Instituto Superior de Engenharia do Porto

Relatório Sprint 3

Feito por:

Rodrigo Pontes, 1221083

Miguel Oliveira, 1211281

Mário Ribeiro, 1221019

Rodrigo Castro, 1220636

Professores:

Alberto Sampaio (ACS)

02/01/2024

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

Instituto Superior de Engenharia do Porto

Índice

Índice de figuras

1. Introdução

Este relatório destaca as conquistas substanciais realizadas pelo grupo de desenvolvimento durante o Sprint 3, focado na implementação de novos algoritmos para a resolução de desafios específicos no âmbito da empresa GFH. A GFH atua como um operador logístico especializado na distribuição de cabazes contendo produtos agrícolas em uma rede dinâmica, composta por produtores, clientes e hubs.

Durante este sprint, o grupo concentrou seus esforços no desenvolvimento e aprimoramento de algoritmos destinados a otimizar a gestão da oferta e da procura agregadas. A oferta agregada representa os cabazes disponibilizados pelos produtores, enquanto a procura agregada é derivada das encomendas feitas pelos clientes. A ênfase foi colocada na resolução eficiente de desafios logísticos, maximizando a eficácia da distribuição de cabazes e, por conseguinte, a satisfação tanto dos produtores quanto dos clientes.

Este relatório abordará em detalhes as estratégias algorítmicas implementadas para otimizar a coordenação entre produtores, clientes e hubs, garantindo uma distribuição eficaz dos produtos agrícolas. O grupo dedicou especial atenção à resolução de questões específicas, tais como a gestão de inventário, roteirização de entrega e sincronização entre os diferentes elementos da cadeia logística.

Ao longo das próximas seções, serão apresentadas as análises e resultados obtidos com a implementação desses novos algoritmos, destacando como contribuíram para a superação dos desafios operacionais no contexto da GFH. Este relatório visa fornecer uma compreensão abrangente das inovações algorítmicas introduzidas, delineando o impacto positivo na eficiência e na qualidade global do serviço prestado pela GFH.

2. Diagrama de Classes

Figura 1 – Diagrama de classes

3. Algoritmos Implementados

3.1. USEI06

O algoritmo da USEI06 foi efetuado por: Mário Ribeiro

Descrição

O objetivo é encontrar, para um produtor, os diversos percursos viáveis entre um local de origem e um hub, levando em consideração a autonomia do veículo elétrico, sem considerar recargas durante o percurso. O critério de aceitação inclui fornecer, para cada percurso identificado, o local de origem, os locais de passagem (identificando hubs), a distância entre todos os locais do percurso, a distância total e o tempo total estimado para o percurso.

Como funciona

Foi utilizado algoritmo de busca em profundidade (DFS) modificado para encontrar todos os percursos possíveis de um local de origem para um destino, tendo, no entanto em consideração a autonomia do veículo elétrico.

Em resumo, o algoritmo percorre todos os caminhos possíveis no grafo do vértice de origem para o vértice de destino, tendo em conta a autonomia do veículo elétrico. Ele armazena informações sobre cada percurso, incluindo distância total, distâncias entre locais, tempo total de viagem, e retorna esses dados num formato estruturado (InfoForUs06).

Os percursos, caso sejam encontrados, são guardados em diferentes LinkedLists e a cada um destes percursos estará associado através de um mapa um objeto do tipo ResultadoTotalSemi. Este objeto armazena todos os dados relevantes do percurso.

No final o mapa é retornado com o formato de InfoForUs06.

Análise de complexidade

A complexidade do código está relacionada com o algoritmo de busca em profundidade (DFS).

