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, após modernizar seu aplicativo de identificação de estações de recarga, deseja adaptá-lo para equipar os multimídias dos veículos.

1.2 Problema

Os veículos BiUaiDi possuem memória limitada. Para viabilizar sua efetiva implementação, é necessário criar uma arquitetura de dois níveis que permita a replicação transparente da parte dos dados necessária para calcular as distâncias para as estações de recarga mais próximas.

1.3 Solução Proposta

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

1. Implementar uma arquitetura de dois níveis e adaptar o programa existente para o pleno funcionamento.
2. Realizar testes para avaliar:
    - Medidas de desempenho, como a latência para determinar as estações de recarga mais próximas.
    - O funcionamento da arquitetura de dois níveis em termos do número e volume de dados transferidos.

2 Método

A principal estrutura introduzida nessa versão é o SMV, outra estrutura alterada em relação a ultima versão é a `QuadTree`. A seguir uma breve descrição das principais estruturas utilizadas:

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 SMV

A classe `SMV` é responsável pelo gerenciamento de páginas de memória em um ambiente simulado, utilizando técnicas de substituição de páginas (LRU) e manipulação de sinais para tratar falhas de segmentação (`SIGSEGV`).

O método `initPage(...)` aloca memória para as páginas físicas e lógicas, cria um arquivo de swap, e aplica a proteção à memória usando `mprotect`. As páginas são alinhadas ao tamanho de página (`PAGESIZE`) e inicializadas com dados nulos no arquivo de swap.

A finalização é realizada pelo método `endPage()`, que grava no arquivo de swap as páginas que foram modificadas, além de liberar a memória e fechar o arquivo de swap.

A classe também instala um manipulador de sinais através do método `installSignalHandler()`, que captura o sinal `SIGSEGV` e o redireciona para o método `segvHandler(...)`, o qual trata a falta de páginas na memória. Este método gerencia a substituição de páginas conforme a técnica LRU, ajusta o status das páginas (como `DISCO`, `VALID` e `DIRTY`), e utiliza `mprotect` para proteger e modificar o acesso às páginas de memória conforme necessário.

2.8 QuadTree

A classe `QuadTree` implementa uma estrutura de árvore quaternária eficiente para armazenamento, manipulação e consulta de pontos em um espaço bidimensional. Esta 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 eficientes para consultas (os métodos de pesquisa serão detalhados na seção de Análise de Complexidade 3).

2.8.1 QuadNode

A classe `QuadNode` representa um nó dentro da estrutura da `QuadTree`. Cada nó cobre uma área retangular definida pela classe `Rectangle` e é capaz de subdividir 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. O ponto armazenado no nó pode ser consultado, inserido ou removido. 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.

2.8.2 QuadNodeManager

A classe `QuadNodeManager` gerencia a alocação e manutenção dos nós da `QuadTree`. Ela oferece uma forma eficiente de manipulação de nós, permitindo a criação, recuperação, modificação e destruição de nós, além da expansão dinâmica da capacidade de armazenamento conforme necessário.

O `QuadNodeManager` utiliza uma abordagem de alocação de memória baseada em um gerenciador de memória específico (`SMV`), o que facilita a gestão dos nós de maneira eficiente, mesmo com grandes volumes de dados. O gerenciamento dinâmico dos nós permite a expansão da árvore de maneira econômica, criando novos nós filhos apenas quando necessário.

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 SMV

Inicialização de Página

A função `initPage(int &bytesAllocated)` aloca a memória necessária para as páginas físicas e lógicas e cria um arquivo de swap. A complexidade da alocação de memória para as páginas e a inicialização do arquivo de swap é $O(n)$, onde n é o número total de páginas (`NUMPAGE`). A razão para isso é que cada página precisa ser inicializada com dados nulos no arquivo de swap e ser alinhada ao tamanho da página física.

Tratamento de Falhas de Segmentação

A função `segvHandler(int sig, siginfo_t *sip, ucontext_t *uap)` trata falhas de segmentação (`SIGSEGV`) quando uma página referenciada não está em memória. No pior caso, o processo de substituição de páginas, utilizando o algoritmo LRU, precisa percorrer todas as páginas para encontrar uma página substituível. Isso leva a uma complexidade de $O(n)$, onde n é o número total de páginas na memória.

