Trabalho Prático 3

[**Luis Antonio Duarte Sousa - 2023001964**]

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil

luisduarte.dev@gmail.com

1 Introdução

Com o avanço da tecnologia e a crescente demanda por soluções mais eficientes e sustentáveis, a BiUaiDi, uma renomada fabricante de carros elétricos, se vê diante do desafio de modernizar seu aplicativo de identificação de estações de recarga. Em Belo Horizonte, cidade caracterizada por seu relevo desafiador, a necessidade de recarga dos veículos é uma constante, e a empresa busca oferecer uma solução que atenda a essa demanda de maneira eficaz e em tempo real. O problema enfrentado atualmente pelo aplicativo é sua ineficiência na identificação das estações de recarga mais próximas, devido à complexidade de sua estrutura de dados e à falta de dinamismo para lidar com a ativação e desativação das estações.

Neste trabalho, será apresentada uma solução que envolve a reformulação completa do aplicativo, passando pela flexibilização do conjunto de estações de recarga, implementação de uma estrutura de dados mais eficiente, como a QuadTree, e a extensão do aplicativo para uma arquitetura de dois níveis de memória utilizando LRU. Essa nova abordagem visa não apenas melhorar o desempenho do sistema, mas também garantir que ele seja escalável e capaz de se adaptar às mudanças dinâmicas no ambiente urbano. A solução proposta será avaliada experimentalmente, comparando o desempenho das versões antiga e nova do aplicativo, com o objetivo de demonstrar a superioridade da nova implementação em termos de eficiência e usabilidade.

2 Implementação

2.1 Configurações do Sistema

- Sistema Operacional: Windows 11 pro
- Ambiente de execução: WSL2
- Linguagem: C++
- Compilador: GCC 13.2
- Processador: AMD Ryzen 5 735HS
- RAM: 24 GB

2.2 Definições

- ***INF***: Representação do infinito utilizando 0x3f3f3f3f

2.3 Utilitários Implementados

2.3.1 Funções Axiliares

- ***max(a,b)***: Função que retorna o máximo entre dois doubles utilizada para calculo de distância;
- ***calculateDistance(a,b)***: Função que retorna a distância entre dois pontos a e b;
- ***minDistanceToQuadrant(minX,maxX,minY,maxY,t)*** Função que retorna a distância mínima de um ponto t para um quadrante com limites minX,maxX,minY e maxY;
- ***MedianInsQuickSort(S, l, r)***: Função que ordena um vetor S de coordenadas das estações por x e y, utilizado para tornar as inserções mais eficientes para o balanceamento pegando as medianas das partições e inserindo. A implementação do quicksort utiliza duas otimizações que é a escolha de pivot como mediana dos extremos e meio.Além disso, o uso de insertion sort para partições pequenas para melhor desempenho de tempo.

2.3.2 Estruturas de dados

- ***RechargeStation***: Estrutura de dados utilizada para representar uma estação de recarga, armazenando todos os seus dados.
- ***Point***: Estrutura utilizada para representar um ponto no plano cartesiano (R^2) com dois doubles representando as coordenadas x e y;

- ***PointID***: Estrutura auxiliar utilizada apenas para tornar mais eficiente a ordenação e inserção das estações pois armazena apenas os dados necessários na quadtree que são id e um ponto, assim, evitando swaps de itens grandes;
- ***DistPair***: Estrutura utilizada dentro no maxheap para que durante a busca dos knn o id da estação seja armazenado em conjunto com sua distancia até o ponto alvo.
- ***Tabela Hash***: A tabela hash foi implementada com intuito de que durante as ativações/desativações pelo id da estação, as coordenadas delas sejam adquiridas em complexidade de tempo próxima a **O(1)** uma vez que a tabela tem tamanho fixo/ controlado evitando sobrecarga e as chaves (ids) são unicas para cada endereço permitindo um endereçamento como string em que cada caractére é poderado pela posição. As colisões são tratadas com endereçamento aberto quadrático e os principais métodos são **insert(s)** que insere na tabela uma estação s e **pesquisa(id)** que pesquisa pelo id os dados da estação.
- ***Max Heap***: O maxheap foi implementado com intuito de armazenar os k vizinhos mais próximos encontrados até o momento durante a busca na árvore em complexidade de tempo **O(logn)**. Ele é importante pois sua raiz mantem a maior distância do alvo encontrada ate o momento e ajuda a tornar a busca mais eficiente e precisa. A implementação usada é a vetorizada e tem como principais métodos **top()** que retorna a raiz, **enqueue(a**) que insere no heap um DistPair e **getSize()** que retorna o tamanho atual do heap;
- ***QuadNode***: Essa estrutura representa cada nó da árvore Quadtree e armazena um ponto que são as coordenas, um booleano que representa estar ou não ativa, o id da estação, os indices dos 4 filhos e os limites horizontais e verticais da região representados por 4 doubles.
- ***Point QuadTree***:Estrutura principal que armazena um array de QuadNodes e possui os principais métodos. ***activate/deactivate(point,id)*** que ativa/desativa o ponto passado como parâmetro em complexidade O(logn) uma vez que utiliza as coordenadas para caminhar diretamente até o nó buscado usando uma lógica parecida a árvore de busca binária em que compara as coordenadas. ***knn(p, k)*** que busca os k pontos pertencentes a árvore com menor distancia ao ponto p e utiliza como lógica principal de otimização que se a distancia minima do ponto p até certo quadrante é maior que a distancia do k-ésimo elemento no heap (ou seja, o vizinho mais distante entre os k mais próximos) então aquele quadrante não pode haver vizinhos importantes e não deve ser explorado.
- ***Lista Duplamente Encadeada***: Foi implementada com intuito de auxiliar na politica de gerenciamento de páginas baseada em LRU uma vez que permite inserções no começo da lista em **O(1)** , remoções no fim em **O(1)** e buscas em **O(n)**. Isso é util pois a página no fim (LRU page) será removida diversas vezes enquanto as inserções irão ocorrer no começo que representa a página mais recentemente usada.

