

|  |
| --- |
| RELATÓRIO ESINF  Projeto Integrador  2022/2023 |
|  |
| 4 de dezembro  JOSÉ GOUVEIA - 1211089  PEDRO PEREIRA – 1211131  TIAGO OLIVEIRA – 1211128  ALEXANDRE GERAÇÃO – 1211151  RICARDO VENÂNCIO – 1210828  DIOGO CARVALHO - 1200611 |

# Índice

Conteúdo

[Índice 2](#_Toc121089196)

[US301 3](#_Toc121089197)

[Load File 4](#_Toc121089198)

[Validate Path 6](#_Toc121089199)

[US302 7](#_Toc121089200)

[Conectividade 8](#_Toc121089201)

[Número mínimo de ligações 9](#_Toc121089202)

[US303 10](#_Toc121089203)

[US304 13](#_Toc121089204)

[US305 15](#_Toc121089205)

[Algoritmos 18](#_Toc121089206)

[Breadth First Search 18](#_Toc121089207)

[Depth First Search 19](#_Toc121089208)

[All Paths 20](#_Toc121089209)

[Shortest Path Dijkstra 21](#_Toc121089210)

[Get Vertex Minimum Distance 22](#_Toc121089211)

[Shortest Path 23](#_Toc121089212)

[Shortest Paths 24](#_Toc121089213)

[Get Path 25](#_Toc121089214)

[Minimum Distance Graph 26](#_Toc121089215)

[Graph Diameter 27](#_Toc121089216)

[Minimum Spanning Tree 28](#_Toc121089217)

[Build MST 29](#_Toc121089218)

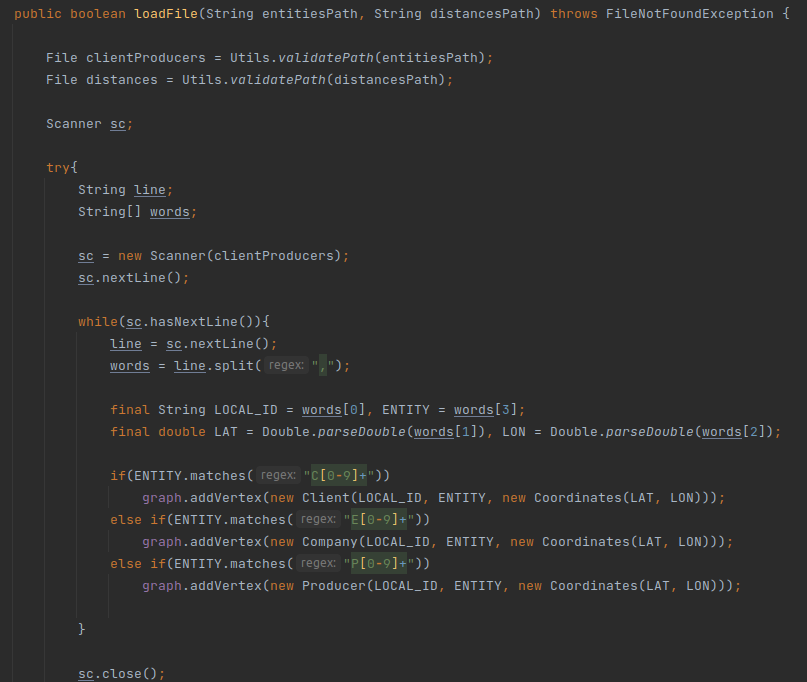
# US301

Construir a rede de distribuição de cabazes a partir da informação fornecida nos ficheiros. O grafo deve ser implementado usando a representação mais adequada e garantindo a manipulação indistinta dos clientes/empresas e produtores agrícolas (código cliente: C, código empresa: E, código produtor: P).

