Universidade Federal de Minas Gerais
Ciência da Computação

DCC205: Estruturas de Dados

Autor: Luisa Lopes Carvalhaes
Matrícula: 2023028064

Trabalho Prático III

"Estações de Recarga da BiUAIDi"

August 20, 2024

1 Introdução

Este trabalho visa desenvolver uma solução para um problema logístico enfrentado pela empresa fictícia *BiUaiDi*, que está se estabelecendo em Belo Horizonte. A empresa necessita de um sistema eficiente para gerenciar suas estações de recarga pela cidade. O objetivo é, dado um ponto (x, y), identificar e retornar os k pontos de recarga ativos mais próximos dessa coordenada, facilitando a navegação até essas estações.

Atualmente, o sistema utiliza uma abordagem simples e ineficiente, que calcula a distância de (x, y) a todos os pontos de recarga e, em seguida, ordena esses pontos para encontrar os mais próximos. Embora funcional, essa abordagem não é escalável e se torna ineficaz com o aumento do número de estações.

O objetivo deste trabalho é melhorar o sistema através das seguintes etapas:

- **Flexibilização do Aplicativo:** Adaptar o sistema para permitir a ativação e desativação dinâmica dos pontos de recarga, proporcionando uma gestão mais flexível.
- **Uso de Estrutura de Dados Eficiente:** Implementar a QuadTree, uma estrutura de dados que se adequa melhor a dados geográficos, para otimizar a busca dos pontos de recarga mais próximos.
- **Comparação das Abordagens:** Avaliar e comparar a abordagem atual com a nova solução em termos de desempenho, carga de trabalho e eficiência.

Essas melhorias visam tornar o sistema mais eficiente e adequado às necessidades reais de gestão das estações de recarga.

2 Método

Nesta seção, será discutida e apresentada a documentação da implementação das estruturas de dados, classes e funções utilizadas no programa. O foco estará nas principais estruturas de dados empregadas, detalhando como elas são usadas para resolver o problema proposto e otimizar a funcionalidade do sistema.

2.1 Classes Simples e Auxiliares

- `Addr.h`: Estrutura que representa os dados de um endereço de forma detalhada. Possui um construtor padrão e uma função que imprime seus próprios dados. Representa essencialmente uma estação de recarga de forma abstrata.
- `Box.h`: Classe que define um retângulo através de dois pontos: o canto superior esquerdo (`topLeft`) e o canto inferior direito (`bottomRight`). Esta classe é utilizada para criar as delimitações da Quadtree e possui um construtor próprio. Inclui a função `contains`, que verifica se um ponto está dentro do retângulo e `minDistanceToPoint`, que calcula a distância mínima entre um ponto e o retângulo.
- `Pair.h`: Classe template que representa um par de valores. Inclui um construtor e sobrecarga dos operadores para facilitar a manipulação de pares de dados.
- `Point.h`: Classe que representa um ponto no espaço 2D. Possui coordenadas `x` e `y`, um construtor próprio e a função `euclideanDistance`, que calcula a distância Euclidiana até outro ponto.
- `Qnode.h`: Classe que representa um nó em uma árvore Quadtree. Armazena uma estação de recarga (`Addr`) como dado principal, além de uma caixa de limites (`Box`) e índices para os filhos na Quadtree: `nw`, `ne`, `sw` e `se`.

2.2 Classes Principais

2.2.1 Hash

A classe `Hash` implementa uma tabela de hash com endereçamento aberto, utilizando sondagem linear. Esta tabela é responsável por armazenar e recuperar as estações (`Addr`). Cada `Addr` possui um ID único, que é utilizado nas operações de inserção e recuperação na tabela de hash. A função `hash()` calcula o índice

na tabela somando os valores ASCII de todos os caracteres da string ID fornecida e aplicando o módulo com o tamanho da tabela, garantindo que o índice fique dentro dos limites da tabela. No caso de colisões, a sondagem linear é usada para percorrer a tabela até encontrar uma posição livre. Um vetor auxiliar é mantido para marcar os índices ocupados. Os métodos `insert` e `get` são responsáveis pela inserção e recuperação dos dados `Addr` das estações, seguindo a lógica apresentada.

