

1 MiningProbabilityMultiStageProofOfWork - Technical Documentation

This package provides the functions to compute the mining probability of a miner, in a context which comprises $M \geq 2$ miners and $k \geq 1$ sequential hash-puzzles. It implements the expression 3 obtained in the paper "On multi-stage Proof-of-Work blockchain protocols: issues and challenges" (D'Arco, Ebadi Ansarodi, Mogavero).

Every subsection in this work describes a different function of the Mathematica package we implemented.

1.1 ComputeMiningProbability

This function computes the mining probability of the first one of M ($M \geq 2$) miners. The Proof-of-Work is composed of k ($k \geq 1$) sequential hash-puzzles. The function implicitly determines the values of M and k by parsing the input.

Input: $args_ = \{\{\lambda_{1,0}, \dots, \lambda_{1,k-1}\}, \dots, \{\lambda_{z,0}, \dots, \lambda_{z,k-1}\}, \dots, \{\lambda_{M,0}, \dots, \lambda_{M,k-1}\}\}$ is a set of positive real-parameters. Each sublist of the form $\{\lambda_{z,0}, \dots, \lambda_{z,k-1}\}$ is the set of λ -parameters relative to the hypoexponential distribution that describes the time miner z takes to complete the Proof-of-Work, $\forall z \in \{1, \dots, M\}$. More in details, $\lambda_{z,s}$ is the positive real parameter that describes the time that miner z takes to complete the hash-puzzle s , $\forall z \in \{1, \dots, M\}$ and $\forall s \in \{0, \dots, k-1\}$.

Output: the mining probability of the first miner (i.e, the miner with $z = 1$).

Local parameters used in MODULE.

- input: The input args to be syntax parsed.
- z: An integer which iterates the set $\{1, \dots, M\}$. It indicates the miner z .
- nStages: The number of sequential hash-puzzles a Proof-of-work is composed of.
- nMiners: The number of miners involved in the mining game.
- listsLambdaParameters: A list of minerZLambdaParameters. The list has length M . It contains a minerZLambdaParameters per miner.
- minerZLambdaParameters: Given $z \in \{1, \dots, M\}$, $minerZLambdaParameters = \{\lambda_{z,0}, \dots, \lambda_{z,k-1}\}$.
- couples: A list of minerZCouples. The list has length M . It contains a minerZCouples per miner.
- minerZCouples: Given $z \in \{1, \dots, M\}$, $minerZCouples$ is an association. We have that $minerZCouples = \langle |\beta_{z,1} \rightarrow r_{z,1}, \dots, \beta_{z,a_z} \rightarrow r_{z,a_z}| \rangle$.
- BminerZ: Given $z \in \{1, \dots, M\}$, $BminerZ$ is the B value relative to miner z . We have that $BminerZ = \beta_{z,1}^{r_{z,1}} \cdot \dots \cdot \beta_{z,a_z}^{r_{z,a_z}}$.
- BValues: A list of BminerZ. The list has length M . It contains a BminerZ per miner.
- aZ: Given $z \in \{1, \dots, M\}$, aZ is the number of distinct β -parameters relative to miner z .
- aValues: A list of aZ . The list has length M . It contains an aZ per miner.

- summationsResult: The result of the k^M iterations of the nested summations in expression 3 of the paper.
- currentParameters: It stores current values of $\{\{r_{1,q_1}, l_1, \beta_{1,q_1}\}, \dots, \{r_{M,q_M}, l_M, \beta_{M,q_M}\}\}$ in the current iteration of the nested summations. In this function, currentParameters is simply initialized to $\{\}$.
- miningProbability: It is the output value. The output is computed as follows:
miningProbability = BValues[[1]] · ... · BValues[[M]] · summationsResult.

1.2 CheckInputErrors

This function checks whether the user's input is syntactically correct.

Input: input = $\{\{\{\lambda_{1,0}, \dots, \lambda_{1,k-1}\}, \dots, \{\lambda_{z,0}, \dots, \lambda_{z,k-1}\}, \dots, \{\lambda_{M,0}, \dots, \lambda_{M,k-1}\}\}\}$.
Output: a Failure message if the output is syntactically not correct. Nothing otherwise.

