

**Mestrado Integrado em Engenharia Informática**

**e Computação**

2.º Ano

Relatório do projeto da Unidade Curricular

**Concepção e Análise de Algoritmos**

Trabalho realizado por **Diogo Maia up201904974@up.pt,**

**Guilherme Garrido up201905407@up.pt**

**e Luís Lucas up201904624@up.pt**

Turma 2 – Grupo 2

**2020/2021**

Índice

[Descrição do problema 3](#_Toc72697368)

[Formalização do problema 4](#_Toc72697369)

[**Dados de entrada** 4](#_Toc72697370)

[**Dados de saída** 4](#_Toc72697371)

[**Restrições** 4](#_Toc72697372)

[**Funções objetivo** 5](#_Toc72697373)

[Descrição da solução 6](#_Toc72697374)

[**Algoritmos a implementar** 6](#_Toc72697375)

[**Cálculo do caminho mais curto** 7](#_Toc72697376)

[Casos de utilização 10](#_Toc72697377)

[Principais funcionalidades e cenários implementados 11](#_Toc72697378)

[Estruturas de dados utilizadas 12](#_Toc72697379)

[Algoritmos implementados 13](#_Toc72697380)

[Conclusão 14](#_Toc72697381)

[Bibliografia 15](#_Toc72697382)

Índice de figuras

[**Figura 1 - Algoritmo de Dijkstra** 7](file:///C:\Users\Utilizador\Desktop\CAL.docx#_Toc70066583)

[**Figura 2 - Código da técnica de Manhattan** 8](file:///C:\Users\Utilizador\Desktop\CAL.docx#_Toc70066584)

[**Figura 3 - Código da técnica da diagonal** 9](file:///C:\Users\Utilizador\Desktop\CAL.docx#_Toc70066585)

[**Figura 4 - Código da técnica Euclidiana** 9](file:///C:\Users\Utilizador\Desktop\CAL.docx#_Toc70066586)

# Descrição do problema

Uma empresa de distribuição de comida pretende implementar um sistema de gestão da sua frota, onde as rotas são determinadas de forma a minimizar o número de veículos utilizados e a distância percorrida por cada um.

Com sede em Vila do Conde, a PapaRica faz recolhas e posteriores entregas de refeições confecionadas em toda a Área Metropolitana do Porto. Os veículos têm como pontos de partida e de chegada a sede da empresa.

**1ª iteração: um único veículo com capacidade ilimitada**

Nesta fase, despreza-se o limite de capacidade do veículo, que irá iniciar uma única viagem com toda a carga necessária de transportar. O objetivo desta iteração é encontrar o caminho mais curto, começando na sede, passando por todos os pontos de entrega e, por fim, regressando novamente ao local de partida em Vila do Conde.

Neste caso, é importante notar que zonas inacessíveis devem ser descartadas de forma a evitar o bloqueio da viagem.

**2ª iteração: um único veículo com capacidade limitada**

Nesta iteração pretende-se encontrar o caminho mais curto respeitando a capacidade de transporte de um único veículo. Assim, este veículo poderá ser submetido a várias viagens se não for possível transportar todos os cabazes de uma só vez.

O veículo parte da sede da empresa e faz as entregas correspondentes às refeições que transporta (utiliza-se a 1ª iteração, encontrando o caminho mais curto) e retorna ao ponto de partida para recarregar a bagagem se necessário (ciclo até não haver mais refeições por entregar).

**3ª iteração: frota de veículos com capacidades diferentes**

Nesta terceira fase são considerados vários veículos com capacidades independentes. O objetivo é minimizar o número de veículos utilizados e a distância percorrida.

É importante salientar que um cabaz não pode ser dividido, ou seja, só pode ser transportado por um veículo.