2.2.2 Priority Queue

A classe `PriorityQueue` é implementada como um template e representa um *Max Heap*. Ela armazena pares `<Pair<double, Addr>`, onde o `double` representa uma distância e `Addr` é a estação associada. A prioridade na fila é determinada pelo valor `double`, que representa a distância. Esta estrutura é crucial para a lógica de KNN (K-Nearest Neighbors) na QuadTree, pois auxilia na busca dos k pontos mais próximos. A estrutura é uma árvore binária com a propriedade de que o valor gravado em um nó é sempre maior ou igual ao valor gravado em seus descendentes. Esta propriedade é mantida pelas operações `heapify-up` e `heapify-down` durante inserções e remoções. O heap é implementado em um array simples, cujo tamanho máximo é dinamicamente ajustado, conforme necessário.

2.2.3 QuadTree

A `QuadTree` é a principal estrutura de dados da implementação. Trata-se de uma estrutura hierárquica projetada para dividir uma área bidimensional em quadrantes menores, divididos em quatro partes, a fim de proporcionar uma organização espacial eficiente dos pontos. Cada nó na árvore delimita uma região retangular no plano, representada pela classe `Qnode`. Cada quadrante pode ser subdividido recursivamente em quatro quadrantes menores, representados pelas direções cardinais `(NW, NE, SW, SE)`, que são os índices dos filhos de um quadrante na estrutura da árvore.

A versão tradicional da `QuadTree` é implementada dinamicamente com ponteiros. No entanto, para este trabalho, foi adotada uma abordagem vetorizada, onde os quadrantes são representados em um array. Assim, um quadrante é subdividido em quatro partes, e seus descendentes são alocados nos próximos índices disponíveis no array. O TAD (Tipo Abstrato de Dados) da `QuadTree` possui um tamanho fixo (devido à implementação estática vetorizada) e começa com o nó raiz (índice 0) e um índice para o próximo `Qnode`.

As principais operações da `QuadTree` são:

- **find**: Percorre a árvore em níveis até encontrar o quadrante que contém o ponto de interesse. Continua iterativamente até encontrar uma folha (marcada como -1) e retorna esse `Qnode`. Este método é sobrecarregado para aceitar tanto pontos (`Point`) quanto estações (`Addr`) na busca.

- **insert**: Primeiro, chama o método `find` para localizar o quadrante correto onde o endereço deve ser inserido. Após encontrar o quadrante, o método subdivide o quadrante criando filhos e aloca cada ponto no quadrante apropriado.

- **KNN**: Função de consulta principal que usa a lógica dos k vizinhos mais próximos para recuperar os pontos mais próximos a um ponto de referência. Utiliza a fila de prioridade (`PriorityQueue`) para manter um Max Heap de pares distância e estação. A operação é dividida em duas partes: `KNN` e `KNNRecursive`. O método `KNN` chama a versão recursiva `KNNRecursive`, que verifica se o índice é de um nó folha. Se não for, percorre a árvore e, se encontrar uma estação de recarga com uma distância menor do que a do elemento no topo da fila de prioridades (com a maior distância), a estação com a maior distância é removida da fila e a nova é inserida. Esta abordagem otimiza a busca, pois evita pesquisar em quadrantes potencialmente improdutivos pois estão "dentro" desses limites, logo não tem possíbilidades de terem distâncias menores neles.

- **Activate/Deactivate**: Operações que modificam o estado de uma estação de recarga. Estas alterações são feitas pelos métodos `activate()` ou `deactivate()`. O método `find` é usado para localizar o elemento em questão, e o hash é utilizado para acessar e modificar o status da estação de ativo para inativo e vice-versa.

3 Análise de Complexidade

Nesta seção, são apresentadas as análises de complexidade de tempo e espaço para os principais procedimentos implementados no programa.

3.1 Solução Inicial Ingênua

Na solução inicial ingênua, calcula-se a distância de um ponto de referência (x, y) para todos os N pontos de recarga. O cálculo das distâncias tem complexidade $\mathcal{O}(N)$. A etapa seguinte é a ordenação dos pontos com base nas distâncias, o que possui complexidade $\mathcal{O}(N \log N)$. Portanto, a complexidade total da solução é dominada pela ordenação, resultando em $\mathcal{O}(N \log N)$.