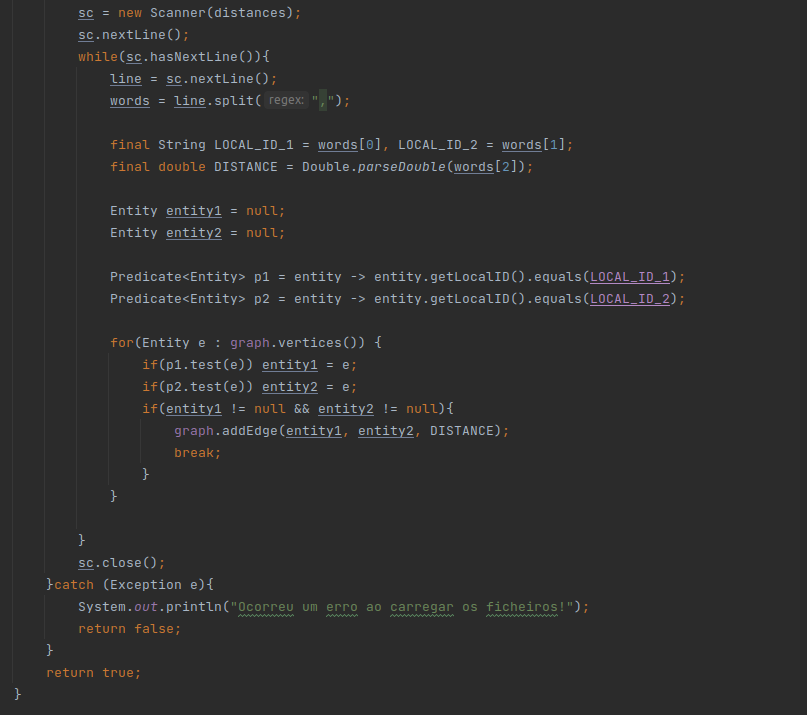
Para implementar esta funcionalidade, foi criada uma classe com o nome de *DataIO (Data Input Output)* que vai pedir por dois ficheiros, um de clientes/produtores (ficheiro de vértices) e outro de distâncias (ficheiro de arestas). Esta classe contém métodos como *loadFile* e *getGraph,* como os nomes sugerem, uma vai carregar ficheiros e adaptá-los numa forma de grafo e o outro vai retornar o grafo. Sempre que um novo objeto do tipo *DataIO* é instanciado, um grafo (com implementação de mapas) é instanciado.

## Load File

O método *loadFile* recebe dois caminhos (um para vértices e outro para arestas) e começa por processar o ficheiro dos vértices. Linha a linha vai identificar o tipo de entidade (produtor, empresa ou cliente) e criar uma instância nova consoante o seu tipo. Através de uma expressão regular simples, vai ser identificado o tipo (todas as empresas começam por “E”, clientes por “C” e produtores por “P”).



De seguida vai ser processado o ficheiro das arestas. Como o ficheiro fornece as identificações de localização dos vértices, é necessário verificar a que nome é associada essa identificação de localização. Para descobrir, são utilizados dois *Predicate* que vão comparar a identificação de localização. É percorrido, através de um *loop*, todas as entidades (vértices) e após encontrar as entidades com a identificações iguais, a aresta é criada com o devido custo, no caso distância.



Conclui-se assim que este método tem complexidade:

## Validate Path

O método validate path recebe um caminho por argumento e vai verificar se realmente é um ficheiro válido, caso seja, vai retornar um objeto do tipo ficheiro, caso contrário, vai lançar uma exceção, ora *NullPointerException*, ora *FileNotFoundException,* se o caminho for nulo e se o caminho não levar a um ficheiro.

