UNIVERSIDADE FEDERAL DO RIO DE JANEIRO

INSTITUTO DE COMPUTAÇÃO

CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

LUCCA MARTINS FELIX

APLICAÇÃO DE ALGORITMOS CERTIFICADORES E VERIFICADORES PARA OTIMIZAÇÃO DE CONTRATOS INTELIGENTES

RIO DE JANEIRO

2023

LUCCA MARTINS FELIX

APLICAÇÃO DE ALGORITMOS CERTIFICADORES E VERIFICADORES PARA OTIMIZAÇÃO DE CONTRATOS INTELIGENTES

Trabalho de conclusão de curso de graduação apresentado ao Departamento de Ciência da Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Vinícius Gusmão Pereira de Sá, D.Sc.

RIO DE JANEIRO

2023

CIP - Catalogação na Publicação

Felix, Lucca Martins
F316a Aplicação de algoritmos certificadores e verificadores para otimização de contratos inteligentes / Lucca Martins Felix. -- Rio de Janeiro, 2023.
50 f.

Orientador: Vinícius Gusmão Pereira de Sá.
Trabalho de conclusão de curso (graduação) - Universidade Federal do Rio de Janeiro, Instituto de Computação, Bacharel em Ciência da Computação, 2023.

1. Blockchain. 2. Contrato-Inteligente. 3. Otimização. I. Sá, Vinícius Gusmão Pereira de, orient. II. Título.

Elaborado pelo Sistema de Geração Automática da UFRJ com os dados fornecidos pelo(a) autor(a), sob a responsabilidade de Miguel Romeu Amorim Neto - CRB-7/6283.

LUCCA MARTINS FELIX

APLICAÇÃO DE ALGORITMOS CERTIFICADORES E VERIFICADORES PARA OTIMIZAÇÃO DE CONTRATOS INTELIGENTES

Trabalho de conclusão de curso de graduação apresentado ao Departamento de Ciência da Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 19 de janeiro de 2023 .

BANCA EXAMINADORA:

Prof. Vinícius Gusmão Pereira de Sá, D.Sc.

Documento assinado digitalmente
gov.br
VINICIUS GUSMAO PEREIRA DE SA
Data: 13/02/2023 13:05:51-0300
Verifique em https://verificador.iti.br

Prof. Mitre Costa Dourado, D.Sc.

Documento assinado digitalmente
gov.br
MITRE COSTA DOURADO
Data: 15/02/2023 19:55:30-0300
Verifique em https://verificador.iti.br

Prof. Daniel Sadoc Menasché, D.Sc.

Documento assinado digitalmente
gov.br
DANIEL SADOC MENASCHE
Data: 15/02/2023 10:54:43-0300
Verifique em https://verificador.iti.br

AGRADECIMENTOS

Quero agradecer, em primeiro lugar, à minha família que sempre esteve ao meu lado durante toda a jornada, em especial à minha mãe Leila e ao meu irmão Luan.

Agradeço também à todos os meus amigos que me acompanharam durante esse trajeto, incluindo aqueles de longa data e também aqueles que fiz durante esse caminho. Não deixo de incluir todos os meus amigos e colegas de trabalho, que exerceram um forte papel em minha posição enquanto profissional de tecnologia.

Por fim, gostaria de agradecer, com imensa admiração, todos professores que me acompanharam durante a graduação, me enriquecendo de conhecimento e abrindo a visão do que é o mundo.

RESUMO

A tecnologia da blockchain está permitindo novas formas de se desenhar uma solução descentralizada, onde o objetivo principal é estabelecer confiança entre partes, em um ambiente aberto e sem que haja a necessidade de uma autoridade centralizada para fazê-la. No mais, contratos inteligentes tornam possível que códigos programáveis sejam executados dentro desta rede blockchain. O custo para que esses contratos sejam executados nela é medido em "gas" (do inglês, gasolina) que baseia-se na quantidade de poder computacional exigido da rede blockchain durante a execução. Ao fim, o preço do "gas" possui uma equivalência ao dinheiro corrente, tal qual o real brasileiro e o dólar americano. Com isso, o objetivo deste trabalho é a redução de custos, associados a execução de contratos inteligentes na blockchain, por meio da otimização do "gas" consumido. Para essa otimização, serão aplicados os conceitos de algoritmos certificadores e verificadores de forma que, assim, seja possível terceirizar ao máximo o poder computacional para fora da blockchain e que, com isso, minimize-se a quantidade de recursos exigidos dentro da mesma.

Palavras-chave: Blockchain; Contrato-Inteligente; Otimização.

ABSTRACT

Blockchain technology is enabling new ways of designing a decentralized solution, where the main objective is to establish trust between parties, in an open environment and without the need for a centralized authority to do so. Furthermore, smart contracts make it possible for programmable code to run within this blockchain environment. The cost for these contracts to run on the network is measured in "gas" which is based on the amount of computing power required of the blockchain network during execution. In the end, the price of "gas" is equivalent to current money, such as the Brazilian real and the US dollar. Thus, the objective of this work is to reduce costs associated with the execution of smart contracts on the blockchain, through the optimization of the "gas" consumed. For this optimization, the concepts of certifying and verifier algorithms will be applied so that, in this way, it is possible to outsource as much as possible the computational power outside the blockchain and that, with this, the amount of resources required within it is minimized.

Keywords: Blockchain; Smart-Contract; Optimization.

SUMÁRIO

1 INTRODUÇÃO

A blockchain [18] trouxe consigo um nova forma de estabelecer confiança em sistemas distribuídos de computação, por meio da possibilidade de torná-los descentralizados. Ao utilizar um sistema totalmente centralizado de terceiros, sempre há a necessidade de depositar confiança na corretude deste. Pode-se citar, por exemplo, a confiança que clientes precisam ter no armazenamento correto de seus saldos bancários em uma instituição financeira, visto que, caso haja alguma inconsistência nestes dados, será muito difícil para esses clientes provarem a inconsistência em seus respectivos saldos. Por isso, a blockchain, ao permitir que entidades diversas façam parte de um mesmo sistema onde só registra-se aquilo que foi computado em comum acordo da maioria, cria um sistema onde a confiança está distribuída não em uma, mas sim em várias entidades, caracterizando-se, assim, como sistema descentralizado.

Dentre as blockchains mais populares, a rede Ethereum [2] inovou ao permitir que códigos programáveis fossem executados dentro de sua rede. Estes códigos são conhecidos como contratos inteligentes e ficam armazenados dentro da blockchain, onde qualquer usuário tem acesso para ler e interagir com os mesmos dentro da rede. Dessa forma, esses contratos permitem que duas ou mais partes possam estabelecer transações de complexidade arbitrária sem a necessidade de terceiros e sem a necessidade de confiar nas partes envolvidas, reduzindo essa preocupação somente para corretude do código do contrato utilizado nesta transação.

Algo muito importante para se destacar entre os aspectos de um contrato inteligente é a sua medida de consumo computacional, conhecida como “gas” [5]. É com esta medida que as entidades participantes dessa rede descentralizada conseguem cobrar uma taxa pela execução do contrato na rede. Assim, quanto maior é o “gas”, maior será o custo de execução do mesmo. Por conta deste viés econômico, há muito interesse nos desenvolvedores de contratos inteligentes em buscarem otimizar o custo de execução de seus códigos sem que isso comprometa a corretude de execução dos mesmos [8].

Neste trabalho, estamos propondo uma técnica de otimização onde uma parte da execução é realizada fora da blockchain e a outra parte é executada dentro dela. Em geral, o grande problema desta proposta está relacionada em garantir a corretude dessa execução, visto que, em termos de confiança, a única entidade computacional na qual todos podem confiar é a blockchain. Logo, ao executar parte do código fora da mesma acresce-se um risco e talvez a indesejada necessidade de se confiar na corretude de execução dessa outra entidade computacional externa. Para resolver isso, o conceito de algoritmos certificadores e verificadores [11] parece prover uma solução promissora, como iremos argumentar.

Algoritmos certificadores, além de produzirem em sua saída o resultado, trazem também um certificado sobre esse resultado que permite avaliar se o mesmo está correto. Ao avaliá-lo com um algoritmo verificador, é possível então garantir que o algoritmo executou corretamente. Em sua maioria, algoritmos verificadores possuem complexidade computacional inferior à complexidade do algoritmo que diretamente produz a saída. Dessa forma, reduziremos a complexidade computacional dentro da blockchain ao substituir um algoritmo que diretamente produz a saída, por um algoritmo verificador. Assim, a tarefa de executar o algoritmo ocorrerá fora da blockchain e, dentro da mesma, somente haverá a tarefa de avaliar a corretude dos resultados.

Na Seção 2, iniciaremos aprofundando sobre a blockchain e os contratos inteligentes, para que seja possível entender sua utilidade e os desafios que os permeiam. A Seção 3 introduz os algoritmos certificadores e verificadores, assim como contextualiza a utilização dos mesmos para a otimização de custos na blockchain. Já a Seção 4 apresenta o formato do problema que queremos resolver, definindo as metodologias e esquemas utilizados para avaliar os resultados. Tais esquema e metodologias são utilizadas em quatro diferentes problemas, que são analisados na Seção 5. Por fim, na Seção 6, apresentamos nossas conclusões e perspectivas de trabalhos futuros.

2 BLOCKCHAIN E CONTRATOS INTELIGENTES

2.1 BLOCKCHAIN E A REDE ETHEREUM

Em 2008, um autor desconhecido, que se auto-intitulou “Satoshi Nakamoto”, publicava o artigo do Bitcoin [14], a primeira e maior rede blockchain para gerenciamento de dinheiro digital. Tratava-se de uma rede que funciona até hoje sem a interferência de nenhuma entidade “responsável” ou “detentora” da mesma. Esse conceito é levado tão a sério que ainda nos dias de hoje, 15 anos após seu lançamento, ninguém sabe dizer quem é o autor que a desenvolveu.

Do inglês, blockchain é uma cadeia sequencial de blocos [1] com informações, nos remetendo a um tipo de banco de dados onde cada novo registro é amarrado criptograficamente ao anterior. Com isso, a tecnologia garante a imutabilidade de registros antigos, por conta dessa correlação entre blocos. No mais, esse formato sequencial de blocos permite que uma blockchain possua seus registros distribuídos em uma rede descentralizada de ponta a ponta.

Nessa rede, cada participante é responsável por manter uma cópia de todos os registros e também validar toda nova proposta de blocos a serem inseridos, verificando se as novas informações são corretas e se condizem com as regras daquela rede. Esses participantes são mais conhecidos como mineradores [7], e oferecem o seu poder computacional para a rede em troca das taxas que a mesma arrecada com novas inserções de registros. Dessa forma, os usuários que desejam inserir registros na rede precisam pagar por cada operação para que, assim, os mineradores sejam recompensados.

Como a rede Bitcoin era limitada a somente transferir seu ativo base, o Bitcoin, entre contas distintas, outras tecnologias surgiram para suprirem novas necessidades. Com isso, a rede Ethereum, que hoje é a segunda maior blockchain em capitalização de mercado, ascendeu por conta de seu diferencial para a época: suportar códigos programáveis como forma de instruções. Assim, essa blockchain não serve apenas como um registro distribuído e descentralizado de saldos e transferências, como

também permite que aplicações descentralizadas sejam desenvolvidas com qualquer conjunto arbitrário de regras. Essas aplicações descentralizadas também são conhecidas como contratos inteligentes.

