Trabalho Prático 3 - Estações de recarga da BiUaiDi

Daniel Augusto Castro de Paula

1. Introdução

O trabalho proposto consiste em implementar um programa para modernizar o aplicativo de identificação de estações de recarga para os carros elétricos da fabricante BiUaiDi. Ele deve ser capaz de armazenar as informações de todos possíveis pontos de recarga na cidade, considerando que essas estações podem estar ativadas ou desativadas. Além da possibilidade de ativar e desativar os pontos de recarga, o aplicativo tem outra funcionalidade que consiste em retornar os “n” pontos mais próximos de determinadas coordenadas x e y.

A versão original armazena as informações das estações e da distância de cada estação para determinado ponto utilizando vetores sem nenhum mecanismo para melhorar a eficiência para consulta e uso dos dados presente neles. Sendo assim, a modernização do aplicativo será baseada no uso de uma estrutura de dados chamada quadTree que funciona efetivamente para modelos nos quais lidamos com coordenadas geográficas e queremos subdividir o espaço para que nossa busca seja mais efetiva. No caso da quadTree seu funcionamento se assemelha a uma árvore onde cada nó pode possuir 4 filhos,na implementação em questão, esses filhos são preenchidos de acordo com a posição relativa de cada ponto.

Em relação ao mecanismo para detectar as estações mais próximas de um ponto criou-se uma função de busca que se baseia na ideia do algoritmo KNN (K-Vizinhos mais próximos). Para auxiliar na implementação também foi desenvolvido uma estrutura de max heap para otimizar essa busca.

2. Método

O programa foi desenvolvido na linguagem c compilada pelo compilador GCC da GNU Compiler Collection

2.1 Classes, TADS e métodos

HashTable: no arquivo “hash” implementou-se uma tabela hash para auxiliar nas funções de ativação e de desativação dos pontos de recarga, já que para utilizar essas funções passamos apenas o id das estações e para caminhar de maneira efetiva na quadTree precisamos das coordenadas. Essa tabela hash armazena as informações do id e das coordenadas de tal maneira que a pesquisa da coordenada seja feita rapidamente já que a complexidade média é constante O(1). Nessa estrutura temos a função hash que recebe o id e transforma em uma chave da tabela, temos também a função inserir que insere um novo elemento na tabela, a função getCoords que recebe dois parâmetros de coordenadas por referência e modifica elas se for identificado o id e por fim a função resize caso seja necessário aumentar a tabela.

MaxHeap: no arquivo “heap” temos a implementação do maxHeap utilizado na otimização da busca do algoritmo KNN. A estrutura do heap consiste em um vetor de uma estrutura chamada distanciaPonto que armazena o valor da distância em relação a um ponto e suas coordenadas x e y, além de dois inteiros para controlar a capacidade e o tamanho do heap. O maxHeap garante que o elemento com maior distância sempre vai estar no topo e isso otimiza a busca pois sempre iremos remover ele se a gente for

substituir algum valor. Nessa estrutura temos funções que criam e destroem ela, a função de push que insere um elemento, as funções heapifyUp e heapifyDown que reorganizam o heap após uma inserção, a função size que retorna seu tamanho, a função top que retorna o maior elemento e a função pop que remove o maior elemento.

Ponto: no arquivo “ponto” temos uma estrutura simples para representação das coordenadas x e y representadas por valores do tipo double, a partir desses valores iremos obter as distâncias e realizar as comparações. Nesse arquivo temos também a função distância que utiliza a fórmula da distância euclidiana para calcular a distância entre dois pontos.

QuadTree: no arquivo “quadTree” implementou-se a estrutura de dados para armazenar as informações das estações de recarga de forma que elas sejam distribuídas de acordo com o que se espera dessa estrutura, que é dividir recursivamente o espaço em 4 regiões e inserir os próximos elementos em alguma dessas sub regiões. A quadTree implementada foi feita utilizando um vetor de nós se baseando nos modelos teóricos da point e region quadTree, onde cada nó é uma estrutura que além de armazenar as informações das estações como id,cep, nome bairro, dentre outras, também armazenam informações que indicam se o ponto está ativo e os índices dos elementos que possuem relação com ele na divisão das sub regiões. Armazenamos qual é o índice do nó pai, e os índices dos nós filhos na direção noroeste (NW), nordeste (NE), sudoeste (SW) e sudeste (SE).

