bc.ExtendedBlockchain(_B, 0, r) #the case where miner 0 wins the final stage
            B_ext1 = bc.ExtendedBlockchain(_B, 1, c) #the case where miner 1 wins the final stage

            #EXPECTED VALUE
            M[r, c, 0] = 0.5 * (finalExpectedPayoff(B_ext0, 0) + finalExpectedPayoff(B_ext1, 0)) #payoff for miner 0
            M[r, c, 1] = 0.5 * (finalExpectedPayoff(B_ext0, 1) + finalExpectedPayoff(B_ext1, 1)) #payoff for miner 1

    return M


def matrixNashEq(_M):
    #
    #finds the Nash equilibrium of the payoff matrix '_M'
    #find mutual best responses!
    #returns a mask with entries 'True' where we have a Nash equilibrium
    #

    #EXTRACT PAYOFFS FOR EACH MINER
    M0 = _M[:,:,0] #payoff matrix for miner 0
    M1 = _M[:,:,1] #payoff matrix for miner 1

    #FIND BEST RESPONSES
    best_responses0 = (M0 == M0.max(axis=0, keepdims=True))
    best_responses1 = (M1 == M1.max(axis=1, keepdims=True))

    #CONFIGURE MASK
    nash_eq = np.logical_and(best_responses0, best_responses1)

    return nash_eq


def getStrategies(_B, _T, _n):
    #
    #determines the Nash equilibrium strategies for all '_n' miners given some blockchain '_B' and game horizon '_T'
    #returns a list of tuples containing the equilibirum strategies
    #nash_strats[0][1] is the strategy of the second miner '[1]' in the first equilibirum '[0]'
    #

    M = intermediatePayoffMatrix(_B, _T, _n)
    nash = matrixNashEq(M)
```

```
    nash_strats = np.transpose(np.where(nash == True))
    nash_strats = [tuple(element) for element in nash_strats] #convert to tuples

    return nash_strats
```

The final function of this module is `intermediatePayoffMatrix()`. It is the most complex function in this notebook, since it makes use of recursion.

The function is recursively called at every decision node in the game tree. This causes the code to take a long time to terminate because the game tree explodes very rapidly in size.

[3]:
```
def intermediatePayoffMatrix(_B, _T, _n):
    #
    #calculates the payoff matrix for an intermediate stage of a game with '_n' players,
    #which starts with blockchain '_B' and ends at stage '_T'
    #makes use of recursion!
    #

    t = _B.horizon + 1 #current stage, we build on top of '_B'
    strats = np.arange(t) #all possible strategies, that is choosing a block 'b_t'
    M = np.zeros((t, t, _n))

    if _T == t: #once we have reached the final stage
        M = finalPayoffMatrix(_B, _n, t)

    else: #otherwise we need backwards induction from the last stage 't = T' to get the payoffs for the current stage
        for strat0 in strats: #for every possible strategy of miner 0
            for strat1 in strats: #for every possible strategy of miner 1

                #EXTEND THE BLOCKCHAIN
                B_ext0 = bc.ExtendedBlockchain(_B, 0, strat0) #the case where miner 0 wins
                B_ext1 = bc.ExtendedBlockchain(_B, 1, strat1) #the case where miner 1 wins

                #CALL RECURSION
                #find the payoff matrices for the current stage, one for each possible winner
                M_ext0 = intermediatePayoffMatrix(B_ext0, _T, _n) #the case where miner 0 wins
                M_ext1 = intermediatePayoffMatrix(B_ext1, _T, _n) #the case where miner 1 wins

                #FIND NASH EQUILIBRIA
                #we are only interested in equilibira in pure strategies
                nash0 = matrixNashEq(M_ext0) #the case where miner 0 wins
                nash1 = matrixNashEq(M_ext1) #the case where miner 1 wins

                #FIND THE NUMBER OF NASH EQUILIBRIA
                n_eq0 = np.count_nonzero(nash0) #the case where miner 0 wins
                n_eq1 = np.count_nonzero(nash1) #the case where miner 1 wins

                #EXPECTED VALUE
                #we average over all Nash equilibria in the next stage to get the expected payoff in the current stage
                M[strat0, strat1, 0] = 0.5 * (M_ext0[nash0].sum(axis=0)[0]/n_eq0 + M_ext1[nash1].sum(axis=0)[0]/n_eq1) #expected payoff for miner 0
```