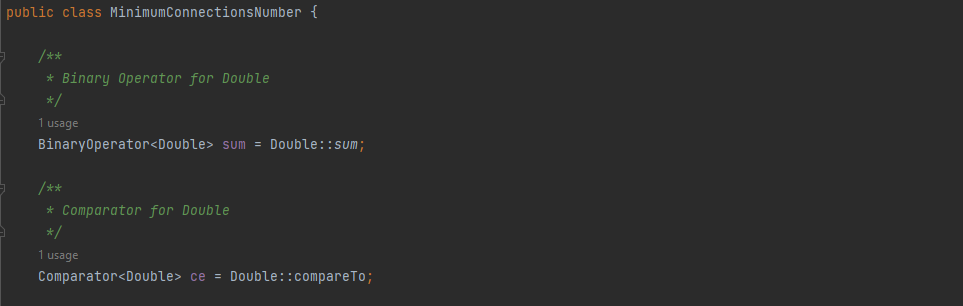
Uma imagem com texto

Descrição gerada automaticamente

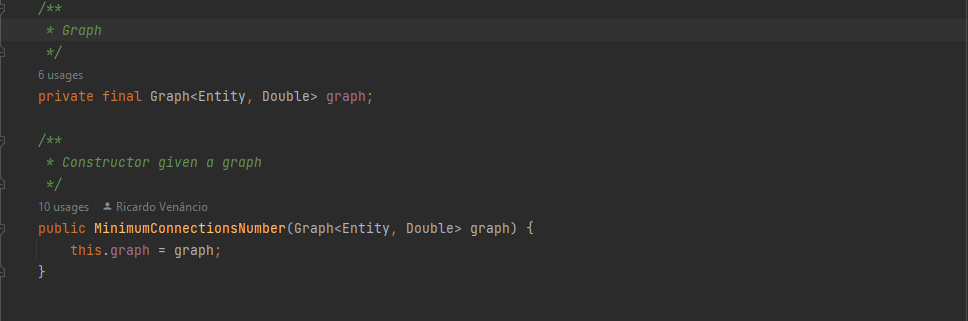
# US302

Verificar se o grafo carregado é conexo e devolver o número mínimo de ligações necessário para nesta rede qualquer cliente/produtor conseguir contactar um qualquer outro.

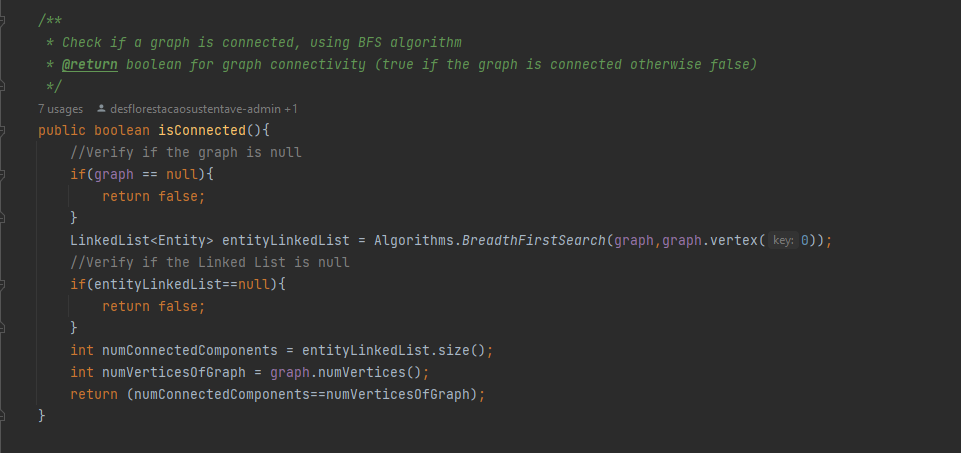
Para a realização desta US começamos por criar um Binary Operator e um Comparator para serem usados futuramente.



De seguida criamos um construtor que usava um Grafo.



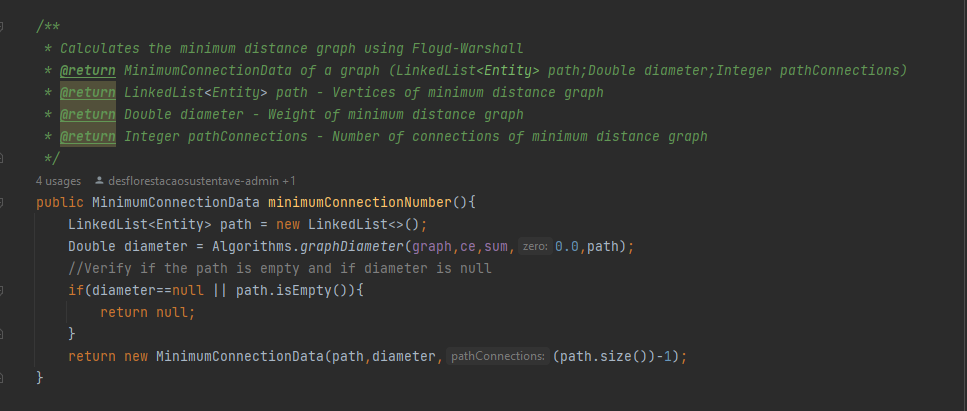
## Conectividade



Para verificar se um grafo é conexo utilizamos o Algoritmo Breadth First Search que apresenta uma complexidade O(|V|+|E|) e de seguida verificamos o tamanho da lista recebida através do algoritmo com complexidade O(1) e a lista com o número de vértices do grafo com complexidade O(1) de seguida comparamos os dois valores e se forem iguais retornamos verdadeiro.

