Projeto e Análise de Algoritmos
Relatório UrbanFast

Autores: Cristiano Larréa, Felipe Lamarca, Lucas Cuan, Paloma Borges e Yonathan Gherman
Docente: Thiago Pinheiro de Araújo

Escola de Matemática Aplicada
FGV EMAp

Rio de Janeiro
2023.2

Sumário

1 Introdução

O objetivo desse trabalho é desenvolver soluções para auxiliar a operação da UrbanFast, um novo serviço de entregas, com o objetivo de apoiar solicitações de clientes, parceiros e operadores do negócio, com base nos algoritmos abordados na disciplina. O foco está na eficiência na gestão de rotas de entrega, considerando diversos fatores, como a localização dos vendedores, clientes, centros de distribuição e entregadores.

2 Solução Arquitetural

Nossa solução arquitetural se baseia na representação da planta da cidade como um grafo, onde cada segmento de uma via é modelado como arestas e as interseções (esquina) como vértices. Além disso, os entregadores, vendedores, e cliente também são inseridos como vértices no grafo.

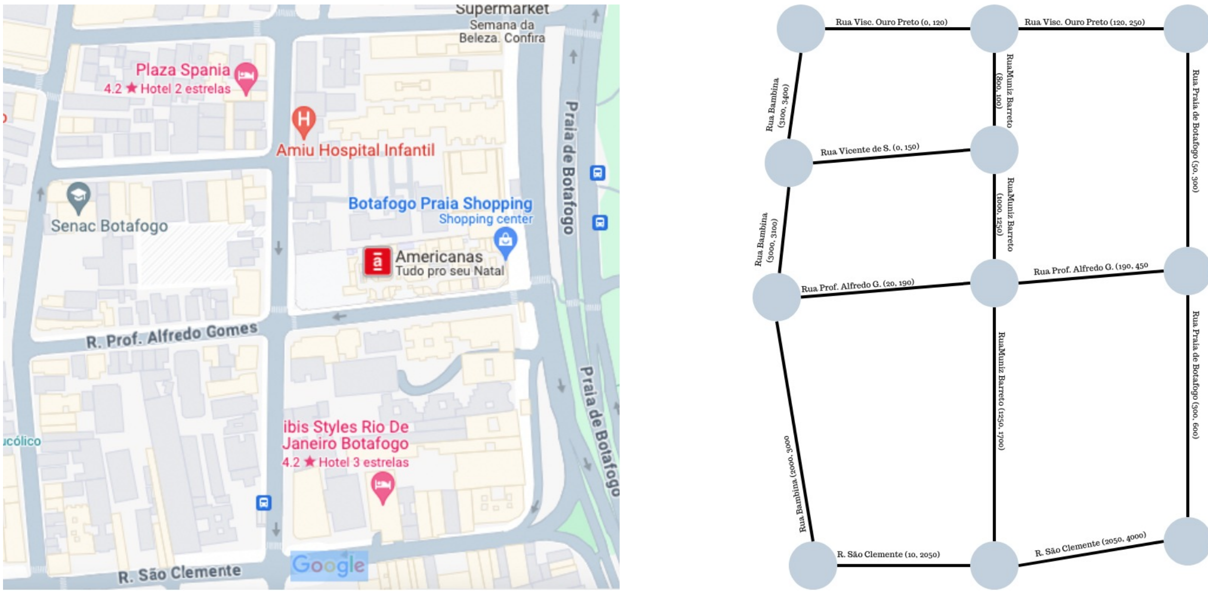

Figura 1 – Modelagem do Mapa

A figura acima representa a modelagem para as esquinas (base do mapa). Ao incluir um centro de distribuição, vendedor, ou entregador ele também é inserido no grafo. No momento no qual é inserido, a aresta referente ao endereço em que será inserido é quebrada em 2, com esse vértice sendo inserido entre elas. As propriedades que os vértices e as arestas tem são:

- endereço: em um vértice, tupla do formato (nome da rua, número na rua). Todos os vértices possuem um vetor de endereços, pois podem ter de 1 (caso esteja no meio da aresta) até n endereços (caso seja uma esquina que ligue n ruas).

- tipo: em um vértice, o seu tipo. Pode obter os valores: Corner, Deliveryman, Client, Seller.
- numeração de início e numeração de fim: em uma aresta, o número do primeiro endereço da aresta e o número do último endereço da aresta;
- distancia: em uma aresta, a diferença absoluta entre os número de início e de fim;

A escolha dessa modelagem foi definida dessa forma por alguns motivos: (1) a modelagem de todas as entidades como vértices com tipos deixa mais flexível a execução dos algoritmos de menor caminho e relacionados de forma que englobe todas as entidades possíveis. Além disso, caso seja preciso desconsiderar algum tipo de entidade, pode-se apenas acrescentar um `if` durante essa busca; (2) Por modelarmos o grafo de uma cidade, nosso grafo é esparso, o que ajuda na complexidade dos algoritmos utilizados. Além disso, a adição de vértices que não são do tipo Corner aumenta o número de arestas, em média, em 1 (pois quebra a aresta no meio existente) - mantendo assim a propriedade de $E \sim V$ e, portanto, o grafo esparso.

