The Attestation Aggregation and Packing Problem

Satalia & Sigma Prime

May 19, 2022

Abstract

The Attestation Aggregation and Packing Problem (AAPP) arises in the context of Ethereum's Proof of Stake (PoS) consensus protocol. This document aims at formally defining the problem. This will provide the necessary level of detail needed to develop optimisation algorithms and state properties of them, if necessary.

Lighthouse, Sigma Prime's Ethereum consensus client, uses a greedy heuristic to generate solutions to the AAPP. The goal of the present investigation is to assess how far the solutions generated by Lighthouse's greedy heuristic are from the real optimum. Ultimately, the goal is to confirm that the current algorithm is sufficiently good, or to develop an algorithm that produces more profitable solutions.

Contents

1 Background

This section aims to provide a high-level discussion of some concepts behind blockchain technology those relevant to understand the AAPP.

1.1 Decentralised trustless architecture

At its core, a blockchain is a database designed to be tamper-proof. Unlike other technologies which rely on *trusted central authorities* to guarantee the integrity of the stored data, a blockchain achieves the same goal using a *decentralised trustless architecture*.

This decentralised trustless architecture is based on a combination of *cryptography*, which allows every participant in a blockchain to independently verify the integrity of the stored data, and *consensus protocols*, which allow participants to reliably agree on what "history" of the blockchain is the correct one. This necessity to agree on the history of the blockchain is a side effect of the distributed nature of the blockchain, which means that the participants' view of the blockchain can become out of sync.

1.1.1 Advantages of decentralisation

The advantage of such a decentralised trustless architecture is that it doesn't require to trust a central authority, which is handy when no authority can be trusted. This feature partially explains why such technologies gained traction in the wake of the global financial crisis (GFC, circa 2007-2009) where people's trust in the banking system collapsed. The historical background of the financial crisis, is also the reason behind the original and most popular (but not only) use of blockchain technology, i.e., decentralised currencies (also known as *cryptocurrencies*).

1.1.2 Disadvantages of decentralisation

One disadvantage of the decentralised trustless architecture is complexity. For a blockchain to be truly tamper-proof and to operate efficiently, a large number of participants need to reach consensus on its correct state. Consensus protocols exist for this purpose, but their design is challenging, because they simultaneously have to deal with

- the unreliable nature of computer networks, and
- deliberate attacks to the blockchain.

Because of the value of the data stored in the blockchain, which always include some form of currency, successful attacks can be very profitable, and are therefore not uncommon. The primary way blockchains protect themselves against such attacks, is by making it very resource-intensive to modify the state of the blockchain, in particular its past history.

Many blockchains (including Ethereum) are secured by Proof-of-Work (PoW) consensus protocols. In these protocols, the resource to be invested to modify the blockchain is computational power, i.e., time and energy (plus, to some extent, hardware). While such protocols have protected blockchains worldwide reasonably well[1], their energy requirements are considered to be a core disadvantage of blockchain technology. This is particularly true in the *current* historical context, where energy consumption is under scrutiny due to its impact on climate change.

1.2 Distributed state

As mentioned above, a blockchain is a distributed database. Because it is a database, it needs to maintain an internal state, i.e., the data stored in it. However, because it is distributed, the data doesn't live at one specific location, e.g., a server somewhere on the Internet. Instead, the state of the blockchain lives in the collective memory of the many participants connected to its network. Each participant must keep a local copy of the blockchain state, and communicate

[1] With the exception of a few severe successful attacks.

with other participants via peer-to-peer protocols in order to synchronise state and carry out other work fundamental to the blockchain's operation.

In the case of the many blockchains whose main goal is to implement a cryptocurrency, the state corresponds to a *ledger*, i.e., a document specifying each participant's "balance". The Ethereum blockchain, however generalises this concept. Like other blockchains, Ethereum contains a ledger for its native currency (ether), however the state of the blockchain also contains the state of a *state machine* called the Ethereum Virtual Machine (EVM). In a state machine, the system changes state through state transitions. In the EVM, some of the blockchain transactions are associated with the execution of arbitrary logic, called *smart contracts*.