2.4 Estratégia de Resolução e Politica de Extensão

A estratégia adotada no código utiliza uma QuadTree vetorizada para otimizar a inserção, busca e manipulação de pontos georreferenciados, como estações de recarga. A QuadTree é particularmente eficiente para dados bidimensionais, organizando os pontos em uma estrutura hierárquica que permite consultas rápidas, como encontrar os k vizinhos mais próximos (k-NN) de um ponto específico.

Inserção Otimizada:

A inserção dos pontos na QuadTree é realizada de maneira otimizada através da escolha das medianas dos arrays ordenados pelas coordenadas x e y, minimizando o desequilíbrio na árvore e garantindo que os pontos sejam distribuídos de maneira eficiente entre os quadrantes. O algoritmo de inserção verifica em qual quadrante o novo ponto pertence (Nordeste, Noroeste, Sudeste ou Sudoeste) e então recursivamente insere o ponto no quadrante correspondente. Se o quadrante ainda não estiver ocupado, ele cria um novo nó e atualiza os limites (minX, maxX, minY, maxY) do nó para refletir a região espacial que ele cobre.

Busca e Ativação/Desativação:

Para operações de busca, como a ativação e desativação de pontos, a estratégia é localizar o ponto na QuadTree utilizando as coordenadas e o identificador (id) associados. A função find percorre a QuadTree começando pela raiz e seguindo o caminho determinado pelas coordenadas do ponto até encontrar o nó correspondente. Se o ponto é encontrado, ele pode ser ativado ou desativado, alterando o estado do nó (isActive). Essas operações são eficientes, pois a estrutura

da QuadTree permite que a busca seja realizada em tempo sub-linear, dado que a árvore está equilibrada.

Consulta de k-Vizinhos Mais Próximos (k-NN):

A consulta k-NN é realizada utilizando uma heap máxima (maxheap) para armazenar os k vizinhos mais próximos durante a busca. O algoritmo inicia a partir da raiz da QuadTree e, para cada nó, calcula a distância do ponto alvo aos pontos armazenados. Se a heap ainda não contém k elementos, o novo ponto é adicionado diretamente; caso contrário, o ponto é adicionado apenas se sua distância for menor que a distância máxima armazenada na heap.

Para otimizar a busca, a função identifica o quadrante primário onde o ponto alvo estaria localizado e o explora primeiro. Em seguida, ela avalia se os outros quadrantes devem ser explorados com base na distância mínima entre o ponto alvo e os limites do quadrante. Se essa distância mínima for menor que a distância máxima na heap, o quadrante é explorado, garantindo assim que todos os possíveis vizinhos próximos sejam considerados.

Uso de Vetorização:

Ao invés de utilizar alocação dinâmica de memória tradicional, a QuadTree é implementada utilizando um vetor de nós, onde os filhos de um nó são referenciados por índices dentro do vetor. Essa abordagem melhora a localidade de referência e pode ser mais eficiente em termos de desempenho, especialmente em arquiteturas que se beneficiam de acesso contíguo à memória. Além disso, essa estrutura vetorizada facilita a manipulação e a destruição da árvore, uma vez que os índices podem ser gerenciados diretamente.