A fim de otimizar a busca pelos pontos mais próximos, armazenamos na estrutura os valores delimitadores das regiões que são obtidos pelo máximo e mínimo das coordenadas x e y. Para criação e delimitação do primeiro ponto inserido, as coordenadas utilizadas foram baseadas no máximo e mínimo das coordenadas apresentadas no arquivo disponibilizado “enderecobh.csv” que contém todos endereços de Belo Horizonte. Os limites dos demais pontos são definidos de acordo com as coordenadas do seu nó “pai”.

Para esse TAD temos as funções de construtor e destrutor dela, a função insert que insere um novo nó baseado nas coordenadas x e y preenchendo os índices dos elementos corretamente de acordo com as posições relativas dos pontos. A função activate e deactivate que ativam e desativam determinado ponto. Temos também a função buscaKNNRecursivo que através dos pontos da quadTree e utilizando o maxHeap retorna os elementos mais próximos, sua implementação não será usada para o vpl já que precisamos melhorar sua complexidade que nesse caso é linear, ela serve apenas para auxiliar na comparação de resultados.

Além dela, temos a sua otimização que é a função buscaKNNRecursivoOtimizado, além do heap ela utiliza uma comparação de distâncias mínimas das regiões filhas com base nos limites armazenados, para otimizar a busca já que algumas regiões serão ignoradas, retornando as K estações mais próximas de uma coordenada específica. Por fim, temos a função de resize que aumenta o tamanho do vetor caso o vetor de nós atinja a capacidade máxima.

2.2 Funções para leitura de arquivos

Para flexibilização do aplicativo a fim de permitir leitura dos dados de entrada das estações de recarga e os comandos para consultar, ativar ou desativar os pontos, o arquivo main foi adaptado e subdividido em funções para cumprir esses requisitos. A função read_csv é responsável por ler um arquivo CSV que contém informações detalhadas sobre

possíveis endereços das estações, extraindo dados relevantes como coordenadas e identificadores. Esses dados são então inseridos em duas estruturas de dados: uma QuadTree, que armazena as coordenadas espaciais e permite a busca eficiente em espaço multidimensional, e uma HashTable, que possibilita a recuperação rápida das coordenadas associadas a um identificador específico.

A função processarGeracarga, por sua vez, é utilizada para processar comandos que determinam operações sobre os dados já carregados. Ela lê um arquivo que contém instruções para ativar ou desativar pontos na QuadTree, ou para realizar consultas de proximidade, como encontrar os k pontos mais próximos de uma coordenada especificada. Essa função utiliza tanto a QuadTree quanto a HashTable para gerenciar de maneira eficiente as operações sobre os dados, garantindo a correta execução dos comandos.

Por fim, a função main garante a efetividade da execução do programa, começando com a leitura do tamanho das estruturas de dados a partir do arquivo das estações. Com base nesse tamanho, ela cria uma QuadTree e uma HashTable, utilizando parâmetros de coordenadas específicas. Em seguida, a função read_csv é chamada para carregar os dados das estações de recarga a partir de um arquivo CSV. Após o carregamento dos dados, a função processarGeracarga é invocada para processar os comandos especificados em outro arquivo, completando a operação do sistema. Ao final, as estruturas de dados são corretamente destruídas para liberar recursos.

3. Análise de complexidade

Para análise de complexidade iremos analisar cada uma das estruturas individualmente para melhor compreensão das funções individuais que compõem e influenciam a complexidade do programa.

Tabela hash: Para criar a hash table temos uma complexidade temporal linear O(n) já que a função de criação requer passar pelas posições da tabela para inicializar elas. A complexidade espacial é O(n) pois aloca a memória para os n elementos da tabela. A respeito da função Hash que cria a regra do hash podemos dizer que possui complexidade temporal O(k) onde k é o comprimento da string do id que usados para calcular o valor. Essa função não requer uso adicional de memória e por isso tem complexidade espacial O(1). Para as funções de inserção,remoção e getCoords na tabela podemos dizer que o caso médio possui complexidade temporal constante O(1) dado a eficiência do hashing, podendo no pior caso apresentar complexidade linear O(n) pois será necessário percorrer a tabela por causa das colisões. A complexidade espacial é O(1) pois não usamos memória adicional.