1.3 Blocks and chains

The blockchain is the historical record of the transactions happened so far. Transactions are grouped into blocks, and valid blocks become part of the history when they are appended. Because of the lack of a centralised authority, any participant can virtually produce a block and broadcast it to the rest of the participants.

1.4 Consensus protocols

Consensus protocols are rules that all participants in a blockchain must respect. Such protocols are designed to provide incentives for participants to do useful work for the blockchain (e.g., producing new blocks, recording new transactions), and to protect the blockchain from attacks that aim at rewriting the blockchain's history. Participants who play by the rules are rewarded by the protocol, and participants who don't are penalised. In this section we give an overview of the two main types of consensus protocols: Proof-of-Work and Proof-of-Stake. We cover the former for historical reasons, and the latter because it's the context for the AAPP.

1.4.1 Proof-of-Work

A Proof-of-Work (PoW) consensus protocol discourages tampering by making it unreasonably hard to modify the history of the blockchain. In order to produce a state change, a participant needs to demonstrate that they have carried out some *work* that is expensive and cannot be avoided. While doing the work is hard, verifying that that work has been done is trivial.

Many blockchains, including Bitcoin and Ethereum, are based on PoW consensus protocols. Here is how they work: apart from its payload (i.e., the data it contains, such as transactions) and other metadata, a block contains

- the hash[2] of (a pointer to) its parent block,

[2] We assume that the reader is familiar with the meaning of a hash and the general properties of hash functions.

- a *nonce*,
- the hash of the current block.

In order for a block to be deemed *valid* it needs to satisfy some properties, which are defined by the blockchain's consensus protocol. For example, a protocol may state that a block is only valid if its hash is smaller than a given number. This mechanism is actually very close to the workings of blockchains such as Bitcoin and Ethereum, and the number hashes are compared with is called *cryptographic difficulty* (or just *difficulty*). In order to produce a block that won't be rejected by the other participants, a participant must find a nonce (the only element of a block that they have control over) such that hashing that number together with the payload of the block and the pointer to its parent will yield a hash with a value lower than the difficulty. Because hash functions are designed to be practically impossible to invert, the participant's only option to generate a valid block is to try a lot of different values for the nonce until a suitable hash is produced. This search is called *mining*, and requires a lot of computational resources, which makes producing a new block hard. In fact, mining is hard by design, and represents half of the mechanism used by blockchains to discourage tampering. An attacker who intends to modify the data in a block will invalidate its hash, and will thus have to re-mine that block to make it valid again. By contrast, verifying the validity of a block is trivial, which makes it easy for participants to spot an irregularity.

The other half of the mechanism to tamper-proof a blockchain, is the *chaining of blocks* itself. At this point, producing a block in the context of an attack and producing a block in the service of the blockchain is equally hard. However, while a legitimate participant will mostly want to serve the chain by appending legitimate blocks to it, an attacker will mostly want to rewrite the blockchain's history to change its state for their own benefit. The linked structure of the blockchain, makes that very hard. Because the data used to produce a block's hash includes the hash of the previous block in the chain, re-mining a block at an arbitrary position in the blockchain's history will automatically invalidate all of the successive blocks. In other words, successfully rewriting the blockchain's history will require a mining effort proportional to the age of the oldest block one wants to rewrite. The financial cost of rewriting a sequence of blocks, together with the fact that mining takes time, and that the blockchain continually grows, makes rewriting the blockchain's history very unlikely.

1.4.2 Proof-of-Stake

In a Proof-of-Stake (PoS) consensus protocol discourages bad behaviors by requiring that participants put forward, or *stake*, a large (fixed) sum of native cryptocurrency in order to earn the right to produce a block. If a participant performs legitimate work, they get rewarded with some of the native currency, but if they fail to do so, they lose the resources that they have staked. The upcoming new version of Ethereum's consensus protocol is of this kind.