Essa estratégia combinada torna o código eficiente para lidar com grandes quantidades de dados espaciais, otimizando operações de inserção, busca e consulta de proximidade.

Politica de páginas:

A politica de páginas escolhida foi a LRU uma vez que páginas de memória que foram visitadas mais recentemente tem maior chance de serem utilizadas novamente.A ideia principal por trás da LRU é substituir a página na memória que não foi utilizada há mais tempo.

A LRU mantém um registro de quais páginas na memória foram acessadas recentemente. Isso é feito através de uma lista duplamente encadeada onde as páginas são ordenadas pela última vez que foram acessadas.

Quando uma página é acessada (seja leitura ou escrita), ela é movida para o início da lista, indicando que foi a última página usada. Se a página acessada já estava na lista, ela é simplesmente reposicionada no início. Se a página não estava na lista (indica um "page fault"), a página menos recentemente usada (que está no final da lista) é removida para liberar espaço para a nova página.

Quando a memória está cheia e uma nova página precisa ser carregada, a página que está há mais tempo sem uso (localizada no final da lista) é removida. A nova página é então carregada e colocada no início da lista.

3 Análise de Complexidade

Inicialmente, é importante notar a relevância da distribuição e da ordem de inserção dos pontos para garantir um comportamento eficiente como o desejado. Os métodos implementados dependem de uma árvore bem balanceada, por isso, é necessário e recomendado um pré-processamento antes da inserção dos dados na árvore.

O programa se inicia com a leitura dos dados em $\mathcal{O}(n)$ e a armazenagem dos dados informativos em uma Tabela Hash (aproximadamente $\mathcal{O}(1)$, dado que o fator de carga é baixo), além da armazenagem dos dados de Id e Coordenadas dentro de um array. Em seguida, o array é ordenado em $\mathcal{O}(n \log n)$ pelas coordenadas x e y, e a mediana de cada partição do array ordenado é inserida na quadtree sequencialmente com o intuito de gerar uma árvore mais balanceada. Cada inserção é garantidamente $\mathcal{O}(\log n)$ se a árvore estiver bem balanceada, uma vez que usamos as coordenadas para procurar a posição correta. No entanto, no pior caso, em uma árvore desbalanceada, a complexidade pode ser $\mathcal{O}(n)$, no qual a árvore se comporta como uma lista encadeada. Temos n inserções, logo, inserimos em $\mathcal{O}(n \log n)$. Dito isso, percebe-se que o processo de criação da quadtree tem complexidade aproximada de:

- **Melhor caso:**

$$\mathcal{O}(n) + \alpha\mathcal{O}(1) + 2 \times \mathcal{O}(n \log n) = \mathcal{O}(n \log n)$$

- **Pior caso:**

$$\mathcal{O}(n) + \alpha\mathcal{O}(1) + \mathcal{O}(n \log n) + \mathcal{O}(n^2) = \mathcal{O}(n^2)$$

No quesito espaço, temos $\mathcal{O}(2n)$ para a Tabela

Hash, $\mathcal{O}(n)$ para o array e $\mathcal{O}(n)$ para a quadtree. Portanto, a complexidade de espaço é dada por:

$$4 \times \theta(n) = \theta(n)$$

Ativações/Desativações

Para os métodos de ativar e desativar estações, utilizamos uma lógica parecida com a da inserção, para encontrarmos o nó desejado na árvore, tornando o processo $\mathcal{O}(\log n)$ no melhor caso e $\mathcal{O}(n)$ no pior caso. O espaço utilizado é constante $\mathcal{O}(1)$.

K Vizinhos Mais Próximos

O processo de adquirir os k vizinhos mais próximos começa com a inicialização de um max-heap de tamanho k, cujas complexidades são $\mathcal{O}(\log n)$ para inserção, $\mathcal{O}(\log n)$ para remoção e $\mathcal{O}(1)$ para consulta do máximo (`top()`). Em seguida, iniciamos uma consulta a partir da raiz da árvore de pontos, com ênfase inicial nos pontos do mesmo quadrante do ponto alvo, calculando a distância ao alvo e inserindo-os no heap, explorando apenas nós cuja distância da região ao ponto alvo seja menor que o maior elemento do heap. Com esse procedimento, é possível obter os k vizinhos mais próximos em uma complexidade próxima de $\mathcal{O}(k \log n)$. O espaço a mais utilizado depende do numero de vizinhos requerido para criação do heap sendo então $\mathcal{O}(k)$.