Portanto este algoritmo tem uma complexidade de O(|V|+|E|).

## Número mínimo de ligações



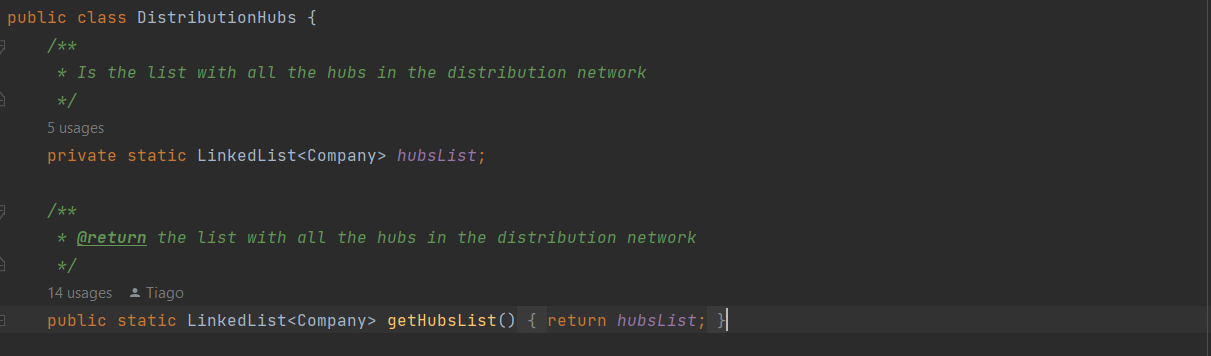
Para achar o número mínimo de ligações neste grafo precisamos de chamar o algoritmo Graph Diameter que utiliza o Algoritmo de Floyd-Warshall que apresenta uma complexidade O(V3) este algoritmo também preenche uma lista passada por parâmetro com os vértices referentes á ligação mínima e por fim cria e retorna um objeto que contem esta lista de vértices, o peso e a quantidade das arestas referentes.

Portanto este algoritmo tem uma complexidade de O(V3)

# US303

Definir os hubs da rede de distribuição, ou seja, encontrar as N empresas mais próximas de todos os pontos da rede (clientes e produtores agrícolas). A medida de proximidade deve ser calculada como a média do comprimento do caminho mais curto de cada empresa a todos os clientes e produtores agrícolas. (small, N=3)

Para realizar esta US é necessário declarar uma lista global para armazenar os *hubs* da rede de distribuição e, por consequente, um método para aceder a esta lista fora da classe *DistributionHubs*.