```
                M[strat0, strat1, 1]  = 0.5 * (M_ext0[nash0].sum(axis=0)[1]/n_eq0␣
                         ↪+ M_ext1[nash1].sum(axis=0)[1]/n_eq1) #expected payoff␣
                         ↪for miner 1

    return M
```

B.4 Conjectures

In this notebook we define the functions that ultimately test our conjectures.

Some functions share a lot of code. It is possible to speed up the code by checking multiple conjectures at once. We opted to sacrifice speed for comprehensibility. Why? Because more speed does not help us reach more stages, unless it is a big improvement (on the order of 100 times faster).

[1]:
```
#IMPORTS
#libraries
import numpy as np
import itertools
import time
import import_ipynb

#notebooks
import blockchain as bc
import helper_functions as helfun
import payoff_matrix as pm
```

B.4.1 Code for Conjecture 1

Forks. We run this function in `main.ipynb` (Appendix B.1.1).

[2]:
```
def Forks(_T, _n):
    #
    #'_T' is the number of stages in the game
    #'_n' is the number of players, which must be 2
    #
    #we consider on-path situations
    #we want to figure out if there is an on-path situation where at least one␣
                         ↪miner does not mine on the most recent block. That␣
                         ↪would mean that they create a fork
    #

    assert _T > 2, "go further!" #we do not need to check the two-stage game, and␣
                         ↪the code is written with that in mind

    #BUILD BASE CHAIN
    #we start in stage t=2, hence the blockchain has two blocks, b0 and b1
    parents = np.array([0])
    winners = np.array([0]) #miner 0 wins the first stage w.l.o.g.

    B = bc.Blockchain(_n, parents, winners)

    nash_strats = pm.getStrategies(B, _T, _n)
```

```
    if len(nash_strats) > 1: #check if there are multiple equilibria
        print('situation with multiple equilibria found!')
        return True #counter-example with more than one nash-eq (on-path) found
    if nash_strats[0][0] != 1 and nash_strats[0][1] != 1: #check if any miner␣
                        ↪mines on a block different from b1 in stage t=2
        print('situation where one player wants to fork found!')
        return True #counter-example found
    for i in range(_n):
        B_ext = bc.ExtendedBlockchain(B, i, nash_strats[0][i]) #extend the␣
                        ↪blockchain in the case where miner 'i' wins
        found = recursionForks(B_ext, 2, _T, _n) #start in the second stage
        if found == True:
            return True

    return False



def recursionForks(_B, _t, _T, _n):
    #
    #'_B' is the blockchain of the previous stage
    #'_t' is the previous stage
    #'_T' is the number of stages in the game
    #'_n' is the number of miners
    #

    t = _t + 1 #current stage

    nash_strats = pm.getStrategies(_B, _T, _n)

    if len(nash_strats) > 1: #check if there are multiple equilibria
        print('situation with multiple equilibria found!')
        return True #counter-example with more than one nash-eq (on-path) found
    if nash_strats[0][0] != _t and nash_strats[0][1] != _t: #check if any miner␣
                        ↪mines on a block different from b_{t-1} in stage 't'
        print('situation where one player wants to fork found!')
        return True #counter-example found
    if t == _T:
        return False #no counter-example found
    for i in range(_n):
        B_ext = bc.ExtendedBlockchain(_B, i, nash_strats[0][i]) #extend the␣
                        ↪blockchain in the case where miner 'i' wins
        found = recursionForks(B_ext, t, _T, _n) #found a counter-example?
        if found == True:
            return True

    return False
```

B.4.2 Code for Conjecture 2

Monotonicity. To test this conjecture we compute the Nash equilibrium strategies in two subsequent stages and check if the conjecture is violated. For example, consider a blockchain of height 5 in a game with $T = 7$ stages. We compute the strategy of miner i in stage $t = 6$ and $t = 7$ and check if monotonicity is violated. This is the reason we only generate blockchains of height up to $T - 2$; in stage $T - 1$ we build on top of a blockchain of height $T - 2$. We must have at least two more stages to go in the game! We run this function in `main.ipynb` (Appendix B.1.2).

[3]:

```
def Monotonicity(_T, _n):
    #
    #check if there are situations where monotonicity is violated
    #for this purpose this function generates EVERY chain of heights 1,2, ...,␣
        ↪_T-2 that is possible in a '_T' stage game,
    #then computes both miner's strategies in two subsequent stages and checks if␣
        ↪monotonicity is violated
    #

    for h in range(1, _T-1): #up to height '_T-2'