A complexidade é afetada pela pilha de chamadas recursivas durante a busca em profundidade. A profundidade máxima da pilha é proporcional à altura máxima da árvore de busca, que pode ser O(V).

```
for (Locals verticeAdj : g.adjVertices(vOrig)) {


    if (verticeAdj == vDest && getTotalPathData(g, path,
averageSpeed).getTotalDistance() <= autonomy * 1000) {
        path.add(vDest);
        ResultDataTotalSemi resultDataTotalSemi = getTotalPathData(g, path,
averageSpeed);
        //System.out.println(getTotalPathData(g, path, averageSpeed).getTotalDistance()
<= autonomy * 1000);
        if (getTotalPathData(g, path, averageSpeed).getTotalDistance() <= autonomy *
1000) {
            pathsData.put(new LinkedList<>(path), resultDataTotalSemi);
        }
        path.removeLast();
    } else if (!visited[g.key(verticeAdj)] && getTotalPathData(g, path,
averageSpeed).getTotalDistance() <= autonomy * 1000) {
        depthFirstSearchPaths(g, verticeAdj, vDest, visited, path, pathsData, autonomy,
averageSpeed);
    }
```

Testes

Foram realizados testes que comprovam que os cálculos dos tempos e distâncias das viagens estão corretos.

Para além destes foi também testado o método de retorno de modo a comprovar a veracidade dos resultados.

3.2. USEI07

O algoritmo da USEI07 foi efetuado por: Miguel Oliveira

Descrição

O problema consiste em encontrar, para um produtor que parte de um local de origem, o percurso de entrega que maximiza o número de hubs pelos quais passa. O algoritmo considera fatores como horário de funcionamento de cada hub, tempo de descarga dos cestos, distâncias a percorrer, velocidade média do veículo e tempos de carregamento. O critério de aceitação exige que sejam fornecidos o local de origem, os locais de passagem (identificando hubs), a hora de chegada em todos os locais do percurso (com indicação da hora de partida em hubs após a descarga dos cestos), a distância total do percurso, o número de carregamentos do veículo e o tempo total do circuito, discriminando o tempo atribuído aos carregamentos do veículo, ao percurso e ao tempo de descarga dos cestos em cada hub.

Como funciona

A função principal, us07_method, realiza passa pelos hubs ordenados com base em critérios como proximidade, horário de abertura e distância ao ponto anterior. Para cada hub selecionado, calcula o menor caminho, considerando autonomia do veículo e horário de funcionamento do hub. Os resultados incluem informações sobre os locais visitados, horários de chegada, distância total percorrida, número de carregamentos e tempo total do circuito.

São utilizadas outras funções auxiliares, como ordenarVerticesDeAcordoComMaisIndicados, ordenarPorDistancia, ordenarVerticesPorProximidade, getEdgesFromPath, calcularHorarioEstimadoChegada e calcularDistanciaProximidade, que contribuem para a implementação do método principal, tratando da ordenação de vértices, obtenção de arestas de um caminho, cálculos de horários e distâncias.

Análise de complexidade

O loop principal passa sobre os hubs, realizando operações como ordenação e cálculo de menor caminho para cada hub. Se V é o número de vértices (locais) e E é o número de arestas (conexões), o pior caso ocorre se todos os vértices forem hubs.

Complexidade: O(V)

A complexidade é maioritariamente afetada dentro do loop principal da função us07_method. Esse trecho envolve a iteração sobre os hubs, a ordenação deles com base em diferentes critérios e o cálculo do menor caminho para cada hub.

```
while (!hubs.isEmpty()) {
    hubs = ordenarVerticesDeAcordoComMaisIndicados(hubs, horaInicio, inicialVertex,
graph, velocidadeMedia);
    if (!hubs.isEmpty()) {
        nextHub = hubs.remove(0);
        // ...
        Integer distance = Algorithms.shortestPathWithAutonomy(graph,
inicialVertex, nextHub, Integer::compare, Integer::sum, 0, shortPath, distances,
Integer.MAX_VALUE, autonomy, chargingPoints, mapForUi);
```

Testes

Foram realizados testes com o objetivo de verificar se a função us07_method retorna resultados esperados para um conjunto específico de parâmetros, incluindo a contagem de carregamentos, a distância total percorrida e o tempo total estimado.