Max Heap: Na estrutura do heap que é utilizado para auxiliar o algoritmo de busca dos pontos mais próximos (KNN) temos a função de criação do heap possui complexidade temporal constante O(1) pois ela apenas inicializa algumas variáveis globais da classe e realiza a alocação da memória com a função calloc. A complexidade espacial dela é O(n) onde n é a capacidade do heap. As funções de push,heapifyUp e heapifyDown possuem complexidade temporal no pior caso O(log n) já que podemos precisar percorrer toda a estrutura para restaurar a propriedade do heap após inserir ou remover um elemento, a complexidade espacial delas é O(1) pois não precisamos de memória adicional.

As funções size que retorna o tamanho e top que retorna o elemento do topo possuem tanto complexidade temporal quanto espacial O(1) pois o número de operações são constantes e não requer memória extra. A função de pop que remove o elemento do topo tem complexidade temporal O(log n) já que após remover ele executa o heapifyDown que possui essa complexidade, sua complexidade espacial é O(1) já que não tem alocação adicional.

QuadTree: A quadTree desenvolvida está utilizando um vetor de nós onde os apontadores entre as regiões filhas são na verdade os índices desses pontos no vetor. A função criaQuadTree possui complexidade temporal O(1) já que suas operações são constantes apenas inicializando variáveis globais da classe e alocando o vetor de nós com calloc que evita o loop para passar por todos os pontos. A complexidade espacial é linear O(n) onde o espaço alocado é proporcional a capacidade da quadTree.

Para compreender a complexidade da função de inserção de uma estação devemos relembrar que estamos modelando o vetor de nós como se fosse uma árvore onde cada ponto pode ter até quatro filhos. Dessa forma dizemos que a sua complexidade temporal média é O(log n) na base 4 já que podemos precisar percorrer toda a altura da árvore, em casos com desbalanceamento essa complexidade se aproxima de um custo linear. Importante destacar que precisamos percorrer o vetor dessa forma para identificar e ajustar as informações que armazenam os índices dos filhos de cada ponto. Sua complexidade espacial é O(1) pois não alocamos espaço adicional.

Para as funções de ativação e desativação das estações a análise é similar com a função de inserção, a principal diferença é que nelas já preenchemos o todo o vetor com os pontos e temos apenas que caminhar sobre eles até chegar na coordenada que queremos ativar ou desativar. Pelo fato de na quadTree a gente armazenar os índices dos filhos, através da comparação dos valores das coordenadas conseguimos caminhar efetivamente pelo vetor também apresentando um complexidade média O(log n). Não utilizamos memória adicional por isso a complexidade espacial é O(1).

Em relação a função de busca dos pontos mais próximos de determinada coordenada temos uma complexidade espacial O(k) onde k seria o espaço adicional ocupado pelo tamanho do maxHeap utilizado. Para compreender a complexidade temporal da função do buscaKNNRecursivoOtimizado que é a função utilizada na execução do vpl, temos que compreender os principais passos executados nela. Primeiramente iniciamos a função com a verificação do ponto que estamos iterando, se esse ponto estiver ativo calcularemos a distância desse ponto até o ponto que estamos utilizando como referência na consulta, esse cálculo é feito com custo constante O(1). Após isso, fazemos algumas comparações com o maxHeap, se o tamanho do nosso heap for menor que os K pontos que estamos buscando apenas inserimos no heap, se for menor temos que comparar com o elemento do topo, se a distância atual que estamos verificando for menor fazemos a substituição desses pontos. Como já mencionado o custo dessa inserção no heap é O(log n).

Após isso, temos uma lógica para percorrer as sub regiões filhas do nó que estamos iterando, nessa parte realizamos uma otimização para evitar chamar recursivamente a função para todos os pontos do vetor desnecessariamente. Para cada filho iremos chamar a função distanciaMinimaRegiao que em tempo constante calcula a distância entre o ponto e a nova sub-região que estamos checando, fazendo essa comparação através das variáveis de coordenadas max e min de cada ponto. Se essa distância for menor que o topo do heap consideramos que é uma região promissora a ser verificada. O uso dessas estruturas

mencionadas em conjunto com a lógica de otimização contribuem para que seja uma função de busca efetiva com complexidade temporal sub-linear próxima de logaritmo.

