Estruturas de Informação

Sprint 3

Feito por:

André Ferreira, 1190378

Bernardo Barbosa, 1220741

Diogo Cunha, 1221071

Tomás Peixoto,1221948

Professores:

Alberto Sampaio (ACS)

10/12/23

Índice

Responsáveis

Nesta secção, apresentamos o aluno responsável por cada uma das *User Stories* desenvolvidas no contexto do *Sprint 3* do **Projeto Integrador**.

- **USEI06** -> André Ferreira (1190378)
- **USEI07** -> Diogo Cunha (1221071)
- **USEI08** -> Bernardo Barbosa (1220741)
- **USEI09** -> Tomás Peixoto (1221948)

Glossário

Aresta: Ligação entre dois vértices em um grafo, representando a relação entre eles.

Algoritmo de Kruskal: Algoritmo utilizado para encontrar a árvore geradora de custo mínimo em um grafo.

Cabaz: Cesto com conjunto de produtos agrícola colhidos (ex.: Tomates, cenouras, etc.).

Caminho de Menor Custo: Rota entre dois vértices em um grafo que possui o menor custo possível.

Complexidade Temporal: Uma medida do número de operações ou recursos necessários para executar um algoritmo, geralmente expressa em termos de tempo (melhor caso, pior caso).

Grafo: Estrutura que representa um conjunto de vértices e arestas que os conectam.

GFH: Operador logístico responsável pela distribuição de cabazes com produtos agrícolas em rede.

Hub: Instituição ou empresa onde são entregues e posteriormente levantados os cabazes.

Localidade: Representação de uma área geográfica onde podem existir hubs de distribuição.

Rede de Distribuição: Estrutura constituída por vários vértices representativos de localidades, onde podem existir hubs, e arestas representativas das distâncias entre essas localidades.

Veículo Elétrico: Meio de transporte utilizado para transportar os cabazes, com autonomia limitada e disponível nos hubs para posterior recolha.

Vértice: Ponto fundamental em um grafo que representa uma entidade ou local.

Introdução

Neste relatório, pretendemos apresentar a logística contida na resolução de cada uma das funcionalidades propostas para o *Sprint* 3 do **Projeto Integrador**, no contexto da unidade curricular **Estruturas de Informação**.

O principal objetivo durante as nossas resoluções foi a aplicação dos conceitos e técnicas aprendidas nessa mesma unidade curricular (nomeadamente, os **grafos**), mantendo as boas práticas de programação orientada a objetos (***OO Programming***).

O grupo pretendeu obter a máxima compreensão e simplicidade ao longo do relatório, mantendo a sua eficácia. Para alcançar este objetivo, foi criado um **diagrama de classes**, que será apresentado posteriormente.

Cada *User Story* contido neste relatório tem um título, figura (ou figuras) e uma breve, mas concisa explicação de como o algoritmo funciona, resolvendo a questão proposta. A análise de complexidade para cada funcionalidade também é incluída.

Nota: As ***USEI10*** e ***USEI11*** não serão apresentadas aqui. A ***USEI10*** não foi atribuída a grupos com apenas 4 elementos, e a ***USEI11*** foi desenvolvida apenas no contexto da unidade curricular **Laboratório/Projeto 3**.

Por fim, o relatório apresenta as fontes das quais foram adquiridas conhecimento utilizado para a resolução e melhor compreensão to trabalho

USEI06 – Encontrar diferentes percursos

Para resolver o problema desta US utilizei uma adaptação conhecida do algoritmo Depth First Search.
Essa adaptação envolve fazer backtracking dos caminhos que já foram descobertos com DFS entre a origem e o destino.
Por outras palavras, uma vez que queremos descobrir todos os caminhos entre a origem e o destino estamos sujeitos a que
uma node (localidade) contenha mais que um caminho possível, daí ser necessário usar backtrack para voltar a essa node e
descobrir os outros caminhos possíveis. Além desta adaptação foi também necessário utilizar uma condição adicional, a da
autonomia do veículo. Essa condição foi implementada no próprio DFS com backtrack onde adicionei a condição adicional de verificar
se a distância entre a node atual e a próxima a ser percorrida é menor ou igual à autonomia. Outro ponto importante a explicar é a
condição de paragem do DFS com backtrack (uma vez que a natureza do algoritmo é recursiva). Esta condição é muito simplesmente o
caso de a node a atualmente percorrida ser igual ao destino. Quando isto acontece o algoritmo prossegue com a realização do backtrack.

