AVALIAÇÃO E MELHORIA DE PERFORMANCE DE ALGORITMOS EM JAVASCRIPT ATRAVÉS DE TÉCNICAS DE CODE TUNING

COSTA, Ramon Elias[1]; DAIBERT, Marcelo Santos[2] TREVIZANO, Waldir Andrade[2]; BAÍA. Joás Wesley[2]

[1] Graduando em Ciência da Computação - UNIFAGOC
[2] Docente do curso de Ciência da Computação – UNIFAGOC

gaspor991@gmail.com
daibert@unifagoc.edu.br
waldir@unifagoc.edu.br
joas.baia@unifagoc.edu.br

RESUMO

O objetivo principal é otimizar a execução de algoritmos em JavaScript, tornando-os mais eficientes e rápidos. Para isso, são utilizadas técnicas como análise de tempo de execução e aplicação de estratégias de otimização de código. Serão exploradas técnicas de otimização de laços, manipulação eficiente de arrays e strings, uso adequado de estruturas de dados, entre outras. Após o estudo destas técnicas a proposta deste projeto é avaliar o desempenho obtido após cada otimização.

Palavras-chave: Code Tuning. Teste de desempenho. Javascript.

INTRODUÇÃO

O Code Tuning, conforme mencionado por McConnell (2004), representa uma abordagem indispensável para aprimorar o desempenho de um programa, ao torná-lo mais eficiente, rápido e, simultaneamente, reduzir o consumo de recursos. Essa prática consiste na análise minuciosa do tempo de execução do programa e na identificação dos gargalos de performance, permitindo a detecção de trechos de código suscetíveis a otimizações e a aplicação de estratégias que visam aprimorar a eficiência geral da execução. Ao adotar o Code Tuning, é possível promover melhorias substanciais na eficiência e na velocidade de um programa, proporcionando um ambiente mais eficiente e econômico em termos de recursoscomputacionais.

Ao longo do tempo, observou-se um crescimento constante no desenvolvimento de aplicações Web, conforme discutido por Zaupa (2007). Esse cenário propiciou a popularização de diversas linguagens de programação, destacando-se o Javascript, que surgiu na década de 1990 com o objetivo primordial de conferir dinamicidade às páginas da Web. Atualmente, essa linguagem conquistou uma ampla disseminação e é empregada não apenas em aplicações Web, mas também em servidores Back-end, conforme apontado por Silva e Sobral (2017). A expansão do uso do Javascript reflete sua versatilidade e adaptabilidade, sendo uma escolha frequente e abrangente para a implementação de soluções tanto no front-end quanto no back-end de sistemas web.

PROBLEMA E SUA IMPORTÂNCIA

A falta de otimização na execução de algoritmos em JavaScript pode causar uma experiência frustrante para o usuário. Por exemplo, em uma aplicação web de bate-papo em tempo real, se os algoritmos responsáveis pela atualização e exibição das mensagens não forem otimizados, a lentidão resultante pode levar a atrasos significativos na entrega das mensagens, tornando a interatividade em tempo real praticamente impossível. Esperar vários segundos ou até minutos para que uma mensagem enviada seja exibida para os outros usuários seria uma experiência ruim que certamente afastaria os usuários e prejudicaria a utilidade da aplicação.

Além disso, em uma situação em que a aplicação web precisa manipular grandes volumes de dados, como em um sistema de análise de dados, se os algoritmos que lidam com esses dados não forem devidamente otimizados, o tempo de processamento pode se tornar excessivamente longo. Isso não só impactaria negativamente a eficiência da aplicação, mas também sobrecarregaria o hardware do usuário, tornando a experiência geral da aplicação lenta e frustrante.

Portanto, é crucial abordar esses problemas de performance. Ao otimizar os algoritmos em JavaScript, é possível obter ganhos significativos em velocidade, eficiência e redução de custos em projetos de software. Isso permitirá a criação de aplicações mais responsivas, capazes de lidar com grandes volumes de dados e oferecer uma experiência fluida ao usuário, evitando assim as frustrações causadas por experiências ruins.