4. Estratégias de robustez

No contexto do desenvolvimento do programa diversas escolhas foram feitas para tornar a implementação mais simples, eficiente e segura, evitando possíveis erros. Dentre elas, destaca-se a modularização das classes utilizadas no programa por meio da separação do header em arquivos .h e da implementação no .c, além dos comentários nas funções dessas classes.

Outra estratégia importante adotada foi a implementação de verificações nas funções para evitar o acesso indevido a posições de memória nas estruturas utilizadas, interrompendo o fluxo de execução caso dê algum erro. Além disso, para tratar corretamente a liberação de memória implementou-se e utilizou-se os destrutores das estruturas implementadas evitando vazamentos de memória. Para ajudar nessa verificação o valgrind foi utilizado.

Por fim, vale destacar também a implementação de funções para ajudar no dinamismo do programa, por exemplo as funções de resize que serviriam para aumentar o tamanho de determinada estrutura caso a capacidade máxima seja atingida. Elas são importantes pois evitam que seja alocado um tamanho de memória extremamente grande para garantir que iremos armazenar todos os dados. Com isso alocamos um tamanho relativamente pequeno e vamos aumentando de acordo com a necessidade.

5. Análise experimental

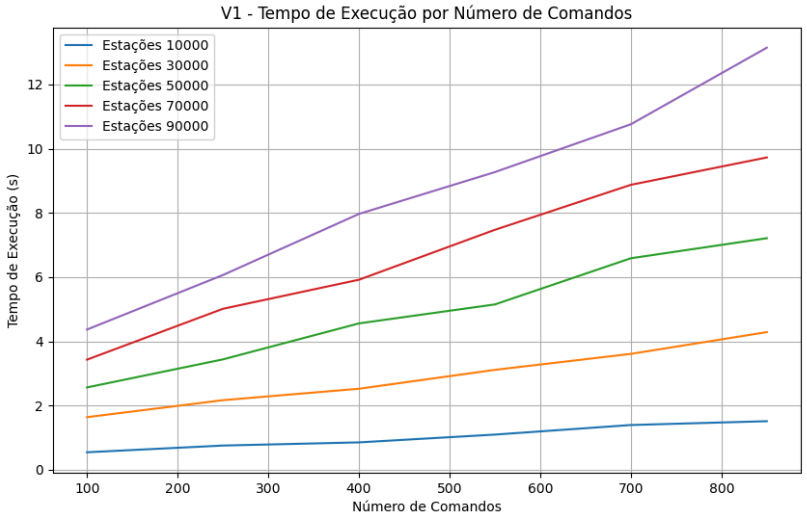
Para a avaliação experimental iremos dividir esse tópico por seções para que seja possível comparar a versão inicial do aplicativo com a versão otimizada do aplicativo utilizando a quadTree. É válido destacar que ambas as versões já estão flexibilizadas no sentido de ler as entradas de arquivos e ser capaz de ativar, desativar e consultar estações. As principais diferenças são em relação à forma que armazenamos as informações das estações e como obtemos os pontos mais próximos. Na versão 1 todas as estações são mantidas em um vetor sem uma ordenação bem definida, apenas inserimos na ordem que lemos o arquivo de entrada, já a consulta é obtida realizando um quickSort no vetor de distâncias que é obtido passando por todos os pontos. Já na versão dois essas operações foram otimizadas pelo uso conjunto da quadTree, heap e hash.

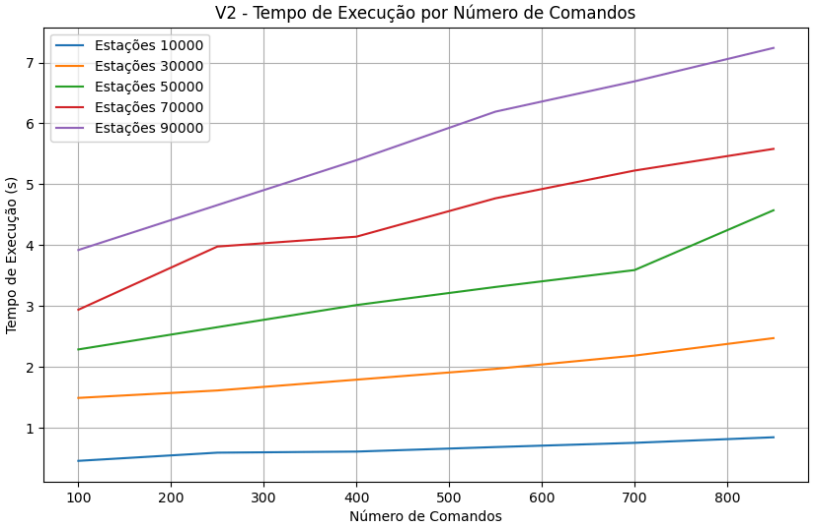
Comparação tempo de execução dos aplicativos