Adicionalmente, as operações de proteção e desproteção de memória (`mprotect`) são feitas por página e, no pior caso, têm complexidade $O(1)$ para cada página, acumulando uma complexidade de $O(n)$ quando aplicadas a todas as páginas.

Substituição de Páginas

No algoritmo de substituição de páginas, o método LRU (Least Recently Used) é empregado para determinar qual página deve ser descartada e escrita no arquivo de swap, caso seja necessário liberar espaço na memória. A complexidade da operação de encontrar a página a ser substituída é $O(n)$ no pior caso, pois pode ser necessário examinar todas as páginas até localizar a menos recentemente usada. Além disso, a operação de escrita ou leitura de uma página no arquivo de swap possui complexidade $O(1)$ por operação. Contudo, no pior caso, se várias páginas precisam ser substituídas, a complexidade total se torna $O(n)$, considerando a soma das operações de I/O para todas as páginas envolvidas.

Instalação de Manipuladores de Sinais

A função `installSignalHandler()` registra os manipuladores de sinais para capturar `SIGSEGV`. Esta operação é feita uma única vez e tem complexidade $O(1)$, pois envolve apenas a configuração das máscaras de sinal e a instalação dos manipuladores usando `sigaction()`.

Finalização de Páginas

A função `endPage()` grava todas as páginas válidas de volta no arquivo de swap e libera a memória alocada. No pior caso, se todas as páginas tiverem sido modificadas e ainda estiverem na memória, a função terá que percorrer todas as páginas e gravá-las no swap, resultando em uma complexidade de $O(n)$.

Resumo das Complexidades

- Substituição de Páginas: $O(n)$
- `initPage(int &bytesAllocated)`: $O(n)$
- `segvHandler(int sig, siginfo_t *sip, ucontext_t *uap)`: $O(n)$
- `installSignalHandler()`: $O(1)$
- `endPage()`: $O(n)$

3.4 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) a serem avaliados em diferentes razoes entre memória primária e secundária. Além disso, foram criados testes com consultar em pontos similares, alterando somente o numero de estacões.

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

A expectativa é que a nova versão do sistema, que faz uso de memória virtual, tenha um tempo de execução mais longo em comparação com a versão inicial, que utilizava apenas a memória física. O gráfico 1 ilustra essa diferença de desempenho de maneira clara. A comparação é feita entre a versão inicial, que opera exclusivamente com memória física, e a nova versão, que adota uma arquitetura de dois níveis com suporte à memória virtual (SMV).

Como observado no gráfico, independentemente do valor do parâmetro MtSR (razão entre memória primária e secundária), a versão que utiliza SMV apresenta tempos de execução consideravelmente mais longos. Isso ocorre porque a necessidade de swaps entre memória primária e secundária adiciona uma sobrecarga significativa ao tempo de processamento. Portanto, mesmo em cenários onde a relação entre memória primária e secundária é otimizada, a versão com SMV tende a ser mais lenta do que a versão que opera apenas com memória física.

5.2 Razão entre memória primária e secundária

A relação entre a memória primária e a secundária é um dos principais fatores que influenciam o desempenho da arquitetura de dois níveis. Quanto maior essa proporção, mais dados podem ser armazenados na memória primária, resultando em menos acessos à memória secundária, que tende a ser significativamente mais lenta. Consequentemente, espera-se que uma maior capacidade de memória primária resulte em tempos de execução mais rápidos para o cálculo das estações de recarga mais próximas.

Essa tendência é claramente observável no Boxplot 2, onde tempos de execução mais curtos estão associados a maiores razões entre a memória primária e secundária.