OBJETIVOS

O objetivo geral deste projeto foi conduzir a avaliação e aprimoramento de desempenho de algoritmos em JavaScript, utilizando técnicas de code tuning com o intuito de otimizar sua execução e torná-los mais eficientes. Para atingir esse propósito, foram delineados objetivos específicos, entre os quais se destaca a revisão dos conceitos relacionados à avaliação de performance em JavaScript, com ênfase na análise do tempo de execução, o que permitirá uma compreensão mais aprofundada das características temporais dos algoritmos, possibilitando a identificação de áreas passíveis de otimização. Dessa forma, almeja-se não apenas a melhoria pontual de códigos, mas também o desenvolvimento de uma compreensão sólida das práticas de avaliação de desempenho, contribuindo para a construção de soluções mais eficazes e eficientes no contexto da linguagem JavaScript.

REFERENCIAL TEÓRICO

Code Tuning

O code tuning, também conhecido como otimização de código, é o processo de aprimorar o desempenho e a eficiência de um programa ou algoritmo, com o objetivo de reduzir o tempo de execução e o consumo de recursos (McConnell, 2004), proporcionando uma experiência melhor ao usuário. É uma prática comum no desenvolvimento de software, especialmente em aplicações que demandam alta performance, como sistemas de processamento de dados em tempo real, jogos e aplicações científicas.

Existem diversas técnicas utilizadas no code tuning, que abrangem desde ajustes nos algoritmos e estruturas de dados até otimizações de baixo nível no código-

fonte. A seguir, são apresentadas algumas das principais abordagens empregadas nesse processo, conforme escrito por McConnell (2019):

• Análise de algoritmos: Consiste em avaliar o algoritmo utilizado no programa e identificar possíveis melhorias. Às vezes, uma mudança no algoritmo pode resultar em ganhos significativos de desempenho.

• Uso eficiente de estruturas de dados: A escolha adequada das estruturas de dados pode ter um impacto significativo no desempenho do programa. Utilizar as estruturasadequadas para cada situação pode reduzir a complexidade computacional e melhorar a eficiência.

• Minimização de operações desnecessárias: Analisar o código em busca de operações redundantes ou desnecessárias pode levar a melhorias de desempenho. Evitar cálculos repetitivos ou operações que não afetam o resultado final é uma prática importante.

• Otimizações de loops: Loops são uma parte comum dos programas, e otimizá-los pode ser crucial para melhorar o desempenho. Técnicas como evitar chamadas de função desnecessárias dentro de loops e utilizar estratégias eficientes de iteração podem trazer benefícios significativos.

• Gerenciamento eficiente de memória: A alocação e a liberação adequadas de memóriasão fundamentais para evitar vazamentos e maximizar o desempenho. Utilizar alocação estática em vez de alocação dinâmica sempre que possível pode reduzir o tempo de execução.

• Utilização de ferramentas de profiling: Ferramentas de profiling permitem identificar as partes do código que consomem mais recursos e tempo de execução. Com base nessas informações, é possível direcionar os esforços de otimização para as áreas querealmente necessitam de ajustes.

JavaScript

JavaScript é uma linguagem de programação amplamente utilizada na internet, desempenhando um papel importante na criação de interatividade e dinamismo em sites e aplicativos modernos (Amazon, 2023). Inicialmente desenvolvido para ser executada em navegadores da web, o JavaScript agora é utilizado também em ambientes de servidor e no desenvolvimento de aplicativos móveis.

Essa linguagem de programação é considerada de alto nível, dinâmica e interpretada,o que significa que não precisa ser convertida em um formato específico antes de ser executada, permitindo que os desenvolvedores escrevam e testem seu código de forma rápida. A natureza dinâmica do JavaScript permite que as variáveis sejam declaradas sem a necessidade de especificar seu tipo, tornando-a uma linguagem flexível e fácil de aprender.

Uma característica interessante do JavaScript é o suporte à programação assíncrona. Por meio de recursos como Promises e async/await (Zakaz, 2010), é possível lidar de maneira eficiente e não bloqueante com operações demoradas, como chamadas de API e acesso a bancos de dados. Isso contribui para melhorar o desempenho e a responsividade das aplicações, proporcionando uma experiência de usuário mais fluida.

METODOLOGIA

Em primeiro lugar, foi fundamental realizar uma pesquisa detalhada sobre as diversastécnicas de code tuning disponíveis. Essa investigação permitiu selecionar as abordagens mais adequadas para otimizar o código em questão. Entre as técnicas comuns estão a otimização de algoritmos, o aprimoramento da estrutura de dados, a redução da complexidade e a otimização de recursos.