# Formalização do problema

## **Dados de entrada**

**baskets** - lista de cabazes

**C** – cabaz:

* **clientName** - nome do destinatário
* **numPack** - número de embalagens
* **idDest** – identificador do vértice de destino
* **numFat** - número de fatura

**fleet** – lista de veículos

**T** – veículo:

* **maxCap** - capacidade de cada veículo
* **baskets** – lista de cabazes dentro do veículo
* **path** - lista de vértices do caminho deste veículo

**Gi = (Vi, Ei)** - grafo da área metropolitana do Porto:

* **V** – vértices (pontos de interesse em que é possível a entrega de cabazes)
  + **id** – identificador do vértice
  + **adj ∈ E** – conjunto de arestas adjacentes
* **E** – arestas (representam as ligações entre os pontos, vértices)
  + **w** - Peso da aresta (pode variar conforme o tipo de estrada)
  + **dest** [**∈**](https://pt.wiktionary.org/wiki/∈) **V** – Destino
  + **name** – identificador da aresta (por ex: rua)

**S** - vértice inicial (representa a sede da empresa em Vila do Conde)

## **Dados de saída**

fleet - lista dos veículos com caminho percorrido com um os parâmetros “baskets” e “path” atualizados.

## **Restrições**

Dados de Entrada:

* **∀ T ∈ fleet ∧ T.path.size = 0: T.baskets.size = 0** , porque cada camião que ainda não esteja em movimento não pode conter cabazes inicialmente.
* **∀ T ∈ fleet: T.maxCap > 0**, já que o camião tem de poder carregar no mínimo um cabaz.
* **∀ C ∈ baskets: ∃ V.id = C.idDest** pois o id que identifica o vértice no cabaz tem de identificar um vértice no grafo.
* **∀ C ∈ baskets: C.numPack > 0** sendo que cada cabaz tem de ter no mínimo uma embalagem.
* **S ∈ V**, logo o ponto inicial tem de ser um vértice do grafo.
* **S.adj.size > 0**, sendo que tem de se garantir a possibilidade de os camiões saírem do vértice inicial, havendo, necessariamente, pelo menos um caminho de saída da sede.
* **∀ E.w > 0**, uma vez que se se associar cada vértice a uma distância, tem de existir uma distância entre cada vértice (não pode haver dois vértices na mesma localização).

Dados de Saída:

* **∀ T ∈ fleet ∧ T.path.size ≠ 0: T.path[0] = S**, porque o início do caminho do camião é sempre a empresa.
* **∀ T ∈ fleet ∧ T.path.size ≠ 0: T.path ⊆ V**, uma vez que todos os vértices do caminho do camião têm de ser vértices do grafo.
* **∀ T ∈ fleet ∧ T.path.size ≠ 0: ∃ T.path[1] = S.adj.dest**, já que o vértice a seguir ao inicial tem de ser adjacente e esta lógica mantém-se em todos os vértices pertencentes ao “path”.
* **∀ T ∈ fleet: T.maxCap(antes) = T.maxCap(depois)**, sendo este um dado que não será alterado ao longo do algoritmo.
* **baskets.size = 0**, porque todos os cabazes foram atribuídos a um camião.
* **baskets.size(antes) = ∑ T.baskets.size**, já que todos os cabazes existentes têm de ser distribuídos pelos camiões.

## **Funções objetivo**

A solução ótima do problema passa por minimizar o número de veículos utilizados e minimizar a distância percorrida (e consequente tempo de entrega) pela frota, de forma a entregar todos os cabazes de refeições:

* min(fleet.size)
* min(Σ valores das arestas percorridas pelos veículos da frota)

# Descrição da solução

## **Algoritmos a implementar**

**Análise da conetividade de grafo**

Este é um passo importante para evitar erros de execução. O objetivo é recusar os pedidos de entrega em zonas inacessíveis da rede, por exemplo, devido a obras.

**Aceitação de pedidos de Cabazes**

Na receção do pedido de um cabaz, o sistema deve procurar pelo vértice ao qual o cabaz deve ser entregue através do id fornecido (morada). Desta forma, pensou-se que seria melhor utilizar uma adaptação do algoritmo de pesquisa em profundidade.

O algoritmo de pesquisa DFS é um algoritmo utilizado para percorrer ou procurar itens dentro de um grafo. Define-se por percorrer todos os nós filho até ao nó raiz o mais profundo possível para somente retroceder quando não for possível continuar nessa direção. É um algoritmo recursivo e assumindo que um grafo tem V vértices e A arcos, consome tempo proporcional a V+A.

