Decentralized Finance and Blockchains

Sander Gribling Pieter Kleer Nikolaus Schweizer

Table of contents

About **2**
Course description . . . 2
Learning goals . . . 2
Acknowledgments . . . 3
Contact information . . . 3

1 Introduction **4**
1.1 A computer science perspective . . . 5
1.2 A game theory perspective . . . 7
1.3 A financial economics perspective . . . 7

2 Preliminaries **9**
2.1 Commonly used symbols . . . 9
2.2 Graph theory . . . 9
2.3 Real analysis . . . 9
2.4 Concave maximization . . . 10

I Computer Science **12**

3 Introduction to distributed computing **14**
3.1 Example: maintaining copies of a file . . . 14
3.2 State Machine Replication problem . . . 15
3.2.1 Protocol achieving consistency . . . 16
3.2.2 Protocol achieving liveness . . . 16
3.3 (Strong) assumptions about the decentralized setting . . . 18
3.3.1 Assumption 1: the set of nodes is known . . . 18
3.3.2 Assumption 2: signatures exist and cannot be forged . . . 18
3.3.3 Assumption 3: synchronous model . . . 18
3.3.4 Discussion of the assumptions . . . 19
3.4 Honest nodes . . . 19
3.4.1 Solving the SMR problem under assumptions 1,2,3,4 . . . 19

4 The Dolev-Strong protocol **21**
4.1 Bounded number of faulty nodes . . . 21
4.2 The Byzantine Broadcast problem . . . 21
4.3 SMR reduces to Byzantine Broadcast . . . 22

4.4 The cases $f = 1$ and $f = 2$ . . . 23
4.5 The Dolev-Strong protocol . . . 24
4.5.1 Convincing messages . . . 24
4.5.2 Protocol description . . . 25
4.5.3 Proof of correctness . . . 25

5 Longest-chain protocols **27**
5.1 Protocol description . . . 27
5.1.1 Honest vs. dishonest behavior . . . 29
5.2 The assumptions . . . 31
5.2.1 Assumption A1: the genesis block is unknown prior to starting the protocol . . . 31
5.2.2 Assumptions about leader selection . . . 32
5.2.3 Assumptions about block production . . . 32
5.2.4 Assumptions about communication . . . 33
5.3 The goals: liveness and *finality* . . . 33
5.4 Finalizing a block . . . 34
5.5 Balanced leader sequences . . . 36
5.5.1 Random leaders are balanced: the intuition . . . 37
5.5.2 Random leaders are balanced: the math . . . 40
5.5.3 Random leaders are balanced: the takeaway message . . . 44
5.6 The common prefix property . . . 46
5.6.1 Proof of Theorem 5.4 . . . 46
5.7 Finality of longest-chain consensus . . . 48
5.8 Liveness of longest-chain consensus . . . 49
5.9 Toward permissionless consensus . . . 50

6 Proof of Work **52**
6.1 What is proof of work? . . . 52
6.2 Cryptographic hash functions . . . 53
6.2.1 The random oracle assumption . . . 53
6.3 SHA-256 . . . 54
6.4 How do we define a puzzle based on SHA-256? . . . 55
6.5 What should puzzle solutions encode? . . . 57
6.6 Sybil-resistance . . . 58
6.7 Nakamoto consensus . . . 58
6.7.1 How to set and adjust the difficulty parameter . . . 59
6.8 Safe blocks . . . 60

II Game Theory **62**

7 Proportional rule **64**
7.1 Axioms . . . 65
7.2 Characterization . . . 67
7.3 Risk aversion . . . 69
7.3.1 Concave utility function . . . 70
7.3.2 Modified axioms . . . 71
7.3.3 Impossibility result . . . 72

7.4 Acknowledgements . . . 72

8 Tullock contest **73**
8.1 Tullock contest . . . 74
8.2 Pure Nash equilibrium . . . 75
8.2.1 Computation for $n = 2$ . . . 78
8.2.2 Computation for $n > 2$ . . . 78
8.3 Best response dynamics . . . 81
8.3.1 Potential function . . . 86
8.3.2 Convergence to ϵ-PNE . . . 89
8.4 Two model variations . . . 90
8.4.1 Concave utilities . . . 90
8.4.2 Economies of scale . . . 94
8.5 Acknowledgements . . . 95

9 Selfish mining attack **97**
9.1 Revisiting longest chain protocol . . . 98
9.1.1 Mining time . . . 99
9.2 Warm-up example . . . 100
9.2.1 Secret mining details . . . 105
9.3 General case . . . 106
9.3.1 Pseudo-code . . . 107
9.3.2 State space . . . 110
9.3.3 Transition probabilities . . . 110
9.3.4 Stationary distribution . . . 115
9.3.5 Reward analysis . . . 116
9.4 Acknowledgements . . . 118

About

Course description

Welcome to the course “Decentralized Finance and Blockchains”!

This course approaches decentralized finance and blockchains from three different angles: the computer science or cryptography angle (CS), the game-theoretic or incentives angle (GT), and the financial economics angle (FE).

The three different parts are taught by three different lecturers: Sander Gribling (CS), Pieter Kleer (GT), and Nikolaus Schweizer (FE). Likewise, these lecture notes are also split into three parts. Below, you can find a brief description of each of the three angles, as well as the learning goals of the course.

Computer Science: A key principle underlying blockchain technology is maintaining consensus about a distributed ledger, the archive of past transactions. We will study the security aspects related to establishing consensus. We discuss various (im)possibility results in the presence of malicious agents (Byzantine Fault Tolerance) in the general setting of distributed networks. We then discuss the mathematical models behind two famous mechanisms to maintain consensus: Proof of Work and Proof of Stake.

Game Theory: Game theory and mechanism design play an important role in the analysis and design of decentralized financial protocols such as those building on blockchain technology. Prominent examples here are cryptocurrencies like Bitcoin. We will be studying such protocols from a game-theoretical perspective by looking at equilibria of their mathematical description, as well as various mechanisms that are used to guide those systems to the desired outcomes.

Financial Economics: One of the most intriguing capabilities of the blockchain technology are so-called smart contracts, computer programs that run on the blockchain in a transparent and decentralized manner, thus providing the basis for decentralized finance, the creation of decentralized counterparts to traditional financial institutions. In recent years, a particular success have been decentralized exchanges such as Uniswap which run in the blockchain and replace the traditional market maker and order book of a centralized exchange with an automated market maker. In the course, we will study and compare both types of exchanges using tools from the classical theory of market microstructure and analyze how different aspects of their design affect the functioning of the resulting financial market.

Learning goals

After successful completion of this course, you are able to:

1. Model blockchain technologies from the perspective of distributed computing and prove the (im)possibility of Byzantine Fault Tolerance under various assumptions.

2. Explain consensus protocols in distributed networks and compute the probability that malicious agents can change the outcome (e.g. for the Dolev-Strong and longest-chain consensus protocols).
3. Model decentralized financial protocols from a game-theoretical perspective and compute their equilibria.
4. Explain how mechanism design and game theory can be used to create the desired incentives in decentralized systems and compute the outcome of the relevant mechanisms.
5. Analyze both centralized and decentralized financial exchanges using tools from market microstructure theory.
6. Explain how the design of a financial exchange affects the properties of the resulting financial market regarding, e.g., liquidity and absence of arbitrage.

Acknowledgments

None of the results presented in this course are the authors' own work; all materials are based on books, lectures notes and research papers present in the literature. We will acknowledge the original results that every part is based on, in the respective part of the course.

Python code snippets were sometimes generated using ChatGPT.

Contact information

Lecturers:

- Sander Gribling
- Pieter Kleer
- Nikolaus Schweizer (course coordinator)

Note that this is a new course, it is likely that we can still improve the clarity of the exposition. We therefore welcome constructive feedback on these lecture notes either via email, Canvas, or in class.

Chapter 1

Introduction

This course is about the science behind blockchain protocols and their applications. The most famous applications being of course cryptocurrencies such as Bitcoin.

Let us first make a disclaimer: in this you will not learn anything about trading in cryptocurrencies. Instead, you will (hopefully) learn the principles behind blockchain protocols and the problems that frequently arise in applications.

At this point, you probably have some rough idea of what a blockchain is and how cryptocurrencies like Bitcoin work, perhaps based on reading some news articles or watching a short video on social media. For the purpose of this course, it is useful to have the following (simplified) picture in mind.

- For a standard currency, for example the euro, there is a central banking agency that can perform tasks such as certifying that a given coin (or entry on a bank account) is indeed a valid euro, or certifying transactions.
- For a cryptocurrency, the guiding principle is often that there is no such central agency. Instead, the current status of "bank accounts" and transactions are maintained in a ***decentralized*** fashion.

The precise meaning of *decentralized* is something that we will explore in this course. For now, imagine that it means that every participant in the (crypto)currency knows the exact amount on every other bank account, as well as a full list of the transactions that have ever taken place.

We often think about the set of participants as a network consisting of *nodes* and *edges*. The nodes represent the participants (we sometimes also referred to participants as parties). The edges represent communication links between two parties. Figure 1.1 below shows an example of a network with three nodes, who can all communicate with each other.

In cryptocurrency applications, it is often convenient (and realistic) to assume that every node can communicate with every other node. This means that all edges are present. This is however not a realistic assumption in every application: if the nodes represent people at a Dutch birthday party for example, then they are likely seated in a circle and can only communicate to their nearest neighbors. While this might seem like a silly example, the resulting network is a common toy model with interesting properties that we will revisit later on.

Every node maintaining a complete list of all bank accounts and transactions might seem like a lot to ask for, and indeed, it is. It is however a useful picture to have in mind. It for example requires us to solve the following problems:

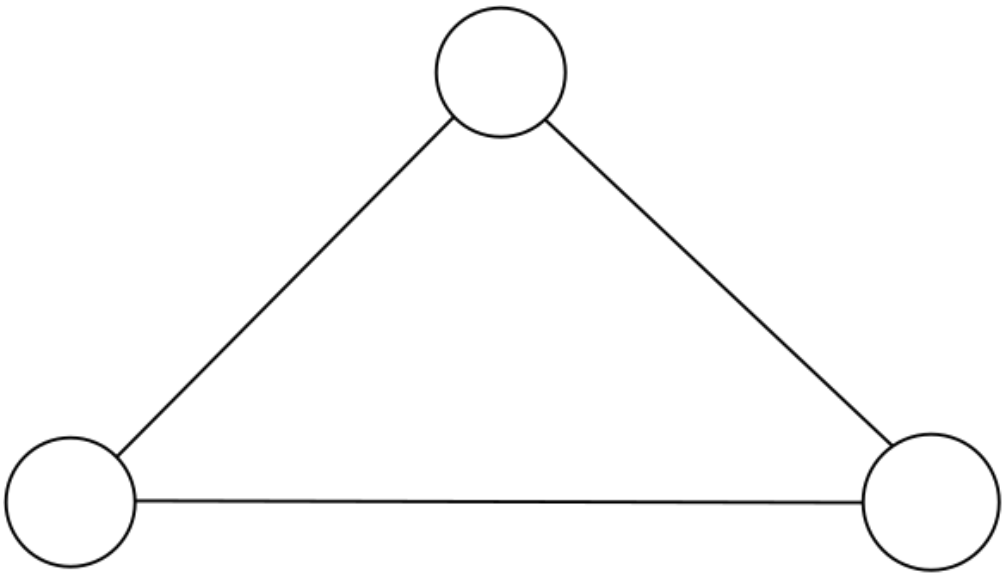


Figure 1.1: A network with three nodes

- If a transaction is to take place, say Alice transfers money to Bob, how do we ensure that everyone updates their records correctly?
- Can we do it in such a way that Alice cannot "double-spend" her money, i.e., also give it to Charlie?

It is not hard to imagine that it is a lot of work to maintain knowledge in a decentralized fashion, indeed, it will take us several chapters to do so! A natural question is therefore, how do we *incentivize* parties to perform this work? If you have ever read a popular science article about Bitcoin, you might have heard the terms "Proof of Work" or "Bitcoin mining" (if not, we will get to this in Chapter 6). These are examples in which parties perform some action (work, e.g., mining a Bitcoin) and are rewarded for this work. Often, the task is too large to perform by a single party and therefore parties form *coalitions*. How do we divide rewards in this case? This is one of the topics covered in Part II.

Once one has settled on the technology and protocols behind a blockchain-based currency, we can turn to applications. This is what Part III is all about.

In the remainder of this introductory chapter, we will give a high-level overview of the three different perspectives on blockchains and their applications that we will discuss in this course. At a first reading, we do not expect you to understand everything in these sections. As the course progresses, you should be able to answer more and more of the questions raised in these sections. (Let us know if we forgot to answer some!)

1.1 A computer science perspective

In this part of the course we will focus on the science behind blockchain protocols. Let's start by unpacking the terminology "blockchain protocols" a bit.

The "protocols" here refers to a set of instructions that all participants have to follow. To make it concrete, you think of it as piece of code (a computer program) that all parties have to execute.[1] Such a protocol is designed to perform a certain task. In this course, we will focus on what we want a protocol to do, rather than

[1] For example, look at all information currently available to the participant, do some computation, and report the outcome to all neighboring participants.

how to implement it on a computer. (In other words, there will not be much coding in this course.) In the case of blockchain protocols, the task is to ensure that all parties (eventually) agree on what is written in the blockchain. This last sentence is intentionally a bit vague, we will be more precise later on about what we mean by "eventually" and "agree".

A "blockchain" simply refers to a chain of blocks. The blocks can be used to store information; again, we will be more precise about which kind of information later on. We want to think of a block all information that is available at a certain moment in time. Logically, we would then like to connect a block to the block that preceded it in time. We can thus think of a chain of blocks as a special kind of network. As opposed to the networks that we described before, a chain requires a *directed* network. What do we mean by *directed*? In our previous description of a network, an edge represented a "communication link" and we implicitly assumed that if, say, Alice can communicate with Bob, then Bob can also communicate with Alice. In a directed network the direction of edges matters. Visually, we will represent this using arrows. If there is an edge between nodes 2 and 1, then we draw an arrow from node 2 to node 1. Here is an example of a chain of three blocks.

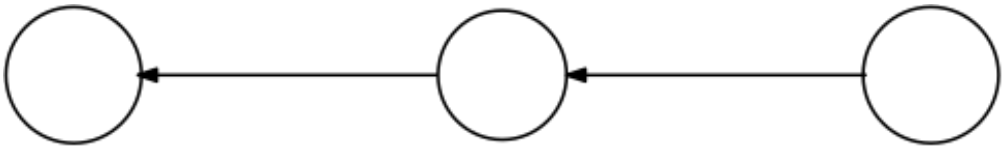


Figure 1.2: A chain with three blocks

We will work towards an understanding of blockchains gradually. We will first get familiar with computation in a decentralized setting. To start, we should decide on a mathematical model for the decentralized setting (or several models). Here one can think for example about questions such as how do we model communication? Is communication instantaneous (the synchronous model), or can there be delays (an asynchronous model)? Do we know all participants in the network in advance or can anyone participate? All of these lead to valid models! We will therefore be explicit about which *assumptions* we are making. When we make an assumption, we should always ask ourselves whether it is a reasonable or realistic assumption. Whether an assumption is realistic or not often depends on the application (instantaneous communication anyone?). Exploring what is and what is not possible under a given set of assumptions is essentially the first part of the course.

To be more specific, we will focus on something we have touched upon before: "agreeing" on information. In the literature we often refer to reaching agreement as "building consensus". You might see "Proof of Work" and/or "Proof of stake" referred to as consensus-building protocols. The concept of consensus is however separate from blockchains and indeed much older. At this point you should wonder, if everyone can communicate with everyone and we are all following the same protocol, how can we ever disagree? This is a very good question! The answer is that it is often very unreasonable to assume that everyone follows the same protocol! For example, if by deviating from the Bitcoin protocol you could earn a lot of money, then probably someone will decide to deviate. We thus need to design protocols that can resist such dishonest parties, allowing the honest parties to reach consensus. We will call dishonest parties "Byzantine agents" later on. One of the problems that we will discuss is the Byzantine broadcast problem.

The general area to which these questions belong is that of ***distributed computing***. Distributed computing is much broader than what we can cover in this course. Nevertheless, after setting up the framework more formally, we will quickly be able to discuss some foundational results in distributed computing: the state machine replication problem, and the Dolev-Strong protocol for Byzantine broadcast. These are results about the synchronous setting, assuming we know the entire network in advance. Neither of these assumptions is very realistic for blockchains, but is a useful starting point due to its simplicity. It will allow us to quickly get

our hands dirty, without getting lost in the details of blockchains.

Having said that, we then move to precisely this: the details of blockchain protocols. More precisely, we will discuss longest chain protocols as a general framework for building consensus in a network that we do not know in advance. We finally discuss in more detail one particular longest chain protocol: the one that is built on the concept of "Proof of work", used for example in Bitcoin where "mining a Bitcoin" is a "proof of work".

1.2 A game theory perspective

This part of the course will be concerned with incentives and strategical aspects that may arise in blockchain protocols. We will illustrate these concepts here using using the example of "Bitcoin mining". This is done by solving a complex mathematical puzzle that requires a lot of computation power and is typically done by multiple parties or miners.

Once a Bitcoin is mined, i.e., created, how do we split it fairly between the miners that were involved? A function that decides on the reward that every miner receives is called an allocation rule and can be defined in many ways. Ideally, we want the allocation rule to have some desirable properties that incentivize miners to act faithfully. One such property is sybil-proofness: A miner should not have an incentive to split itself up in multiple parties and receive, in total, more reward from the allocation rule then it would have received when participating as one single party or miner. Reversely, we would also like the allocation rule to be collusion-proof meaning that different miners should not receive more total reward in the allocation rule if they pretend to be one miner, as opposed to the sum of their individual rewards. Our goal here will be to understand and characterize which allocation rules satisfy these, and other, desirable properties by looking at this problem through the lens of cooperative game theory.

One can also take a more competitive view towards Bitcoin mining by considering it to be a so-called Tullock contest. Here every miner invests a certain amount of computing power, but instead of sharing the reward generated by mining a Bitcoin, the Tullock contest gives the whole Bitcoin to the first miner to solve the mathematical puzzle. The probability for a miner to win this contest depends on the amount of computing power that was invested. The miners have a strategic choice to make on how much computing power they want to invest, while optimizing their chance of winning the contest. Our goal will be to analyze this contest through the lens of non-cooperative game theory using stability concepts such as the Nash equilibrium.

1.3 A financial economics perspective

From a financial economics perspective, one of the greatest promises and challenges of the blockchain technology is the possibility of organizing counterparts to traditional financial institutions in decentralized ways. An important impulse for this development was the great financial crisis of 2008 when financial institutions that were considered "too big to fail" placed a huge burden on societies and created considerable distrust in traditional "centralized" finance. After the invention of bitcoin, a second important step towards decentralizing financial institutions was the invention of so-called smart contracts, computer programs that can be embedded in later-generation blockchains like Ethereum and that can be used to create automated protocols for financial transactions.

Since the advent of smart contracts, people have looked into various ways of creating decentralized counterparts to traditional financial institutions using smart contracts. Yet there are challenges, some of them as old as financial markets, some of them rather new. Think of decentralized car insurance, implemented as a computer program that automatically sends you an agreed upon amount of cryptocurrency when you upload a photo

of your damaged car. What could possibly go wrong? There is more than one answer to this question. Let us just say that, traditionally, most successful insurance markets have operated in environments with a well-functioning legal system in the background.[2]

Automated market makers such as Uniswap are one of the most successful developments in decentralized finance, exhibiting tremendous trading volumes. These are automated exchanges that enable market participants to exchange assets with an automated mechanism that was inspired by the way sports betting markets are organized. Again, traders in these exchanges face some of the same challenges seen in traditional financial exchanges. Yet, there are also new challenges. For instance, due to the full transparency and the block structure of transactions, submitted orders can be seen by others before they are executed – and the order of submissions is not necessarily the order of executions, creating all sorts of potential problems.

The goal in the financial economics part of the course is to get some understanding of the relevant decentralized financial institutions and then take some steps towards understanding them even better by applying models from quantitative finance.

[2] Conversely, in general, the *potential* benefits of decentralized financial institutions may be greatest in environments without well-functioning legal systems, where citizens have to be concerned that those in power confiscate their traditional bank accounts and so forth.

Chapter 2

Preliminaries

Here we gather frequently used notation. We also recall some basic concepts that we expect as prior knowledge and we provide some pointers to related literature.

2.1 Commonly used symbols

We let $\mathbb{N}$ denote the set of natural numbers, i.e., $\mathbb{N} := \{1, 2, 3, ...\}$. For $n \in \mathbb{N}$, we write $[n] := \{1, 2, \dots, n\}$.

We let $\mathbb{R}$ denote the set of all real numbers and $\mathbb{R}_{\geq 0}$ the set of all nonnegative real numbers.

We use $\log()$ to denote the natural logarithm and $\log_2()$ to denote the logarithm with base 2.

2.2 Graph theory

We will use the concepts of undirected and directed graphs, which we recall below.

A *graph* G is defined by a set of vertices V and a set of edges E. We write $G = (V, E)$. Here the set of edges is a subset of pairs of vertices. Throughout, we assume an edge consists of a pair of distinct vertices $u, v \in V$ (distinct meaning $u \neq v$). When such an edge is *undirected* we write $\{u, v\}$. When the edge is directed we write (u, v) to denote an edge from vertex u to vertex v. We have seen an example of an undirected graph in Figure 1.1: this is the graph with vertex set $V = [3]$ and edge set $E = \{\{1, 2\}, \{2, 3\}, \{1, 3\}\}$.

We have also seen an example of a directed graph in Figure 1.2: here the vertex set is again $V = [3]$ and if we label the vertices 1, 2, and 3 from left to right, then the edge set is $E = \{(2, 1), (3, 2)\}$.

2.3 Real analysis

For a given *domain* $A \subseteq \mathbb{R}^n$, a *function* f maps every number $x = (x_1, \dots, x_n) \in A$ to a number $f(x) \in \mathbb{R}^m$. We denote this mapping by $f : A \to \mathbb{R}^m$. We assume familiarity with basic concepts from real analysis such as continuity and differentiability of functions, as well as limit calculations, in n-dimensional (Euclidean) space.

If $m = 1$ and f is differentiable, we denote the *gradient* of f by

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \dots, \frac{\partial f}{\partial x_n} \right)$$

which is the vector containing the partial derivatives of f.

A one-dimensional function $f : A \to \mathbb{R}$ is *increasing* on $A \subseteq \mathbb{R}$ if $f(x) \leq f(y)$ for all $x, y \in A$ with $x \leq y$. If f is differentiable on A, then f is increasing if and only if the derivative of f is non-negative on (the interior of) A.

2.4 Concave maximization

For $A \subseteq \mathbb{R}^n$, we say that a function $f : A \to \mathbb{R}^m$ is *concave* on A if for all $x, y \in A$ and $\lambda \in (0, 1)$ it holds that

$$\lambda f(x) + (1 - \lambda) f(y) \leq f(\lambda x + (1 - \lambda) y).$$

An example of a concave functions is an *affine/linear function*, which is of the form

$$f(x) = a_1 x_1 + a_2 x_2 + \dots a_n x_n - b.$$

A function f is *strictly concave* if the inequality above is strict ($<$ instead of $\leq$) for every $x, y \in A$ and $\lambda \in (0, 1)$. Note that an affine function is not strictly concave.

A differentiable, univariate function $f : A \to \mathbb{R}$ is concave on A if and only if $f'(x)$ is a nonincreasing function on A.

A (strictly) concave function is nice to optimize over: There are neat characterizations determining when a point $x \in A$ is a maximizer, i.e., we can characterize the solutions to the problem

$$\max_{x \in A} f(x).$$

We will consider two such cases here: The unconstrained case where $A = \mathbb{R}^n$ and the case where $A \subseteq \mathbb{R}^n$ can be described by a set of linear (in)equalities. If $A = \mathbb{R}^n$, then x is a maximizer of f if and only if $\nabla f(x) = 0$.

If A is described by a set of q linear inequality and r linear equality constraints, the maximization problem is of the form

$$\begin{array}{lll} \max_{x \in \mathbb{R}^n} & f(x) & \\ \text{subject to} & g_i(x) \leq 0 & \text{for } i = 1, \dots, q \\ & h_j(x) = 0 & \text{for } j = 1, \dots, r \end{array}$$

where the g_i and h_j are affine functions.

We can use the well known Karush-Kuhn-Tucker (KKT) conditions to characterize a maximizer of f over A. To state these conditions, we introduce for every inequality constraint a (dual) multiplier $\lambda_i \leq 0$ for $i = 1, \dots, q$, summarized in the vector $\lambda = (\lambda_1, \dots, \lambda_q)$, and (dual) multiplier $\nu_j \in \mathbb{R}$ for $j = 1, \dots, r$, summarized in the vector $\nu = (\nu_1, \dots, \nu_r)$. The difference in domain of the multipliers, $\lambda_i \leq 0$ and $\mu_j \in \mathbb{R}$, comes from the fact that they are identified with different type of constraints, inequalities and equalities, respectively.

We define the *Lagrangian* as

$$L(x, \lambda, \nu) = f(x) + \sum_{i=1}^{q} \lambda_i g_i(x) + \sum_{j=1}^{r} \nu_j h_j(x).$$

Then $x \in A$ is a maximizer of f if and only if there exists vectors $\lambda \in \mathbb{R}^q$ and $\nu \in \mathbb{R}^r$ such that the point (x, λ, ν) satisfies the following four KKT conditions:

1. **(Zero gradient).** The partial derivatives of L with respect to x are zero, i.e.,

$$\frac{\partial L(x, \lambda, \nu)}{\partial x_k} = \frac{\partial f(x)}{\partial x_k} + \sum_{i=1}^{q} \lambda_i \frac{\partial g_i(x)}{\partial x_k} + \sum_{j=1}^{r} \nu_j \frac{\partial h_j(x)}{\partial x_k} = 0$$

   for $k = 1, \ldots, n$.
2. **(Primal feasibility).** $g_i(x) \leq 0$ for $i = 1, \ldots, q$ and $h_j(x) = 0$ for $j = 1, \ldots, r$.
3. **(Dual feasibility).** $\lambda_i \leq 0$ for $i = 1, \ldots, q$.
4. **(Complementary slackness).** $\lambda_i g_i(x) = 0$ for $i = 1, \ldots, q$.

Note that if there are no constraints, these conditions simply reduce to the statement that the gradient of f should be zero. The complementarity conditions imply that if a dual variable $\lambda_i > 0$, then the corresponding constraint $g_i(x) \leq 0$ must be binding, i.e., equality must hold in the point x. Similarly, if a constraint is not binding in a point x, i.e., $g_i(x) < 0$, then the corresponding dual variable $\lambda_i = 0$.

Part I

Computer Science

The computer science part of this course is heavily inspired by a course taught by Tim Roughgarden called "Foundations of Blockchain Protocols". You can find this course here: Course website, Lecture notes.

Note that Roughgarden's course is geared towards computer science students. The expected prior knowledge is thus very different compared to this course. The scope of Roughgarden's course is therefore also much larger than ours: it covers the computer science angle in much more depth than we do here. If you want to learn (much) more about the CS perspective, this is a great starting point.

Chapter 3

Introduction to distributed computing

In this chapter we will define some basic concepts and problems from the area of distributed computing.

We start this chapter with an example of a common distributed computing problem: that of making backups of a database and ensuring that these backups are synchronized. This is a problem that you have probably struggled with at some point: how do you properly backup the files on your hard disk? To look ahead, the problem that we aim to solve when we want to base cryptocurrencies on blockchains is very similar: the database would correspond to a complete list of the transactions that have ever occurred.

We then give a formal definition of the underlying problem: the State Machine Replication (SMR) problem. The second half of this chapter is devoted to formalizing and discussing various assumptions about distributed computing networks. The next chapters are dedicated to solutions of the SMR problem under increasingly more realistic assumptions.

3.1 Example: maintaining copies of a file

To set the stage, consider the following situation. On my computer, I have an excel file with the grades of all students who take this course. Naturally, I would like to make sure that I don't lose this file. To do so, I can create multiple copies of this file and store them on different computers. That way, if one of the computers breaks, I still have a copy of the file on another computer. Making such a backup once, is easy enough. The problem that we will consider arises when I want to be able to update the file (for example, after the resit) in such a way that the various backups to agree with each other.

We start to see an outline of a distributed computing problem: we can view the different computers as nodes in a graph. If there are n computers, there would be n nodes in the graph. Our task would be to ensure that the n computers each have an up to date copy of the file.

The above example is written from a *centralized* perspective. There is one agent, the "I" persona, that can simply perform the update to each copy of the file. In a *decentralized* setting, we would like the computers to run some sort of protocol that ensures that if I change the file on one of the computers, then the same change is made on all other devices. One quickly realizes that this protocol will require the devices to communicate. This brings us to the second part of the graph: the communication links between the computers determine the edge set of the graph.[1]

[1] In this problem, it is natural to assume that communication works both ways. Therefore the graph would be undirected.

Connection to blockchains: If we want to construct a cryptocurrency based on blockchains, then we are essentially interested in solving a similar type of problem. Indeed, we can draw the following analogy. Each participant in the cryptocurrency would correspond to a node in the network. The file that we want each participant to maintain consists of two parts: a list of the current balance of each account, and a full history of all transactions that have ever taken place between participants. In this variant of the problem, there is no natural "I" persona. The decentralized setting is built in: we want each of the nodes to be able to change the file. We do of course want all the nodes to agree on the same file! A change made by one of the nodes should be replicated by all other nodes.

In the next section we will formally define a generalization of the above examples.

3.2 State Machine Replication problem

Let us now give a formal definition of a problem that encompasses both examples from the previous section. In the distributed computing literature this problem is known as the State Machine Replication (SMR) problem. To explain the terminology: the state of the machine corresponds to the file that we wanted to maintain in the previous examples.

In the SMR problem we consider the following:

1. There is a set of *nodes* responsible for running a consensus protocol, and a set of *clients* who may submit "transactions" to one or more of the nodes.
2. Each node maintains a local file that we will call its *history*.[2]
3. Nodes can send messages to other nodes, and receive messages from other nodes.

Some remarks are in order. We differentiate here between nodes and clients. The nodes are responsible for maintaining copies of the file. The clients may suggest modifications to the file by sending messages (instructions) to the nodes. For simplicity we assume here that modifications consist of adding information to the file (so no deletions). In a cryptocurrency application this is a reasonable assumption: the file – or history – represents the history of all transactions, which can only grow over time.

Informally, the SMR problem asks to keep all the nodes in sync. Meaning that the local histories of all nodes are the same.

More formally, the SMR problem is to design a protocol that is to be executed by each of the nodes. (Think of a piece of code.) The protocol is allowed to do the following operations:

- maintain or change the local state of the node,
- receive messages from other nodes and from clients,
- send messages to other nodes.

We will see examples of protocol shortly. First, we need to define the properties that we want the protocol to have. In other words, how do we formalize "maintaining a file"? Here we can distinguish two key properties. The first is a *safety* guarantee: we want all nodes to agree on the same file. The second is a *liveness* guarantee: we want to be able to modify the file.

Goal 1: Consistency. We say that a protocol satisfies consistency if all the nodes running it always agree on the history. That means that the local history of all the nodes is equal.

In particular, in the case where the local history is supposed to be a list of transactions, all nodes would agree on the order of the transactions.

[2] It represents for example an ordered list of transactions that only grows over time.

Goal 2: Liveness. Every "transaction" submitted by a client to at least one node is eventually added to every node's local history.

For the moment, we view "transactions" as simply adding an entry in the file. That is, we ignore the very important financial question of whether the transaction is "valid" – agreeing with the current balance in each of the accounts.

The two goals together are non-trivial to satisfy. It is however not so hard design protocols that reach exactly one of the two goals.[3]

3.2.1 Protocol achieving consistency

Here is our first protocol. It will achieve consistency, but not liveness. We will describe the protocol by giving its *pseudocode*. That is, we describe the protocol mostly in words, without committing to a specific programming language.

```
Alg_Consistency:
1.  Initialize: local history H = [ ].
2.  Upon receiving a message m from a client do:
        Nothing.
3.  Upon receiving a message m from a node do:
        Nothing.
```

As you can see, the protocol consists of three lines. The first line describes the initialization that the node performs. In this case, it defines its local history H to be the empty list $H = [\,]$. The second line describes the behavior of the node when it receives a message from a client. The third line does the same for when it receives a message from a node. Here the last two actions are rather trivial: do nothing.

Although this is an extremely naive protocol, we can show that it does satisfy the first goal: consistency.

Lemma 3.1. *If all nodes in a distributed network run* `Alg_Consistency`*, then it satisfies consistency.*

Proof. Assume all nodes in a distributed network run `Alg_Consistency`. To argue that we satisfy consistency, we observe that for each node and every moment in time the local history state equals the empty list $[\,]$. In particular, this means that all nodes agree on the same local history, at all times. □

You are recommended to think about the following exercise before proceeding to the next section.

> **Exercise**
>
> Convince yourself that a distributed network whose nodes all run `Alg_Consistency` does not guarantee liveness.

3.2.2 Protocol achieving liveness

As a second example, we will give a protocol that achieves liveness, but not necessarily consistence.

[3] Warning: these protocols might seem a bit silly, they are meant as an easy introduction to thinking about protocols.

```
Alg_Liveness:
1.  Initialize: local history H = [ ].
2.  Upon receiving a message m from a client do:
        Append message m to H.
3.  Upon receiving a message m from a node do:
        Append message m to H.
4.  At midnight do:
        Send local history H to all other nodes.
```

A couple of remarks are in order. First, the protocol now actually "does something"! In particular, a node will act upon a message that it receives from either a client or another node. Second, we see that nodes are no longer passive: they occasionally send messages to other nodes as well. Third, the protocol now – implicitly – assumes that a node is aware of the concept of time: it needs to perform a specified action every day at midnight. This is in stark contrast to the protocol `Alg_Consistency` which was purely *event-driven*: a node only had to take action when a message arrived. ("Took action" is maybe a slight exaggeration: the protocol doesn't do anything when a message arrives.) For the purpose of this example, we will assume that all nodes agree on the current time. When an event happens once per day, this is a relatively mild assumption. It also means that messages from clients only get shared once per day, which might not be sufficiently quick depending on the application. At the other extreme, we could imagine the nodes want to share incoming messages every millisecond. In that case however, you might run into all kinds of issues: nodes might be too far apart for a message to pass from node A to node B within that time frame, or nodes might disagree on the current time (we are now measuring milliseconds after all). In that case, agreeing on the time is a much stronger assumption. We will revisit these – and other – assumptions in more detail later on. For now, let us prove that `Alg_Liveness` guarantees the liveness property.

Lemma 3.2. *If all nodes in a distributed network run `Alg_Liveness`, then it satisfies liveness.*

Proof. Assume all nodes in a distributed network run `Alg_Consistency`. To argue that we satisfy liveness, we need to show that if a client sends a message m to one or more nodes in the network, then it *eventually* gets added to the local history of every node. To that end, assume client i sends a message m to a group of nodes that includes node A, on day 1. On day 1 node A receives message m and adds it to its local history H_A. Now consider an arbitrary node B in this network that is distinct from A. (It might have received message m from client i on day 1 as well, in which case it added m to H_B on day 1 and there is nothing left to show.) At the end of day 1, node A sends their local history H_A to all other nodes. Therefore, on day 2, node B receives H_A and appends it to H_B. Since H_A included the message m, this means that H_B now contains the message m as well. Thus, the message m is eventually added to the local history of every node.[4] □

It is important to note however that `Alg_Consistency` does not guarantee the consistency property. Can you see why?

> **Exercise**
>
> Consider a distributed network containing two nodes A and B where on day 1 client i sends message m_A to A and j sends message m_B to B. Describe the local history of each of the nodes on days 1 and 2. What do you observe?

[4] In this case, *eventually* means at most one day after the client sends the message to at least one node in the network.

3.3 (Strong) assumptions about the decentralized setting

We will be working with a mathematical model of a real-world situation. We are therefore making some assumptions. In this section we list one possible set of assumptions. Some of the assumptions that we are making here are more restrictive than others. We will call such an assumption relatively *strong*. In later chapters we will replace these strong assumptions with weaker assumptions. While reading the next three assumptions, try to answer the following questions: is it a weak or strong assumption? Do I know an application where it holds / does not hold?

3.3.1 Assumption 1: the set of nodes is known

This assumption is also referred to as the *permissioned* setting. It assumes the set of nodes in the network is fixed and known to all nodes, moreover it assumes that each node has a unique identifier that is also known to all other nodes. We will frequently use $n \in \mathbb{N}$ to denote the number of nodes. This allows us to use the numbers between 1 and n as unique identifiers.

Key advantages:

- It allows *majority voting*.
- One can order the nodes based on their identifier.

3.3.2 Assumption 2: signatures exist and cannot be forged

This assumption can be viewed as an extension of Assumption 1. We assume that nodes can add their signature to a message. By this we mean that if node A sends a message m (to an arbitrary node), then they can add their signature to it. All other nodes have a verification procedure that can correctly determine whether node A signed message m. No other node can *forge* A's signature: no other node can add a 'signature' that the verification procedure would accept as A's signature. This is an example of a *trusted setup*. This assumption is also referred to as assuming Public Key Infrastructure (PKI).

We will not go into further details about this assumption in this course; we will (happily) assume that it holds and not go into details of how one would implement it in a real-world scenario. (It is a relatively mild assumption however.)