Complexidade:

A complexidade para o pior caso é o(2^n), onde n representa o número de vértices do grafo. Isto é assim porque no pior caso o algoritmo tem de visitar todos os caminhos possíveis contidos no grafo entre a origem e o destino, e este número é exponencial. Isto também significa
que a autonomia do veículo permite percorrer todos os caminhos possíveis entre a origem e o destino.
* A complexidade para o melhor caso é constante uma vez que o melhor caso é quando existe apenas
um caminho possível entre a origem e o destino e o tamanho desse caminho (número de edges) é 1.

Método principal:

```
public static List<LinkedList<Localidade>> calcularPercursosLimite(MapGraph<Localidade, Integer> rede, Localidade origem, Localidade destino, double autonomia) {
        List<LinkedList<Localidade>> percursos = new ArrayList<>();
        Map<Localidade, Boolean> visitadas = new HashMap<>();
        for (Localidade vertex : rede.vertices()) {
            visitadas.put(vertex, false);
        }
        LinkedList<Localidade> currentPath = new LinkedList<>();
        depthFirstSearch(rede, origem, destino, visitadas, currentPath, percursos, autonomia, 0.0);
        return percursos;
    }
```

USEI07 – Encontrar percurso com número máximo de *hubs*

Explicação da Resolução

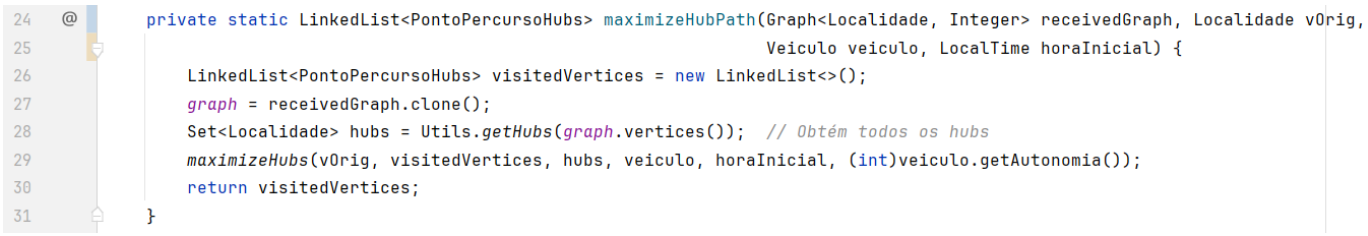
```
99  @   public static PercursoHubs obterPercursoComMaxNumHubs(Graph<Localidade, Integer> grafo, int n, LocalTime horaInicial,
100                                                           Localidade localInicial, Veiculo veiculo){
101         Map<CriteriosLocalidade, Localidade> map = ObterHubsController.obterTopNLocalidades(grafo, n);
102         for(Map.Entry<CriteriosLocalidade, Localidade> entry : map.entrySet())
103             Utils.getLocalidadeById(entry.getValue().getId(), grafo.vertices()).setHub(true);
104         List<PontoPercursoHubs> pontos = maximizeHubPath(grafo, localInicial, veiculo, horaInicial);
105
106         return new PercursoHubs(localInicial, pontos.get(pontos.size()-1).getLocal(), horaInicial,
107                 pontos.get(pontos.size() - 1).getHoraDeSaida(), pontos);
108     }
```

*Figura 1. Método **obterPercursoComMaxNumHubs***

Nesta *User Story*, este método, ***obterPercursoComMaxNumHubs***, é o método principal. Contudo, é apenas o início dos cálculos feitos para o objetivo que foi pedido. Este método aceita os seguintes objetos como parâmetros:

- O grafo (um clone do original, para o original não sofrer alterações permanentes),
- Um número inteiro que determina quantos *hubs* o grafo irá ter,
- A hora em que este percurso irá começar,
- O local inicial do percurso,
- O veículo utilizado, com os seus dados (velocidade média, autonomia, tempo médio de carregamento da bateria e tempo médio de descarregamento dos cabazes);

