Trabalho Prático 3 - Estações de Recarga da BiUAIDi

Cauã Magalhães Pereira
2023028234

Departamento de Ciência da Computação - Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brazil

cauamp11@gmail.com

1 Introdução

1.1 Contexto

A fabricante de carros elétricos BiUaiDi deseja modernizar seu aplicativo de identificação de estações de recarga. O aplicativo deve ser capaz de, dada a posição atual do carro, identificar as estações de recarga mais próximas.

1.2 Problema

A versão atual do aplicativo da BiUaiDi é ineficiente, pois calcula a distância entre o ponto de origem e todas as estações de recarga disponíveis, que são cerca de 100 na cidade de Belo Horizonte. Além disso, o aplicativo não permite a ativação e desativação temporária de estações de recarga, nem a adição de novas estações em tempo real.

1.3 Solução Proposta

Para resolver essas limitações, propomos uma estratégia em cinco estágios:

- Evoluir o aplicativo para que trabalhe com um conjunto variável de pontos de recarga e funcione como um serviço que receba variados pontos de origem.
- Substituir a estrutura de dados original por uma mais eficiente, utilizando uma QuadTree para armazenar as estações de recarga.

2 Método

Nesta seção, abordaremos o Primeiro Estágio: modificações feitas no aplicativo original (`biuaidinaive.c`) e o Segundo Estágio: principais Tipos Abstratos de Dados (TADs) implementados - o que simbolizam, como são utilizados, etc. - na versão utilizando QuadTree.

Primeiro Estágio: `biuaidinaive2.c`

Essa versão consiste unicamente na adaptação do programa original para receber comandos de forma iterativa. Além disso, foi adicionada a lógica de ativação/desativação de cada estação, mantendo toda a lógica existente.

As principais modificações incluem:

- Implementação de um loop de entrada que processa comandos em tempo real, permitindo uma interação mais dinâmica com o sistema.
- Adição de funções para ativar e desativar estações, possibilitando um controle mais flexível sobre quais estações estão ativas em qualquer momento dado.

Estas mudanças visam melhorar a interatividade e o controle do aplicativo, sem alterar a lógica central de processamento e armazenamento de dados.

Segundo Estágio:

No Segundo Estágio, focamos na implementação de Tipos Abstratos de Dados (TADs) para aprimorar a eficiência e a funcionalidade do sistema. A principal estrutura introduzida é a `QuadTree`, que é utilizada para melhorar a gestão e busca de dados espaciais.

2.1 PriorityQueue

A classe `PriorityQueue` implementa uma fila de prioridade utilizando um heap binário. Seu diferencial é a capacidade de ser inicializada tanto no modo de min-heap quanto no modo max-heap. Além disso, ela possui as funções `toggleMode()` - que alterna o modo de operação reordenando o heap conforme necessário - e `forcePush()` - que forçadamente adiciona um novo item na fila, substituindo o elemento de menor prioridade.

2.2 Pair

A classe `Pair` é um template genérico que representa um par de valores, permitindo armazenar dois valores de tipos diferentes ou iguais. Ela fornece operadores de comparação, que consideram sempre o primeiro item para operações de desigualdade, facilitando a ordenação e comparação entre pares.

2.3 Point

A classe `Point` representa pontos geográficos por meio de um par de coordenadas. Além das coordenadas, cada ponto pode conter atributos opcionais como `_active` e `_id`, que podem ser utilizados em diversos contextos. A classe também oferece a função `distance()`, que recebe outro ponto como parâmetro e retorna a distância euclidiana entre os dois.

2.4 AddressInfo

A struct `AddressInfo` armazena as informações nominais dos endereços das estações (nome do logradouro, bairro, CEP, etc.) e, indiretamente, as informações geográficas, referenciando um objeto `Point` que contém as coordenadas.

2.5 Rectangle

A classe `Rectangle` representa um retângulo em um espaço qualquer, definido por dois pontos (2.3): `LB`, que indica o vértice inferior esquerdo, e `RT`, que indica o vértice superior direito. A classe possui o método `contains()`, que recebe um ponto como parâmetro e retorna verdadeiro se o ponto estiver dentro do retângulo, ou falso caso contrário.