Key advantage:

- It allows us to trust who sent which message.

3.3.3 Assumption 3: synchronous model

This is an assumption about the (reliability of the) communication network. Formally, we require the following two sub-assumptions:

1. All nodes have access to a shared clock.
2. Bounded message delays: messages arrive within a predetermined amount of time.

Together, these two assumptions allow us to divide time into smaller intervals in such a way that that messages sent at the start of an interval arrive before the end of the interval. To avoid having to specify the bounded message delay, we will simply number the intervals. Concretely, we thus assume that messages sent at the start of (or simply *in*) interval t arrive at their intended recipient before the start of interval $t + 1$. We will often refer to the intervals as *rounds*.

Key advantage:

- It allows us to define *rounds* (see above).

3.3.4 Discussion of the assumptions

Assumption 1 is realistic in some scenarios: if we are using multiple computers to create a backup of a file (e.g. a list of grades), then it reasonable to assume that we know how many computers we are going to use. (There is a central entity that determines the number of nodes.) For our second motivating example however, blockchains, this is a very unrealistic assumption! Not knowing the set of nodes participating in the blockchain protocol is in fact a key feature that we are aiming for. We would like a blockchain protocol to be able to function in a completely decentralized manner, with nodes being able to enter (or leave) the network while the protocol is active.

Assumption 2, as mentioned above, is one that we will simply assume throughout the course.

Assumption 3 is again realistic in some scenarios, but not in others. We have seen an example of a protocol that worked in the synchronous model in Section 3.2.2: the `Alg_Consistency` protocol. The synchronous model makes optimistic assumptions and therefore serves as a good sanity check when designing protocols: the protocol should at least function correctly in the synchronous model. In Chapter 4 we will work with the synchronous model. In the later chapters Chapter 5 and Chapter 6 we will encounter protocols that make milder assumptions.

3.4 Honest nodes

The final assumption is about whether we assume nodes to be *'honest'* or *'dishonest'*. We say that a node is *honest* if it executes the intended protocol. Any node that deviates from the intended protocol is called *dishonest* or *faulty*. Note that honesty in this context is a description of the nodes behavior, not its intentions. In the example of creating backups of a list of grades, it is for example perfectly reasonable to assume that all nodes have good intentions, but we would like a protocol to 'work well' even if one of the nodes breaks and is therefore unable to follow the protocol. We therefore prefer the term *faulty* for nodes that are not honest.

The assumption that we will make in this section (and only this section) is a very strong one:

Assumption 4: All nodes are honest.

Naturally, we would like to relax this assumption as soon as possible. In the next chapter we will indeed replace it with a much milder assumption: there we assume a bound f on the number of faulty nodes.

3.4.1 Solving the SMR problem under assumptions 1,2,3,4

Here we show how to solve the SMR problem under the assumptions 1, 2, 3, and 4. That is, we work in the permissioned, synchronous model, we assume PKI and that all nodes are honest.

As a reminder, we want to design a protocol that guarantees consistency and liveness for the SMR problem. We have already seen two protocols that achieve either consistency or liveness. In particular, in Lemma 3.2 we have shown that `Alg_Liveness` guarantees the liveness property. In the subsequent discussion we have seen that this protocol does not guarantee consistency: it can happen that two nodes disagree on the order of transactions in their local history. The 'issue' here was that every node had the 'right' to append transactions to the local history of other nodes, which could lead to disagreements on the order that transactions are written down. The protocol that we describe here resolves this issue by selecting a leader in each round, who is the only one with the permission to write in that round.

Coordinating via rotating leaders: since we are in the permissioned setting, we know the number of nodes participating in the network, say n. We can there do the following:

- in round 1, node 1 is called the leader and all other nodes are called followers,
- in round 2, node 2 is called the leader and all other nodes are called followers, …
- in round n, node n is called the leader and all other nodes are called followers After n rounds, we reset the clock and start again as in round 1. In other words, we are rotating the leaders.

We now describe the protocol for the leader and follower nodes separately. For ease of notation, we always assume that nodes initialize their local history to $H = [\,]$ at the start of the protocol. We additionally allow the nodes to use some local workspace W, which they can use to store information (temporarily); it is not part of the local history state.

```
Alg_Leader(t):
1.  Upon receiving a message m from a client do:
        Append message m to the local workspace W.
2.  At the end of the round do:
        Remove from W the messages that are already part of H.
        Append W to H and send W to all other nodes.
        Reset W = [ ].
```

```
Alg_Follower(t):
1.  Upon receiving a message m from a client do:
        Append message m to the local workspace W.
2.  Upon receiving a message m from a node do:
        If m is signed by the leader of the current round t do:
                Append m to H.
        Else do:
                Nothing.
```

Our claim is that if the nodes in a distributed network adhere to the above protocol, then both liveness and consistency are guaranteed.

Lemma 3.3. *If all nodes in a distributed network run the 'coordinating via rotating leaders' protocol, then it satisfies liveness and consistency.*

> Exercise
>
> Prove Lemma 3.3.

Chapter 4

The Dolev-Strong protocol

In the previous chapter we have seen a protocol for the SMR problem under the assumptions 1,2,3, and 4. Here we replace the last assumption by a more realistic one: we no longer assume all nodes are honest. Instead, we assume there are at most f faulty nodes in the network, where f is some number between 0 and n.

Our goal in this chapter is to solve the SMR problem under the assumptions 1,2,3 and assuming a bound f on the number of faulty nodes (for some values $f > 0$). The protocol that we will study is due to Dolev and Strong (1983). At a high level, it works similarly to the rotating leaders protocol that we have seen in Section 3.4. If there is even a single faulty node, the rotating leaders protocol no longer satisfies consistency. Can you see why?[1] Informally, the Dolev-Strong protocol gives us a way to detect faulty leaders that try to 'trick' the other nodes into inconsistency. To formalize this, we introduce the *Byzantine Broadcast problem* in Section 4.2. We show how the SMR problem *reduces* to the Byzantine broadcast problem, see Section 4.3. We finally discuss the Dolev-Strong protocol and show that it solves the Byzantine Broadcast problem.

4.1 Bounded number of faulty nodes

We recall from the previous chapter that a node is called *honest* if it never deviates from the intended protocol. Any node that is not honest is called *faulty*. Assumption 4 from the previous chapter was that all nodes are honest, or, equivalently, that the number of faulty nodes is equal to 0. Here we replace this with a more realistic assumption:

Assumption 4': the number of faulty nodes in the network is at most f.

We in fact assume the protocol knows the upper bound f on the number of faulty nodes. This means the protocol may depend on f. In the previous chapter we have seen a protocol that assumed $f = 0$. Interesting values of f to keep in mind are $f = n/3$ or $f = n/2$, meaning that at most a certain fraction of the nodes is faulty. To start building some intuition, we will consider $f = 1$ and $f = 2$ later on in this chapter.

4.2 The Byzantine Broadcast problem

In the *Byzantine Broadcast* problem we consider the following setting:

[1] Suppose there is one faulty node. In some round, it will be elected as the leader. It can then simply choose to violate consistency by sending different messages to different nodes.

1. There are n nodes, one is designated *sender* and the other $n-1$ nodes are *non-sender*. The identity of the sender is known to all non-senders.
2. The sender has a *private input* v^* that belongs to some set V. (Private means that only the sender knows v^* at the start of the protocol.)

The set V here represents the set of possible private inputs. In a cryptocurrency application, this might be the set of all valid transactions. For intuition, it suffices to think of only two possible inputs: $V = \{0, 1\}$.

Informally, the goal in the Byzantine broadcast problem is for the sender to send v^* to all other nodes in such a way that all other nodes can be certain that everyone received the same message. Formally, we say that a protocol is a solution to the Byzantine broadcast problem if it guarantees the following three properties:

1. **Termination**: Every honest node i eventually halts with some output $v_i \in V$.
2. **Agreement**: All honest nodes halt with the same output.
3. **Validity**: If the sender is an honest node, then the common output of the honest nodes is the private input v^* of the sender.

Some remarks are in order. First, the requirements are different for honest nodes and faulty nodes. This is by necessity: faulty nodes can deviate in *any* way from the protocol, so we cannot hope to guarantee anything about their output. Second, the condition *agreement* is required to hold both if the sender is honest and if it is faulty. Agreement is comparable to the *consistency* property that we have seen in the SMR problem. Third, note that the *validity* property, necessarily so, is conditioned on the sender being honest. Therefore *validity* is trivially satisfied when the sender is faulty.

Exercise

Design a protocol, for any $0 \leq f \leq n$ specifying both the behavior of the sender and non-sender nodes, that guarantees *termination* and *agreement* whenever assumptions 1,2 and 3 hold.

Solution

Fix some $v_0 \in V$ (e.g. if V is a set of numbers, pick the smallest). Consider the simple protocol in which every node (both sender and non-sender) terminates in round 1 by outputting v_0. This protocol clearly satisfies *termination* (every node halts in round 1) and *agreement* (every honest node halts with the same output v_0).

4.3 SMR reduces to Byzantine Broadcast

Here we show that any protocol that solves the Byzantine Broadcast problem can be used as a black box to solve the State Machine Replication problem. In our original definition of the SMR problem, in particular for liveness and consistency, we assumed all nodes were honest. In the presence of faulty nodes we need modify the guarantees slightly:

Goal 1: Consistency. We say that a protocol satisfies consistency if all the *honest* nodes running it always agree on the history.

Goal 2: Liveness. Every "transaction" submitted by a client to at least one *honest* node is eventually added to every node's local history.

(The only difference is adding the word *honest* in the right places.)

We now present a protocol that solves SMR, given a protocol for the Byzantine Broadcast problem. Concretely, let us assume we are in the synchronous and permissioned setting (assumptions 1,2,3) and that there are at most f faulty nodes (assumption 4'). Moreover, assume we are given a protocol π that solves the Byzantine Broadcast problem under those assumptions. Assume π always terminates in at most T rounds. As before, we allow the nodes to use some local workspace W that they can use to store information, but which is not part of the local history state (this is used to implement step 2).

```
Alg_SMR_from_BB(pi,f):

At each round t=0, T, 2T,... that is a multiple of T do:
1.  Define the current leader to be node t/T modulo n.
2.  The leader constructs a list L of transactions it has
    received in the past T rounds, which are not yet part of H.
3.  Use the protocol pi with
    as leader node t/T modulo n and private input L.
4.  At round t+T-1, every node i appends
    its output L_i in the Byzantine Broadcast problem to its local history.
```

Lemma 4.1. *Under assumptions 1,2,3,4', assuming π solves the Byzantine broadcast problem in at most T rounds, the protocol* `Alg_SMR_from_BB` *solves the SMR problem.*

Proof. We need to argue that consistency and liveness are satisfied. For consistency, we can argue in an inductive manner. At the start of the protocol, all honest nodes initialize their local history to $H = [\,]$. Assume all honest nodes agree on the local history at some time that is a multiple of T. By the guarantees of π, when π terminates, which happens within T rounds, every honest node agrees on a common output L. Therefore all honest nodes append the same message L to their local history state H in round $t + T - 1$, which ensures that the local history states of all honest nodes agree between rounds t and $t + T$.

For liveness, it suffices to observe that every honest node is elected as a leader once very nT rounds. □

4.4 The cases $f = 1$ and $f = 2$

In this section we introduce the idea of "cross-checking". We show that this solves the Byzantine broadcast problem when there is at most 1 faulty node, and that it fails when there are 2 faulty nodes. This allows us to build op some intuition about the main idea underlying the Dolev-Strong protocol, without the technicalities that arise when there are multiple faulty nodes. Technically, the Dolev-Strong protocol that we present in the next section is independent from this section, which means that this section is "optional" and can be skipped (at your own risk).

Intuitively, the protocol works by asking the honest nodes to do one simple step of cross-checking: each node verifies whether all other nodes received the same messages from the sender.

Formally, we consider the following protocol.

```
Alg_Cross_Check:

The protocol consists of three rounds:
1.  The sender sends its private value v* to all non-senders (signed).
```

```
2.  Every non-sender i sends the message m_i it
    received from the sender in round 1 to
    all other non-senders with their signature added.
3.  All non-senders choose the most frequently
    received value among the values it received in rounds 1 and 2.
    (Breaking ties in some consistent way.)
    The sender outputs v*.
```

In the following two exercises you are asked to show that `Alg_Cross_Check` solves the Byzantine broadcast problem when there is at most $f = 1$ faulty node and there are at least 4 nodes in total, but that it breaks when $f = 2$.

> **Exercise**
>
> Under assumptions 1,2,3, $f = 1$, and $n \geq 4$, the protocol `Alg_Cross_Check` solves the Byzantine broadcast problem.

> **Exercise***
>
> Under assumptions 1,2,3, $f = 2$, and $n \geq 4$ even, the protocol `Alg_Cross_Check` does not solve the Byzantine broadcast problem.

Showing that the protocol breaks when $f = 2$ is a bit tricky (hence the * next to the exercise). You are recommended to *try* to solve the exercise on your own, but don't worry if you don't succeed.

A key takeaway from the above two exercises is that one round of cross-checking allows us to deal with one Byzantine node, but not with 2. It thus seems that *more cross-checking* is necessary when there are multiple Byzantine nodes. In a nutshell this is precisely what the Dolev-Strong protocol does: every additional round of cross-checking allows for one more Byzantine node.

4.5 The Dolev-Strong protocol

Here we describe a classic protocol due to Dolev and Strong (1983). It solves the Byzantine broadcast problem in the permissioned and synchronous setting, assuming an upper bound f on the number of faulty nodes. Together with the reduction from Section 4.3, this solves the SMR problem under the same assumptions. To describe the protocol, we need one more definition, that of *convincing messages*.

4.5.1 Convincing messages

A node i is *convinced of value v in round t* if it receives a message prior to round t that satisfies the following three conditions:

1. It contains the value v;
2. It is first signed by the sender;
3. It is also signed by at least $t - 1$ other, distinct nodes, none of which are i.

4.5.2 Protocol description

```
The Dolev-Strong protocol(f):

1.  In round 0 the sender:
    Sends its private value v* to all the non-senders
    Outputs v*
2.  In round t = 1,...,f+1 a non-sender i does:
        If i is convinced of a value v by some message m received
        prior to round t and has not been convinced of v before:
            i adds its signature to m and sends it to all non-senders
3.  At the end of round f+1 a non-sender i does:
        If i is convinced of exactly one value v:
            Output v
        Else:
            Output "failure" (some message not in V)
```

In the above protocol we are outputting "failure" to signal that we have detected a Byzantine sender. The precise message "failure" is of course arbitrary, what matters is that it cannot be confused with a valid private input of the sender. The message "failure" is assumed to be distinguishable from inputs from V.

Intuitively, in the Dolev-Strong protocol the $f+1$ rounds after the first correspond to $f+1$ rounds of cross-checking. Note that for $f = 1$ the algorithm is not equal to `Alg_Cross_Check`. As we will see in the next section, the list of *distinct* signatures is used to detect Byzantine senders. For now, remark that a node that is newly convinced of a message at the end of round $f+1$ observes precisely $f+1$ distinct signatures, at least one more than the number of Byzantine nodes.

4.5.3 Proof of correctness

We need to show that under the assumptions 1,2,3 the Dolev-Strong protocol satisfies *termination*, *validity*, and *agreement*. The property *termination* is clear from the description of the protocol. We prove the remaining two properties separately.

Lemma 4.2. *Under the assumptions 1,2,3, assuming at most f faulty nodes, the Dolev-Strong protocol satisfies* validity.

Proof. Assume that the sender is honest (why are we allowed to do so?) and has private value v^*. The sender thus follows the protocol and sends a signed copy of v^* to all non-senders in the first round, it then outputs v^* and terminates. In round 1 all non-senders are therefore convinced of the value v^*. Indeed, they have received a message that 1) contains the value v^*, 2) is first signed by the sender, and 3) is signed by $1-1=0$ other, distinct nodes, none of which are i.

It remains to observe that since the sender is honest, no node is ever convinced of a value other than v^*: a *convincing message* needs to contain the signature of the sender. (Here we are using our assumption that signatures cannot be forged.)

At the end of the protocol, all non-senders therefore output v^* as well, establishing *validity*. □

The hard(er) part is to show that the protocol satisfies *agreement*, and this is where we need the multiple rounds.

Lemma 4.3. *Under the assumptions 1,2,3, assuming at most f faulty nodes, the Dolev-Strong protocol satisfies* agreement.

Proof. Assume that the sender is Byzantine (why are we allowed to do so?). We will show that at the end of the protocol, all honest nodes are convinced of exactly the same set of values. This suffices to establish agreement (can you see why?[2]).

Suppose on an honest node i gets newly convinced of a value v by a message received before the end of time step t for some $t \in \{0, 1, 2, \dots, f+1\}$. We show that all other honest nodes also get convinced of value v before the end of the protocol. Now we need to distinguish two cases: i) $t \leq f$ or ii) $t = f + 1$.

Case i): if $t \leq f$, then node i still has time to communicate to other honest nodes. In particular, in round $t + 1$ it will add its signature to the message that convinced them of value v and it will send that to all other non-senders. Since node i was convinced of v in round t, this newly signed message is a *convincing message* for all other nodes that had not yet been convinced of v (check this!).

Case ii): if $t = f + 1$, then node i no longer has time to communicate. We thus need to show that all other honest nodes had already been convinced of the value v prior to the end of round t. To do so, we will crucially use that node i is convinced *for the first time* of value v at the end of round $f + 1$. This means that at the end of round $f + 1$ it receives a message that is signed first by the (Byzantine) sender, and also by f distinct other nodes ($f = t - 1$). Crucially, since the sender is Byzantine, at least one of these f nodes is an *honest* node. Let j be such an honest node. Since i received a message containing the value v and j's signature, the honest node j was newly convinced of the value v in some round $t' \leq f$.[3] We can thus apply case i) to the *honest* node j, showing that all non-sender nodes have received a convincing message containing the value v. □

[2] Either every honest node is convinced of exactly one value v, in which case all honest nodes output v. Alternatively, every honest node is convinced of at least two distinct values, in which case all honest nodes output "failure".

[3] In fact, $t' = f$ is the only possibility (since node i is only convinced in round $f + 1$).

Chapter 5

Longest-chain protocols

In this chapter we study a second type of consensus protocols, those based on blockchains and in particular the concept of longest chains. This type of consensus protocol lies at the heart of several cryptocurrencies (e.g. Bitcoin). The Bitcoin protocol is one example of a longest-chain protocol, but it is not the only one. In fact, most cryptocurrencies are based on blockchain protocols, see for example the long list on Wikipedia here. All of these protocols follow the same general recipe, but they use different ingredients in certain steps. In this chapter we focus on the general recipe, in the next we will see some of the ingredients that Bitcoin uses (namely proof of work).

To simplify the exposition, we will work in the permissioned setting, assuming PKI, and especially the synchronous model of communication. If we restrict to that setting however, then we already know a good consensus protocol! Indeed, this is precisely what we achieved in the previous chapter. So what is the advantage of doing it again? There are several answers to this question. First, the consensus protocol based on Dolev-Strong is rather slow. It requires many rounds of cross-checking before new information gets added to the local history of each of the nodes. A second answer is that the Dolev-Strong protocol relies on the *permissioned* setting, whereas the protocol that we design in this chapter extends very naturally to the *permissionless* setting. We will go into this in more detail at the end of this chapter.

We start this chapter with a high-level description of a blockchain protocol and the behavior of honest nodes. As usual, we then explicitly state the assumptions that we make in this chapter. We will see that the notion of consensus trivially holds (under the assumptions that we make). Instead, we introduce a new notion – *finality* – that is harder to achieve, but very useful in a cryptocurrency application! In a nutshell, a block is final when it is 'deep enough in the chain'. We will formalize this later on in the chapter, for now the picture to have in mind is that a block that is final can no longer be changed (surprise); if we imagine the block to contain a transaction, we can thus trust that this transaction actually took place. We spend the second half of this chapter to prove that longest-chain protocols achieve finality when, roughly speaking, the majority of the nodes is honest.

5.1 Protocol description

The starting point for a blockchain protocol is the concept of a blockchain. As the name suggests, a blockchain is a set of blocks that are connected to form a *chain*. What do we mean by a chain? We say that a set of blocks forms a chain if:

- There is one "genesis" block, forming the start of the chain, we typically denote it with B_0,

- Every block that is not B_0 points to exactly one other block as its predecessor, in such a way that if we follow the chain of predecessors, we arrive at the genesis block B_0.

The above is a somewhat convoluted way of saying that a blockchain looks like Figure 9.1. It is a directed graph where each node has out-degree 1,[1] except for a single out-degree 0 block B_0. It moreover has the property that every block is connected to B_0 by a directed path. As a graph, a blockchain is thus a directed tree whose root is the block B_0. If you are not familiar with directed graphs, but you are familiar with undirected graphs, then it suffices to think of a blockchain simply as a tree whose root is the block B_0. The direction of the arcs is the one that follows the path towards B_0.

A *blockchain* is a directed graph G that is an *in-tree* whose root corresponds to B_0.[2]

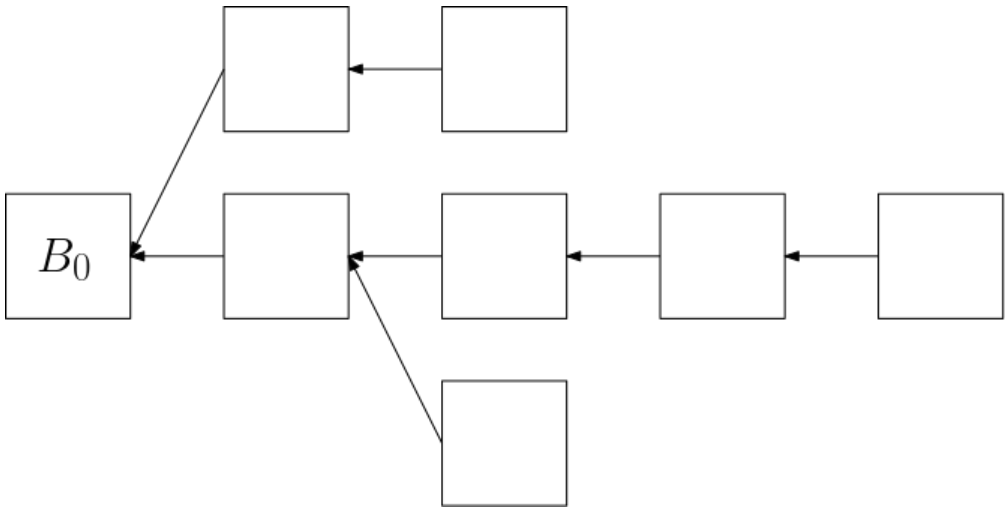


Figure 5.1: A chain of blocks

The purpose of a blockchain is again to contain information. By this we mean that every block represents a chunk of information (say a 1MB file). By storing the information in a chain, we construct a notion of time: the information in a block is created after that of its predecessor. Needless to say, this is a useful property to have when you want to keep track of a list of transactions.

So how does a blockchain fit in the distributed computing framework? We want to think of the blockchain as the information that all nodes have access to. This describes the ideal scenario. If we assume it, then consistency becomes trivial. Such an assumption however essentially amounts to instantaneous communication. We will formalize the assumptions that we work with in the next section.

Now that we have cleared up the concept of a blockchain, we sketch the protocol that we have in mind.

```
A blockchain protocol:

1.  Initialize with a hard-coded genesis block B0
2.  In each round r = 1,2,3,.... do:
    a)  Choose one node i as the leader of round r.
```

[1] Recall that the *out-degree* of a node in a directed graph is the number of arcs leaving the node. (In the picture, the number of arrows coming out of a block.)

[2] Here an in-tree is a directed graph that is defined as follows. If we treat all arcs as undirected edges, then the graph is a simple tree. If we treat B_0 as the root node of this tree, then all arcs are directed in such a way that they point towards B_0.

```
    b)  Node i proposes a set of blocks,
        each specifying a single predecessor block.
```

This protocol is under-specified: we will fill in the details of step 2 later. In particular, one can imagine several different ways of choosing a leader in step 2a), for example:

1) In the permissioned setting that we used in the previous chapter, we simply select node i as leader in rounds $i, n+i, 2n+i, \ldots$.
2) In a Proof of Work protocol (which we will discuss in the next chapter), the leader in round r is the first node to provide a proof of work after round $r+1$.
3) Proof of Stake is another way to select leaders.

For the moment, either one of these three options is good to have in mind. In the second half of the chapter we prove correctness of the longest-chain protocol under certain assumptions. At that point, you are recommended to revisit the above three options and see whether or not they match the assumptions. We do want to point out that the second option already hints at the fact that *rounds* don't have to correspond to time slots, they rather refer to the periods in between certain events. We will come back to this later.

5.1.1 Honest vs. dishonest behavior

Let us now describe the intended behavior of *honest* nodes. In a longest-chain protocol an honest node will do the following when it is elected as the leader of a round r:

- it proposes exactly one new block,
- this new block points to exactly one predecessor,
- the predecessor was created in a previous round,

Naturally, the new block may contain some information. For example, a list of newly made transactions or a copy of your favorite recipe for pasta. However, for this chapter we will ignore such information: that information is relevant for applications, but not for the guarantees that we want to achieve here. All that matters for us is that a block contains a pointer to precisely one predecessor that was created in a previous round.

In a longest-chain protocol, there is one additional requirement on the behavior of *honest* nodes:

- the new block extends a *longest chain*.

We should of course define what we mean by a longest chain. A longest chain in a blockchain refers to a sequence of blocks that are on a longest path in the blockchain. Let us revisit the example from Figure 9.1. We will label the blocks for ease of reference; the particular labels that we use are not important. In a blockchain we typically assume that a block is signed by its creator and this signature can be used as a label (it includes sufficient identifying information such as the name of creator, time of creation, predecessor,…).

In this example, we see three distinct (maximal) paths.[3] There is the path $B_0 \leftarrow B_1 \leftarrow B_4$ which contains three blocks. The path $B_0 \leftarrow B_2 \leftarrow B_3 \leftarrow B_6 \leftarrow B_7$ contains five blocks. Finally, there is also the path $B_0 \leftarrow B_2 \leftarrow B_5$ which contains three blocks. In this example, the path containing 5 blocks is the longest. This path would thus be referred to as the longest chain. An honest node in this example would thus create a new block, say B_8, that points to B_7 as its predecessor.

In the above example, there is a unique longest path, which means the *honest* node does not have to make a choice: its new block has to point to B_7. This does not have to be the case however. In the example

[3] A path is *maximal* if it cannot be extended to a longer path, i.e., it is not a subset of a larger path.

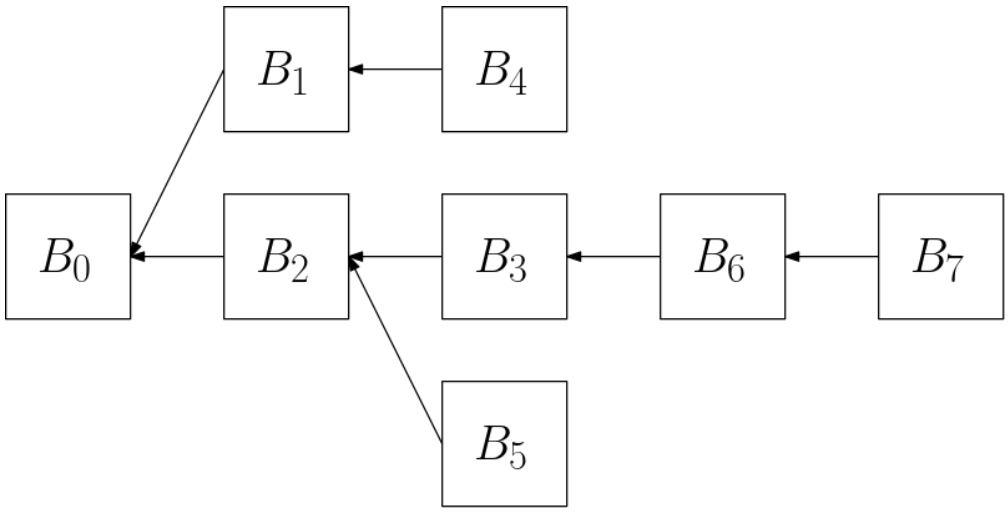


Figure 5.2: A labelled blockchain

below, there are two maximal paths, both containing exactly three blocks (namely $B_0 \leftarrow B_1 \leftarrow B_4$ and $B_0 \leftarrow B_2 \leftarrow B_3$). Either of the two paths would be a longest chain in this example. This is the reason that we ask honest nodes to extend *a* longest chain and not *the* longest chain. When there are several longest chains, honest nodes may break ties in an arbitrary (but fixed) way.

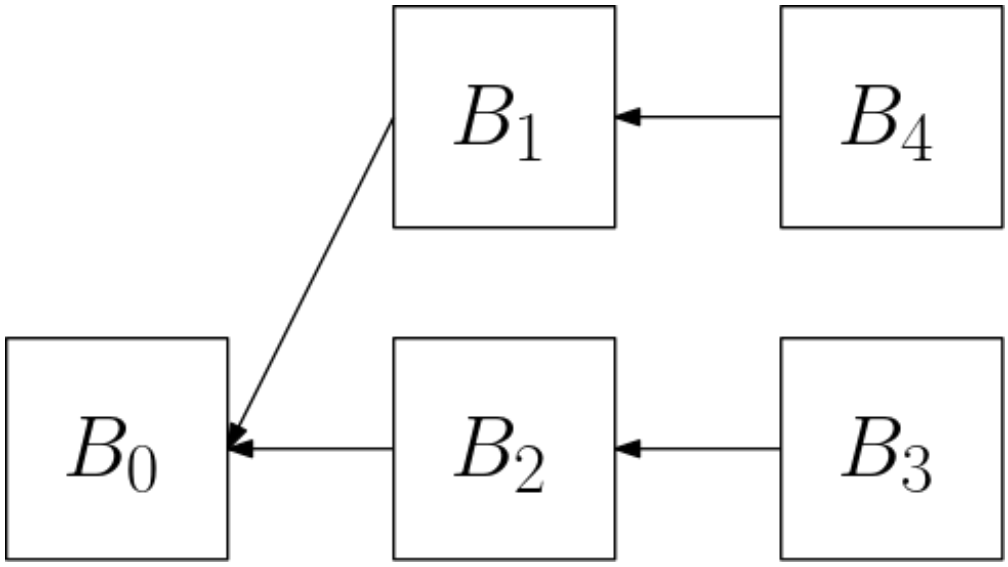


Figure 5.3: A labelled blockchain with 2 longest chains

We have now described the intended behavior of honest nodes. What about nodes that are *dishonest* or *faulty*? As usual, we make no assumptions whatsoever about their behavior. After all, a node is called *faulty* whenever it deviates from the intended protocol (no matter the reason or the manner in which deviates). There is however something we can say. Since the intended behavior is that a new block specifies *exactly* one predecessor, the honest nodes can always disregard any blocks that specify 0 or at least 2 predecessors. We can therefore assume that a block created by a *faulty* node also specifies exactly one predecessor which moreover comes from a previous round.

Looking ahead, let's think of a blockchain as storing a list of transactions. In that case, the honest nodes are working together to maintain a correct history of all the transactions that have taken place. They do so by recording transactions on the longest chain and they regard the longest chain as the one containing the *true* list of transactions.

> Exercise
>
> Assume all nodes are honest. What does the blockchain look like after 10 rounds?

If we think of *faulty* nodes as malicious, then their goal could be for example to double-spend some of their money. They could do so by adding (many) blocks to a chain that is currently not the longest. Eventually this would create a new longest chain. If we recall that honest nodes view the longest chain as the one containing the true transactions, then this would allow a faulty node to spend their money twice. Indeed, if we assume they spent some money in the original longest chain in some block B_i, then they could extend the path ending at the predecessor of B_i and by doing so, they could spend their money a second time (buying something else).

Figure 5.3 is (or could be) an example of such a situation. Here you can imagine that the honest nodes created blocks B_2 and B_3 in rounds 1 and 2 (recall that the labels are arbitrary). In rounds 3 and 4 the leaders happened to be *faulty* and they collaborated to start a new path with the blocks B_1 and B_4. At this point, the *faulty* nodes have successfully confused the *honest* nodes: there is no way to tell which of the two chains was constructed by *honest* nodes. It might thus be the case that an *honest* node that is elected as leader in round 5 decides to indicate B_4 as its predecessor. By doing so, the top chain would become the unique longest chain. At this point even honest nodes would start extending the top chain. This means the faulty nodes have successfully convinced the honest nodes to abandon their original chain, thus reverting some previously made transactions (blocks).

5.2 The assumptions

We now formalize the assumptions that we make about blockchains in a distributed network.

5.2.1 Assumption A1: the genesis block is unknown prior to starting the protocol

This first assumption is a trusted setup assumption.

> **A1)** We assume that no node has knowledge of the genesis block prior to the deployment of the protocol.

At this point, it should not be clear why we need this assumption. But if you are already familiar with proof-of-work based protocols, try to answer the following question. (If you are not yet familiar, please revisit this question after we have studied the next chapter.)

> Exercise
>
> What could go wrong in a proof of work setting if we don't make this assumption?

Solution

The faulty nodes could cheat by creating valid blocks before the deployment of the protocol! If they manage to create K blocks before the start of the protocol, this gives them the power to create a path of length K "for free". This means we cannot trust the first K blocks on the blockchain. Another way to phrase assumption 1 is thus that we assume that $K = 0$.

So how do we verify this assumption? Technically we can't, we just have to take it on faith. There are ways to make the assumption more plausible however. For example, the first block of Bitcoin was created on 3 January 2009. It contained the text "The Times 03/Jan/2009 Chancellor on brink of second bailout for banks", which is a reference to a headline of that day's issue of the newspaper The Times. Assuming Nakamoto had no way to influence this headline, it is thus reasonable to assume that nobody knew the genesis block (long) before the deployment of the Bitcoin protocol.

5.2.2 Assumptions about leader selection

We need to make two assumptions about how leaders are selected. The first is related to the PKI assumption that we have seen in the BFT protocol.

> **A2)** All nodes can efficiently verify whether a given node is the leader of a given round.
>
> **A3)** No node can influence the probability with which it is selected as the leader of a round in step 2a.

In the protocols that we have studied in the previous chapter both were trivially satisfied. Indeed, we were working in the permissioned setting and the synchronous model and the leader-selection protocol simply asserted that in round t node t would be the leader (counting rounds modulo n, i.e., node 1 is the leader in round $n + 1$ as well and so on). Since we assumed signatures exist and cannot be forged, only node t could pretend to be the leader in round t. This shows that A2 was satisfied. Since the leader-selection protocol is deterministic, A3 is satisfied trivially. In this chapter, and the next, we want to move away from the permissioned, synchronous setting. Assumptions A2 and A3 clarify the conditions that our leader-selection protocol should satisfy.

In the next chapter we will argue that these assumptions also hold in the proof of work setting.

5.2.3 Assumptions about block production

We assume the following about blocks produced in round r:

> **A4)** Every block produced by the leader in round r must claim as its predecessor a block that belongs to a previous round.

Remember that we assume signatures exist. We can thus assume that when a block is created, the creator includes the round number in the identifier of a block. This assumption implies that if you trace the sequence of predecessors of a single block, you always end up at a/the block that was created in round 0: the genesis block.

Assumption A4) seems a bit redundant at first: after all how can you create a block that points to a predecessor that doesn't exist yet? Honest nodes would of course not do this, but faulty nodes might have incentives to do this. This assumption puts some restrictions on their behavior, which we should thus verify for any blockchain protocol. For example, under the assumption A4), faulty nodes are still allowed to create multiple blocks in a single round, but each of these new blocks has to point to a predecessor from a previous round.

The faulty node can thus not propose blocks that point to each other for example. It also prevents faulty nodes from "delaying": if they are correctly selected as leader in round 10, they might want to wait to see what happens in rounds 11 and 12 before announcing their block for example in round 13. This assumption prevents them from using the blocks created in rounds 11 and 12 as predecessors for their block.

Assumption A4) will be crucial in the analysis, but in implementations it is typically not so hard to enforce. For example, in the setting from the previous chapter we would just require the leader of round r to include the round number in the description of the block.

In the proof of work protocol that we will study in the next chapter, we can in fact enforce a stronger version of A4). This stronger version is not needed for the lemmas and theorems in this chapter, but it sometimes simplifies the proofs.

> **A4')** The leader of round r produces exactly one block, and this block claims as its predecessor a block that belongs to a previous round.

We will revisit this assumption in the next chapter. In a nutshell, the proof of work is valid proof only for a single block (the creator has to commit to a block before starting the work).

5.2.4 Assumptions about communication

The last assumption that we make is one about our communication model.

> **A5)** At all times, all honest nodes know about the exact same set of blocks.

This is a (very) restrictive assumption; it essentially trivializes the consistency problem that we had to deal with in the previous chapter. We will see how to relax this assumption in the next chapter in the setting of proof of work. So why do we make this assumption? For one, it simplifies our lives (and certainly the exposition) a bit. The main reason however is that the key ideas behind longest-chain protocols are already needed even when we add this restrictive assumption. When we relax the assumption in the next chapter we will see that it still "holds in spirit"; it is therefore a reasonable way to think about longest-chain protocols.[4]

5.3 The goals: liveness and *finality*

As in the previous chapter, we will have two goals that we want to achieve in a blockchain protocol. The first will be liveness, as in the previous chapter.