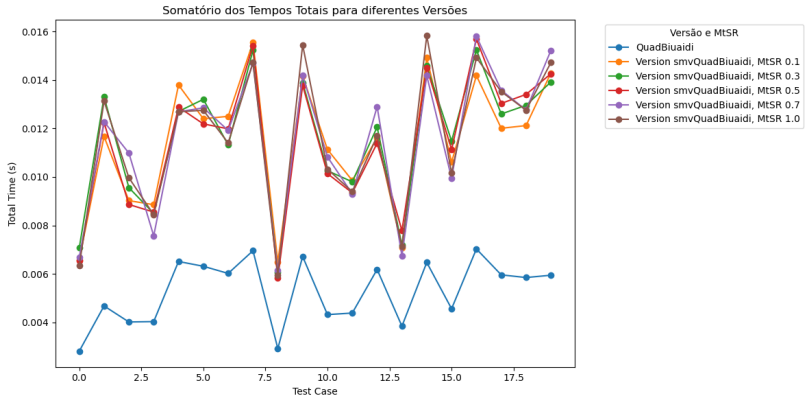

Figure 1: Comparação entre o tempo de execução das versões do sistema: memória física versus arquitetura de dois níveis com SMV.

O gráfico de linhas em 3 também confirma essa correlação, mostrando uma queda progressiva no tempo total à medida que a capacidade de memória primária aumenta.

Entretanto, é importante analisar os outliers (como em em MtST = 0.5 no gráfico 3) que indicam aumentos de tempo, mesmo com o aumento da memória primária. Esses desvios podem ser atribuídos a diversos fatores, como variações nos padrões de consulta ou alocação de dados que podem não ser ideais. Em alguns casos, pode ocorrer um aumento no overhead associado ao gerenciamento de uma memória primária maior.

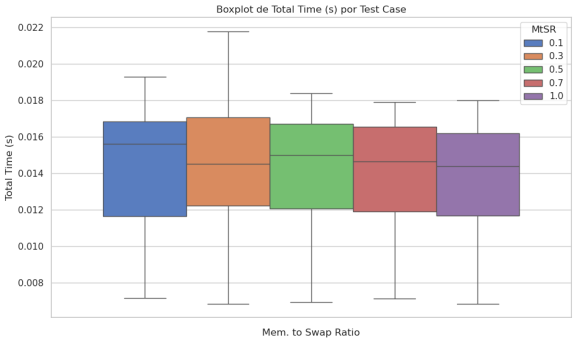

Figure 2: Boxplot representando o tempo total de execução em função da relação entre a memória primária e secundária (MtST)

5.3 Número de estações considerado nas consultas

O desempenho do sistema em consultas de vizinhos mais próximos (KNN) é diretamente influenciado pelo número de estações consideradas em cada consulta. Uma maior amplitude na busca por vizinhos (ou seja, um valor maior para o parâmetro K) implica em uma maior quantidade de dados a serem processados, resultando em um aumento no número de operações de leitura e escrita entre a memória primária e a memória secundária.

Essa maior complexidade leva a uma maior frequência de swaps entre as memórias, o que aumenta significativamente o volume de dados transferidos. Consequentemente,

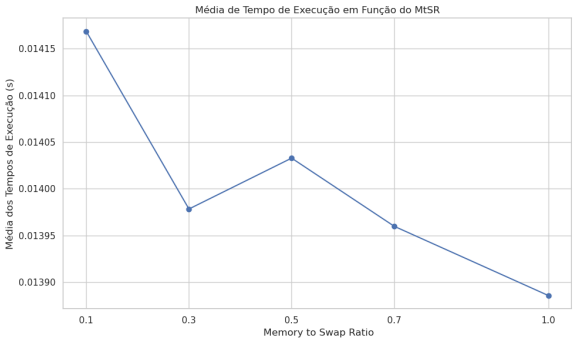

Figure 3: Gráfico de linhas mostrando a variação do tempo total de execução conforme a razão entre a memória primária e secundária (MtST).

a latência das consultas tende a aumentar à medida que o número de estações consideradas cresce. Esse comportamento pode ser atribuído ao fato de que o processamento de um número maior de estações exige mais ciclos de computação e mais tempo de acesso à memória, sobretudo quando os dados precisam ser trocados entre a memória primária (mais rápida) e a memória secundária (mais lenta).