2.2 CONTRATOS INTELIGENTES

Hoje existentes em várias blockchains, os contratos inteligentes são códigos programáveis que residem dentro da rede descentralizada e que, geralmente, permitem algum tipo de interação dos clientes com a rede. Mais especificamente na rede Ethereum, podemos dizer que o contrato inteligente é composto por uma coleção de códigos (seu métodos) e um conjunto de dados que representa o seu estado, tudo isso alocado em um endereço específico para ele dentro da blockchain. Uma boa analogia para isso é pensarmos na blockchain como um grande banco de dados descentralizado; nela os contratos inteligentes são um espaço reservado de armazenamento de informações. Nesse espaço há um conjunto de regras pré-determinadas sobre como deve-se armazenar e ler essas informações. Essas regras são escritas em Solidity [15], uma linguagem Turing-completa [13] desenvolvida especificamente para contratos inteligentes.

Com toda a capacidade de uma linguagem Turing-completa em mãos e um ambiente descentralizado onde não há o risco de terceiros na execução de um código, desenvolver um contrato inteligente em Solidity na rede Ethereum torna-se uma solução interessante para muitos problemas. Entre eles, formalizações de relações, tais quais os contratos de cunho jurídico. Por exemplo, se duas partes X e Y desejam trocar seus respectivos ativos A e B, estes podem realizar essa operação por meio de um contrato inteligente de troca. Neste contrato deve haver alguma condição que, para efetuar a troca, garanta-se primeiro que ambos possuem os respectivos ativos e que estão disponíveis para serem transferidos, efetuando assim a operação ou cancelando-a caso a condição exigida não seja atendida. Muitas outras soluções também já existem dentro da blockchain, tais como representação de ativos digitais fungíveis e não fungíveis, plataformas de empréstimo descentralizadas, corretoras para trocas de ativo com liquidez e muitas outras soluções ainda estão por vir.

2.3 CONSUMO DE “GAS”

O “gas” dentro da rede Ethereum é a unidade de medida utilizada para precificar o custo de uma transação dentro da blockchain. Este custo existe para que os mineradores sejam recompensados financeiramente pelo poder computacional oferecido à rede. Como trabalhado na dissertação de Michael Kong [6], o contrato Solidity após compilado se transforma em um conjunto instruções de máquinas, muito semelhantes ao assembly, específicas para essa blockchain (conhecidas como “OP CODES”). Cada um desses “OP CODES”, possuem um custo associado [17]. Alguns com valor fixo, tal qual o “OP CODE” *ADD* que possui um custo de 3 “gas” e outros com custo determinado pelos dados utilizados na própria instrução.

Para interagir com a rede por meio de contratos inteligentes existem dois tipos de métodos que um cliente é capaz de chamar: os métodos puros e os métodos não-puros. Basicamente, métodos puros não alteram o estado da blockchain e, por conta disso, não possuem custos já que não é necessário que seja feita uma transação. Esses métodos, em sua maioria, servem apenas para que os dados dentro daquele contrato sejam lidos por qualquer um que deseje. Já os métodos não-puros, ao contrário, realizam alterações no estado da blockchain por meio de “OP CODES” dentro de suas instruções que inserem, excluem ou alteram dados da blockchain. Para esses, é necessário que o cliente efetue uma transação na blockchain, onde o mesmo pagará as taxas de rede envolvidas com a execução. Observe que, mesmo que haja instruções (“OP CODES”) que não alterem estado da blockchain dentro do método não-puro, ainda assim esses métodos entrarão no cálculo de “gas” a ser pago, visto que os resultados dessas instruções podem ser usados nas instruções de registro e, por conta disso, precisam ser executados e validados por todos os mineradores da rede.

Com esse contexto, é possível perceber que os problemas reais nos quais a nossa solução será útil não são meramente relacionados a descobrir o resultado de um problema para uma dada entrada, visto que isso se encaixaria na categoria de métodos puros. Na verdade, o que queremos é permitir que a blockchain realize o menor trabalho possível para garantir que o resultado de um problema, dada uma entrada,

foi obtido corretamente dentro de um método não-puro para ser usado em outros fins, tais quais armazenar esse resultado ou utilizá-lo dentro do contrato inteligente para alguma outra tomada de decisão que configure-se em um método não-puro.

2.4 LIMITAÇÕES

Por mais que a linguagem Solidity seja Turing-Completa, existem várias restrições relacionadas ao ambiente da blockchain Ethereum que impedem muitos problemas de serem resolvidos por lá. Em primeiro lugar, por mais que o custo computacional do "gas" seja alto, mesmo que alguém estivesse disposto a custear, ainda assim existe um limite máximo de "gas" para cada transação (conforme discutido no trabalho de Michael Kong [6]). O que significa dizer que algoritmos de alta complexidade podem, muitas vezes, nem serem possíveis de serem executados dentro da blockchain, até mesmo para pequenas entradas.

No mais, vale mencionarmos que as operações aritméticas possuem algumas limitações tais como a ausência de números em ponto flutuante de maneira nativa junto com o tamanho máximo de inteiros que restringem-se a 256 bits. Para que sejam feitas operações aritméticas fora dessas restrições é necessário recorrer a alguma solução que não seja nativa e que, naturalmente, irá requerer um alto custo de "gas" em cima das operações. Outra operação que vale ser destacada como limitada é a geração de números pseudo-aleatórios, visto que, por se tratar de um ambiente onde há o requisito do consenso, todas as máquinas precisam estar de comum acordo. Por conta disso, qualquer algoritmo que precise de um número aleatório precisa ser adaptado para a utilização de um número determinístico.

3 ALGORITMOS CERTIFICADORES E VERIFICADORES

Desenvolvidos em torno do problema de corretude nas implementações de algoritmos em geral, os algoritmos certificadores [11] são responsáveis não só por produzirem uma saída y, como também um certificado w. Esse certificado permite que o resultado seja verificado de uma outra forma, não apenas re-executando o algoritmo original[1], tornando possível que a corretude do resultado seja atestada por um algoritmo verificador que, por sua vez, é quase certamente isento de erros. Tal isenção se verifica facilmente, dada a natureza quase trivial da verificação, por meio de simples inspeção de código.

Assim, o algoritmo verificador [9] recebe a entrada x do algoritmo certificador junto com a saída y e do certificado w produzidos pelo mesmo (veja o esquema em Algoritmo 1). Com esses três valores, ele será capaz de avaliar se a saída y para aquela entrada x é correta. Em geral, espera-se que a implementação de algoritmos verificadores seja mais simples e também computacionalmente mais eficiente que a de seus respectivos certificadores e algoritmos originais (ou seja, as soluções clássicas que apenas retornam o resultado y, sem a necessidade de produzir w). E é exatamente em cima dessa eficiência dos algoritmos verificadores que este trabalho foi desenvolvido.

Como foi discutido na Subseção 2.1, a única entidade computacional em quem devemos confiar, quando falamos de uma solução totalmente descentralizada, é a própria blockchain. Dessa forma, para um algoritmo qualquer que precise ser calculado pela blockchain, mesmo sendo economicamente mais eficiente receber o resultado já pronto vindo de fora da blockchain, há sempre o risco desse resultado ter sofrido alguma adulteração por quem o enviou. Por isso, o contrato inteligente dentro da blockchain deve sempre garantir a corretude de todas as operações realizadas nas interações com ele.

[1] Repetir a execução do algoritmo pode ajudar a identificar e mitigar erros durante a execução, ou diminuir a probabilidade de erro em algumas classes de algoritmos randomizados, mas não tem qualquer valia contra erros na própria implementação.

Algoritmo 1 Certificador e Verificador

1: **função** CERTIFICADOR(x): Retorna (y, w) onde y é a saída do algoritmo e w o seu certificado
2: ...
3: **fim função**
4:
5: **função** VERIFICADOR(x, y, w): Retorna Verdadeiro se y for a saída correta para x com o auxílio de w para verificar, senão retorna Falso
6: ...
7: **fim função**
8:
9: **função** EXECUTEEVERIFIQUE(x): Retorna a saída y somente se verificada, senão emite um erro
10: $(y, w) \leftarrow$ Certificador(x)
11: **se** Verificador(x, y, w) **então**
12: **devolve** y
13: **senão**
14: **devolve** Erro de implementação
15: **fim se**
16: **fim função**

Com essa necessidade de assegurar a exatidão dos cálculos, associada à eficiência dos algoritmos verificadores, veremos como é possível otimizar os custos das operações na blockchain terceirizando os cálculos para fora dela sem o risco de sofrer uma adulteração nos resultados. Para isso, o algoritmo certificador será executado no computador do cliente e, ao enviar o resultado y junto ao certificado w, o contrato inteligente será capaz de garantir a corretude por meio do algoritmo verificador (que espera-se ser computacionalmente mais eficiente).

4 PROBLEMA E METODOLOGIA

4.1 ESTRUTURA DO PROBLEMA

Iremos confrontar a utilização de algoritmos verificadores dentro da blockchain contra algoritmos que solucionam todo o problema dentro da mesma. Dessa forma, o trabalho baseia-se na análise de um conjunto de problemas algorítmicos no qual queremos verificar se a nossa proposta reduzirá os custos de "gas" envolvidos com uma transação.

Para que essa análise seja efetuada, conforme diagramado na Figura 1, iremos realizar até 4 implementações. Sendo elas:

- $Soluciona(X)$: Trata-se da solução clássica do problema implementada dentro do contrato inteligente. Retorna o resultado esperado e o custo em "gas" associado com a sua execução

- $Consulta()$: Método puro (já discutido na Subseção 2.3), opcional ao problema que estamos abordando, para que o cliente possa consultar quais serão os valores de entrada utilizados pela blockchain para resolver o problema. A existência desse método só se justifica quando parte da entrada do algoritmo já está salva dentro da blockchain e será consultada em tempo de execução pelo contrato inteligente.

- $SolucionaECertifica(X)$: Refere-se a implementação realizada dentro da máquina do cliente e fora da blockchain (no inglês, costuma-se usar a expressão "off-chain"). Os valores retornados aqui são a saída Y e o certificado W.

- $Verifica(X, Y, W)$: É a implementação do algoritmo verificador dentro da blockchain. Busca somente retornar se a saída Y está correta para a entrada X, usando W como um auxiliar para essa verificação. Além disso, junto a essa verificação também será retornado o custo em "gas" associado a sua execução.

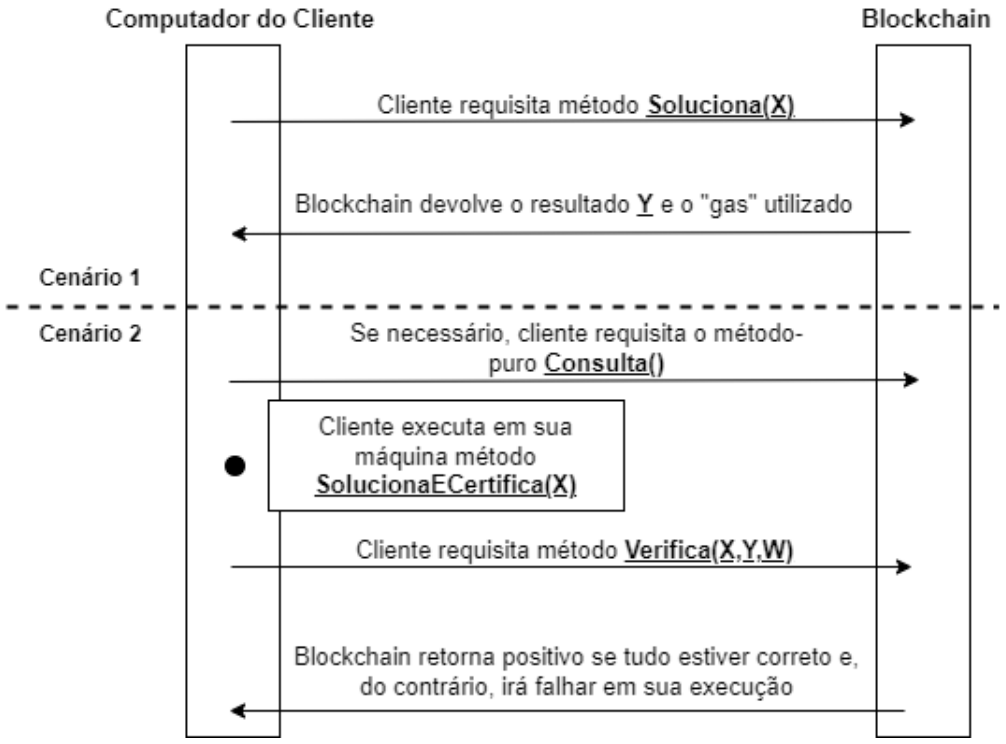

Figura 1: Cenário utilizado para análise do problema

4.2 AMBIENTE DE SIMULAÇÃO

Realizar os testes e simulações dentro da blockchain Ethereum seria algo custoso e pouco prático. Por conta disso, utilizaremos a ferramenta Truffle [16] que funciona como um ambiente de desenvolvimento e testes para o desenvolvimento de contratos inteligentes. Neste ambiente é criada uma versão da blockchain Ethereum onde não há nenhum dado histórico (ou seja, inicia-se do bloco zero) e cujo único minerador é exatamente o computador onde os testes serão executados. Como a medida de "gas" é determinística aos "OP CODES" do método no contrato, utilizar somente um minerador não impactará nas medidas aferidas durante as simulações.