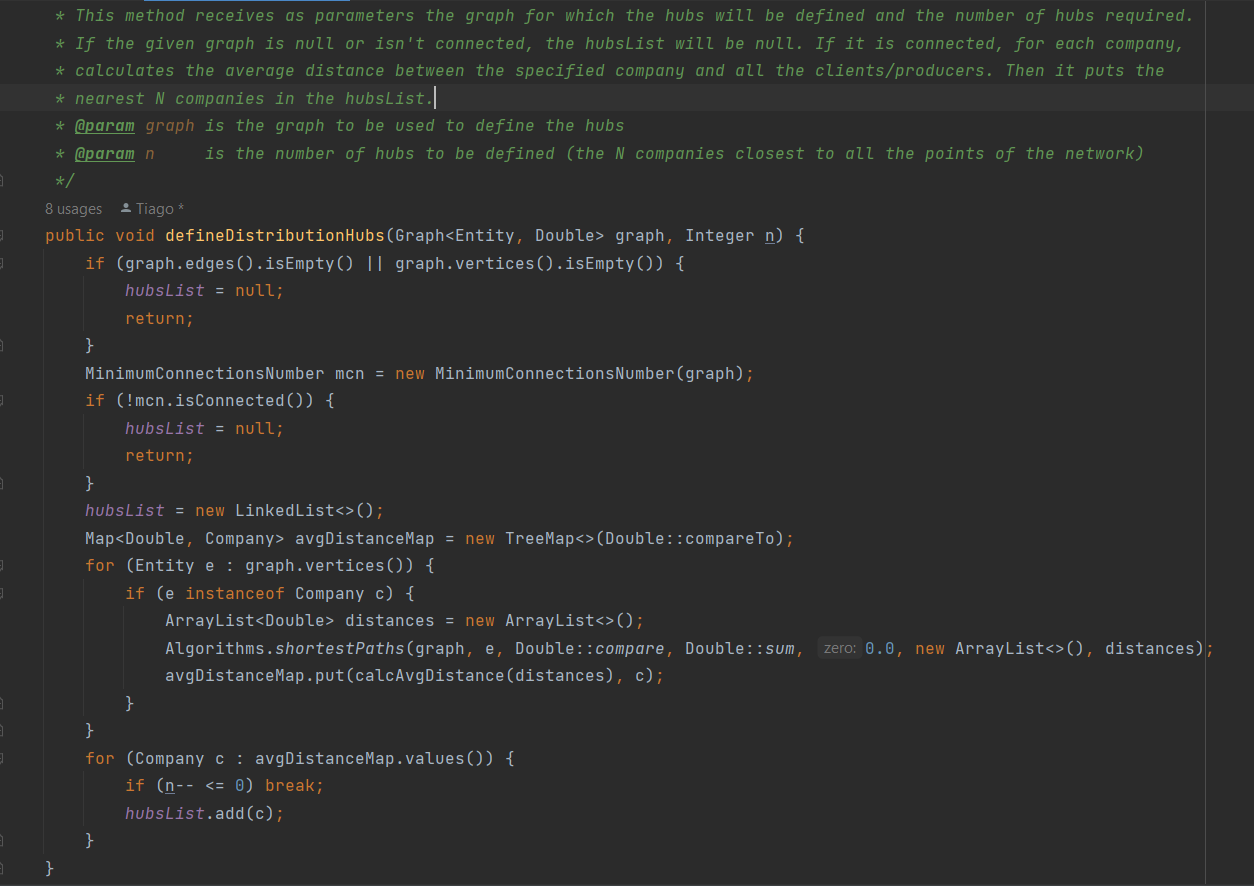


De seguida, é criado o método que irá realmente realizar a funcionalidade pretendida. O método *defineDistributionHubs* recebe como parâmetro o grafo da rede de distribuição e o número de *hubs* que são pretendidos criar.

Inicialmente é feita a validação do grafo, verificando se ele está vazio ou não. Se estiver vazio, a lista dos hubs (*hubsList*) será nula e não realizará a funcionalidade, uma vez que não existem os dados necessários para isso. Se o grafo não estiver vazio, irá verificar se este é conexo. Se não o for, *hubsList* será nula e não realizará a funcionalidade. Se for conexo, significa que está tudo em ordem para executar a funcionalidade.

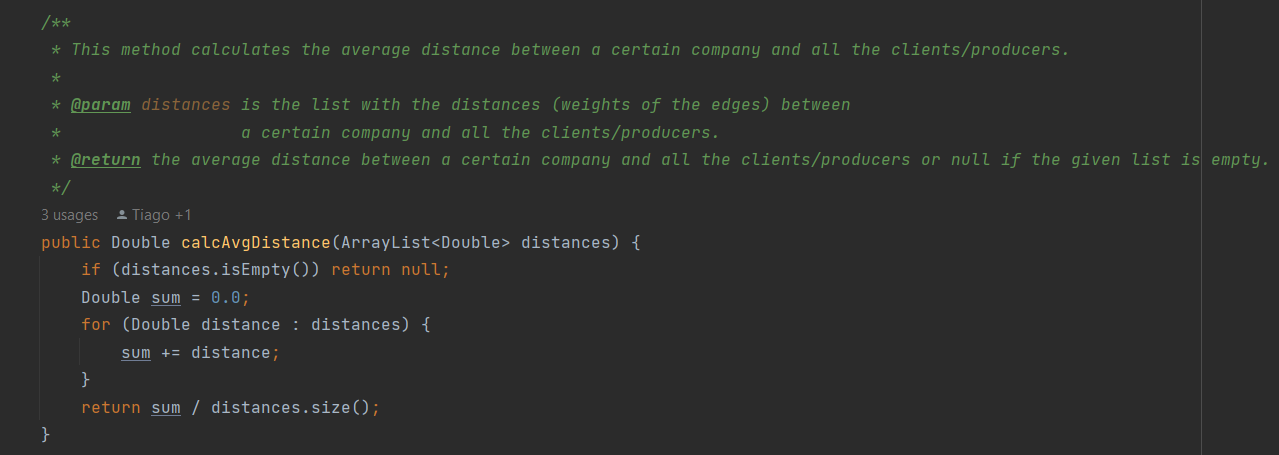
Então é criada uma nova *hubsList* e iremos buscar todos os vértices do grafo. É também criado um mapa (*avgDistanceMap*) que irá guardar as empresas juntamente com a sua proximidade, organizando-as de forma descendente por proximidade. De seguida, o ciclo *for* irá percorrer todos os vértices do grafo e, para aqueles que forem empresas, vai encontrar os menores caminhos entre a empresa e os clientes/produtores, armazenando as distâncias no *ArrayList distances.* Este *ArrayList distances* será passado como parâmetro para o método *calcAvgDistance* onde irá ser calculada a distância média da empresa até todos os clientes/produtores. Esta distância média será armazenada no mapa *avgDistanceMap,* juntamente com a empresa correspondente. Por fim, no último ciclo *for,* irão ser adicionadas as N empresas mais próximas de todos os clientes e produtores à *hubsList*.