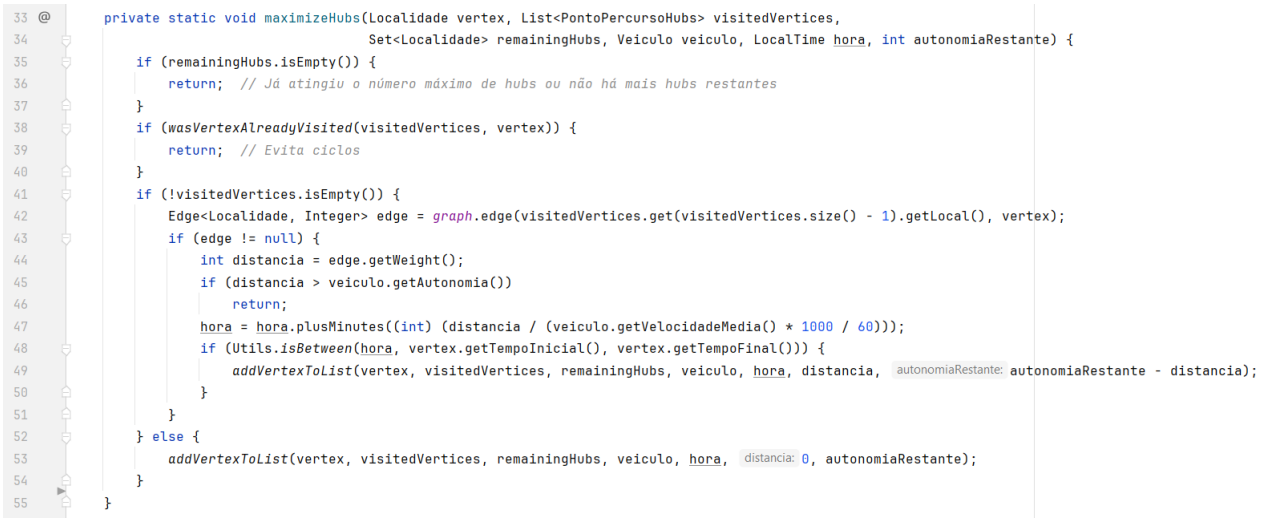
Aqui, invocamos a função desenvolvida na ***USEI02*** (linha 101), tornando todas as localidades retornadas em *hubs* (linhas 102 e 103). Após obter essa informação, chama o método ***maximizeHubPath*** para calcular o caminho e devolver todos os seus pontos, cada um com as informações necessárias. Estes detalhes serão explicados posteriormente.



```
24  @   private static LinkedList<PontoPercursoHubs> maximizeHubPath(Graph<Localidade, Integer> receivedGraph, Localidade vOrig,
25                                                                   Veiculo veiculo, LocalTime horaInicial) {
26          LinkedList<PontoPercursoHubs> visitedVertices = new LinkedList<>();
27          graph = receivedGraph.clone();
28          Set<Localidade> hubs = Utils.getHubs(graph.vertices());  // Obtém todos os hubs
29          maximizeHubs(vOrig, visitedVertices, hubs, veiculo, horaInicial, (int)veiculo.getAutonomia());
30          return visitedVertices;
31      }
```

*Figura 2. Método **maximizeHubPath***

Neste método, criamos uma lista (linha 26) que será preenchida pelos dois próximos métodos. A invocação do primeiro destes, o ***maximizeHubs***, é preparada também neste método, invocando-o com o primeiro local do percurso, uma lista vazia dos locais visitados (que será preenchida), uma lista preenchida com todos os *hubs* do grafo (criada na linha 28), a hora inicial do percurso, e a autonomia inteira do veículo.



```
33  @   private static void maximizeHubs(Localidade vertex, List<PontoPercursoHubs> visitedVertices,
34                                       Set<Localidade> remainingHubs, Veiculo veiculo, LocalTime hora, int autonomiaRestante) {
35          if (remainingHubs.isEmpty()) {
36              return;  // Já atingiu o número máximo de hubs ou não há mais hubs restantes
37          }
38          if (wasVertexAlreadyVisited(visitedVertices, vertex)) {
39              return;  // Evita ciclos
40          }
41          if (!visitedVertices.isEmpty()) {
42              Edge<Localidade, Integer> edge = graph.edge(visitedVertices.get(visitedVertices.size() - 1).getLocal(), vertex);
43              if (edge != null) {
44                  int distancia = edge.getWeight();
45                  if (distancia > veiculo.getAutonomia())
46                      return;
47                  hora = hora.plusMinutes((int) (distancia / (veiculo.getVelocidadeMedia() * 1000 / 60)));
48                  if (Utils.isBetween(hora, vertex.getTempoInicial(), vertex.getTempoFinal())) {
49                      addVertexToList(vertex, visitedVertices, remainingHubs, veiculo, hora, distancia,  autonomiaRestante: autonomiaRestante - distancia);
50                  }
51              }
52          } else {
53              addVertexToList(vertex, visitedVertices, remainingHubs, veiculo, hora,  distancia: 0, autonomiaRestante);
54          }
55      }
```

*Figura 3. Método **maximizeHubs***

Este método é o primeiro dos métodos que processam o caminho a calcular, recebendo os parâmetros necessários para cada vez que será invocado (o método será invocado repetidamente ao longo do cálculo do caminho).

No início, este método verifica duas das condições de paragem.