        #GENERATE POSSIBLE BLOCKCHAINS
        possible_parents = helfun.generatePossibleParents(h)
        possible_winners = helfun.generatePossibleWinners(h, _n)

        #FOR ALL BLOCKCHAINS OF HEIGHT 'h'
        for parents in possible_parents:
            for winners in possible_winners:
                B = bc.Blockchain(_n, parents, winners)

                M = pm.intermediatePayoffMatrix(B, _T, _n)

                nash = pm.matrixNashEq(M) #get the equilibria
                nash_strats = np.transpose(np.where(nash == True)) #get the␣
                    ↪strategy profiles
                nash_strats = np.array([np.array(element) for element in␣
                    ↪nash_strats]) #convert to array

                for strat in nash_strats: #for every possible Nash equilibrium

                    B_ext0 = bc.ExtendedBlockchain(B, 0, strat[0]) #extend the␣
                        ↪chain if miner 0 won
                    B_ext1 = bc.ExtendedBlockchain(B, 1, strat[1]) #extend the␣
                        ↪chain if miner 1 won

                    M_ext0 = pm.intermediatePayoffMatrix(B_ext0, _T, _n) #payoff␣
                        ↪matrix if miner 0 won
                    M_ext1 = pm.intermediatePayoffMatrix(B_ext1, _T, _n) #payoff␣
                        ↪matrix if miner 1 won

                    #GET STRATEGIES FOR BOTH CASES
                    nash0 = pm.matrixNashEq(M_ext0) #case where miner 0 won
                    nash_strats0 = np.transpose(np.where(nash0 == True))
                    nash_strats0 = np.array([np.array(element) for element in␣
                        ↪nash_strats0])
                    nash1 = pm.matrixNashEq(M_ext1) #case where miner 1 won
                    nash_strats1 = np.transpose(np.where(nash1 == True))
                    nash_strats1 = np.array([np.array(element) for element in␣
                        ↪nash_strats1])

                    #CHECK CONJECTURE
                    #see if any miner keeps mining on the same block in the next␣
                        ↪stage
                    #also check if the most recent block is appended to the block␣
                        ↪the miner mined at in the previous stage 't', only␣
                        ↪then are we in a situation where monotonicity could be␣
                        ↪violated
                    for strat0 in nash_strats0: #the case where miner 0 won
                        if strat0[0] == strat[0] and strat0[0] == B_ext0.
                            ↪sequence[-1].parent: #check for miner 0
                            print("Found a counter-example!")
                            return 0
                        if strat0[1] == strat[1] and strat0[1] == B_ext0.
                            ↪sequence[-1].parent: #check for miner 1
                            print("Found a counter-example!")
                            return 0
                    for strat1 in nash_strats1: #the case where miner 1 won
```

```
                        if strat1[0] == strat[0] and strat1[0] == B_ext1.
                         ↪sequence[-1].parent: #check for miner 0
                            print("Found a counter-example!")
                            return 0
                        if strat1[1] == strat[1] and strat1[1] == B_ext1.
                         ↪sequence[-1].parent: #check for miner 1
                            print("Found a counter-example!")
                            return 0

        print(f"checked height {h} in games with T = {_T} stages") #print status
```

B.4.3 Code for Conjecture 3

A miner's behaviour after winning their first block. Again, because we look at two subsequent stages, we consider blockchains with height of up to $T-2$. We run this function in `main.ipynb` (Appendix B.1.3).

[4]:
```
def FirstBlock(_T, _n):
    #
    #check if there are situations where a miner ever switches away from the␣
                        ↪branch they won their first block on
    #for this purpose this function generates EVERY chain where ONE miner won ALL␣
                        ↪blocks of heights 1,2, ..., _T-2 that are possible in␣
                        ↪a '_T' stage game,
    #then computes the other miner's strategy, iterates through the game␣
                        ↪according to the equilibrium strategies and checks␣
                        ↪after each stage if the conjecture is violated
    #

    for h in range(1, _T-1):

        #GENERATE POSSIBLE BLOCKCHAINS
        possible_parents = helfun.generatePossibleParents(h)
        winners = np.zeros(h) #generate a history where one miner (w.l.o.g. miner␣
                        ↪0) won all rounds