Este método é não determinístico e tem como melhor caso O(|V|+|E|), uma vez o método *isConected* tem essa complexidade e se o grafo não for conexo não irá executar a funcionalidade. Como pior caso tem uma complexidade de O(V3), uma vez que vai executar o primeiro ciclo *for* V vezes e o método *shortestPaths* tem complexidade O(V2).



Este método auxilia a execução da funcionalidade, uma vez que vai receber uma lista de *Doubles*, no caso desta US serão distâncias, e vai calcular a média destas distâncias.

Este método é determinístico e tem complexidade O(n) uma vez que vai percorrer todos os valores da lista *distances.*



# US304

Para cada cliente (particular ou empresa) determinar o hub mais próximo.

Para esta US a classe ClosestDistributionHub ira receber no construtor uma instância de um grafo contendo os seus vértices e as ligações. O método closestHub irá retornar um mapa contendo as entidades e os hubs (empresas) que estejam mais próximas, o algoritmo vai buscar a classe DistributionHubs a lista dos hubs que foram definidos na US anterior, caso estes ainda não estejam definidos o algoritmo retorna null. Depois o algoritmo inicializa o mapa de retorno, de seguida vai percorrer cada vértice (todos os clientes) , por cada iteração dos clientes são inicializados duas variáveis, uma Company, que é a closestCompany a ser guardada e a shortestPathWeight que guarda o peso do caminho de menor custo. De seguida o algoritmo vai iterar sobre cada hub, para cada hub vai determinar o caminho de menor custo entre o cliente e e a company c, sempre que encontrar um caminho de custo inferior vai atribuir a closestCompany com a empresa mais próxima e shortestPathWeight com o custo associado a aquela distância. Uma vez o algoritmo iterar sobre os hubs todos vai colocar no mapa o valor do cliente e o hub mais próximo. Sendo que no final dos ciclos todos o algoritmo ira retornar o mapa closestHubToClient.