- A lista de *hubs* estar vazia (linha 35),
- Este vértice já está contido na lista de locais visitados (assim, evitamos ciclos) (linha 38);

A partir daqui, se a lista de vértices visitados não estiver vazia (a verificação está na linha 41), verificamos se existe uma ligação entre o vértice atual e o último vértice na lista de vértices visitados (linha 43), caso exista, vemos se o carro tem autonomia suficiente para fazer essa distância (linha 45) e se sim, a duração da viagem (linha 47) e verificamos se a hora de chegada está contida no horário do local para o qual vamos (linha 48).

Se passarmos estas verificações todas, passamos para o próximo método ***addVertexToList***, sendo que a distância é a calculada previamente e a autonomia restante é a prévia menos a distância do percurso (linha 49).

Se a lista de vértices visitados tiver vazia, passamos para o método ***addVertexToList*** diretamente sem ver as outras verificações, com a distância igual a 0 e a autonomia não alterada (linha 53).

```
62 @    private static void addVertexToList(Localidade vertex, List<PontoPercursoHubs> visitedVertices, Set<Localidade> remainingHubs,
63                                         Veiculo veiculo, LocalTime hora, int distancia, int autonomiaRestante) {
64          LocalTime horaFinal = hora;
65          if (vertex.isHub() && remainingHubs.contains(vertex)) {
66              //System.out.println(vertex);  // Se desejar imprimir apenas hubs
67              remainingHubs.remove(vertex);  // Remove o hub da lista de hubs restantes
68              horaFinal = hora.plusMinutes((int)veiculo.getTempoMediaDescargaCabazes());
69          }
70
71          if(autonomiaRestante < getBiggestDistance(vertex, Utils.getLocalidadesFromPontosDePercurso(visitedVertices))) {
72              autonomiaRestante = (int) veiculo.getAutonomia();
73              horaFinal = horaFinal.plusMinutes((int)veiculo.getTempoMediaCarregamentoBateria());
74          }
75
76          visitedVertices.add(new PontoPercursoHubs(vertex, hora, horaFinal, distancia));
77
78          currentVertex = vertex;
79          List<Localidade> locaisAdjacentes = (List<Localidade>) graph.adjVertices(vertex);
80          locaisAdjacentes.sort(cmp);
81
82          for (Localidade neighbor : locaisAdjacentes) {
83              int previousSize = visitedVertices.size();
84              maximizeHubs(neighbor, visitedVertices, remainingHubs, veiculo, horaFinal, autonomiaRestante);
85              if(visitedVertices.size() != previousSize)
86                  break;
87          }
88      }
```

*Figura 4. Método **addVertexToList***

Este é o método responsável por inserir o vértice na lista e preparar a próxima chamada do ***maximizeHubs***. Este recebe os seguintes objetos como parâmetros:

- Vértice a ser inserido,
- Lista de vértices já visitados,
- Lista de *hubs* ainda não presentes na lista,
- Veículo a ser usado,
- Hora de chegada no vértice atual,
- Distância do vértice anterior a este,
- Autonomia restante no veículo;

Antes de inserirmos o novo ponto na lista de vértices visitados, pretendemos calcular a sua hora de saída. Para isso, verificamos duas coisas:

- Se o local atual for um *hub*, adicionamos o tempo médio de descarregamento dos cabazes desse veículo, e eliminamos esse local da lista de *hubs* ainda não visitados (linhas 65-69),
- Se a autonomia restante do veículo for menor que a maior distância nos locais adjacentes ainda não visitados, teremos de carregar a bateria do automóvel, adicionando assim o tempo médio de carregamento da sua bateria (linhas 71-74);

Depois disto, adicionamos este novo ponto do percurso à lista de pontos visitados (linha 76). De seguida, vamos invocar o método ***maximizeHubs*** para os pontos adjacentes a este, esta lista de pontos é ordenada pelos seguintes critérios de comparação entre um ponto *X* e um ponto *Y*:

1. Se *X* for um *hub* e *Y* não, então *X* tem mais prioridade,
2. Se ambos forem *hubs* (ou nenhum for), então tem prioridade o que estiver mais próximo do vértice atual (o que foi recentemente adicionado à lista);

Por último, se a chamada do ***maximizeHubs*** alterar a lista (vemos isso pela mudança do seu tamanho), quebramos o *loop* ***for***, assim, evitamos um percurso errado deste género:

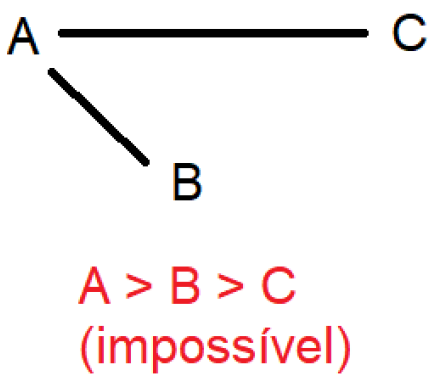


Figura 5. Exemplo de um percurso a evitar

Análise da Complexidade

Vendo que esta funcionalidade chama a da ***USEI02***, também tem a sua complexidade de **$O(V^3xE)$**.