        #FOR ALL BLOCKCHAINS OF HEIGHT 'h'
        for parents in possible_parents:
            B = bc.Blockchain(_n, parents, winners)

            M = pm.intermediatePayoffMatrix(B, _T, _n)

            nash = pm.matrixNashEq(M) #get the equilibria
            nash_strats = np.transpose(np.where(nash == True)) #get the strategy␣
                        ↪profiles
            nash_strats = np.array([np.array(element) for element in␣
                        ↪nash_strats]) #convert to array

            for strat in nash_strats: #for every possible Nash equilibrium
                found = recursionFirstBlock(B, _T, 1, strat[1], B.horizon + 1)␣
                        ↪#call recursion when miner 1 wins their first block
                if found == True:
                    print("Found counter-example!")
                    return 0

        print(f"checked height {h} in games with T = {_T} stages") #print status


def recursionFirstBlock(_B, _T, _i, _t, _c):
    #
```

```
    #'_B' is the blockchain of the previous stage
    #'_T' is the number of stages of the game
    #'_i' is the index of the winning miner of the previous stage
    #'_t' is the index of the block that winner chose
    #'_c' is the index of the first block miner 1 won.
    #

    B_ext = bc.ExtendedBlockchain(_B, _i, _t) #extend the chain if miner '_i' won

    if B_ext.horizon == _T: #we have reached the end of the game
        return False

    M_ext = pm.intermediatePayoffMatrix(B_ext, _T, 2)

    nash = pm.matrixNashEq(M_ext) #get a mask of nash equilibria
    nash_strats = np.transpose(np.where(nash == True)) #get the strategy profiles
    nash_strats = np.array([np.array(element) for element in nash_strats])␣
                        ↪#convert to np.array

    found = False
    for strat in nash_strats: #for every possible Nash equilibrium

        #CHECK IF WE FOUND A COUNTER-EXAMPLE
        found = not helfun.isOnSameBranch(B_ext, _c, strat[1])
        if found == True:
            break
        else: #else, go deeper
            found = recursionFirstBlock(B_ext, _T, 0, strat[0], _c) #the case␣
                        ↪where miner 0 wins
            if found == True:
                break
            found = recursionFirstBlock(B_ext, _T, 1, strat[1], _c) #the case␣
                        ↪where miner 1 wins
            if found == True:
                break

    return found
```

B.4.4 Code for Conjecture 4

Switching to shorter chains. Again, because we look at two subsequent stages, we consider blockchains with height of up to $T-2$. We run this function in `main.ipynb` (Appendix B.1.4).

[5]:
```
def Switching(_T, _n):
    #
    #check if there are situations where a miner ever switches to a shorter branch
    #for this purpose this function generates EVERY chain of lengths 1,2, ...,␣
                        ↪_T-2 that is possible in a '_T' stage game, computes␣
                        ↪both miner's strategies,
    #and compares them to their strategies in the subsequent stage
    #

    for h in range(1, _T-1):

        #GENERATE POSSIBLE BLOCKCHAINS
        possible_parents = helfun.generatePossibleParents(h)
        possible_winners = helfun.generatePossibleWinners(h, _n)

        #FOR ALL BLOCKCHAINS OF HEIGHT 'h'
        for parents in possible_parents:
            for winners in possible_winners:
```

```
                B = bc.Blockchain(_n, parents, winners)

                M = pm.intermediatePayoffMatrix(B, _T, _n)

                nash = pm.matrixNashEq(M) #get the equilibria
                nash_strats = np.transpose(np.where(nash == True)) #get the␣
                    ↪strategy profiles
                nash_strats = np.array([np.array(element) for element in␣
                    ↪nash_strats]) #convert to array

                for strat in nash_strats: #for every possible Nash equilibrium

                    B_ext0 = bc.ExtendedBlockchain(B, 0, strat[0]) #extend the␣
                        ↪chain if miner 0 wins
                    B_ext1 = bc.ExtendedBlockchain(B, 1, strat[1]) #extend the␣
                        ↪chain if miner 1 wins

                    M_ext0 = pm.intermediatePayoffMatrix(B_ext0, _T, _n) #payoff␣
                        ↪matrix if miner 0 wins
                    M_ext1 = pm.intermediatePayoffMatrix(B_ext1, _T, _n) #payoff␣
                        ↪matrix if miner 1 wins