As stated above, assumption A5) trivializes the *consistency* requirement. At least, the way we thought about consistency in the previous chapter. In the previous chapter we thought about consistency as keeping all nodes in sync: their local history states had to be identical at all points in time. This is indeed a key aspect of consistency, but it ignores another very important aspect: consistency over time. By that we mean that there is some notion of consistency between the local history of an honest node at time t and the local history of that same node at some later time $t + t'$. For example, the list of transactions recorded at time t is a prefix[5] of the list of transactions recorded at time $t + t'$. If you take another look at the protocols that we have seen so far, you will realize that they also satisfy this second property. The reason for this is that we only allowed information (transactions) to be added to the local history.

In a blockchain, we think of the (shared) local history state as the information that is stored in the blocks on the longest chain. From our previous discussion, it should be clear that the first aspect of consistency

[4] This is certainly a handwavy statement. We will not make it more precise here.

[5] The first part of…

(consistency across nodes at a given time) is trivial, but the second aspect is not! Indeed, consistency over time is the main goal that we will work towards in this chapter. Concretely, we aim for the following.

> **Goal: Finality** (first version) If an honest node i considers a block B as *finalized* at time t, then this block remains finalized at all times after t.

Some remarks are in order. First, we have not yet formalized the concept of *finalized*. We will do so in the next section. Second, even without knowing what *finalized* means, we can make sense of the goal *finality*: if we consider the set of finalized blocks as our local history state, then we have achieved consistency *over time*. Indeed, finality precisely ensures that whatever is part of the local history at time t will remain part of the local history at all future times $t + t'$. Third, the observant reader might have noticed the addition "(first version)". This strongly suggests that there will be a second version in the future. There will be one indeed. In the later sections we will end up talking about protocols involving randomness. In such protocols, the above version of finality is too much to ask for. We will need to replace the first version of finality with a slightly weaker version that asks the same probability to hold with high probability. (What do we consider "high probability"? That depends again on the application…)

5.4 Finalizing a block

So how do we finalize a block? Recall that we want to think of the longest chain in the blockchain as the one that stores the list of transactions. Intuitively, we would like to argue that if a block B is far enough from the end of this longest chain, then it will (likely) always be a part of a/the longest chain. It would after all require the faulty nodes to extend the chain ending at the predecessor of B to a new chain that is longer than the currently longest chain.

To formalize this intuition, we introduce the parameter k.

> The parameter k corresponds to the number of blocks at the end of the longest chain that are still considered not finalized.

Let G be a directed in-tree rooted at a node B_0. For an integer k we define

$$\mathcal{B}_k(G) := \text{ the longest chain of } G\text{, with the last } k \text{ blocks removed.}$$

For example, in Figure 5.2 we have

$$\mathcal{B}_0(G) = (B_0 \leftarrow B_2 \leftarrow B_3 \leftarrow B_6 \leftarrow B_7),$$

and

$$\mathcal{B}_1(G) = (B_0 \leftarrow B_2 \leftarrow B_3 \leftarrow B_6).$$

Before reading the rest of this chapter, try to answer the following questions by yourself.

> **Exercise**
>
> In the above example Figure 5.2, what is $\mathcal{B}_2(G)$? What about $\mathcal{B}_1(G)$ and $\mathcal{B}_2(G)$ in the example Figure 5.3? Is there a unique answer?

Solution

For the first question, simply remove B_6 from $\mathcal{B}_1(G)$. For the second question, it is not uniquely defined for $\mathcal{B}_1(G)$! In the next sections we will indeed need to argue that under certain assumptions, for k large enough, we can indeed speak of *the* chain $\mathcal{B}_k(G)$ (meaning it's unique). In this example, $\mathcal{B}_2(G)$ is in fact uniquely defined, for both of the longest chains, if you remove the last two blocks, you end up with the chain (B_0).

Exercise*

Assume a network with 10 nodes, at most 1 of which is *faulty*. Suppose we cyclically change the leader in each round. For which value of k is $\mathcal{B}_k(G)$ well-defined?

Coming back to our goal of *finality*, given a parameter k, we would like to design a protocol that satisfies *finality* when the honest node i considers the blocks in $\mathcal{B}_k(G_t)$ as finalized. Here we use G_t to denote the graph corresponding to the blockchain in round t.

So what should k be? If you back to the blockchain protocol description that we gave in Section 5.1, you will see that there was no mention of a parameter k. Indeed, the parameter k is not a parameter of the protocol! Instead, it is a parameter that the user / node / client should decide for themselves, depending on their application!

If we again take cryptocurrencies as motivating example, then it's easy to see that there are competing interests when it comes to the parameter k. A smaller value of k means that blocks get finalized more quickly, and thus transactions can be handled more quickly. However, at the same time, a smaller value of k also means that it becomes easier for faulty nodes to roll back the chain to a block that was more than k blocks deep in the chain. Meaning that you might not be able to trust the fact that the transactions in $\mathcal{B}_k(G)$ actually took place.

Continuing the example, a client selling a cup of coffee might be willing to trust all transactions in $\mathcal{B}_1(G)$. Meaning that they will hand you your cup of coffee after waiting for only a single new block to be added to the longest chain. A client selling a house however, would probably want a bit more security before handing you the keys to the house, they might want to wait until the payment is contained in $\mathcal{B}_{100}(G)$.

In the remainder of this chapter we will study various scenarios and prove that

(1) $\mathcal{B}_k(G)$ is well-defined (provided k is large enough),
(2) Blocks contained in $\mathcal{B}_k(G)$ can be considered final.

What do we mean by well-defined? As we have seen in the second-to-last exercise, it can happen that there are several longest chains. In such a case, we say that $\mathcal{B}_0(G)$ is not well-defined because it is not a unique object. The blockchain from Figure 5.3 is an example. Here $\mathcal{B}_0(G)$ is not well-defined, because there are two longest chains. An important observation is that $\mathcal{B}_2(G)$ in that example is well-defined: the two longest chains agree on all blocks, except the last 2. In other words, if you take either of the two longest chains and remove the last two blocks, then you end up with the same chain. In this example, we have $\mathcal{B}_2(G) = (B_0)$.

If you think about it, $\mathcal{B}_k(G)$ is always well-defined "provided k is large enough": if we take k to be one less than the number of blocks in the longest chain, then we always have $\mathcal{B}_k(G) = (B_0)$. Of course, this is not a very satisfying situation for practical applications. We would like to show that $\mathcal{B}_k(G)$ is well-defined even if k is a (small) constant.

A final remark before we continue: in the rest of the chapter we will deal with protocols involving randomness,

for example in the decision of who gets to create a block. We therefore cannot guarantee that properties (1) and (2) hold with certainty. The best we can hope for is a statement that (1) and (2) hold *with high probability*. We will make this more precise as we go along.

If we go back to our discussion in Section 5.1.1, we see that honest nodes will always extend the longest chain, whereas faulty nodes might choose to try to extend a shorter chain in the hope of turning it into the longest chain. From the perspective of the honest nodes, the worst case scenario is that the honest nodes keep adding nodes to one chain, while the faulty nodes are collaborating to create an entirely different chain. If the faulty nodes ever manage to extend their chain further than the one of the honest nodes, it would become the longest chain, causing the honest nodes to abandon their original chain. In this worst-case example, no block on the original chain of the honest nodes is finalized. Can we prevent this?

As a motivating example, let us consider the case where two thirds of the nodes are honest and a third of the nodes is faulty. If a new block is proposed by a node chosen uniformly at random[6], then a new block is thus proposed by an honest node with probability $2/3$ and by a faulty node with probability $1/3$. In the long run, we thus expect that the vast majority of the blocks is proposed by honest nodes. In the worst-case situation described above, the chain of the faulty nodes had to become longer than the chain of the honest nodes. In this example, we expect the chain of the honest nodes to be twice as long as the chain of the faulty nodes. It is a nice probability question to turn this "expectation" into a statement that holds with high probability: for a fixed number of blocks K, what is the probability that at least $K/2 + 1$ blocks are created by faulty nodes?

This example shows that it is very unlikely that the faulty nodes are able to change the longest chain *entirely*, meaning that the only block in common will be the genesis block B_0. Can we say more? Intuitively, I hope you agree that the answer is yes. If two thirds of the nodes are honest, then if the chain becomes long enough, we can probably safely say that the first couple of blocks on the chain are *final with high probability*. Here, by *final with high probability* we mean that with high probability over the randomness in the protocol (i.e., the leader selection) the first couple of blocks will remain part of the longest chain indefinitely.

In the next two sections we make this intuition more precise. We first introduce the concept of *balanced leader sequences*. This allows us to distill the key properties of the "two thirds vs. one third" example. We then show that balanced leader sequences ensure that "the first couple of blocks will remain part of the longest chain indefinitely": we define the common prefix property.

5.5 Balanced leader sequences

Eventually, we would like to show that our blockchain protocol satisfies for example *liveness*. For that property, it is important to distinguish between two honest nodes. From the perspective of the blockchain however, there is no need to make this distinction: all honest nodes would act the same. Similarly, we always assume that the faulty nodes are collaborating anyway, so we might as well treat all faulty nodes as identical.

This means that if we think about the sequence of leaders[7] in our protocol, we only need to keep track of who was honest and who was faulty. We will thus represent the leader sequence as a sequence of H's and F's, where H stands for *honest* and F for *faulty*. For example, if the first two leaders are honest, the third is faulty, and the fourth is honest, then this would correspond to the sequence H, H, F, H.

We now state the key definition of this section. It depends on a parameter w, a positive integer, that stands for window.

[6] This might sound artificial, but it is actually a realistic assumption for many protocols. Proof of work protocols, such as Bitcoin, satisfy this property for example.

[7] Remember, the leader is the one that proposes a new block.

Definition 5.1 (w-balanced leader sequence)**.** We say that a leader sequence $\ell_1, \ell_2, \ell_3, \ldots \in \{H, F\}$ is *w-balanced* if, in every window $\ell_i, \ell_{i+1}, \ldots, \ell_{j-1}, \ell_j$ of length at least w, the number of H's is strictly larger than the number of F's.

Here are some observations to help you get familiar with the notion of w-balanced leader sequences. (The explanation is contained in the footnotes, please take a minute to think about the statements before looking at the footnotes.)

- If a leader sequence contains at least one F, then it is not 1- or 2-balanced.[8]
- If a leader sequence is w-balanced, then it is also w'-balanced for any integer w' that satisfies $w' > w$.[9]

The second observation motivates us to find the smallest value w for which a leader sequence is w-balanced.

We will see in the next couple of sections that, as long as the leader sequence is w-balanced, then the blocks in $\mathcal{B}_k(G)$ can be considered final, for a large enough k. We now investigate several scenarios in which we can prove that a leader sequence is w-balanced for some value of w.

The first scenario is about the permissioned setting and assumes we elect the leader in a cyclic fashion. Recall that in the permissioned setting, the number of nodes is fixed and denoted by n. By electing the leader in a cyclic fashion, we mean that we fix an arbitrary order in which the n nodes will be elected as leader. To obtain our leader sequence, we then repeat this fixed order indefinitely.

> Exercise 1
>
> Consider a set of n nodes, with $f < n/3$ faulty nodes. Assume we select the leader in a cyclic fashion. Show that the resulting leader sequence is n-balanced.

> Exercise 2
>
> Consider a set of n nodes, with $0 < f < n/2$ faulty nodes. Assume we select the leader in a cyclic fashion. Show that there is a worst-case scenario such that the resulting leader sequence is not n-balanced.

The second scenario that we consider adds randomness to the process. We still consider the permissioned setting (for now), which means there are n nodes. In each round, we elect the leader uniformly at random from the set of n nodes. That is, each node is equally likely to be chosen as the leader. This is a second natural approach to take in the permissioned setting. More importantly, we will see that this approach extends very naturally to the *permissionless* setting.

5.5.1 Random leaders are balanced: the intuition

To build some intuition, let's see how many *faulty* nodes are in a typical leader sequence. Let α denote the fraction of *faulty* nodes in the network, i.e., let $\alpha = f/n$. We assume from now on that $\alpha < 1/2$. This is a pretty harmless assumption: if more than half the nodes are faulty, there is no hope for achieving a w-balanced leader sequence for any value of w.

[8] Indeed, for $w \in \{1, 2\}$, take a window of length w that contains an F. In that window the number of H's is at most 1 (could be zero) and the number of F's at least one.

[9] The key observation here is that the definition of w-balanced leader sequences requires the property (strict majority of H's) to hold for every window of length *at least* w. If the property holds for every window of length at least w, then it in particular holds for every window of length at least w' whenever w' is strictly larger than w. Another way to put this is that the definition becomes easier to satisfy the larger w is, since there are less windows to verify.

- If we elect a leader uniformly at random from the set of n nodes, then the probability that the leader is faulty is thus equal to α.
- If we elect K leaders in this way, then we thus expect αK of the leaders to be faulty.
- Since $\alpha < 1/2$, we thus expect strictly less than half the leaders to be faulty.

Can we turn this "on expectation" statement into a "with high probability" statement? To gain some intuition, you are invited to experiment with different values of α and K, using the Python code snippets below.

To create one such leader sequence, use the following code:

```
import numpy as np

import matplotlib.pyplot as plt

def sample_leaders(K, alpha):
    """
    Samples K Bernoulli random variables, each with probability alpha.

    Parameters:
        K (int): Number of Bernoulli random variables to sample.
        alpha (float): Probability of success (1) for each Bernoulli trial.

    Returns:
        numpy.ndarray: Array of sampled Bernoulli random variables (0s and 1s).
    """
    samples = np.random.binomial(n=1, p=alpha, size=K)
    return samples
```

```
# Example usage:
K = 10      # number of leaders
alpha = 0.45 # probability of leader being faulty
samples = sample_leaders(K, alpha)
print("Sampled leaders:", samples)

print("Fraction of faulty leaders:", np.sum(samples)/K)
```

```
Sampled leaders: [0 0 0 0 0 0 0 1 1 1]
Fraction of faulty leaders: 0.3
```

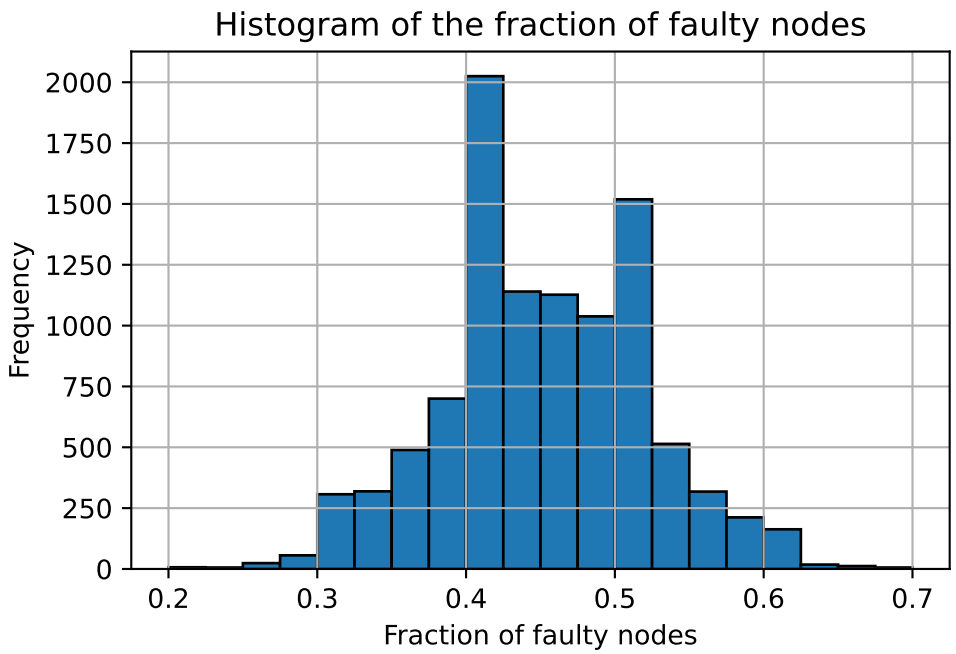
If you run the above code several times, you will see that the outcome varies a bit. The following code allows you to run the process T times and create a histogram of the fractions of faulty nodes.

```
def sample_T_sequences(T, K, alpha):
    """
    Generates T leader sequences of length K,
    where alpha is the fraction of faulty nodes.

    Returns:
```

```
        fractions: List of fractions representing the
        fraction of faulty leaders in each leader sequence.
    """
    fractions = []
    for _ in range(T):
        sample = sample_leaders(K, alpha)
        fractions.append(np.sum(sample)/K)
    return fractions

def plot_histogram(fractions, bins=20):
    """
    Plots a histogram of the fractions.

    Parameters:
        fractions (list): List of fraction values.
        bins (int): Number of bins for the histogram.
    """
    plt.hist(fractions, bins=bins, edgecolor='black')
    plt.title("Histogram of the fraction of faulty nodes")
    plt.xlabel("Fraction of faulty nodes")
    plt.ylabel("Frequency")
    plt.grid(True)
    plt.show()
```

```
# Example usage:
T = 10000     # Number of leader sequences
K = 50       # Number of leaders per sequence
alpha = 0.45  # Probability of success in each Bernoulli trial

fractions = sample_T_sequences(T,K,alpha)

print("The fraction of leader sequences that are balanced is:",
    sum(1 for f in fractions if f < 0.5)/T)

# Plot the histogram of fractions
plot_histogram(fractions)
```

```
The fraction of leader sequences that are balanced is: 0.7238
```



Try changing the value K in the above snippet to 500, what do you observe?

If you are lucky[10], you have been able to make the following two observations:

- The longer the leader sequence is, the larger the probability is that strictly less than half the leaders are faulty.
- If you choose α closer to $1/2$, you need to take a longer leader sequence to ensure that a large fraction of the T leader sequences has the property that strictly less than half the leaders are faulty.

5.5.2 Random leaders are balanced: the math

To prove that our intuition from above is correct, we will two useful results from probability theory.

The first one is a so-called *concentration inequality*. Roughly speaking, a concentration inequality provides a quantitative statement about the probability that a random variable deviates more than a certain amount from its mean. They typically take on the form[11]

$$\Pr(|Z - \mathbb{E}[Z]| \geq t) \leq \text{ something small.}$$

Hoeffding's inequality: The concentration inequality that we use is called Hoeffding's inequality. This inequality is often used in the computer science literature. It applies to a more general setting than what we need here. We first state it in its general form and then we discuss the implications for our setting.

Let $X_1, \ldots, X_K$ be *independent* random variables such that $a_i \leq X_i \leq b_i$. Consider the sum of the random variables:

$$S_K := X_1 + X_2 + \ldots + X_K.$$

[10] In the next section, you will learn that not much luck was needed…

[11] If you want to know (much) more about concentration inequality, I highly recommend Vershynin's book on high-dimensional probability.

As usual, let us use $\Pr(\cdot)$ and $\mathbb{E}[\cdot]$ to denote probability and expectation respectively. Then Hoeffding's inequality states the following.

Theorem 5.1 (Hoeffding)**.** *Fix a $t > 0$. Then we have*

$$\Pr(|S_k - \mathbb{E}[S_k]| \geq t) \leq 2\exp\left(-\frac{t^2}{\sum_{i=1}^{k}(b_i - a_i)^2}\right).$$

To get a feeling for this bound, let us discuss how we will apply it. In our setting, we want to count the number of faulty nodes in a leader sequence of length $K \geq w$. The variable X_i will take on the values 0 and 1 representing honest and faulty respectively. In this way, S_k indeed counts the number of faulty nodes in the leader sequence of length K. As bounds, on X_i we can take $a_i = 0$ and $b_i = 1$. Hoeffding's inequality then shows that

$$\Pr(|S_K - \mathbb{E}[S_K]| \geq t) \leq 2\exp\left(-\frac{t^2}{K}\right).$$

At this point, a natural reaction would be: wait a minute, we have not specified the distribution of the X_i's! Indeed, the distribution of the X_i's is irrelevant for Hoeffding's inequality. All that matters is that the bounds a_i and b_i are known.

So what is the distribution of "our" X_i? In our application of counting the number of faulty nodes in a leader sequence, each X_i will be a random variable that takes on the value 0 with probability $1 - \alpha$ and the value 1 with probability α.[12] Note that each of the X_i's is identically distributed. They are independent random variables by due to the design of our protocol. So what does Hoeffding's inequality say for our setting? The only thing that remains is to compute $\mathbb{E}[S_K]$. By linearity of the expectation, we have

$$\begin{aligned}\mathbb{E}[S_K] &= \mathbb{E}[X_1 + X_2 + ... + X_k] \\ &= \mathbb{E}[X_1] + \mathbb{E}[X_2] + ... + \mathbb{E}[X_K] \\ &= \alpha K.\end{aligned}$$

Hoeffding's inequality thus gives us the following bound:

$$\Pr(|S_K - \alpha K| \geq t) \leq 2\exp\left(-\frac{t^2}{K}\right).$$

Our goal is to show that the sequence $X_1, X_2, \ldots, X_K$ is *balanced*: we want to prove that $S_K < K/2$ *with high probability*. Hoeffding's inequality allows us to quantify this probability. Indeed, we will apply the bound with $t = \left(\frac{1}{2} - \alpha\right)k$ (or a tiny bit smaller if we want the strict inequality). We then obtain

$$\Pr(S_K \geq K/2) \leq 2\exp\left(-\left(\frac{1}{2} - \alpha\right)^2 K\right). \tag{5.1}$$

The key observation here is that the right hand side decays exponentially with K! A second observation is that the bound deteriorates as α approaches $1/2$, as it should (do you see why?). Both of these observations support the intuition from the previous section: if the fraction of faulty nodes α is strictly less than $1/2$, then if we look at a long enough leader sequence (length K), the honest nodes outnumber the faulty nodes with high probability.

[12] As a side remark for those who recently took a course on probability, this means each X_i is drawn according to the Bernoulli distribution with parameter α.

> **Exercise**
>
> Revisit the numerical examples from the previous section, does the bound from Equation 5.1 match your observations?

The union bound: The above allows us to argue that one particular window is balanced, with high probability. However, to argue that the entire leader sequence is w-balanced, we have to consider *all* windows of length at least w. To do so, we will need a second fundamental tool from probability theory: the union bound.

We will need the bound in the following form. Let $E_1, \ldots, E_N$ be some events, then we have the inequality

$$\begin{aligned}&\Pr(\text{at least one of the events } E_1, \ldots, E_N \text{ occurs})\\ &\leq \sum_{i=1}^{N} \Pr(\text{event } E_i \text{ occurs}).\end{aligned}$$

For two events A and B, the union bound is a simple consequence of the identity

$$\Pr(A \cup B) = \Pr(A) + \Pr(B) - \Pr(A \cap B).$$

Indeed, it suffices to observe that $\Pr(A \cap B) \geq 0$.

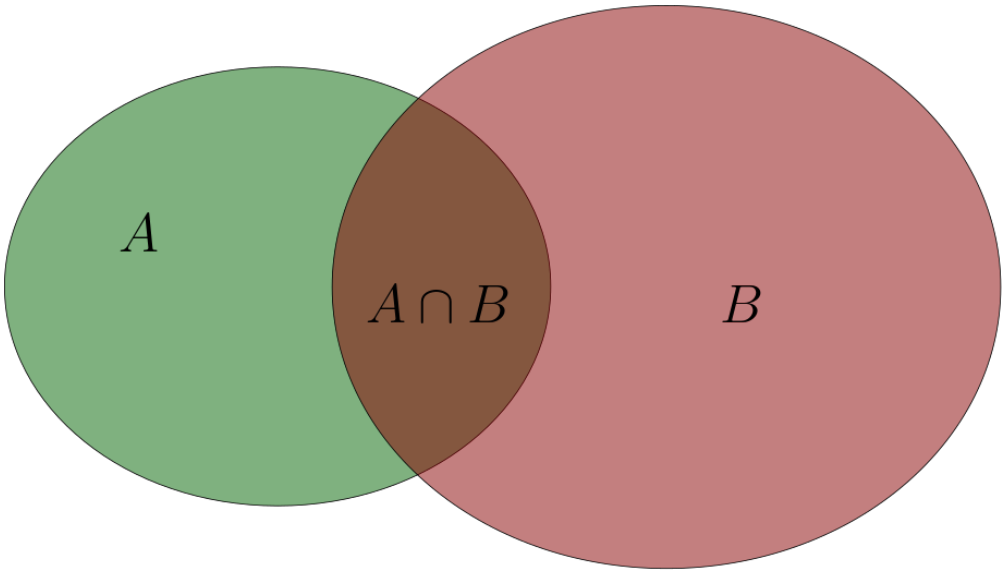


Figure 5.4: Union bound

The union bound for N events can be derived from the one for two events using induction.

> **Exercise**
>
> Prove the above union bound for N events.

How will we apply the union bound? We want to argue that for a given leader sequence, all windows of length at least w have the property that there is a majority of honest leaders. To each window i of length at least w we associate the event E_i that states that at least half the leaders in window i are *faulty*. Notice that

we define the event as the *bad event*. In this way, the union bound gives us an upper bound on the probability that at least one bad event occurs. Why do we care about such an upper bound? It precisely gives us a lower bound on the complementary event: the probability that all windows are balanced. In formulas,

$$\begin{aligned}&\Pr(\text{none of the events } E_1, \dots, E_N \text{ occurs})\\ &\geq 1 - \sum_{i=1}^{N} \Pr(\text{event } E_i \text{ occurs}).\end{aligned}$$

Random leader sequences are balanced: We now combine the above two key ingredients. Our goal is to show that random leader sequences are w-balanced, provided w is large enough. Formally, we will show the following.

Theorem 5.2 (Random leader sequences are balanced)**.** *A random leader sequence of length T in which the probability that a leader is faulty is $\alpha \in [0, 1/2)$ is w-balanced with probability $1 - \delta$ if*

$$w \geq \frac{(1 + 2\log(T) + \log(1/\delta))}{\left(\frac{1}{2} - \alpha\right)^2}.$$

Proof. Let $\alpha \in [0, 1/2)$ and consider a leader sequence $\ell_1, \ell_2, \dots, \ell_T$ of length T in which each leader is chosen independently. The probability that a leader is faulty is α. Fix some integer w. We give a lower bound on the probability that the leader sequence is w-balanced.

To do so, we have to show that in each window of length at least w, there is a strict majority of honest leaders. Let N denote the number of windows of size at least w in a sequence of length T. For future reference, let us observe that $N \leq T^2$: each window is uniquely defined by its start and end point, both of which are integers between 1 and T.

For $i \in [N]$, let E_i denote the event that at least half the leaders in the i-th window are faulty. Then our leader sequence is w-balanced precisely if none of these events occur. The union bound gives us the following lower bound:

$$\begin{aligned}&\Pr(\text{none of the events } E_1, \dots, E_N \text{ occurs})\\ &\geq 1 - \sum_{i=1}^{N} \Pr(\text{event } E_i \text{ occurs}).\end{aligned}$$

We are thus left with providing an upper bound on the quantity

$$\sum_{i=1}^{N} \Pr(\text{event } E_i \text{ occurs}).$$

For that, let us first consider a fixed $i \in [N]$. By assumption, the i-th window has length K_i that is at least w. Hoeffding's inequality, and in particular Equation 5.1, then gives us the upper bound

$$\begin{aligned}\Pr(\text{event } E_i \text{ occurs}) &\leq 2\exp\left(-\left(\frac{1}{2} - \alpha\right)^2 K_i\right)\\ &\leq 2\exp\left(-\left(\frac{1}{2} - \alpha\right)^2 w\right),\end{aligned}$$

where in the second inequality we have used that $K_i \geq w$. Note that the latter upper bound does not depend on i! We therefore have

$$\sum_{i=1}^{N} \Pr(\text{event } E_i \text{ occurs}) \leq 2N \exp\left(-\left(\frac{1}{2}-\alpha\right)^2 w\right)$$
$$\leq 2T^2 \exp\left(-\left(\frac{1}{2}-\alpha\right)^2 w\right).$$

This almost concludes the proof: we have obtained an upper bound on the probability that our leader sequence is not w-balanced. By inspection, the bound is decreasing in w. For which value of w is this probability at most δ? To determine that, we solve the equation

$$\delta = 2T^2 \exp\left(-\left(\frac{1}{2}-\alpha\right)^2 w\right)$$

for w. Taking the natural logarithm on both sides gives

$$\log(\delta) = \log(2) + 2\log(T) - \left(\frac{1}{2}-\alpha\right)^2 w.$$

Rearranging gives the claimed lower bound on w (using that $\log(2) \leq 1$):

$$w \geq \frac{(1 + 2\log(T) + \log(1/\delta))}{\left(\frac{1}{2}-\alpha\right)^2}.$$

□

5.5.3 Random leaders are balanced: the takeaway message

Since the proof was a bit lengthy, let us restate what we have just shown, so that we can discuss the main takeaway message.

Theorem 5.3 (Random leader sequences are balanced)**.** *A random leader sequence of length T in which the probability that a leader is faulty is $\alpha \in [0, 1/2)$ is w-balanced with probability $1 - \delta$ if*

$$w \geq \frac{(1 + 2\log(T) + \log(1/\delta))}{\left(\frac{1}{2}-\alpha\right)^2}.$$

What do we learn from this theorem? Let us work through an example. Suppose that the fraction of faulty nodes in the network is at most $1/3$, that is, let $\alpha = 1/3$. Then $\left(\frac{1}{2}-\alpha\right)^2 = 1/36$, so the lower bound becomes

$$w \geq 36(1 + 2\log(T) + \log(1/\delta)).$$

Which values of T and δ are realistic? This of course again depends on your application. The important observation here is that the logarithm function grows *very slowly*; here is a table of the value of the natural logarithm for the first 10 powers of 10.

Power of 10	Value (10^k)	Natural Logarithm (log(10^k))
10^0	1	0.0000
10^1	10	2.3026
10^2	100	4.6052
10^3	1,000	6.9078
10^4	10,000	9.2103
10^5	100,000	11.5129
10^6	1,000,000	13.8155
10^7	10,000,000	16.1181
10^8	100,000,000	18.4207
10^9	1,000,000,000	20.7233

For example, if we consider a leader sequence of length $T = 1,000,000,000$, with $\alpha = 1/3$ and $\delta = 1/1,000,000,0000$, then the sequence is w-balanced with probability at least $1 - \delta$ whenever $w \geq 26(1 + 3 \cdot 20.7233) \approx 1643$. A blockchain of length $T = 1,000,000,000$ is probably more than we need in any real application, but the point is that w is very small compared to T and $1/\delta$.

Is the bound tight?

So far, we have proven *an* upper bound on the probability that a random leader sequence is not w-balanced. It is natural to ask whether that bound is *tight*? In other words, can the upper bound be improved? The short answer is yes; the bound can be improved. If you revisit the arguments that we made you will notice that there were essentially three estimates that we made:

1. We used Hoeffding's inquality to upper bound the probability that S_K deviates a lot from its mean.
2. We used the union bound to upper bound the probability of a union of events by the sum of the individual probabilities.
3. We overestimated the number of events in our application of the union bound.

Each of these estimates is likely not tight. Hoeffding's inequality for example only requires the assumption that the random variables are independent and satisfy $a_i \leq X_i \leq b_i$. In our application, we know much more. Indeed, we know the distribution of each of the X_i's: they are Bernoulli random variables. As such, their sum follows the binomial distribution. We could thus replace Hoeffding's inequality with properties of the binomial distribution.[13] The second estimate, the union bound, is perhaps the most loose upper bound. The union bound is an equality only if the all the events are pairwise disjoint. Can you see why? (Hint, consider the case of two events A and B, when do you have equality?) In our application, the events correspond to windows not being balanced. Since windows can overlap, it is reasonable to imagine that the events also overlap to some extent. The third estimate, counting the number of possible windows in a leader sequence of length T is in fact pretty close to tight: we used as upper bound T^2 and one (you?) can show that the number of windows of length at least w is at least cT^2 for some small constant c.

From a qualitative perspective though, our analysis "got the job done": the theory supports the intuition that we gathered through numerical examples. To complete the circle, you can try to use numerical methods to investigate how tight the above bound is. The following exercise assumes some familiarity with Python (and hence is marked by a *).

[13] Here it is important that each of the X_i's is independent and *identically* distributed according to the Bernoulli distribution. In the permissionless setting, the random variables might not satisfy the same distribution anymore.

Exercise*

Consider a leader sequence of length $T = 100$, with $\alpha = 1/3$, set $\delta = 0.01$. What is the lower bound on w that follows from Theorem 5.3? Use Python to 1000 such leader sequences. For each of the leader sequences, determine the smallest value of w for which it is w-balanced. Make a histogram of the w values. What do you observe?

Permissioned vs permissionless: At the start of this section we restricted ourselves to the *permissioned* setting. That is the setting that we are familiar with. It also gives us a clear definition for the probability that a leader that is selected uniformly at random is faulty (namely $\alpha = f/n$). In the remainder of the section, we then only worked with α. What about the permissionless setting? The simple definition of α from before no longer works since nodes are allowed to join and leave the network freely. However, if you read the arguments carefully, you will realize that all we needed was *an upper bound* on the probability that a leader is faulty. Later on, in the permissionless setting, we will turn this into an assumption.

5.6 The common prefix property

In Section 5.4 we discussed the problem of *finalizing* blocks on a blockchain. Intuitively, we argued that the blocks on the longest chain are final, except for the last few.

More concretely, for a blockchain represented by a directed in-tree G and an integer k, we defined the object $\mathcal{B}_k(G)$ as

$$\mathcal{B}_k(G) := \text{ the longest chain of } G, \text{ with the last } k \text{ blocks removed.}$$

We have seen examples in which $\mathcal{B}_k(G)$ was not well-defined (in a nutshell: consider a blockchain with two equally long chains and a very small value of k). Intuitively, equally long longest chains occur when the number of honest leaders is roughly equal to the number of faulty leaders. In this section we show that this is the only obstruction. For this, we will crucially use the concept of w-balanced leader sequences that we introduced and studied in Section 5.5. We show that if the leader sequence is $(2k+2)$-balanced, then $\mathcal{B}_k(G)$ is well-defined. Formally, we prove the following theorem.

Theorem 5.4 (Common prefix property of longest-chain consensus)**.** *If the leader sequence $\ell_1, \ell_2, \ell_3, \ldots$ is $(2k+2)$-balanced, and assumptions A1, A4', A5 hold, then for every possible resulting in-tree G of blocks known to the honest nodes, $\mathcal{B}_k(G)$ is well-defined.*

Before proving the theorem, let us reflect on the statement for a moment. As usual, we want to assume as little as possible about the behavior of faulty nodes. In this case, that means that we do not place any restrictions on where faulty nodes can add blocks to the blockchain. As a consequence, the leader sequence does not completely specify the resulting in-tree! The statement of the theorem accounts for this: we show that regardless of the behavior of the faulty nodes – "for every possible resulting in-tree G" – the object $\mathcal{B}_k(G)$ is well-defined.

We prove the theorem under the assumption A4', which is a slightly stronger version of assumption A4. We do so for two reasons: it simplifies the proof, and it is an assumption that in fact holds when we consider proof of work protocols such as Bitcoin.

5.6.1 Proof of Theorem 5.4

The plan is to prove the contrapositive statement: if there is an in-tree G of blocks known to the honest nodes for which $\mathcal{B}_k(G)$ is not well-defined, then the leader sequence is not $(2k+2)$-balanced.

To that end, assume G is an in-tree of blocks known to the honest nodes for which $\mathcal{B}_k(G)$ is not well-defined. Since $\mathcal{B}_k(G)$ is not well-defined, this means that there exist two longest chains in G that do not agree if we remove the last k blocks from each of these two chains. For the remainder of our argument, we will focus on these two chains. It will be useful to visualize these two chains. The following is the picture you should have in mind:

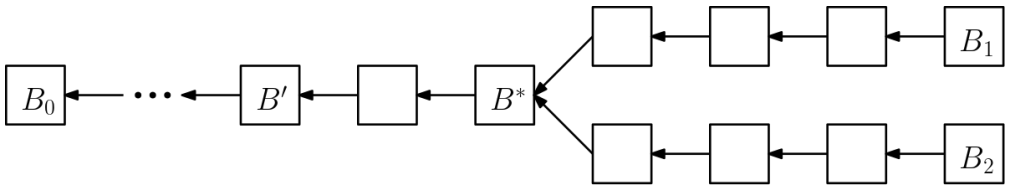


Figure 5.5: Two longest chains

Formally, we assume that there are two longest chains. The first ends at a block labeled B_1 (the top chain) and the second ends at a block B_2 (the bottom chain). We assume that the block where the two chains meet for the first time is labeled by B^*. Since we are assuming that $\mathcal{B}_k(G)$ is not well-defined, the length of the path between B_1 and B^* is at least $k+1$, and the same holds for the path between B_2 and B^*.

To make it concrete, in Figure 5.5, we have drawn 4 blocks after B^* on either of the chains, so this would be an example in which $\mathcal{B}_1(G)$, $\mathcal{B}_2(G)$, and $\mathcal{B}_3(G)$ are not well-defined. It should be clear how to generalize the figure for larger values of k.

We will now use the fact that the length of the paths connecting either B_1 or B_2 to B^* is at least $k+1$ to argue that the leader sequence is not $(2k+2)$-balanced. In other words, we will identify a window in the leader sequence of length at least $2k+2$ in which there are at least as many faulty leaders as honest leaders.

To do so, it will be useful to introduce the notion of *height* of a block.

Definition 5.2 (Height)**.** Given an in-tree G with root B_0, we define the height of a block B as the length of the path connecting B to B_0.