**Distribuição de Cabazes pelos Camiões**

Na primeira iteração, este algoritmo consiste em passar todos os cabazes para o único veículo. No entanto, nas restantes iterações, a ideia é dividir os cabazes entre cada um dos veículos da frota (ou o mesmo veículo mas em diferentes viagens).

Primeiro, enche-se um camião. Se ainda sobrarem cabazes e houver mais veículos, enche-se os outros de seguida. Caso a capacidade da frota seja inferior ao número de cabazes a entregar, os veículos que retornem à sede estarão encarregues de realizar mais viagens até que todos os cabazes estejam entregues.

**Identificação do pedido mais perto do camião**

Pretende-se para cada camião utilizado procurar dentro dos seus pedidos o cabaz com o local de entrega mais perto do vértice atual do camião. Utilizando um algoritmo de pesquisa de vértices.

**Determinar o percurso mais curto entre os dois vértices e fazer o camião percorrê-lo**

Após determinar a entrega mais próxima do local atual do veículo, procede-se para a sua concretização, retirando o cabaz entregue da lista de cabazes do camião e adicionando este vértice à lista de vértices do camião em questão.

Estes dois últimos passos são repetidos até todos os cabazes do camião serem entregues. Quando o veículo não tiver mais entregas por fazer deve retornar à sede e ficará disponível para novas viagens.

## **Cálculo do caminho mais curto**

**Algoritmo de Dijkstra**

O algoritmo de Dijkstra é um algoritmo ganancioso que resolve o problema do caminho mais curto partindo de um início único num gráfico pesado dirigido, para o caso em que todos os pesos das arestas não são negativos, com um custo computacional:

O((V|+|E|) \* log |V|).

O algoritmo começa por fazer um pré-processamento a todos os vértices do grafo marcando a sua distância como infinita e o vértice antecessor a nulo.

De seguida, altera a distância do vértice inicial para zero e insere-o numa fila de prioridade de mínimos de forma a priorizar sempre o vértice com a menor distância, tornando-se assim um algoritmo ganancioso, já que procura minimizar sempre uma solução imediata, de forma a posteriormente, minimizar uma solução global.

O terceiro passo do algoritmo passa por entrar num ciclo enquanto houver vértices na fila de prioridade e para cada adjacência desses vértices, se a distância, no momento, guardada for maior que o peso da aresta somada com a distância do vértice, então essa distância da adjacência deve ser atualizada e, consequentemente, a variável que indica o vértice antecessor.

Uma imagem com texto

Descrição gerada automaticamentePor fim, percorre-se o caminho inverso, ou seja, do vértice destino até ao vértice inicial, de forma a encontrar o caminho mínimo entre os dois partindo da origem (inicial) até ao final (destino).

**Figura 1 - Algoritmo de Dijkstra**

**Figura 1: Algoritmo de Dijkstra**

**Algoritmo A\***

O algoritmo A\* é um algoritmo que procura o caminho mais curto entre dois vértices previamente dados, sendo um o inicial e outro o final. Pode-se considerar que este algoritmo é a combinação de aproximações heurísticas do algoritmo de pesquisa em largura e da formalidade do algoritmo de Dijkstra (difere deste último apenas no cálculo do peso das arestas). Este algoritmo nem sempre garante o caminho ótimo.

O cálculo do peso das arestas segue a função:

f(v) = h(v) + g(v)

onde h(v) é a função heurística (que é nula no algoritmo de Dijkstra).