No mais, essa ferramenta permite a realização de testes unitários. Esses testes são escritos em JavaScript e são executados de fora da blockchain, chamando os métodos da mesma e avaliando os resultados associados a essas chamadas. Assim, o método $SolucionaECertifica(X)$, descrito na Subseção 4.1, será a única implementação realizada em JavaScript e os outros três métodos serão implementados dentro da blockchain em Solidity.

4.3 COLETA DE RESULTADOS

Para o nosso trabalho, tão importante quanto os resultados (se não mais) são os custos envolvidos com a execução de cada solução. Mais especificamente, queremos comparar o custo de "gas" dos métodos $Soluciona(X)$ e $Verifica(X, Y, W)$, descritos na Subseção 4.1. Para isso, precisamos resolver como iremos aferir somente o "gas" de ambos os métodos, sem incluirmos a taxa base de 21.000 (conforme mencionada no trabalho de Michael Kong [6]) e qualquer outro custo envolvido.

A técnica que utilizaremos para aferir as medidas dos métodos isoladamente são baseadas em uma publicação de Jules Goddard [3]. Nela, o autor utiliza um método nativo de Solidity conhecido como *gasleft()*, que retorna quanto "gas" ainda há disponível antes de atingir o limite de uma transação (conforme discutido na Subseção 2.4). No mais, observamos também que, ao enviarmos o certificado como parâmetro de entrada, de acordo com o tamanho dele, o mesmo aumenta o custo inicial da transação (sem contar os 21.000 iniciais). Por isso, a quantidade de "gas" utilizada pelo método será aferida em duas partes no retorno: o custo de inicialização, que representa a quantidade de "gas" envolvida com a transmissão dos parâmetros de entradas e inicialização da execução do método, retirados apenas a taxa base inicial de 21.000 "gas"; e o custo de execução, que representa qual o gasto relacionado com a execução do algoritmo em si. Abaixo é possível verificar um trecho de código em Solidity utilizado para aferir o método $Soluciona(X)$ para o problema do MDC (que será discutido na Seção 5.1).

```
function Soluciona(int256 a, int256 b) public
            returns (int256, uint256, uint256)
{
    // block.gaslimit é o limite de gas de uma transação
    uint256 consumoInicial = block.gaslimit - gasleft() - 21000;
    int256 resultado;
    uint256 gasAntesDaFuncao = gasleft();
    resultado = algoritmoEuclidiano(a, b);
```

```
    uint256 gasUsado = gasAntesDaFuncao - gasleft();
    return (resultado, gasUsado, consumoInicial);
}
```

Mais detalhes sobre implementações e captura dos resultados podem ser visualizados e replicados a partir do meu GitHub [10], onde disponibilizei todos os códigos e instruções para validação dos testes.

5 PROBLEMAS ANALISADOS

Nessa seção, analisaremos alguns problemas que se prestam muito adequadamente ao esquema "resolver e certificar, depois verificar", detalhado na seção anterior. Nem todos os problemas representam situações comuns a serem resolvidas dentro de uma blockchain, mas ilustram o quão promissora pode ser a ideia de resolver externamente e apenas verificar internamente (à blockchain).

5.1 MÁXIMO DIVISOR COMUM

Encontrar o inteiro g, maior divisor comum de dois inteiros a e b, positivos e com pelo menos um deles diferente de zero, é uma tarefa que não possui solução em tempo constante. Por conta disso, mesmo que trate-se de uma operação simples, ainda é possível encontrar cenários onde a execução da mesma possa ser muito custosa computacionalmente para uma transação dentro da blockchain. Podemos pegar como exemplo o algoritmo euclidiano, que é a solução mais comum para o problema. Nele, o pior caso se dá quando a e b são números consecutivos da sequência de Fibonacci, levando o algoritmo a rodar em tempo $\Theta(\log b)$.

Para estas situações, veremos que g acompanhado de dois outros inteiros, que assumem o papel de certificado no problema, permitem que o contrato inteligente verifique a corretude de g para o problema em tempo constante. Com isso, não só reduzimos os custos nos piores cenários como também aumentamos a previsibilidade e uniformidade do valor cobrado em cima dessa operação.

5.1.1 Algoritmo Euclidiano

Sendo um dos algoritmos mais antigos, o algoritmo euclidiano baseia-se no conceito de que o MDC não muda quando o menor número for subtraído do maior. Em formato recursivo, seu pseudocódigo será conforme o Algoritmo 2.

Algoritmo 2 Euclidiano

1: **função** MDC(a, b): a e b são inteiros tais que $a \geq b \geq 0$ e $a > 0$
2: **se** $b = 0$ **então**
3: **devolve** a
4: **senão**
5: **devolve** MDC($b, a \bmod b$)
6: **fim se**
7: **fim função**

5.1.2 Algoritmo Certificador e Verificador

Para este problema, o algoritmo capaz de emitir um certificado é uma extensão do próprio algoritmo euclidiano, também conhecida como algoritmo euclidiano estendido (veja em Algoritmo 3). Nesta versão, além de calcularmos o valor de g, também retornamos outros dois inteiros x e y, diferentes de zero. Os mesmos funcionam como certificado pois $g = MDC(a, b)$ é o único divisor de a e b que satisfaz a condição $g = xa + yb$.

Para provarmos [11] a unicidade dessa condição, seja d um inteiro divisor qualquer de a e b:

$$g = xd\frac{a}{d} + yd\frac{b}{d} = d(x\frac{a}{d} + y\frac{b}{d})$$

A partir dessa expressão sabemos que qualquer d divide g, entre eles o $MDC(a, b)$. Além disso, g divide a e b, e por consequência também divide o $MDC(a, b)$. Se g divide o $MDC(a, b)$ e o $MDC(a, b)$ divide o g, logo $g = MDC(a, b)$.

Com esta prova em mãos, agora somos capazes de verificar a corretude do resultado obtido pelo algoritmo euclidiano estendido utilizando um algoritmo verificador [11] (veja em Algoritmo 4). Para isso, basta realizarmos 2 verificações, dentre elas: se o g informado é um divisor de a e b; se g é tal que $g = xa + yb$ para os certificados x e y informados.

Algoritmo 3 Euclidiano Estendido

1: **função** MDCe(a, b): a e b são inteiros tais que $a \geq b \geq 0$ e $a > 0$; Retorna os inteiros (g, x, y)
2: **se** $b = 0$ **então**
3: **devolve** $(a, 1, 0)$
4: **senão**
5: $(g, x, y) \leftarrow \text{MDCe}(b, a \bmod b)$
6: **devolve** $\text{MDCe}(g, x, y - \lfloor \frac{a}{b} \rfloor)$
7: **fim se**
8: **fim função**

Algoritmo 4 Verificador do Euclidiano Estendido

1: **função** VERIFICADORMDCE(a, b, g, x, y): Retorna verdadeiro se g for o MDC de a e b usando x e y como certificados. Do contrário, retorna falso
2: **se** $a \geq b \geq 0$ e $a > 0$ e $a \bmod g = 0$ e $b \bmod g = 0$ e $g = xa + yb$ **então**
3: **devolve** Verdadeiro
4: **senão**
5: **devolve** Falso
6: **fim se**
7: **fim função**

5.1.3 Aplicação em Contratos Inteligentes

Com os algoritmos certificador e verificador em mãos, agora podemos trabalhar em detalhes sobre como ambos serão aplicados sobre o contexto de interação com um contrato inteligente, conforme apresentado na Subseção 4.1.

Para isso, o contrato inteligente precisará implementar o algoritmo verificador (veja em Algoritmo 4) junto com alguma forma de consulta que permita ao cliente antecipar qual será o par a e b que precisará ser calculado durante a interação na blockchain.

Dessa forma, o cliente inicia o seu processo consultando a blockchain para, de alguma forma, descobrir qual será o par a e b que precisa ter seu máximo divisor comum calculado. Com isso, ele executará em sua própria máquina o algoritmo euclidiano estendido (veja em Algoritmo 3) e, ao interagir com o contrato inteligente na blockchain, informará os valores g, x e y para que o contrato verifique a corretude do resultado.

Uma observação importante que precisamos ter com o formato que esse algoritmo exige para a solução é sobre a volatilidade do par (a, b). É importante lembrar que num cenário onde o cliente envia os valores (g, x, y) errados, a transação falhará e o mesmo, ainda assim, terá que pagar as taxas de rede pela execução dessa interação com a blockchain. Por isso, se os valores (a, b) mudam constantemente dentro da blockchain, há um risco de que o cliente, num primeiro momento, calcule a solução e certificado (g, x, y) para um par (a, b) mas, na hora de interagir com a blockchain, a mesma tenha sofrido uma atualização e o par correto agora seja (a', b'), levando assim a transação do cliente a falhar.

5.1.4 Resultados obtidos

Conforme discutido na Subseção 4.3, iremos capturar, para cada resultado, o consumo inicial e o consumo de execução dos métodos dentro da blockchain. Exatamente para que seja possível entender se os valores g, x e y, que assumem o papel de certificado, impactam de maneira notória o consumo inicial de forma a prejudicar o

"gas" total para o método $Verifica(X, Y, W)$.

Para isso, precisamos primeiro entender como classificamos se um problema de MDC entre dois números A e B é mais complexo que outro para os números A' e B'. Tomando o algoritmo euclidiano como solução, quanto maior o número de passos, ou seja, quanto maior é o número de módulos diferentes de zero nas iterações, maior é a duração do algoritmo. Para buscar a simplicidade nessa análise, tomaremos como medida a ordem de grandeza de B pois, quanto maior é o B, maior é a chance de que hajam mais iterações no algoritmo até encontrar a solução.

Partindo desse pressuposto, cada uma das 3.000 amostras criadas foi gerada uniformemente conforme a ordem de grandeza de B. Ou seja, para gerar a amostra era necessário primeiro um número uniformemente aleatório entre 1 e 30 fosse gerado. Este número então seria usado para definir a ordem de grandeza da amostra para que aí então a mesma fosse gerada.