3 Estrutura de Dados

Para a estruturação do grafo, foram criados três classes: `Vertex`, `Edge` e `Graph`, sendo esta última um vetor de ponteiros pros Vertex.

```
class Vertex {
public:
    string name;
    int32_t id;
    string type;
    vector<tuple<Edge*, int>> streets_connected;

    Vertex() noexcept;
};
```

```
class Edge {
public:
    std::string name;
    Vertex* vertex1;
    Vertex* vertex2;
    int starting_number;
    int ending_number;
    double distance;

    Edge(const string n, Vertex* v1, Vertex* v2, int sn, int en);
};
```

Figura 2 – Estrutura criadas para representar o grafo.

De maneira geral, a ideia foi de modelar o grafo como uma lista de adjacências, já que é esparso e isso otimiza a memória que ocupa. Entretanto, ao invés do vértices possuírem ponteiro para outro vértices, agora um vértice possui ponteiro para uma aresta, que por consequinte possui ponteiro para o outro vértice ao qual é conectada e, assim por diante. Foi necessário essa adaptação já que agora as nossas arestas também possui propriedades que vão ser exploradas (número de inicío e número de fim) ao inserir novos vértices.

A figura 3 abaixo representa como um grafo de apenas dois nós conectados (a esquerda) é representado na nossa estrutura (a direita). As setas cinzas representam os ponteiros e os retângulos representam espaços de memória.

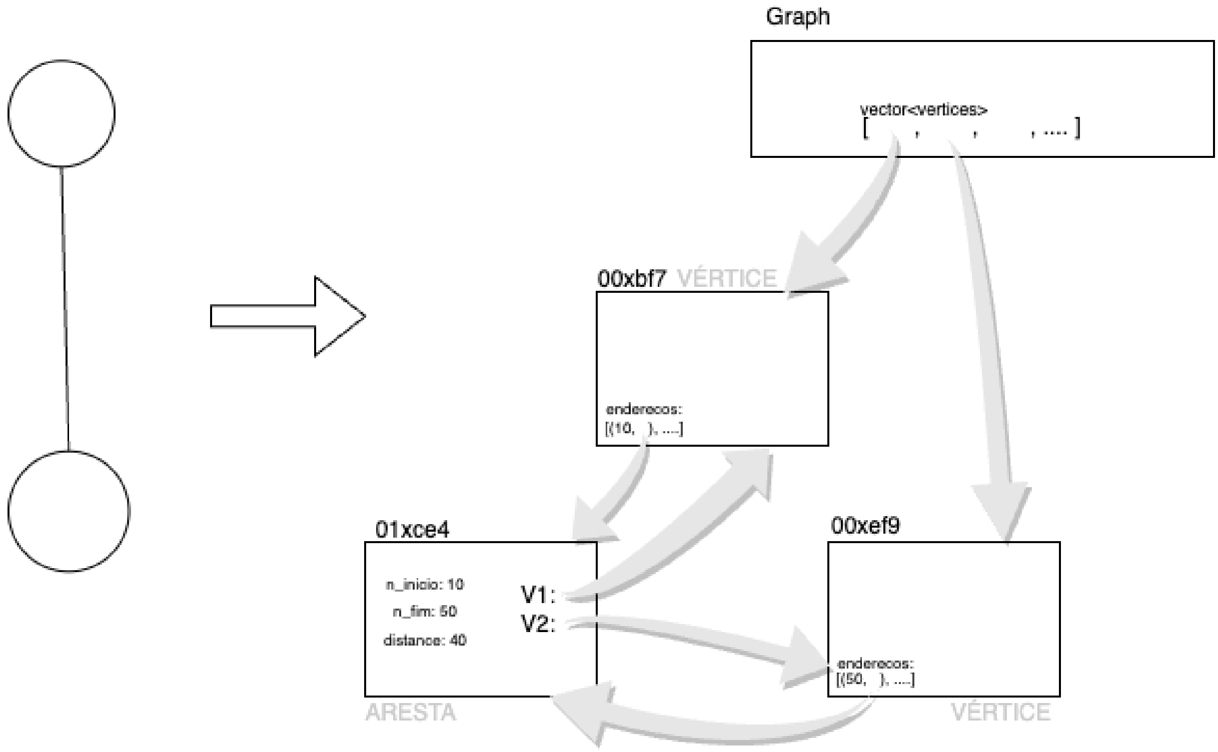

Figura 3 – Estrutura de dados que representa o grafo

3.1 Operações

A solução para cada uma das operações encontra-se abaixo.