2.6 HashTable

A classe `HashTable` implementa uma tabela hash que utiliza listas encadeadas para resolver colisões. Sua estrutura básica consiste em um ponteiro para ponteiros de `HashNode`, uma struct auxiliar.

2.6.1 HashNode

O `HashNode` representa um nó na tabela hash, armazenando um par chave-valor e um ponteiro para o próximo nó em caso de colisão.

2.7 QuadTree

A classe `QuadTree` implementa uma estrutura de árvore quaternária eficiente para armazenamento, manipulação e consulta de pontos em um espaço bidimensional. Essa estrutura divide o espaço em regiões menores (sub-regiões) à medida que novos pontos são inseridos, permitindo uma organização que facilita a execução de operações como busca de K vizinhos mais próximos, inserção, e remoção de pontos.

O `QuadTree` é inicializado com um número máximo de nós e limites espaciais definidos pela classe `Rectangle`. Ele permite a inserção de pontos e a realização de buscas baseadas em distância, oferecendo métodos para consultas eficientes (os métodos de pesquisa serão comentados na seção de Análise de Complexidade 3).

2.7.1 QuadNode

A classe `QuadNode` representa um nó dentro da estrutura de uma `QuadTree`. Cada nó cobre uma área retangular definida pela classe `Rectangle`, dividindo o espaço em quatro quadrantes: Nordeste, Noroeste, Sudeste e Sudoeste. O nó pode armazenar um ponto e também gerenciar referências a nós filhos, que representam sub-regiões dentro da área do nó.

Além disso, o `QuadNode` possui uma chave única que o identifica, bem como endereços para os nós filhos. Um ponto armazenado no nó pode ser consultado, inserido ou removido, e a estrutura pode ser subdividida dinamicamente à medida que novos

pontos são inseridos. A função `reset()` redefine o conteúdo do nó para valores padrão, preparando-o para reutilização.

2.7.2 QuadNodeManager

A classe `QuadNodeManager` gerencia a alocação e manutenção dos nós da `QuadTree`. Ela permite a criação, manipulação e destruição de nós, bem como a expansão dinâmica da capacidade de armazenamento de nós conforme necessário.

O `QuadNodeManager` mantém um vetor dinâmico de nós, ajustando sua capacidade conforme o número de nós cresce. Ele fornece métodos para criar novos nós, recuperar nós existentes por endereço, e excluir nós quando necessário. O gerenciador também pode localizar qual quadrante de um nó contém um ponto específico, tornando mais econômico a expansão da arvore, já que para criação de um novo no, o no pai pode ganhar apenas um filho, não havendo necessidade de criar imediatamente os 4 filhos.

3 Análise de Complexidade

Nessa seção abordaremos a análise de complexidade para as principais fincões implementadas nos TADs 2

3.1 TAD de Fila de Prioridades

- `push()`: $O(\log n)$
- `pop()`: $O(\log n)$
- `forcePush()`: $O(\log n)$
- `top()`: $O(1)$
- `empty()`: $O(1)$
- `toggleMode()`: $O(n \log n)$

3.2 TAD de Tabela Hash

Onde m é o comprimento da chave utilizada, temos:

- `insert()`: Melhor Caso: $O(1)$ Pior Caso: $O(m + n)$
- `search()`: MelhorCaso: $O(1)$ Pior Caso: $O(m + n)$
- `hashFunction()`: $O(m)$

Taxa de Colisão

A fórmula aproximada para a probabilidade de colisão com base na fórmula utilizada é dada por:

$$P(\text{colisão}) \approx 1 - e^{-\frac{n^2}{2m}}$$

3.3 TAD QuadTree

Inserção

A complexidade da função `insert()` depende do número de subdivisões necessárias até encontrar uma região onde o ponto possa ser inserido. No pior caso, se a QuadTree precisar ser subdividida várias vezes, a complexidade será $O(\log n)$, onde n é o número total de nós na árvore. Isso ocorre porque a árvore é dividida recursivamente em quadrantes menores, de forma semelhante a uma pesquisa binária em duas dimensões.

Busca

A função `search()` tem complexidade $O(\log n)$ no pior caso, onde n é o número de nós na árvore. A busca segue uma lógica de divisão espacial em quadrantes, semelhante à operação de inserção. Dependendo do ponto a ser encontrado, a árvore pode precisar ser percorrida de forma recursiva até a profundidade da folha correspondente.