Além disso, foi essencial identificar algoritmos mal otimizados. Foi necessário criar exemplos específicos que simulam cenários de baixo desempenho. Esses casos de uso serviram como base para testar as técnicas de code tuning e avaliar o impacto das otimizações implementadas.

A medição do desempenho do código foi outra etapa crucial nesse processo. Utilizando ferramentas de análise de complexidade ciclomática e perfis de execução, foi possível identificar os gargalos de desempenho. Medir o tempo de execução de trechos de código críticos auxiliou na identificação das partes que consumiam mais recursos ou que eram executadas de forma menos eficiente.

Foi importante documentar todos os trechos de código mal otimizados. Essa documentação contém informações detalhadas sobre os problemas identificados, bem comoos resultados da medição de desempenho. Essa prática permitiu acompanhar as áreas que precisavam de melhorias e serviu como referência para avaliar o progresso durante o processo de otimização.

Com base na documentação dos trechos mal otimizados, foram aplicadas as técnicas de code tuning mais adequadas para cada cenário. Essa etapa envolveu a reestruturação de algoritmos, adoção de estruturas de dados mais eficientes, remoção de código redundante ea utilização de operações de baixo nível, entre outras técnicas específicas para melhorar o desempenho do código.

Após aplicar as técnicas de otimização, foram realizados testes para medir novamenteo desempenho do código. Comparando os resultados com as medições anteriores, foi possível verificar se houve melhorias significativas. Foi importante garantir que as otimizações não tenham introduzido erros ou efeitos colaterais indesejados.

Em alguns casos foi necessário iterar e refinar o processo, repetindo as etapas de medição, aplicação das técnicas de code tuning e testes para outros trechos de código mal otimizados ou áreas identificadas como pontos de melhoria. O processo de code tuning muitas vezes requer várias iterações até que se atinja o desempenho ideal ou um compromisso adequado entre desempenho e outros requisitos.

Nesse projeto, a escolha da linguagem JavaScript como base para o desenvolvimentodevei-se à sua familiaridade e à sua flexibilidade.

RESULTADOS OBTIDOS

Fibonacci

O algoritmo de Fibonacci, conhecido por sua simplicidade e eficiência, é amplamente estudado na literatura. Sua estrutura recursiva proporciona uma oportunidade única para otimização, uma vez que a implementação padrão pode resultar em complexidade exponencial. Com a aplicação de técnicas específicas, como memoização e o uso de uma abordagem iterativa, essas soluções podem se mostrar

mais eficientes, reduzindo o tempo de execução e a utilização de recursos. Pode ser vista a implementação desse algoritmo usando a abordagem iterativa presente no Código 1 e a recursiva no Código 2; e ainda utilizando memoização no Código 3, bem como o trecho de código utilizado para executar o teste de tempo de execução no Código 4.

Código 1 - Algoritmo de Fibonacci com abordagem iterativa

```
1.  function fibonacciIterative(n) {
2.      if (n <= 0) return 0;
3.      if (n === 1) return 1;4.
5.      let fibPrev = 0;
6.      let fibCurrent = 1;
7.      let fibNext;8.
9.      for (let i = 2; i <= n; i++) {
10.         fibNext = fibPrev + fibCurrent;
11.         fibPrev = fibCurrent;
12.         fibCurrent = fibNext;
13.     }
14.     return fibCurrent;15.}
```

Fonte: elaborado pelos autores.

Código 2 - Algoritmo de Fibonacci com abordagem recursiva

```
1.  function fibonacciRecursive(n) {
2.      if (n <= 0) return 0;
3.      if (n === 1) return 1;
4.      return fibonacciRecursive(n - 1) + fibonacciRecursive(n - 2);5. }
```

Fonte: elaborado pelos autores.

Código 3 - Algoritmo de Fibonacci com abordagem utilizando memoização

```
1.  const memo = {};
2.  function fMemo(n) {
3.      if (n in memo) return memo[n];
4.      if (n <= 2) return 1;
5.      memo[n] = fMemo(n - 1) + fMemo(n - 2);
6.      return memo[n];7. }
```

Fonte: elaborado pelos autores.