Foi também testado a exatidão da função para um conjunto diferente de parâmetros.

Foi também testado comportamento em cenários de baixa autonomia, verificando se o resultado é uma lista vazia quando a autonomia é insuficiente para cobrir a distância total.

Por fim, foi realizado um teste que examina o impacto de um horário de início muito cedo, validando se a função retorna uma lista vazia quando a hora de início inviabiliza a definição de uma rota.

3.3. USEI08

O algoritmo da USEI08 foi efetuado por: Rodrigo Castro

Descrição

O problema consiste em encontrar um circuito de entrega para um produtor, partindo de um local de origem, passando por N hubs com o maior número de colaboradores uma única vez e retornando ao local de origem, com o objetivo de minimizar a distância total percorrida. Os valores possíveis para N são 5, 6 e 7. O critério de aceitação requer a devolução do local de origem do circuito, os locais de passagem (indicando o número de colaboradores em cada hub), a distância entre todos os locais do percurso, a distância total, o número de carregamentos e o tempo total do circuito, discriminando o tempo dedicado aos carregamentos do veículo, ao percurso e ao tempo de descarga dos cestos em cada hub.

Como funciona

Utilizando o algoritmo TSP modificado, onde cada hub é visitado exatamente uma vez, minimizando a distância total percorrida e retornando ao ponto de origem. O método findOptimalCircuit é a interface principal, utilizando recursão para encontrar o próximo hub mais próximo e calcular o caminho mínimo até ele, considerando a autonomia do veículo. A verificação da presença correta dos hubs e o cálculo da distância total são fundamentais para garantir a validade do circuito. Estas implementação oferece uma abordagem eficaz para otimizar rotas de entrega, considerando as restrições específicas dos hubs e as características do veículoAnálise de complexidade.

Análise de complexidade

A complexidade temporal do algoritmo é dominada pela recursão no método tsp adaptado. Se n é o número de hubs, o algoritmo percorre cada hub uma vez, calculando a distância mínima para o próximo hub mais próximo. Dessa forma, a complexidade seria O(n^2) devido ao loop sobre os hubs e a chamada a shortestPathWithAutonomy, que tem complexidade linear. Vale a pena ressaltar que o algoritmo pode ser afetado pelo tamanho do grafo, mas em cenários práticos, o número de hubs geralmente será pequeno em comparação com o número total de vértices.

```
for (Locals hub : remainingHubs) {
    LinkedList<Locals> path = new LinkedList<>();
    Integer distance = Algorithms.shortestPathWithAutonomy(graph,
currentVertex,inicialVertice, Integer::compare, Integer::sum, 0, path,distances,
Integer.MAX_VALUE,autonomy,chargingPoints,mapForUi);

    if (distance.intValue() < minDistance) {
        minDistance = distance;
        nextHub = hub;
        minPathForHub = path;
    }
}
```

Neste excerto, a busca pelo próximo hub mais próximo em um loop sobre todos os hubs resulta na maior contribuição para a complexidade temporal.

Testes

Para os testes da USEI08, foram realizados três casos distintos para verificar a integridade dos resultados obtidos pelo algoritmo. Cada teste consistiu na execução do algoritmo de otimização para encontrar o circuito de entrega mais eficiente a partir de um local de origem, passando por uma quantidade específica de hubs (5, 6 e 7), e retornando ao local de origem. Esses testes utilizaram um arquivo de dados menor ("small") devido à complexidade temporal do algoritmo, que demonstrou apresentar uma eficiência inferior a um minuto na execução com arquivos de dados grandes ("big"). Cada teste verificou se todos os hubs específicos estavam presentes na lista de arestas do circuito, fornecendo confiança na correta identificação e inclusão dos hubs no percurso otimizado.