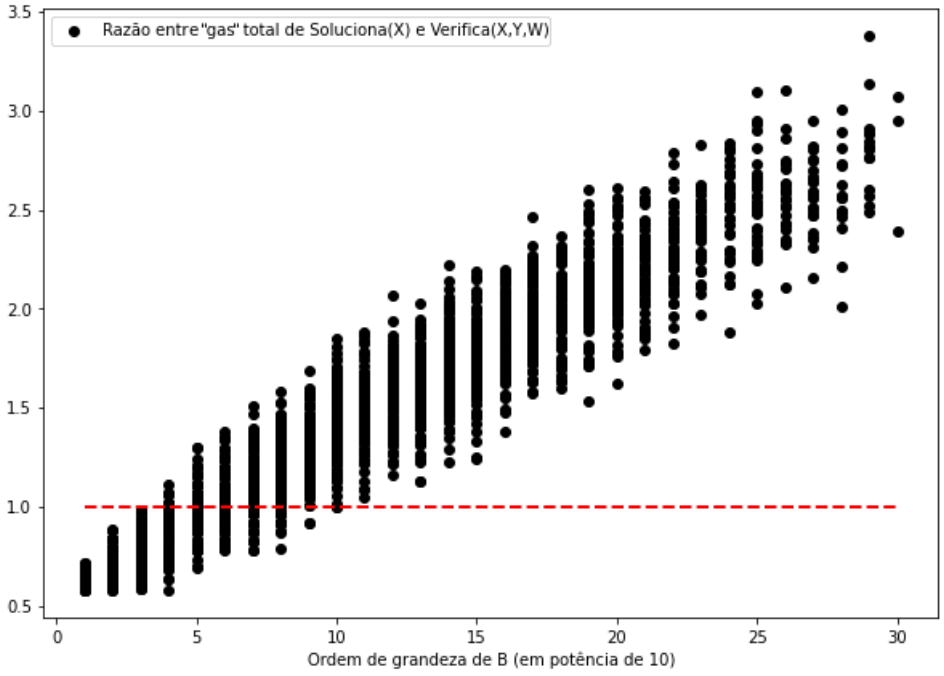

Figura 2: Gráfico em pontos da razão entre o "gas" total de $Soluciona(X)$ e $Verifica(X, Y, W)$ para cada uma das amostras, agrupadas pela ordem de grandeza de B. Quanto maior a razão, maior é o consumo de "gas" exigido por $Soluciona(X)$

Como é possível observar na Figura 2, a razão de "gas" total consumido entre os

dois métodos tende a crescer conforme ordem de grandeza de B aumenta. O que nos leva a entender que, quanto maior é a complexidade do problema, mais vantajosa é a utilização da técnica que estamos propondo. Além disso, podemos também observar que para todo B maior ou igual a 10^{11} não tivemos nenhuma amostra cuja a razão de "gas" indicasse desvantagem no uso de $Verifica(X, Y, W)$ no lugar de $Soluciona(X)$.

Analisando detalhadamente o comportamento de cada um dos dois métodos por meio da Figura 3, conseguimos entender como realmente o número de parâmetros de um método impacta no consumo inicial de "gas". Enquanto $Soluciona(X)$ recebe somente os inteiros a e b e possui um consumo inicial variando entre 627 e 915, $Verifica(X, Y, W)$ recebe também o resultado g e os certificados x e y que resultam em um consumo inicial maior que varia de 1138 e 1942. Contudo, ao compararmos o custo de execução o $Verifica(X, Y, W)$ apresenta um consumo constante de 217 para todas as amostras, enquanto $Soluciona(X)$ varia o consumo entre 301 e 6321 conforme a complexidade do problema aumenta.

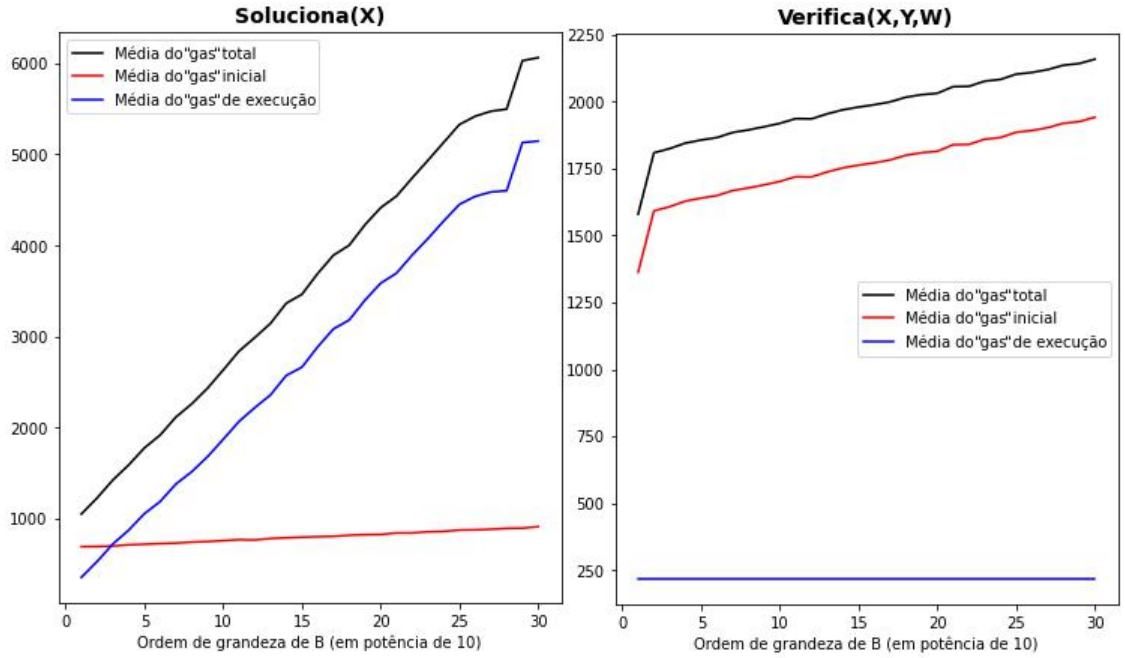

Figura 3: Gráficos das médias de "gas" consumidos pelos métodos $Soluciona(X)$ e $Verifica(X, Y, W)$ agrupadas pela ordem de grandeza de B

Agora que entendemos o comportamento de consumo de "gas" para cada um dos

métodos, onde $Verifica(X, Y, W)$ possui um custo inicial maior e $Soluciona(X)$ uma taxa de crescimento maior, podemos analisar em que cenários é vantajosa a utilização da solução proposta. A partir da Figura 4 vemos um comparativo entre a média de "gas" total utilizada por cada um dos métodos e observa-se que valores de B maiores ou iguais a 10^6 já favorecem a implementação da técnica estudada neste trabalho.

Mesmo que pareça um valor alto em situações cotidianas, dentro da própria blockchain, por exemplo, o saldo em ETH (moeda utilizada pela rede) é representado em unidades de "Wei", que são equivalentes a 10^{-18} ETH. Ou seja, a representação do saldo de 1 ETH dentro da blockchain já é da ordem de grandeza de 10^{18}, o que indica um cenário favorável para utilizar a técnica de MDC no saldo de duas contas, por exemplo.

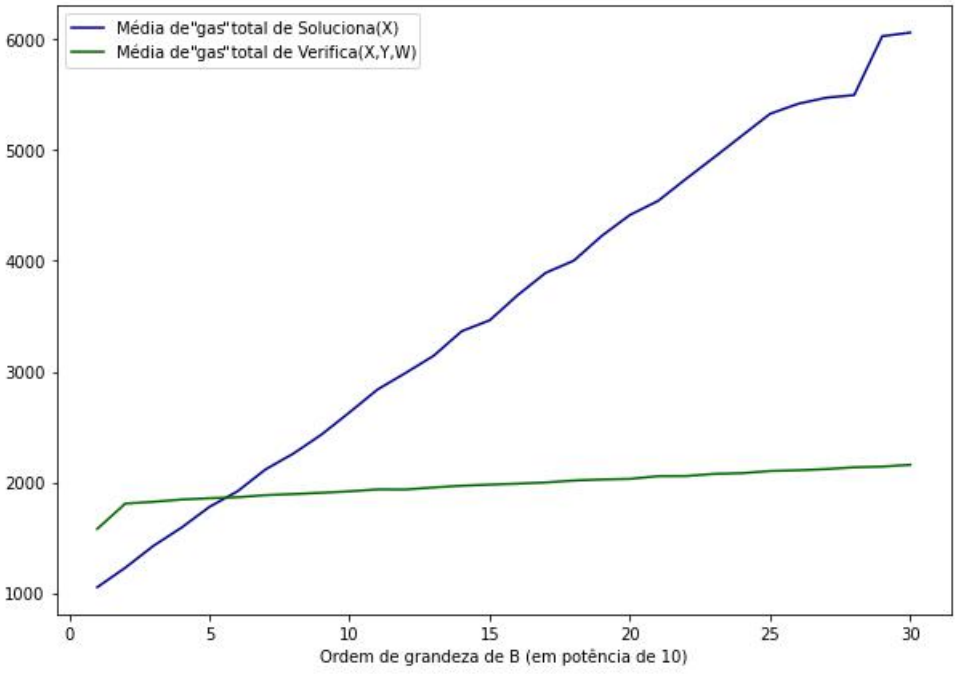

Figura 4: Gráfico comparativo da média total de "gas" consumida pelos métodos $Soluciona(X)$ e $Verifica(X, Y, W)$ agrupadas pela ordem de grandeza de B

5.2 K-SELECTION

Em um conjunto de N valores não ordenados, escolher os k menores é uma tarefa que cresce conforme o tamanho de N. Diferente dos outros problemas, conforme trabalhado na dissertação de mestrado da Anne Rose [9], a própria resposta produzida pelo algoritmo já assumirá o papel de certificado. Ou seja, trata-se de um problema onde achar a resposta é um desafio maior do que verificar se a mesma está correta, mesmo sem auxílio de certificados. Resumidamente, trata-se de um problema onde a entrada X é um inteiro k e uma lista L com N elementos, a saída Y é uma lista com os k menores elementos de X e W é um conjunto vazio.

No entanto ao inserirmos o problema dentro do contexto da blockchain, onde queremos otimizar o consumo de "gas", enfrentamos um outro problema com o valor de K. Quanto maior é o número de itens que precisamos selecionar, maior será o tamanho da resposta Y que passaremos como parâmetro em $Verifica(X, Y, W)$. Como já discutido na Subseção 2.3, sabemos que valores grandes de entrada já impactam num valor de consumo de "gas" inicial ainda maior, que pode desfavorecer a técnica proposta.

5.2.1 Quick Select

Com o conceito muito semelhante ao QuickSort [4], o algoritmo QuickSelect soluciona o problema em tempo linear $O(N)$ para qualquer valor de k. Para isso, o algoritmo seleciona o pivô de uma lista que contenha os N elementos e o utiliza para mover aqueles maiores que o pivô à direita e os menores à esquerda. Ao realizar esse procedimento, o algoritmo agora sabe a i-ésima posição do pivô na lista se a mesma estivesse ordenada. Com isso em mãos, basta o algoritmo repetir esse procedimento escolhendo entre o intervalo à esquerda ou à direita do pivô conforme for a entrada k e repetir os passos iniciais até que se ache um pivô cuja posição após a movimentação de itens seja exatamente k.

Um detalhe importante que se deve observar também é a impossibilidade de trabalharmos com números pseudo-aleatórios dentro da blockchain. Conforme já

discutido na Subseção 2.4, números pseudo-aleatórios não são suportados dentro da blockchain e sempre devem ser substituídos por alguma opção determinística. No caso da minha implementação, optei por utilizar sempre o valor mais à esquerda do intervalo de busca como pivô (o que é tão bom quanto qualquer outra escolha determinística), visando reduzir o número de instruções necessárias para determinar o mesmo. Para mais detalhes, a implementação em Solidity está disponível no meu repositório do Github [10] e também apresentada em pseudo-código no Algoritmo 6.