Código 4 - Função utilizada para realizar os teste de tempo de execução

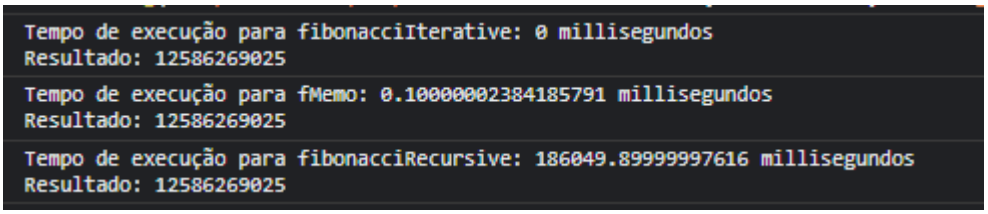
```
1.  function measureExecutionTime(func, ...args) {
2.      const start = performance.now();
3.      const result = func(...args);
4.      const end = performance.now();
5.      const executionTime = end - start;
6.      return {result, executionTime};7. }
8.
9.  const value = 50;
10. var { result, executionTime } = measureExecutionTime(fibonacciIterative, value);
11. console.log(`Tempo de execução para fibonacciIterative:
${executionTime} millisegundos \nResultado: ${result}`);
12. 12.
13. var {result, executionTime} = measureExecutionTime(fMemo, value);14.console.log(`Tempo de
execução para fMemo: ${executionTime}
millisegundos \nResultado: ${result}`);
15.
16. var { result, executionTime } = measureExecutionTime(fibonacciRecursive, value);
17. console.log(`Tempo de execução para fibonacciRecursive:
${executionTime} millisegundos \nResultado: ${result}`);
```

Fonte: elaborado pelos autores.

Após serem rodados alguns testes na execução desse código para encontrar o 50º valor gerado na sequência de Fibonacci, foram obtidos os resultados relatados a seguir.

Na primeira execução do algoritmo, foi percebido que a implementação iterativa do algoritmo alcançou o resultado mais rapidamente, tendo encontrado a resposta em 0 milissegumndos; já na implementação utilizando memorização, o tempo de execução foi de aproximadamente 0.100 milissegundos, enquanto na implementação com recursão,o algoritmo termina sua execução com 186,05 segundos (Figura 1).

Figura 1 - Primeiro teste do algoritmo de Fibonacci

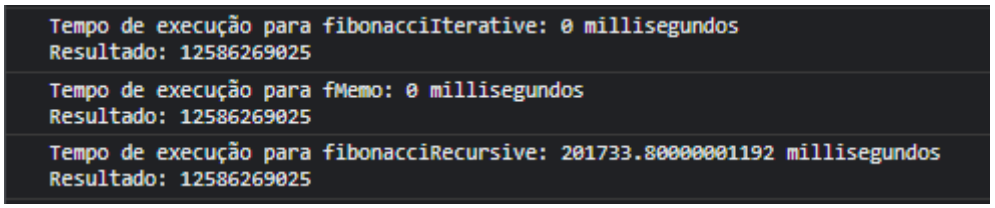


Fonte: elaborada pelos autores.

Após a segunda execução dos testes, notou-se uma diferença na execução do

algoritmo utilizando a técnica de memoização. Com essa técnica, demorou-se bem menos para encontrar o mesmo resultado, como pode ser visto na execução do teste da Figura 2.

Figura 2 - Segundo teste do algoritmo de Fibonacci



Fonte: elaborada pelos autores.

Esse comportamento ocorre, pois a técnica de memoização consiste em guardar valores já calculados em um vetor na memória, sem que seja necessário recalculá-lo; portanto, o uso dessa técnica é recomendado para casos em que ocorrem cálculos mais complexos e demorados.

Jamming/Fusion

Neste caso, teremos um cenário fictício, em que devemos construir uma aplicação que seja capaz de salvar, no formato JSON, elementos e seus valores. Seu nome é definido pela concatenação de "Elemento" junto com sua posição, e seu valor é definido pela sua posição multiplicado por 10, como mostrado no Código 5.

Código 5 - Código sem Jamming

```
1.      const TOTAL_ELEMENTS = 10000000;
2.      let elements = [];3.
4. const start = performance.now();5.
6.      for (i = 0; i < TOTAL_ELEMENTS; i++) {
7.          elements.push({name: `Elemento ${i}`});8. }
9.
10./* Código que não interaje com elements */11.
12.     for (i = 0; i < TOTAL_ELEMENTS; i++) {
13.         elements[i].value = i * 10;14.}
15.
16.const end = performance.now(); 17.const executionTime = end - start;
18.console.log(`Tempo de execução: ${executionTime} milisegundos`);19.console.log(elements);
```

Fonte: elaborado pelos autores.