Unlike in Proof-of-Work protocols, there is no artificial computation that needs to happen for a participant to be able to produce a new block, instead the participants "prove" their intentions to behave well by committing to lose their resources if they don't. Producing a block still takes some time, because there is some unavoidable work to do, however this time is not dictated by the complexity of mining the block, instead it is enforced by the protocol. In the case of Ethereum, a block can be produced every 12 seconds. This time discretisation in the production of blocks, and the fact that not all participants can create blocks at their convenience, helps reducing the situations where the blockchain becomes a tree of blocks.

1.4.3 Incentives

Due to the fact that it relies on a distributed trustless architecture, a blockchain requires participants to carry out useful work, e.g., producing new valid blocks, and to avoid disruptive behavior, e.g., tampering with the blockchain history. To this end, consensus protocols make extensive use of (positive or negative) reward mechanisms to incentivise participants to align with the interests of the chain. Rewards are quantified in a native cryptocurrency, e.g., Bitcoin, Ether, etc.

1.5 Agreeing on one history

The distributed nature of the blockchain leads to some more complications. Because participants can produce and share new blocks all the time, the chain doesn't always look like a linear *sequence of blocks.* For instance, two participants may produce two "competing" blocks with the same predecessor, resulting in a *tree of blocks.*

The trouble with trees of blocks is that each branch of the three corresponds to a different "truth" about the state of the blockchain, which needs to be reconciliated in order for the chain to operate correctly and efficiently. From the perspective of the participants, this means agreeing on which branch of the tree represents the real truth. Blocks that are not part of such branch are called *uncle blocks.*

1.5.1 Longest chain

In some blockchains, a convention among the participants is that, in presence of a tree of blocks, the *longest branch* of the tree represents the one true history. In order to incentivise participants to commit to the longest branch, consensus protocols reward block producers asymmetrically, i.e., non-uncle blocks are rewarded significantly more than uncle blocks.

1.5.2 Voting on one history

Aside from the introduction of Proof-of-Stake, Ethereum's new consensus protocols introduces some novelties also around the mechanism to agree on the true history of the blockchain. Instead of adopting the longest branch as the one true blockchain, *validators*, i.e., participants who agreed to stake their currency and provide services to the blockchain, cast a vote to elect (among other things) what they believe is the *head*, i.e., the latest block of the correct branch, of the blockchain.

Such "votes" are called *attestations*, and are the subject of the AAPP defined in this document.

2 Attestations in Ethereum

As discussed in the previous section, a validator vote for the head of the chain by producing an attestation. A fundamental property of attestations is that, under certain conditions they can be aggregated.

An attestation is defined by the following data

- a set of **attesters** (possibly 1) that "back" it,
- some **attestation data** (among other things, the slot in which the attestation was produced, the hash of the block believed to be the head of the chain, and the hashes of two - source and destination - blocks, which are related to the mechanism by which older parts of the blockchain are finalised),
- a Boneh–Lynn–Shacham (BLS) **signature** produced using the private keys of the attesters.

The signature allows to verify the integrity of an attestation. Moreover, a crucial property of the BLS signature scheme is that it allows to *aggregate* multiple signatures into a single signature. This allows one to aggregate multiple attestations into a single one. In order for two attestations to be aggregated, the following conditions must hold

1. their attestation data must be identical,
2. their sets of attesters must be disjoint.

An aggregated attestation can contain a **maximum of 2048** attesters and, while it carries more weight than an unaggregated attestation, it counts as one for our purposes.

2.1 Validators

Validators play an important role in Ethereum's Proof-of-Stake protocol, and understanding their role will help understanding where attestations come from.

Validators perform a number of different *duties* (responsibilities)

- **Block creation duties.** A validator with block creation duties is responsible for collecting transactions and attestations from the peer-to-peer network of validators, choosing what to include in the next block, and then produce and share a block.

- **Attestation duties.** A validator with attestation duties will synchronise their local state of the blockchain, and cast a vote on what they believe to be the head of the blockchain.
- **Attestation aggregation duties.** A validator with attestation aggregation duties is responsible for grouping compatible attestations and producing aggregate attestations.