4 Estratégias de Robustez

O código implementa várias estratégias de robustez através de mecanismos de programação defensiva e tolerância a falhas, garantindo a confiabilidade da QuadTree em diferentes cenários de uso. Por exemplo, durante a inicialização da QuadTree, o código verifica se a memória alocada é suficiente para a capacidade especificada; caso contrário, a execução é interrompida com uma mensagem de erro clara, evitando falhas inesperadas. Adicionalmente, as operações de busca, inserção e manipulação dos nós da QuadTree são cuidadosamente implementadas para lidar com situações em que o nó filho esperado não exista (indicado por índices de valor -1), prevenindo acessos inválidos a memória. Essa abordagem de programação defensiva ajuda a evitar erros comuns, como desreferenciamento de ponteiros nulos, e assegura que o programa se comporte de maneira previsível mesmo em casos de dados incompletos ou incorretos. Além disso, as funções de ativação e desativação verificam explicitamente o estado atual do nó antes de realizar qualquer modificação, garantindo que as operações sejam idempotentes e evitando alterações indesejadas. Esses mecanismos, em conjunto, fortalecem a robustez do código, tornando-o mais resistente a falhas e mais confiável em operações críticas.

5 Análise Experimental

5.1 Comparação com Versão Antiga

A versão inicial do aplicativo (naive) é capaz de consultar apenas 10 vizinhos mais próximos com uma complexidade de **O(nlogn)**. Além disso, o estado das estações e as consultas são estáticos. Em um cenário onde é necessário realizar apenas uma consulta para encontrar 10 vizinhos e onde as estações permanecem inalteradas, essa abordagem pode ser aceitável. No entanto, em cenários dinâmicos com várias consultas sequenciais e mudanças nos estados das estações, essa solução torna-se ineficiente. Isso ocorre porque, para cada consulta, seria necessário realizar novamente uma operação completa de **O(nlogn),** e, para alterar o estado das estações, seria necessário modificar manualmente o dataset do aplicativo para adicionar ou remover estações. A proposta de melhoria

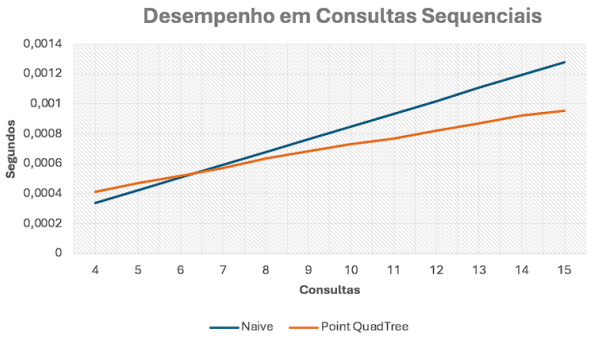

Figura 1: Comparação de desempenho das consultas (tempo total de leitura,execução e impressão)

envolve a construção de uma árvore eficiente, que inicialmente tem um custo superior ao da consulta de vizinhos da versão antiga. Este custo superior esta ligado a construção eficiente da árvore de busca, o que ocorre apenas uma vez durante toda a execução do aplicativo. No entanto, nas consultas subsequentes de vizinhos próximos, a nova abordagem reduz o tempo de processamento em pelo menos 2/3 em comparação com a versão antiga. Isso é ilustrado na Figura 1.È importante notar que a versão nova é mais rápida mesmo que versão naive não tenha que ler arquivos, fazer muitas comparações e nem pré-processamento.

5.2 Memória, Localidade e Referência

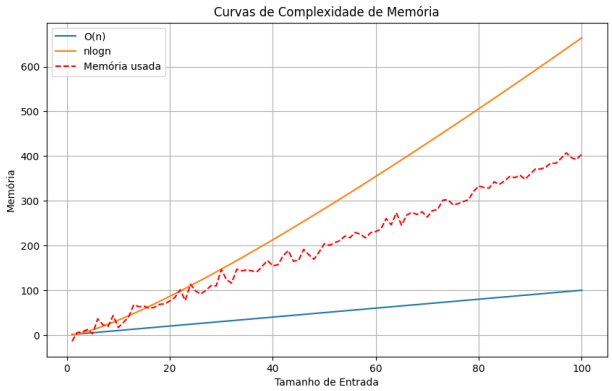

Figura 2: Memória x Tamanho

Como esperado a quantidade de memória é totalmente dependente do tamanho da entrada e varia também de acordo com k por causa do heap durante buscas.