3.4. USEI09

O algoritmo da USEI09 foi efetuado por: Rodrigo Pontes

Descrição

O algoritmo proposto tem como objetivo organizar localidades em N clusters, assegurando a presença de apenas um hub por cluster. Essa organização é alcançada iterativamente, removendo conexões com o maior número de caminhos mais curtos entre as localidades até formar clusters isolados.

Como funciona

1. O método recebe um grafo e o número desejado de clusters.
2. Inicializa um mapa para armazenar os hubs e os seus clusters associados.
3. Verifica se o grafo não é nulo antes de prosseguir.
4. Inicia um loop enquanto o número de hubs e clusters no mapa for menor que o número desejado de clusters.
5. Irá identificar a aresta que, quando removida, resultará no maior número de caminhos mais curtos entre os vértices.
6. Inicia-se a busca de todos os clusters após a remoção da aresta.
7. Para cada cluster, identifica um hub, de acordo com o critério de maior grau, assim irá associar-se ao cluster correspondente.
8. Retorna o mapa inicializado anteriormente contendo hubs e seus respetivos clusters associados após o número de clusters ser alcançado.

Análise da complexidade

O algoritmo fornecido é determinístico. A determinação da natureza determinística de um algoritmo está associada à previsibilidade de seu comportamento. Neste caso, o algoritmo não envolve elementos de aleatoriedade ou não-determinismo. Cada ação e decisão no algoritmo são baseadas em lógica determinística, sem depender de fatores imprevisíveis.

1. Método “organizeClusters”

Para identificar o hub no cluster foi efetuado a seguinte iteração

```
  for(Set<V> component : connectedComponents){
     V hub = findHubInCluster(g, component);
     hubsAndClusters.put(hub, component);
}
```

Neste método irá percorrer os N clusters criados apartir da remoção da aresta para encontrar o hub num dos clusters.

Este método apresentará a complexidade linear (O (n)).

2. Método “findEdgeWithMaxShortestPaths”, “calculateShortestPaths”

Para calcular o número de caminhos mais curtos por cada aresta, foi efetuado o seguinte ciclo:

```
for (Edge<V, E> edge : g.edges()) {
    int numPaths = calculateShortestPaths(g, edge, ce, sum, zero, infinity);
```

A complexidade irá depender do método “calculateShortestPaths”, mas irá percorrer todas as arestas do grafo.

```
boolean success = Algorithms.shortestPaths(g, sourceVertex, ce, sum, zero, paths,
dists, infinity);
```

O shortestPaths tem complexidade O ($(V + E) * \log(V)$) e, como irá percorrer todas as arestas, a complexidade no pior e melhor caso do primeiro método terá complexidade O ($((V + E^2) * \log(V))$).

3. Método “getConnectedComponents”

Esse método irá percorrer todos os vértices do grafo, como diz no seguinte método

```
for (V vertex : g.vertices()) {
    if (!visitedVertices.contains(vertex)) {
        Set<V> connectedComponent = depthFirstSearch(g, vertex);
```

Depois de verificar se as listas de vértices contêm o vértice escolhido irá fazer a pesquisa em profundidade para procurar todos os vértices conectados a esse diretamente ou indiretamente. Como o algoritmo de pesquisa em profundidade apresenta complexidade O ($V + E$), o método apresentado terá complexidade O ($V^2 + E$).

Em suma, o algoritmo completo irá apresentar a complexidade máxima possível das expressadas, no melhor e pior caso, ou seja, O ($((V + E^2) * \log(V))$).

Testes

Para a USEI09, foram efetuados 3 testes para a fiabilidade dos resultados no algoritmo. Esses testes apresentam a lista de Hubs, e o conjunto de vértices que pertencem ao cluster desse hub. Foi usado o ficheiro mais pequeno (“small”) na execução dos testes, devido ao que o algoritmo apresentar uma complexidade temporal superior a 1 minuto na execução com os ficheiros grandes (“big”).