Algoritmo 5 Quick Select

1: **função** QUICKSELECT(L, k): L é a lista com N elementos e k é a quantidade de menores elementos que queremos de L; Retorna os k menores valores de L
2: $esquerda \leftarrow 0$
3: $direita \leftarrow$ tamanho de L - 1
4: inicializa variável $indicePivo$
5: **enquanto** $esquerda$! = direita **faça**
6: $indicePivo \leftarrow esquerda$
7: $(L, indicePivo) \leftarrow Pivoteia(L, esquerda, direita, indicePivo)$
8: **se** $k = indicePivo$ **então**
9: **devolve** k primeiros valores de L
10: **fim se**
11: **se** $k < indicePivo$ **então**
12: $direita \leftarrow indicePivo - 1$
13: **senão**
14: $esquerda \leftarrow indicePivo + 1$
15: **fim se**
16: **fim enquanto**
17: **fim função**

5.2.2 Algoritmo Verificador

Um dos diferenciais desse problema é a ausência do certificado, o que significa dizer que basta a resposta Y e a entrada X para realizarmos a verificação. Dessa

Algoritmo 6 Quick Select - Função de Pivoteamento

1: **função** PIVOTEIA(L, *equerda*, *direita*, *indicePivo*): Re-arranja a lista L movendo elementos menores que o pivô para a esquerda enquanto os maiores são movidos à direita; Retorna L re-arranjada e o novo índice do pivô após o arranjo
2: $valorPivo \leftarrow L[indicePivo]$
3: Trocar os valores em L dos índices *direita* e *indicePivo*
4: $i \leftarrow 0$
5: $novoIndicePivo \leftarrow esquerda$
6: **enquanto** $i \leq direita$ **faça**
7: **se** $L[i] < valorPivo$ **então**
8: Trocar os valores em L dos índices $novoIndicePivo$ e i
9: $novoIndicePivo \leftarrow novoIndicePivo + 1$
10: **fim se**
11: $i \leftarrow i + 1$
12: **fim enquanto**
13: Trocar os valores em L dos índices *direita* e $novoIndicePivo$
14: **devolve** $(L, novoIndicePivo)$
15: **fim função**

forma, o mesmo algoritmo implementado em $Soluciona(X)$ dentro da blockchain será também implementado em $SolucionaECertifica(X)$ fora da mesma para que o cliente o execute.

Trazendo aos detalhes, esse algoritmo resume-se em um conjunto de 4 verificações que, se atendidas corretamente, garantem que a saída Y está correta para a entrada X. São elas:

1. Se a quantidade de itens fornecidas em y é igual a k
2. Se todos os elementos da lista L cujo valor é menor ou igual ao k-ésimo menor valor estão em Y
3. Se a multiplicidade de cada elemento menor que o k-ésimo é igual em ambas as listas
4. Se a multiplicidade do k-ésimo elemento na lista Y é menor ou igual a multiplicidade do mesmo em L

O algoritmo verificador executa sua tarefa em tempo linear mas, diferentemente do trabalho da Anne Rose [9] onde a complexidade era $O(n+k) = O(n)$, na solução implementada dentro da blockchain chegamos na complexidade $O(n + k \log k) = O(n)$. Essa diferença se deu na segunda verificação, por conta de algumas limitações (discutidas na Subseção 2.4) para o suporte a uma estrutura de *HashSet* local. Esta seria utilizada para que pudéssemos realizar consultas da existência de valores no conjunto Y em $O(1)$. Por conta disso, tivemos que realizar essas consultas com uma busca binária, cuja complexidade era de $O(\log k)$.

No mais, outro fator que impactou durante a implementação é referente aos passos 3 e 4. Neles, precisamos contabilizar a multiplicidade da lista de entrada L e da lista de saída Y. Com a ausência de *HashMaps* locais em Solidity, tratamos o problema com L sendo implementado como variável global, o que nos permitiu criar um *HashMap* global para contabilizar a multiplicidade dos itens do mesmo. Já para a lista Y adaptamos o problema exigindo que a mesma fosse passada de forma ordenada pois, dessa forma, não temos dificuldade em contabilizar a multiplicidade

dos itens. O custo desta exigência é a necessidade de uma validação extra no algoritmo verificador para garantir que Y foi retornado de forma ordenada. De qualquer forma, essa validação extra não interfere na complexidade algorítmica do problema.

5.2.3 Aplicação em Contratos Inteligentes

Diferentemente do problema anterior, por conta da lista L (que faz parte da entrada X) ter um potencial de crescimento relevante para problemas reais, optamos por assumir que a mesma já está armazenada dentro da blockchain antes mesmo dos métodos $Soluciona(X)$ e $Verifica(X, Y, W)$ serem chamados. Isso reduz o custo inicial de ambos os problemas e permite trabalharmos com listas que possuam mais elementos. No entanto, isso exigiu que incluíssemos um passo extra na implantação do problema dentro da blockchain, que é a etapa de inserção de itens dentro da mesma.

Conforme discutido na Subseção 5.2.2, na etapa de verificação precisamos ter em mãos a multiplicidade dos itens presentes na lista L. Essa multiplicidade é registrada no momento que os itens de L são inseridos dentro da blockchain. O que acontece é que essa multiplicidade somente é necessária para solução de $Verifica(X, Y, W)$ e não para $Soluciona(X)$. Dessa forma, além de precisarmos capturar o custo de "gas" desses dois métodos, teremos também que fazer o mesmo para duas versões diferentes do método de inserção: a inserção para $Soluciona(X)$ e a inserção para o $Verifica(X, Y, W)$.

5.2.4 Resultados Obtidos

Seguindo a ordem de eventos da simulação para o problema, a primeira análise que precisa ser feita é relacionada aos custos de inserção. Para isso, foram feitas 10 diferentes simulações onde quantidades N distintas de itens foram inseridas por meio dos métodos de inserção para $Soluciona(X)$ e $Verifica(X, Y, W)$.

Como naturalmente já se esperava, na Figura 5 é possível ver que o custo de inserção para o método proposto nesse trabalho é mais custoso, vide a necessidade

de contabilizarmos a multiplicidade dos itens em uma estrutura de HashMap nativa da linguagem Solidity. Para grandes quantidades de itens, pode-se observar que o custo de inserção para o método de verificação é praticamente o dobro do outro método.

N	Custo Médio de Inserção para Soluciona	Custo Médio de Inserção para Verifica	Razão de Custo Soluciona/Verifica
1	43365.000000	63662.000000	0.681176
2	35165.000000	55554.000000	0.632988
4	31065.000000	51500.000000	0.603204
8	29007.000000	49465.000000	0.586415
16	27982.000000	48451.500000	0.577526
32	27469.562500	47944.750000	0.572942
64	27216.390625	47694.375000	0.570642
128	27086.898438	47566.187500	0.569457
256	27023.085938	47502.843750	0.568873
512	26993.054688	47472.671875	0.568602

Figura 5: Tabela comparativa do custo de inserção por item para os dois diferentes métodos de inserção. Além da razão entre esses dois custos.

Agora partindo para a análise da resolução do problema em si, uma das dificuldades na observação dos resultados é a questão de que o problema precisa ser analisado sob a perspectiva de duas medidas diferentes: o tamanho N da lista L e a quantidade de itens k a serem selecionados dela. Para resolver isso, fizemos simulações para 10 valores diferentes de N e, para cada uma dessas, fizemos variações no tamanho de k em potências de 2 até que atingisse a metade do tamanho de N. Simplificando com um exemplo, para a simulação de $N = 8$ foram simulados os seguintes valores de k: $1, 2, 4$.

Conforme se observa no gráfico da Figura 6, valores menores de k resultaram em cenários mais favoráveis para a utilização do método $Verifica(X, Y, W)$. O que já era de se esperar era que valores maiores de k resultassem em cenários piores para o método $Verifica(X, Y, W)$. No entanto, esses resultados indicaram que, para valores baixos de k, existem cenários onde o método proposto pelo trabalho supera a solução clássica $Soluciona(X)$ consumindo em alguns casos até metade do custo da mesma mostram um resultado favorável.

Trazendo aos detalhes dos métodos, no gráfico da Figura 7 o que podemos obser-

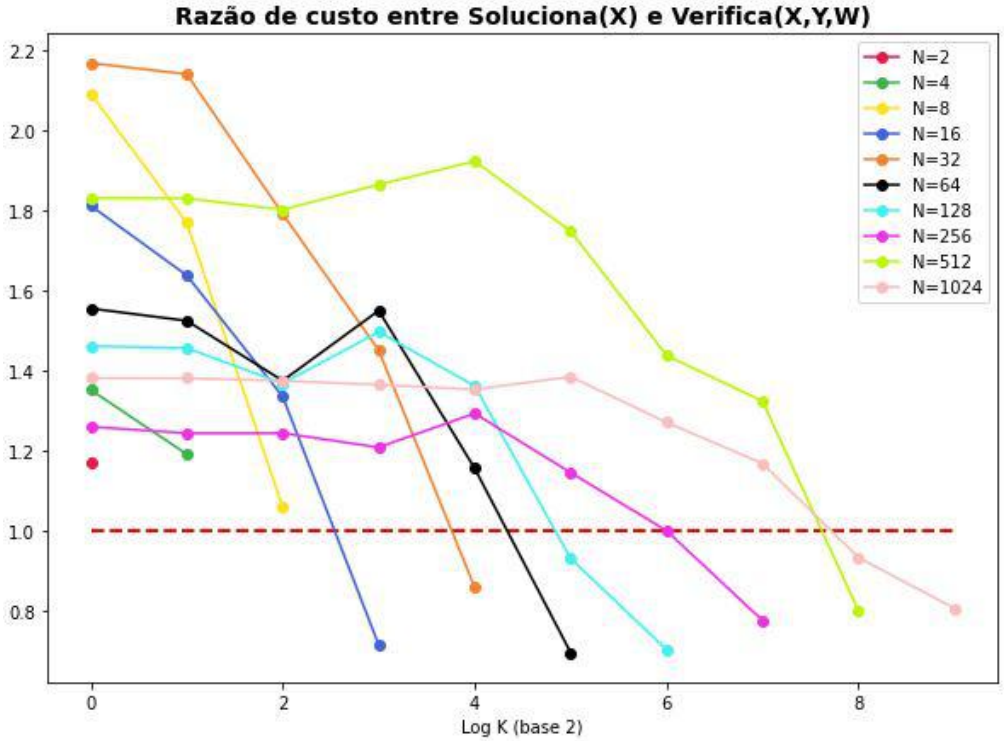

Figura 6: Gráfico com a razão entre o custo de $Soluciona(X)$ e $Verifica(X, Y, W)$. Nele temos diferentes cenários onde varia-se o tamanho da lista N para diferentes valores de k.

var é algo que foge daquilo que se esperava: os custos de execução de $Verifica(X, Y, W)$ destoam de maneira muito significativa dos custos iniciais do mesmo. Isso porque os custos iniciais estão diretamente ligados ao tamanho da entrada e, levando-se em consideração que Y tem o tamanho definido por k, mesmo assim o crescimento foi ínfimo se comparado à execução em si.

Com isso, é possível chegar a duas conclusões relevantes: o custo de inserção de dados para $Soluciona(X)$ é praticamente a metade do custo de inserção de $Verifica(X, Y, W)$; e o custo de solução do problema por meio de $Verifica(X, Y, W)$ só é favorável para valores baixos de k. Dessa forma, só faz-se interessante a utilização do método proposto neste trabalho se houver um volume grande de consultas para valores baixos de k e um volume baixo de inserções de itens na lista L.