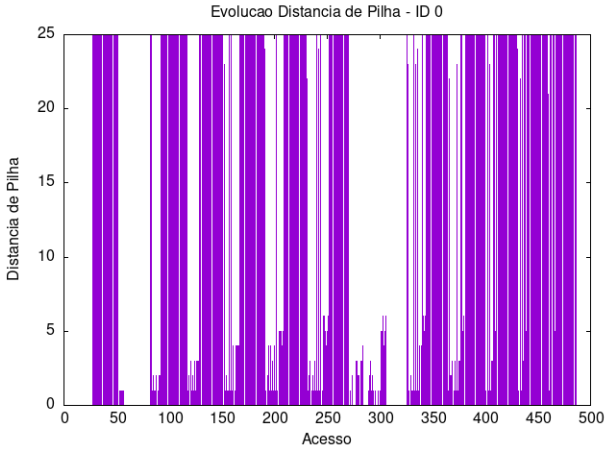

Figura 3: Evolução Dist.Pilha

Há uma distribuição bastante variada das distâncias de pilha, com muitas barras que chegam ao valor máximo (25) e outras que estão mais próximas de zero.

As barras altas representam que os dados não foram reutilizados rapidamente enquanto barras baixas significa que os mesmos dados estão sendo reutilizados rapidamente, sugerindo uma boa localidade temporal (mesmos dados acessados frequentemente).

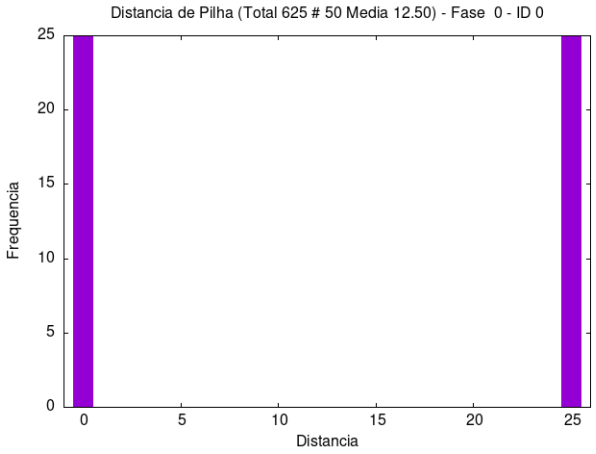

Figura 4: Distancia de pilha

5.3 Efeitos do Numero e Distribuição das Estações

Para avaliar o desempenho em função do numero de Estações será testado o código com uma desativação e uma consulta com k = 10 variando o numero de estações. Além disso, o tempo total foi divido por 10 para fins de visualização no gráfico. O total explodiu graças as extensas leituras de arquivo e impressões dos resultados.

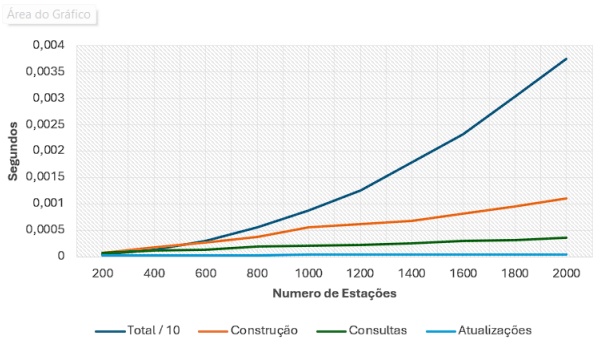

Figura 5: Tempo x Estações

Dado os resultados apresentados, é notório que a função mais afetada pelo aumento do número de nós na árvore é a de construção, uma vez que as inserções possuem complexidade **O(nlogn)**. Além disso, a função de KNN mostrou um comportamento eficiente, mantendo-se próxima de **O(klogn)**, enquanto as funções de atualização foram pouco afetadas. O único fator que poderia impactar essas funções seria uma árvore extremamente desbalanceada ou com profundidade muito alta.É essencial analisar e tratar a distribuição dos pontos antes da inserção, uma vez

que o balanceamento da QuadTree está intimamente ligado à distribuição e à ordem de inserção dos pontos.Além disso, é importante ressaltar que a densidade da região e diretamente proporcional a altura então regiões muito densas tendem a desbalancear a árvore caso não tratado.