Text

Description automatically generated

Text

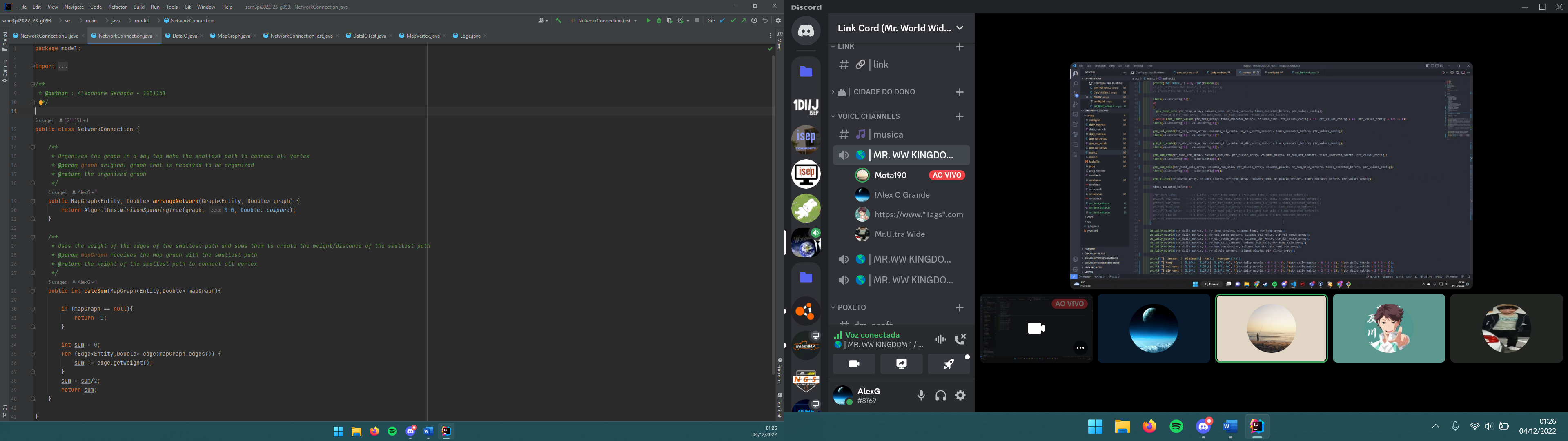
Description automatically generated

O primeiro ciclo vai ocorrer vezes, e o segundo ciclo vai ocorrer vezes, sendo que o que está dentro do ciclo vai ocorrer . Assim a complexidade total do algoritmo é de .

# US305

Determinar a rede que conecte todos os clientes e produtores agrícolas com uma distância total mínima

Para a US foi criada a classe NetworkConnection que irá organizar e mostrar a distância mínima entre os clientes e os produtores agrícolas.



O método arrangeNetwork recebe o grafo criado anteriormente que contem todas as ligações entre os clientes e os produtores agrícolas do tipo Graph<Entity, Double> e manda-o para o método minimumSpanningTree da classe Algorithms com complexidade do algoritmo: . Neste, o grafo é organizado de forma a ter o grafo com o mínimo número de ligações entre os clientes e produtores agrícolas. Retorna este grafo em formato MapGraph<Entity, Double>.

Uma imagem com texto, monitor, eletrónica, captura de ecrã

Descrição gerada automaticamente

O método calcSum recebe um grafo do tipo MapGraph<Entity, Double> e faz a soma de todas as arestas do grafo para obter a distância deste. Para a US305, este método recebe o grafo organizado anteriormente pelo método arrangeNetwork e verifica se este é nulo.

Caso este caso não seja verdadeiro, cria a variável sum e faz um loop em que percorre todas as arrestas, recebe o seu peso, e soma à variável sum.

Por fim, divide o valor do sum por 2 e retorna este valor.

# Algoritmos

## Breadth First Search

Algoritmo genérico que faz uma pesquisa em largura. Recebe um grafo e um vértice para iniciar o algoritmo.

É verificado se o grafo é nulo e se o vértice existe no grafo, para assim, começar o algoritmo propriamente dito. É criado e inicializado um vetor de *boolean*s, uma lista ligada *Queue Breadth First Search* (qbfs) e uma *Auxiliar Queue* (qaux). Enquanto a fila auxiliar não estiver vazia, todos os vértices de lá vão ser removidos (claramente utilizando a estrutura LIFO) e todos os seus adjacentes vão ser visitados.

Este algoritmo tem complexidade:

Uma imagem com texto

Descrição gerada automaticamente

## Depth First Search

Um algoritmo que deverá dar exatamente o mesmo resultado que a pesquisa em largura, porém, implementado de uma forma um pouco diferente.

Trata-se de um algoritmo que vai, recursivamente, pesquisar pelos adjacentes até todos serem visitados. A grande diferença entre estes dois algoritmos é que o BFS vai, como o nome sugere, pesquisar em largura, ou seja, adicionar todos os vértices adjacentes de um vértice à fila, entretanto o DFS vai adicionar os adjacentes do primeiro vértice apenas no final. É também um algoritmo de complexidade: .

Uma imagem com texto

Descrição gerada automaticamente

Uma imagem com texto

Descrição gerada automaticamente

## All Paths

Como o próprio nome indica, é um algoritmo que vai indicar todos os caminhos tendo em conta um vértice origem e outro destino. Retorna uma lista com todas os caminhos (em listas) de um vértice ao outro.