Busca K-Nearest Neighbors (KNN)

Na função `KNNSearch()`, a busca dos K pontos mais próximos requer a verificação de todos os nós ativos da árvore. No pior caso, quando todos os nós precisam ser explorados, a complexidade da busca é $O(n \log K)$, onde n é o número de nós e K é o número de vizinhos mais próximos desejados. A inserção de um novo ponto na fila de prioridades (`PriorityQueue`) leva tempo $O(\log K)$, e essa operação é repetida para cada nó.

Busca KNN com Heurística

A função `HeuristicKNNSearch()` tenta otimizar a busca KNN utilizando uma função heurística ($O(1)$) que considera a distância euclidiana entre o ponto de busca e os limites dos quadrantes. A complexidade desta função, no entanto, depende tanto do número de subdivisões necessárias quanto do desempenho da heurística. No pior caso, ela pode alcançar $O(n \log K)$, como a busca KNN normal. No entanto, na prática, a heurística reduz o número de nós visitados, tornando a função mais eficiente para algumas distribuições de pontos e padrões de busca como veremos na seção experimental 5.

4 Estratégias de Robustez

Durante a implementação dos TADs foram tomadas diversas medidas de programação defensiva para garantir o funcionamento, facilitar o desenvolvimento (no processo depurar por exemplo), e garantir tolerância a falhas e inclusive a erros do usuário. Alguma das medidas que podem ser citadas são:

- Utilização de `try-catch` para capturar e tratar exceções inesperadas.
- Mensagens de erro claras para entradas inválidas.

5 Análise Experimental

Preparação dos Dados

Com a implementação dos algoritmos em C++ concluída, foram criados scripts auxiliares em Python para gerar testes e coletar dados de execução.

Primeiramente, foram criados testes mistos com densidades e números variados de tipos de comandos, bem como arquivos .base com diferentes distribuições (número e densidades).

Em seguida, outro script executou esses testes e armazenou os resultados em arquivos CSV correspondentes à bateria de testes. Esses arquivos contêm informações como número do caso de teste, número de estações mapeadas, versão do programa utilizado e tempo total de execução.

Finalmente, foi realizada uma análise exploratória dos dados coletados usando bibliotecas específicas. O objetivo foi criar visualizações que fornecessem insights sobre o desempenho dos algoritmos e comparassem os resultados obtidos com as previsões teóricas.

5.1 Versão

Implementamos três versões do programa: `biuaidinaive2`, `QuadBiuaidi` e `HeuristicQuadBiuaidi` (as duas ultimas se diferenciam apenas no método de busca KNN 3.3).

Como mostrado no Gráfico 1, em testes mistos, a versão com QuadTree e heurística (`HeuristicQuadBiuaidi`) destacou-se facilmente como a melhor. A versão `biuaidinaive2` ficou em segundo lugar, seguida pela versão `QuadBiuaidi`, com uma pequena diferença entre elas. Isso se deve ao fato de que, mesmo utilizando `QuadTree` para a pesquisa KNN, para garantir a precisão exata é necessário iterar por todas as estações.

O custo adicional da manipulação de uma fila de prioridade, somado à diferença de linguagem (C++ para `QuadBiuaidi` e C para `biuaidinaive2`), resultou em uma melhor performance para `biuaidinaive2`. No entanto, tudo muda ao adicionar uma heurística à QuadTree, ao adicionar uma heurística, obtem-se um desempenho superior ao priorizar áreas mais promissoras durante a pesquisa KNN, reduzindo o número de comparações e otimizando o tempo de execução, especialmente em cenários com alta densidade de estações e consultas concentradas. Isso resulta em maior eficiência abrindo mão de uma exatidão, superando as outras versões.

Nos próximos testes, será analisado o impacto de cada versão em diferentes configurações e cenários geográficos complexos.

5.2 Estações de Recarga

A primeira bateria de testes demonstrou que, para uma faixa mais restrita de valores, o número de estações inicialmente não afeta significativamente o tempo de execução. No entanto, ao aumentar o número de estações, observou-se um aumento expressivo, embora não linear, no tempo de execução. A Figura 2 ilustra o tempo médio de execução entre as versões, considerando consultas aleatórias mistas em um cenário com o número fixo de estações.