Note that validators are not only rewarded for producing blocks, but also for attesting.

2.2 Epochs, slots, committees

Time in Ethereum is split in successive *epochs* of 6.4 minutes. Each epoch is further split into 32 *slots* of 12 seconds. Each block has a corresponding slot, however not all slots necessarily have a corresponding block. In fact, there should be only one block per slot, and validators are penalised for proposing more than one.

At each epoch one validator (the *proposer*) is selected[3] to propose a new block at a given slot. When its assigned slot starts, the proposer has a chance to produce a block.

At each epoch, the validators are shuffled and grouped into *committees*. Each committee has $n \in [128, 2048]$ validators. A slot has $m \in [1, 64]$ committees. The majority of the validators in a committee become *attesters* and will have to produce an attestation. Among these a small amount, on expectation 16, will become *aggregators* and will have to aggregate attestations.

2.3 Timeline

This section gives a brief overview of the timelines involved. These are not crucial to the problem formulation, but help understanding where the problem comes from, and why there is only a very limited amount of time available to solve it.

At each epoch, the work of each validator (be it a proposer, an attester, or an aggregator) needs to be carried out in the time of the slot assigned to them. The time before and after the assigned slot is still used to do useful work, mostly around gathering information, or finishing work that wasn't completed on time.

Before the start of their assigned slot, a **proposer** will start gathering attestations produced up to 32 slots earlier, as well as transactions, to include in the block. When the slot starts, the proposer will produce and share the block as quickly as possible, to maximise the likelihood that it will be considered by the attesters.

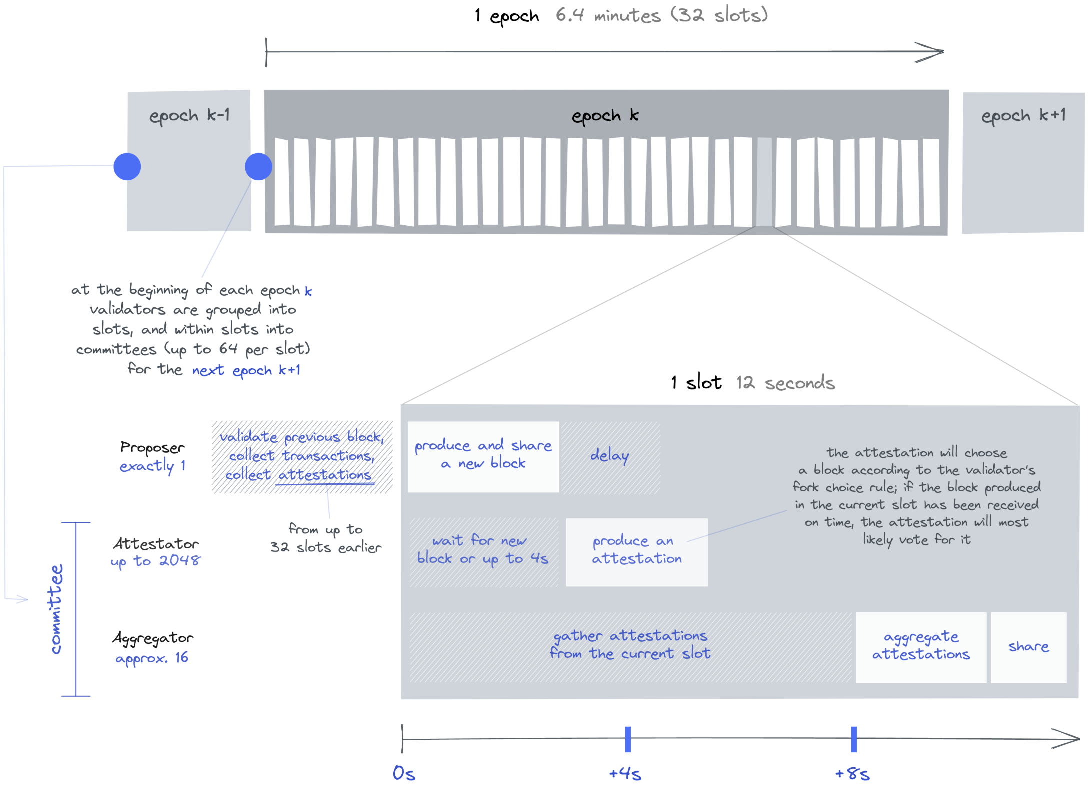
The **attesters** will wait for the block to be received, or up to 4 seconds from the start of the slot, and then make a decision on the head of the block.