De uma forma recursiva, este algoritmo vai verificando se vértices adjacentes são o vértice destino ou não, caso sejam o destino, um caminho vai ser adicionado, caso não sejam o vértice destino vão, na ocasião de ainda não terem sido visitados, recursivamente, visitados e todo o algoritmo novamente, até todos os vértices serem visitados. Este algoritmo tem complexidade: *.*

Uma imagem com texto

Descrição gerada automaticamente Uma imagem com texto

Descrição gerada automaticamente

## Shortest Path Dijkstra

Algoritmo que calcula o caminho mais curto de um vértice para todos os seus vértices alcançáveis. Este algoritmo é conhecido por fazer escolhas “gulosas” o que o pode levar a não funcionar perfeitamente. Os vértices adjacentes vão ser todos processados, cada um deles vai ser comparado com a distância (caso exista) que anteriormente fora definida e a menor vai ser a escolhida (e devidamente atualizada no vetor *pathKeys*). O próximo vértice a ser processado vai ser o que tiver a menor distância para todos os que já foram visitados. Enquanto todos os vértices não forem visitados, o algoritmo vai ser executado. Este algoritmo tem complexidade:

Uma imagem com texto

Descrição gerada automaticamente Uma imagem com texto

Descrição gerada automaticamente

## Get Vertex Minimum Distance

Obtém o vértice mais perto daqueles que ainda não foram visitados. Ignora a distância 0 (que será o vértice origem) e faz uma pesquisa pela menor distância, retornando o índice associado. Complexidade do algoritmo: .

Uma imagem com texto

Descrição gerada automaticamente

## Shortest Path

Este algoritmo recorre a Dijkstra para determinar o caminho mais curto entre dois vértices, trata-se simplesmente uma aplicação de Dijkstra. É retornado o peso do caminho mais curto e é carregado na variável *shortPath* o caminho mais curto.

Como tal, a complexidade do algoritmo: .

Uma imagem com texto

Descrição gerada automaticamente

## Shortest Paths

Exatamente como o anterior, porém vai determinar todos os caminhos mais curtos a partir de um vértice (deixa de ter um vértice destino apenas). Na ocasião do grafo/vértice origem não existam, é retornado falso. Caso contrário, é retornado verdadeiro. É carregado na lista de listas *paths* os caminhos com as respetivas distâncias na lista *dists (associação por índice).*

Conclui-se assim que a complexidade deste algoritmo: .

Uma imagem com texto

Descrição gerada automaticamente

## Get Path

Este algoritmo vai “converter” o *pathKeys* numa forma clara e simples do caminho. Vai percorrer um *loop* enquanto o vértice origem não for iterado, sendo assim um algoritmo de complexidade: .

Uma imagem com texto

Descrição gerada automaticamente

## Minimum Distance Graph

Também conhecido como *Floyd-Warshall,* fecho transitivo, etc…

É um algoritmo aplicável apenas num grafo com implementação de matriz e vai calcular as distâncias entre todos os vértices (se alcançáveis) criando um grafo novo (baseado no original) com arestas a fazer a conexão entre todos os vértices. Este método tem uma complexidade: .

Uma imagem com texto

Descrição gerada automaticamente

## Graph Diameter

Este algoritmo tem como objetivo calcular o diâmetro do grafo, como tal, utiliza o *Floyd-Warshall* para obter a maior distância e os vértices que a formam. Obtém o caminho através do *Shortest Path*. Retorna o peso do *shortest longest path* e carrega *path* com o caminho. Complexidade do algoritmo:

Uma imagem com texto

Descrição gerada automaticamente

## Minimum Spanning Tree

Em português, árvore geradora mínima, tem como objetivo criar um grafo que vai ter exatamente os mesmos vértices que o original, porém as arestas vão ser apenas as escolhidas pelo algoritmo de forma que todos os vértices sejam alcançáveis da forma menos “pesada”. O algoritmo que este algoritmo foi baseado é Prim que utiliza o método “guloso”. Complexidade do algoritmo:

Uma imagem com texto

Descrição gerada automaticamente Uma imagem com texto

Descrição gerada automaticamente

## Build MST

Como o próprio nome sugere, este algoritmo constrói a árvore geradora mínima através das listas *path* e *dist*. Tem complexidade:

Uma imagem com texto

Descrição gerada automaticamente