3.2 Hash

3.2.1 Insert:

A operação `insert` em uma tabela hash insere um novo elemento em uma posição determinada pelo valor hash da chave. O tempo de inserção geralmente é $\mathcal{O}(1)$ no melhor caso, assumindo que não ocorrem colisões. No entanto, no pior caso, quando ocorrem muitas colisões e é necessário lidar com várias entradas em uma mesma posição, a complexidade pode se tornar $\mathcal{O}(N)$, onde N é o número total de elementos.

3.2.2 Get (Pesquisa):

A operação `get` recupera o valor associado a uma chave específica na tabela hash. Similar à operação de inserção, o tempo de recuperação é $\mathcal{O}(1)$ no melhor caso, quando a chave é mapeada diretamente para a posição correta sem colisões. No entanto, no pior caso, a complexidade pode ser $\mathcal{O}(N)$ devido à necessidade de buscar em uma lista de colisão, onde N é o número total de elementos.

3.2.3 Complexidade de Espaço:

A complexidade de espaço de uma tabela hash, considerando uma implementação com uma estrutura de dados simples e linear, é $\mathcal{O}(N)$, onde N é o número total de elementos armazenados, como as estações de recarga.

3.3 Priority Queue

3.3.1 Heapify Up / Heapify Down:

As operações `heapify up` e `heapify down` são utilizadas para manter a propriedade de heap após inserções e remoções. Ambas as operações têm complexidade $\mathcal{O}(\log N)$, onde N é o número total de elementos na fila de prioridade. Isso ocorre porque essas operações ajustam a posição dos elementos no heap para restaurar a propriedade de heap, percorrendo no máximo o caminho da folha até a raiz ou vice-versa. A operação `heapify up` é utilizada ao inserir um elemento, enquanto `heapify down` é utilizada na remoção de um elemento. Em ambos os casos, a complexidade é $\mathcal{O}(\log N)$ devido ao balanceamento mantido na estrutura do heap.

3.3.2 Insert / Remove:

A operação `insert` em uma fila de prioridade envolve adicionar um novo elemento ao heap e, em seguida, aplicar `heapify up` para manter a propriedade de heap, resultando em uma complexidade de $\mathcal{O}(\log N)$. A ação de inserir em si tem complexidade $\mathcal{O}(1)$, enquanto o `heapify up` tem complexidade $\mathcal{O}(\log N)$, gerando uma complexidade final de $\mathcal{O}(\log N)$. O mesmo se aplica para a operação `remove`, com a diferença de que, como o elemento removido é sempre o topo, encontrar o elemento tem complexidade $\mathcal{O}(1)$, mas o `heapify down` ainda é necessário.

3.3.3 Complexidade de Espaço:

A complexidade de espaço de uma fila de prioridade é $\mathcal{O}(N)$, onde N é o número total de elementos. Isso se deve ao fato de que a fila de prioridade é uma estrutura linear, com cada elemento ocupando um espaço constante.

3.4 QuadTree

3.4.1 Find:

A operação `find` na estrutura `QuadTree` busca o quadrante que contém um ponto específico. Como a `QuadTree` é uma árvore que pode ficar desbalanceada, sua complexidade varia. No melhor caso, onde o ponto está diretamente na raiz ou na folha desejada, a complexidade é $\mathcal{O}(1)$. No pior caso, a complexidade de tempo para `find` é $\mathcal{O}(\log N)$, onde N é o número total de elementos na `QuadTree`. A profundidade máxima da árvore determina essa complexidade.

3.4.2 Insert:

A operação `insert` na `QuadTree` envolve a localização do quadrante correto para inserção e a posterior subdivisão desse quadrante. Considerando que as operações de atribuição e divisão são realizadas em tempo constante $\mathcal{O}(1)$, e que a inserção ocorre apenas nas folhas, a complexidade assintótica depende da operação `find`. Assim, a complexidade de tempo para `insert` é $\mathcal{O}(1)$ no melhor caso e $\mathcal{O}(N)$ no pior caso, onde N é o número total de elementos.