1.3 ComputeSummationRecursive

This function computes the result of the K^M iterations of the nested summations in expression 3 of the paper. The output is computed recursively by using the array currentParameters, received as input from *ComputeMiningProbability*.

Consider z , currentParameters, nMiners, couples, aValues as described for the function *ComputeMiningProbability*. Let thetaCumulativeResult be a variable defined as $\text{thetaCumulativeResult} = \Phi'_{1,q_1 l_1}(\beta_{1,q_1}) \cdot \Psi'_{2,q_2 l_2} \cdot \dots \cdot \Psi'_{M,q_M l_M}(\beta_{M,q_M})$.

The recurrence relationship we used to compute the output of ComputeSummationRecursive is the following:

ComputeSummationRecursive[z, nMiners, couples, aValues, currentParameters, thetaCumulativeResult] =

$$= \left\{ \begin{array}{ll} \sum_{q_1=1}^{a_1} \sum_{l_1=1}^{r_{1,q_1}} \text{ComputeSummationRecursive}[2, \text{nMiners}, \text{couples}, \text{aValues}, \text{currentParameters}[[1]] = \{r_{1,q_1}, l_1, \beta_{1,q_1}\}, \Phi'_{1,q_1 l_1}(\beta_{1,q_1})] & \text{iff } z = 1 \\ \sum_{q_z=1}^{a_z} \sum_{l_z=1}^{r_{z,q_z}} \text{ComputeSummationRecursive}[z+1, \text{nMiners}, \text{couples}, \text{aValues}, \text{currentParameters}[[z]] = \{r_{z,q_z}, l_z, \beta_{z,q_z}\}, \Psi'_{z,q_z l_z}(\beta_{z,q_z}) \cdot \text{thetaCumulativeResult}] & \text{iff } 2 \leq z \leq M-1 \\ \sum_{q_M=1}^{a_M} \sum_{l_M=1}^{r_{M,q_M}} \left(\Psi'_{z,q_z l_z}(\beta_{z,q_z}) \cdot \text{thetaCumulativeResult} \cdot \text{multinomialCoefficient} / \text{iterationDenominator} \right) & \text{iff } z = M \end{array} \right\}$$

Regarding the third case, we have:

A. For each possible value of q_M and l_M , `currentParameters`[[M]] is set as the following:
`currentParameters`[[M]] = $\{r_{M,q_M}, l_M, \beta_{M,q_M}\}$.

B. For each possible value of q_M and l_M , multinomialCoefficient = $\frac{\left(\sum_{z=1}^M (r_{z,q_z} - l_z)\right)!}{\prod_{z=1}^M ((r_{z,q_z} - l_z)!)}.$

C. For each possible value of q_M and l_M , iterationDenominator = $\left(\sum_{z=1}^M \beta_{z,q_z}\right)^{1+\sum_{z=1}^M (r_{z,q_z} - l_z)}.$

In A., B. and C., the current iteration values of r_{z,q_z} , l_z , β_{z,q_z} , $\forall z \in \{1, \dots, M\}$ are retrieved from the array `currentParameters`.

The function `ComputeSummationRecursive` is invoked for the first time by the function `ComputeMiningProbability`. The function call is:

```
summationsResult = ComputeSummationRecursive[1, nMiners, couples,
aValues, currentParameters, null];
```

1.4 GroupEqualLambdas

Let $z \in \{1, \dots, M\}$. Given as input a set $\{\lambda_{z,0}, \dots, \lambda_{z,k-1}\}$, this function groups together the positive real λ -parameters which have the same value. The function stores also the multiplicity (i.e. the number of occurrences) of each unique λ -parameter.