O gráfico 5 ilustra claramente essa tendência, agrupando testes entre as classes de forma que o mesmo teste para classes diferentes (por exemplo, 10.1 e 10.2) difere somente no número de estações consideradas nas consultas KNN (parâmetro K). Nota-se que à medida que o valor médio de K aumenta, o tempo total de execução também aumenta, refletindo o impacto do maior volume de estações processadas.

A mesma relação pode ser observada de maneira ainda mais clara no gráfico 4, que apresenta uma visualização detalhada da correlação entre o número médio de estações consideradas (AVGK) e o tempo total de execução. Este gráfico permite visualizar diretamente o quanto o aumento no número de estações leva a um aumento no tempo de execução, destacando o peso que o valor de K exerce sobre o desempenho do sistema.

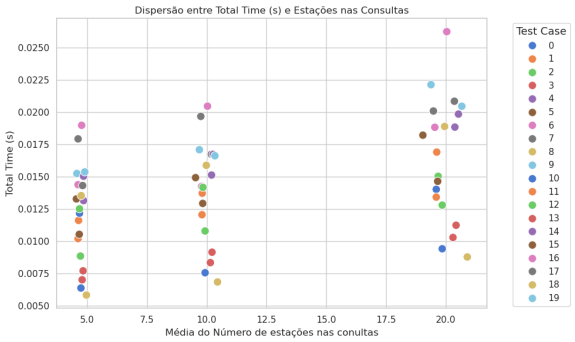

Figure 4: Gráfico de linhas mostrando a variação do tempo total de execução conforme a razão entre a memória primária e secundária (MtST).

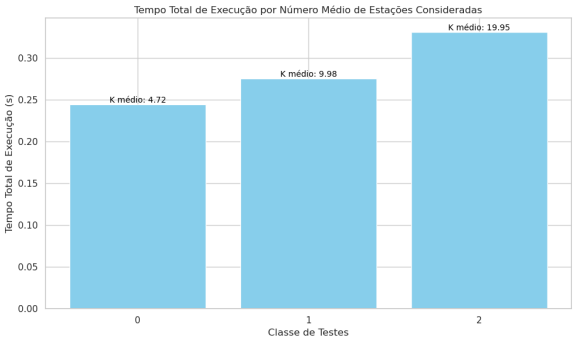

Figure 5: Gráfico de linhas mostrando a variação do tempo total de execução conforme a razão entre a memória primária e secundária (MtST).

6 Conclusões

A implementação da arquitetura de dois níveis foi um sucesso, como esperado arquitetura mostrou-se eficaz em viabilizar o cálculo das distâncias para as estações de recarga mais próximas no contexto de memoria limitada, porém, com um aumento significativo no tempo de execução, devido ao uso de memória virtual.

Os testes realizados confirmaram as previsões teóricas: o tempo de execução da nova versão, que faz uso de memória virtual, foi consideravelmente maior em comparação à versão que utilizava apenas a memória física. Esta diferença é principalmente atribuída à necessidade de swaps entre a memória primária e secundária, que adiciona uma sobrecarga ao processamento.

Além disso, a relação entre a memória primária e secundária mostrou-se um fator determinante no desempenho do sistema. Maiores proporções de memória primária resultaram em menos acessos à memória secundária e, consequentemente, tempos de execução mais rápidos. No entanto, outliers em alguns cenários sugerem que o gerenciamento de uma maior capacidade de memória primária pode introduzir overheads que impactam negativamente o desempenho.

Outro relevante é o número de estações consideradas nas consultas KNN. O aumento do parâmetro K acarretou uma maior quantidade de dados processados, levando a mais operações de leitura e escrita e, consequentemente, a uma maior latência.

Em resumo, a arquitetura de dois níveis com suporte à memória virtual se mostrou uma solução viável para os veículos com memória limitada, permitindo a funcionalidade desejada de cálculo de distâncias. No entanto, para maximizar o desempenho, é necessário balancear cuidadosamente os parâmetros de memória e minimizar o valor de K nas consultas.

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.