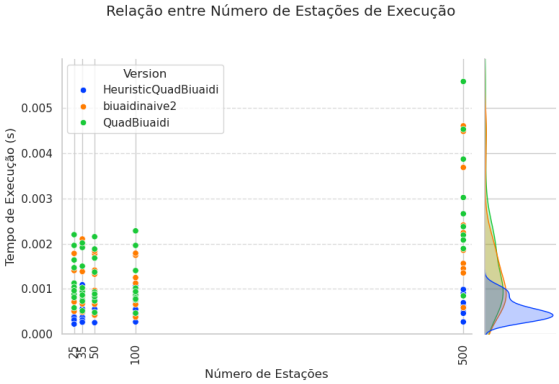

Figure 1: Tempo de execução entre versões para testes mistos.

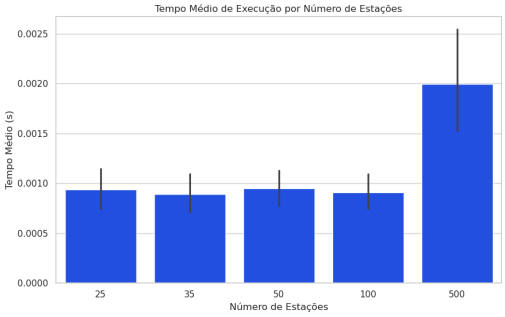

Figure 2: Tempo médio de execução para um número fixo de estações.

A segunda bateria de testes visou avaliar a influência da proximidade das estações, ou seja, a densidade das estações, no tempo de execução. A Figura 3 mostra dois cenários com 50 estações cada: um com as estações concentradas em uma região específica de Belo Horizonte e outro com as estações distribuídas aleatoriamente. O Gráfico 4 revela que os algoritmos apresentaram melhor desempenho no cenário com estações distribuídas aleatoriamente. Esse resultado pode ser atribuído a uma melhor cobertura espacial e menor impacto das consultas em regiões altamente concentradas.

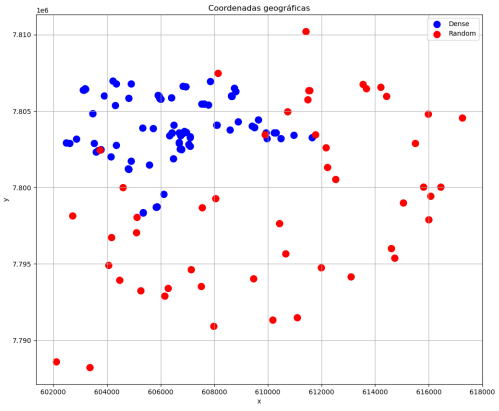

Figure 3: Distribuição das estações em dois cenários: concentração em uma região e distribuição aleatória.

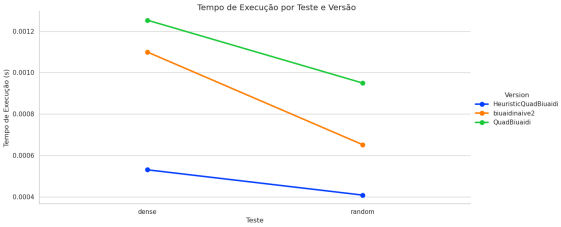

Figure 4: Comparação do desempenho dos algoritmos em cenários com diferentes distribuições de estações.

5.3 Consultas e Ativações/Desativações

Os comandos foram avaliados em diversas baterias de testes. Os resultados relacionados à natureza dos comandos (consultas, ativações e desativações) estão diretamente relacionados, por isso serão apresentados em conjunto.

Como ilustrado nos Gráficos 5 e 6, nos testes com maior concentração de consultas, o tempo de execução é consideravelmente maior. Isso se deve ao fato de que as consultas demandam muito mais computação, resultando em um aumento significativo no tempo de execução.