Considerando as hipóteses apresentadas na análise teórica, nessa primeira avaliação podemos concluir que a versão 2 (com a quadTree e hash) pode demorar um pouco mais para executar o passo de leitura das estações de recarga já que nela inserimos na quadTree respeitando a hierarquia dos filhos o que é mais demorado do que a versão 1 que apenas insere na última posição do vetor. Além disso, também precisamos inicializar a tabela hash para ajudar na busca das coordenadas já que nas funções de ativar e desativar temos apenas o id.

Por outro lado, o conjunto das estruturas de quadTree, hash e maxHeap melhorou significativamente a complexidade temporal das funções de ativar, desativar e consultar que de acordo com a análise teórica estão apresentando custo sub-linear próximo de logaritmo.

Sendo assim, a versão 2 é extremamente efetiva para grande número de comandos executados. Uma vez armazenado todos os pontos temos muitos ganhos que permitiram uma eficiência considerável nas funções de ativação,desativação e consulta. Podemos concluir que esse é um bom trade off pois uma aplicação real com esse mesmo objetivo não fica lendo as estações de recarga diversas vezes, ele lê uma vez e depois executa os comandos solicitados.





Esse experimento apesar de utilizar parâmetros aleatórios para escolher as estações e os comandos que serão executados nos permite concluir que a versão 2 é bem mais rápida, principalmente para casos com muitos comandos. Além disso, percebemos a tendência logarítmica da v2 em comparação da linearidade da v1.

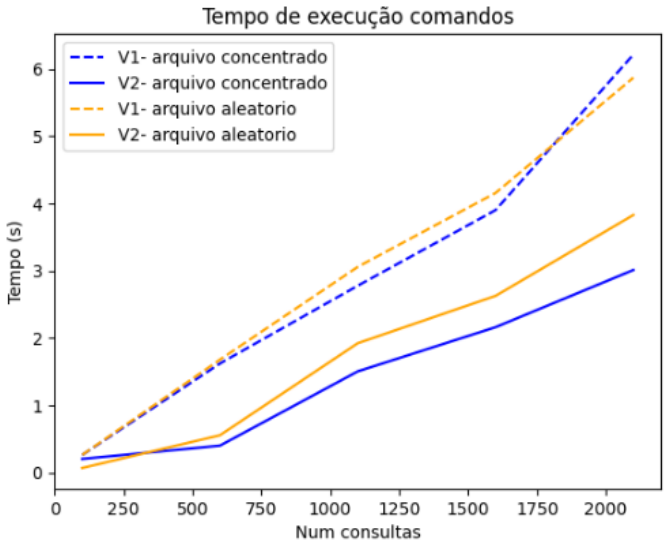
Em relação a quantidade das estações de recarga que são armazenadas logo no início da execução do aplicativo percebemos que quanto mais estações armazenadas mais os programas vão demorar mais para retornar os resultados dos comandos.

Análise distribuição das estações

Além da quantidade das estações, podemos analisar o impacto da distribuição desses pontos. Nesse sentido, a versão 2 deve possuir uma performance bem melhor para realizar a operação de consulta para casos onde armazenamos regiões que apresentam uma grande disparidade de concentração, já que devido ao uso do heap e da heurística de

otimização do algoritmo de busca essa versão será mais efetiva para identificar rapidamente a região mais promissora e concentrar as buscas nela.