Por exemplo, considere um conjunto de pontos {A(0,0), B(2,2), C(4,4), D(6,6), E(8,8), F(10,10)} com a seguinte distribuição:

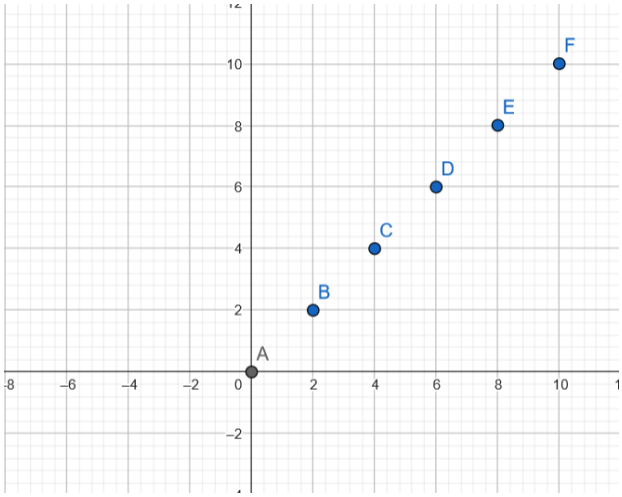

Figura 6: Distribuição Exemplo

Se os pontos forem inseridos em ordem lexicográfica, a árvore quaternária resultante esta representada na figura 4:

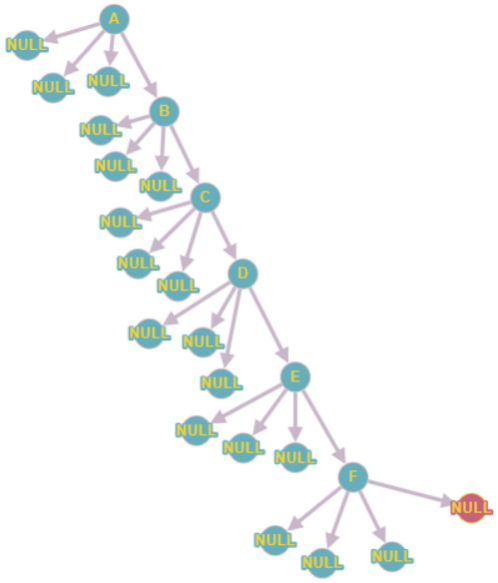

Figura 7: Árvore Desbalanceada

Com a árvore nesse estado, todos os procedimentos que antes tinham complexidade **O(logn)** se tornam **O(n)**. Portanto, é essencial tratar distribuições concentradas em quadrantes específicos (muito densos). Observe agora como a árvore ficaria ao utilizar o método de inserções baseado em medianas:

- **Ordenação**: O conjunto {A(0,0), B(2,2), C(4,4), D(6,6), E(8,8), F(10,10)} já está ordenado, não sendo necessária uma nova ordenação.
- **Ordem de Inserções por Medianas**: C(4,4), B(2,2), A(0,0), E(8,8), D(6,6) e F(10,10)
- **Árvore Resultante**:

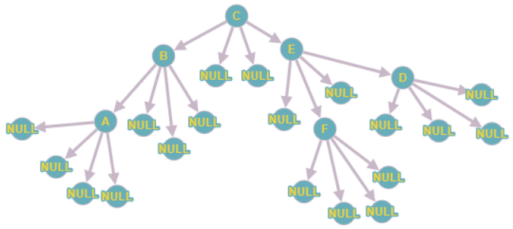

Figura 8: Árvore Balanceada

Todas as operações realizadas em uma árvore com distribuição ruim tendem a ter um tempo de execução linear, evidenciando os efeitos negativos do desbalanceamento. Portanto, a utilização de técnicas como a inserção por medianas pode ser crucial para manter a eficiência das operações em uma QuadTree.

5.4 Efeitos do Numero e Natureza de Consultas

O valor de k=5 foi selecionado para as consultas de teste, e a análise foi conduzida comparando dois cenários: um com todas as estações ativas e outro com 70% das estações ativas. (Para esta análise, o tempo de leitura dos arquivos foi desconsiderado.)

O impacto das desativações no tempo de execução das consultas se mostrou relativamente pequeno. Isso pode ser explicado pelo aumento da profundidade explorada na QuadTree para encontrar os vizinhos mais próximos em alguns casos. Quando certas estações estão desativadas, a busca pode ter que explorar mais nós, o que pode aumentar ligeiramente o tempo de execução. No entanto, o impacto não é significativo, indicando que o algoritmo é robusto o suficiente para lidar com desativações sem uma perda substancial de desempenho.

Em cenários onde há picos temporais de consul-

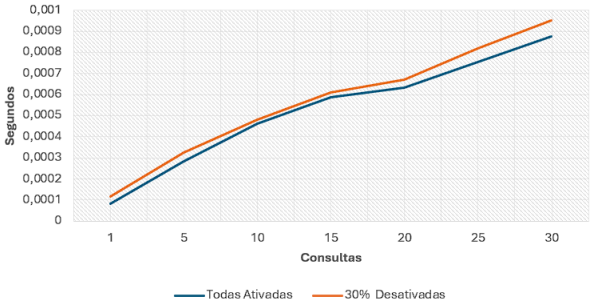

Figura 9: Efeito de desativações nas consultas

tas, como em momentos específicos durante um trajeto, a vetorização e a localidade de referência desempenham um papel crucial. Quando consultas são feitas em intervalos muito curtos, a localidade de referência tende a ser melhorada devido à reutilização de dados que já estão no cache. Isso significa que, após a primeira consulta, as consultas subsequentes podem ser realizadas mais rapidamente. Isso é evidenciado na forma da curva na Figura 6, onde o tempo de execução não cresce de forma linear quando múltiplas consultas são feitas em sequência.