                    #GET STRATEGIES FOR BOTH CASES
                    nash0 = pm.matrixNashEq(M_ext0) #case where miner 0 wins
                    nash_strats0 = np.transpose(np.where(nash0 == True))
                    nash_strats0 = np.array([np.array(element) for element in␣
                        ↪nash_strats0])
                    nash1 = pm.matrixNashEq(M_ext1) #case where miner 1 wins
                    nash_strats1 = np.transpose(np.where(nash1 == True))
                    nash_strats1 = np.array([np.array(element) for element in␣
                        ↪nash_strats1])

                    length_before_m0 = B.chainLength(strat[0]) #length of chain␣
                        ↪where miner 0 mined at in stage 't'
                    length_before_m1 = B.chainLength(strat[1]) #length of chain␣
                        ↪where miner 1 mined at in stage 't'

                    #COMPARE THE LENGTH OF THE CHAINS
                    for strat0 in nash_strats0: #the case where miner 0 won
                        if B_ext0.chainLength(strat0[0]) < length_before_m0:␣
                            ↪#comparison for miner 0
                            print("Found a counter-example!")
                            return 0
                        if B_ext0.chainLength(strat0[1]) < length_before_m1:␣
                            ↪#comparison for miner 1
                            print("Found a counter-example!")
                            return 0
                    for strat1 in nash_strats1: #the case where miner 1 won
                        if B_ext1.chainLength(strat1[0]) < length_before_m0:␣
                            ↪#comparison for miner 0
                            print("Found a counter-example!")
                            return 0
                        if B_ext1.chainLength(strat1[1]) < length_before_m1:␣
                            ↪#comparison for miner 1
                            print("Found a counter-example!")
                            return 0

            print(f"checked height {h} in games with T = {_T} stages") #print status
```

B.4.5 Code for Conjecture 5

Coordination. We check if a situation like the one in Table XIII exists. Such a situation exists if and only if one of the strategy profiles in the Cartesian product of possible

strategies in the Nash equilibrium is not a Nash equilibrium. To see this, consider the situation in Table XIII. The Nash equilibria are (b_0, b_0) and (b_1, b_1) and the strategies that could theoretically be played are b_0 and b_1 for both miners. The Cartesian product is $\{(b_0, b_1), (b_0, b_1), (b_1, b_0), (b_1, b_1)\}$, and since (b_0, b_1) is not a Nash equilibrium the conjecture would be violated in this case.

We run this function in `main.ipynb` (Appendix B.1.5).

[6]:
```
def Coordination(_T, _n):
    #
    #check if coordination is ever necessary to play a Nash equilibrium
    #for this purpose it generates EVERY chain of lengths t = 1,2, ..., _T-1 that␣
                        ↪is possible in a '_T' stage game, then computes the␣
                        ↪payoff matrix for each,
    #and checks if coordination is necessary
    #

    for h in range(1, _T):

        #GENERATE POSSIBLE BLOCKCHAINS
        possible_parents = helfun.generatePossibleParents(h)
        possible_winners = helfun.generatePossibleWinners(h, _n)

        #FOR ALL BLOCKCHAINS OF HEIGHT 'h'
        for parents in possible_parents:
            for winners in possible_winners:
                B = bc.Blockchain(_n, parents, winners)

                M = pm.intermediatePayoffMatrix(B, _T, _n)

                nash = pm.matrixNashEq(M) #get the equilibria
                nash_strats = np.transpose(np.where(nash == True)) #get the␣
                        ↪strategy profiles
                nash_strats = np.array([np.array(element) for element in␣
                        ↪nash_strats]) #convert to array

                nash_strats0 = nash_strats[:,0] #extract possible strategies for␣
                        ↪miner 0
                nash_strats1 = nash_strats[:,1] #extract possible strategies for␣
                        ↪miner 1

                for element in itertools.product(nash_strats0, nash_strats1):␣
                        ↪#check if the mask 'nash' is 'False' at any strategy␣
                        ↪profile in the Cartesian product. If yes, then we have␣
                        ↪an example where the conjecture is violated.
                    if nash[element] == False:
                        print("found!")
                        return 0

        print(f"checked stage t = {h+1} in games with T = {_T} stages") #print␣
                        ↪status
```

B.5 Helper Functions Notebook

This notebook includes some useful functions.

[9]:
```
#IMPORTS
#libraries
import numpy as np
import time
import itertools
from treelib import Tree
import import_ipynb