Percebe-se que, para a alocação de todos os elementos, deve ser feito total de iterações para ser feita atribuição nos elementos. O erro da primeira implementação (Código 5) consiste em percorrer e fazer a atribuição de um mesmo elemento 2 vezes sem necessidade, pois o código que está entre esses laços de repetição (Código 5, Linha 6 e Código 5, Linha 12) não interage com os valores do objeto elements; a otimização

conhecida como “Jamming” ou “Fusão”, que pode ser vista no Código 6, consiste em unir os laços de repetição que manipulam o mesmo elemento.

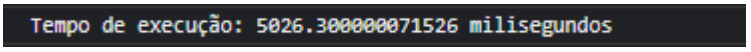
Código 6 - Otimização com Jamming

```
1.  const TOTAL_ELEMENTS = 10000000;
2.  let elementsOptimized = [];3.
4. const startOptimized = performance.now();5.
6.  for (i = 0; i < TOTAL_ELEMENTS; i++) {
7.      elementsOptimized.push({
8.          name: `Elemento ${i}`,
9.      value: (i * 10)10.    });
11.}
12.
13./* Código que não interaje com elements */14.
15.  const endOptimized = performance.now();
16.  const executionTimeOptimized = endOptimized - startOptimized; 17.console.log(`Tempo de execução otimizado: ${executionTimeOptimized}
milisegundos`); 18.console.log(elementsOptimized);
```

Fonte: elaborado pelos autores.

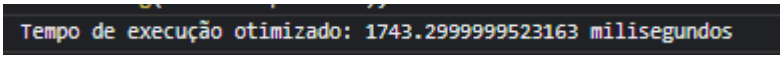
Após implementar a otimização, a execução do código teve uma melhoria de 65% em relação ao código original, como pode ser observado na Figura 3 (Teste do algoritmo original) e na Figura 4 (Teste do algoritmo otimizado).

Figura 3 - Execução sem otimização



Fonte: elaborada pelos autores.

Figura 4 - Execução com otimização



Fonte: elaborada pelos autores.

Uso de Promise.All()

Neste cenário foram desenvolvidos 2 códigos de promises, independentes uma da outra. A primeira promise demora 5 segundos para terminar sua execução, enquanto a segunda demora 1 segundo. Sua implementação pode ser vista no Código 7.

Código 7 - Código-Fonte de Promises

```
1.  const myPromise1 = () => new Promise((resolve, reject) => {
```

```
2.          setTimeout(() => {
3.              console.log("Operação 1 resolvida!");
4.          resolve("Operação 1 concluída");5.  }, 5000);
6. });
7.
8.      const myPromise2 = () => new Promise((resolve, reject) => {
9.          setTimeout(() => {
10.             console.log("Operação 2 resolvida!");
11.         resolve("Operação 2 concluída");12.}, 1000);
13.});
```

Fonte: elaborado pelos autores.

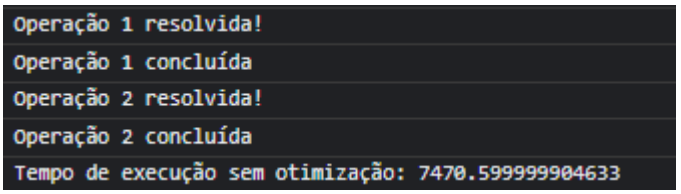
Abaixo, é encontrada sua implementação (Ilustrado no Código 8) em uma função,juntamente com o teste de tempo de execução dessa abordagem (Figura 5).

Código 8 - Primeira implementação de Promise

```
1.      async function doSomething() {
2.          const start = performance.now();
3.          const result1 = await myPromise1();
4.          console.log(result1);5.
6.          const result2 = await myPromise2();
7.          console.log(result2);8.
9.          const end = performance.now();
10.         const executionTime = end - start;
11.         console.log(`Tempo de execução sem otimização: ${executionTime}`);12.}
```

Fonte: elaborado pelos autores.

Figura 5 - Teste de execução de promises sem otimização



Fonte: elaborada pelos autores.