A concentração espacial das consultas também exerce uma influência significativa no desempenho do algoritmo KNN. Existem dois cenários principais a serem considerados:

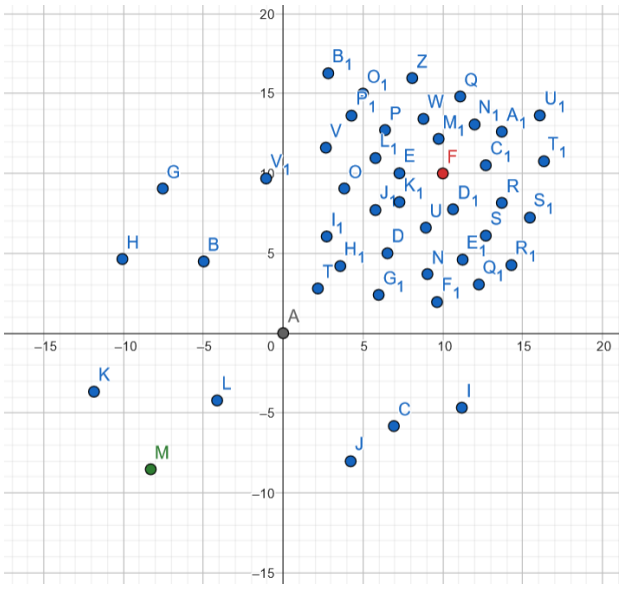

Figura 10: Espaço com regiões esparsas e densas

Busca em Regiões Densas (F): Nas regiões densas, onde há muitas estações próximas umas das outras, a QuadTree pode ter uma profundidade maior. Isso resulta em um maior número de nós a serem explorados durante a consulta KNN, o que aumenta o tempo de execução. A densidade dos pontos faz com que a busca precise analisar mais subdivisões da árvore para garantir que os vizinhos mais próximos sejam identificados corretamente.

Busca em Regiões Esparsas (M): Em contrapartida, em regiões esparsas, onde as estações estão mais distantes umas das outras, a QuadTree tende a ser menos profunda. Isso permite que as consultas KNN sejam executadas mais rapidamente, pois menos nós precisam ser explorados para encontrar os vizinhos mais próximos. A menor densidade facilita a identificação dos vizinhos, resultando em um tempo de execução mais rápido.

5.5 Efeitos do Numero e Natureza das Ativações e Desativações

Para esse experimento foi mantida apenas uma consulta com k = 20 e variando o numero de desativações.

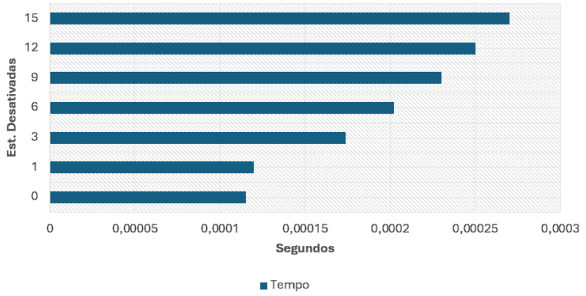

Figura 11: Desativações

As ativações e desativações não alteram a estrutura da árvore, apenas um valor booleano, portanto, o impacto em termos de número é muito pequeno. No entanto, dependendo dos pontos desativados, o algoritmo de busca k-NN pode precisar procurar por um substituto. Se esse substituto estiver distante ou em um nível mais profundo da árvore, isso pode resultar em uma busca mais extensa e, consequentemente, afetar o desempenho do algoritmo.