Here are some simple observations:

- The height of B_1 is equal to the height of B_2. We let h denote the height of B_1.
- The height of B^* is at most the height of B_1 (or B_2) minus $k+1$, that is, its height is at most $h-(k+1)$.

We make one more observation that is slightly less simple and therefore deserves a short proof.

Lemma 5.1 (One honest block per height)**.** *Under assumption A5, the heights of honestly produced blocks are strictly increasing over time. In particular, there is at most one honestly produced block per height.*

Proof. Suppose that A and B are two blocks produced by honest nodes. Since there is only one leader per round, one of the blocks was created first. Suppose that A was created before B (otherwise, swap the roles of A and B in the following argument). Now let h_A denote the height of block A. Since A was created by an honest node and we are working under the assumption A5, A was immediately announced to all honest nodes at the time of its creation. This that at the time that B was created, its honest creator was aware of at least one block at height h_A. Since an honest node only extends the longest chain that it is aware of, the height of B is at least h_A+1. This proves that the height of honestly created blocks is strictly increasing over time.

The final statement of the lemma is an immediate corollary of the first part. □

We are now ready to finish up the proof of Theorem 5.4. Recall that B^* is the first block that is common on the two longest chains. Let B' denote the first block that is common to both longest chains *and produced by an honest leader*. If B^* is produced by an honest leader, then $B' = B^*$. Otherwise, it will be some block on the path between B^* and B_0. (Existence of such a block is ensured by assumption A1: the block B_0 is produced by an honest leader.) Let r' be the round in which B' was created.

Since B' is created by an honest node, all blocks that have B' as an ancestor (on the path to B_0) are created *after* round r'. Let r_{end} denote the round in which the last of B_1 and B_2 was created. We argue that the leader sequence starting at r' and ending at r_{end} is not $(2k+2)$-balanced.

Indeed, let h^* denote the height of block B^* and similarly h' the height of B'. By our assumption, the $h^* - h'$ blocks between B' and B^* are created by faulty nodes. Since $\mathcal{B}_k(G)$ is not well-defined, we know that the h, height of B_1 (or equivalently B_2), is at least $h * +k+1$. By Lemma 5.1, we know that for each of the heights $h^*+1, h^*+2, \ldots, h$, there is at most one honestly produced block. The same argument shows that for each of those heights, there was also at least one block produced by a faulty node. This means there are at most $h - h^*$ honest leaders after r' and at least $h - h^*$ faulty leaders. In other words, there was no strict majority of honest leaders in the window from r' to r_{end}. It remains to observe that this window has length at least $2k+2$. Indeed, the two chains ending at B_1 and B_2 contain at least $k+1$ blocks that were produced after B^*, and therefore after B'. Since in each round we create exactly one block (assumption A4'), this means that $r_{end} - r' \geq 2k+2$. This concludes the proof.

Now that we have established a sufficient condition under which $\mathcal{B}_k(G)$ is well-defined, we can proceed to show that longest-chain protocols achieve the goals of finality and liveness.

5.7 Finality of longest-chain consensus

Let us recall the (informal) definition of *finality* that we gave earlier this chapter:

> **Goal: Finality** (first version) If an honest node i considers a block B as *finalized* at time t, then this block remains finalized at all times after t.

The above goal is a bit vague since it does not specify *when* an honest node should consider a block finalized. Given the results from the previous sections, it should come as no surprise that we will declare the blocks in $\mathcal{B}_k(G)$ to be final. Indeed, we will prove the following theorem.

Theorem 5.5 (Finality of longest-chain consensus)**.** *Let $G_1 \subseteq G_2 \subseteq \cdots \subseteq G_T$ be a sequence of in-trees with each in-tree G_t having exactly one more block than G_{t-1}. If the common prefix property holds for a given value of k in each of the in-trees $G_1, \ldots, G_T$, then we have*

$$\mathcal{B}_k(G_1) \subseteq \mathcal{B}_k(G_2) \subseteq \cdots \subseteq \mathcal{B}_k(G_T).$$

The sequence of in-trees naturally corresponds to the evolution of a blockchain over time, with G_t denoting the in-tree in round t. In words, the theorem tells us that once a block is part of $\mathcal{B}_k(G_t)$, it will be part of $\mathcal{B}_k(G_{t'})$ for all subsequent rounds $t' \geq t$. The honest nodes can thus trust the blocks in $\mathcal{B}_k(G)$ to be final.

At this point in the course, you are trained to always question the assumptions. For this particular theorem, try to answer the following question:

> Exercise
>
> Why do we not (need to) refer to any of the assumptions A1-A5 in Theorem 5.5?

We now proceed with the proof of Theorem 5.5.

Proof. We assume the common prefix property, with value k, holds for each of the in-trees $G_1, \ldots, G_T$. In other words, we assume that $\mathcal{B}_k(G_t)$ is well defined for each $t \in [T]$. Let us also assume that there exists a pair of rounds $t, t' \in [T]$ with $t < t'$ and a block B with the property that $B \in \mathcal{B}_k(G_t)$, but also $B \notin \mathcal{B}_k(G_{t'})$. We show that this leads to a contradiction. In particular, we will show that there exists some round s in between rounds t and t' for which $\mathcal{B}_k(G_s)$ is not well defined.

By assumption, $B \in \mathcal{B}_k(G_t)$, the block B belongs to the common prefix of all longest chains in round t. Similarly, it does *not* belong to the common prefix of all longest chains in round t'. Let $s \in \{t+1, \ldots, t'\}$ denote the smallest round number for which B is excluded from at least one longest chain in G_s. Let B_s be the block that was added to G_{s-1} to form G_s. By minimality of s, the block B is contained in the common prefix of all longest chains in G_{s-1}. This means that B_s cannot extend a longest chain in G_{s-1}, since otherwise B would also be contained in the common prefix of all (in fact *the*) longest chain in G_s. The block B_s must thus be added to a chain that was previously not the longest chain, creating a new longest chain. Let C_1 denote a longest chain in G_{s-1}. Note that B is part of the chain C_1. Let C_2 denote the newly created longest chain in G_s. Let B^* denote the first block that the chains C_1 and C_2 have in common. Figure 5.6 below illustrates the situation.

We now argue that $\mathcal{B}_k(G_s)$ is not well defined. To do so, observe that $B \in \mathcal{B}_k(G_t)$ implies that B is at least k blocks removed from the end of chain C_1. That means that B^* is at least $k+1$ blocks removed from the end of chain C_1. Since C_1 and C_2 have the same length, B^* is also at least $k+1$ blocks removed from the end of C_2. Both C_1 and C_2 are longest chains in G_s, and we have just shown that we need to remove at least $k+1$ blocks from the ends of C_1 and C_2 to reach the first block that they have in common. This proves that $\mathcal{B}_k(G_s)$ is not well defined. □

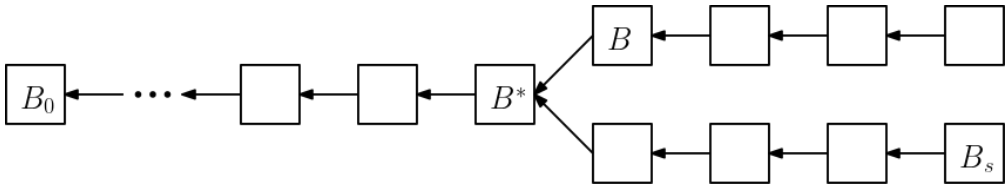


Figure 5.6: Proof of Theorem 5.5.

5.8 Liveness of longest-chain consensus

We finally establish liveness of the longest-chain consensus protocol. Recall from the previous chapter:

> **Goal 2: Liveness.** Every "transaction" submitted by a client to at least one *honest* node is eventually added to every node's local history.

We will in fact only be able to show a weaker version of liveness:

> **Goal 2: Liveness.** (weak version) Every "transaction" submitted by a client to all *honest* nodes is eventually added to every node's local history.

The difference between the two lies in whether a transaction is known to at least one honest node or all of them. In a nutshell, we will be able to guarantee that the longest chain will include infinitely many blocks produced by honest nodes, but we cannot guarantee the same for blocks produced by one specific honest node. Formally, we prove the following.

Theorem 5.6 (Liveness of longest-chain consensus)**.** *Assume the leader sequence $\ell_1, \ell_2, \ldots$ is $(2k+2)$-balanced and assume that the assumptions A1, A4' and A5 hold. Let $G_1, G_2, \ldots$ denote the corresponding sequence of in-trees. For every transaction that is at some point known to all honest nodes, there exists a t_0 such that the transaction will be included in $\mathcal{B}_k(G_t)$ for all $t \geq t_0$.*

Proof. First note that the assumptions A1, A4', A5 and the fact that the leader sequence is $(2k+2)$-balanced allow us to conclude, via Theorem 5.4, that $\mathcal{B}_k(G_t)$ is well-defined for all t. Moreover, we have shown in Theorem 5.5 that finality holds:

$$\mathcal{B}_k(G_1) \subseteq \mathcal{B}_k(G_2) \subseteq \cdots.$$

It thus remains to show that if a transaction is known to all honest nodes at some point, then there is a t_0 such that $\mathcal{B}_k(G_{t_0})$ includes that transaction. Since the transaction is known to *all* honest nodes, and honest nodes include all transactions that they are aware of in a new block, it suffices to show that infinitely many finalized blocks are created by honest leaders.

To do so, we will use the property that the leader sequence is $(2k+2)$-balanced. Let us divide the leader sequence into groups of $2k+2$ consecutive leaders. That is, the first group consists of the leaders $\ell_1, \ell_2, \ldots, \ell_{2k+2}$, the second group consists of the leaders $\ell_{(2k+2)+1}, \ldots, \ell_{2(2k+2)}$, and so on. Since the leader sequence is $(2k+2)$-balanced, every group contains at least $k+2$ honest leaders and at most k faulty leaders.

We now argue that the longest chain grows significantly over time: indeed, Lemma 5.1 shows that the height of honestly produced blocks increases strictly over time. Now consider a longest chain at the end of the T-th group of leaders. There are at least $(k+2)T$ honest leaders in those groups and at most one honestly produced block per height, therefore the longest chain in $G_{(2k+2)T}$ has length at least $(k+2)T$.

How many honestly produced blocks are there on such a longest chain? We know that there are at most kT faulty leaders in the first T groups. Together, they produced at most kT blocks. Therefore, there are at least $(k+2)T - kT = 2T$ honestly produced blocks on such a chain. How many honestly produced blocks are finalized? We consider blocks finalized when they are part of $\mathcal{B}_k(G)$. For $\mathcal{B}_k(G_{(2k+2)T})$ this means it contains at least $2T - k$ honestly produced blocks, since we are removing the last k blocks and they could happen to be honestly produced. This proves that as T grows, the number of honestly produced blocks that are finalized grows as well. In other words, honestly produced blocks get finalized infinitely often and therefore transactions known to all honest nodes *eventually* get finalized. □

5.9 Toward permissionless consensus

In the spirit of "always question the assumptions", for which theorems in this chapter did we rely on the assumption that we are working in the *permissioned* setting?

If you revisit all theorems, you will notice that we used it only to argue that choosing each leader uniformly at random leads to a balanced leader sequence, as long as the fraction of faulty nodes $\alpha = f/n$ is strictly less than $1/2$. Afterwards, all theorems relied on the assumption that the leader sequence is w-balanced for a sufficiently large value of w.

In other words, if we – somehow – establish that a leader election mechanism in the permissionless setting leads to a w-balanced leader sequence, then Theorem 5.5 and Theorem 5.6 immediately show that the longest-chain consensus protocol achieves finality and liveness in that setting as well.

A crucial difference between the permissioned and the permissionless setting is that nodes can join and leave the network freely. In particular, this means that the fraction of faulty nodes in the network might change

over time. Let us assume for the moment that we still select a leader uniformly at random from the set of nodes in the network in that round. Can we still use the arguments from Section 5.5 to argue that the resulting leader sequence is sufficiently balanced with high probability? The answer is yes, under mild assumptions. Indeed, if we assume that the fraction of faulty nodes at any point in time is *at most* $\alpha < 1/2$, then we can still apply Hoeffding's inequality in the same way as we did in the proof of Theorem 5.2.

The assumption that we can elect a leader uniformly at random in the permissionless setting is not one that we can make freely. Selecting a uniformly random element from a set of nodes that the protocol does not know is a hard problem. In the permissionless setting we will need to work to design a leader election protocol that has this property. We will do so in the next chapter.

Chapter 6

Proof of Work

In this final chapter on the computer science aspects of blockchains we make the step to permissionless consensus protocols. More accurately, we discuss the concept of proof of work as a means to select a new block proposer.

We will see that proof of work gives us many desirable properties:

- As long as the strict majority of the work is done by honest nodes, we will get a balanced leader sequence.
- In applications such as Bitcoin it also guarantees assumption A4': each leader can only produce one block.

Proof of work also comes with its own set of problems / challenges. We will see some of them in this lecture and others in the next part of the course.

6.1 What is proof of work?

Throughout your studies you have been exposed to many different proofs. But what constitutes a proof of work? In a nutshell, it is a proof that you have done a certain amount of work.[1]

Let us make this a bit more concrete by connecting it to our application. In the previous chapter we have discussed longest-chain protocols at length. We did that in the relatively safe environment of the *permissioned* setting. Because we were in the permissioned setting, we had a well-defined way of selecting a leader uniformly at random. Indeed, if there are n nodes in the network, then we simply draw a uniformly random number between 1 and n to decide the leader of the next round (who gets to propose a new block).

In the *permissionless* setting this approach no longer works. That's where proof of work comes in: we will use it as a mechanism to choose the leader of the next round. The basic idea is that we decide on some (hard) puzzle and then we declare as the leader of the next round the first node that comes up with a solution to that puzzle. If the puzzle is chosen appropriately, and we will see how to do so soon, then the likelihood of being the first node to solve the puzzle is directly proportional to the computational effort put into solving this puzzle.

[1] The proof here is not a figure of speech: under well-defined assumptions, the proof establishes that you have done a certain amount of work with high probability.

Notice that the mechanism described above does not allow us to sample a node uniformly at random, instead, we sample nodes proportional to their computational efforts. How does this affect the balanced-leader sequence analysis from the previous chapter? Perhaps surprisingly, it almost doesn't! In the permissioned setting we required the fraction $\alpha = f/n$ of faulty nodes to be strictly less than a half. Now, we require the total fraction of computational effort controlled by faulty nodes to be strictly less than a half.

As a side remark, the notion of proof of work actually predates Bitcoin by about 17 years. Dwork and Naor introduced the idea in 1992 in the context of spam-fighting "Combatting Junk Mail", actually before spam was really a problem. See also the related 2002 paper by Back on HashCash.

To illustrate some desirable properties of the "proof", consider the following example. The puzzle is this week's NY Times crossword puzzle. The key property that we want to illustrate here is that the proof is *concise* and *efficiently verifiable*: the proof that you need to provide is simply the filled-in crossword puzzle. To verify it, every node can simply check that all the words are correctly spelled and fit together nicely. Would the NY Times crossword be a good puzzle to use in a blockchain protocol? Probably not… It is reasonable to say that the more effort you put into solving the crossword, the more likely you are to solve it first. Although the likelihood is probably not directly proportional to the effort you put in (at least for me).

6.2 Cryptographic hash functions

To define the puzzles that we want to solve, we need to introduce *cryptographic hash functions*. You might have heard about *hash functions* at some point in your academic career (but don't worry if you didn't); they have many applications in the context of compressing or expanding data, or "spreading out" data (hash tables).

For our purposes, a hash function is just a function from some domain to some range. A unifying feature of hash functions is that they are very easy to compute. A *cryptographic hash function* is designed with the following goal in mind: it should be completely unpredictable. Ideally, if you put a new input into the function, you get an output that looks completely random.

We will formalize this goal as an assumption.

6.2.1 The random oracle assumption

The *random oracle assumption* states that the hash function that we are using may as well be a random function, in the sense that there is no (efficient) way of telling the difference. Here by a *random function* we really mean a random function: chosen uniformly at random from the set of all functions with the given domain and range.

Another way to think of a random function is that it is defined through *lazy evaluation*: the first time the function is applied to a certain input, it picks a uniformly random element from the range as the output. Why do we do this only the first time? If we would also do it a second time, then a single input gets mapped to two different outputs, which would violate the fact that we are talking about a function.

As with any assumption, you should question how realistic this assumption is. It turns out that this assumption is both clearly false and very true at the same time. In the next section we will see an example of a completely deterministic function (you can look up the source code on the internet), that is still very much unpredictable. This deterministic function looks completely random to anyone with bounded computational power.

A very important consequence of the random oracle assumption is that the function is *hard* to invert. By this we mean that for a given output (or set of outputs) it is hard to find an input that maps to that output.

6.3 SHA-256

Here we give a brief illustration of how hashing works. We will use as example the SHA-256 hash function. SHA stands for Secure Hash Algorithm. The number 256 refers to the number of bits of the output.

We will use the python `hashlib` module, which we first need to import.

```
import hashlib
```

Here is an example of how to use SHA256 to hash a short message. The input message will be "Decentralized finance is cool!' ' and the output is a bitstring of length 256.

```
# The message to hash
message = "Decentralized finance is cool!"

# Encode the message to bytes, then hash it using sha256
hashed_message = hashlib.sha256(message.encode()).digest()

# Convert the hash to a bitstring
bitstring1 = ''.join(f'{byte:08b}' for byte in hashed_message)

print(f"The SHA-256 hash as a bitstring is:\n{bitstring1}")

print(f"Length of the bitstring: {len(bitstring1)}")
```

```
The SHA-256 hash as a bitstring is:
10000001011011001111000110001010110010100111111010110010011100011100010111010001010110111010100
Length of the bitstring: 256
```

Let us now investigate the key property of hash functions: they look like a random function. In particular, this implies that it is hard to predict the output of the function. To illustrate this, let us hash a second message that is very similar to the first: "Decentralized finance is cool!!!' '

```
### Second message
# The message to hash
message = "Decentralized finance is cool!!!"

# Encode the message to bytes, then hash it using sha256
hashed_message = hashlib.sha256(message.encode()).digest()

# Convert the hash to a bitstring
bitstring2 = ''.join(f'{byte:08b}' for byte in hashed_message)

print(f"The SHA-256 hash as a bitstring is:\n{bitstring2}")
print(f"Length of the bitstring: {len(bitstring2)}")
```

```
The SHA-256 hash as a bitstring is:
00101011111010100001000000111100100001001110100110110100100111011101011010000111001011101001010
Length of the bitstring: 256
```

If you compare the two strings, you will notice some differences right away. How different are the two strings? If the hash function assigns a random output to every input, then you would expect the two bitstrings to differ in roughly half their entries. Let us use python to count the number of different entries in the two strings.

```
def count_differences(bitstring1, bitstring2):
    # Ensure the bitstrings are the same length
    if len(bitstring1) != len(bitstring2):
        raise ValueError("Bitstrings must be of the same length")

    # Count the differences
    differences = sum(1 for b1, b2 in zip(bitstring1, bitstring2) if b1 != b2)
    return differences


# Calculate the number of differences
difference_count = count_differences(bitstring1, bitstring2)

print(f"Number of different characters: {difference_count}")
```

```
Number of different characters: 131
```

Indeed, for this particular example, the number of different characters is very close to 256/2 = 128.

The takeaway: SHA-256 is widely considered to satisfy the random oracle assumption.

6.4 How do we define a puzzle based on SHA-256?

Now that we have seen a concrete example of a cryptographic hash function, it is time to define our puzzle(s). Remember that SHA-256 takes any input and maps it to a bitstring of length 256. In other words, the range is $\{0,1\}^{256}$.

There is a natural way to think of bitstrings as numbers between 0 and $2^{256}-1$: if $x = (x_{256}, x_{255}, \dots, x_1) \in \{0,1\}^{256}$, then we associate to it the number

$$\sum_{i=1}^{256} x_i 2^{i-1}.$$

To make this a bit less abstract, consider the case of only three bits. In this case,

- the string 000 corresponds to the value 0,
- the string 001 to $1 \cdot 2^{1-1} = 1$,
- the string 111 to $1 \cdot 2^2 + 2 \cdot 2^1 + 1 \cdot 2^0 = 7$.

Why did we discuss this interpretation of bitstrings? This is precisely how we will define our cryptographic puzzles! If we let h denote our hash function (think SHA-256), then the puzzle is as follows.

Definition 6.1 (Canonical Proof-of-Work puzzle)**.** For a given threshold τ, find an input z such that $h(z) \leq \tau$.

The threshold τ determines the difficulty of the puzzle. The two extreme cases being

- $\tau = 2^{256} - 1$: in this case any input is a valid solution. This is thus an extremely easy puzzle!
- $\tau = 0$: in this case we are asked to find an input that maps to the all-zero string. This is an extremely hard task under the random oracle assumption!

> **Exercise**
>
> Under the random oracle assumption, what is the probability of any given input being mapped to the all-zero string?

> **Exercise**
>
> Under the random oracle assumption, what is the probability of any given input z being mapped to an output with $h(z) \leq \tau$?

To build some intuition, it is helpful to think of τ as a number of the form $2^{256-k} - 1$ for some k between 0 and 256. In this case, k specifies the number of zeros that the output should start with. If we go back to the previous section, then the message "Decentralized finance is cool!!!" would thus be a valid solution for the threshold $2^{256-2} - 1$: its hash starts with two zeros.

What is the probability that a single input gets mapped to a string of length 256 starting with k zeros? Well, under the random oracle assumption the output is chosen uniformly at random from the set $\{0,1\}^{256}$. A good way of thinking about a uniformly random element from that set is that, for each of the 256 bits independently, we flip an unbiased coin: if it's heads, we get 1, and if it's tails, we get 0. What is then the probability that we start the sequence with k zeroes? Note that this depends only on the outcomes of the first k coin flips: they all need to be tails. The probability of that is 2^{-k}.

How many inputs would you expect to need to try to find one that maps to a bitstring starting with k zeroes? This is the expectation of a geometric random variable with probability $p = 2^{-k}$. It's expectation is $1/p$ and thus we expect to need to try 2^k different inputs.

The fact that our canonical Proof-of-Work puzzle has such a nicely tunable complexity is crucial for the stability of protocols relying on Proof-of-Work and longest-chain consensus. By tuning the complexity of the puzzle, we can control how frequently new blocks get created. We will get back to how to choose the complexity when we discuss Bitcoin in a bit more depth.

Now that we have decided on a puzzle, it is a good moment to ask ourselves the following question:

> **Exercise**
>
> What would go wrong in a longest-chain protocol if we simply elect as the next leader, the next node that provides us any input z that satisfies $h(z) \leq \tau$ for the current value of τ?

> **Exercise**
>
> Would an NP-hard problem such as Traveling Salesperson be a good alternative for a hard puzzle? Explain why (not).

6.5 What should puzzle solutions encode?

To make use of Proof-of-Work in a longest-chain protocol, we thus need to avoid that nodes can save up valid solutions for use in later rounds. If you go back to the previous chapter, there was another assumption that we promised Proof-of-Work would take care of:

> **A4')** The leader of round r produces exactly one block, and this block claims as its predecessor a block that belongs to a previous round.

Here we discuss how to solve both these problems in one go. We essentially do so by merging the leader election and block-proposal steps. Our protocol will not just accept any solution z that satisfies $h(z) \leq \tau$. Instead, we will require the solution z to have some structure. We ask for the following structure on z:

- It starts by writing down a block B that it proposes,
- It is followed by the identifier $pred$ of the predecessor of the block B,
- The third part is pk, the public key (think identifier), of the node proposing the block.
- The fourth and final part is a $nonce$, which stands for *number used once*.

Definition 6.2 (Format for puzzle solutions)**.** A valid puzzle solution is of the form $z = B||pred||pk||nonce$, where $||$ denotes concatenation.

We can think of the four parts of the solution as four fields in which we can write information. The first two neatly ensure that the node that submits this solution gets to propose exactly one block, and that block has to point to a predecessor that was known at the time of creating the solution. In fact, the block proposer already has to commit to a block *before* they find a valid solution. The third field, containing the identifier of the block proposer can be used to determine who gets the reward for creating that block (we will get to rewards briefly when we discuss Bitcoin). The $nonce$ is perhaps the most mysterious part of the solution. It is there simply as a way to make it easy to generate many inputs corresponding to the same triple $B||pred||pk$: one simply has to change the $nonce$ at the end.

Recall from the previous chapter, our generic blockchain protocol was as follows:

```
A blockchain protocol:

1.  Initialize with a hard-coded genesis block B0
2.  In each round r = 1,2,3,.... do:
    a)  Choose one node i as the leader of round r.
    b)  Node i proposes a set of blocks,
        each specifying a single predecessor block.
```

In a Proof-of-Work protocol, we merge the last two steps:

```
A Proof-of-Work blockchain protocol:

1.  Initialize with a hard-coded genesis block B0
2.  In each round r = 1,2,3,.... do:
    Choose as leader in round $r$ the first node
    that provides a valid puzzle solution
    z = B||pred||pk||nonce with h(z) <= tau.
```

6.6 Sybil-resistance

Now that we understand Proof of Work, let us introduce one of the key problems that Proof of Work solves: *resistance to Sybil attacks*.

A Sybil attack[2] on a computer system is a type of attack in which an attacker tries to gain an advantage by creating a large number of identities. We call a protocol *Sybil-resistant* if no such Sybil attack is possible.

As an example, consider the leader selection mechanism from Chapter 5: we chose a leader uniformly at random from the set of nodes participating in the protocol. It is easy to see that for such a leader selection protocol, a Sybil attack would be very dangerous (if possible!). Indeed, if a faulty node can pose as many nodes, then it greatly increases the chances of being selected as a leader in the next round. For a faulty node, it is a huge advantage if it is selected as a leader much more often, so if possible, this would constitute a Sybil attack. Note that we wrote "if possible": in Chapter 5 we were working in the permissioned setting: nodes need permission to join the network before the start of the protocol and presumably this permission is only granted after some sort of verification of identity.

If we use Proof-of-Work to select the next leader, then we get Sybil-resistance automatically:

Theorem 6.1 (Sybil-resistance of Proof-of-Work)**.** *Suppose n nodes make $\mu_1, \mu_2, \ldots, \mu_n$ distinct nonce guesses per second. Then, for every node i and round r, we have*

$$\Pr(\textit{node } i \textit{ is the round-}r \textit{ leader}) = \frac{\mu_i}{\sum_{j=1}^{n} \mu_j}.$$

At least two remarks are in order here. First, although the theorem references a set of n nodes, this is simply for notational purposes. The Proof-of-Work protocol works in the permisionless setting. Second, the numbers μ_i are often referred to as *hashrates*.

The most important observation about Theorem 6.1 is that the quantity on the right hand side is simply node i's fraction of the overall hashrate; it does not depend on how many identities it controls.

Looking at Theorem 6.1, there is a clear way in which a node can influence the probability with which they are selected as the leader of the next round: they need to increase their hashrate. Assuming the other nodes don't do the same, this would increase the fraction of node i's hashrate in the total hashrate. Creating new hashrate is however not cheap! It essentially amounts to buying new computers to try new nonces. If we for instance assume that currently the amount of hashrate controlled by faulty nodes is less than $1/4$ of the total hashrate, then to mount a successful Sybil-attack, the faulty nodes would have to at least double their hashrate (and then some). This is practically impossible for large implementations such as Bitcoin…

Looking ahead, in the next part of the course you will revisit Sybil-resistance, but then from a different perspective: that of sharing rewards.

6.7 Nakamoto consensus

Combining Proof-of-Work and Longest-chain consensus protocols is sometimes referred to as *Nakamoto consensus*. Bitcoin being of course the most famous example. In this final part of the chapter we will go a bit deeper into the specifics of Bitcoin. In particular, we will discuss how Bitcoin sets and adjusts the difficulty parameter of the puzzles. We then briefly discuss rewards in the Bitcoin protocol (with a more in-depth discussion being deferred to subsequent chapters).

[2]Named after a 1973 book called Sybil, a case study of a woman diagnosed with a dissociative identity disorder.

6.7.1 How to set and adjust the difficulty parameter

First, let us recall the definition of our canonical Proof-of-Work puzzle:

Definition 6.3 (Canonical Proof-of-Work puzzle)**.** For a given threshold τ, find an input z such that $h(z) \leq \tau$.

Which value of τ should we pick? At first glance, the larger τ, the easier it is to find a solution and therefore the more frequently new blocks get created. Moreover, we have argued before that as long as the fraction of hashrate controlled by faulty nodes is strictly less than $1/2$, then the longest-chain consensus protocol based on Proof-of-Work has the property of *balanced leader sequences*. In particular, it will then satisfy liveness and consistency. Note that this argument does not rely on τ at all. So if we create blocks say twice as fast, then it takes half as long before a block gets finalized. So is there any reason to bound τ?

Yes! In Chapter 5 we assumed that honestly created blocks were instantly known to all honest nodes. What this really means is that we assume honest nodes do not inadvertently create a fork. They are always aware of the longest chain(s) in the network. Is that a reasonable assumption? If we assume instantaneous communication for example, then it holds. But instantaneous communication itself clearly does not hold. A more reasonable assumption is that communication is not instantaneous, but happens for example with a delay of at most 10 seconds between any pair of nodes. In that case, if the (expected) time between blocks is much longer than 10 seconds, say 10 minutes, then it is fair to say that if a block is honestly created, then it is "immediately" known to all honest nodes.

The time frame of 10 minutes is in fact the one that Bitcoin aims for. It chooses the parameter τ in such a way that on expectation it takes 10 minutes for a new block to be created.

If the total hashrate is H nonces per second, and the probability of success is $p = \tau/2^{256}$, then the average duration L of the round in seconds satisfies

$$L = \frac{1}{Hp} = \frac{2^{256}}{H \cdot \tau}.$$

In other words, if the total hashrate is known, then it is easy to choose the difficulty of the puzzles in such a way that the expected duration of a round is 10 minutes.

In a permissionless protocol, there is however no way to know the total hashrate. Nakamoto came up with a very elegant solution for this problem. Nakamoto anticipated that the total hashrate could vary significantly over time. They therefore built into the protocol a mechanism for how to adjust the difficulty. The idea is quite simple. If we expect a block to be created every 10 minutes, then after two weeks we expect to see $14 \cdot 24 \cdot 6 = 2016$ new blocks. The mechanism Nakamoto put in place is thus the following: every time 2016 new blocks have been created the protocol evaluates how many days have passed since the previous batch of 2016 blocks. If that time is $\beta \cdot 14$ days, then the difficulty parameter τ gets updated to $\beta \cdot \tau$.

As an example, if it took 7 days instead of 14, then $\beta = 1/2$, meaning that τ will get divided by two. As a consequence, if the hashrate doesn't change, the expected time between blocks goes up by a factor 2.

Note: to implement this mechanism, it is important that blocks include a timestamp. We will simply treat this as an exact timestamp, but is should be clear that the protocol is relatively robust against slight errors in the timestamps the nodes put on their blocks.

Exercise*

Prove that a faulty node that controls 10% of the total hashrate can break finality of a longest-chain consensus protocol if we measure the length of a chain as the number of blocks it contains (as in Chapter 5).

In a chain based on Proof-of-Work we therefore slightly modify the definition of the length of a chain: we take into account the (expected) work that went into creating each of the blocks. Concretely, this means that every block gets assigned a weight that is equal to the expected number of nonces that needed to be evaluated for it to be created: $2^{256}/\tau$. We will not do so here, but one can show that with these modifications the liveness and consistency properties from the previous chapter still hold, mostly unchanged.

6.8 Safe blocks

Above, we gave a handwavy argument stating that if messages arrive within a bounded delay of, say, 10 seconds, and the expected duration of a round was 10 minutes, then that is "as good as" messages arriving instantaneously.

In this section we make this argument more precise. Let us assume time is measured in seconds. We also assume our communication network is such that messages arrive with a delay of at most Δ seconds. Suppose we have set the difficulty parameter of our Proof-of-Work puzzles in such a way that the expected duration of a round is L seconds.

We now introduce the notion of a *safe block*. We call a block *safe* if it is produced by an honest node at some time t, and if it is the only honestly produced block in the time steps $t - \Delta + 1, t - \Delta + 2, \ldots, t$.

Lemma 6.1 (One safe block per height)**.** *The heights of safe blocks are strictly increasing over time.*

Exercise

Prove Lemma 6.1

Let β denote the fraction of the blocks that are safe.

Exercise*

Prove a lower bound on β, the fraction of the blocks that are safe, by computing
Pr(puzzle solution creates safe block) = Pr(solution found by honest node) $\times$ Pr(no puzzle solutions in last Δ timesteps).
Your answer should be a function of α, Δ, L.

Let us call a block *unsafe* if it is honestly produced, but not safe. Unsafe blocks are blocks that we cannot trust: even though they are honestly produced, the producer might not have been aware of the last previously produced honest block. So even though the unsafe block is honestly produced, it might inadvertently produce a fork.

We thus need to update our analysis. Before, we argued that as long as α, the fraction of the total hashrate that was controlled by faulty nodes, is strictly less than $1/2$, then the longest chain consensus protocol works well (i.e. we get a balanced leader sequence and therefore consistency and liveness).

We now want to consider unsafe blocks as blocks produced by faulty nodes. The inequality we want to satisfy then becomes

$$\text{fraction of safe blocks} > \text{fraction of faulty blocks} + \text{fraction of unsafe blocks}.$$

Exercise*

Prove that the above inequality holds when

$$\alpha < \frac{1 - 2\frac{\Delta}{L}}{2 - 2\frac{\Delta}{L}}.$$

Notice that if $\Delta = 0$, then the above inequality recovers the usual condition $\alpha < 1/2$.

Exercise

What fraction of the hashrate can be controlled by faulty nodes if $\Delta = 10$ and $L = 600$ (as in Bitcoin)?

Part II

Game Theory

In this part of the course, we will focus on various game-theoretic aspects that arise in blockchain protocols and related settings. We will see various ways (both cooperative and non-cooperative) of looking at these problems.

All of the materials presented in this section are based on scientific research papers and books. We will acknowledge these at the end of a chapter in a section called Acknowledgements to keep the main text free from references. Sometimes (small) changes were made compared to the results in the original papers/books for the sake of exposition. None of the presented content in this Game Theory part of the course is the author's (Pieter Kleer) own original work.

We sometimes given Python code to illustrate algorithmic concepts. These code snippets are not part of the exam material. If you have followed a course such as Computational Aspects in Econometrics (35V3A1) in the bachelor Econometrics and Operations Research, you should be able to understand all the code. You can find the course document of that course here.

Chapter 7

Proportional rule

In the previous chapter, we saw that in the Proof-of-Work protocol the leader for the next round is chosen at random with a probability proportional to a miner's (or node's) hash rate μ_i with $i = 1, \dots, n$ the indices of the participating miners. That is, the probability that miner i becomes the next leader is

$$r_i = \frac{\mu_i}{\sum_{j=1}^{n} \mu_j}.$$

Recall that the hash rate (or hashing power) of a miner is, roughly speaking, the amount of effort that a miner puts into solving the puzzle that is used to determine the leader of the next round in the Proof-of-Work protocol.

This seems like a decent idea: The more effort you put into solving the puzzle, the more likely it is that you become the next leader. Is this also the "best" thing to do, in some (for now undefined) sense?

For example, as an alternative, we might also simply select the leader uniformly at random after some miner has solved the puzzle.[1] However, this would be a bad idea, because then miners do not have any incentive to put effort into the mining process, because performing more work does not increase their chances of becoming the next leader. Furthermore, this also increases the chances of a Sybil attack, as was discussed in the previous chapter (and that we will come back to later).

In this chapter, we will see that cooperative game theory here provides the correct handles to formally prove that the probabilities r_i are indeed the optimal way to select the next leader. We will approach this formalization through the lens of allocation rules. We first elaborate on this viewpoint.

Assuming that becoming the leader of the next round yields a reward of 1 unit of cryptocurrency, e.g., Bitcoin, we may also interpret the numbers r_i as the *expected reward* of miner i. From the perspective of expected rewards, we may therefore also interpret the individual probabilities of becoming the next leader as individual rewards, and the probability distribution as an *allocation rule* that allocates every miner its share r_i. We would now like to achieve the following.

Goal (informal): Show that the *proportional allocation rule*, giving every miner its (expected) share r_i, is the only allocation rule that satisfies some desirable properties.

[1] This would only work in the permissioned setting, where we know the number of nodes/miners in the system. The statement here is only meant to be a thought experiment. In principle, there are (theoretical) protocols that could mimic any type of permissioned allocation in the permissionless setting.

To make this goal more rigorous, let us first give a formal definition of an allocation rule. For simplicity, we will assume that the hash rates are integers values (this is essentially without loss of generality). Recall that $\mathbb{N} = \{1, 2, 3, \dots, \}$ is the set of strictly positive integer numbers. We define $\mathbb{N}^* = \cup_{n=1} \mathbb{N}^n$ as the union of all possible vectors of all possible length containing integers (we will explain the need for this later on).

Definition 7.1 (Allocation rule)**.** An allocation rule is a function x that takes as input an (ordered) hash rate vector $= (\mu_1, \dots, \mu_n) \in \mathbb{N}^*$ and outputs a reward share vector $x() = (x_1(), \dots, x_n())$ that indicates for every $i = 1, \dots, n$ the share $x_i()$ they receive.