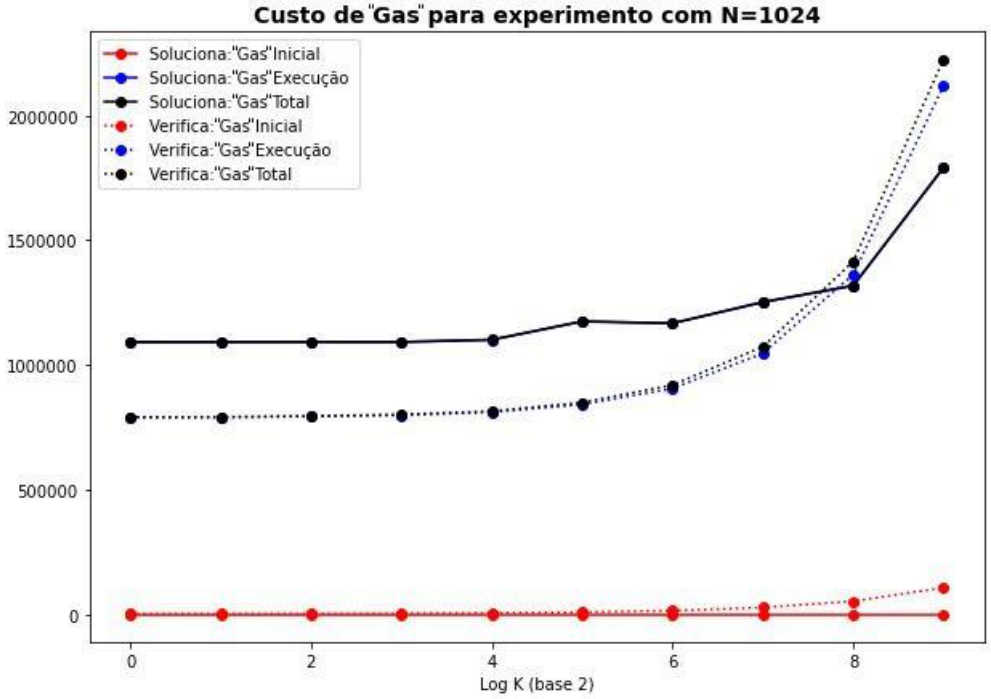

Figura 7: Gráfico de análise para um cenário isolado onde $N = 1024$. Em detalhes foram comparados os métodos $Soluciona(X)$ e $Verifica(X, Y, W)$ onde observa-se o custo inicial, de execução e total.

5.3 EXISTÊNCIA DE UM VALOR EM UM CONJUNTO DE DADOS

Muito comum em casos de uso dentro da blockchain, provar a existência de um valor dentro de um conjunto é uma tarefa que pode ser solucionada de diversas formas. Uma abordagem é a de simplesmente armazenar todos os valores dentro de uma estrutura de dados e, em seguida, realizar qualquer que seja a consulta. Como já se pode imaginar, essa solução é extremamente limitada por conta do custo de armazenamento, visto que para consultar a existência de um elemento a dentro de um conjunto de dados C teríamos que, no mínimo, ter um armazenamento da ordem de $\Omega(C)$".

Tendo em conta que o nosso objetivo é meramente assegurar que um certo elemento a pertence a conjunto C, veremos que é possível criar um mecanismo de verificação no qual apenas será preciso armazenar um hash de tamanho fixo dentro da blockchain. Para isso, uma estrutura conhecida como árvore de hash ou Árvore

de Merkle [12] será utilizada e discutida mais a frente na Subseção 5.3.2.

5.3.1 Consulta em HashMap

A primeira abordagem, que representará o método $Soluciona(X)$, baseia-se na simples solução onde o conjunto C de dados é inserido em uma estrutura de HashMap já suportada pela linguagem Solidity e, em sequência, a consulta ao item a é realizada diretamente a essa estrutura. Notem que, semelhante ao trabalho anterior, a configuração desse problema nos exigirá, além de analisar o custo das consultas, estudar os custos em "gas" dos meios pelos quais inseriremos as informações dentro da blockchain.

5.3.2 Árvore de Merkle

Desenvolvido inicialmente para ser utilizado como uma forma de verificação de integridade para uma base de dados, a árvore de merkle ou árvore de hash é uma estrutura em árvore que agrupa um conjunto de dados com operações de concatenação e hash. O interessante dessa árvore é que basta apenas armazenar o nó raiz da mesma que você já possui informação suficiente para, mais a frente, verificar se qualquer elemento a existe ou não dentro de C. Antes de entrarmos nos seus detalhes, é preciso enfatizar que, para haver integridade e segurança, deve-se escolher uma função hash $H(n)$ criptograficamente segura e no caso da minha implementação [10] foi escolhida a Keccak256.

Para iniciar a construção da árvore a partir do conjunto C de dados, o primeiro passo que precisamos fazer é construir uma árvore binária completa vazia cujo número de folhas seja maior ou igual ao conjunto de elementos C. Agora para, cada uma das folhas, um elemento C_i será armazenado após passar pela função hash, ou seja, cada uma das folhas armazenará $H(C_i)$. No mais, cada uma das folhas sobressalentes será atribuída com o hash de um valor vazio ou nulo, apenas para que sejam preenchidas.

Já a construção dos nós "pais" requer uma única informação que é exatamente os

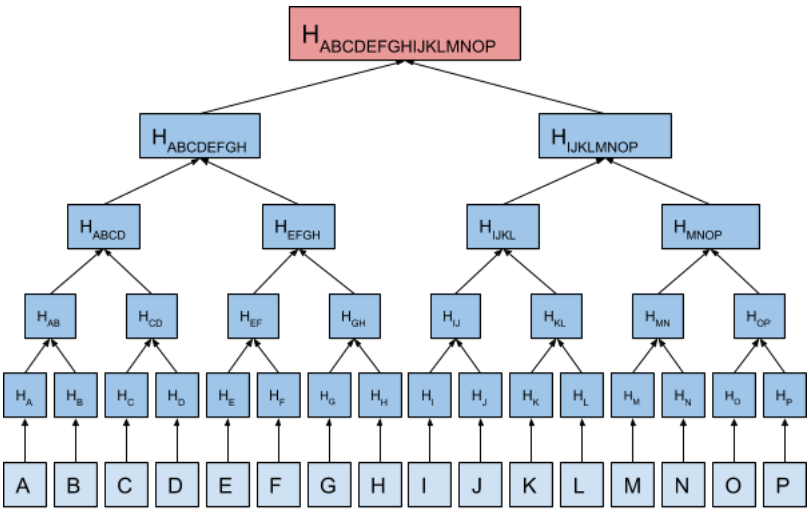

Figura 8: Repesentação de uma árvore de Merkle para um conjunto C com 16 itens

valores preenchidos pelos dois nós filhos, valores esses que são resultados de um hash. Assim, é feita a atribuição $NoPai = H(concat(NoFilho_{esq}, NoFilho_{dir}))$. Escolhe-se a concatenação como operação para combinar os dois filhos pois esta é, para dois operandos de tamanhos fixos, uma função injetora. Agora basta repetir o processo em todos os nós pais até que preencha-se completamente a árvore. A Figura 8 ajuda a ilustrar o procedimento.

Construída a árvore, fora da blockchain, temos em mãos tudo aquilo que precisamos para enviar à blockchain e também para construir um certificado W da existência de um elemento a no conjunto C. Para isso, armazenaremos dentro da blockchain somente o valor contido no nó raiz dessa árvore. No caso do algoritmo de hash Keccak256, estamos falando de nó com tamanho de 32 bytes. Ou seja, nessa proposta estamos substituindo o armazenamento de uma quantidade indefinida de elementos, que pode alcançar a ordem de terabytes, por simples 32 bytes de armazenamento.

Em contrapartida, para provarmos que um elemento a existe dentro do conjunto C precisaremos fazer mais do que apenas uma simples consulta, como é o caso da solução com HashMap. Para isso, precisamos produzir um certificado W fora da blockchain que permita a mesma reproduzir os passos de construção do valor do nó pai. Se ao fim de reconstrução o resultado obtido for igual ao já armazenado dentro

Algoritmo 7 Validação de um elemento em uma árvore de Merkle

1: **função** VALIDA(a, W): a é o elemento que queremos validar se existe em C. W W é uma lista de hashes; Retorna verdadeiro caso a exista em C
2: $i \leftarrow 0$
3: $hashMerkle \leftarrow consultaHashArmazenadoNaBlockchain()$
4: $hashReconstrucao \leftarrow H(a)$
5: **enquanto** $i < tamanho(W)$ **faça**
6: $hashReconstrucao \leftarrow concat(hashReconstrucao, W[i])$
7: $i \leftarrow i + 1$
8: **fim enquanto**
9: **se** $hashReconstrucao = hashMerkle$ **então devolve** Verdadeiro
10: **senãodevolve** Falso
11: **fim se**
12: **fim função**

da blockchain, então sabemos que trata-se de um valor pertencente ao conjunto.

Como vemos no Algoritmo 7, W é uma lista de hashes que serão iterados para serem concatenados e aplicados novamente à função hash reproduzindo os mesmos passos de construção. O que encurta o caminho dessa reconstrução é exatamente W que nos fornece somente o nós pais. Assim, por se tratar de um conjunto de iterações equivalente a altura da árvore, sabemos que W é da ordem de $\log C$.

Utilizando a Figura 9 como exemplo, se o elemento I fosse escolhido para validarmos a existência, o certificado deveria ser a seguinte lista nesta ordem: H_j, H_{KL}, $H_{MNOP}, H_{ABCDEFGH}$. Nesse exemplo, $H_{ABCDEFGH}$ encurta a reconstrução de 8 itens em um simples hash. No mais, a corretude desse algoritmo se dá pela natureza da operação de concatenar dois valores de tamanhos fixos e submetêlos a um hash. Tornando praticamente impossível a tarefa de se criar uma lista W que permita que um item b, não existente em C, seja verificado como verdadeiro.

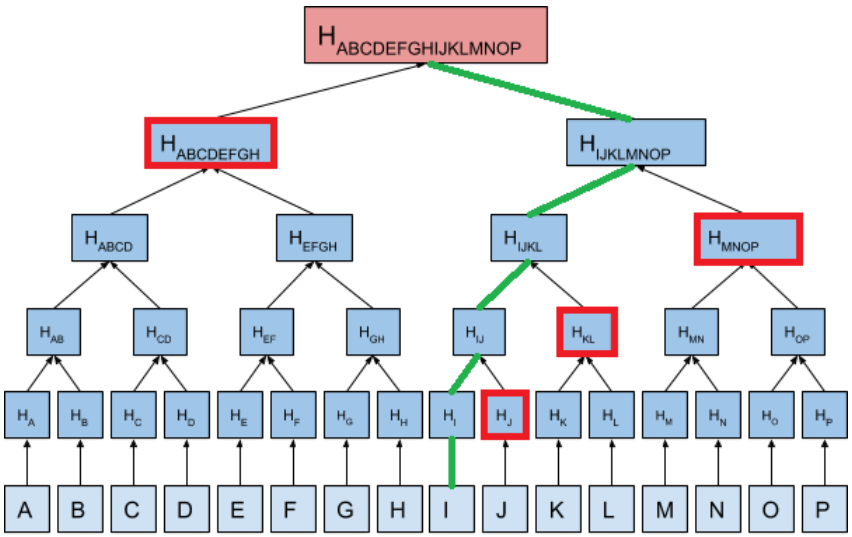

Figura 9: Exemplo de reconstrução do nó pai a partir do nó I. O caminho desta reconstrução é feito dentro da blockchain e representado pelas linhas em verde. Em vermelho são representados os hashes necessários para essa reconstrução e que, por isso, assumem papel de certificado.