5.6 Razão entre memória primária e secundária

Esse experimento será conduzido em um cenário de duas consultas de 10 vizinhos enquanto a constante "MEMTOSWAPRATIO"é variada. Esse fator representa a razão entre memória primária e secundária ,

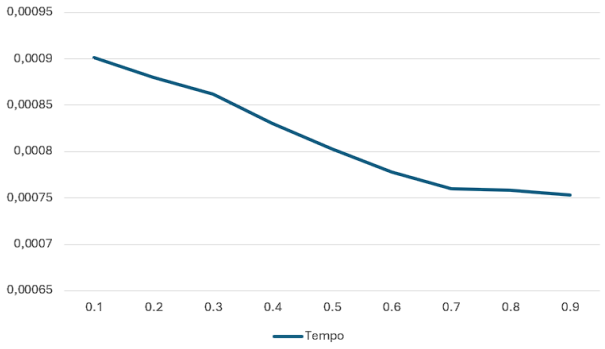

Figura 12: Tempo X Razão

logo, quanto maior mais conteudo pode ser mantido na memória primária. As razões que serão utilizadas serão {0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9}. O aplicativo será executado 10 vezes para cada razão de memória e será utilizado a média dos valores.

Quando você aumenta a razão, mais páginas podem ser mantidas na memória primária, o que significa que o número de trocas de páginas entre a memória primária e a secundária tende a diminuir.Como a memória primária (RAM) é significativamente mais rápida que a memória secundária (swap, geralmente no disco), manter mais páginas na RAM pode reduzir a latência de acesso, pois menos operações de swap são necessárias.Logo, faz sentido que o tempo se reduza um pouco conforme a razão aumenta.

5.7 Número de estações consideradas nas consultas

Serão realizadas uma consulta de k vizinhos em que será variado o k em função do tempo.Além disso, também será registrado o numero de pontos explorados antes de chegar na solução dado um conjunto de 50 estações (pontos).

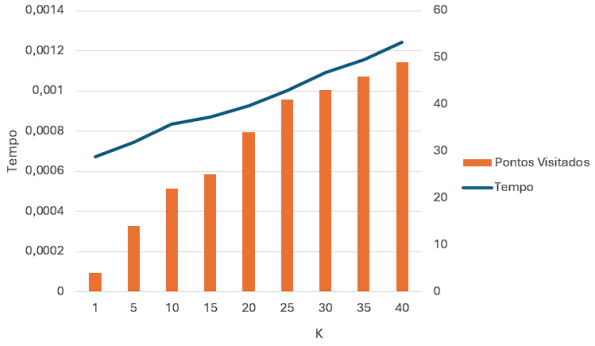

Figura 13: Tempo x K x Pontos Visitados

Quando K aumenta, o algoritmo precisa acessar mais estações, o que pode levar a mais páginas sendo carregadas na memória primária (RAM). Se o número de páginas que o algoritmo precisa acessar simultaneamente exceder a capacidade da memória primária, o sistema de memória virtual precisará usar a memória secundária (swap) para armazenar as páginas que não cabem na RAM. ALém disso, expandir o número de estações consideradas (K) resulta em maior tempo de execução e mais pontos visitados. Portanto, ao escolher K , é importante equilibrar a precisão (que pode aumentar com um K maior) com a eficiência (tempo de execução). No quesito

6 Conclusão

Este trabalho apresentou uma reformulação significativa do aplicativo de identificação de estações de recarga da BiUaiDi, adaptando-o para lidar com as exigências de um ambiente urbano dinâmico e desafiador como o de Belo Horizonte. A implementação de uma estrutura de dados mais eficiente, utilizando a QuadTree, combinada com a extensão para uma arquitetura de dois níveis de memória com política LRU, resultou em uma solução que não apenas melhora o desempenho do sistema, mas também o torna mais escalável e adaptável a mudanças.

A análise experimental demonstrou que, embora o custo inicial de construção da nova árvore seja mais elevado em comparação com a abordagem anterior, os ganhos de eficiência nas consultas subsequentes superam amplamente esse custo. A nova versão do aplicativo consegue realizar consultas de vizinhos próximos de maneira significativamente mais rápida, mesmo em cenários dinâmicos onde as estações são ativadas e desativadas constantemente. A comparação com a versão antiga evidenciou a superioridade da nova implementação, que reduz o tempo de processamento em pelo menos 2/3, mesmo sob condições de alta demanda.

Além disso, as estratégias de robustez incorporadas ao código garantem a confiabilidade do sistema em diferentes cenários, prevenindo falhas e assegurando que o aplicativo se comporte de maneira previsível mesmo em situações adversas.

Em suma, a solução proposta não só resolve os problemas de ineficiência e falta de dinamismo da versão anterior do aplicativo, como também estabelece uma base sólida para futuras melhorias e adaptações, permitindo à BiUaiDi continuar oferecendo um serviço de alta qualidade em um mercado em constante evolução.

7 Referencias

Wagner Meira. (2024). Slides Virtuais de Estrutura de dados

https://www.w3schools.com/

https://en.wikipedia.org/wiki/Quadtree