3.1.1 Operação 1 - Encontrar entregadores próximos

Quando um pedido é postado, é necessário encontrar um entregador para fazer o transporte. Para buscar os n entregadores mais próximos do local de coleta, utilizamos o algoritmo de Dijkstra. A ideia base foi: aplicar o algoritmo de Djikstra a partir do vértice do vendedor (local de coleta), de forma a obter a menor distância desse vértice para todos outros vértices do grafo. Depois, itera-se sobre a vetor de distâncias, selecionando as n menores e que, ao mesmo tempo, o vértice referente aquele índice seja do tipo "Deliveryman". O pseudo-código encontra-se abaixo.

Em relação à complexidade: no Djkistra, iteramos sobre todos os vértices do grafo. Para cada vértice, precisamos iterar sobre todas as w arestas conectadas aquele vértice. Como o Djikstra foi implementado com heap, as operações de inserção no heap custam $O(logV)$. Assim, essa etapa possui complexidade $O(V * w + ElogV)$. Após, iteramos n vezes sobre o vetor de distâncias de tamanho V, totalizando uma complexidade $O(nV)$.

Portanto, nossa complexidade total é $O(V * w + ElogV + nV)$. Entretanto, re-

Algorithm 1 Encontrar Entregadores Mais Próximos

```
procedure Operacao1(grafo, n_entregadores, requisicao)
    contador ← 0
    entregadores ← vetor de tamanho n_entregadores
    distancias ← vetor de tamanho grafo.nVertices
    Chama dijkstra(requisicao.vendedor, grafo, distancias)
    while contador < n_entregadores do
        MIN ← ∞
        indice_min ← indefinido
        entregador ← nulo
        for i ← 0 até tamanho(distancias) − 1 do
            if distancias[i] ≤ MIN e grafo.vertices[i].tipo == "Entregador then
                MIN ← distancias[i]
                indice_min ← i
                entregador ← grafo.vertices[indice_min]
            end if
        end for
        entregadores[contador] ← entregador
        contador ← contador + 1
        distancias.apaga(distancias.comeco() + indice_min)
    end while
    devolve entregadores
end procedure
```

pare que w, por ser o número de arestas conectado a um vértice, normalmente terá valor baixo em nossa modelagem: uma esquina normalmente terá $w = 4$, os outros vértices normalmente terão $w = 2$. Por isso, podemos considerar w como constante e desconsiderar na notação Big O. Assim, a complexidade da operação 1 é $O(V + ElogV + nV) = O(ElogV + nV)$.

3.1.2 Operação 2 - Definir a rota de uma entrega simples

Para determinar a rota mais curta considerando a coleta do produto no endereço do vendedor e a entrega no endereço do cliente, estaremos aplicando o algoritmo Dijkstra duas vezes.

A ideia base foi a seguinte: utilizar o Dijkstra para calcular a menor distância do entregador a todos outros vértices do grafo. Dijkstra também retorna um vetor de pais, de maneira que podemos encontrar vértice pai do local de venda, o pai desse vértice pai, e assim por diante, até que o vértice encontrado seja o entregador — obtendo, assim, a sequência de vértices que devem ser percorridos do entregador até o vendedor.

Após isso, é utilizado o Dijkstra novamente para calcular, dessa vez, a rota do vendedor ao cliente. De maneira semelhante, busca-se os pais até chegar no vértice do vendedor, obtendo a sequência de vértices que devem ser percorridos nesse segundo momento.

Uma pequena adequação é que precisamos garantir que o caminho indicado seja dado por arestas (segmentos de ruas), e não por vértices. Como sabemos que cada vértice contém como atributo as ruas conectadas a ele, podemos iterar os vértices na sequência obtida e, assim, obter as ruas que os conectam. Naturalmente, como se trata de um grafo esparso e não é esperado que um vértice possua um número grande de ruas conectadas a ele, a complexidade dessa operação não afeta a ordem assintótica do algoritmo.

Em relação à complexidade do algoritmo, executamos o algoritmo de Djkistra seguido da procura pelos vértices pais. No pior caso, teremos que acessar a o vetor de pais V vezes (em que o entregador seja uma raíz e o vendedor seja uma folha, por exemplo). Assim, a complexidade nesse momento é $O(ElogV + V) = O(ElogV)$.