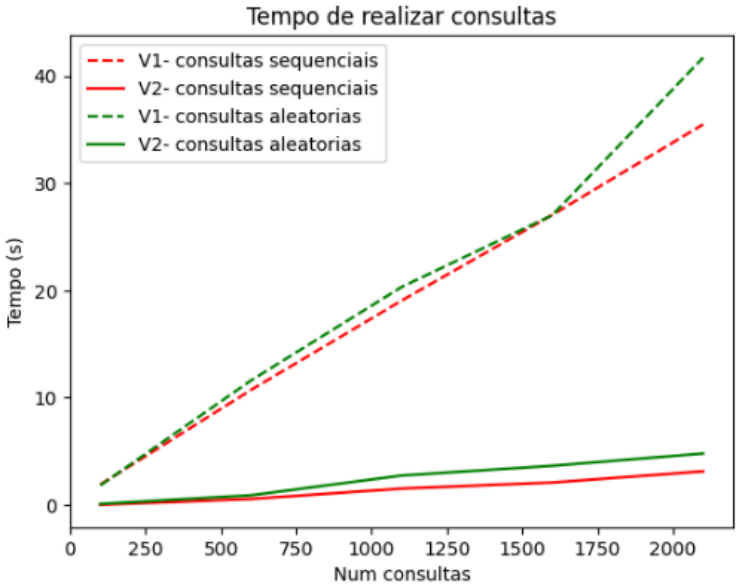
Para essa hipótese testaremos os programas selecionando estações de apenas dois bairros da cidade, um no extremo da zona norte e o outro no extremo da zona sul. Esse teste será comparado com uma distribuição aleatória para confirmar se a diferença foi mais perceptível e realmente ele foi mais efetivo. Ou seja, esse parâmetro influencia diretamente a efetividade da consulta.



A partir dos resultados concluímos que as hipóteses estão corretas, já que a versão 1 sempre demora mais para realizar as consultas que a versão 2 e além disso com o arquivo de teste no qual temos a distribuição de estações concentrada em apenas dois bairros distantes a consulta na versão 2 se mostrou ainda mais efetiva.

Análise natureza das consultas

Além da influência da quantidade de consultas, a natureza das consultas deve ser considerada para analisar a eficiência e comparar as versões dos aplicativos. Para isso iremos utilizar uma base fixa de cinquenta mil estações e variar os tipos das consultas, no primeiro teste as consultas serão com pontos sequências simulando a trajetória de um veículo, na segunda os pontos serão aleatórios.

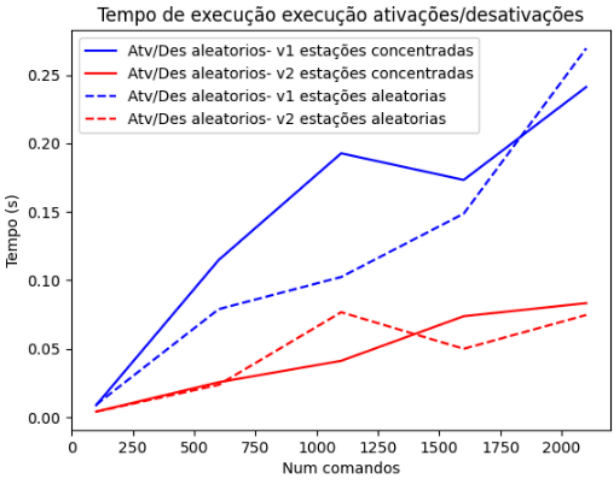


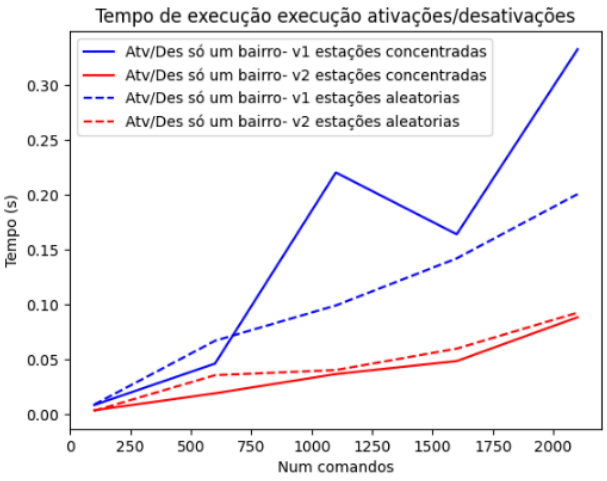
A partir desses resultados concluímos que a QuadTree utilizada na versão 2 se beneficia das consultas sequenciais. Como essa estrutura organiza os dados de forma hierárquica, quando as consultas seguem uma sequência, a QuadTree pode reutilizar as informações de consultas anteriores e acelerar ainda mais o processo, pois áreas do espaço já exploradas não precisam ser reavaliadas. Além disso, como as posições acessadas do vetor de nós são similares pois estamos percorrendo um caminho sequencial temos também uma influência positiva de aspectos de localidade que serão abordados com mais detalhes no último tópico das avaliações. Sendo assim, tanto pela sua natureza otimizada para buscas espaciais quanto pela eficiência adicional em consultas sequenciais, a QuadTree mostra-se superior ao vetor na execução dessas tarefas.