Algoritmo

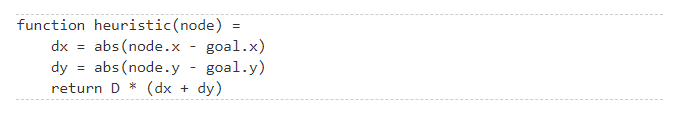
Sejam:  
-> Q = conjunto de vértices a serem pesquisados

->S = o vértice inicial

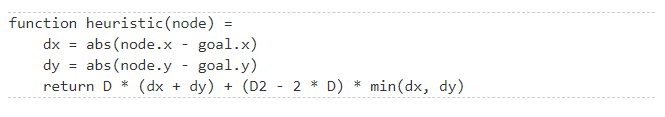
1. Inicializar Q com S como única entrada.
2. Se Q está vazio, acabar o processo. Se não, escolher o melhor elemento de Q
3. Se o vértice(n) é o objetivo, retorná-lo;
4. Senão remover n de Q;
5. Encontrar os descendentes de **n** que não estão em visitados e criar todas as extensões de **n** para cada descendente;
6. Adicionar os caminhos expandidos a **Q** e voltar ao passo **2**;

Existem várias técnicas que podem ser usadas para escolher o melhor vértice de Q. O que o algoritmo faz é utilizar uma destas técnicas para modificar os pesos das arestas ligadas a esse vértice de acordo com a distância deste ao vértice final, sendo que as arestas que remetem para vértices mais perto do vértice final serão beneficiados com uma redução do peso e vice-versa. No final, para não haver perdas de informação, o cálculo da distância total sofre modificações de modo a corresponder à distância atual.

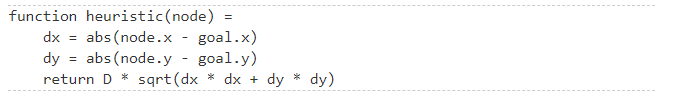
Algumas destas técnicas são:

* a técnica de distância de Manhattan (Mais eficiente para movimentações em 4 direções):

**Figura 2 - Código da técnica de Manhattan**

* a técnica da distância diagonal (mais eficiente para movimentações em 8 direções):

**Figura 3 - Código da técnica da diagonal**

* a técnica da distância Euclidiana (mais eficiente para movimentações em qualquer direção):

**Figura 4 - Código da técnica Euclidiana**

No contexto do problema, a técnica mais adequada é a Euclidiana, pois remete apenas para uma aproximação entre dois pontos, não importando a direção.

# Casos de utilização

Pretende-se que a aplicação seja composta por uma interface intuitiva que permita ao utilizador editar (criar, remover, alterar):

* Cabazes de refeições
* Veículos da frota da empresa
* Local de uma entrega

Além disso, na interface deverá ser possível visualizar um mapa com a região propícia a entregas (zona metropolitana do Porto).

Para a criação de dados, será possível inserir os dados manualmente ou importar um ficheiro já com dados de cabazes, veículos e encomendas.

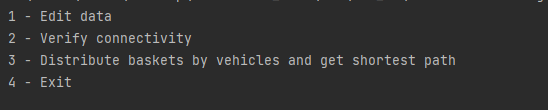
Após os dados necessários estarem inseridos, o programa permite:

* distribuir automaticamente os cabazes pelas carrinhas
* verificar a acessibilidade a e de todos os pontos de entrega
* efetuar o cálculo do caminho ótimo, apresentando a melhor solução para as viagens da frota da empresa.

# Principais funcionalidades e cenários implementados

Ao iniciar o nosso programa, o utilizador terá acesso a uma pequena interface onde poderá decidir o que pretende fazer.

Esta interface contém três grandes módulos, sem contarmos com a opção de sair.



**Figura 5 – Menu do programa**

O primeiro módulo remete para a inserção e remoção da informação sobre o tema do trabalho, neste caso, a companhia, os seus veículos e os cabazes que precisam de ser entregues.

O segundo módulo permite ao utilizador informar-se sobre a conetividade do grafo que representa a cidade do Porto. Poderá ver como um pequeno mapa com o grafo lá representado e em destaque estarão os nós que pertencem a componentes de forte conexão com uma cor diferente. Por fim, serão mostradas algumas estatísticas sobre o assunto como a quantidade total de vértices e a percentagem de vértices que pertencem a pelo menos um componente de forte conexão.

O terceiro módulo permite ao utilizador, com os dados atribuídos no primeiro grande módulo, observar os caminhos atribuídos a cada veículo. Este módulo é responsável por atribuir os cabazes aos veículos e calcular o menor caminho possível, de acordo com a distância, para entregar os cabazes que lhe foram atribuídos.