Ignorando essa parte, vemos que os métodos ***maximizeHubs*** e ***addVertexToList*** comportam-se como um método recursivo que se repete até uma das condições de paragem anteriormente explicadas ser encontrada. Por mais que seja uma execução recursiva, não tem muitos ciclos.

Estudando os métodos que estes chamam e ciclos que começam, uns feitos para atravessarem todos os vértices e outros para várias arestas, mas não os dois ao mesmo tempo, podemos atribuir uma complexidade temporal final de **$O((V+E)log(V+E))$** a esta funcionalidade.

USEI08 – Encontrar circuito de entrega

Explicação da Resolução

```
95  @   public static CircuitoHubs obtainCircuit(Graph<Localidade, Integer> grafo, Localidade origin, Veiculo veiculo, int n){
96          Map<CriteriosLocalidade, Localidade> map = ObterHubsController.obterTopNLocalidades(grafo, n);
97          for(Map.Entry<CriteriosLocalidade, Localidade> entry : map.entrySet())
98              Utils.getLocalidadeById(entry.getValue().getId(), grafo.vertices()).setHub(true);
99          LinkedList<PontoCircuitoHubs> localidades = maximizeHubPath(grafo, origin, veiculo);
100         Localidade ultimoLocal = localidades.removeLast().getLocal();
101         ShortestPathMapGraph<Localidade, Integer> smpg = new ShortestPathMapGraph<>();
102         addShortestPathToList(localidades, smpg.getShortestPathLinkedList(grafo, ultimoLocal, origin), grafo, veiculo);
103         return new CircuitoHubs(origin, localidades);
104     }
```

*Figura 6. Método **obtainCircuit***

Nesta *User Story*, este método, ***obtainCircuit***, é o método principal. Contudo, semelhante ao que aconteceu na resolução da **USEI07**, é apenas o início dos cálculos feitos para o objetivo que foi pedido. Este método aceita os seguintes objetos como parâmetros:

- O grafo (um clone do original, para o original não sofrer alterações permanentes),
- Um número inteiro que determina quantos *hubs* o grafo irá ter,
- O local inicial do percurso,
- O veículo utilizado, com os seus dados (velocidade média, autonomia, tempo médio de carregamento da bateria e tempo médio de descarregamento dos cabazes);

Aqui, invocamos a função desenvolvida na ***USEI02*** (linha 96), tornando todas as localidades retornadas em *hubs* (linhas 97 e 98). Após obter essa informação, chama o método ***maximizeHubPath*** para calcular o caminho e devolver todos os seus pontos, cada um com as informações necessárias. Estes detalhes serão explicados posteriormente.

Após o cálculo deste caminho, calculamos o **caminho mais curto entre este ponto final do caminho calculado anteriormente e o ponto inicial, enviado por parâmetro**, para terminar o circuito pedido. Neste segundo caminho, não nos preocupamos com a passagem ou não passagem por vértices já passados anteriormente (decisão tomada devido à uma resposta no fórum).