[3] Since we're operating in a decentralised and trustless context, nobody is actually "selecting" a validator to propose a block. Instead, the validators will independently and deterministically derive their role based on the agreed state of the blockchain.

Depending on their *fork choice rule*, they may choose the newly created block, the block produced at the previous slot, or some other block. Within 8 seconds from the start of the slot, they will send their attestation to a sub-network where aggregators are listening for attestations.

Aggregators will gather all the attestations, and try to aggregate as many as possible into one. Before the end of the slot, the aggregators will broadcast the aggregated attestations with the network. These aggregated attestations (and possibly some unaggregated ones) will be then gathered by the proposer of the next slot, in a continuous loop.

The diagram in Figure **??** summarises the above timeline. These aggregated attestations (and possibly some unaggregated ones) will be then gathered by the proposer of the next slot, in a continuous loop.



1 epoch 6.4 minutes (32 slots)
epoch k-1
epoch k
epoch k+1
at the beginning of each epoch k validators are grouped into slots, and within slots into committees (up to 64 per slot) for the next epoch k+1
1 slot 12 seconds
Proposer
exactly 1
validate previous block, collect transactions, collect attestations
produce and share a new block
delay
the attestation will choose a block according to the validator's fork choice rule; if the block produced in the current slot has been received on time, the attestation will most likely vote for it
Attestator
up to 2048
from up to 32 slots earlier
wait for new block or up to 4s
produce an attestation
committee
Aggregator
approx. 16
gather attestations from the current slot
aggregate attestations
share
0s
+4s
+8s

Figure 1: Timeline for the work of proposers, attesters, and aggregators.

3 The Attestation Aggregation and Packing Problem

As covered in the previous section, one key responsibility of the proposer is to collect attestations from the network, bundle them into a block (along with transactions), and then share the newly minted block with the network. Because the space in a block is limited, the proposer faces the problem of deciding *which* attestations to include in their block. Moreover, because different attestations contribute differently to the efficient operation of the blockchain, and because some attestations have overlaps, the problem of choosing a set of attestations to include is an *optimisation* problem. The problem is known as the Attestation Aggregation and Packing Problem (hereinafter AAPP), and this section provides a more formal definition for it in terms of entities, constraints, and objectives.

The creation of a block B^s, where $s \in \mathbb{N}^{\geq 32}$ represents the index of the slot[4] at which the block is proposed, is unequivocally associated with one and only one instance of the AAPP. For this reason, and to simplify the notation, we will therefore treat the index s as implicit, and drop it from the problem definition.

3.1 Entities

Let V be the set of all **validators**, and A be the set of all **attestations** available to the proposer for inclusion in a given block B. Given an attestation $a \in A$, we define the following properties

- $d_a \in D$ the attestation data of the a,
- $V_a \subseteq V$ a set of attesters agreeing on d_a.

To keep the problem formulation general, we intentionally avoid specifying what attestation data d_a (and its domain D) looks like. Instead, we focus on what can be done with it, e.g., compute the epoch in which it was created[5].

Let the set $E \subseteq \mathbb{N}^{\geq 0}$ denote the epochs[6] of the blockchain, the **epoch** at which each attestation data $d_a \in D,\ a \in A$ was created[7] is a function

$$e : D \to E. \tag{1}$$

Note that A is assumed to satisfy a number of properties that reflect the context of the particular problem instance being considered. For instance, at-

[4] For convenience, we avoid considering the first 32 slots of the blockchain, as some special handling happens at the start of a blockchain.