The output is an association of the form $\langle |\beta_{z,1} \rightarrow r_{z,1}, \dots, \beta_{z,a_z} \rightarrow r_{z,a_z}| \rangle$, where a_z is the number of distinct λ -parameters relative to the hypoexponential distribution of miner z . At the same time, β_{z,q_z} is a unique λ -parameter and the positive integer r_{z,q_z} is its multiplicity, $\forall q_z \in \{1, \dots, a_z\}$.

1.5 B

Let $z \in \{1, \dots, M\}$. Given as input a_z and an association of the form $\langle |\beta_{z,1} \rightarrow r_{z,1}, \dots, \beta_{z,a_z} \rightarrow r_{z,a_z}| \rangle$, this function returns as output the value $B_{minerZ} = \beta_{z,1}^{r_{z,1}} \cdot \dots \cdot \beta_{z,a_z}^{r_{z,a_z}}$.

1.6 PhiPrime

This function computes $\Phi'_{1,q_1 l_1}(t)$. The relevant inputs of this function are \underline{t} , the variables q_1 , a_1 and l_1 , and the variable `miner1Couples`. The variable `miner1Couples` in this function is equal to the association $\langle |\beta_{1,1} \rightarrow r_{1,1}, \dots, \beta_{1,a_1} \rightarrow r_{1,a_1}| \rangle$ relative to the first miner. Following the expression 3 in the paper, $\Phi'_{1,q_1 l_1}(\beta_{1,q_1})$ is computed as follows:

$$\Phi'_{1,q_1 l_1}(t) = (-1)^{l_1-1} \cdot \text{summationResultPhi}.$$

Here, summationResultPhi = $\sum_{\Omega_{2_1}(1)} \prod_{j_1} \binom{i_{j_1} + r_{1,j_1} - 1}{i_{j_1}} \cdot \tau_{j_1}$. See function *RecursivelyComputeSumPhiPrimeWithArray* for further details on the computation of `summationResultPhi`.

1.7 RecursivelyComputeSumPhiPrimeWithArray

This function computes recursively the value of `summationResultPhi`, following this definition $\underline{\text{summationResultPhi}} = \sum_{\Omega_{2_1}(1)} \prod_{j_1} \binom{i_{j_1} + r_{1,j_1} - 1}{i_{j_1}} \cdot \tau_{j_1}$.

The relevant input parameters are \underline{t} , the variables q_1 , a_1 and l_1 , `miner1Couples`, an array, named `array`, having `arraySize` equal to $a_1 - 1$, and a positive integer `amountToAssign` (such that $1 \leq \text{amountToAssign} \leq l_1 - 1$). In the first recursive step, `array` is an array of zeros.

The definition of $\Omega_{2_1}(1)$ is $\Omega_{2_1}(1) = \sum_{\substack{j_1=1 \\ j_1 \neq q_1}}^{a_1} i_{j_1} = l_1 - 1 : i_{j_1} \in \mathbb{N}_0 \forall j_1$. Following the definition, it holds that there are $a_1 - 1$ variables of the following type: i_{j_1} ; that is, one variable for each possible value of the variable j_1 . j_1 spans any value in the set: $S = \{w : 1 \leq w \leq a_1 \text{ and } w \neq q_1\}$. Each one of the variables of the form i_{j_1} must be a non negative integer, and their sum must be equal to $l_1 - 1$.

Following the definition of $\Omega_{2_1}(1)$, every iteration of the summation corresponds *bijectively* to a distinct assignment of the variables i_{j_1} . In order to consider every possible combination of values for the variables i_{j_1} , we work recursively. In the recursive steps, we use the array `array` to store the current values of the variables, and the non-negative integer variable `amountToAssign` to store the amount which has not been already assigned to one of the variables i_{j_1} . The cell v ($v \geq 1$) in the array stores the value of i_{j_1} . We have that $j_1 < q_1 \ ? \ v = j_1 : v = j_1 - 1$.