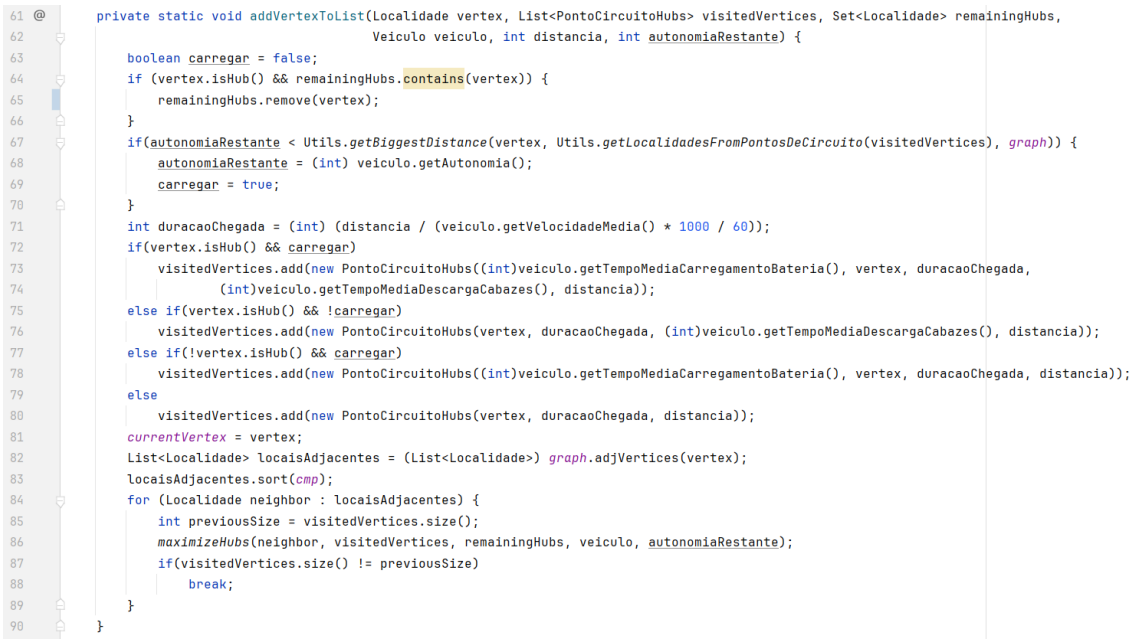
```java
private static LinkedList<PontoCircuitoHubs> maximizeHubPath(Graph<Localidade, Integer> receivedGraph,
                                                             Localidade vOrig, Veiculo veiculo) {
    LinkedList<PontoCircuitoHubs> visitedVertices = new LinkedList<>();
    graph = receivedGraph.clone();
    Set<Localidade> hubs = Utils.getHubs(graph.vertices());  // Obtém todos os hubs
    maximizeHubs(vOrig, visitedVertices, hubs, veiculo,(int)veiculo.getAutonomia());
    return visitedVertices;
}

2 usages  Diogo
private static void maximizeHubs(Localidade vertex, List<PontoCircuitoHubs> visitedVertices,
                                 Set<Localidade> remainingHubs, Veiculo veiculo, int autonomiaRestante) {
    if (remainingHubs.isEmpty()) {
        return;  // Já atingiu o número máximo de hubs ou não há mais hubs restantes
    }
    if (wasVertexAlreadyVisited(visitedVertices, vertex)) {
        return;  // Evita ciclos
    }
    if (!visitedVertices.isEmpty()) {
        Edge<Localidade, Integer> edge = graph.edge(visitedVertices.get(visitedVertices.size() - 1).getLocal(), vertex);
        if (edge != null) {
            int distancia = edge.getWeight();
            if (distancia > veiculo.getAutonomia())
                return;
            addVertexToList(vertex, visitedVertices, remainingHubs, veiculo, distancia,  autonomiaRestante: autonomiaRestante - distancia);
        }
    } else {
        addVertexToList(vertex, visitedVertices, remainingHubs, veiculo,  distancia: 0, autonomiaRestante);
    }
}
```

*Figura 7. Método **maximizeHubPath***

Método praticamente igual ao anteriormente encontrado na **USEI07**. As diferenças sendo:

- Na função inicial:
  - A hora não está incluída,
  - Em vez de ser devolvida uma lista de ***PontoPercursoHub***, é devolvida uma lista de ***PontoCircuitoHub***, um objeto com atributos diferentes de acordo com o pedido;
- Na função recursiva:
  - A hora não está incluída, logo as verificações com os horários das localidades também não existem,



```
61  @   private static void addVertexToList(Localidade vertex, List<PontoCircuitoHubs> visitedVertices, Set<Localidade> remainingHubs,
62                                          Veiculo veiculo, int distancia, int autonomiaRestante) {
63          boolean carregar = false;
64          if (vertex.isHub() && remainingHubs.contains(vertex)) {
65              remainingHubs.remove(vertex);
66          }
67          if(autonomiaRestante < Utils.getBiggestDistance(vertex, Utils.getLocalidadesFromPontosDeCircuito(visitedVertices), graph)) {
68              autonomiaRestante = (int) veiculo.getAutonomia();
69              carregar = true;
70          }
71          int duracaoChegada = (int) (distancia / (veiculo.getVelocidadeMedia() * 1000 / 60));
72          if(vertex.isHub() && carregar)
73              visitedVertices.add(new PontoCircuitoHubs((int)veiculo.getTempoMediaCarregamentoBateria(), vertex, duracaoChegada,
74                      (int)veiculo.getTempoMediaDescargaCabazes(), distancia));
75          else if(vertex.isHub() && !carregar)
76              visitedVertices.add(new PontoCircuitoHubs(vertex, duracaoChegada, (int)veiculo.getTempoMediaDescargaCabazes(), distancia));
77          else if(!vertex.isHub() && carregar)
78              visitedVertices.add(new PontoCircuitoHubs((int)veiculo.getTempoMediaCarregamentoBateria(), vertex, duracaoChegada, distancia));
79          else
80              visitedVertices.add(new PontoCircuitoHubs(vertex, duracaoChegada, distancia));
81          currentVertex = vertex;
82          List<Localidade> locaisAdjacentes = (List<Localidade>) graph.adjVertices(vertex);
83          locaisAdjacentes.sort(cmp);
84          for (Localidade neighbor : locaisAdjacentes) {
85              int previousSize = visitedVertices.size();
86              maximizeHubs(neighbor, visitedVertices, remainingHubs, veiculo, autonomiaRestante);
87              if(visitedVertices.size() != previousSize)
88                  break;
89          }
90      }
```