3.4.3 KNN (Consulta):

A operação `KNN` realiza a consulta dos k vizinhos mais próximos usando uma `PriorityQueue` para manter a ordem das distâncias. Como descrito na seção anterior, o uso dessa estrutura é benéfico, pois permite utilizar a distância do topo da fila de prioridades para evitar "descer" em quadrantes que não têm potencial para conter as estações mais próximas. A operação de busca é adaptável, pois no melhor caso percorre exatamente os k nós e ignora os demais. No pior caso, pode ser necessário percorrer todos os N pontos de recarga. Devido à `PriorityQueue`, a complexidade é $\mathcal{O}(k \log k)$ no melhor caso e $\mathcal{O}(N \log N)$ no pior caso. O grande diferencial dessa abordagem é seu desempenho médio, pois, em cenários típicos onde k é muito menor que N, a complexidade assintótica é sublinear em relação a N, particularmente logarítmica para valores pequenos de k.

3.4.4 Ativação/Desativação:

A operação de ativação ou desativação de uma estação de recarga tem operações de custo constante. Portanto, a complexidade dominante é o acesso e recuperação da estação por meio da tabela hash, que, como discutido anteriormente, tem complexidade $\mathcal{O}(N)$ no pior caso.

3.4.5 Complexidade de Espaço:

A complexidade de espaço do sistema é analisada com base nas principais estruturas de dados utilizadas. A `QuadTree` instanciada gera quatro novos nós a cada divisão. Assim, para n inserções, a complexidade de espaço é proporcional ao número de nós criados, resultando em uma complexidade de $\mathcal{O}(N)$, onde N é o número total de inserções. A estrutura vetorizada da árvore limita o tamanho máximo da estrutura, mas o crescimento ainda é linear em relação ao número de inserções.

4 Estratégias de Robustez

Para garantir robustez e programação defensiva, o código inclui extensivos tratamentos de exceções e validações. Verificações são realizadas para evitar acessos inválidos a índices e garantir operações seguras com dados. O sistema também valida arquivos de entrada e opções fornecidas, oferecendo feedback imediato em

caso de erros, como arquivos corrompidos ou opções inválidas. Métodos de busca nas classes tratam exceções como "item não encontrado", garantindo que o sistema lide adequadamente com situações inesperadas.

O programa é executado via scripts, minimizando erros de uso, e fornece mensagens de erro claras em caso de falhas. Essa abordagem reduz o risco de entrada inadequada e melhora a confiabilidade do sistema, tornando-o mais resiliente e fácil de manter.

5 Análise Experimental

Neste tópico, será abordada a análise experimental da versão atualizada da aplicação, onde diversas cargas de trabalho propostas serão testadas, e a eficiência e o desempenho desta aplicação serão comparados com a versão inicial (denominada "naive"). Para a realização dessa análise, utilizou-se um arquivo `.csv` contendo todos os endereços de Belo Horizonte (725.917 registros), que foram considerados como potenciais locais para a instalação de estações de recarga.

5.1 Número e Distribuição das Estações de Recarga

A avaliação das estações de recarga foi realizada considerando tanto o número total de estações quanto sua distribuição espacial. Nas imagens abaixo, é possível visualizar as abordagens testadas: um mapa denso e concentrado e um mapa esparso e desbalanceado. Nesta etapa, variou-se a densidade das estações em ambos os mapas para testar o desempenho das três operações principais (consulta, construção e atualização/ativação e desativação) tanto no modelo "naive" quanto no modelo utilizando `QuadTree`.