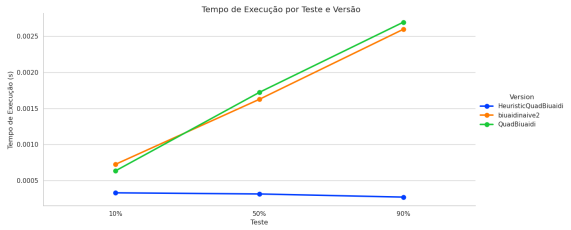

Figure 5: Tempo de execução em função da concentração de consultas.

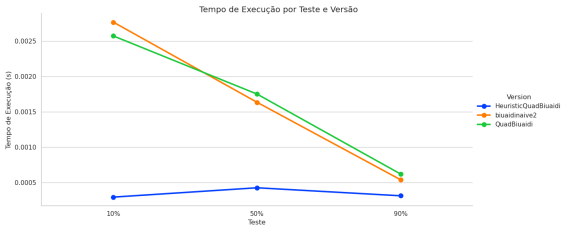

Figure 6: Tempo de execução em função da concentração de ativações e desativações.

Figure 7: Tempo de execução em função da concentração de consultas (esquerda) e ativações/desativações (direita).

Além disso, realizamos uma bateria de testes para avaliar a diferença no tempo de execução com base na natureza geográfica das consultas, ou seja, como os programas se comportam ao realizar consultas sucessivas em locais próximos. Para isso, realizamos testes em um cenário controlado, variando apenas a localização das consultas.

Consultas

Realizamos uma bateria de testes para avaliar o impacto da natureza geográfica das consultas no tempo de execução, focando em como os programas respondem a consultas sucessivas em locais próximos, simulando uma trajetória real. Os resultados, apresentados no Gráfico 8, mostram que a versão `biuaidinaive2` apresentou um desempenho significativamente pior em consultas aleatórias. Em contraste, a versão com o `QuadTree` simples demonstrou eficiência consistente, enquanto a versão com o `QuadTree` e busca por heurística teve o melhor desempenho. Estes resultados destacam como a combinação de estruturas espaciais eficientes e técnicas heurísticas pode melhorar substancialmente a eficiência em sistemas de consulta de dados complexos, sublinhando a importância de otimizações apropriadas.

Ativações/Desativações

Por último, realizamos uma bateria de testes para avaliar o impacto das ativações e desativações em estações de recarga próximas. Como mostrado no Gráfico 9, todas as versões se beneficiaram de operar em estações próximas, evidenciando a relevância da localidade de referência. No entanto, a versão `biuaidinaive2` apesar de uma uma complexidade pior para as operações de ativação/desativação - com uma complexidade O(n), em comparação com a nova versão que possui complexidade O(n + m) 2.6 - apresentou um desempenho melhor, possivelmente devido a uma vantagem na implementação em C, que tende a ser ligeiramente mais eficiente do que a implementação em C++ da nova versão.

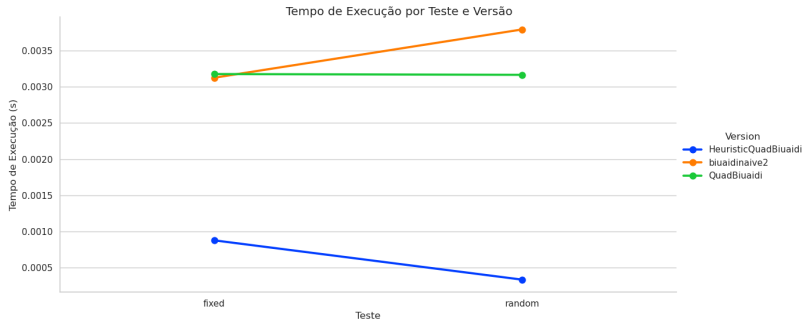

Figure 8: Comparação do tempo de execução para consultas em locais próximos versus consultas aleatórias.

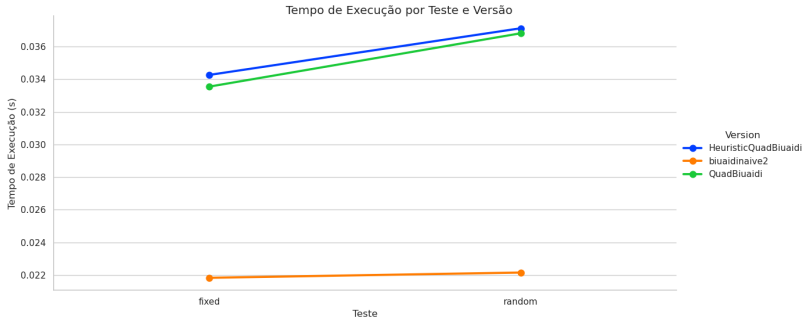

Figure 9: Comparação do tempo de execução para ativações desativações em locais próximos versus locais aleatórios.