[5] In other words, we can think of each element $d \in D$ as implementing some kind of `AttestationData` *trait*.

[6] For the purpose of solving the AAPP for a given block B, it is sufficient for E to include the epoch during which the block is being created and the one immediately before, however this doesn't change the present problem formulation.

[7] While the practicalities of computing the $e(d_a)$ function for a given d_a is not strictly relevant to the problem formulation, we can count on d_a carrying information about the slot $s \in \mathbb{N}^{\geq 32}$ in which that d_a was created. Under this assumption, $e(d_a) = \lfloor s/32 \rfloor$.

testations that are too old to be included in B are excluded from A. Moreover, it is assumed that

$$e(d_a) = e(d_b) \Rightarrow V_a \cap V_b = \emptyset, \quad \forall a, b \in A, \tag{2}$$

in other words a validator can produce at most one attestation data per epoch.

Each individual vote produced by an attester $v \in V$ at a given epoch $e \in E$ carries a **reward**. Such a reward is a function

$$r : E \times V \to \mathbb{N}. \tag{3}$$

This is the reward that the block proposer will collect by including the corresponding vote in the produced block. Like for $e(\cdot)$, the way $r(\cdot, \cdot)$ is computed is irrelevant to this problem formulation.

3.2 Solution

A solution S to the AAPP for the block B is a *set of subsets* of A. Formally

$$S = \{P \mid P \subseteq A\}. \tag{4}$$

Note that, according to the Ethereum consensus protocol, a block contains attestations, not *sets* of attestations. This particular formulation allows us to consider, as candidates for inclusion in B, not only attestations in A, but also attestations that can be *produced* from attestations in A by means of aggregation. In this sense, one can think of each set $P \in S$ as a set of attestations that are meant to be aggregated before being included in the proposed B. Of course, aggregation is subject to conditions, which we explicitly model as constraints.

3.3 Constraints

While the previous section defines the general structure of a solution S, a *feasible* solution needs to satisfy some additional constraints.

Aggregation constraints. For a solution S to yield a valid block, the attestations within each element of S must be compatible for aggregation. Recall that two attestations $a, b \in A$ can only be aggregated if their aggregation data is identical, and their sets of attesters are disjoint. More formally, the following two properties must hold

$$d_a = d_b \qquad \forall a, b \in P, P \in S \tag{5}$$

$$V_a \cap V_b = \emptyset \qquad \forall a, b \in P, P \in S. \tag{6}$$

Because each set of attestations in a feasible S represents a valid aggregated attestation, it is convenient to lift the properties of attestations introduced above to sets of attestations in the natural way. Specifically, let S be a feasible solution, and $P \in S$ be a set of attestations, we will denote by

$$V_P = \bigcup_{a \in P} V_a \tag{7}$$

the set of attesters of the aggregated attestation represented by P, and by

$$d_P \tag{8}$$

the attestation data shared by each attestation $a \in P$.

Capacity constraint. A feasible solution S must respect the following capacity constraint

$$|S| \leq N \tag{9}$$

where $N = 128$ according to the Ethereum consensus protocol at the time of writing. This corresponds to the maximum number of attestations that can be included in a newly proposed block.

3.4 Objective

The quality of a solution S is a function of the sets of attestations that are chosen to be part of it. Let

$$P_e = \{P \mid e(d_P) = e, P \in S\}, \ \ \forall e \in E \tag{10}$$

be a partition (by epoch) of the solution, and let denote by

$$V_{P_e} = \bigcup_{P \in P_e} V_P, \ \ \forall e \in E. \tag{11}$$

the set of attesters covered by each set $P \in S$. The quality of a solution can be then defined as

$$R(S) = \sum_{e \in E} \sum_{v \in V_{P_e}} r(e, v). \tag{12}$$

In other words, the quality of a solution is the sum of the rewards that can be collected for including attestations, and considering each attester at most once per epoch. Given this definition, the objective function of the AAPP is then

$$\textbf{maximise } R(S). \tag{13}$$

4 Existing algorithm

Currently, the Lighthouse client developed by Sigma Prime solves the above problem using a two-stage approach. The first stage deals with the aggregation of attestations. The second stage deals with the packing of attestations. Both stages rely on greedy heuristics.