*Figura 8. Método **addVertexToList***

Outro método semelhante à versão da **USEI07**. Novamente com a falta de horas, mas há mais diferenças.

Depois de ser analisado se o local em questão é um *hub* (linhas 64-67) e se vai ser necessário carregar o veículo antes da próxima viagem (linhas 68-71), vamos analisar estes dois resultados para inserir um novo ***PontoCircuitoHubs*** com os atributos necessários (linhas 73-81).

Assim como na **USEI07**, iremos ver os vértices adjacentes ao atual, mas o critério de ordenação destes é feito de outra forma:

1. Se *X* for um *hub* e *Y* não, então *X* tem mais prioridade,
2. Se ambos forem *hubs*, *X* tem mais prioridade se tiver mais colaboradores que *Y* (visível no nome da localidade),
3. Se nenhum for *hub*, então tem prioridade o que estiver mais próximo do vértice atual (o que foi recentemente adicionado à lista);

```java
private static void addShortestPathToList(List<PontoCircuitoHubs> pontoCircuitoHubs, List<Localidade> shortestPath,
                                          Graph<Localidade, Integer> grafo, Veiculo veiculo){
    int distancia = grafo.edge(shortestPath.get(0), pontoCircuitoHubs.get(pontoCircuitoHubs.size()-1).getLocal()).getWeight();
    int duracaoChegada = (int) (distancia / (veiculo.getVelocidadeMedia() * 1000 / 60));
    if(shortestPath.get(0).isHub())
        pontoCircuitoHubs.add(new PontoCircuitoHubs((int)veiculo.getTempoMediaCarregamentoBateria(), shortestPath.get(0), duracaoChegada,
                (int)veiculo.getTempoMediaDescargaCabazes(), distancia));
    else
        pontoCircuitoHubs.add(new PontoCircuitoHubs((int)veiculo.getTempoMediaCarregamentoBateria(), shortestPath.get(0), duracaoChegada, distancia));
    int autonomiaRestante = (int)veiculo.getAutonomia();

    int size = shortestPath.size();
    boolean carregar = false;
    for(int i = 1; i < size; i++){
        distancia = grafo.edge(shortestPath.get(i), shortestPath.get(i-1)).getWeight();
        duracaoChegada = (int) (distancia / (veiculo.getVelocidadeMedia() * 1000 / 60));
        if(autonomiaRestante < distancia) {
            autonomiaRestante = (int) veiculo.getAutonomia();
            carregar = true;
        }
        if(shortestPath.get(i).isHub() && carregar)
            pontoCircuitoHubs.add(new PontoCircuitoHubs((int)veiculo.getTempoMediaCarregamentoBateria(), shortestPath.get(i), duracaoChegada,
                    (int)veiculo.getTempoMediaDescargaCabazes(), distancia));
        else if(shortestPath.get(i).isHub() && !carregar)
            pontoCircuitoHubs.add(new PontoCircuitoHubs(shortestPath.get(i), duracaoChegada, (int)veiculo.getTempoMediaDescargaCabazes(), distancia));
        else if(!shortestPath.get(i).isHub() && carregar)
            pontoCircuitoHubs.add(new PontoCircuitoHubs((int)veiculo.getTempoMediaCarregamentoBateria(), shortestPath.get(i), duracaoChegada, distancia));
        else
            pontoCircuitoHubs.add(new PontoCircuitoHubs(shortestPath.get(i), duracaoChegada, distancia));
        carregar = false;
    }
}
```

*Figura 9. Método **addShortestPathToList***

Após o percurso inicial ser criado nos métodos anteriores, este o caminho mais curso (calculado no método ***obtainCircuit***) entre o ponto final anterior (que foi removido, para ser adicionado novamente aqui) e o ponto inicial do percurso, de forma a terminar o ciclo) serão enviados para o método ***addShortestPathToList***.