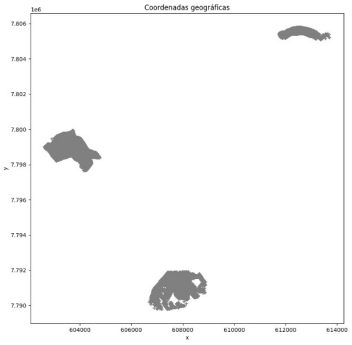

Figure 1: Mapa Desbalanceado

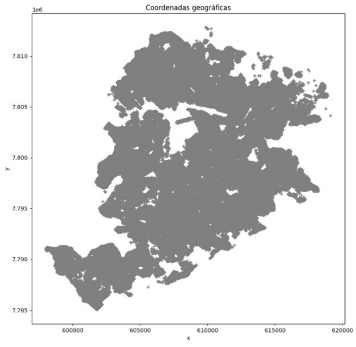

Figure 2: Mapa Denso

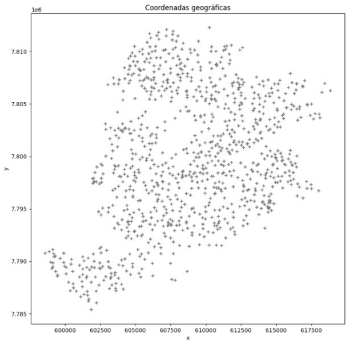

Figure 3: Variação de Densidade

Os resultados apresentados a seguir oferecem uma visão geral do comportamento das operações à medida que o número de pontos nos mapas, tanto com configuração uniforme e desbalanceada, tem seu tempo crescente. As configurações para as operações incluíram um número crescente de consultas aleatórias, com uma proporção de 0.1 para as operações de atualização (ativação e desativação), e o número de consultas foi padronizado para buscar os 10 pontos mais próximos.

Observa-se que, no caso das consultas, os gráficos tanto para a versão esparsa quanto para a versão uniforme utilizando a estrutura `QuadTree` mostraram uma maior "consistência". Isso ocorre porque foi utilizado um número fixo de k o que é esperado devido à análise anterior de complexidade $\mathcal{O}(k \log k)$ média.

Também é notável a tendência geral do código utilizando a versão "naive", que apresenta um crescimento quase linear em termos de tempo de execução. De modo geral, observa-se que a construção e a consulta são as operações mais custosas, independentemente da estrutura utilizada. Além disso, neste caso específico, não foi identificado um impacto significativo da dispersão ou do desbalanceamento nos gráficos, o que pode ser atribuído ao fato de que os pontos amostrados para as operações foram distribuídos aleatoriamente por todo o mapa, logo não necessáriamente ficaram todos longe ou próximos dos clusters (regios densas no mapa desbalanceado).