5.3.3 Aplicação em Contratos Inteligentes

Trazer este problema para dentro da blockchain requer que criemos um cenário muito semelhante ao problema já discutido na Subseção 5.2. Isto porque, para trabalharmos com um conjunto C de dados que será armazenado dentro da blockchain, precisamos não só realizar uma análise no custo de consulta como também verificarmos o custo de atualização de valores no conjunto C. Dessa forma, além de trabalharmos com os métodos apresentados na estrutura geral dos problemas (Subseção 4.1), também analisaremos os métodos de inserção de valores para os cenários de verificação na blockchain e para os cenários de solução na blockchain.

Uma observação importante, que mais a frente vai ser trabalhada na análise de resultados, é o fato de que diferentes pessoas podem interagir com os contratos inteligentes e cabe ao criador do mesmo restringir o acesso dos métodos que julgar necessário. Para este problema em questão, podemos imaginar um contrato inteligente de uma moeda cujo método de transferência só pode ser executado se a origem e destino estão em uma "allowlist" de endereços autorizados (equivalente ao conjunto C).

Nesse cenário, os correntistas dessa moeda terão autorização de chamar o método de transferência e, dentro deste, ocorrerá a consulta para saber se os endereços de origem e destino estão na "allowlist". Observe que administrar os endereços da "allowlist" deve ser uma tarefa somente autorizada para o criador, enquanto transferir deverá ser autorizado para todos aqueles que se provarem presentes na lista.

5.3.4 Resultados Obtidos

Para este problema a coleta de dados foi realiza para simulações onde o tamanho N do conjunto C variava de 1 até 1024 crescendo em potências de 2. Para cada um dos diferentes tamanho de C, todos os elementos foram consultados e a partir deste o custo médio foi calculado.

Seguindo a ordem de eventos, o primeiro analisado será a inserção de dados. A partir da Figura 10 podemos constatar um resultado já esperado. Enquanto que para $Soluciona(X)$ é preciso inserir todos os elementos dentro do contrato inteligente, a inserção de $Verifica(X, Y, W)$ requer apenas que seja armazenado um hash de tamanho fixo para qualquer que seja o tamanho. Com isso, observamos uma razão que cresce linearmente conforme o tamanho do conjunto.

Já trabalhando os eventos de consulta, os resultados observados pela Figura 11 também demonstram aquilo que esperava-se. Enquanto $Soluciona(X)$ possui um custo de consulta aproximadamente fixo, por se tratar de uma consulta em um Hashmap, $Verifica(X, Y, W)$ possui um custo que cresce conforme o tamanho do conjunto. Esse crescimento ocorre porque o número de iterações do algoritmo de verificação e o tamanho de W crescem logaritmicamente se comparados ao tamanho N de C.

Por fim, conforme discutido na Subseção 5.3.3 é muito provável, na maioria dos cenários práticos para a utilização dessa solução, que hajam dois tipos de participantes que interajam com o contrato inteligente. Um deles seriam os usuários, que utilizariam o cenário do método de consulta enquanto que os outros seriam os

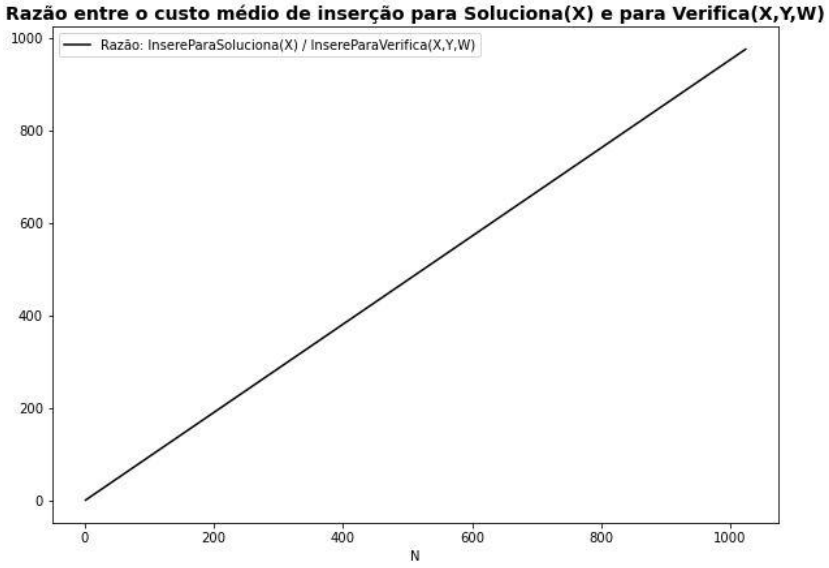

Figura 10: Análise da razão entre os custos de inserção para $Soluciona(X)$ e $Verifica(X, Y, W)$. A medida N representa a quantidade de elementos inseridos.

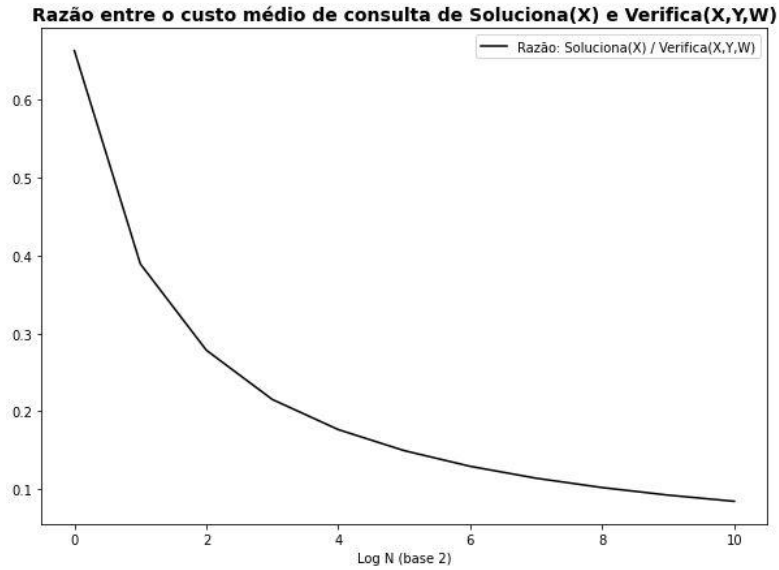

Figura 11: Razão entre o custo médio de uma consulta para um conjunto C de tamanho N nos dois métodos.

administradores, que utilizariam o método de inserção/alteração de dados. Com isso, na Figura 12 podemos observar que a solução da árvore de Merkle beneficia os administradores às custas dos usuários enquanto que a solução com hashmap faz exatamente o contrário.

Levando em consideração o crescimento linear do custo de inserção para a solução

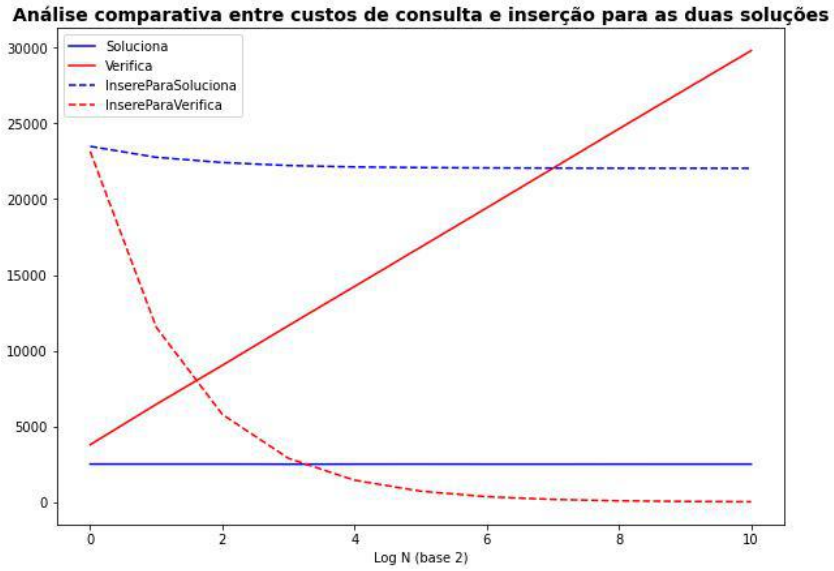

Figura 12: Comparação entre os custos de inserção e consulta para as duas abordagens possíveis conforme o aumento do tamanho N do conjunto C. Observe que os dois custos de inserção analisados nessa imagem representam o custo total de inserir N elementos dividido por essa mesma quantidade N.

usando em HashMap e o crescimento logarítmico do custo de consulta na solução usando árvore de Merkle, já temos um forte indício favorável a utilização de árvores. É o papel então do desenvolvedor interessado em utilizar a técnica analisar o volume de consultas, alterações e tamanho do conjunto C para que então possa decidir qual técnica é mais favorável. Na seção de trabalhos futuros, discutiremos sobre uma possível técnica onde o usuário custearia a inserção de dados para reduzir os custos de consulta.

5.4 SOMA DE SUBCONJUNTOS

Muito comuns na teoria de complexidade computacional, os problemas NP-completos pertencem a uma categoria que sempre permitem a aplicação da técnica proposta neste trabalho. Justamente como são mencionados em [11], essa categoria engloba problemas de decisão cuja obtenção da saída requer complexidade não polinomial enquanto que a sua verificação, ao contrário, pode ser feita em tempo polinomial.

Para isso, vamos trabalhar com um problema da categoria equivalente a muitos outros, que é o problema da soma de subconjuntos. Nele, temos um conjunto C de N elementos e queremos saber se existe algum subconjunto C' cuja soma dos elementos resulte em um valor determinado S. O resultado é uma simples decisão sim/não que informe se há ou não um subconjunto que atenda os requisitos.

5.4.1 Solução em Recursão, Memoização e Programação Dinâmica

Dentre as diversas opções de soluções para problema, temos diferentes caminhos não polinomiais que buscam otimizar a complexidade por meio de alguma técnica que favoreça uma variável ou outra do problema. Dentre elas, as variáveis que impactarão a complexidade de execução são exatamente o tamanho do conjunto C e o valor da variável soma S.

A primeira abordagem que pode-se pensar é a estratégia recursiva, onde utiliza-se a força bruta para simular todos os 2^N conjuntos distintos. Ela é, claro, uma solução que apresenta maior complexidade computacional mas que, ao menos, não requer nenhum armazenamento extra de memória. Já a técnica de memoização parte da mesma força bruta mas encurta alguns alguns caminhos ao armazenar resultados já calculados anteriormente. Por melhor que seja, essa técnica ao reduzir a complexidade do problema para $O(SN)$ também inclui uma complexidade extra de armazenamento equivalente da ordem de $O(SN)$.

Já a última implementação a ser abordado utiliza da programação dinâmica. Nele, busca-se utilizar da mesma técnica de memoização, com mesma complexidade computacional e em espaço mas sem utilização da recursão. Neste caso, cada cenário intermediário é calculado por meio de iterações no espaço em memória reservado para armazená-los.

5.4.2 Algoritmo de Certificação e Verificação

Conforme a natureza já discutida do problema, para verificarmos a sua solução conseguimos alcançar complexidade polinomial. Mais especificamente para este pro-

blema, o certificado que precisamos emitir é exatamente o conjunto C'. Este pode ser obtido por meio de qualquer um dos algoritmos discutidos na subseção anterior, desde que os mesmos armazenem os elementos que escolheram em seus subconjuntos.

A verificação precisa apenas garantir duas coisas: que C' de fato é subconjunto de C e que a soma dos elementos de C' resultem em S. No mais, um detalhe de implementação que vale mencionar é o de que utilizamos o C' ordenado para que, assim, a verificação de existência de todos os elementos possa ser feito em uma única iteração em C, sem a utilização de nenhuma estrutura de dados extra.