Para todos os pontos do segundo caminho, a distância desse para o anterior, assim como o tempo que demora a lá chegar e os possíveis tempos de carregamento e descarregamento serão calculados, usando uma lógica igual à apresentada na explicação do método ***addVertexToList***.

Assim, teremos uma lista que começa no ponto que queremos, passa por N *hubs* e volta ao ponto inicial de forma a ter o caminho mais curto possível.

Análise da Complexidade

Vendo o número de métodos maioritariamente partilhados entre esta *User Story* e a anterior, podemos concluir que a sua complexidade temporal será igual a essa (os métodos únicos a esta *User Story* e o cálculo do caminho mais curto entre dois pontos não apresentam uma complexidade superior à restante).

Logo, afirmamos que a ***Time-Complexity*** desta funcionalidade é de **O((V+E)log(V+E))**.

USEI09 – Obter localidades em *clusters*

```java
package controller;

import ...

2 usages  1221948 +1
public class USEI09OrganizarLocalidadesController {

    1 usage  1221948
    public static Map<Localidade, Collection<Localidade>> isolateGraph(Graph<Localidade, Integer> graph) {
        Map<Localidade, Collection<Localidade>> map = new HashMap<>();
        ArrayList<Localidade> vertices = graph.vertices();
        while (graph.numEdges() > 0) {
            for (Localidade vertex : vertices) {
                Collection<Localidade> adjVertices = graph.adjVertices(vertex);
                map.put(vertex, new ArrayList<>(adjVertices)); // Store the adjacent vertices before removing the edges
                for (Localidade adjVertex : adjVertices) {
                    graph.removeEdge(vertex, adjVertex);
                }
            }
        }
        return map;
    }
}
```

```java
package ui;

import ...

1 usage  ± 1221948 +1
public class OrganizarLocalidadesUI {
    ± 1221948 +1
    public static void run(Graph<Localidade, Integer> grafo){
        Map<Localidade, Collection<Localidade>> map = USEI09OrganizarLocalidadesController.isolateGraph(grafo);

        for (Map.Entry<Localidade, Collection<Localidade>> entry : map.entrySet()) {
            System.out.println("Vertex: " + entry.getKey());
            System.out.println("Adjacent vertices: " + entry.getValue());
        }
    }
}
```

Método `USEI09OrganizarLocalidadesController.isolateGraph(Graph<Localidade, Integer> graph):

Este método é um método estático na classe USEI09OrganizarLocalidadesController, projetado para isolar um grafo removendo todas as suas arestas e organizando as informações em um mapa. O método recebe um grafo genérico (Graph<Localidade, Integer> graph) como entrada, onde Localidade representa os vértices do grafo e Integer representa os pesos das arestas.

- O método inicializa um HashMap chamado map para armazenar o resultado, onde cada vértice está associado a uma coleção de seus vértices adjacentes.
- Ele recupera todos os vértices do grafo de entrada usando ArrayList<Localidade> vertices = graph.vertices();.
- O método então entra em um loop while que continua até que todas as arestas do grafo sejam removidas.
- Dentro do loop, ele itera sobre cada vértice no grafo, recupera seus vértices adjacentes, armazena os vértices adjacentes no map e remove as arestas entre o vértice atual e seus vértices adjacentes.
- O processo continua até que não haja mais arestas no grafo.
- Finalmente, o método retorna o map contendo as informações isoladas.

Método `OrganizarLocalidadesUI.run(Graph<Localidade, Integer> grafo):

Este método faz parte da classe OrganizarLocalidadesUI e serve como ponto de entrada para executar o processo de isolamento e exibir os resultados.

- O método chama o método isolateGraph da classe USEI09OrganizarLocalidadesController, passando o grafo de entrada (grafo) como argumento. O resultado é armazenado em um mapa chamado map.
- Em seguida, ele itera sobre as entradas do mapa, imprimindo o vértice e seus vértices adjacentes associados no console.
- A saída fornece informações sobre cada vértice e seus vértices adjacentes isolados, ajudando a visualizar a organização do grafo após a remoção das arestas.

Observações

Referências

Informação Global do Projeto:

- Ficheiro ISEP Moodle descrição do Projeto:
  - *Enunciado Projecto Integrador - Versao 1.3 (29/nov)Ficheiro*

- Outros ficheiros de informação no Moodle:
  - *distancias_big.csv*
  - *distancias_small.csv*
  - *localidades_big.csv*
  - *localidades_small.csv*

Recursos adicionais usados para implementações de métodos Java:

- YouTube Video: "**java Unit Testing with JUnit - Tutorial - How to Create And Use Unit Tests**" de *Coding with John*. Link: https://youtu.be/gN29Y600k5g