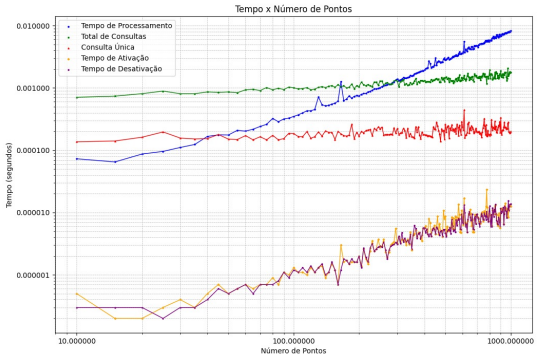

Figure 4: Quad - Uniforme

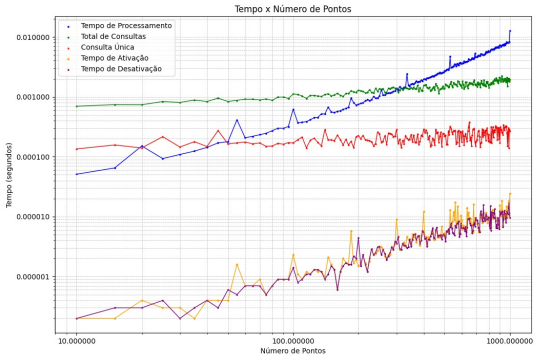

Figure 5: Quad - Desbalanceado

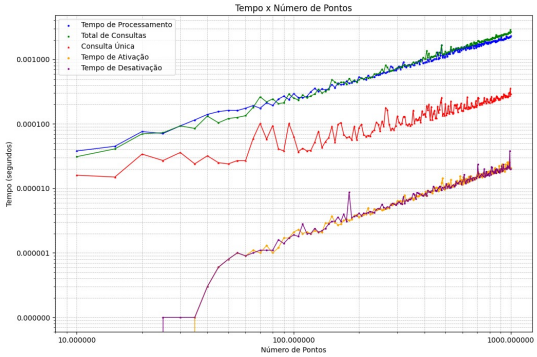

Figure 6: Naive - Uniforme

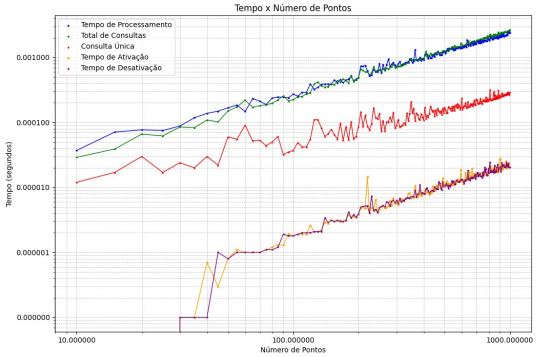

Figure 7: Naive - Desbalanceado

5.2 Número e natureza das consultas:

Neste trabalho, avaliamos a dimensão das consultas em relação ao seu número absoluto e à sua natureza, comparando a performance das operações de ativação e desativação. A análise abrange aspectos de concentração temporal e espacial no processo do programa.

Na Figura 8, apresentamos uma comparação entre o tempo de execução e o número de pontos por consulta. Observa-se que a performance do código utilizando o algoritmo Quad é significativamente superior em relação ao algoritmo Naive. A mesma tendência é observada ao comparar o tempo de execução com o número de consultas. À medida que a quantidade de consultas aumenta, a demanda sobre ambos os algoritmos cresce, porém o Quad demonstra uma vantagem considerável, exigindo menos tempo do que o Naive.

Esses resultados corroboram nossas observações anteriores, onde a complexidade $O(n \log n)$ do algoritmo Naive se mostra menos eficiente comparada à complexidade $O(k \log k)$ do algoritmo Quad, especialmente quando o valor de k é pequeno. A diferença de desempenho é evidente e destaca a superioridade do algoritmo Quad na execução das consultas.

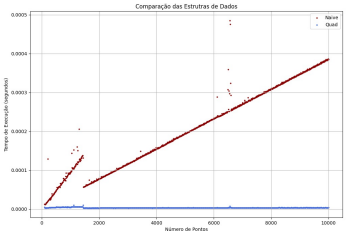

Figure 8: Consultas x Num. Pontos

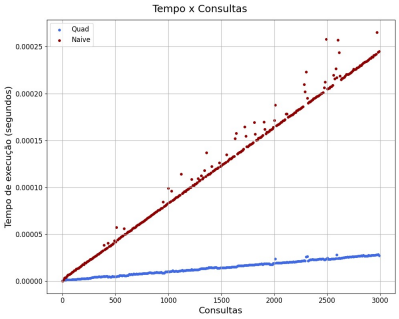

Figure 9: Consultas x Tempo

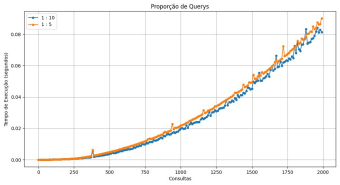

Figure 10: Proporção Atualizações x Consulta

Na última figura à esquerda, apresentamos uma comparação da proporção de atualizações (ativação e desativação) em relação ao número de consultas no programa. Nos experimentos realizados, foram analisadas duas condições: uma consulta gerava 5 atualizações e outra gerava 10 atualizações. Observou-se uma variação pequena entre esses cenários, indicando que uma quantidade menor de atualizações apresentou desempenho superior. Uma das hipóteses para essa observação é que as proporções de atualizações não são tão significativas a menos que as ativações e desativações estejam diretamente relacionadas com os pontos mais próximos da coordenada consultada. Em outras palavras, a eficácia das atualizações pode depender de quão precisamente as alterações correspondem às consultas realizadas.

5.3 Número e natureza das ativações e desativações:

Nesta seção, discutiremos o número e a natureza das ativações e desativações, sua dimensão e proporção. Analisaremos como esses fatores impactam o desempenho do programa.

Como observado, o custo associado às ativações e desativações no programa não depende diretamente da estrutura da árvore quad (quad tree), mas sim da tabela hash onde as ativações são endereçadas. Dessa forma, a complexidade de desempenho está mais intimamente relacionada à qualidade da tabela hash do que à própria árvore quad. A eficiência do programa é, portanto, influenciada pela ocorrência de colisões na tabela hash.