Análise natureza das ativações e desativações

Podemos analisar também o impacto da natureza das ativações e desativações para aumentar ou diminuir a eficiência da execução dos aplicativos. Nesse sentido iremos fazer uma avaliação a respeito de duas variações de carga de trabalho, a primeira é a respeito da densidade espacial desses comandos e a segunda é a sua frequência temporal.

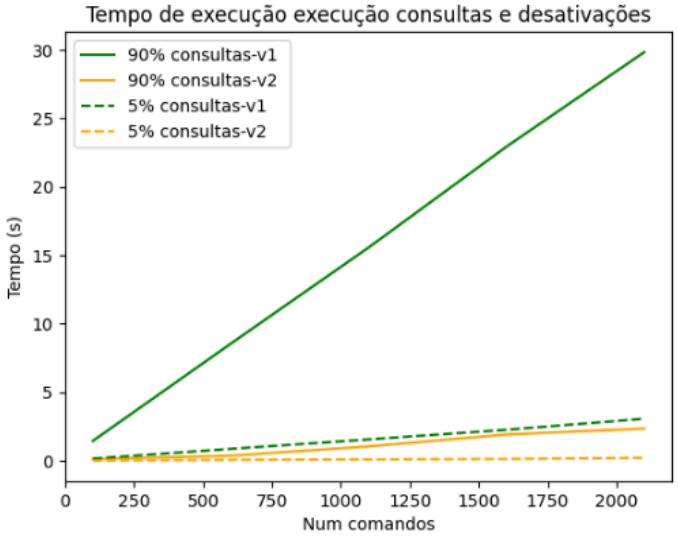
A densidade espacial, ou seja, a forma como as estações estão distribuídas também influencia diretamente na eficiência das ativações e desativações. É esperado que para um conjunto de comandos de ativações e desativações para estações que estão concentrados geograficamente em poucas regiões, a versão com a quadTree seja mais eficiente pois irá direcionar o loop que procura a estação correta mais rapidamente. Além disso, se tivermos casos nos quais desativamos todas as estações de um bairro, iremos também acessar pontos similares do vetor que foram acessados a pouco tempo, o que geralmente é mais rápido por questões de localidade que serão abordadas.





Os resultados acima foram obtidos com dois tipos de distribuições de estações, uma é referente a um arquivo de entrada com estações concentradas,ou seja, apenas de dois bairros, um na zona norte e outro na sul. O outro tem o mesmo número de estações mais de bairros aleatórios com dispersão aleatória. Na imagem da esquerda as ativações/desativações foram feitas de forma aleatórias, na imagem da direita selecionamos apenas um bairro.Comprovamos assim nossa hipótese, já que a versão 2 sempre está sendo melhor que a versão 1, e para casos em que as estações apresentam uma distribuição mais concentrada entre os bairros a versão 2 se mostrou ainda mais eficiente.

Em relação a frequência temporal, ou seja, a intercalação entre ativações e desativações em comparação ao número de consultas, podemos prever que quanto maior a porcentagem de desativações mais rápido serão feitas as funções de consultas já que iremos verificar menos pontos. Para o teste iremos utilizar um arquivo de estações fixo com cinquenta mil estações distribuídas aleatoriamente, os comandos executados serão apenas de desativação e consultas, no primeiro caso será testando uma porcentagem de 90% de consultas e no segundo 5%.



O resultado do experimento comprova totalmente a teoria de que quanto mais desativações mais rápido vão ser os programas, e a versão 2 sempre ganha no tempo de execução dos comandos em relação a versão 1.