#notebooks
import blockchain as bc
import payoff_matrix as pm
```

The first two functions, `pickWinners()` and `pickParents()`, are exclusively used to test basic functionality of the blockchain classes (they are only used in `tests.ipynb`). They randomly pick a winner among the miners for every stage of a T-stage game, as well as a parent for every block. This can then be used to generate a random blockchain, allowing us to see if the code behaves as expected.

The function `drawChain()` allows us to look at the structure of the blockchain in console output. This makes visualizing what is going on a lot easier. Next, we have two quality-of-life function called `timeElapsed()` and `printPayoffMatrix()`, which make printing passed time and payoff-matrices to the output a lot simpler. Finally, we have the function `isOnSameBranch()`. It simply checks whether two blocks b and c are on the same branch or not.

[10]:
```
def pickWinners(_T, _n):
    #
    #returns an array with the index of the winner of each stage 't' in a␣
                        ↪'_T'-stage game with '_n' players.
    #

    winners = np.random.randint(0, _n, _T) #'winners[t]' is the index of the␣
                        ↪winning miner of the block mined in stage 't+1'

    return winners


def pickParents(_T, _n):
    #
    #draw random parents
    #'_T' is the horizon and '_n' the number of players
    #

    parents = np.full(_T, 0)

    for t in range(1, _T):
        parents[t] = np.random.randint(0, t+1) #'parents[t]' is the index of the␣
                        ↪parent of the block mined in stage 't+1'

    return parents
```

```
def drawChain(_B):
    #
    #draw the blockchain to console output for visualisation
    #'_B' is an object of class 'Blockchain'
    #

    T = _B.horizon

    tree = Tree()
    for t in range(T+1):
        if t == 0:
            tree.create_node(f"b{_B.sequence[0].height}", f"{_B.sequence[0].
                        ↪height}")
        else:
            tree.create_node(f"b{_B.sequence[t].height}", f"{_B.sequence[t].
                        ↪height}", parent=f"{_B.sequence[t].parent}")

    print("")
    tree.show()


def timeElapsed(_start, _end):
    #
    #simply returns a string with the amount of time elapsed between '_start' and␣
                        ↪'_end'
    #
    seconds = round((_end - _start), 1)
    if seconds < 60:
        return f"{seconds} seconds"
    if seconds >= 60 and seconds < 3600:
        return f"{int(round(seconds/60, 1))} minutes"
    if seconds >= 3600:
        return f"{int(round(seconds/3600, 1))} hours"


def printPayoffMatrix(_n, _parents, _winners, _name, _T):
    #
    #prints the blockchain defined by '_parents' and '_winners' together with its␣
                        ↪payoff-matrix to output
    #
    B = bc.Blockchain(_n, _parents, _winners)
    print(f"blockchain {_name}:")
    drawChain(B)
    B.printMiner(0)
    B.printMiner(1)
    print("")
    start = time.process_time()
    print(f"payoff-matrix for blockchain {_name} in {_T}-stage game:\n\n", pm.
                        ↪intermediatePayoffMatrix(B, _T, _n))
    end = time.process_time()
    print("\n", timeElapsed(start, end), "\n")


def isOnSameBranch(_B, _b, _c):
    #
    #simply checks if two blocks are on the same branch in the blockchain'_B',
    #i.e. if we can get to block '_b' by jumping from parent to parent starting␣
                        ↪from block '_c'
    #

    isOnSameBranch = False
    t = _c

    #ITERATE BACKWARDS
```

```
    #starting from the block mined in stage 't', which corresponds to block '_c'
    while _B.sequence[t].height != 0:
        if _B.sequence[t].parent == _b:
            isOnSameBranch = True
            break
        else:
            t = _B.sequence[t].parent

    if _b == _c: #trivial case
        isOnSameBranch = True

    return isOnSameBranch
```

The next two functions are exclusively used in the notebook `conjectures.ipynb` (Appendix B.4). They generate all possible sequences of parents and winners. This is used to iterate through all possible blockchains to test the conjectures.

[11]:
```
def generatePossibleParents(_T):
    #
    #generates all possible sequences of parents for a game with '_T' stages
    #returns an array
    #

    possible_parents = [] #list of lists of parents

    #GENERATE EVERY SEQUENCE
    blocks = np.arange(_T)
    for element in itertools.product(blocks, repeat=_T):