Embora tenhamos tentado criar gráficos para visualizar o impacto das ativações e desativações no desempenho do programa, não conseguimos identificar um impacto significativo. A seguir, apresentamos três gráficos que mostram o desempenho do programa para diferentes proporções de ativações e desativações. O primeiro gráfico ilustra uma proporção igual de ativações e desativações (1:1). O segundo gráfico mostra uma proporção de 90% de ativações e 10% de desativações. O terceiro gráfico é uma variação do segundo, mas com 10% de ativações e 90% de desativações. Observa-se que, ao aumentar a proporção de desativações em relação às ativações, o programa tende a apresentar um desempenho um pouco mais lento. Isso sugere que a predominância de desativações pode influenciar negativamente a eficiência do processo.

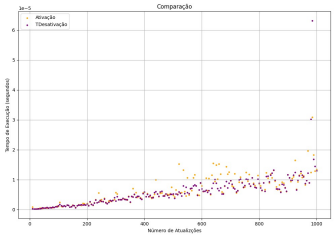

Figure 11: Proporção igual de ativações e desativações (1:1).

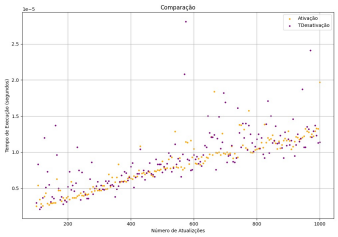

Figure 12: Proporção de 90% ativações e 10% desativações.

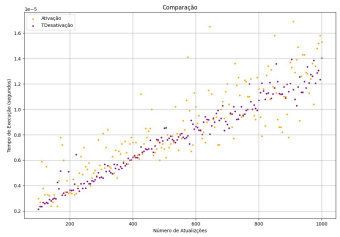

Figure 13: Proporção de 10% ativações e 90% desativações.

5.4 Localidade e Referência:

A **localidade de referência** é um conceito fundamental na análise de desempenho de sistemas de memória e cache. Refere-se ao comportamento dos acessos à memória em relação à proximidade temporal e espacial. Esse conceito pode ser dividido em duas categorias principais: localidade temporal e localidade espacial.

A **localidade temporal** refere-se à tendência de acessar um mesmo dado repetidamente em um curto intervalo de tempo. Por outro lado, a **localidade espacial** refere-se à tendência de acessar dados que estão fisicamente próximos uns dos outros na memória.

Para analisar a localidade de referência, utilizei o arquivo `memlog` para gerar gráficos que mostram a relação entre os endereços de memória e o tempo de acesso. Esses gráficos permitem observar os padrões de localidade de referência, com as leituras destacadas em roxo e as escritas em amarelo.

No primeiro gráfico, que representa o preenchimento da estrutura de dados, é possível observar a quantidade de leituras e escritas realizadas. Como esperado, o gráfico mostra muitas leituras e escritas, pois é necessário localizar um endereço para realizar a inserção (escrita) de um elemento. Este gráfico evidencia o comportamento típico de estruturas de dados que requerem tanto leitura quanto escrita durante o processo de inserção e atualização.

O segundo gráfico foca apenas nas consultas. Nele, observamos apenas leituras, com um padrão consistente de endereços e tempos de acesso. Isso sugere que os dados são armazenados em endereços espacialmente próximos, refletindo uam boa localidade espacial e evidenciando que a estrutura de dados mantém dados próximos uns dos outros para otimizar a performance das consultas.

O terceiro gráfico corresponde às atualizações, ativação e desativação dos dados. Nele, podemos observar uma combinação de escritas (que indicam mudanças de estado de atividade) e um grande número de apenas leituras, que pode indicar que a estação já estava ativada ou desativada. As leituras e escritas mostram variações espaciais e temporais, com algumas regiões mais aleatórias, mas ainda seguindo um padrão geral. Este gráfico ilustra como as atualizações na estrutura de dados mesmo em operações mais dispersas, mentém uma certa consistência temporal e espacial.