3.1.3 Operação 3 - Definir a rota de uma entrega considerando centros de distribuição

O objetivo da operação 3 é encontrar a melhor rota, considerando que o entregador deverá passar por algum centro de distribuição para obter o produto e levá-lo até o cliente. A solução parte da ideia de que, se conseguirmos obter a menor distância entre um entregador e todos os centros de distribuição, e a menor distância de todos os centros de distribuição até o cliente, conseguimos obter a combinação que minimiza a distância total percorrida. O passo a passo é o seguinte:

1. Aplicar Dijkstra a todos os vértices do grafo, obtendo a menor distância de cada um deles para todos os outros. Guardamos o resultado da operação um vetor de tuplas, onde a entrada 0 da tupla guarda as distâncias desse vértice para todos os outros, e a entrada 1 guarda o vetor de parents que permite recuperar menor caminho a ser percorrido. Como aplicamos Dijkstra a todos os vértices do grafo, essa operação é realizada em $O(V^2 * w + VE \log V)$;

2. Iterando esse vetor e os vértices do grafo, conseguimos obter um vetor de triplas que contenha informações do tipo $(entregador, centro\_de\_distribuicao, distancia)$. Fazemos o mesmo para os centros de distribuição, obtendo as triplas que informem a distância entre os centros de distribuição e o cliente. Os dois vetores iterados possuem V elementos, de maneira que a complexidade da operação é $O(V^2)$;

3. Finalmente, munidos dos vetores com as combinações

   $(entregador, centro\_de\_distribuicao, distancia)$ e $(centro\_de\_distribuicao, cliente, distancia)$, conseguimos iterar os dois vetores em loops aninhados encontrando a soma de distâncias que minimiza o caminho percorrido pelo entregador até chegar no cliente. No pior caso a operação é executada em $O(V^2)$.

4. Como guardamos o vetor de pais dos vértices escolhidos (centro de distribuição e entregador que minimizam o caminho percorrido), conseguimos retornar o caminho (lista de arestas) a ser percorrido pelo entregador até o cliente.

Abaixo o pseudo-código da operação:

Algorithm 2 Operacao 3

```
procedure OPERACAO3(Pedidootimizado)
    Declare vetor finals

    //Gera vetor de ([d1,..,d2], [Vertex1, Vertex2, ...])
    distances_and_parents ←vetor de tamanho V
    for vértice v no grafo do
        Chama dijkstra(v, grafo, distancias, parents)
        distances_and_parents[v] ←(distancias, parents)
    end for

     //Pega as triplas de entregadores-CD-distancias‘s
    for vértice v no grafo do
        if v.type==Deliveryman then
            dist_to_v ← distances_and_parents[i][0]
        end if
        for distancia d em dist_to_v do
            if grafo.vertices[d.index]==”Distribution Center” then
                Insere (v, grafo.vertices[d.index], d) em finals
            end if
        end for
    end for

     //Pega as duplas de CD‘s - clientes
    for vértice v no grafo do
        if v.type==Centro de Distribuicao then
            dist_to_v ← distances_and_parents[i][0]
        end if
        for distancia d em dist_to_v do
            if grafo.vertices[d.index]==”Cliente” then
                Insere (v, grafo.vertices[d.index], d) em finals
            end if
        end for
    end for
     //Pega o par que minimiza a soma das distancias
    for cada tripla na lista de triplas de entregador-CD-distancia do
        for cada tripla na lista de triplas CD-cliente-distancia do
            if os centros de distribuicao forem iguais then
                calcula a distancia somada
                if a soma for menor que a distancia minima then
                    atualiza a distancia minima e a melhor rota
                end if
            end if
        end for
    end for
     Encontra o caminho do entregador escolhido ate o centro de distribuicao escolhido
(mesma logica usada na operacao 2)
     Encontra o caminho do centro de distribuicao escolhido ate o cliente (mesma
logica usada na operacao 2)
     Compoe os caminhos na mesma lista
    devolve entregador, CD, melhor_caminho
end procedure
```

4 Resultados e discussões

Como as operações e os algoritmos implementados dependem que o grafo em questão seja conexo, infelizmente não foi possível realizar testes variando o número de instâncias. Ao gerar n vértices e conectá-los através de arestas aleatórias, não conseguimos garantir essa propriedade. Na pasta `tests` do repositório, mais especificamente no arquivo `testOperation1.cpp`, há um exemplo de como poderia ser desenvolvida a testagem da operação caso fosse possível garantir que o grafo é conexo. No arquivo, variamos as instâncias de entregadores e vendedores no grafo, bem como o número n de entregadores mais próximos buscados pela operação 1, calculando o tempo a cada execução.

Para tentar suprir a falta dos testes, na pasta `demos` do projeto instrumentalizamos as operações para serem executadas em grafos previamente avaliados e construídos, garantindo que são conexos. Embora essa não seja a maneira adequada de avaliar a complexidade assintótica dos algoritmos, torna-se possível avaliar a corretude dos algoritmos.