Localidade de referência

Em relação às análises de localidade de referência precisamos destacar os conceitos da localidade temporal e espacial. A localidade temporal diz que se determinado item é acessado, é provável que ele seja acessado novamente em um período relativamente curto de tempo. A espacial diz que se um item é acessado, é provável que itens de memória próximos sejam acessados em breve. Essas características são importantes de serem levadas em consideração na implementação de soluções para otimizar determinados tipos de problemas, pois podem reduzir consideravelmente o tempo de acesso aos dados.

Na primeira versão do programa, as estações são inseridas na ordem em que são lidas do arquivo de entrada, sem qualquer consideração por sua posição espacial. Após essa inserção, um vetor de distâncias é criado e, em seguida, ordenado usando o algoritmo QuickSort. Essa abordagem resulta em uma localidade de referência relativamente pobre. Como os dados não são organizados em uma estrutura que favoreça a proximidade espacial ou temporal, os acessos subsequentes durante as consultas e atualizações podem levar a um número maior de falhas na memória.

Por outro lado, na segunda versão do programa, a utilização de uma quadTree em conjunto com o KNN otimiza o programa em termos de localidade de referência. A quadTree organiza as estações de recarga de acordo com suas coordenadas espaciais, dividindo o espaço em regiões menores e mais gerenciáveis. Isso melhora a localidade espacial, pois consultas que buscam os k pontos mais próximos tendem a acessar regiões contíguas da memória, mesmo que a estrutura em si não seja armazenada em blocos de memória

contíguos. Além disso, a hierarquia da quadTree pode promover uma melhor localidade temporal, ainda mais se considerarmos casos de testes onde múltiplas consultas são realizadas em uma mesma região do espaço.

Ao medir a distância de pilha, que avalia a localidade de referência com base na diferença entre os endereços de memória acessados consecutivamente, espera-se observar uma maior distância média na primeira versão do programa. Na segunda versão, a organização dos dados em uma hierarquia espacial pode resultar em distâncias de pilha menores e mais consistentes, refletindo uma melhor utilização da localidade de referência.

Para testar essas hipóteses utilizamos o módulo de cachegrind do valgrind para avaliar parâmetros no cache que englobam a análise de localidade, o teste foi feito para um conjunto com cinquenta mil estações rodando 2100 comandos aleatórios. Conseguimos perceber que a versão 2 com a quadTree apresenta uma localidade bem melhor já que quanto menor o valor, mais efetivo.

Tipo de modelagem	I1	LLi	D1	LLd	LL
quadTree	492837	2487	29411779	982422	984609
Vetor com QuickSort	509226	151556	662586121	32386601	32533157

6. Conclusões

O trabalho em questão englobou o desenvolvimento de soluções para problemas de consultas de dados geograficamente distribuídos. A escolha de uma quadTree para modelar a distribuição colaborou para um estudo e compreensão maior sobre essas estruturas. Além disso, o escopo do trabalho também exigiu uma boa compreensão acerca de heaps e hashTable.

O uso conjunto dessas estruturas escolhidas foram responsáveis por garantir uma boa modelagem do problema otimizando o tempo de execução nas principais funções do programa que era ativação, desativação e consulta. A versão 1 apesar de possuir uma lógica de implementação mais simples tem seu uso limitado, servindo apenas para casos de uso com cargas de trabalho menores devido a sua complexidade temporal ruim.

Com as realizações dos testes ficou evidente que em determinados aspectos uma versão ainda pode ser melhor que a outra, no caso em questão, a versão 2 pode demorar mais para instanciar a quadTree e a hashTable do que a versão 1 que apenas insere as estações no final do vetor. Por outro lado, ela se mostrou bem mais efetiva para realização dos comandos do aplicativo. Esse trade off pode ser considerado ótimo já que lemos as estações de carga apenas uma vez e os comandos são executados em grandes quantidades a todo momento.

Sendo assim, o uso em conjunto de diversas estruturas aprendidas durante a disciplina de estruturas de dados foi essencial para a melhoria e otimização do aplicativo. Nessa nova versão, o produto se tornou mais escalável e foi viabilizado para uma aplicação real.

7. Bibliografia

Wikipedia - Quadtree

GeeksForGeeks - K-Nearest Neighbor(KNN) Algorithm