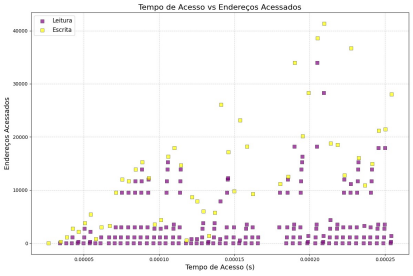

Figure 14: Querys Diversas

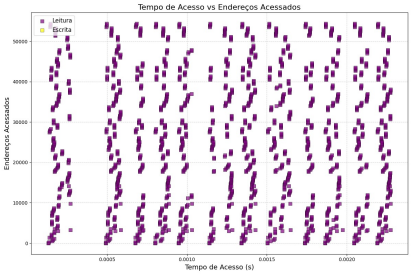

Figure 15: Apenas Consultas

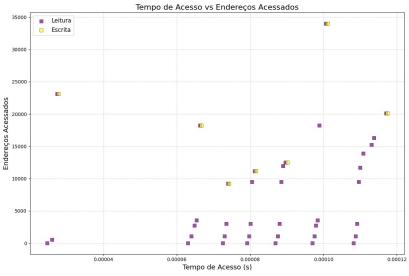

Figure 16: Apenas Atualizacoes

Por fim, abaixo, é possivel visualizar a distância de pilha para uma entrada de querys mistas medida durante a execução do programa:

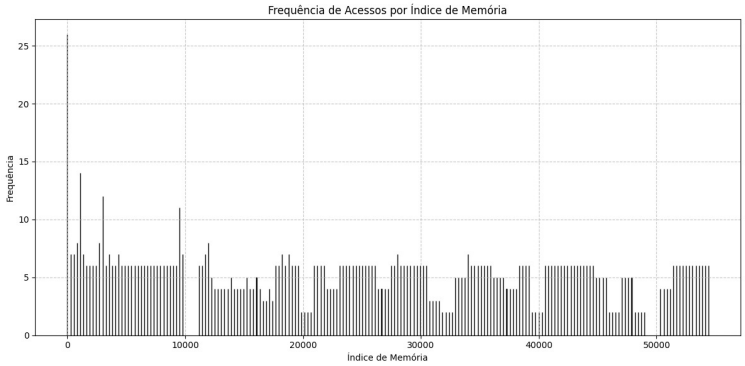

Figure 17: Apenas Atualizacoes

A distância de pilha é uma métrica crucial para analisar o comportamento dos acessos a dados em sistemas de memória. Ela quantifica a diferença entre o número de operações de atualização (ativação e desativação) e o número de acessos subsequentes aos mesmos endereços de memória.

No gráfico apresentado, observa-se que a maioria dos acessos ocorre de forma quase constante, com alguns picos ocasionais. Esses picos podem indicar períodos de maior atividade, nos quais o sistema realiza mais operações de atualização ou acessos a dados. Em geral, a pilha exibida sugere que os acessos subsequentes à memória têm uma frequência relativamente constante, com variações pontuais que refletem esses momentos de maior atividade, como no início da execução, quando os itens são inseridos na estrutura de dados.

6 Conclusões

Em suma, este trabalho proporcionou reflexões valiosas sobre o uso de estruturas de dados e como elas podem ser eficazmente empregadas para resolver problemas específicos. No caso do problema que buscamos resolver, a utilização da quadtree proposta levou a resultados mais eficientes e de acordo com as análises teóricas e experimentais, em comparação com a solução inicialmente proposta. Observamos que essa estrutura é particularmente eficiente para problemas envolvendo dados geolocalizados, grandes volumes de dados e consultas com um valor de k potencialmente pequeno.

Durante a execução média do problema, conseguimos manter uma complexidade sublinear, que era o objetivo desejado. Além disso, foi possível observar a importância da integração com outras estruturas de dados, como a fila de prioridade e o hash, e como essas estruturas impactam significativamente o desempenho do programa. A análise também revelou como a memória foi gerenciada no programa, destacando a importância da compreensão e otimização desse aspecto relacionado principalmente à localidade e referência.

Essas observações ajudam a consolidar o entendimento sobre como diferentes estruturas e técnicas podem ser aplicadas para melhorar o desempenho e a eficiência dos sistemas de processamento de dados. As lições aprendidas oferecem uma base sólida para futuros trabalhos e aplicações nesta área.

7 Bibliografia

References

[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, *Introduction to Algorithms*, 3ª edição, MIT Press, 2009.

[2] N. Ziviani, *Projeto de Algoritmos com Implementações em Pascal e C*, 3ª edição, Cengage Learning, 2011.