By defining the allocation rule as a function with domain $\mathbb{N}^*$, instead of $\mathbb{N}^n$ for a fixed n, we do not impose any hard restriction on the number of miners can participate. Formally speaking we could also define $\mathbb{R}^* = \cup_n \mathbb{R}^n$ and say that $x(\mu) \in \mathbb{R}^*$.

To give an example we define the *proportional (allocation) rule* x^p formally as

$$x_i^p(\mu) = \frac{\mu_i}{\sum_{j=1}^n \mu_j}$$

for $i = 1, \dots, n$ for hash rate vector $\mu \in \mathbb{N}^*$ of length n.

7.1 Axioms

We will now start to define some desirable properties, also called *axioms*, that we would like our protocol to have (as a thought exercise, before reading along, you could try to think of some properties yourself).

Perhaps the most natural property to impose is that we want an allocation rule to be *nonnegative*, meaning that a miner should never get a negative utility from investing in hashing power. Another way of looking at this, is that there can be no form of payment that the miner is charged to participate in the mining.

> (P1) **Nonnegativity**: For any $\mu \in \mathbb{N}^*$, the reward vector $x(\mu)$ is nonnegative, i.e., $x_i(\mu) \geq 0$ for every $i = 1, \dots, n$.

Furthermore, if we assume the mining process gives rise to 1 unit of currency, we have to make sure that we do not in total allocate more than 1 unit to the miners This leads to the notion of *budget-balance*. We can make a distinction between two scenarios: We allocate the mined unit completely (strong budget-balance) or partly (weak budget-balance).

> (P2a) **Weak budget-balance**: The vector $x(\mu)$ satisfies $\sum_{i=1}^n x_i(\mu) \leq 1$. (P2b) **Strong budget-balance**: The vector $x(\mu)$ satisfies $\sum_{i=1}^n x_i(\mu) = 1$.

There exist other (than Proof-of-Work) blockchain protocols whose allocation schemes could be called weakly budget-balanced, but we do not go into those here.

Moving on to a next natural property, we do not want the name (public key) of a node to have an influence on the outcome of the allocation. In our setting this roughly means that we do not want the index i of a miner to play a role in the allocation, but only its hash rate μ_i (as well as the hash rates of the other miners). This also means that miners with the same hash rate will get allocated the same amount.

One way to naturally phrase such an assumption in terms of a symmetry condition. For this we need the notion of a bijective permutation mapping π that takes as input the ordered vector of hash rates μ and returns a permutation of these rates, i.e., the numbers in the vector in a different order. We slightly abuse notation and write $\pi(x(\mu))$ for the re-ordering of the allocation vector according to the same different order.

(P3) **Symmetry**: For any hash rate vector $\mu \in \mathbb{N}^*$ and permutation π, it holds that $x(\pi(\mu)) = \pi(x(\mu))$.

This condition states that if we first interchange the hash rates of some nodes and then compute the allocation, this is the same as first computing the allocation on the original hash rate vector and then permuting the allocated amounts.

One important property of a symmetric allocation rule is that miners with the same hash rate get allocated the same amount of cryptocurrency.

> Exercise 7.1
>
> Give an allocation rule that is non-negative and strongly budget-balanced, but not symmetric.

The axioms we have stated so far are rather natural, in the sense that you might imagine they should also hold for other type of allocation problems. We will now continue with two axioms that are more tailored towards the mining environment of cryptocurrencies like Bitcoin.

In a nutshell the following two axioms ensure that a miner should not be able to benefit from splitting herself up in multiple "proxy" miners, and multiple (true) miners should not be able to benefit from posing as one miner.

Let us start with the first case, which is known as *Sybil-proofness*. This term comes from the notion of a Sybil attack, where an attacker splits himself up in multiple identities (as was discussed in the previous chapter). Intuitively, Sybil-proofness states that if one miner would split herself up in s identities, call this set of identities S, and would collect rewards from all the identities in the set S according to the allocation rule x, then the sum of these rewards should not exceed the reward that miner would have gotten when posing as her true self. In this set-up, the hash rates of all the other miners remains unchanged. More formally, we have the following definition.

(P4) **Sybil-proofness**: For every hash rate vector $\mu \in \mathbb{N}^*$ of length n and hash rate vector $\mu' \in \mathbb{N}^*$ that is derived by replacing one miner i with hash rate μ_i with a finite set S of proxy minors with hash rates μ'_j for $j \in S$ satisfying $\sum_{j \in S} \mu'_j \leq \mu_i$, and $\mu'_k = \mu_k$ for all $k \notin S$, it holds that

$$\sum_{j \in S} x_j(\mu') \leq x_i(\mu).$$

We leave one ambiguity in this definition, and that is that we do not specify the position of the hash rates of the "proxy" miners in the vector μ'. For example, if we have the hash rate vector $\mu = (\mu_1, \mu_2, \mu_3)$ and we split the second miner into four proxies $\mu_{2a}, \mu_{2b}, \mu_{2c}$ and μ_{2d}, then we can create the (obvious choice)

$$\mu' = (\mu_1, \mu_{2a}, \mu_{2b}, \mu_{2c}, \mu_{2d}, \mu_3),$$

but we might also have chosen

$$\mu' = (\mu_{2a}, \mu_1, \mu_{2b}, \mu_{2d}, \mu_3, \mu_{2c}).$$

If you would like to fix a convention you can go for the first choice of μ' above, that inserts the proxy miners at position i. However, if we assume that the allocation rule x is also symmetric, then it does not matter where we insert the proxy miners in the vector, because the allocation only depends on the hash rates then, and not the miner identities.

Exercise 7.2

Show that the uniform allocation rule, defined for a vector $\mu \in \mathbb{N}^*$ of length n by

$$x_i(\mu) = \frac{1}{n}$$

for $i = 1, \dots, n$, is *not* Sybil-proof.

We continue with the second case, known as *collusion-proofness* (collusion means "samenzwering" in Dutch). Intuitively, this axiom states that if multiple miners would pose as one miner, with a hash rate that is at most the sum of their individual hash rates, then the reward of this single miner is at most the sum of the rewards that the individual miners would have gotten. Again, the hash rate of all the other miner remains unchanged.

> (P5) **Collusion-proofness**: For every hash rate vector $\mu \in \mathbb{N}^*$ of length n and hash rate vector $\mu' \in \mathbb{N}^*$ that is obtained by replacing a set of miners T by one new miner i^* with hash rate μ'_{i^*} satisfying $\mu'_{i^*} \leq \sum_{i \in T} \mu_i$, and $\mu'_k = \mu_k$ for all $k \neq i^*$, it holds that
>
> $$x_{i^*}(\mu') \leq \sum_{i \in T} x_i(\mu).$$

The same ambiguity as in the definition of Sybil-proofness is also present here: There is some freedom in where to place i^* in the vector of miners. If you want, you can use the convention that we place i^* at the lexicographically smallest position chosen among those from miners $j \in T$. Again, if x is a symmetric allocation rule, then it does not matter where we insert (the hash rate of) i^* in the vector.

Exercise 7.3

Show that the proportional allocation rule x^p satisfies Sybil- and collusion-proofness (the arguments for both properties are similar).

At this point, you are invited to thing about other natural properties/axioms that an allocation rule of a blockchain protocol should satisfy. It turns out, however, that the properties we have defined so far are sufficient to characterize the proportional allocation rule.

7.2 Characterization

In this section, we will show that the proportional rule is the only allocation rule that satisfies strong budget-balance (P2b), symmetry (P3), Sybil-proofness (P4) and collusion-proofness (P5)! We summarize this in the following theorem.

Theorem 7.1 (Characterization of proportional rule)**.** *The proportional allocation rule x that maps a vector $\mu \in \mathbb{N}^*$ of length n to the vector $x(\mu)$ with*

$$x_i(\mu) = \frac{\mu_i}{\sum_{j=1}^{n} \mu_j}$$

is the only allocation rule that satisfies strong budget-balance (P2b), symmetry (P3), Sybil-proofness (P4) and collusion-proofness (P5).

Proof. In the previous section, we have shown (either in the text or in an exercise) that the proportional allocation rule satisfies (P2b), (P3), (P4) and (P5). It therefore remains to show that this is the *only* allocation rule that satisfies these four properties simultaneously. Therefore, assume that x is an allocation rule that satisfies (P2b), (P3), (P4) and (P5).

Recall that the vector μ only contains integers greater or equal than 1. We will prove the statement using induction on the number of entries strictly greater than 1 in the vector μ. For a given vector μ, we use t to denote this number, i.e.,

$$t = |\{i : \mu_i > 1\}|.$$

The base case is $t = 0$, i.e., μ is the all-ones vector $\mu = (1, 1, \dots, 1)$. Because of symmetry we then know that $x_i(\mu) = x_j(\mu)$ for all $i, j \in [n] = \{1, \dots, n\}$. Because of strong budget-balance we have $\sum_i x_i(\mu) = 1$, but then it must be that $x_i(\mu) = 1/n = \mu_i/(\sum_j \mu_j)$ for all $i = 1, \dots, n$, which means that x is indeed the proportional allocation rule.

For the inductive step, with respect to t, the induction hypothesis now states that x corresponds to the proportional allocation rule for all $\mu \in \mathbb{N}^*$ with at most $t - 1$ entries strictly greater than 1. We will show that the same holds for all vectors with t entries strictly greater than 1. (It is important to observe that the induction hypothesis does not fix the value of n.)

Therefore, let x be an allocation rule with precisely t entries strictly greater than 1. We assume without loss of generality (again because of symmetry) that $\mu_1, \dots, \mu_t > 1$ and $\mu_{t+1}, \dots, \mu_n = 1$. Let

$$m = \sum_{i=1}^{n} \mu_i$$

be the total hash rate of the vector μ.

We first show that for all $i \in \{1, \dots, t\} = [t]$ it holds that $x_i(\mu) = \mu_i/(\sum_j \mu_j)$, i.e., that every miner $i \in [t]$ is allocated their proportional share. We will do this using a proof by contradiction. We show that if $x_i(\mu) > \mu_i/(\sum_j \mu_j)$, then x is not collusion-proof, and, using a similar argument, that if $x_i(\mu) < \mu_i/(\sum_j \mu_j)$, then x is not Sybil-proof. We will come back to the miners with index $i \in \{t + 1, \dots, n\}$ at the end of the proof.

First suppose that $x_i(\mu) > \mu_i/(\sum_j \mu_j)$ for some $i \in [t]$. Consider the new vector μ' in which we split miner i into μ_i "proxy" miners, each with a hash rate of 1, that we denote with the set S (so that $\mu_i = |S|$). Note that μ' has length $n - 1 + |S|$. Recall that because x is symmetric, it does not matter where in the vector μ we insert these new miners. Note that μ' has at most $t - 1$ entries strictly greater than 1 (because μ_i is no longer there) and that

$$\sum_{k=1}^{n-1+|S|} \mu'_k = m.$$

The induction hypothesis now tells us that in particular all the miners in $j \in S$ (with $\mu'_j = 1$) get their proportional share, i.e., $x_j(\mu') = 1/(\sum_k \mu'_k) = 1/m$, meaning that

$$\sum_{j \in S} x_j(\mu') = \frac{|S|}{m} = \frac{\mu_i}{\sum_j \mu_j} < x_i(\mu)$$

The last inequality states that if in μ', the miners in S would collude and pose as one miner i, they can together get a higher total reward. This violates collusion-proofness (P5).

Exercise 7.4

Show that if, for $i \in [t]$, it holds that $x_i(\mu) < \mu_i/(\sum_j \mu_j)$, then x is not Sybil-proof.

From the above (including the exercise), we can conclude that $x_i(\mu) = \mu_i/(\sum_j \mu_j)$ for all $i \in [t]$. It remains to show that for $i \in \{t+1, \dots, n\}$, i.e., the miners with $\mu_i = 1$, it holds that $x_i(\mu) = 1/m$. Because all miners $i \in [t]$ get their proportional share, strong budget-balance implies that

$$\sum_{i=t+1}^{n} x_i(\mu) = \frac{n-t}{m}$$

and because of symmetry, it then must hold that $x_i(\mu) = 1/m$ for all miners $i \in \{t+1, \dots, n\}$. This completes the proof. □

The characterization in Theorem 1.1 also implies that any other allocation rule will violate at least one of the four axioms that characterize the proportional allocation rule.

Exercise 7.5

Consider the allocation rule that gives the unit reward to the miner with the highest hash rate in μ. If there are multiple miners with the highest hash rate, the unit reward is split evenly among them. Which of the axioms (P2b), (P3), (P4) and (P5) are satisfied, and which are not?

7.3 Risk aversion

Our characterization in Theorem 7.1 tells us that the proportional allocation rule is collusion-proof. Does this correspond with miner's behavior in reality? The answer (unfortunately) is no: Miners often join a *mining pool* that effectively acts as a single miner with the combined hash rate of the individual miners, i.e., miners collude. You can have a look at the biggest mining pools for Bitcoin here.

Is there a way to explain this behaviour in our axiomatic framework? After all, our characterization says that miners should have no incentive to collude. Let us have a look at the following two scenarios:

1. There are 100 individual miners each with hash rate 1.
2. There are 4 mining pools consisting each of 25 miners. Every pool has a hash rate of 25. If a pool is the first to solve the puzzle, the earned unit of cryptocurrency is split evenly (i.e., proportionally) between the 25 miners in the pool.

Exercise 7.6

Which scenario would you, as a miner, prefer and what is your (mathematical) motivation for this?

A potential answer to this questions is given next.

Let's make the scenarios a bit more formal and interpret the reward of an individual miner as random variables X_1 and X_2 for Scenarios 1 and 2, respectively. Then we have

$$X_1 = \begin{cases} 1 & \text{with probability } \frac{1}{100} \\ 0 & \text{with probability } \frac{99}{100} \end{cases} \quad \text{and} \quad X_2 = \begin{cases} \frac{1}{25} & \text{with probability } \frac{1}{4} \\ 0 & \text{with probability } \frac{3}{4} \end{cases}.$$

Observe that $\mathbb{E}[X_1] = \frac{1}{100} = \frac{1}{4} \cdot \frac{1}{25} = \mathbb{E}[X_2]$. This means that a *risk-neutral* miner, that only cares about maximizing their expectation, is indifferent between the scenarios.

However, looking at the variance we see a significant difference. We have

$$\text{Var}[X_1] = \frac{1}{100}\left(1 - \frac{1}{100}\right)^2 + \frac{99}{100}\left(0 - \frac{1}{100}\right)^2 \approx 0.01$$

and

$$\text{Var}[X_2] = \frac{1}{4}\left(\frac{1}{25} - \frac{1}{100}\right)^2 + \frac{3}{4}\left(0 - \frac{1}{100}\right)^2 \approx 0.0003,$$

so the variance in the first scenario is rouhgly 33 times higher than in the second scenario. A *risk-averse* miner who (in this example) also wants to keep the variance small in case of equal expected reward between different scenarios, will therefore have a strong preference for the second scenario.

7.3.1 Concave utility function

There are many different ways to formalize the notion of *risk-aversion*. One way of doing this is by introducing a strictly concave utility function $U : [0, 1] \to \mathbb{R}_{\geq 0}$ with $U(1) > U(0) = 0$ for the miners. The domain of this function is $[0, 1]$ to model, roughly speaking, that in total we have one unit of cryptocurrency as reward. The utility of a miner is its (random) reward evaluated in the utility function.

In theory, you could introduce a different utility function U_i for every miner i, but to keep the exposition simple, we use a common utility function. Recall that the definition of concavity is as follows.

Definition 7.2 (Concavity)**.** A function $U : [0, 1] \to \mathbb{R}_{\geq 0}$ is *concave* if for every $\lambda \in (0, 1)$ and $y, z \in [0, 1]$, it holds that

$$\lambda U(y) + (1 - \lambda)U(z) \leq U(\lambda y + (1 - \lambda z)).$$

A function U is *strictly concave* if the inequality holds with strict inequality always.

We emphasize at this point that the variance-example above cannot directly be explained by introducing a concave utility function U, but was used instead only to develop some intuition.

So then how does a concave utility function model risk aversion? For this, we can have a look at Jensen's inequality[2] which in our setting says that $\mathbb{E}[U(X)] \leq U(\mathbb{E}[X])$. Let us look at a small example, where this inequality follows directly from the definition of concavity.

Recall the random variable $\mathbb{E}[X_1]$ from above and consider the following two scenarios.

3. All miners collude and get a deterministic reward of $1/100 = \mathbb{E}[X_1]$ each.
4. The winner is chosen as a realization of X_1, i.e., every miner is chosen with probability $1/100$ and the chosen miner receives a reward of 1.

In Scenario 3, the (deterministic) utility for every miner is $U(\mathbb{E}[X_1])$ and in Scenario 4 the expected utility under U is

$$\mathbb{E}[U(X_1)] = \frac{1}{100}U(1) + \frac{99}{100}U(0) \leq U\left(\frac{1}{100} \cdot 1 + \frac{99}{100} \cdot 0\right) = U(\mathbb{E}[X_1])$$

by the concavity of U. In other words, a miner prefers Scenario 3 over 4 if their objective is to maximize their (expected) utility (in case of equality they are indifferent between the scenarios).

[2] Jensen's inequality is typically stated for convex functions. Recall that a function U is concave if and only if the function $-U$ is convex.

7.3.2 Modified axioms

To study the utility model in the context of allocation rules, we need to slightly adjust some of the axioms introduced in the first section. In this section, we go back to the probabilisitic interpretation of allocation rules. That is, for a given allocation rule x and hash rate vector μ, $x_i(\mu)$ is the probability with which miner i receives the unit reward, so that the expected utility of miner i becomes

$$x_i(\mu) \cdot U(1).$$

What are the appropriate ways of formulating the axioms of symmetry, budget-balacne, Sybil-proofness and collusion-proofness in this setting?

With the probabilistic interpretation of the allocation rule x, we can leave the definitions of symmetry and budget-balance unchanged. Symmetry states that if two miners have the same hash rate they should have the same probability of winning the unit reward, and budget-balance states that the probabilities with with the reward is allocated should sum up to 1 (strong budget-balance) or at most 1 (weak budget-balance).

The definition of Sybil-proofness only requires a conceptual change, that does not alter the mathematical definition that we gave in $(P4)$. If a miner split herself up in a set T of proxy miners, then here expected utility is the sum of the utilities derived from the individual proxy miners. This leads to the following definition of Sybil-proofness for an arbitrary allocation rule x.

> (P4-U) **Sybil-proofness**: For every hash rate vector $\mu \in \mathbb{N}^*$ of length n and hash rate vector $\mu' \in \mathbb{N}^*$ that is derived by replacing one miner i with hash rate μ_i with a finite set S of proxy minors with hash rates μ'_j for $j \in S$ satisfying $\sum_{j \in S} \mu'_j \leq \mu_i$, and $\mu'_k = \mu_k$ for all $k \notin S$, it holds that
>
> $$\sum_{j \in S} x_j(\mu') \cdot U(1) \leq x_i(\mu) \cdot U(1).$$

This definition is equivalent to $(P4)$ because $U(1) > 0$. This follows from the fact that $U(0) = 0$ and that we assume that $U(1)$ is strictly increasing.

A more significant change has to be made in the definition of collusion-proofness (P5). We want to protect ourselves against the formation of pools in which miners deterministically share the unit reward (it they win it) proportionally to their hash rates (similar to the opening example of this section). In other words, the sum of the expected utilities of the colluding miners posing as one miner i^* should be no greater that the sum of original expected utilities of these miners. This leads to the following definition for an arbitrary allocation rule x.

> (P5-U) U**-collusion-proofness**: For every hash rate vector $\mu \in \mathbb{N}^*$ of length n and hash rate vector $\mu' \in \mathbb{N}^*$ that is obtained by replacing a set of miners T by one new miner i^* with hash rate μ'_{i^*} satisfying $\mu'_{i^*} \leq \sum_{i \in T} \mu_i$, and $\mu'_k = \mu_k$ for all $k \neq i^*$, it holds that
>
> $$\sum_{i \in T} x_{i^*}(\mu') \cdot U\left(\frac{\mu_i}{\sum_{i \in T} \mu_i}\right) \leq \sum_{i \in T} x_i(\mu) \cdot U(1).$$

One implication of (P5-U) is that whenever a group of miners collude, at least one of them will have a worse utility than before the collusion.

Unfortunately, our proportional allocation rule x^p that assigns the unit reward to miner i with probability proportional to μ_i, does not satisfy U-collusion-proofness. We investigate this in the following two exercises.

Exercise 7.7

Suppose that $U(y) = \sqrt{y}$ and consider the hash rate vector $\mu = (3, 4)$. Show that the proportional allocation rule x^p that assigns the unit reward with probability $3/7$ to the first miner, and with probability $4/7$ to the second miner, does not satisfy U-collusion-proofness.

It is possible to generalize this example to any choice of strictly concave utility function U and hash rate vector $\mu \in \mathbb{N}^*$.

Exercise 7.8

Let U be any strictly concave function with $U(0) = 0$. Show that (P5-U) is not satisfied for any choice of $\mu \in \mathbb{N}^*$ when $x = x^p$. *Hint: Take T to be the set of all miners.*

7.3.3 Impossibility result

We ended the previous section with the observation that the proportional allocation rule, i.e., the allocation rule used in the Proof-of-Work protocol, does not satisfy U-collusion-proofness in the context of risk-averse miners. In fact, the assumption is violated for any strictly concave utility function and hash rate vector. Is there another allocation rule that does satisfy the adjusted notion of collision-proofness?

Yes, we could define allocation rules that satisfy the adjusted collusion-proofness definition, however, it turns out that this is not possible if we simultaneously also want to satisfy symmetry, budget-balance and Sybil-proofness!

This shows that risk-aversion is not only a problem for the proportional allocation rule, but in fact this is an inherently difficult behavior to deal with in a blockchain protocol as it cannot be addressed by adjusting the allocation rule.

We summarize the formal impossibility result in the theorem below.

Theorem 7.2 (Impossibility result)**.** *Let U be a strictly concave utility function with $U(1) > U(0) = 0$. There does not exist an allocation rule that is strongly budget-balanced (P2b), symmetric (P3), Sybil-proof (P4-U), and U-collision-proof (P5-U).*

Exercise 7.9

Prove Theorem 7.2. *Hint: First have a suitable chosen miner split into proxies, which then collude again. Derive a contradiction based on the properties of U using an argument of a similar flavor as for the last exercise of the previous section.*

7.4 Acknowledgements

This chapter is based on the the paper An Axiomatic Approach to Block Rewards by Chen, Papadimitriou and Roughgarden (2019). Our assumption (P5-U) is slightly different than the corresponding assumption in this paper. Feel free to have a look at the other results of this paper!

Chapter 8

Tullock contest

In the previous chapter we saw that (in theory) the proportional allocation rule is a good way to divide the (expected) rewards of mining a unit of cryptocurrency. One aspect that this rule did not take into account is the fact that it costs time and money to guess solutions for the puzzle contest that determines the leader of the next round in the Proof-of-Work protocol.

A more realistic way of denoting the reward, or said better, the *(expected) payoff* of miner i, is

$$\frac{\mu_i}{\sum_j \mu_j} - c_i(\mu_i)$$

where the function $c_i : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ is a nondecreasing cost function: The more hashing power miner i uses, the higher the cost $c_i(\mu_i)$ she has to pay. For example, this cost could model the energy consumption of the computers used to guess solutions with. Note that c_i is a miner-specific function.

In this chapter we will allow real values of μ_i, as opposed to only integer values, so that the payoff of miner i is continuous. For simplicity, we will assume that for every $i = 1, \dots, n$, c_i is a linear function of the form $c_i(\mu_i) = a_i \cdot \mu_i$ for $a_i > 0$, so that the payoff of miner i becomes

$$\frac{\mu_i}{\sum_j \mu_j} - a_i \cdot \mu_i.$$

In the literature, the payoff is often normalized by the value a_i, so that we finally define

$$p_i(\mu) = w_i \cdot \frac{\mu_i}{\sum_j \mu_j} - \mu_i$$

where $w_i = 1/a_i$. This will be the payoff function that we work with in this chapter.

The proportional rule had the property that a miner i could increase its reward by increasing their hash rate μ_i. If the goal of a miner is to maximize its payoff $p_i(\mu)$, it is no longer necessarily optimal for her to increase her hash rate, because $\lim_{\mu_i \to \infty} p_i(\mu) = -\infty$ (note that the other hash rates are kept constant in this reasoning).

This gives rise to a *strategic* consideration for miner i. Knowning the hash rates

$$\mu_{-i} = (\mu_1, \dots, \mu_{i-1}, \mu_{i+1}, \dots, \mu_n)$$

of the other miners, what hash rate μ_i maximizes the payoff function $p_i(\mu)$? This setting, in which every miner has the goal of maximizing their indivual payoff function, can be seen as a *non-cooperative* game. It is

important to observe that the payoff of miner i depends on the choices made by the other miners, i.e., the vector μ_{-i}!

The assumption here that we know the hash rates of all the other miners is quite strong, but not necessarily unrealistic. If we interpret the miners as mining pools, then in real-life there are good estimates of the hash rates of these pools, see for example the overview here.

In the next section we will summarize the above setting, known as a *Tullock contest*. Although we approach this problem from the perspective of blockchains, Tullock contests have many more application areas. For example, think of an election, where different candidates invest money in their campaign, but only one candidate can win in the end.

8.1 Tullock contest

A *Tullock contest* is a winner-takes-all game in which there are n miners (also called contestants or players). Every miner has *strategy space* $S_i = \mathbb{R}_{\geq 0}$ from which she chooses a *strategy* (i.e., hash rate) $\mu_i \geq 0$. The vector μ is called a *strategy profile*.

The goal of every miner is to maximizer their payoff function

$$p_i(\mu) = w_i \cdot \frac{\mu_i}{\sum_j \mu_j} - \mu_i.$$

We make one conventional assumption, which is that $p_i(0, 0, \dots, 0) = 0$, i.e., if all the hash rates are 0 then every player has a payoff of 0. Observe that the Tullock contest is completely described by its *input parameters* $w_1, \dots, w_n$.

How is this game played? There are many ways to define the *dynamics* of the game; we will consider an elementary setting. Suppose that at the start of the game, every miner i has chosen some hash rate μ_i and that all miners know the hash rate vector $\mu = (\mu_1, \dots, \mu_n)$.

We first allow miner 1 to change it hash rate μ_1 so that its payoff function $p_1(\mu)$ is maximized (while keeping the other hash rates μ_j fixed). Suppose miner 1 changes to hash rate μ_1'. Next, we allow miner 2 to change its hash rate μ_2 to maximze its payoff function $p_2(\mu_1', \mu_2, \dots, \mu_n)$. We keep repeating this process until all miners i have had the option to change their hash rate to some othe rate μ_i'. Then we go back to the first miner.

Because one or more miners might have changed their initially chosen hash rate, the payoff of miner 1 might have changed in the meantime, in which case they need to re-optimize their payoff. We keep iterating over the miners until we reach the point where no miner has an incentive to change its hash rate under the present hash rate vector, i.e., until we have reached an *equilibrium* state.

One question that should be in your mind right now is the following: Are the above-described dynamics guaranteed to converge, i.e., is it guaranteed that in a *finite number* iterations we reach an equilibrium state? How do we describe the resulting equilibrium state mathematically? These are questions we will explore in the coming sections.

As a final remark here: The most important aspect of the dynamics is that at most one miner at a time changes it hash rate. It is also possible to consider dynamics in which miners can change simulateously, but we do not allow this for now.

8.2 Pure Nash equilibrium

We start this section with a mathematical description of an equilbrium state. A unitlateral deviation of miner i to hash rate μ_i' is described by the hash rate vector

$$(\mu_i', \mu_{-i}) = (\mu_1, \dots, \mu_{i-1}, \mu_i', \mu_{i+1}, \dots, \mu_n),$$

i.e., the vector in which only miner i changes its rate, while all the other rates are kept fixed. An equilibrium, also called *pure Nash equilibrium*, is a hash rate vector in which no miner i can unilaterally deviate in order to improve her payoff. We next give the formal definition of this statement.

Definition 8.1 (Pure Nash equilibrium)**.** A hash rate vector μ is a pure Nash equilibrium (PNE) if for all $i = 1, \dots, n$, it holds that

$$p_i(\mu) \geq p_i(\mu_i', \mu_{-i})$$

for all $\mu_i' \geq 0$.

Another way to look at the inequalities in the definition of a pure Nash equilibria μ is that for every i the value μ_i maximizes the *best response function*

$$g_i(x) = p_i(x, \mu_{-i}),$$

i.e., the function in which the hash rates of all the other miners are kept fixed.

There is a small technical issue if μ_{-i} is the all-zeros vector, in which case the best response of player i is to send $x \to 0$. We will instead assume that in this case the best response of miner i is to set $\mu_i = \delta$ with $\delta > 0$ very small, such as $\delta = 10^{-10} \cdot \min_i w_i$.

> You might have seen the concept of a Nash equilibrium in an earlier game theory course. A Nash equilbrium is usually defined as a probability distribution over all possible strategies (i.e., all possible hash rates in our case). By using the term "pure", we emphasize that we do not allow randomization over strategies, but that every miner deterministically (meaning with probability 1) chooses a hash rate.

Let us look at a small example in which we verify the definition of a pure Nash equilibrium for a given vector μ. Let us take $w_1 = 4$ and $w_2 = 2$, so that

$$p_1(\mu) = p_1(\mu_1, \mu_2) = 4 \cdot \frac{\mu_1}{\mu_1 + \mu_2} - \mu_1$$

and

$$p_2(\mu) = p_2(\mu_1, \mu_2) = 2 \cdot \frac{\mu_2}{\mu_1 + \mu_2} - \mu_2$$

and recall that by convention, we have $p_1(0,0) = p_2(0,0) = 0$.

We claim that $\mu = (8/9, 4/9)$ is a pure Nash equilibrium of this game. We need to show that $\mu_1 = 8/9$ maximizes the best response function

$$g_1(x) = 4 \cdot \frac{x}{x + 4/9} - x$$

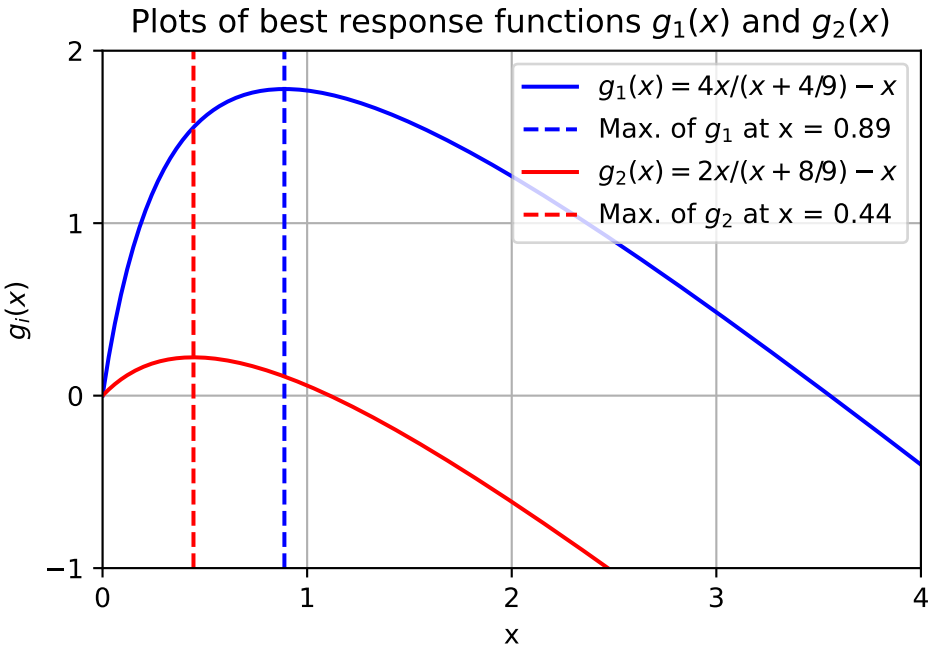
of the first miner, and that $\mu_2 = 4/9$ maximizes the best response function

$$g_2(x) = 2 \cdot \frac{x}{8/9 + x} - x$$

of the second miner. This is illustrated in the Python figure below.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize

# Define the functions g1(x) and g2(x)
def g1(x):
    return 4 * x / (x + 4 / 9) - x

def g2(x):
    return 2 * x / (x + 8 / 9) - x

# Generate x values
x = np.linspace(0, 4, 400)

# Calculate the corresponding y values for both functions
y1 = g1(x)
y2 = g2(x)

# Function for maximization of g1
def neg_g1(x):
    return -g1(x)

# Function for maximization of g2
def neg_g2(x):
    return -g2(x)

# Minimize the negative of the functions to find the maximum
result_g1 = minimize(neg_g1, 1)
result_g2 = minimize(neg_g2, 1)

# Extract the x values at the maxima
x_max_g1 = result_g1.x[0]
x_max_g2 = result_g2.x[0]

# Calculate the corresponding y-values at the maxima
y_max_g1 = g1(x_max_g1)
y_max_g2 = g2(x_max_g2)

# Create the plot
plt.figure()

# Plot functions and maxima
plt.plot(x, y1, label=r"$g_1(x) = 4x/(x + 4/9) - x$", color='blue')
plt.axvline(x_max_g1, color='blue', linestyle='--',
                label=f"Max. of $g_1$ at x = {x_max_g1:.2f}")
```

```
plt.plot(x, y2, label=r"$g_2(x) =  2x/(x + 8/9) - x$", color='red')
plt.axvline(x_max_g2, color='red', linestyle='--',
                label=f"Max. of $g_2$ at x = {x_max_g2:.2f}")

# Add titles and labels
plt.title("Plots of best response functions $g_1(x)$ and $g_2(x)$")
plt.xlabel("x")
plt.ylabel(f"$g_i(x)$")
plt.xlim(0,4)
plt.ylim(-1,2)
plt.legend()
plt.xticks([0,1,2,3,4])
plt.yticks([-1,0,1,2])