Localidade de Referência

A eficiência dos algoritmos espaciais é diretamente influenciada pela localidade de referência. Estruturas como a `QuadTree` se beneficiam ao reutilizar dados acessados recentemente e limitar atualizações aos nós relevantes, especialmente em cenários de alta densidade de consultas em áreas geográficas específicas. O gráfico 10 ilustra esse comportamento, mostrando o desempenho do programa com operações de consulta concentradas em regiões próximas no início, seguidas de operações mistas, e, ao final, operações de ativação e desativação de estações. O gráfico foi gerado utilizando o `MemLog` para registrar e monitorar as operações de acesso e modificação.

Nesse tipo de gráfico, a inclinação está diretamente relacionada ao tempo de execução, de forma que áreas mais íngremes indicam eventos mais rápidos, enquanto áreas com menor inclinação refletem eventos mais demorados. Nota-se um maior aclive no início, durante as primeiras consultas, devido à concentração das operações em localidades próximas. Em seguida, ocorre uma estabilização, seguida de uma menor inclinação durante as operações mistas, finalizando com uma inclinação maior nas ativações e desativações em localidades próximas.

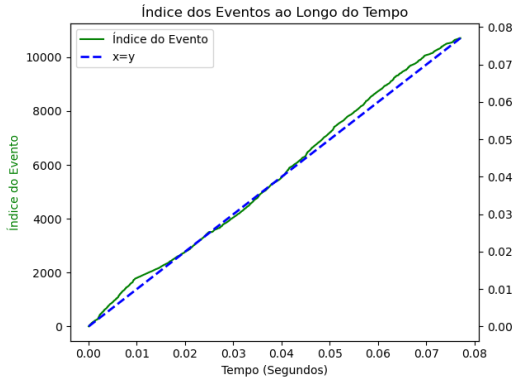

Figure 10: Eventos em função do tempo

6 Conclusões

Em conclusão, a análise comparativa das três versões do programa evidenciou o impacto de diferentes abordagens de implementação e otimização em termos de eficiência no processamento de consultas geográficas. A versão `biuaidinaive2`, baseada em C, apresentou uma boa performance devido à simplicidade e à alta eficiência da linguagem, mas ainda possui limitações em cenários de consultas complexas, especialmente ao lidar com um grande volume de dados.

Por outro lado, as versões com `QuadTree` mostraram-se mais adequadas para manipulação espacial, permitindo uma organização hierárquica dos dados e proporcionando maior precisão em consultas KNN. No entanto, o custo adicional da estrutura, aliado ao uso de uma fila de prioridade para gerenciamento dos dados, acabou penalizando o desempenho da versão `QuadBiuaidi` em comparação à simplicidade da versão `biuaidinaive2`.

A introdução de heurísticas na versão `HeuristicQuadBiuaidi` foi o diferencial que otimizou drasticamente o desempenho, especialmente em cenários onde as consultas estão concentradas em regiões geográficas específicas. Essa abordagem permitiu priorizar áreas promissoras durante as buscas, reduzindo o número de comparações necessárias e melhorando o tempo de execução em cenários densos e com alto volume de consultas. Assim, a `HeuristicQuadBiuaidi` se mostrou a solução mais eficaz, conseguindo um equilíbrio ideal entre velocidade e precisão em cenários de alta complexidade.

Em resumo, a combinação de técnicas de otimização, como estruturas espaciais e heurísticas, se destaca como fundamental para a eficiência de sistemas de consulta geográfica, principalmente em contextos que exigem escalabilidade e robustez.

7 Referências Bibliográficas

CORMEN, Thomas H. et al. Introduction to Algorithms. 3rd Edition. ed. [S. l.: s. n.], 2009.

ZIVIANI, Nivio. Projeto de Algoritmos com Implementações em Pascal e C, 3ª edição, Cengage Learning, 2011.