Given these considerations, the recurrence relationship we used to compute the output of `summationResultPhi` is the following:

`RecursivelyComputeSumPhiPrimeWithArray`[v , q_1 , t , `arraySize`, `miner1Couples`, `amountToAssign`, `array`, a_1] =

$$= \left\{ \begin{array}{ll} \text{PhiProduct}[\text{array}[[v]] = c, q_1, t, \text{arraySize}, \text{miner1Couples}, a_1] & \text{iff } v = \text{arraySize} \\ \text{PhiProduct}[\text{array}[[idx]] = 0 \forall \text{idx} : v \leq \text{idx} \leq \text{arraySize}, & \\ q_1, t, \text{arraySize}, \text{miner1Couples}, a_1] & \text{iff } v \leq \text{arraySize} \\ & \text{and amountToAssign} = 0 \\ \sum_{c=0}^{\text{amountToAssign}} \text{RecursivelyComputeSumPhiPrimeWithArray}[& \\ v + 1, q_1, t, \text{arraySize}, \text{miner1Couples}, \text{amountToAssign} - c, & \\ \text{array}[[v]] = c, a_1] & \text{iff } 1 \leq v \leq \text{arraySize} - 1 \\ & \text{and amountToAssign} > 0 \end{array} \right\}$$

In the first two cases, we invoke the function *PhiProduct*. The latter function computes $\prod_{j_1} \binom{i_{j_1} + r_{1,j_1} - 1}{i_{j_1}} \cdot \tau_{j_1}$ for the current iteration of the summation (i.e. the current values inside `array`).

Finally, the function `RecursivelyComputeSumPhiPrimeWithArray` is invoked for the first time inside the function `PhiPrime`. The function call is:

```
summationResultPhi = RecursivelyComputeSumPsiPrimeWithArray [1 ,
q1 , t , a1 , miner1Couples , l1 - 1 , array , a1 ];
```

1.8 PhiProduct

This function computes $\prod_{j_1} \binom{i_{j_1} + r_{1,j_1} - 1}{i_{j_1}} \cdot \tau_{j_1}$. The inputs of this function are \underline{t} , the variables q_1 , a_1 , `miner1Couples`, `array` and `arraySize`.

Following the expression 3 in the paper, τ_{j_1} is computed as follows:
 $\tau_{j_1} = (\beta_{1,j_1} + t)^{-(r_{1,j_1} + i_{j_1})}$.

For every j_1 in the set $S = \{w : 1 \leq w \leq a_1 \text{ and } w \neq q_1\}$, this function retrieves the values of i_{j_1} from the variable `array`. At the same time, for every j_1 in the set $S = \{w : 1 \leq w \leq a_1 \text{ and } w \neq q_1\}$ this function retrieves the values of r_{1,j_1} and β_{1,j_1} from the variable `miner1Couples`.

1.9 PsiPrime

Let $z \in \{2, \dots, M\}$. This function computes $\Psi'_{z,q_z l_z}(t)$. The relevant inputs of this function are \underline{t} , the variables q_z , a_z and l_z , and the variable `minerZCouples`. The variable `minerZCouples` in input to this function is equal to the association $\langle |\beta_{z,1} \rightarrow r_{z,1}, \dots, \beta_{z,a_z} \rightarrow r_{z,a_z}| \rangle$ relative to the miner z .

The computation proceeds by prepending a *fictious* new couple as the new first couple in the variable `minerZCouples`. Indeed, the couple $\beta_{z,0} \rightarrow r_{z,0}$ is added to `minerZCouples`, with $\beta_{z,0} = 0$ and $r_{z,0} = 1$. Consequently, by following the expression 3 in the paper, $\Psi'_{z,q_z l_z}(\beta_{z,q_z})$ is computed as follows:

$$\Psi'_{z,q_z l_z}(t) = -(-1)^{l_z-1} \cdot \text{summationResultPsi}.$$

Here, $\text{summationResultPsi} = \sum_{\Omega_{2z}(0)} \prod_{j_z} \binom{i_{j_z} + r_{z,j_z} - 1}{i_{j_z}} \cdot \tau_{j_z}$. See function *RecursivelyComputeSumPsiPrimeWithArray* for further details on the computation of `summationResultPsi`.