5.4.3 Aplicação em Contratos Inteligentes

Por se tratar de um problema de complexidade não polinomial, o tamanho de C não precisa ser abordado como fator que impacta o consumo de "gas" inicial relevantemente. Por conta disso, o mesmo será passado via parâmetro de entrada, sem utilizarmos a técnica de inserção de elementos que apresentamos nos problemas apresentados na Subseções 5.2 e 5.3.

No mais, por mais que cenários com resposta "não" para o problema tenham sido simuladas, somente nos atentaremos aos casos positivos. Isto, visto que o objetivo do trabalho é entender a utilização da técnica para otimização dentro da blockchain e, para este problema em específico, somente somos capazes de certificar a existência do conjunto C'. Caso quiséssemos trabalhar com a certificação e verificação da inexistência de um C', não poderíamos usar a solução aqui apresentada.

5.4.4 Resultados Obtidos

Para a análise de resultados, foram simulados 400 cenários distribuídos uniformemente para tamanhos N de 1 até 10 do conjunto C. Os conjuntos eram preenchidos randomicamente com valores de 1 a 10 e os valores de soma S utilizados eram randomicamente escolhidos entre 1 até 10N. Para este problema, analisaremos 3 algoritmos "clássicos" utilizados diretamente na blockchain e os compararemos com a técnica de verificação.

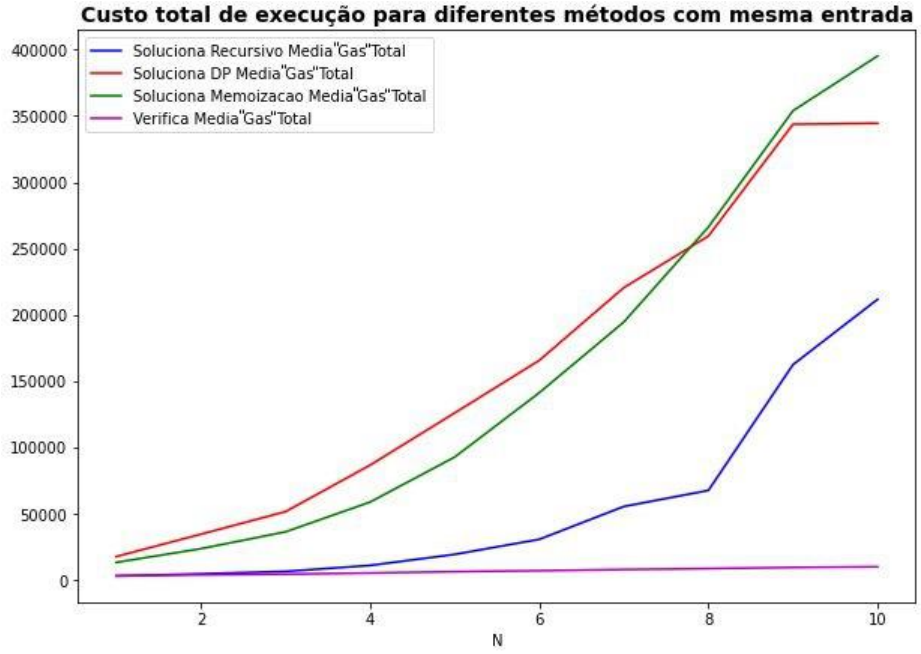

Figura 13: Gráfico comparativo do consumo médio de "gas" para cada configuração de tamanho N do conjunto C. As médias foram obtidas dos diferentes resultados observados para os valores S simulados em cada uma dessas configurações.

Conforme podemos observar na Figura 13, a técnica de certificação fora da blockchain com a verificação dentro da mesmo traz uma economia de "gas" extremamente relevante. Levando em consideração os altos valores de consumo "gas" para pequenas unidades de N, é possível afirmar que a única maneira de trabalhar com altos valores de N para este problema, e muito provavelmente todos os outros NP-completos, é a partir da técnica apresentada dentro desse trabalho. Principalmente levando em conta o limite máximo de "gas" já discutido na Subseção 2.4.

Já os resultados apresentados na Figura 14 reforçam a eficácia da técnica mas também apresentam observações interessantes para as 3 implementações do método $Soluciona(X)$ dentro da blockchain. A implementação recursiva por mais que apresenta uma média melhor, apresenta também maior inconsistência nos custos, isso porque a execução se interrompe assim que uma solução é descoberta na árvore de subconjuntos.

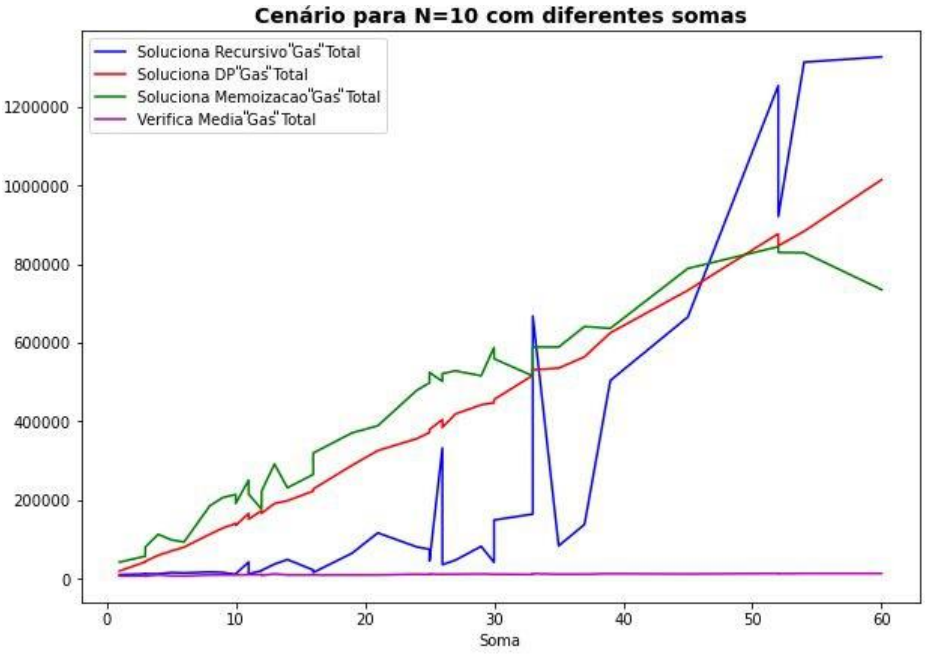

Figura 14: Gráfico para a análise isolada do caso N=10. Nela poderemos observar o impacto da variável de soma S no custo de "gas" da solução.

6 TRABALHOS FUTUROS E CONCLUSÃO

Ao longo do trabalho, muitos questionamentos e ideias que extrapolam o escopo do projeto acabaram surgindo ao isolar um problema e analisá-lo em detalhes. Dessa forma, listo aqui nesta seção algumas das ideias e propostas que surgiram e que podem ser usadas como insumo para futuros trabalhos, de minha autoria ou de outros autores.

Ao analisar a construção da árvore de Merkle, fiquei um pouco decepcionado com o crescimento logarítmico da complexidade de consulta conforme o aumento do tamanho do conjunto de dados. Em pesquisa de alguma solução que exigisse ainda mais poder computacional fora da blockchain para que, dentro da mesma, atingíssemos um menor consumo, cheguei a uma proposta que ainda não pode ser resolvida mas que, futuramente, pode ter alguma solução. Caso um dia se crie uma função hash $H()$ com propriedades associativas, ou seja, onde H(A,B,C,D,E) = H(H(A,B), C, H(D,E)), poderíamos construir métodos de validação de existência com ordem de $O(1)$. Isto se daria exatamente por não precisarmos construir o hash do nó raiz por meio de uma árvore mas sim operando todos os valores simultaneamente.

Além desta, a árvore de Merkle dentro da blockchain abre também margem para uma outra discussão sobre método de redução de custo para o consumo de "gas". Conforme discutido na Subseção 5.3.3, por se tratarem de duas entidades diferentes para os métodos de inserção e consulta, vemos que o custo de inserção no hashmap é caro mas em troca possui uma consulta barata. Já a árvore de Merkle possui a dinâmica de custo invertida. Dessa forma, uma solução que poderia ser trabalhada e analisada seria um cenário onde o próprio usuário custearia seu custo de inserção no hashmap para que, com isso, seus futuros custos de consulta barateassem.

No mais, outra linha de pesquisa que surge com esse trabalho são técnicas para realizar uma mensuração mais precisa em cima do consumo de "gas" de um método. Não há uma solução hoje bem estabelecida e há muito espaço para se desenvolver o que hoje é feito com aproximações. Assim como também é o caso da implementação, levando em consideração que os custos da rede estão diretamente ligados ao consumo

de instruções de máquina ("OP_CODES"), soluções com implementações ainda mais otimizadas podem ser desenvolvidas.

Por fim, após analisar todos os resultados em torno de 4 diferentes problemas, podemos concluir com este trabalho que a técnica demonstrou-se eficaz em muitos cenários. No entanto, como a maioria das soluções há sempre um conjunto de condições que favorecem a utilização da nossa técnica ou uma abordagem mais tradicional. Isso se dá, na maioria dos problemas que analisamos, pelo custo de transportar a resposta pronta para dentro da blockchain. Em todos os casos, é importante que um desenvolvedor interessado em aplicar a técnica entenda o contexto do problema para que seja possível tomar a decisão de qual é a melhor implementação.

REFERÊNCIAS

[1] How bitcoin works under hood. http://www.imponderablethings.com/2013/07/how-bitcoin-works-under-hood.html, 2013. Accessado em 09/01/2023.

[2] Buterin, V. Ethereum whitepaper. https://ethereum.org/en/whitepaper/, 2014. Accessado em 28/10/2022.

[3] Goddard, J. Gas cost of solidity functions. https://medium.com/coinmonks/gas-cost-of-solidity-library-functions-dbe0cedd4678, 2020. Accessado em 09/11/2022.

[4] Hoare, C. A. R. Quicksort. *The Computer Journal 5*, 1 (1962), 10–16.

[5] Khan, M., Sarwar, H., e Awais, M. Gas consumption analysis of ethereum blockchain transactions. *Concurrency and Computation: Practice and Experience 34* (11 2021).

[6] Kong, M. A scalable method to analyze gas costs, loops and related security vulnerabilities on the ethereum virtual machine.

[7] Malone, D., e O'Dwyer, K. Bitcoin mining and its energy footprint. pp. 280–285.

[8] Marchesi, L., Marchesi, M., Destefanis, G., Barabino, G., e Tigano, D. Design patterns for gas optimization in ethereum. pp. 9–15.

[9] MARINHO, A. R. A. F. Algoritmos certificadores e verificadores.

[10] Martins, L. Algorithms solidity. https://github.com/lucca30/algorithms-solidity, 2022.

[11] McConnel, R. M. Certifying algorithms. *Computer Science Review 5*, 2 (2011), 119–161.

[12] Merkle, R. C. A digital signature based on a conventional encryption function. *CRYPTO '87 293* (1988), 369–378.

[13] Michaelson, G. Programming paradigms, turing completeness and computational thinking. *The Art, Science, and Engineering of Programming 4* (02 2020).

[14] Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system.

[15] Solidity. Solidity. `https://docs.soliditylang.org/en/v0.8.17/`, 2014. Accessado em 28/10/2022.

[16] Truffle. Truffle. `https://github.com/trufflesuite/truffle`, 2015.

[17] Wood, G. Ethereum: A secure decentralised generalised transaction ledger.

[18] Zheng, Z., Xie, S., Dai, H.-N., Chen, X., e Wang, H. An overview of blockchain technology: Architecture, consensus, and future trends.