# Show the plot
plt.grid(True)
plt.show()
```



We can formally prove this with some elementary calculus,[1] which we will do next for the first miner. We have to show that $x = 8/9$ maximizes the best response function g_1. Note that we want to compute the maximum of the function $g_1(x)$ on the interval $[0, \infty)$. Observe that $g_1(0) = 0$ and $\lim_{x\to\infty} g_1(x) = -\infty$. Furthermore, g_1 is strictly concave on $[0, \infty)$, as the second derivative $g_1''(x) < 0$ there. This means that g_1 has an internal maximum on $(0, \infty)$ if the equation $g_1'(x) = 0$ has a solution on $[0, \infty)$, i.e., if the first derivative is 0 for a point in that interval. Otherwise, g_1 is decreasing in which case the maximum is attained in $x = 0$ and $g_1'(0) \leq 0$. Show yourself that $x = 8/9$ is indeed a solution to $g_1'(x) = 0$.

The above example shows that for a given μ we can show with elementary calculus whether or not it satisfies

[1]There are many ways of doing this. You can use your own method as long as you add the proper argumentation for it.

the definition of a pure Nash equilibrium (convince yourself that this approach also works for games with more than two miners).

Exercise 8.1

Consider a game with three miners whose payoff functions are

$$p_i(\mu_1, \mu_2, \mu_3) = w_i \cdot \frac{\mu_i}{\mu_1 + \mu_2 + \mu_3} - \mu_i$$

for $i = 1, 2, 3$ with $w_1 = 4, w_2 = 2$ and $w_3 = 0.5$. Show that $\mu = (8/9, 4/9, 0)$ is a pure Nash equilibrium of this game (part of your solution should be to formalize the calculations above).

8.2.1 Computation for $n = 2$

To determine a pure Nash equilibrium of the game, we do not want to keep guessing solutions by checking the definition. Instead we want a constructive way to compute an equilibrium given the payoff functions of the miners. For a game with two miners, this is the purpose of the next exercise.

The reasoning in the example of the previous section can be very helpful to successfully solve the coming exercise, so make sure you understand that example.

Exercise 8.2

Let $w_1 \geq w_2 > 0$ and consider the payoff functions

$$p_1(\mu) = w_1 \cdot \frac{\mu_1}{\mu_1 + \mu_2} - \mu_1 \text{ and } p_2(\mu) = w_2 \cdot \frac{\mu_2}{\mu_1 + \mu_2} - \mu_2.$$

Compute the (unique) pure Nash equilibrium $\mu = (\mu_1, \mu_2)$ of this Tullock contest by deriving explicit formulas for $\mu_1 = \mu_1(w_1, w_2)$ and $\mu_2 = \mu_2(w_1, w_2)$, i.e., the equilibrium hash rates in terms of w_1 and w_2. You can use the following steps for this:

1. Show that if one or both of the μ_i equal zero, then $\mu = (\mu_1, \mu_2)$ cannot be a pure Nash equilibrium. *Recall that a best response for miner i is to choose a (small) hash rate of $\delta > 0$ when all other miners have hash rate 0 (see remark under PNE definition).*
2. Argue next that μ is a pure Nash equilibrium if and only if the following *first-order conditions* are satisfied:
$$\frac{\partial p_1}{\partial \mu_1}(\mu) = 0 \text{ and } \frac{\partial p_2}{\partial \mu_2}(\mu) = 0.$$
3. Solve the first-order conditions for μ_1 and μ_2 in terms of w_1 and w_2.

8.2.2 Computation for $n > 2$

The approach above can be extended to games with more than two miners. There even exists a closed-form solution for the (unique) pure Nash equilibrium. We do not go into this exact closed-form solution, but we give a short sketch of how to obtain the equilibrium for n miners. We will assume that $w_1 \geq w_2 \geq \cdots \geq v_n$. You can do this without loss of generality by renumbering the miners if needed.

First, it can be argued (just as we did in the exercise above) that in a pure Nash equilibriu μ at least two entries must be strictly positive. This implies that for every i, the quantity $\sum_{j \neq i} \mu_j > 0$ (do you understand why)?

Second, the pure Nash equilibrium μ must satisfy the first order conditions

$$\begin{cases} \frac{\partial p_i}{\partial \mu_i}(\mu) = 0 & \text{if } \mu_i > 0 \\ \frac{\partial p_i}{\partial \mu_i}(0) \leq 0 & \text{if } \mu_i = 0 \end{cases}$$

That is, if $\mu_i > 0$ then it is an internal maximizer of the best response function g_i, and if $\mu_i = 0$, then it must be that the best response function is decreasing, which is equivalent to the derivative in zero being nonnegative; the concavity of g_i then implies that the function is decreasing on $[0, \infty)$.

The problem now is that we do not know *a priori* which μ_i's will be zero, and which will be strictly great than zero. It turns out, though, that there is some nice structure in this regard: If $w_i \geq w_j$ and $\mu_i = 0$, then also $\mu_j = 0$ (this is a somewhat tricky calculus exercise). Recall that we assumed that $w_1 \geq w_2 \geq \cdots \geq w_n$. This means that in the pure Nash equilibrium, if one or more miners have $\mu_i = 0$, then there exists an index $\hat{n}$ such that all zeros in the equilibrium are attained for indices $i = \hat{n} + 1, \dots, n$, i.e., the equilibrium is of the form

$$\mu = (\mu_1, \dots, \mu_{\hat{n}}, 0, 0, \dots, 0).$$

There exists an explicit formula known for $\hat{n}$, namely

$$\hat{n} = \max \left\{ i \in [n] : w_i > \frac{i-2}{\sum_{j=1}^{i-1} \frac{1}{w_j}} \right\}. \tag{8.1}$$

In other words, $\hat{n}$ is the largest index i for which the inequality $w_i > \frac{i-2}{\sum_{j=1}^{i-1} \frac{1}{w_j}}$ holds. We use the convention that $\sum_{j=1}^{0} \frac{1}{w_j} = 1$.

Note that $\hat{n}$ can be computed using only the input parameters $w_1, \dots, w_n$ of the Tullock contest. Also observe that we always have $\hat{n} \geq 2$, because we assume the w_i to be strictly positive. This correspond with the fact that in a pure Nash equilibrium, there will always be at least two miners with a strictly positive hash rate. Proving the correctness of the formula for $\hat{n}$ requires some nontrivial arguments, that we omit here.

The remaining (strictly positive) values of $\mu_1, \dots, \mu_{\hat{n}}$ can be determined by solving the system of equations

$$\frac{\partial p_i}{\partial \mu_i}(\mu) = 0 \text{ for } i = 1, \dots, \hat{n} \tag{8.2}$$

with $\mu_{\hat{n}+1} = \cdots = \mu_n = 0$ substituted in these equations. This leaves a (non-linear) system of $\hat{n}$ equalities with $\hat{n}$ unknown variables.

"Solving" this system means that we would like to express the μ_i as a function of the input parameters $w_1, \dots, w_n$, just as we did in the exercise earlier in this section with $n = 2$.

Exercise 8.3

This exercise is dedicated to solving the system of equations in Equation 8.2 to determine the strictly positive μ_i in the pure Nash equilibrium.

1. Show that the system in Equation 8.2 is equivalent to the equations

$$\mu_i = \left(\sum_{j=1}^{\hat{n}} \mu_j \right) - \frac{1}{w_i} \left(\sum_{j=1}^{\hat{n}} \mu_j \right)^2 \quad \text{for } i = 1, \dots, \hat{n}$$

2. Define $\lambda = \sum_{j=1}^{\hat{n}} \mu_j$. Determine an equation for λ by adding up all the equations of part 1.
3. Solve the resulting equation for λ to show that

$$\lambda = \lambda(w_1, \dots, w_n) = \frac{\hat{n} - 1}{\sum_{j=1}^{\hat{n}} \frac{1}{w_j}}.$$

Note that we can then finally express the μ_i as a function of $w_1, \dots, w_n$ (through λ) by

$$\mu_i = \lambda - \frac{1}{w_i}\lambda^2$$

for $i = 1, \dots, \hat{n}$.

Exercise 8.4

Consider the Tullock contest with $w_1 = 4, w_2 = 2, w_3 = 0.5$ and $w_4 = 0.2$. Compute the value of $\hat{n}$ and use the formulas of the previous exercise to show that $\mu = (8/9, 4/9, 0, 0)$ is the pure Nash equilibrium of this game (i.e., compute λ and then μ_1 and μ_2 using λ).

To summarize, we have the following theorem.

Theorem 8.1 (Equilibrium computation with n miners.)**.** *Let* $w_1 \geq \cdots \geq w_n > 0$ *and*

$$\hat{n} = \max \left\{ i \in [n] : w_i > \frac{i - 2}{\sum_{j=1}^{i-1} \frac{1}{w_j}} \right\}.$$

Then the (unique) pure Nash equilibrium $\mu = (\mu_1, \dots, \mu_n)$ *of the Tullock contest is given by*

$$\mu_i = \begin{cases} \lambda - \frac{1}{w_i}\lambda^2 & \text{for } i \leq \hat{n} \\ 0 & \text{for } i > \hat{n} \end{cases} \quad \text{with} \quad \lambda = \frac{\hat{n} - 1}{\sum_{j=1}^{\hat{n}} \frac{1}{w_j}}.$$

Exercise 8.5

Consider the Tullock contest with $w_1 = w_2 = \cdots = w_n = w$. Use Theorem 8.1 to show that its pure Nash equilibrium is given by $\mu = (\mu_1, \dots, \mu_n)$ with

$$\mu_i = \frac{n - 1}{n^2} w$$

for $i = 1, \dots, n$. This is called a *symmetric* pure Nash equilibrium, because all μ_i are the same.

Exercise 8.6

We will derive a concise expression for the market share $x_i(\mu) = \mu_i / (\sum_j \mu_j)$ of miner i in the pure Nash equilibrium.

1. Show that $x_i(\mu) = 1 - \frac{1}{w_i}\lambda$ for $i = 1, \dots, \hat{n}$ with λ as in Theorem 8.1.
2. Based on part 1., show that

$$x_i(\mu) = \max\left\{0, 1 - \frac{1}{w_i}\lambda\right\}$$

for $i = 1, \dots, n$.

8.3 Best response dynamics

We conclude this chapter with some remarks about best response dynamics. Formally, such dynamics are defined as an iterative process where in every iteration, given a hash rate profile $\mu = (\mu_1, \dots, \mu_n)$ one miner i gets to compute their best response against μ_{-i}, i.e., maximize their best response function $g_i(x) = p_i(x, \mu_{-i})$. We keep repeating this procedure until no miner wants to change their hash rate anymore.

By solving the first order conditions, and recalling that the best response to the all-zeros vector $\mu_{-i} = (0, \dots, 0)$ is a small number δ, we can express the best response of miner i given μ_{-i} as

$$\text{BR}_i(\mu_{-i}) = \begin{cases} \delta & \text{if } \sum_{j\neq i} \mu_j = 0 \\ \sqrt{w_i(\sum_{j\neq i} \mu_j)} - \sum_{j\neq i} \mu_j & \text{if } 0 < \sum_{j\neq i} \mu_j < w_i \\ 0 & \text{if } \sum_{j\neq i} \mu_j \geq w_i \end{cases}$$

Exercise 8.7

Verify the correctness of the formula for BR_i above.

We will assume the iterative process is done in a *round-robin* fashion, that is, we iterate over miners $1, \dots, n$ and then start at miner 1 again. More precisely, the miner updating their best response in round t is $i = t \bmod n$. The pseudo-code of best response dynamics is therefore as follows.

```
Best response dynamics

1.  Initialize
    - Weight vector w = [w_1,..,w_n]
    - Initial hash rate vector mu = [mu_1,...,mu_n]
    - Number of rounds T
    - Parameter delta (best response against all-zeros vector).
2.  In each round t = 1,2,3,...,T do:
    - Compute best response of miner i = t (mod n)
    - Update value mu_i with the computed best response
```

A Python implementation of this procedure with $n = 4$ miners is given below with

- $w = (5, 2, 0.5, 0.2)$
- $\mu = (1, 1, 1, 1)$
- $T = 40$
- $\delta = 1/(w_{\text{max}} \cdot n^3) \approx 0.0039$ (any small enough number is allowed here).

```
import numpy as np
import pandas as pd

# Weight vector
w = np.array([4, 2, 0.5, 0.2])

# Number of miners
n = np.size(w)

# Initial hash rate vector
mu = np.array([1.0, 1.0, 1.0, 1.0])

# Number of best response iterations
T = 40

# Value of delta (brd against all-zeros vector)
delta = 1/(np.max(w)*n**3)

# Function that computes best response for miner i given hash rate vector mu
def best_response(i, mu, wi, delta):
    rate_minus_i = np.sum(mu) - mu[i]
    if np.allclose(rate_minus_i, 0):  # First BRD case
        br = delta
    elif (rate_minus_i > 0) & (rate_minus_i <= wi):  # Second BRD case
        br = np.sqrt(wi * rate_minus_i) - rate_minus_i
    elif rate_minus_i > wi:  # Third BRD case
        br = 0
    return br

# Executing best response dynamics procedure
def best_response_dynamics(w,mu,T,delta):
    # Create an empty list to store the results
    mu_history = []

    # Append initial hash rate vector to results list
    mu_history.append(mu.copy())

    for t in range(T):
        i = np.mod(t, n)
        mu[i] = best_response(i, mu, w[i], delta)
        mu_history.append(mu.copy())  # Append hash rate vector to results list

    # Convert the mu_history list to a pandas DataFrame
    mu_df = pd.DataFrame(mu_history, columns=[f'mu_{i+1}' for i in range(n)])

    # Add an "Iteration" column for the index
```

```
    mu_df.insert(0, 'Iteration', np.arange(T + 1))

    # Display the DataFrame without the index column
    results = mu_df.to_string(index=False)

    return results

brd = best_response_dynamics(w,mu,T,delta)
print(brd)
```

```
 Iteration     mu_1     mu_2     mu_3  mu_4
         0 1.000000 1.000000 1.000000   1.0
         1 0.464102 1.000000 1.000000   1.0
         2 0.464102 0.000000 1.000000   1.0
         3 0.464102 0.000000 0.000000   1.0
         4 0.464102 0.000000 0.000000   0.0
         5 0.003906 0.000000 0.000000   0.0
         6 0.003906 0.084482 0.000000   0.0
         7 0.003906 0.084482 0.121836   0.0
         8 0.003906 0.084482 0.121836   0.0
         9 0.702127 0.084482 0.121836   0.0
        10 0.702127 0.459753 0.121836   0.0
        11 0.702127 0.459753 0.000000   0.0
        12 0.702127 0.459753 0.000000   0.0
        13 0.896349 0.459753 0.000000   0.0
        14 0.896349 0.442568 0.000000   0.0
        15 0.896349 0.442568 0.000000   0.0
        16 0.896349 0.442568 0.000000   0.0
        17 0.887948 0.442568 0.000000   0.0
        18 0.887948 0.444680 0.000000   0.0
        19 0.887948 0.444680 0.000000   0.0
        20 0.887948 0.444680 0.000000   0.0
        21 0.889006 0.444680 0.000000   0.0
        22 0.889006 0.444415 0.000000   0.0
        23 0.889006 0.444415 0.000000   0.0
        24 0.889006 0.444415 0.000000   0.0
        25 0.888874 0.444415 0.000000   0.0
        26 0.888874 0.444448 0.000000   0.0
        27 0.888874 0.444448 0.000000   0.0
        28 0.888874 0.444448 0.000000   0.0
        29 0.888891 0.444448 0.000000   0.0
        30 0.888891 0.444444 0.000000   0.0
        31 0.888891 0.444444 0.000000   0.0
        32 0.888891 0.444444 0.000000   0.0
        33 0.888889 0.444444 0.000000   0.0
        34 0.888889 0.444445 0.000000   0.0
        35 0.888889 0.444445 0.000000   0.0
```

```
36 0.888889 0.444445 0.000000   0.0
37 0.888889 0.444445 0.000000   0.0
38 0.888889 0.444444 0.000000   0.0
39 0.888889 0.444444 0.000000   0.0
40 0.888889 0.444444 0.000000   0.0
```

In the output above we can see that the vector μ seems to converge to $(8/9, 4/9, 0, 0)$ which was indeed the PNE that we saw earlier in this chapter for this specific choice of weight vector w (see Exercise 8.4). It should be noted though, that we never *exactly* reach this vector μ. The dynamics converges to the PNE over time, but does not attain it in a finite number of iterations.

To deal with this, we can look at a relaxed equilibrium notion, called the *ϵ-approximate pure Nash equilibrium (ϵ-PNE)*. Intuitively, if μ is an ϵ-PNE then some miners might have an incentive to deviate, but the improvement in payoff that this will give them is at most an additive term ϵ. Note that a pure Nash equilibrium is an ϵ-PNE for $\epsilon = 0$.

Definition 8.2 (ϵ-approximate Pure Nash equilibrium)**.** A hash rate vector μ is an ϵ-approximate pure Nash equilibrium (PNE) if for all $i = 1, \dots, n$, it holds that

$$p_i(\mu) \geq p_i(\mu_i', \mu_{-i}) - \epsilon$$

for all $\mu_i' \geq 0$.

Exercise 8.8

Consider a Tullock contest with two miners with weights $w_1 = 3, w_2 = 1$. In the computations below you may round any number to two decimals in your calculation.

1. Consider the hash rate vector $\mu = (1, 1)$. Show that μ is not an ϵ-PNE for $\epsilon = 0.01$.
2. What is the smallest ϵ for which μ is an ϵ-PNE?

We can now instead run the dynamics until an ϵ-PNE is found. The updated pseudo-code is given below.

```
Best response dynamics for computing epsilon-PNE

1.  Initialize
    - Weight vector w = [w_1,..,w_n]
    - Initial hash rate vector mu = [mu_1,...,mu_n]
    - Number of rounds T
    - Parameter delta (best response against all-zeros vector).
    - Parameter epsilon
2.  While mu is not an epsilon-PNE, do:
    - Compute best response of miner i = t (mod n)
    - Update value mu_i with the computed best response
```

In general, best response dynamics are not guaranteed to converge to an (ϵ-)PNE. This means that the while-loop in the pseudo-code above is not guaranteed to terminate. Let us look at such an example with two miners and $\epsilon = 0$.

```
# Weight vector
w = np.array([0.1, 1])
```

```
# Number of miners
n = np.size(w)

# Number of best response iterations
T = 12

# Value of delta (best reponse against all-zeros vector)
delta = 10**(-5)

# Initial hash rate vector
mu = np.array([0,delta])

# Function best_response_dynamics() is defined in previous snippet
brd = best_response_dynamics(w,mu,T,delta)
print(brd)
```

```
 Iteration     mu_1     mu_2
         0 0.000000 0.000010
         1 0.000990 0.000010
         2 0.000990 0.030474
         3 0.024729 0.030474
         4 0.024729 0.132526
         5 0.000000 0.132526
         6 0.000000 0.000010
         7 0.000990 0.000010
         8 0.000990 0.030474
         9 0.024729 0.030474
        10 0.024729 0.132526
        11 0.000000 0.132526
        12 0.000000 0.000010
```

After $t = 6$ iterations we are back at the hash rate vector that we started with! And so, the best response dynamics keeps cycling through the first six hash rate vectors in the above list without ever approaching the PNE of the game.

This means we have to reflect a bit on what we did before. We studied the pure Nash equilibrium as "natural" outcome of the best response dynamics process, but the dynamics is not guaranteed to end up there! Despite the existence of examples where the best response dynamics does not converge, it does converge in many cases, though.

One class where best response dynamics are guaranteed to converge is when the miners have a common value $w_1 = \cdots = w_n = w$ as in the symmetric setting we considered in Exercise 8.6. For simplicity, we will set $w = 1$. The common payoff function of all miners $i = 1, \dots, n$ is then

$$p_i(\mu) = \frac{\mu_i}{\sum_j \mu_j} - \mu_i.$$

8.3.1 Potential function

We can use a *(best response) potential function argument* to show that best response dynamics always converge under equal payoff functions. The idea of such an argument is to construct a (bounded from above) function $P : \mathbb{R}^n_{\geq 0} \to \mathbb{R}$ that takes as input a hash rate vector $\mu = (\mu_1, \dots, \mu_n)$ and outputs a real number. Bounded from above here means that there exists a constant $c \in \mathbb{R}$ such that $P(\mu) \leq c$ for all $\mu \in \mathbb{R}^n_{\geq 0}$.

The function P will be constructed in such a way that if a miner i makes a best response move in $\mu = (\mu_1, \dots \mu_{i-1}, \mu_i, \mu_{i+1}, \mu_n) = (\mu_i, \mu_{-i})$ to $\mu' = (\mu_1, \dots \mu_{i-1}, \mu'_i, \mu_{i+1}, \mu_n) = (\mu'_i, \mu_{-i})$, i.e, miner i unilaterally deviates from μ_i to μ'_i, then the value of $P(\mu')$ will be higher than $P(\mu)$. In other words, whenever a miner makes a best response move, the value of the potential function goes up.

Because P is bounded from above, the intuition is that if we perform enough best response rounds, and increase the potential function value in every round a little bit, then we must get close to an (approximate) pure Nash equilibrium in a finite number of steps, because the potential function is bounded from above and so it cannot increase indefinitely. There is a small caveat in this reasoning, which is that the increments might get smaller and smaller (as we saw in one of the examples above as well), so we need to guarantee that enough "progress" is made in every round. We will get back to this last point later on.

For now, let us look at a potential function that has the property that its value increases whenever a miner makes a best response move. To find such a function, you typically have to make an educated guess and show that it does what it needs to do. In our setting, we can define

$$P(\mu) = -\frac{1}{3}\left(\sum_{i=1}^{n} \mu_i\right)^3 + \sum_{i<j} \mu_i \mu_j$$

as the potential function. As an example, for $n = 3$, we have

$$P(\mu) = -\frac{1}{3}\left(\mu_1 + \mu_2 + \mu_3\right)^3 + \mu_1\mu_2 + \mu_1\mu_3 + \mu_2\mu_3.$$

We have to address one technicality with respect to the domain of the function P. Recall that, to avoid limit arguments, we define the best response of a miner i to be a small (fixed) number δ whenever the hash rates of all the other miners are zero in μ. To deal with this appropriately in the potential function P, we formally define P as a function $P : \mathbb{R}^n_{\geq 0} \setminus (\cup_{i=1}^n Z_i) \to \mathbb{R}$ where for $i = 1, \dots, n$,

$$Z_i = \{(0, \dots, 0, \mu_i, 0, \dots, 0) : 0 \leq \mu_i < \delta\}.$$

That is, we do not allow hash rate vectors μ with one non-zero entry that is smaller than δ.

We first show that the function P is indeed bounded from above.

> Exercise 8.9
>
> Show that $P(\mu) \leq \frac{4}{3}$ for all $\mu \in \mathbb{R}^n_{\geq 0}$. *Hint: Use that* $\sum_{i<j} x_i x_j \leq (\sum_i x_i)^2$ *and then the substitution* $y = \sum_i x_i$.

In fact, we can show a stronger result, namely that P has a global maximizer. Recall that in Exercise 8.5, we showed that the PNE μ of a symmetric Tullock contest is given by $\mu_i = (n-1)/n^2$ for $i = 1, \dots, n$ if $w = 1$. It can be shown that this μ is the global maximizer of P over its formal domain, as well as over $\mathbb{R}^n_{\geq 0}$.

Exercise 8.10

For $n = 2$, show that $\mu = (\frac{1}{4}, \frac{1}{4})$ is the global maximizer of

$$P(\mu) = -\frac{1}{3}(\mu_1 + \mu_2)^3 + \mu_1\mu_2$$

on $\mathbb{R}^2$ by executing the following steps (that together analyse P on the whole of $\mathbb{R}^2$):

1. Show that $P(\mu) \leq 0$ for all $\mu \in \{(\mu_1, \mu_2) : \mu_1 + \mu_2 > 3, \mu_1, \mu_2 \geq 0\}$.
2. Show that $P(\mu) \leq 0$ on the boundary of the set

$$D = \{(\mu_1, \mu_2) : \mu_1 + \mu_2 \leq 3, \mu_1, \mu_2 \geq 0\}.$$

3. Show that in the interior of D, we have

$$\nabla P = \left(\frac{\partial P}{\partial \mu_1}, \frac{\partial P}{\partial \mu_2}\right) = (0, 0)$$

if and only if $\mu = (\frac{1}{4}, \frac{1}{4})$.

Conclude from the above three steps that $\mu = (\frac{1}{4}, \frac{1}{4})$ is indeed the global maximizer of P.

With a bit more effort, the above exercise can be generalized to an arbitrary value of n (feel free to try this yourself).

Now comes the most important part: We want to show that if a miner makes a best response, then the potential function increases. This is summarized in the following theorem, that states that doing a single-variable optimization of P over μ_i yields the same optimal point as computing a best response of miner i.

Theorem 8.2 (Best response potential function)**.** *Let $w_1 = \cdots = w_n = 1$ so that the common payoff function of miners $i = 1, \ldots, n$ is given by*

$$p_i(\mu) = \frac{\mu_i}{\sum_j \mu_j} - \mu_i.$$

Then for any $\mu \in \mathbb{R}^n_{\geq 0} \setminus (\cup_{i=1}^n Z_i)$, it holds that

$$\operatorname{argmax}_{x \geq 0} P(x, \mu_{-i}) = \operatorname{argmax}_{x \geq 0} p_i(x, \mu_{-i}).$$

Exercise 8.11

Show that for a given fixed μ_{-i}, it holds that

$$\operatorname{argmax}_{x \geq 0} P(x, \mu_{-i}) = \begin{cases} \delta & \text{if } \sum_{j \neq i} \mu_j = 0 \\ \sqrt{\sum_{j \neq i} \mu_j} - \sum_{j \neq i} \mu_j & \text{if } 0 < \sum_{j \neq i} \mu_j < 1 \\ 0 & \text{if } \sum_{j \neq i} \mu_j \geq 1 \end{cases}$$

which shows that $\operatorname{argmax}_{x \geq 0} P(x, \mu_{-i})$ is the same function as $\mathrm{BR}_i(\mu_{-i}) = \operatorname{argmax}_{x \geq 0} p_i(x, \mu_{-i})$, and thereby proves Theorem 8.2.

Let us finally illustrate that the function P indeed increases by considering a small example of best response dynamics with initial hash rate vector $\mu = (2, 3)$. Recall that $w_1 = w_2 = 1$.

```
import numpy as np
import pandas as pd

# Weight vector
w = np.array([1, 1])

# Number of miners
n = np.size(w)

# Initial hash rate vector
mu = np.array([2.0,3.0])

# Number of best response iterations
T = 5

# Value of delta (brd against all-zeros vector)
delta = 1/(np.max(w)*n**3)

# Function that computes best response for miner i
def best_response(i, mu, wi, delta):
    rate_minus_i = np.sum(mu) - mu[i]
    if np.allclose(rate_minus_i, 0):
        br = delta
    elif (rate_minus_i > 0) & (rate_minus_i <= wi):
        br = np.sqrt(wi * rate_minus_i) - rate_minus_i
    elif rate_minus_i > wi:
        br = 0
    return br

# Function to compute P(mu)
def compute_P(mu):
    return -1/3 * (mu[0] + mu[1])**3 + mu[0] * mu[1]

# Executing best response dynamics procedure
def best_response_dynamics(w, mu, T, delta):
    # Create an empty list to store the results
    mu_history = []
    P_history = []

    # Append initial hash rate vector to results list
    mu_history.append(mu.copy())
    P_history.append(compute_P(mu))

    for t in range(T):
        i = np.mod(t, n)
        mu[i] = best_response(i, mu, w[i], delta)
```

```
            mu_history.append(mu.copy())
            P_history.append(compute_P(mu))

        # Convert the mu_history and P_history lists to a pandas DataFrame
        mu_df = pd.DataFrame(mu_history,
                             columns=[f'mu_{i+1}' for i in range(n)])

        # Add an "Iteration" column for the index
        mu_df.insert(0, 'Iteration', np.arange(T + 1))

        # Add P(mu) as an extra column
        mu_df['P(mu)'] = P_history

        # Display the DataFrame without the index column
        results = mu_df.to_string(index=False)

        return results

    brd = best_response_dynamics(w, mu, T, delta)
    print(brd)
```

```
 Iteration     mu_1     mu_2      P(mu)
         0 2.000000 3.000000 -35.666667
         1 0.000000 3.000000  -9.000000
         2 0.000000 0.125000  -0.000651
         3 0.228553 0.125000   0.013838
         4 0.228553 0.249519   0.020607
         5 0.250000 0.249519   0.020833
```

8.3.2 Convergence to ϵ-PNE

Now that we now that the potential function value increases in ever round of the best response dynamics, what is left is to argue that the dynamics reaches a hash rate vector that comes close to the global maximizer of P, which is the PNE of the Tullock contest.

This turns out to be a highly nontrivial task, but some results are known in the literature on Tullock contests. Here we (informally) mention one such a result.

Theorem 8.3 (Ghosh and Golberg, 2023)**.** *Consider an initial hash rate vector $\bar{\mu}$ and let $\epsilon > 0$. Then best response dynamics for $n = 2$ miners with $w_1 = w_2 = 1$ reaches an ϵ-PNE in at most*

$$T = \log_2\left(\log_2\left(\frac{1}{\epsilon}\right)\right) + C_{\bar{\mu}}$$

rounds, where $C_{\bar{\mu}}$ is a constant number depending on $\bar{\mu}$, but independent of ϵ.

Note that this means the convergence goes quite quickly with respect to ϵ. For example, for $\epsilon = 10^{-10}$ we have $\log_2(\log_2(1/\epsilon)) \approx 5$. The proof of Ghosh and Golberg (2023) makes use of the best response potential function P that we introduced, but the analysis is very involved and beyond the scope of this course.

8.4 Two model variations

In this section we will study extensions of the basic Tullock contest, in which the payoff functions are of the form

$$p_i(\mu) = u_i\left(\frac{\mu_i}{\sum_j \mu_j}\right) - \mu_i$$

where the $u_i : \mathbb{R}_{\geq 0} \to \mathbb{R}$ are given functions for $i = 1, \ldots, n$.

We will first consider the setting where the u_i are concave functions, modeling risk averse miners as in Section 7.3. We will also consider a case corresponding to a so-called *economy of scale*. Here we provide some evidence why there tend to be only a couple of "big players" in the mining contest.

8.4.1 Concave utilities

In this section we assume that the utility function of miner i is given by

$$p_i(\mu) = u_i\left(\frac{\mu_i}{\sum_j \mu_j}\right) - \mu_i$$

with u_i a strictly increasing, concave function with $u_i(0) = 0$ for $i = 1, \ldots, n$.

Also for these payoff functions, we make the assumption that a best response of miner i against the all-zeros hash rate vector is a sufficiently small number $\delta > 0$. For our purposes below, it will be sufficient to choose $\delta = 10^{-1} \cdot \min_i u_i(\frac{1}{2})$. Note that $\delta > 0$ because we assume $u_i(0) = 0$ and u_i to be strictly increasing.

Tullock contests of this form still have a pure Nash equilibrium, but it can no longer be explicitly computed as in Theorem 8.1, nor is a potential function known (whose maximizer is the pure Nash equilibrium) that can be used to analyze best response dynamics for this setting.

Nevertheless, it turns out we can compute a PNE by solving a (concave) maximization problem![2] To be more precise, if we define

$$x_i = \frac{\mu_i}{\sum_j \mu_j}$$

as the market share (or winning probability) of miner i, we can determine the optimal market shares by solving an optimization problem. Because the problem turns out to be a concave maximization problem, there exist many efficient optimization algorithms that can solve this problem quickly!

From the solution we can also obtain the individual hash rates μ_i for $i = 1, \ldots, n$. We will discuss the second point after showing the main result for the market shares.

The (concave) maximization problem for determining the shares x_i is given in the theorem below. We will prove its correctness by showing that the first-order conditions of a PNE correspond to the KKT conditions (see Section 2.4) of the optimization problem.

Theorem 8.4 (Computing pure Nash equilibrium under concave utilities)**.** *Suppose that the payoff of miner $i = 1, \ldots, n$ is given by*

$$p_i(\mu) = u_i\left(\frac{\mu_i}{\sum_j \mu_j}\right) - \mu_i$$

[2]The PNE is in fact unique, but we do not focus on this aspect here.

with u_i a strictly increasing, concave function with $u_i(0) = 0$ for $i = 1, \dots, n$. Then the market shares $x_i = \frac{\mu_i}{\sum_j \mu_j}$ of a pure Nash equilibrium of the resulting Tullock contest can be computed by solving the concave maximization problem

$$\begin{aligned} \max_{x \in \mathbb{R}^n} \quad & \sum_{i=1}^{n} \hat{u}_i(x_i) \\ \textit{subject to} \quad & \sum_{i} x_i = 1 \\ & x_i \geq 0 \qquad \textit{for } i = 1, \dots, n \end{aligned}$$

where for $i = 1, \dots, n$ we have

$$\hat{u}_i(x_i) = (1 - x_i)u_i(x_i) + \int_0^{x_i} u_i(z)\mathrm{d}z.$$

> **Exercise 8.12**
>
> Show that $\hat{u}_i(x_i) = w_i x_i \left(1 - \frac{1}{2}x_i\right)$ for $u_i(z) = w_i z$.

Proof of Theorem 8.4. We first argue that if μ is a PNE, then at least two miners have a strictly positive hash rate. (Recall that this was also the case in the basic model in Section 8.2.)

Claim 1: If $\mu \in \mathbb{R}^n_{\geq 0}$ is a PNE, then at least two miners have a strictly positive hash rate.

Proof of Claim 1. First suppose that $\mu = (0, 0, 0, \dots, 0)$ is the all-zeros vector. Note that then $p_i(\mu) = u_i(0) - 0 = 0$ for every i. Now any fixed miner can improve their utility by playing δ. For example, if miner $i = 1$ deviates to $\mu_1' = \delta$, we obtain $p_i(\mu_1', \mu_{-1}) = u_i(1) - \delta > 0$ by our choice of $\delta = 10^{-1} \cdot \min_i u_i(\frac{1}{2})$.

Second, suppose that exactly one miner i has strictly positive hash rate in μ. Then this miner chooses their hash rate as small as possible, i.e., $\mu_i = \delta$, to maximize their payoff. Now any other miner $j \neq i$ can improve their payoff by also choosing $\mu_j' = \delta$, because then $p_i(\mu_j', \mu_{-j}) = u_i(1/2) - \delta > 0$. This means μ cannot be a best response. This completes the proof of Claim 1.

Note that Claim 1 implies that for every $i = 1, \dots, n$ it holds that

$$\sum_{j \neq i} \mu_j > 0.$$

We will next write down the first-order conditions characterizing a pure Nash equilibrium μ and then argue that these correspond to the KKT conditions of the maximization problem in the theorem statement.

Note that, because u_i is concave, the best response function $g_i(z) = p_i(z, \mu_{-i})$ is also concave (which is to be shown in the next exercise).

> **Exercise 8.13**
>
> Assuming that $\sum_{j \neq i} \mu_j > 0$, show that the function $g_i(z) = p_i(z, \mu_{-i})$ is concave. *You may use the fact that if $s : \mathbb{R} \to \mathbb{R}$ is a concave function, and $r : \mathbb{R} \to \mathbb{R}$ concave and increasing, then the composition given by $x \mapsto r(s(x))$ is also concave (you can prove this yourself using the definition of concavity).*

Because $\lim_{z\to\infty} g_i(z) = -\infty$ it follows that the maximum of $g_i(z)$ on $[0,\infty)$ is either attained in $z = 0$ with $g_i'(0) \leq 0$, or at an internal maximum z with $g_i'(z) = 0$. In other words, if μ is a pure Nash equilibrium it (just as in the basic model) satisfies the first order conditions

$$\begin{cases} \frac{\partial p_i}{\partial \mu_i}(\mu) = 0 & \text{if } \mu_i > 0 \\ \left.\frac{\partial p_i}{\partial \mu_i}\right|_{\mu_i=0} \leq 0 & \text{if } \mu_i = 0 \end{cases}$$

which in this case translates to the equations

$$\begin{cases} u_i'\left(\frac{\mu_i}{\sum_j \mu_j}\right)\left(1 - \frac{\mu_i}{\sum_j \mu_j}\right) = \sum_j \mu_j & \text{if } \mu_i > 0 \\ u_i'(0) \leq \sum_j \mu_j & \text{if } \mu_i = 0 \end{cases}.$$

> Exercise 8.14
>
> Verify the correctness of the first order conditions above.

We next write down the KKT conditions of the maximization problem above. The objective function is in fact concave, because the sum of concave functions is again a concave function. To see that $\hat{u}_i$ is concave for every i, observe that $\hat{u}_i' = (1 - x_i)u_i'(x_i)$. This is the product of two decreasing functions (the latter function is decreasing because u_i is concave), and, hence, also decreasing. Therefore, we may conclude that $\hat{u}_i$ is concave. Therefore, we are in fact dealing with a concave maximization problem subject to linear constraints for which the KKT conditions form a characterization of an optimal solution (see Section 2.4).

Note that in what is to come, we switch between the hash rates μ_i and the market shares x_i as variables. To align our setting with the description in Section 2.4, we write the nonnegativity constraints as $-x_i \leq 0$ for $i = 1, \dots, n$. Furthermore, we write the equality constraint as $-(\sum_i x_i - 1) = 0$ for notational convenience.

The Lagrangian of the maximization problem is then

$$L(x, \lambda, \nu) = \sum_{i=1}^{n} \hat{u}_i(x_i) - \sum_{i=1}^{n} \lambda_i x_i - \nu\left(\sum_{i=1}^{n} x_i - 1\right),$$

where $\lambda = (\lambda_1, \dots, \lambda_n)$. Note that ν is a scalar and not a vector, because we only have one equality constraint. The KKT conditions reduce to the following:

1. **(Zero gradient).** The partial derivatives of L with respect to the x_i are zero, i.e., $\hat{u}_i'(x_i) - \lambda_i - \nu = 0$ for $i = 1, \dots, n$. Note that $u_i'(x_i) = -u_i(x_i) + (1 - x_i)u_i'(x_i) + u_i(x_i) = (1 - x_i)u_i'(x_i)$ so that the equations reduce to
$$(1 - x_i)u_i'(x_i) = \lambda_i + \nu \text{ for } i = 1, \dots, n.$$
2. **(Primal feasibility).** $x_i \geq 0$ for $i = 1, \dots, n$ and $\sum_i x_i = 1$
3. **(Dual feasibility).** $\lambda_i \leq 0$ for $i = 1, \dots, n$.
4. **(Complementary slackness).** $\lambda_i x_i = 0$ for $i = 1, \dots, n$.

A point x solves the maximization problem if and only if there exists a point (x, λ, ν) satisfiying the KKT conditions. In turn, it can be shown that a point (x, λ, ν) satisfies the KKT conditions if and only if x corresponds to the market shares of a pure Nash equilibrium μ. This will be the purpose of the following two exercises.

Exercise 8.15

Show that if μ is a pure Nash equilibrium, then there exists a point (x, λ, ν) that satisfies the KKT conditions where x contains the markets shares of PNE μ. *Hint: Use that μ satisfies the first-order conditions given earlier.*

Exercise 8.16

Show that if a point (x, λ, ν) satisfies the KKT conditions, then x corresponds to the market shares of the pure Nash equilibrium $\mu = \nu \cdot x$ where $\nu = (1 - x_i)u_i'(x_i)$ for any choice of i with $x_i > 0$.

This completes the proof of the theorem.

Recall from Exercise 8.11 that $\hat{u}_i(x_i) = w_i x_i \left(1 - \frac{1}{2}x_i\right)$ for $u_i(z) = w_i z$. Below we solve the optimization problem for a Tullock contest with four miners with $w_1 = 4, w_2 = 2, w_3 = 0.5, w_4 = 0.2$ using SciPy's optimization package.