1.10 RecursivelyComputeSumPsiPrimeWithArray

Let $z \in \{2, \dots, M\}$. This function computes recursively the value of `summationResultPsi`, following this definition $\text{summationResultPsi} = \sum_{\Omega_{2z}(0)} \prod_{j_z} \binom{i_{j_z} + r_{z,j_z} - 1}{i_{j_z}} \cdot \tau_{j_z}$.

The relevant input parameters are \underline{t} , the variables q_z , a_z and l_z , `minerZCouples`, an array, named `array`, having `arraySize` equal to a_z , and a positive integer `amountToAssign` (such that $1 \leq \text{amountToAssign} \leq l_z - 1$). In the first recursive step, `array` is an array of zeros.

The definition of $\Omega_{2_z}(0)$ is $\Omega_{2_z}(0) = \sum_{\substack{j_z=0 \\ j_z \neq q_z}}^{a_z} i_{j_z} = l_z - 1 : i_{j_z} \in \mathbb{N}_0 \forall j_z$. Following the definition, it holds that there are a_z variables of the following type: i_{j_z} ; that is, one variable for each possible value of the variable j_z . j_z spans any value in the set: $S = \{w : 0 \leq w \leq a_z \text{ and } w \neq q_z\}$. Each one of the variables of the form i_{j_z} must be a non negative integer, and their sum must be equal to $l_z - 1$.

Following the definition of $\Omega_{2_z}(0)$, every iteration of the summation corresponds *bijectively* to a distinct assignment of the variables i_{j_z} . In order to consider every possible combination of values for the variables i_{j_z} , we work recursively. In the recursive steps, we use the array `array` to store the current values of the variables, and the non-negative integer variable `amountToAssign` to store the amount which has not been already assigned to one of the variables i_{j_z} . The cell v ($v \geq 1$) in the array stores the value of i_{j_1} . We have that $j_z < q_z ? v = j_z + 1 : v = j_z$.

Given these considerations, the recurrence relationship we used to compute the output of `summationResultPsi` is the following:

`RecursivelyComputeSumPsiPrimeWithArray[v, q_z, t, arraySize, minerZCouples, amountToAssign, array, a_z] =`

$$= \left\{ \begin{array}{ll} \text{PsiProduct}[\text{array}[[v]] = c, q_z, t, \text{arraySize}, \text{minerZCouples}, a_z] & \text{iff } v = \text{arraySize} \\ \text{PsiProduct}[\text{array}[[idx]] = 0 \forall idx : v \leq idx \leq \text{arraySize}, & \\ q_z, t, \text{arraySize}, \text{minerZCouples}, a_z] & \text{iff } v \leq \text{arraySize} \\ & \text{and amountToAssign} = 0 \\ \sum_{c=0}^{\text{amountToAssign}} \text{RecursivelyComputeSumPsiPrimeWithArray}[& \\ v + 1, q_z, t, \text{arraySize}, \text{minerZCouples}, \text{amountToAssign} - c, & \\ \text{array}[[v]] = c, a_z] & \text{iff } 1 \leq v \leq \text{arraySize} - 1 \\ & \text{and amountToAssign} > 0 \end{array} \right\}$$

In the first two cases, we invoke the function *PsiProduct*. The latter function computes $\prod_{j_z} \binom{i_{j_z} + r_{z,j_z} - 1}{i_{j_z}} \cdot \tau_{j_z}$ for the current iteration of the summation (i.e. the current values inside `array`).

Finally, the function `RecursivelyComputeSumPsiPrimeWithArray` is invoked for the first time inside the function *PsiPrime*. The function call is:

```
summationResultPsi = RecursivelyComputeSumPsiPrimeWithArray[1,
q_z, t, a_z, minerZCouples, l_z - 1, array, a_z];
```

1.11 PsiProduct

Let $z \in \{2, \dots, M\}$. The code, the inputs, and the documentation of this function are the similar to the function *PhiProduct*. The only difference is that in *PsiProduct* we consider the subscript index z instead of 1.