        #CHECK IF ELEMENT IS LEGAL
        legal = True
        for i in range(len(element)):
            if element[i] > i: #parent must have strictly lower index
                legal = False
                break
        if legal == False:
            continue

        possible_parents.append(list(element)) #append legal combination

    possible_parents = np.array(possible_parents)

    return possible_parents


def generatePossibleWinners(_T, _n):
    #
    #generates all possible sequences of winners for a game with '_n' miners and␣
                        ↪ '_T' stages
    #returns an array
    #

    possible_winners = [] #list of lists of winners

    #GENERATE EVERY SEQUENCE
    miners = np.arange(_n)
    for combination in itertools.combinations_with_replacement(miners, _T):
        for permutation in itertools.permutations(combination, _T):
            if permutation[0] == 1: #w.l.o.g. m0 wins the first stage, hence a␣
                        ↪situation where m1 wins is skipped
                #we just need to make sure to check the strategies for BOTH␣
                        ↪miners later, since we exclude half the game-tree here
                continue
            possible_winners.append(list(permutation))
```

```
    possible_winners = np.array(possible_winners)
    possible_winners = np.unique(possible_winners, axis=0) #kick duplicate␣
                         ↪entries from array, this is needed because the␣
                         ↪permutations consider winning stage 2 and stage 1␣
                         ↪different from winning stage 1 and stage 2, even tough␣
                         ↪they are in fact the same

    return possible_winners
```

B.6 Tests

This notebook runs some tests to verify that our code is working as intended. It was used a lot during development to iron out bugs. It is important to note that testing in our case is not trivial. The best we can do is to check that the code works properly for games with $T = 2$ and $T = 3$ stages and trust in our ability to write code that generalizes to any number of stages. However, basic functionality like constructing and extending the blockchain is easily verified.

[1]:
```
#IMPORTS
#libraries
import numpy as np
import time
import import_ipynb

#notebooks
import blockchain as bc
import helper_functions as helfun
import payoff_matrix as pm
```

B.6.1 Draw a Random Blockchain

Print the miners, blocks, the blockchain, and the longest chain.

[2]:
```
T = 7 #time horizon
n = 2 #number of players

#SET A SEED
seed = int(np.random.randint(0, 1000, 1)[0])
print("seed:", seed, "\n")
np.random.seed(seed)

#GENERATE A RANDOM VALID BLOCKCHAIN
parents = helfun.pickParents(T, n)
winners = helfun.pickWinners(T, n)

B = bc.Blockchain(n, parents, winners) #build

#PRINT THE MINERS
B.printMiner(0)
B.printMiner(1)
print("")
```

```
#PRINT ALL BLOCKS
for t in range(1, T+1): #for each element in the blockchain
    B.sequence[t].printBlock()

#DRAW THE BLOCKCHAIN USING TREELIB
helfun.drawChain(B)
print("longest chains", B.longestchains)
```

```
seed: 994

miner 0 wins stages [2 3 4]
miner 1 wins stages [1 5 6 7]

block 1 mined by miner 1, parent is block 0
block 2 mined by miner 0, parent is block 0
block 3 mined by miner 0, parent is block 1
block 4 mined by miner 0, parent is block 3
block 5 mined by miner 1, parent is block 1
block 6 mined by miner 1, parent is block 5
block 7 mined by miner 1, parent is block 5

b0
├── b1
│   ├── b3
│   │   └── b4
│   └── b5
│       ├── b6
│       └── b7
└── b2

longest chains [[0, 1, 3, 4], [0, 1, 5, 6], [0, 1, 5, 7]]
```

B.6.2 Payoff Calculation

[3]:
```
#PRINT THE NUMBER OF BLOCKS EACH MINER MINED
#we use the random blockchain generated above
for m in range(2):
    print(f"\nthe number of blocks mined by miner {m} in the chain ending in...")
    for t in range(1, T+1):
        print(f"... block {t} is {B.getPayoff(m, t)}")
print("")

#'TEST' THE FUNCTION 'finalExpectedPayoff()'
#calculates the expected payoff at the end of the game BEFORE nature chooses a␣
                            ↪the 'winning' chain
for m in range(2):
    print(f"final expected payoff for miner {m} is", pm.finalExpectedPayoff(B, m))
```