No Código 8, observa-se que a execução da myPromise1 (Código 8, Linha 3) acarretao bloqueio subsequente do restante do código, impossibilitando a execução da função myPromise2 até a conclusão da primeira. Uma falha evidente nesta abordagem reside no fato de que a segunda promise não depende da execução da primeira. Portanto, uma alternativa viável consiste na utilização do método Promise.all(), proporcionando a execução simultânea de ambas as Promises. A seguir, apresentam-se a implementação (Código 9) e o teste (Figura 6), destacando a aplicação do

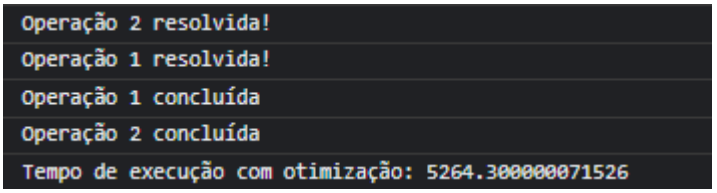
Promise.all().

Código 9 - Teste de execução de promise utilizando Promise.all()

```
1.  async function doSomethingWithPromiseAll() {
2.      const start = performance.now();
3.
4.      const [result1, result2] = await Promise.all([myPromise1(),myPromise2()]);
5.      console.log(result1);
6.      console.log(result2);
7.
8.      const end = performance.now();
9.      const executionTime = end - start;
10.     console.log(`Tempo de execução com otimização: ${executionTime}`);11.}
```

Fonte: elaborado pelos autores.

Figura 6 - Teste de execução de promises após otimização



Fonte: elaborado pelos autores.

ALTERAÇÃO DE ESTRUTURAS DE DADOS

No contexto abordado, viabiliza-se a utilização de diversas estruturas de dados por meio do JavaScript. No presente estudo, efetuou-se uma análise comparativa de operaçõesde busca em três dessas estruturas: Array (Código 10), Objeto (Código 11) e Map (Código 12).

Código 10 - Implementação de busca em Array.

```
1.  function buscaArray(arr, elemento) {
2.      const inicio = performance.now();
3.      const encontrado = arr.includes(elemento);
4.      const fim = performance.now();
5.      const tempoExecucao = fim - inicio;
6.      console.log(`Busca em Array. \nTempo de execução: ${tempoExecucao}ms`);
7. }
8.
9. const arrayExemplo = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
10.buscaArray(arrayExemplo, 10);
```

Fonte: elaborado pelos autores.

Código 11 - Implementação em Objeto

```
1.  function buscaObjeto(obj, chave) {
2.      const inicio = performance.now();
3.      const encontrado = obj[chave] !== undefined;
4.      const fim = performance.now();
5.      const tempoExecucao = fim - inicio;
6.      console.log(`Busca em Objeto. \nTempo de execução: ${tempoExecucao}ms`);
7. }
8.
9.  const objetoExemplo = {
10.     chave1: 'valor1',
11.     chave2: 'valor2',
12.     chave3: 'valor3',
13.     chave4: 'valor4',
14.     chave5: 'valor5',
15.     chave6: 'valor6',
16.     chave7: 'valor7',
17.     chave8: 'valor8',
18.     chave9: 'valor9',
19.     chave10: 'valor10',20.};
21.
22.buscaObjeto(objetoExemplo, 'chave10');
```

Fonte: elaborado pelos autores.

Código 12 - Implementação em Map

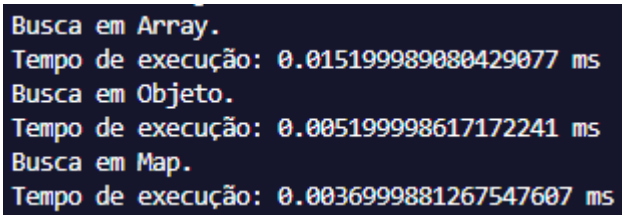
```
1.  function buscaMap(map, chave) {
2.      const inicio = performance.now();
3.      const encontrado = map.has(chave);
4.      const fim = performance.now();
5.      const tempoExecucao = fim - inicio;
6.      console.log(`Busca em Map. \nTempo de execução: ${tempoExecucao}ms`);
7. }
8.
```

```
9. const mapExemplo = new Map([10.      ['chave1', 1],
11.             ['chave2',2],
12.             ['chave3',3],
13.             ['chave4',4],
14.             ['chave5',5],
15.             ['chave6',6],
16.             ['chave7',7],
17.             ['chave8',8],
18.             ['chave9',9],
19.             ['chave10',10]
20.]);
21.
22.buscaMap(mapExemplo,'chave10');
```

Fonte: elaborado pelos autores.