3.5. USEI11

O algoritmo da USEI11 foi efetuado por: Miguel Oliveira

Descrição´

O problema consiste na necessidade de um Product Owner carregar um arquivo contendo os horários de funcionamento de uma lista de hubs. Se os hubs já existirem, os horários devem ser redefinidos, caso contrário, uma mensagem de erro deve ser emitida. O objetivo é garantir a gestão adequada dos horários de funcionamento dos hubs, evitando conflitos e assegurando que as informações sejam carregadas corretamente.

Como funciona

O código da classe USEI11 é projetado para carregar os horários de funcionamento de hubs a partir de um arquivo. As informações sobre os hubs são fornecidas no formato de linhas no arquivo, onde cada linha contém o código do hub, a hora de abertura e a hora de fechamento, separados por vírgulas. Durante a leitura do arquivo, o código verifica se cada hub já existe no grafo (representado pela estrutura 'graph'). Se o hub existir, seus horários são atualizados com os valores fornecidos no arquivo. Caso contrário, uma mensagem é exibida indicando que o hub não existe.

Análise de complexidade

A análise geral de complexidade para este código é O(N * V), onde N é o número de linhas no arquivo e V é o número de vértices no grafo. A complexidade dominante é frequentemente determinada pelo número de linhas no arquivo, a menos que a busca no grafo seja significativamente mais cara.

Testes

Os testes para a classe USEI11 foram concebidos para avaliar a funcionalidade do código responsável por carregar e atualizar os horários de funcionamento dos hubs a partir de um arquivo específico. O primeiro passo consistiu na leitura de um grafo de locais a partir de um arquivo de dados grande. Em seguida, foram verificados os horários de abertura e fechamento de dois locais específicos, "CT1" e "CT214", antes da execução do método loadHubSchedules. Após a execução deste método, que carrega os horários de

funcionamento dos hubs a partir de um arquivo, os horários dos mesmos locais foram verificados novamente para confirmar se foram atualizados corretamente. A validação envolveu a comparação dos horários esperados com os horários reais, usando objetos LocalTime do Java para representar as horas. Estes testes forneceram garantias de que o código atende ao requisito de carregar e atualizar horários de hubs de maneira precisa e confiável.

4. Melhoramentos Possíveis

Com a conclusão de mais um sprint, é necessário refletir sobre o que é possível melhorar para uma próxima vez, e tendo em conta tudo que foi apontado nos últimos Sprints uma das coisas que poderia ser melhorado é o tempo despendido na realização das tarefas. Neste Sprint foi despendido muito tempo em casa US, comparativamente a outros sprints e a outras Us. Esta demora pode ser justificada devido à maior dificuldade das tarefas propostas, mas é algo que pode definitivamente ser melhorado.

Tirando este detalhe, como grupo, achamos que este sprint correu muito bem.

5. Conclusão

Em síntese, o término do Sprint 3 representa um marco significativo para a GFH, onde a implementação eficaz dos novos algoritmos impulsionou melhorias notáveis nas operações logísticas. Os avanços alcançados na gestão da oferta e procura agregadas, otimização de inventário e roteirização de entregas destacam-se como conquistas substanciais.

A coordenação aprimorada entre produtores, clientes e hubs, viabilizada pelos novos algoritmos, traduziu-se em benefícios tangíveis. Esta sinergia reforça a importância crucial dessas inovações para a eficiência operacional da GFH, delineando uma trajetória promissora para uma distribuição mais eficiente e satisfatória.

Este êxito não apenas reflete a profunda compreensão dos desafios logísticos pelo grupo de desenvolvimento, mas também destaca sua capacidade de fornecer soluções pragmáticas e eficazes. A GFH, como operador logístico, emerge fortalecida, preparada para enfrentar os desafios dinâmicos do cenário logístico com uma abordagem inovadora e eficiente.