```
the number of blocks mined by miner 0 in the chain ending in…
... block 1 is 0
... block 2 is 1
... block 3 is 1
... block 4 is 2
... block 5 is 0
... block 6 is 0
... block 7 is 0

the number of blocks mined by miner 1 in the chain ending in…
... block 1 is 1
... block 2 is 0
... block 3 is 1
... block 4 is 1
... block 5 is 2
```

```
... block 6 is 3
... block 7 is 3

final expected payoff for miner 0 is 0.6666666666666666
final expected payoff for miner 1 is 2.3333333333333335
```

B.6.3 Blockchain Extension

Take the random blockchain from above and append a block to it. Then print the new blockchain.

[4]:
```
m = 0 #index of the miner who wins the next stage, arbitrarily chosen as '0'
t = T #index of the block the winning miner mines at, also arbitrarily chosen as␣
                          ↪ 'T'

B_ext = bc.ExtendedBlockchain(B, m, t)

print("extended chain:")
B_ext.printMiner(0)
B_ext.printMiner(1)
helfun.drawChain(B_ext)
```

```
extended chain:
miner 0 wins stages [2 3 4 8]
miner 1 wins stages [1 5 6 7]

b0
├── b1
│   ├── b3
│   │   └── b4
│   └── b5
│       ├── b6
│       └── b7
│           └── b8
└── b2
```

B.6.4 Payoff Matrix and Strategies

Print a payoff matrix and calculate the strategies for each player given that payoff matrix.

This section was used extensively to verify that the payoff calculation works. We calculated payoffs for all situations in the two- and three-stage games manually (on paper), and subsequently checked if the code is able to verify our results. This process obviously took a lot of time. How one arrives at the payoff matrices is explained in sections Section 4.1.1 and Section 4.2.2.

The reader may experiment with different blockchains. Some examples are given below for blockchains with two, three, and four blocks. More examples can be found

in `main.ipynb` (Appendix B.1.6). Any payoff matrix can be calculated with relative ease. Please be aware that the program will take a considerable amount of time when T is much larger than the number of blocks in the specified blockchain (i.e. when looking many stages into the future).

[5]:
```
#define variables
T = 4 #time horizon of the game
n = 2 #number of players, DO NOT CHANGE THIS!

#EXAMPLES
#BLOCKCHAIN WITH TWO BLOCKS
parents = np.array([0])
winners = np.array([0]) #miner 0 wins the first stage

#BLOCKCHAIN WITH THREE BLOCKS
#parents = np.array([0, 0]) #both blocks b1 and b2 are appended to the genesis␣
                            ↪block b0
#winners = np.array([0, 1]) #miner 0 wins the first stage, miner 1 the second

#BLOCKCHAIN WITH FOUR BLOCKS
#parents = np.array([0, 0, 2]) #blocks b1 and b2 are appended to the genesis␣
                            ↪block b0, b3 is appended to b2
#winners = np.array([0, 1, 1]) #miner 0 wins the first stage, miner 1 the second␣
                            ↪and third


### ASSERTIONS ###
assert winners.shape[0] == parents.shape[0], "arrays 'parents' and 'winners' must␣
                            ↪be of same length"
t = len(parents) + 1 #current stage
for s in range(t-1):
    assert parents[s] < s+1, "the parent of a block b_t must have a strictly␣
                            ↪lower index s < t"


print('T =', T)

B = bc.Blockchain(n, parents, winners)
helfun.drawChain(B)

B.printMiner(0)
B.printMiner(1)

start = time.process_time()
M = pm.intermediatePayoffMatrix(B, T, n)
end = time.process_time()

print(f"list of Nash equilibrium tuple(s) in stage {t}:", pm.getStrategies(B, T,␣
                            ↪n))
print("\npayoff matrix:\n\n", M)
print("\n", helfun.timeElapsed(start, end))
```

```
T = 4

b0
└── b1

miner 0 wins stages [1]
miner 1 wins stages []
list of Nash equilibrium tuple(s) in stage 2: [(1, 1)]

payoff matrix:
```

```
[[[1.6875 1.1875]
  [2.     1.5   ]]

[[2.1875 1.1875]
  [2.5    1.5   ]]]

0.2 seconds
```

C Declaration of Authorship

I hereby confirm that this thesis was written in accordance with the Code of Honor of the University of Zurich. That is, I wrote this thesis by myself and without receiving any inadmissible support.

Kenan Öztürk, 24.07.2023