```
import numpy as np
from scipy.optimize import minimize

# Given data
w = np.array([4, 2, 0.5, 0.2])  # Weights w_1, w_2, ..., w_n
n = len(w)  # Number of variables

# Define the utility function hat_u_i(x_i) for each i
def hat_u(x_i, w_i):
    return w_i * x_i*(1 -  x_i / 2)

# Define the total utility function to be maximized
# (negated for minimization)
def objective(x):
    return -np.sum([hat_u(x[i], w[i]) for i in range(n)])

# Define the constraint: sum of x_i's must be 1
def constraint_sum(x):
    return np.sum(x) - 1

# Bounds for each x_i (since x_i >= 0)
bounds = [(0, None) for _ in range(n)]

# Initial guess for the decision variables
x0 = np.ones(n) / n  # Initialize with equal distribution

# Define the constraint dictionary (sum of x_i's must be 1)
constraints = [{'type': 'eq', 'fun': constraint_sum}]

# Solve the optimization problem using the SLSQP method
```

```
result = minimize(objective, x0, method='SLSQP',
                  bounds=bounds, constraints=constraints)

# Output the results
if result.success:
    print("Optimal decision variables (x_i):")
    print(np.round(result.x, 2))  # Round to 2 decimal places
else:
    print("Optimization failed:", result.message)
```

```
Optimal decision variables (x_i):
[0.67 0.33 0.   0.  ]
```

In Exercise 8.4 we showed that the pure Nash equilibrium of this contest is given by $\mu = (8/9, 4/9, 0, 0)$, and so its market shares are $x = (2/3, 1/3, 0, 0)$ using $x_i = \mu_i/(\sum_j \mu_j)$. This indeed corresponds to the solution found by Python.

Can we also obtain the pure Nash equilibrium μ from the solution as opposed to only the market shares x? Yes, by observing from the proof of Theorem 8.4 that the normalizing constant is given by $\nu = (1 - x_i)u_i'(x_i)$ for any i with $x_i > 0$.[3] In our case we have $u_i'(x_i) = w_i$. Taking $i = 1$, we obtain

$$\nu = (1 - x_i)u_i'(x_i) = \left(1 - \frac{2}{3}\right) u_1'\left(\frac{2}{3}\right) = \frac{1}{3} \cdot 4 = \frac{4}{3},$$

so that $\mu = \nu \cdot x = \frac{4}{3} \cdot (2/3, 1/3, 0, 0) = (8/9, 4/9, 0, 0)$.

8.4.2 Economies of scale

The second variation that we consider is when the payoff functions of the miners are, for $\alpha > 1$, given by

$$p_i(\mu) = \frac{\mu_i^\alpha}{\sum_j \mu_j^\alpha} - a_i\mu_i.$$

Recall that for $\alpha = 1$ this is the payoff form we started this chapter with (before doing the transformation $w_i = 1/a_i$).

The idea here is that the mining power grows quicker than the cost associated with mining. Mathematically speaking, if we invest $a_i\mu_i$ in mining power, then the hash rate becomes μ_i^α, which grows (as a function of μ_i) at a faster rate than $a_i\mu_i$ because $\alpha > 1$. Such a setting, where mining power grows at a faster rate than investment costs, is called an *economy of scale* in the literature.

For sake of exposition we will assume that $a_1 = \cdots = a_n = 1$. Furthmore, also here we make the assumption that a best response against the all-zeros hash rate vector $\mu_{-i} = (0, \ldots, 0)$ is a sufficiently small number $\delta > 0$.

Computing a pure Nash equilibrium in closed form is not possible for this model. It is also not possible to phrase this problem as a concave maximization problem as we did in the previous section. Nevertheless, we

[3] If you would use an optimization package like CVXPY in Python then there are more direct ways of obtaining the dual multipliers of the Lagrangian.

can prove an interesting property of this model. If μ is a pure Nash equilibrium of this Tullock contest and miner i has a strictly positive market share, i.e.,

$$x_i(\mu) = \frac{\mu_i^\alpha}{\sum_j \mu_j^\alpha} > 0$$

then it must be that $x_i(\mu) = 1 - 1/\alpha$, i.e., miner i's market share can be bounded away from zero independent of the other hash rates.

This implies that there can be at most $\alpha/(\alpha - 1)$ miners active in the mining process in an equilibrium! For sake of illustration if $\alpha = 1.1$ then at most 11 miners are active. This might explain, to some extent, why typically only a couple of "big players" are active in a mining markets (under the assumption of having an economy of scale).

We summarize the main result in the theorem below.

Theorem 8.5 (Economies of scale)**.** *If the common payoff function of the Tullock contest, for $\alpha > 1$, is given by*

$$p_i(\mu) = \frac{\mu_i^\alpha}{\sum_j \mu_j^\alpha} - \mu_i = x_i(\mu) - \mu_i,$$

then in a pure Nash equilibrium μ, for every $i = 1, \dots, n$ either $x_i(\mu) = 0$ or $x_i(\mu) \geq 1 - 1/\alpha$. This implies there can be at most $\alpha/(\alpha - 1)$ miners with strictly positive market share $x_i(\mu)$.

We will prove this theorem in the following exercise.

Exercise 8.17

Let μ be a pure Nash equilibrium of the above Tullock contest.

1. Show that $\mu_i = \alpha x_i(\mu)(1 - x_i(\mu))$ for every $i = 1, \dots, n$ by optimizing the best response function $g_i(z) = p_i(z, \mu_{-i})$ over $z \geq 0$.
2. Argue that, if $x_i(\mu) > 0$, then $x_i(\mu) \geq 1 - 1/\alpha$. *Hint: Use that $p_i(\mu) \geq 0$ always holds and combine this with the equation from 1.*

Exercise 8.18

Show that if the common payoff function of the Tullock contest is given by

$$p_i(\mu) = \frac{\mu_i^\alpha}{\sum_j \mu_j^\alpha} - \mu_i$$

for $\alpha > 2$ and $i = 1, \dots, n$ with $n \geq 2$, then no pure Nash equilibrium exists in the game.

8.5 Acknowledgements

Section 8.1, 8.2 and 8.4.1 are loosely based on Chapter 4.4 of the book Contest Theory: Incentive Mechanisms and Ranking Methods (2016) by Milan Vojnović. The Tullock contest has originally been defined by Tullock (1980).

Section 8.3 is based on the paper Best-Response Dynamics in Lottery Contests by Ghosh and Goldberg (2023). In particular, the example showing that best response dynamics might not converge is taken from this

paper. The fact that the function P in Section 2.3.1 is a best response potential function has been shown in the paper The Lottery Contest is a Best-Response Potential Game by Ewerhart (2017).

Section 8.4.1 is based on the paper Bitcoin: A Natural Oligopoly by Arnosti and Weinberg (2018).

Chapter 9

Selfish mining attack

In Chapter 7 we argued that the proportional allocation rule is collusion-proof in the Proof-of-Work protocol, meaning that no group of miners can increase their (expected) mining rewards by posing as one miner. That is, if a group of miners would combine their mining power in a *mining pool* then the expected reward that the pool gets is not more than the sum of the expected rewards that the individual miners would have gotten if they would have mined on their own.

A key assumption that is made in this reasoning[1] is that we are only considering the rewards of mining one block on the blockchain. In practice, miners can participate in the mining of multiple blocks sequentially. All kinds of other strategic considerations then come into play that miners could potentially exploit to gain more expected reward from a sequence of blocks in a colluding pool than they would have gotten in expectation by mining on their own.

Furthermore, if mining in a pool is more profitable than mining by yourself, more and more miners will join a pool (which are typically "open to all"). At some point a pool might become so large that more than 50% of all miners are in it. In this case the Proof-of-Work protocol loses its decentralized nature and the majority pool could compromise the intended functioning of the blockchain; recall from Chapter 6 that we need a majority of honest nodes to make sure the Proof-of-Work protocol functions correctly.

Informally speaking, we would like a protocol to be *incentive compatible (with respect to collusion)*, with respect to miners whose objective is to maximize their total expected reward, meaning they have no miner has an incentive to collude with other miners in a pool (and with that obtain more reward). This term comes from the area of mechanism design, and can be seen as the reverse of game theory: Instead of designing a game and then studying its outcome, one designs the game so that it has an *intended* outcome. In our case the intented outcome of "the game" should be that miners do not want to collude.

The main purpose of this chapter is to show that the Proof-of-Work protocol is *not* incentive compatible when considering reward maximizing miners. It turns out that there exists a "block withholding" strategy, also called *selfish mining attack*, where the colluding pool can strategically decide not to announce a newly mined block to the chain right away, but instead secretly continue mining on a new block that could extend it (i.e., working on a secret *fork* of the chain). If this is done cleverly, even some minority pools are able to obtain relatively more reward than they would have gotten if all the miners in it would have mined on their own.

Theorem 9.1 (Selfish mining, informal)**.** *For any mining pool that contains at least* $1/3$ *of the total mining power of the blockchain, running a Proof-of-Work protocol, there exists a strategy with which the pool can*

[1] Apart from ignoring other considerations such as risk aversion.

obtain a total expected reward that is strictly larger than $1/3$ of all the rewards handed out by the protocol.

9.1 Revisiting longest chain protocol

To better understand this type of attack, let us first recall briefly the relevant notions of *blockchain (e.g., Bitcoin) mining* in the Proof-of-Work protocol from the Computer Science part of this course (you can also recap Chapter 5 and Chapter 6 yourself). A blockchain can be represented as a directed graph where every node has out-degree 1.

Each node represents a block containing a record of transactions that were made. To distinguish between blocks, each of them contains a unique identifier as well as the identifier of the block that it points towards, i.e., its predecessor. The root block B_0 is called the *genesis block*.

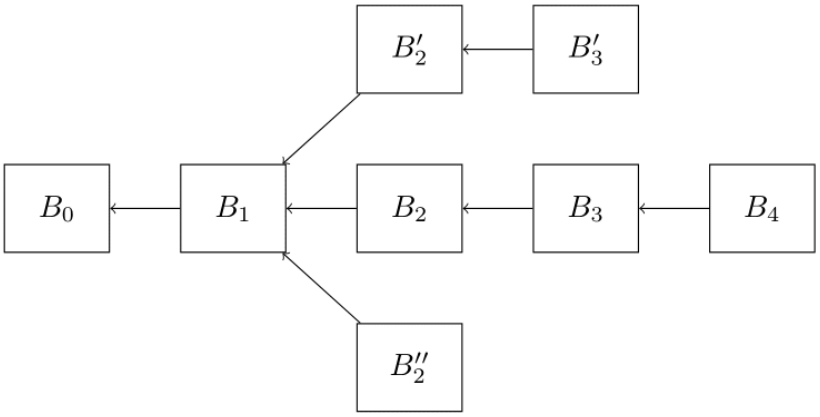


Figure 9.1: A labelled blockchain

Miners can add a new block to the blockchain by appending it to a *leaf* block which is a block in the current configuration that is not a predecessor of another block. In Figure 9.1 these are B_2'', B_3' and B_4. In other words, these are the blocks at the end of a chain (or path) starting in B_0.

If a miner adds a block so that the resulting new chain is the unique longest chain of the new graph, then as a reward, they receive a unit of cryptocurrency. Adding a new block to an existing leave is only allowed if the miner is the first to solve a leave-specific cryptographic puzzle; the process of solving this puzzle is called *mining*. Adding a block to a chain that is not the longest eventually leads to the transactions in those blocks being ignored, i.e., not accepted, by the network (they can be resubmitted though).

Due to the fact that communication is often not 100% instantaneous it might happen that two different miners add distinct blocks F_1 and F_2 to the current longest chain in the network (ending in B_4), creating a *fork*, or forking, with two branches. The block F_1 is called the head of the first branch, and F_2 the head of the second branch.

At this point, the units of cryptocurrency for adding the blocks F_1 and F_2 is not yet awarded to the respective miners that mined the blocks, because the paths leading to F_1 and F_2 are at the moment not part of the unique longest chain in the graph. Emperical evidence suggests this type of "accidental bifurcation" happens roughly once every 60 blocks.

If forking happens then all the miners active in the blockchain network can choose to continue mining on either F_1 or F_2. A miner usually mines on the first branch that they have heard of. As soon as one of the

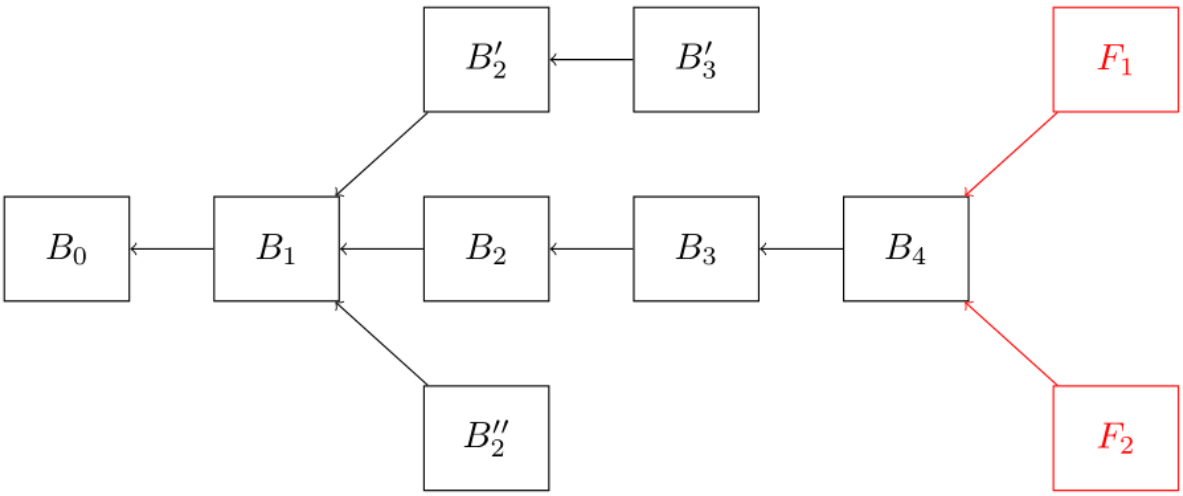


Figure 9.2: A fork in the currently longest chain

branches gets extended with an additional block (and is accepted by the network as the branch containing the longest chain in the network) the rewards of this block, as well as that of the F_i that is its predecessor, are rewarded to the miners that mined those respective blocks.

Finally, if a group of miners forms a pool, then their *expected relative reward* (given a configuration of the blockchain network) is the total expected number of cryptocurrency units awarded to the pool, i.e., the number of blocks that they successfully mined and added to the chain, divided by the total number of units awarded to all miners (including the pool) in the system. This latter number is precisely the length of the longest chain in the network.

9.1.1 Mining time

Recall that the probability with which a miner i is the first to solve the cryptographic puzzle (giving the right to add the newly mined block), is proportional to the amount of computation power (or hash rate) μ_i that they put into the mining process.

Given that we are now considering the mining of multiple blocks over time, we also need a notion of time to model how long it takes for a miner to mine a block (consistent with the fact that the probability of being the first miner to solve the puzzle is proportional to the hash rate of that miner).

> **(A1)** We assume that the time that it takes for miner i with hash rate μ_i to solve the cryptographic puzzle is a random variable X_i that follows an exponential distribution with parameter $\lambda_i = \mu_i$, i.e., for $x \geq 0$,
>
> $$\mathbb{P}(X_i \geq x) = 1 - \mathrm{e}^{-\lambda_i x}.$$

Proposition 9.1 states that if two miners with hash rates μ_1 and μ_2 are trying to extend the same leaf block, then μ_1 is the first to solve the cryptographic puzzle with probability $\mu_1/(\mu_1 + \mu_2)$, and similarly, miner 2 is the first solve it with probability $\mu_2/(\mu_1 + \mu_2)$.

Proposition 9.1 (Minimum of exponential random variables)**.** *If X_1 and X_2 are random variables following*

an exponential distribution with parameters $\lambda_1 = \mu_1$ and $\lambda_2 = \mu_2$, respectively, then

$$\mathbb{P}(X_1 \leq X_2) = \mathbb{P}(X_1 = \min\{X_1, X_2\}) = \frac{\mu_1}{\mu_1 + \mu_2}.$$

Similarly, if n miners, each with an exponential distribution are trying to solve the puzzle with hash rate μ_i for miner i, then

$$\mathbb{P}(X_i = \min\{X_1, \dots, X_n\}) = \frac{\mu_i}{\sum_j \mu_j}.$$

This is exactly what we wanted.

Another useful and desirable property of the exponential distribution is that it is *memoryless*. If a miner has already spend some time s on trying to solve the crypto puzzle, then the probability that they will solve it after time $t + s$, for some t, is the same as the probability that they will solve it after time t.

Proposition 9.2 (Memoryless property)**.** *If X is a random variable following an exponential distribution, then for any $s, t \geq 0$,*

$$\mathbb{P}(X > t + s | X > s) = \mathbb{P}(X > t).$$

Said differently, the fact that you have already spend some time on solving the puzzle is not going to help you to solve it quicker compared to someone who just started to work on the puzzle. This is a valid assumption because miners essentially only provide random guesses for the puzzle solution (see Chapter 6).

9.2 Warm-up example

If risk-neutral miners, whose goal is only to maximize the expected reward that they receive on the blockchain, follow the longest chain protocol and always add new blocks to the chain right after they have mined them, there is no incentive for them to form a pool if within the pool the unit reward is split proportional to the mining power of the miners participating in the pool (see the example in Section 7.3).

However, it turns out that there is an incentive for a pool to not *directly* reveal their newly mined block to the network, but instead secretly continue mining on a new block that could be appended to the already mined block. At this point, we assume this *secret mining* is possible. Under some mild assumptions we justify this assumption in Section 9.2.1.

If it would happen that in the meantime an honest node announces another new block, the pool quickly makes its secret block public as well, resulting in a forking of the longest chain, see, e.g., Figure 9.2. The justification for making this possible is the fact that accidental bifurcation is a known phenomenon, as we mentioned earlier. To emphasize the importance of this assumption, we state it below.

> **(A2)** If a pool has a secretly mined block F_p (not yet known to miners outside of the pool) extending a leaf node B of the currently longest chain, and a miner from outside the pool announces their own new block F_h extending B, then the pool can quickly announce its secret block F_p, resulting in a forking of the longest chain.

To keep the exposition clean, we will always assume that the ensemble of miners can be divided into two groups for a given $\alpha \in (0, 1)$: A pool of colluding miners with a total hash rate of α, and a group of individual (honest) miners with a total hash rate of $1 - \alpha$. We sometimes say that the *pool has size* α.

To develop some intuition about why it could be useful to withhold a block, let us consider an example (or thought experiment) in which we analyze the reward of a mining pool of size α for the next two blocks that are added to the longest chain.

We will describe a strategy that the pool can follow so that their expected reward for the two newly added blocks exceeds their share α of the total mining power in the network, assuming α is large enough (we return to this point later). This strategy does not conflict with the Proof-of-Work protocol, as the protocol itself says nothing about when miners should announce newly found blocks; this is not relevant if miners do not care about rewards.

Suppose that both the pool and the honest miners are trying to mine a new block to append to the current longest chain with leaf (i.e., most recent) node B. We will describe a strategy that makes it beneficial (if α is large enough) for the pool to withhold a newly mined block.

To describe this strategy, which is called the *selfish mining attack*, we make a case distinction depending on whether the honest miner or the pool mines a new block first given the currently (unique) longest chain.

> In all the figures below, the top branch is always branch containing blocks mined by the pool and the bottom branch contains blocks mined by honest miners. Furthermore, blocks known to *all* miners are denoted with solid squares, whereas secret blocks of the pool are denoted with dashed squares.

- **Case 1: An honest miner mines a block $F_{h,1}$ extending B before the pool does**. This happens with probability $1-\alpha$. In this case, both the honest miners and pool accept this block as the next block of the longest chain; recall that the pool follows the protocol except for that it might delay the announcement of a block they found themselves. The pool gets no reward.

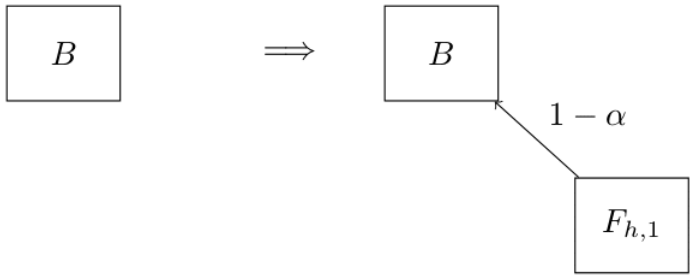


Figure 9.3: Case 1

- *Case 1a: An honest miner mines a new block $F_{h,2}$ extending $F_{h,1}$ before the pool finds a block*. This happens with probability $(1-\alpha)$ and so overall this subcase happens with probability $(1-\alpha)^2$. Everyone accepts this block and again the pool gets no reward.

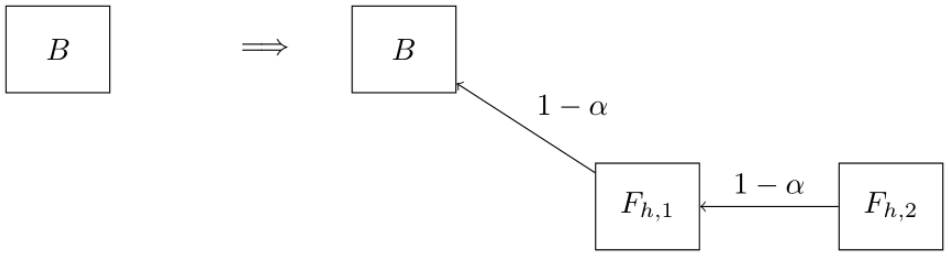


Figure 9.4: Case 1a

- *Case 1b: The pool mines a block $F_{p,2}$ extending $F_{h,1}$, before an honest miner does.* This happens with probability α, and so overall this subcase happens with probability $(1-\alpha)\cdot\alpha$. In this case the pool simply makes the block public and collects its unit reward.

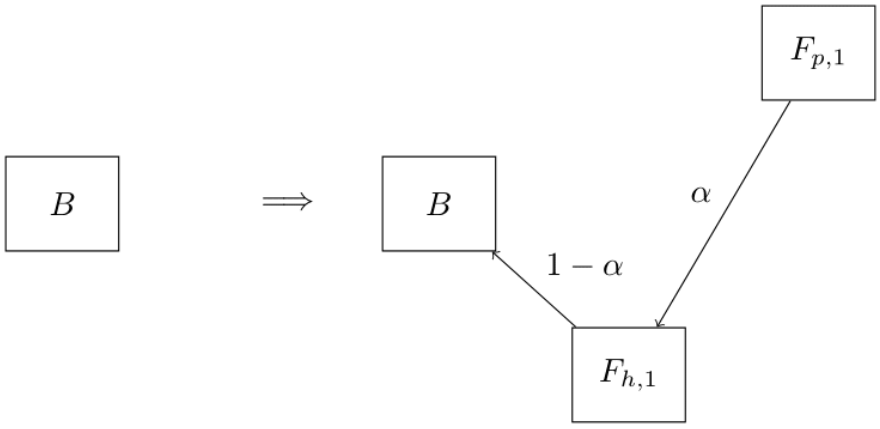


Figure 9.5: Case 1b

- **Case 2: The pool mines a block $F_{p,1}$ extending B before an honest miner does.** This happens with probability α. In this case the strategy of the pool is to not reveal this block right away, but instead *secretly* start mining on a new block to extend $F_{p,1}$ whereas the honest miners continue mining on block B.

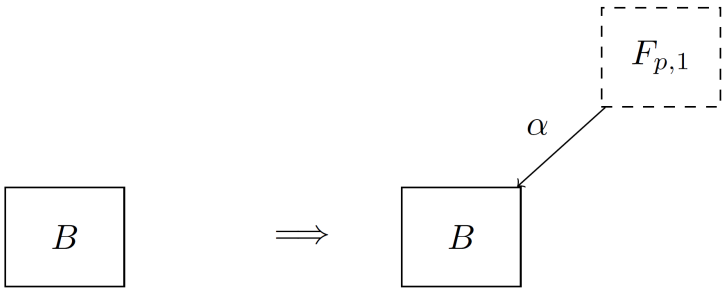


Figure 9.6: Case 2

- *Case 2a: The pool mines a block $F_{p,2}$ extending $F_{p,1}$ before an honest extends B.* Observe that honest miners don't have the option to try and extend $F_{p,1}$ because they are not yet aware of its existence. The probability that the pool finds $F_{p,2}$ before the honest miners extend B is α. This follows from Proposition 9.2: Although the honest miners have been mining longer on extending B than the pool has been mining on extending $F_{p,1}$, this does not give an advantage to the honest miners. Therefore, overall this case happens with probability α^2. The pool now announces the block $F_{p,1}$ followed quickly by the announcement of $F_{p,2}$. Both blocks are accepted and the pool gets two units of reward, one for $F_{p,1}$ and one for $F_{p,2}$.
- *Case 2b: An honest miner mines a block $F_{h,1}$ extending B before the pool mines a block $F_{p,2}$ extending the (secret) block $F_{p,1}$.* This happens overall with probability $\alpha(1-\alpha)$. The pool now quickly also announces their (until now secret) block $F_{p,1}$. This is possible because of Assumption A2. In this case

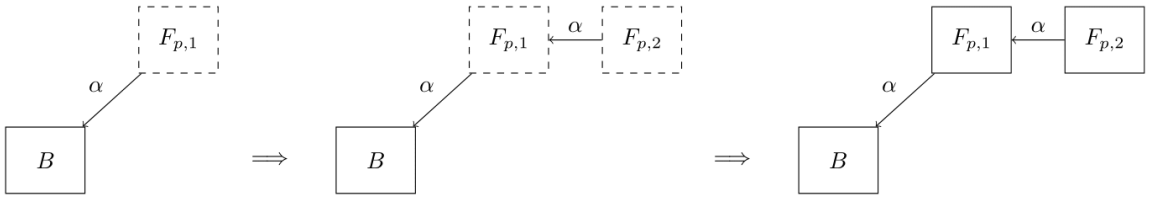


Figure 9.7: Case 2a

we have a forking in the longest chain. The strategy of the pool is now to keep mining to extend $F_{p,1}$, whereas the honest miners can choose between extending $F_{p,1}$ and $F_{h,1}$.

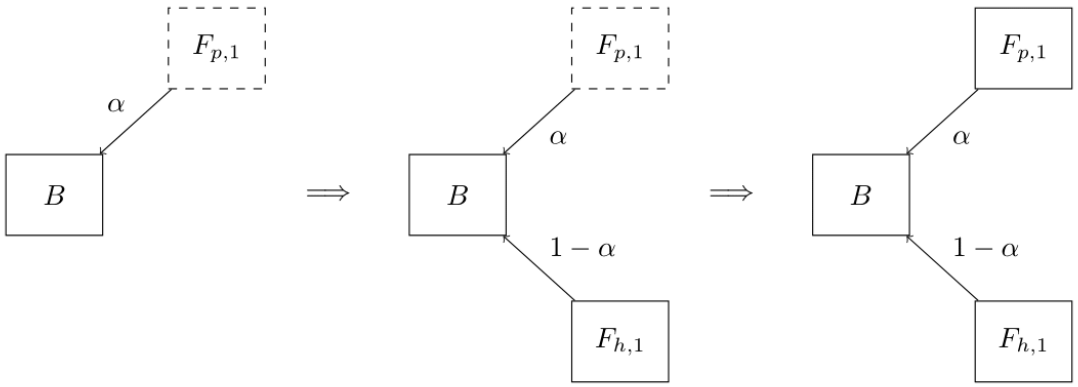


Figure 9.8: Case 2b

Assume in Case 2b that all honest miners will start mining on $F_{h,1}$. We will show in Exercise 9.1 that if some fraction $\gamma \in [0,1]$ of the honest miners also mines on $F_{p,1}$, this can only increase the total expected reward obtained by the pool.

- **Case 2b-I: An honest miner mines a block $F_{h,2}$ extending $F_{h,1}$ before the pool mines a block extending $F_{p,1}$.** This happens overall with probability $(\alpha(1-\alpha)) \cdot (1-\alpha) = \alpha(1-\alpha^2)$. Everyone accepts $F_{h,2}$ and the pool gets no reward.
- **Case 2b-II: The pool mines a block $F_{p,2}$ extending $F_{p,1}$ before an honest miner extends $F_{h,1}$.** This happens overall with probability $(\alpha(1-\alpha)) \cdot \alpha = \alpha^2(1-\alpha)$. In this case, because the pool has been the only one mining on both $F_{p,1}$ and $F_{p,2}$ they receive a total reward of two for these blocks.

Taken altogether, the pool receives one unit of reward in Case 1b with probability $\alpha(1-\alpha)$, two units of reward in Case 2a with probability α^2, and two units of reward in Case 2b-II with probability $\alpha^2(1-\alpha)$. That is, the total expected reward of this selfish mining attack is

$$1 \cdot \alpha(1-\alpha) + 2 \cdot \alpha^2 + 2 \cdot \alpha^2(1-\alpha) = -2\alpha^3 + 3\alpha^2 + \alpha$$

Given that 2 units of cryptocurrency were handed out, the expected relative reward of the pool is $(-2\alpha^3 + 3\alpha^2 + \alpha)/2$. Elementary calculus shows that

$$\frac{-2\alpha^3 + 3\alpha^2 + \alpha}{2} > \alpha$$

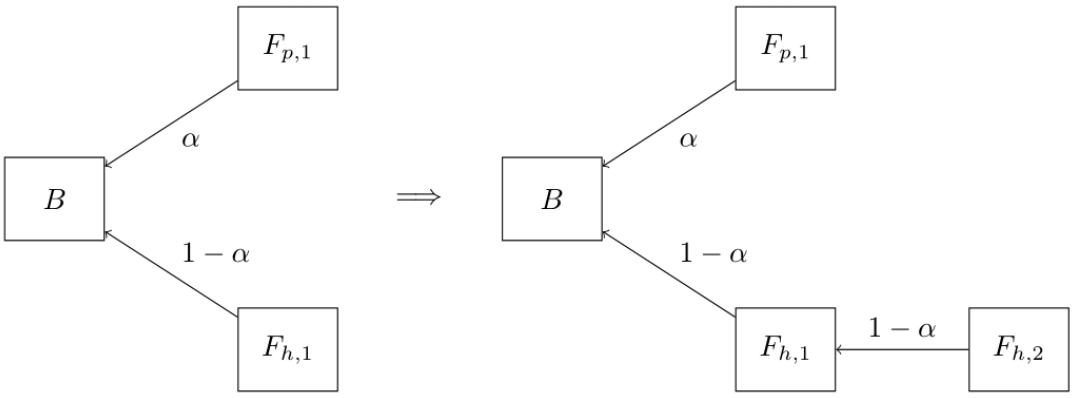


Figure 9.9: Case 2b-I

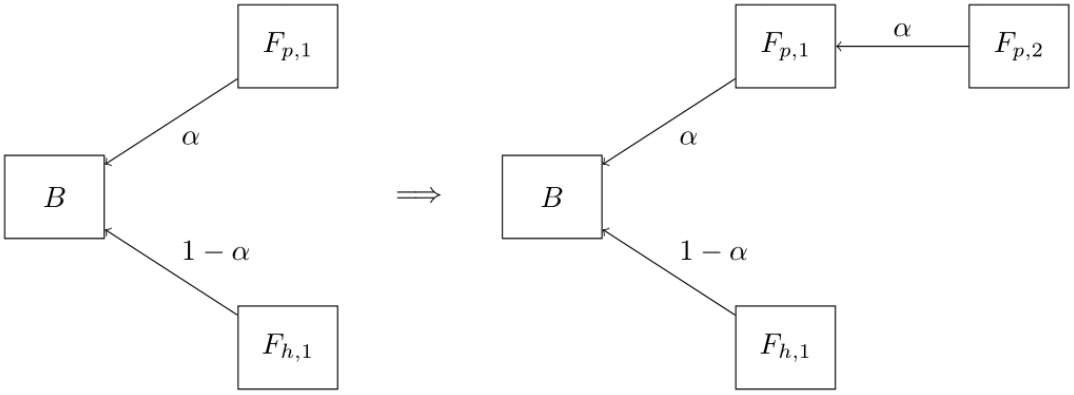


Figure 9.10: Case 2b-II

if and only if $\alpha > 0.5$. This means that the selfish mining attack is profitable for a group of colluding miners that form a majority in the system!

Although collusion has been observed in practice, the assumption that there is a majority of miners that colludes from the start is unrealistic. In fact, the desirable guarantees that we have seen for the Proof-of-Work protocol rely on the fact that at least half of the nodes are honest, i.e, $\alpha < 0.5$. Otherwise, the network is subject to all other kinds of manipulation (that might be way worse than simply withholding newly mined blocks for some time).

Very suprisingly though, it turns out that a more elaborate analysis of the selfish mining attack (not restricted to the addition of two new blocks) shows that even minority pools can profit from such an attack and obtain more reward than they would have gotten individually! In particular, we will show that also a pool of size $1/3 < \alpha < 1/2$ can carry out a selfish mining attack and obtain a reward that is relatively more than α.

Exercise 9.1

Let $\gamma \in (0, 1)$ and suppose that after the pool announces its secret block $F_{p,1}$ in Case 2b, a fraction γ of the honest miners starts mining to extend $F_{p,1}$ and a fraction $(1 - \gamma)$ of the honest miners starts mining to extend $F_{h,1}$.
By redoing the case analysis starting from the final situation in Case 2b, show that the expected contribution of the subsequent cases to the total expected reward for the pool is

$$2\alpha^2(1 - \alpha) + \alpha(1 - \alpha)^2 \cdot \gamma.$$

Not all cases give a nonzero reward for the pool, but work out all (three) of them for completeness. *Note that if an honest miner ends up extending $F_{p,1}$ with a block $F_{h,2}$, then the pool gets no reward for this latter block, but it does get the reward for the block $F_{p,1}$, as the branch containing $F_{p,1}$ and $F_{h,2}$ is now part of the new longest chain.*

9.2.1 Secret mining details

To understand how secretly trying to extend a secret block works from a technical perspective, let us recall the basics of mining. The idea is that we are given a function h, that takes as input a "guess" z and outputs $h(z)$, and a *difficulty level* τ. If $h(z) \leq \tau$, then z is a solution to the crypto puzzle and a miner can use this solution to obtain the right to add a block to the blockchain.

A solution z is the concatenation of four parts (see Definition 6.2 for the full structure of z), the most important for us being that z specifies which predecessor block the solution z is trying to extend. In secret mining this part of the solution is simply the last secretly mined block of the pool that has not been announced publicly.

Recall that the function h never changes in the Proof-of-Work protocol, i.e., it is fixed in the blockchain software, but the difficulty level τ is updated periodically, to keep the expected interval between mined blocks the same (this is done roughly every two weeks). In this chapter we assume that, while the pool is carrying out a selfish mining attack, the difficulty level τ does not change. Otherwise, it might be that already secretly mined blocks become invalid as a result of τ being lowered (if τ would be increased, this is not problematic with respect to secret mining).

9.3 General case

We continue with the general description of the selfish mining attack. It essentially follows the example with the difference being Case 2a (explained below). If the pool is the first to mine a block extending leaf node B, they keep it secret. If the honest miners mine a block, and the pool has a secret block, they announce it right away to create a forking. If then the pool is the first to mine a second block, they announce this as well so that their chain of two blocks is the new longest chain and their last block becomes the new leaf node B.

> **(A3)** We assume an honest miner never tries to extend a block mined by the pool in case of a forking consisting of two branches of equal length. Similar as in Exercise 9.1, in the final reward analysis that we do in Section 9.3.5, if honest miners would also try to extend a block mined by the pool, this can only increase the expected reward of the pool.

In Case 2a, when the pool has mined both the secret blocks $F_{p,1}$ and $F_{p,2}$, it releases them both and collects a reward for both of them. Instead of doing this, the pool could also have simply kept mining to get a longer *private/pool branch* (a branch that contains some blocks only known to the pool).

Of course, we have to take into account that honest miners might also find a block in the meantime, which they could use to extend B. An extension of B using such a block of an honest miner is called an extension into the *public branch*.

Roughly speaking, the core idea of the selfish mining attack is that the pool keeps trying to extend their pool branch and in this way stay in the lead with respect to the public branch. As an example, suppose that (although probabilistically unlikely) the pool mines five blocks for the pool branch, and keeps all these secret, before an honest miner mines a first block for the public branch. Next, suppose an honest miner finds a block to extend B that they add to the public branch. To make sure the pool branch becomes the longest branch in the long run, the pool right away must make public their first block mined in the secret pool branch to create a fork in the publicly known network. This block we now refer to as the *publicly known part of the pool branch*.

If the pool waits with making public their secret block, the honest miners will agree on the first block of the public branch being added to the longest chain, after which the secret block(s) of the pool becomes useless.

The *secret part of the pool branch* are the remaining four secret blocks. The pool can keep mining on the last block of the secret part of the pool branch with the hope of extending it. Whenever an honest miner mines a block to extend the public branch, the pool right away also releases a next block to extend the public part of the pool branch. In this way, the public part of the pool branch (known to both the pool and all honest miners) always stays as long as the public branch.

If at some point the difference in length between the total pool branch and the public branch becomes one block, the selfish mining strategy prescribes to also announce the last secret block of the pool branch right away, at which point the publicly known part of the pool branch (which is now the complete pool branch) is one block longer than the public branch and therefore the unique longest chain. In Figure 9.11 this would happen if the honest miners mine two additional blocks for the public branch, before the pool manages to extend the pool branch. The pool now receives one unit of reward for *every* block that is on the pool branch, which is now fully publicly known; in Figure 9.11 the pool would receive 5 units of reward. Intuitively, if the difference becomes equal to one, the pool doesn't want to risk losing its advantage over the public branch and therefore announces their last block to secure the rewards of all blocks on the pool branch.

At this point, the process starts again, i.e., the pool tries to get a secret lead of two or more blocks starting in the last node of the previously pool branch. This node is now the new leaf node; in Figure 9.11 this is the right-most block of the pool branch. Whenever the difference between the pool and public branch drops

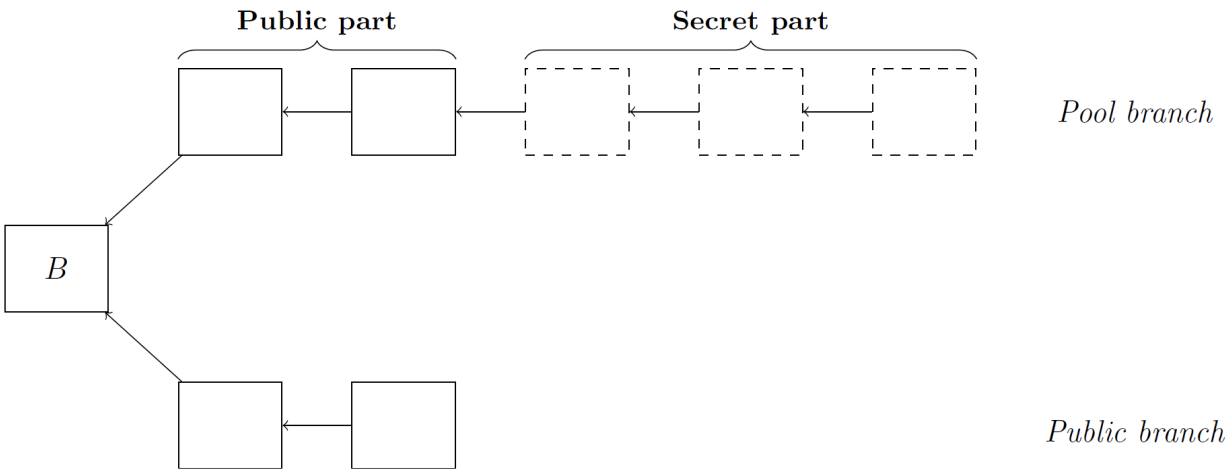


Figure 9.11: Schematic overview of branches

down to one block, the pool releases its last secretly mined block of the new pool branch. The process then again repeats, etc.

9.3.1 Pseudo-code

We next provide the pseudo-code for the above description. To understand this code correctly it is important to realize that the only time that the public branch becomes longer than the pool branch, is when both have length 0 (or both length 1) and the honest miners are the first to find a block.

We also emphasize once more that, assuming the public part of the pool branch (its first, say, k nodes) has length at least one, this part is never shorter than the public branch (apart from the negligible time interval where the honest miners have announced a new block and the pool responds to this by making one of its secret blocks public).

Finally, recall that if both the public branch and the public part of the pool branch have equal length, then all honest miners are mining on the public branch, as stated in Assumption A3. Just as in the warm-up example, if some honest miners would also branch on the publicly known pool branch, this can only increase the overall expected reward that the pool obtains in the end.

In the pseudocode below, we write `length(public)` for the number of blocks in the public branch at a given point in time, and `length(pool)` for the total number of blocks (both secret and public) in the pool branch.