Pode ser constatado que a melhor estrutura de dados disponibilizada pelo Javascriptpara fazer essa busca é o Map, como pode ser visto no teste na Figura 7:

Figura 7 - Execução do teste de busca



Fonte: elaborada pelos autores.

CONCLUSÃO

Numa abordagem abrangente, este artigo conduziu uma revisão dos conceitos relacionados à avaliação de desempenho em JavaScript, incluindo a análise do tempo de execução. A pesquisa envolveu a investigação e a seleção criteriosa de técnicas de otimização de código, específicas para a melhoria de performance de algoritmos em JavaScript. Posteriormente, essas técnicas foram aplicadas aos algoritmos em questão, com o objetivo claro de aprimorar sua eficiência e velocidade de execução.

A validação da eficácia das técnicas de otimização foi realizada por meio de testes e avaliações comparativas do desempenho dos algoritmos, antes e depois da implementação das otimizações. Os resultados obtidos foram identificados e documentados, destacando as melhorias de desempenho alcançadas e os insights adquiridos durante o processo de ajuste de código. Essa abordagem integrada proporciona uma compreensão das estratégias adotadas e de seus impactos na performance dos algoritmos em JavaScript, contribuindo assim para o avanço do conhecimento e prática na otimização de código.

Considerando o horizonte futuro, algumas sugestões para trabalhos subsequentes incluem a investigação de novas técnicas emergentes de otimização de código e a avaliação do impacto dessas técnicas em ambientes específicos. Além disso, explorar a interação entreotimizações de código e segurança, bem como a adaptação das técnicas existentes para ambientes de computação em nuvem, são áreas promissoras para futuras pesquisas. Esses estudos podem contribuir para o avanço contínuo no campo da otimização de código e proporcionar soluções mais eficazes para os desafios enfrentados pelos desenvolvedores.

REFERÊNCIAS

AMAZON SERVIÇOS DE VAREJO DO BRASIL LTDA. **O que é o JavaScript (JS)?**. [*S. l.*], 1 jan. 2023. Disponível em: https://aws.amazon.com/pt/what-is/javascript/. Acesso em: 20 set. 2023.

ANJOS, Luiz Felipe Rosa. Evolução do Javascript em aplicações MULTIPLATAFORMA: como projetos podem se beneficiar dos frameworks e bibliotecas disponíveis. **Evolução do Javascript em aplicações multiplataforma**, [*s. l.*], 2017. Disponível em: https://repositorio.ifsc.edu.br/bitstream/handle/ 123456789/1019/20190311_TCC_LuizFelip eDosAnjos.pdf. Acesso em: 31 maio 2023.

GRONER, Loiane. **Estruturas de dados e algoritmos com JavaScript**: escreva um código JavaScript complexo e eficaz usando a mais recente ECMAScript. 2. ed. [*S. l.*]: Novatec Editora, 2019.

HAVERBEKE, Marijn. **Eloquent JavaScript**. 3. ed. [*S. l.*]: No Starch Press, 2018.

MCCONNELL, Steve C. **Code Complete**. 2nd. ed. [*S. l.*: *s. n.*], 2004.

SILVA, Daniela Rocha; SOBRAL, Luis Felipe Bentin. **Um estudo em larga escala sobre a estrutura do código-fonte de pacotes JavaScript.** 2017. Trabalho de conclusão de curso (Bacharelado em Sistemas de Informação) - Universidade Federal do Estado do Rio de Janeiro - Centro de Ciências Exatas e Tecnologia Escola ee Informática Aplicada, [*S.l.*], 2017. Disponível em: https://bsi.uniriotec.br/wp-content/uploads/sites/31/2020/05/201707DanielaRochaLuisSobr al.pdf. Acesso em: 14 jun. 2023.

ZAKAS, Nicholas C. **JavaScript de alto desempenho**. 1. ed. [*S. l.*: *s. n.*], 2010.

ZAUPA, Fabio; GIMENES, Itana M. S.; COWAN, Don; ALENCAR, Paulo; LUCENA, Carlos. **Um processo de desenvolvimento de aplicações web baseado em serviços.**, [s. l.], 2007. Disponível em: https://www.ic.unicamp.br/sbcars2007/tecnicas/files/sbcars2007-zaupa-processo.pdf. Acesso em: 17 maio 2023.