4.1 Aggregation stage

The aggregation stage maintains a pool of attestations, and each time a new attestation is received, a greedy heuristic attempts to aggregate it with everything else in the pool. In case of success, the attestations already in the pool will be aggregated with the new attestation. In case of failure, the new attestation will be added to the pool as a separate attestation.

4.2 Packing stage

The packing stage starts with an empty block and, at each iteration, greedily includes the attestation that contributes most to the reward of the block. This algorithm is a well-known approximation algorithm for the Maximum Covering Problem with a bound on the number of sets to include (Maximum k-Coverage Problem).

The approximation guarantees of this algorithm on the original problem carry over to its application in the current context. Specifically, if the set of attestations to include is final, i.e., there isn't any further aggregation beyond the aggregation stage, the algorithm produces a packing whose value is guaranteed to be at least ≈ 0.632 of the optimal value [?].

5 Experimental analysis

This section describes the experimental analysis carried out to compare the quality of the solutions produced with the existing algorithm to the quality of the optimal solutions produced by an exact algorithm.

5.1 Instances

This experimental analysis is based on problem instances provided by Sigma Prime. Instances come in JSON format, and each instance must be interpreted in the context of a block proposal.

5.1.1 Structure

Below we provide a JSON-like representation of what the instances look like. We have taken the liberty to include comments (starting with `#`) and some loose typing to give some semantics to the fields. The basic types are

- `<slot_id>` the index of a slot,
- `<epoch_id>` the index of an epoch,
- `<attester_id>` the ID of an attester,
- `<number>` a number $n \in \mathbb{N}$.

all types except for `<number>`, which is represented as a JSON number, are JSON strings. For convenience, we also consider an `<attestation>` type, it is a JSON object with the following fields

- **attestation_indices** an array `[<attester_id>, ...]` of one (in case of an unaggregated attestation) or more (in case of an aggregated attestation) `<attester_id>`,
- **data** attestation data, a JSON object with the following fields

  - **slot** the `<slot_id>` during which this attestation data was created,
  - **other data** whose role, for the purpose of this problem formulation is to decide whether an attestation can be aggregated with another.

Each instance includes the following relevant fields

- **slot** a `<slot_id>` representing the slot at which the block will be proposed,
- **unaggregated_attestations** an object mapping `<slot_id>`s to arrays `[<attestation>, ...]` of unaggregated `<attestation>`s that are available to the proposer,
- **aggregated_attestations** an object mapping `<slot_id>`s to arrays `[<attestation>, ...]` of aggregated attestations that are available to the proposer,
- **reward_function** an object mapping `<epoch_id>`s to objects mapping `<attester_id>`s to `number`s. In other words, for each epoch, this field provides the reward for including the attestation of any given attester[8].

Below is a representation of the structure of an instance in a pseudo-JSON language. Note that the fields that are irrelevant to the problem formulation have been ignored.

```
{
  "slot": <slot_id>,
  "unaggregated_attestations": {
    <slot_id>: [
      [
        <attestation>
      ],
      # ...
    ]
  },
  "aggregated_attestations": {
    <slot_id>: [
      [
        <attestation>,
        # ...
      ],
      # ...
    ]
  },
  "reward_function": {
    <epoch_id>: {
      <attester_id>: <number>,
      # ... (reward assumed to be 0 if attester key is missing)
```

[8] The reward for including an attestation by an attester not included in this map is assumed to be zero.

```
    },
    # ... (should be limited to 2 epochs)
  }
}
```

Where each `<attestation>` is of the following form

```
{
  "attesting_indices": [
    <attester_id>,
    # ... (1 for unaggregated attestations, >1 for aggregated ones)

  ],
  "data": {
    "slot": <slot_id>,
    # ... (other data needed to check aggregation eligibility)
  }
}
```