```
Initialization:
     - Leaf node B of unique longest chain
     - Emptpy (full) pool branch; Empty public branch

If pool finds a new block F_p:
    If length(public) = 1 and length(pool) = 1:
    # Forking in which both blocks are public (Case 2b-II)
        - Append F_p to pool branch as public block
        - F_p becomes the new leaf node B (= pools wins)
```

```
        - Public and pool branch are set to be empty
    Else:
        - Append F_p to pool branch as a secret block

If honest miner finds a new block F_h for public branch:
    Append F_h to public branch
    If length(pool) = 0: # Case 1
        - F_h becomes the new leaf node B (= honest miner wins)
        - Public branch is set to be empty
    Else if length(pool) = 1 and length(public) = 1:
    # Case 2b; pool branch's block was secret until now
        - Pool branch directly makes secret block public
        # Public part of pool and public branch now have length 1
    Else if length(pool) = 1 and length(public) = 2:
    # Case 2b-II; pool branch's block was already public
        - F_h becomes the new leaf node B (= honest miner wins)
        - Public and pool branch set to be empty
    Else if length(pool) - length(public) = 1
    # All but two last blocks of pool branch are then public
        - Pool announces last two secret blocks (= pool wins)
        - Last secret block becomes the new leaf node B
        - Public and pool branch set to be empty
    Else if length(pool) - length(public) = d > 1
    # Last d + 1 blocks of pool branch are then secret
        - Pool makes oldest secret block public in pool branch
        # Pool is guaranteed to eventually win this block
```

Below is also the pseudo-code without comments.

```
Initialization:
     - Leaf node B of unique longest chain
     - Emptpy (full) pool branch; Empty public branch

If pool finds a new block F_p:
    If length(public) = 1 and length(pool) = 1:
        - Append F_p to pool branch as public block
        - F_p becomes the new leaf node B (= pools wins)
        - Public and pool branch are set to be empty
    Else:
        - Append F_p to pool branch as a secret block

If honest miner finds a new block F_h for public branch:
    Append F_h to public branch
    If length(pool) = 0:
        - F_h becomes the new leaf node B (= honest miner wins)
        - Public branch is set to be empty
```

```
Else if length(pool) = 1 and length(public) = 1:
    - Pool branch directly makes secret block public
Else if length(pool) = 1 and length(public) = 2:
    - F_h becomes the new leaf node B (= honest miner wins)
    - Public and pool branch set to be empty
Else if length(pool branch) - length(public branch) = 1
    - Pool announces last two secret blocks (= pool wins)
    - Last secret block becomes the new leaf node B
    - Public and pool branch set to be empty
Else if length(pool branch) - length(public branch) = d > 1
    - Pool makes oldest secret block public in pool branch
```

Exercise 9.2

Suppose that blocks are mined sequentially according to the sequence $(H, P, P, H, P, P, P, H, P, H)$ starting from an initial (genesis) block B. An H means that the honest miners found a block, and a P means that the pool found a block.
Carry out the pseudo-code above and give the final state of the blockchain (including secret blocks, if any) for this sequence of mined blocks. Label every block with an H or P, denote public blocks with a solid square, and denote secret blocks with a dashed square (as in, e.g., Figure 9.11).

Solution

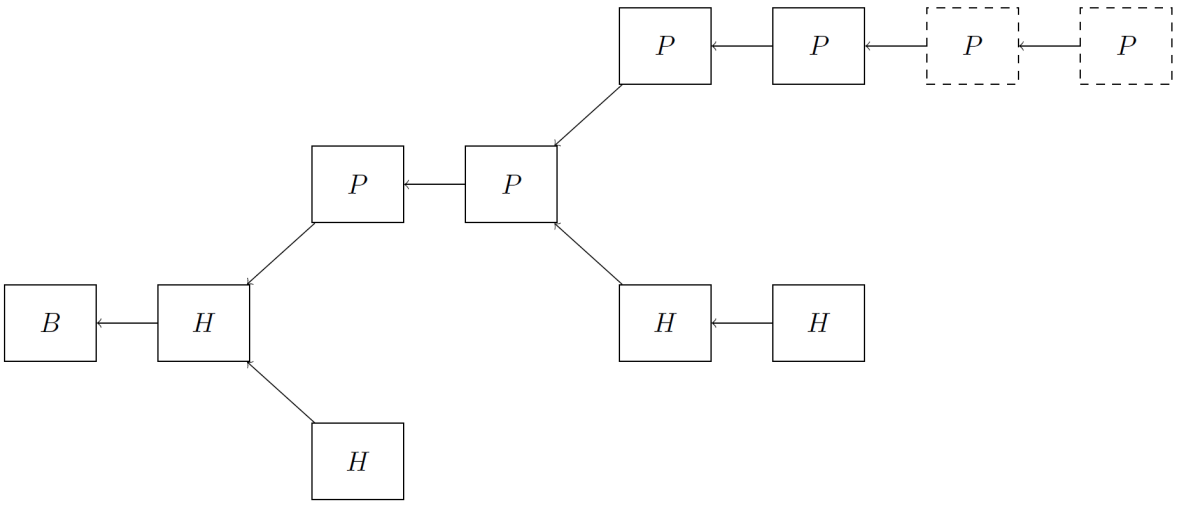


Figure 9.12: Solution to Exercise 9.2

Exercise 9.3

Repeat Exercise 9.2, but now for the sequence $(P, P, P, H, P, H, H, H, P, H, H)$. How many units of reward has the pool received in this sequence (if any secret blocks remain, do not take these into account)?

Feel free to write down a self-chosen sequence, as in Exercises 9.2 and 9.3, and execute the pseudocode.

> **Exercise 9.4**
>
> Explain that at no point in the selfish mining attack both the (full) public branch and the pool branch consist of two blocks (regardless of which blocks in the pool branch are public or secret).
> Observe that the same reasoning shows that at no point the public and pool branch both consist of $d \geq 2$ blocks.

> **Exercise 9.5**
>
> Explain that after the honest miners have announced a newly found block F_h to their public branch, but right before the pool has responded to this:
> - if `length(pool) - length(public) = 1`, then it must be that precisely the last two blocks of the pool branch are still secret.
> - if `length(pool) - length(public) = d > 1`, then it must be that the last $d + 1$ bloks of the pool branch are still secret.

It turns out there is a convenient way to represent the selfish mining process using a *Markov chain* $\mathcal{M} = (S, P)$. A Markov chain consists of a *state space* S and a *transition matrix* $P \in [0,1]^{S \times S}$ that describes the probabilities with which we move from one state to another.

9.3.2 State space

In our case the states will be the difference between the length of the (full) pool branch and the public branch. This difference is precisely the number of secret blocks that the pool branch still has. Therefore our state space is $\mathbb{N}_0 = \{0, 1, 2, 3, \dots\}$.

Looking at the pseudo-code, there is one special case, which is when the difference is 0. The selfish mining attack makes different choices depending on whether both the public and pool branch have length 0, or when they both have length 1 (in which case the block of the pool branch is public). Recall from Exercise 9.2 that it cannot happen that both branches have a length of $d \geq 2$.

Therefore, we split the state 0 into two states: 0 and $0'$. The state 0 corresponds to the setting where both branches have length 0, and $0'$ correspond to the case where both branches have length 1 (i.e., the start of the fork). Overall our state space then becomes

$$S = \{0, 0', 1, 2, 3, \dots\}.$$

Also note that if the difference is 1, and both the pool and the honest miners are mining, this can only happen when the public branch has length 0 and the pool branch has length 1 (with one secret block). In all other cases, a situation with a difference of 1 would result in the public branch becoming part of the new longest chain, or the pool branch becoming part of the new longest chain.

9.3.3 Transition probabilities

The transition probabilities in the matrix P are drawn in the figure below.

Note that for every node, the sum of its outgoing edges equals 1. We will next explain the transition probabilities.

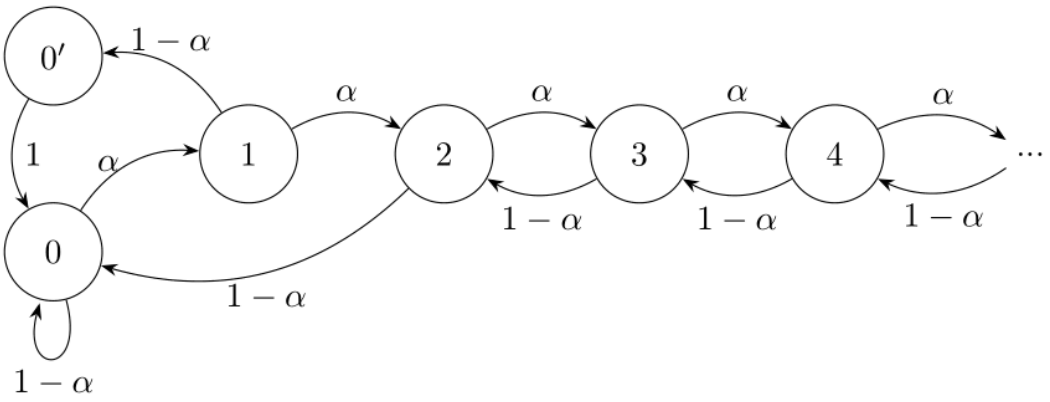


Figure 9.13: Transition probabilities in state space S

Similar as before, in all the figures below, the top branch is always the pool branch, and the bottom branch is the public branch. Furthermore, publicly known blocks (on the public branch or the public part of the pool branch) are denoted with solid squares, whereas secret blocks on the pool branch are denoted with dashed squares.

- **State** 0: Both the pool and the public branch have length 0.
  - If honest miners find a block F_h first, which happens with probability $1-\alpha$, it is added to the public branch. The public branch becomes part of the longest chain and the process then restarts with F_h being the new leaf node B. We therfore start again with a public and pool branch of length 0 and so do not move in the state space.

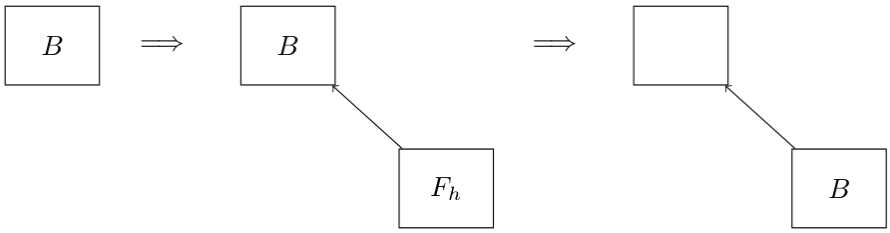


Figure 9.14: Honest miner finds block first in state 0

  - If the pool find a block F_p first, which happens with probability α, the pool branch becomes one block (which is secret) longer than the public branch so we move to state 1.
- **State** $0'$: Both pool and publich branch have length 1.
  - If honest miners find a block first, which happens with probability $1-\alpha$, the public branch becomes part of the longest chain, and the process restarts.
  - If the pool finds a block first, which happens with probability α, the pool branch becomes part of the longest chain, and the process restarts.

Since the process restarts in both cases here, with an empty pool and public branch, we move to state 0 with probability $\alpha + 1 - \alpha = 1$.

- **State** 2: The pool branch is two (secret) blocks longer than the public branch.

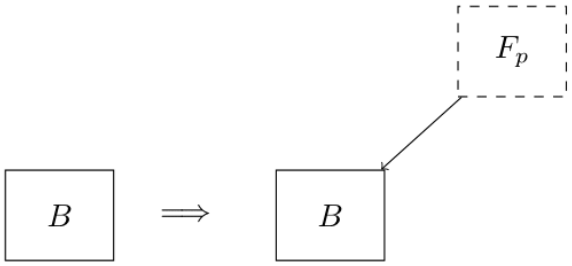


Figure 9.15: Pool finds block first in state 0; F_p stays secret for now (hence, dashed)

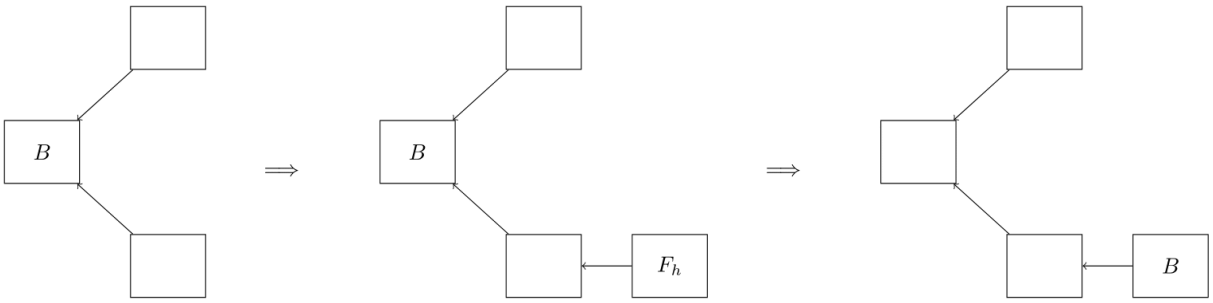


Figure 9.16: Honest miner finds block first in state $0'$.

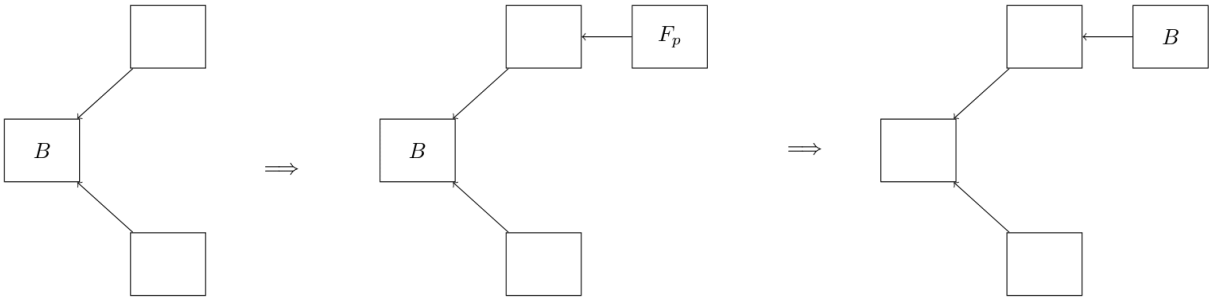


Figure 9.17: Pool finds block first in state $0'$.

– If honest miners find a block first, which happens with probability $1 - \alpha$, the pool right away announces their two secret blocks so that the pool branch becomes part of longest chain. The process now restarts so we move to state 0.

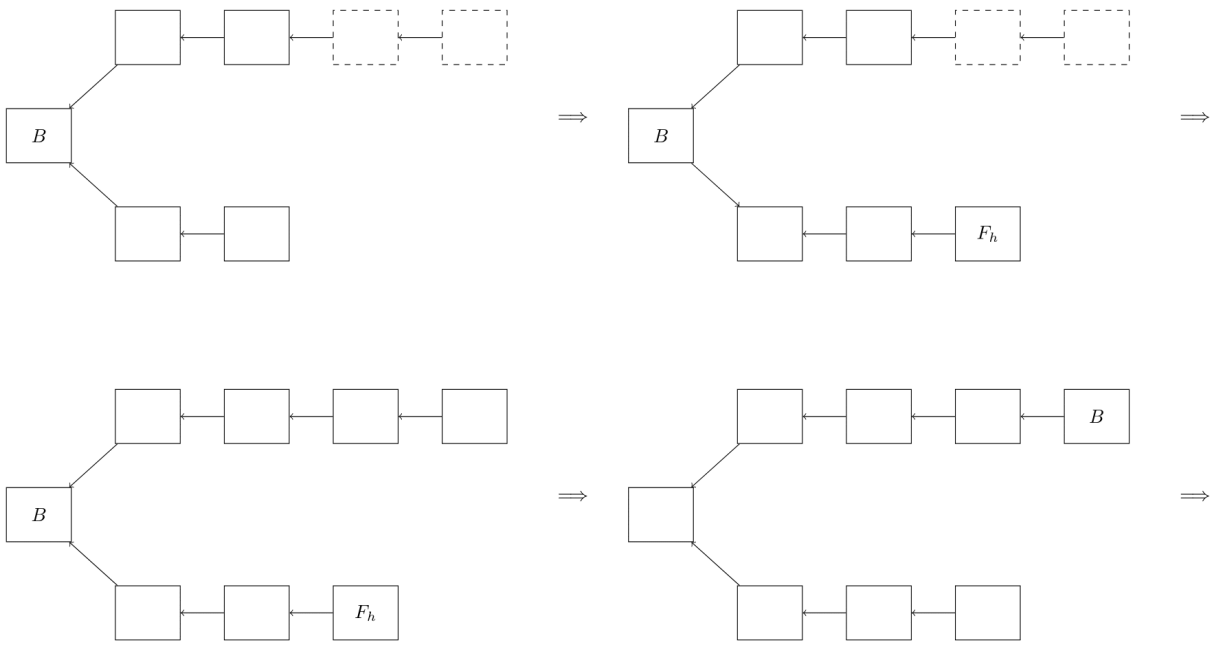


Figure 9.18: Honest miners find block first in state 2. In this example the pool branch started out with four blocks of which the first two were public and the last two secret (hence, dashed).

– If the pool finds a block first, which happens with probability α, we add it to the pool branch as a secret block and so the pool branch becomes three (secret) blocks longer than the public branch (which are the last three blocks). We therefore move to state 3.

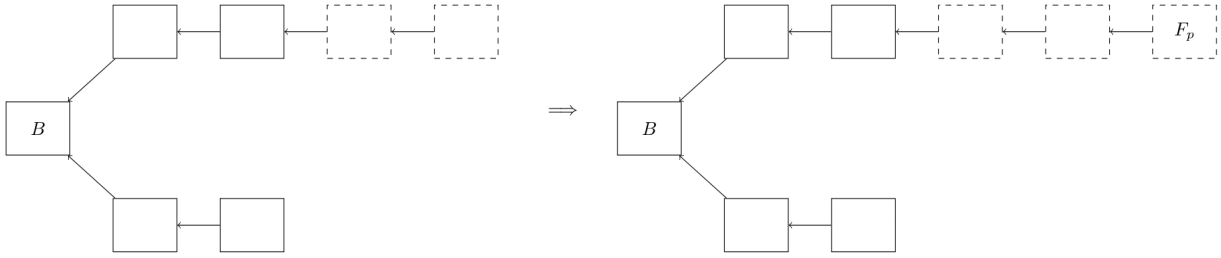


Figure 9.19: Pool finds block first in state 2, so we now move to state 3.

Exercise 9.6

Before reading along, explain the transition probabilities for states 1 and $3, 4, \ldots$ using similar descriptions and figures as above.

Solution

- **State** 1: The pool branch consists of one secret block.
  - If honest miners find a block first, which happens with probability $1-\alpha$, the pool right away makes there secret block public to create a fork in the publicly known network. In this case we move to state $0'$.

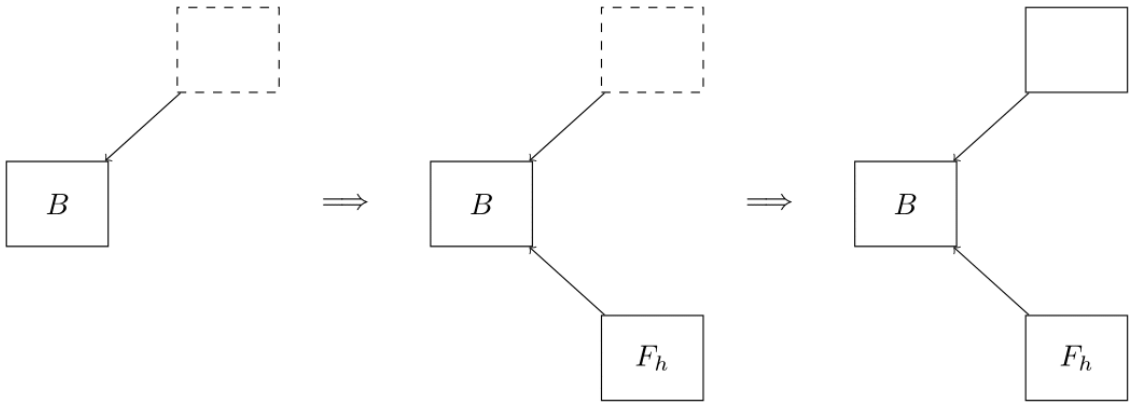


Figure 9.20: Honest miners find a block first in state 1.

  - If the pool finds a block first, which happens with probability α, we add it to the pool branch as a secret block and so the pool branch now consists of two secret blocks.

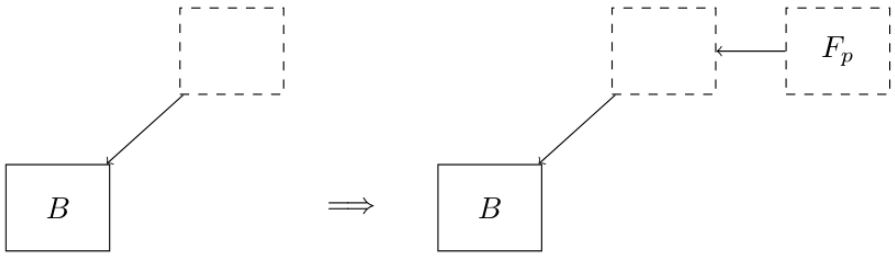


Figure 9.21: Pool finds block first in state 1, so we now move to state 2.

- **State** $d \geq 3$: The pool branch is d (secret) blocks longer than the public branch.
  - If honest miners find a block first, which happens with probability $1-\alpha$, the pool right away makes the left most secret block of the pool branch public, so that its public part is as long as the publich branch. Since we now have one secret block less, we move to state $d-1$.

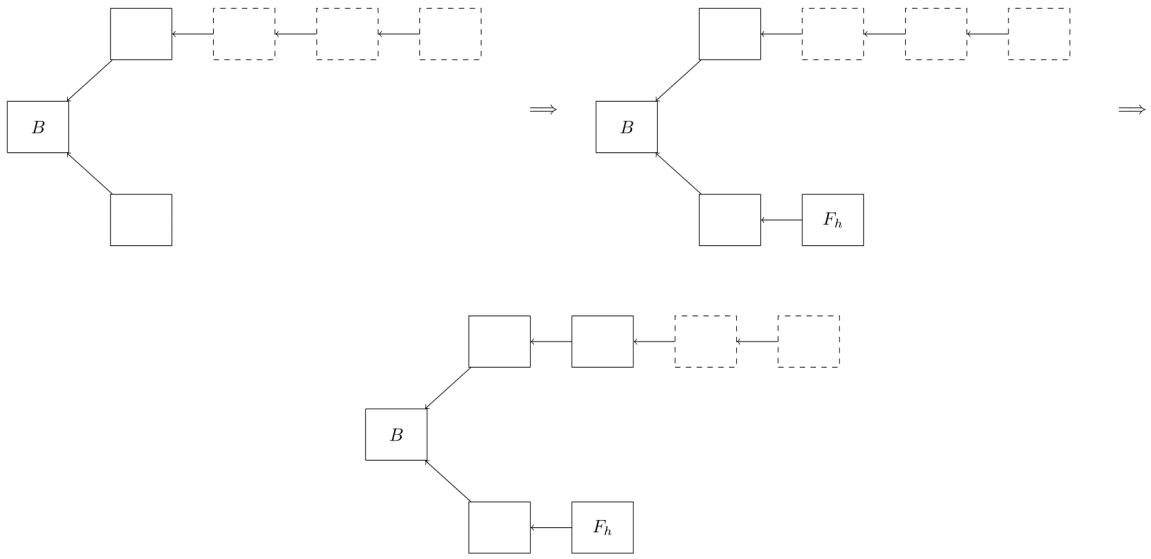


Figure 9.22: Honest miners find a block first in state $d = 3$. In this example the length of the pool branch is 4 and that of the public branch 1 (so that the difference is three). It could also have been the case, e.g., that the public branch had length 3 and the pool branch length 6 (again so that the difference would have been 3).

– If the pool finds a block first, which happens with probability α, we add it to the pool branch as a secret block and so the pool branch now consists of $d + 1$ secret blocks.

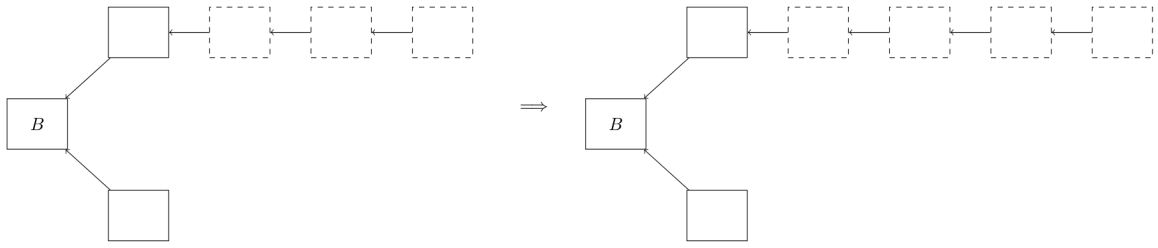


Figure 9.23: Pool finds block first in state $d = 3$, so we now move to state $d + 1 = 4$.

9.3.4 Stationary distribution

Assume we start in state 0, with B being the genesis block of the blockchain. We would like to compute the *stationary probabilities* π_i of every state i. That is, given that we have performed enough probabilistic transitions, what is the probability that we are in state i of the state space? It turns out that this question only has a valid answer if $0 < \alpha < 1/2$, so we will make this assumption from now on (recall that we already saw in the warm-up example that pools of size $\alpha > 1/2$ can obtain more reward when using a type of selfish mining attack).

The (row) vector

$$\pi = (\pi_{0'}, \pi_0, \pi_1, \pi_2, \dots)$$

is called the *stationary distribution* of the Markov chain. It can be computed by solving the (infinite) system

of equations $\pi P = \pi$, where P is the transition matrix, whose entries P_{ij} or transitioning from state i to j are given in Figure 9.23. For example, for state $i \geq 3$, the i-th row of the system $\pi P = \pi$ reads

$$\alpha p_{i-1} + (1-\alpha)p_{i+1} = p_i$$

Exercise 9.6

Write down the equations in $\pi P = \pi$ for states $i \in \{0', 0, 1, 2\}$.

Exercise 9.7

Show that for $0 < \alpha < 1/2$, the row vector $\pi = (\pi_{0'}, \pi_0, \pi_1, \pi_2, \dots)$ given by

$$\begin{cases} \pi_{0'} = \dfrac{(1-\alpha)(\alpha - 2\alpha^2)}{1 - 4\alpha^2 + 2\alpha^3} \\ \pi_0 = \dfrac{\alpha - 2\alpha^2}{\alpha(1 - 4\alpha^2 + 2\alpha^3)} \\ \pi_i = \left(\dfrac{\alpha}{1-\alpha}\right)^{i-1} \dfrac{\alpha - 2\alpha^2}{1 - 4\alpha^2 + 2\alpha^3} \quad \forall i \geq 1 \end{cases}$$

is the stationary distribution solving $\pi P = \pi$ by verifying that π satisfies the equations from Exercise 9.6, and the equations $\alpha p_{i-1} + (1-\alpha)p_{i+1} = p_i$ for $i \geq 3$.
Also show that π is indeed a probability distribution, by showing that $\pi_{0'} + \sum_{i=0}^{\infty} \pi_i = 1$ and $\pi_i \geq 0$ for all i.
Alternatively, you can also try the more challenging problem of solving from scratch the system $\pi P = \pi, \pi_{0'} + \sum_{i=0}^{\infty} \pi_i = 1$.

9.3.5 Reward analysis

To determine the reward of the pool when using the selfish mining attack, we extend the Markov chain description with the notion of rewards. Given that we move from a state $i \in \{0', 0, 1, 2, \dots\}$ to some other state, we can compute the expected rewards associated with moving away from i.

If we multiply these rewards with the stationary probabilities of every state, we obtain the expected overall reward per block from the selfish mining strategy in the long run. In mathematical terms, we turn our Markov chain into a *Markov Decision Process (MDP)* by introducing a reward associated to every transition.

We will next consider all the states and their associated awards.

- **State** $0'$. Recall that in this state there is a forking in which both the pool and public branch have length 1. If the honest miners manage to mine a block first extending the public branch they receive a reward of 2. This happens with probability $1 - \alpha$ and so their expected reward is $2(1-\alpha)$. If the pool manages to mine a block first extending the pool branch, they obtain a reward of 2. This happens with probability α, and so the expected reward when transition away from state $0'$ is 2α.
- **State** 0**.** In this case both the public and pool branch are empty. If the honest miners mine a block first, they obtain a reward of one, and so an expected reward of $1 \cdot (1-\alpha)$. If the pool mines a block first, they keep it secret, and so no rewards are (yet) handed out to the pool when transitioning away from this state.

- **State** 1**.** In this case the the pool has one secret block. If the pool manages to mine a block first, this is kept secret as well, and so no reward is handed out. If the honest miners mine a block first, the pool announces its secret block, thereby creating a forking. Also then, no rewards are handed out.

We purposely skip state 2 at this point, and first discuss state $i \geq 3$.

- **State** $i \geq 3$**.** If the pool is the first to mine a new block, then we transition to state $i+1$, with one more secret block, and no reward is handed out. If the honest miners are the first to mine a new block, the pool makes its oldest secret block public. Because the pool has a lead of more than two blocks on the honest miners (i.e., the pool branch is at least two blocks longer than the public branch), we know that eventually, once the difference in length of the branches drops down to one, the pool will get the reward for this block. Therefore, we count the reward for the secret block that was made public in this transition.
- **State** 2**.** If the pool is the first to mine a new block, we transition to state 3, with one more secret block, and no reward is handed out. If the honest miners mine a block first, the pool makes its last two secret blocks public, and collects the rewards of *all* the blocks in the pool branch (recall that being in state 2 means that the difference in length between the pool and public branch is two secret blocks; the state does not specify how long the public parts of these branches are). Recall that we already counted the rewards of all the earlier (secret) blocks that were made public in states $i \geq 3$ (previous caes), and so here we only count the rewards for the last two secret blocks that were made public in this transition. Since the honest miners are the first to mine a block with probability $1-\alpha$, the expected reward of the pool here is $2(1-\alpha)$.

To summarize the last two cases, we count the reward of a block in the pool branch at the moment it is made public. Below we have given a summary of the rewards obtained by the pool and the honest miners per state.

State i	Expected pool reward (r^i_{pool})	Expected honest miners' reward (r^i_{honest})
$0'$	2α	$2(1-\alpha)$
0	0	$(1-\alpha)$
1	0	0
2	$2(1-\alpha)$	0
$i \geq 3$	$1-\alpha$	0

Figure 9.24: Overview reward analysis per state.

This means that in stationarity, the overall expected reward of the poolper block is equal to

$$\begin{aligned} r_{\text{pool}} &= \pi_{0'} \cdot r^{0'}_{\text{pool}} + \sum_{i=1}^{\infty} \pi_i \cdot r^i_{\text{pool}} \\ &= \pi_{0'} \cdot 2\alpha + \pi_2 \cdot 2(1-\alpha) + \left(\sum_{i=3}^{\infty} \pi_i \right) \cdot (1-\alpha) \end{aligned}$$

and that of the honest miners is

$$r_{\text{honest}} = \pi_{0'} \cdot r^{0'}_{\text{honest}} + \sum_{i=1}^{\infty} \pi_i \cdot r^i_{\text{honest}} = \pi_{0'} \cdot 2(1-\alpha) + \pi_0 \cdot (1-\alpha).$$

It can be shown that $r_{\text{pool}} + r_{\text{honest}} < 1$, which intuitively follows from the fact that because of possible forkings, the protocol will not hand out a unit reward for every mined block.

To compute the share of the rewards that the pool gets, we want to compute the *expected relative reward* of the pool, which is

$$R_{\text{pool}} = \frac{r_{\text{pool}}}{r_{\text{pool}} + r_{\text{honest}}}.$$

Exercise 9.8

Show that

$$R_{\text{pool}} = R_{\text{pool}}(\alpha) = \frac{4\alpha^2 - 9\alpha^3 + 4\alpha^4}{1 - \alpha - 2\alpha^2 + \alpha^3}$$

On the domain $(0, \frac{1}{2})$, the expression in Exercise 9.8 is greater then α if and only if $\frac{1}{3} < \alpha < \frac{1}{2}$, as Figure 9.25 shows.

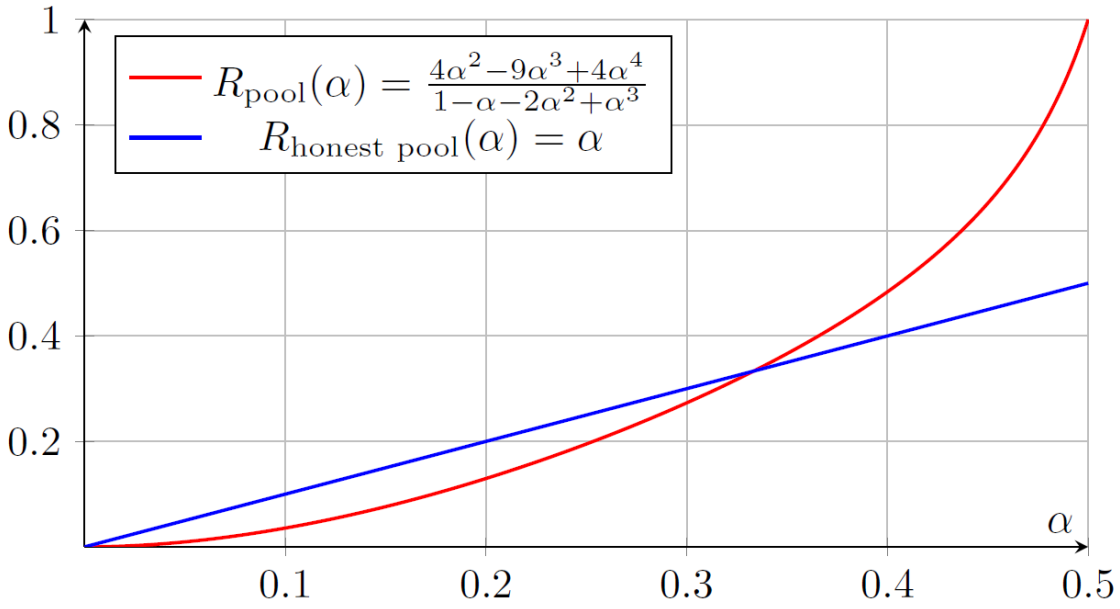


Figure 9.25: Relative reward of pool when using selfish mining.

This leads us to the main theorem, showing that indeed also minority pools of size at least $1/3$ can benefit from selfish mining.

Theorem 9.2 (Selfish mining)**.** *For any mining pool of size $0 < \alpha < \frac{1}{2}$, the expected relative reward of the selfish mining strategy is*

$$\frac{4\alpha^2 - 9\alpha^3 + 4\alpha^4}{1 - \alpha - 2\alpha^2 + \alpha^3}.$$

The above quantity is greater than α, the expected relative reward when the pool also mines honestly, if and only if $\frac{1}{3} < \alpha < \frac{1}{2}$.

9.4 Acknowledgements

This section is based on the paper Majority is not Enough: Bitcoin Mining is Vulnerable by Eyal and Sirer (2013).