Quanto aos cenários implementados, os cenários previamente explícitos foram utilizados como forma de permitir uma rápida e fácil progressão ao longo do desenvolvimento do projeto de modo que apenas faríamos o posterior após finalizar o atual. Desse modo, o cenário atualmente implementado é a **3ª iteração -frota de veículos com capacidades diferentes**.

# Estruturas de dados utilizadas

Ao longo do trabalho, nós tentamos organizar o código de forma a utilizar da forma mais eficiente o possível certas estruturas de dados.

Algumas das estruturas de dados utilizadas e as suas respetivas utilizações são:

* Array: Ao longo do trabalho, um array foi utilizado para implementar uma matriz nxn sendo n o número de elementos do vetor usado para povoá-la. Esta matriz é a matriz de custos do algoritmo de Clarke e Wight e esta estrutura foi utilizada de forma a podermos aceder facilmente a qualquer posição do array, sendo uma das melhores estruturas a nível de eficiência quando se trata de aceder a dados
* Vetor: Muitos vetores foram utilizados ao longo do projeto de modo a guardar várias informações dos objetos utilizados por este. Por exemplo, este representa o conjunto de veículos da companhia ou a sequência de paragens de um veículo durante o transporte, etc.
* Conjunto: Os conjuntos foram utilizados quando foi atestada a conetividade do grafo. De modo a guardar os componentes de forte conexão, o conjunto foi uma boa escolha, já que esta estrutura de dados possui ordem, logo impedia a junção de componentes de forte conexão de igual conteúdo a algum que já se encontrava no conjunto.  
  Como exemplo, se tivéssemos no conjunto de SCC’s {1,3,2}, este impediria que o conjunto {1,2,3} se juntasse sendo que este tem os mesmos elementos.
* Pilha: A pilha foi implementada durante o algoritmo de Tarjan para permitir resgatar os vértices por ordem oposta à ordem de colocação (FILO)

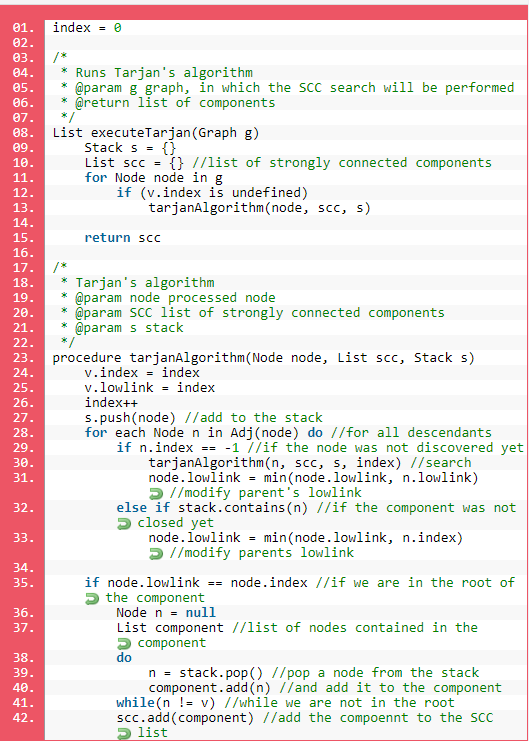
# Algoritmos implementados

Nesta secção iremos realçar os algoritmos que acabaram por ser implementados ao longo do projeto.

# **Algoritmo de Tarjan**

Este algoritmo procura componentes de forte conexão num grafo dirigido, sendo um componente de forte conexão um conjunto máximo de vértices em que existe pelo menos um caminho orientado entre cada dois vértices.

Este algoritmo é baseado na pesquisa em profundidade (DFS). Os vértices são indexados à medida que são percorridos pelo DFS. Ao retomar da recursão do DFS, cada vértice V recebe um vértice L como representante. L é um vértice com o menor índice que pode alcançar V. Os nós com o mesmo representante atribuído estão localizados no mesmo componente fortemente conectado.

Como este algoritmo é muito parecido ao DFS, cada nó é visitado uma vez e apenas uma vez. Para cada nó, realizamos alguma quantidade constante de trabalho e iteramos sobre sua lista de adjunto. Assim, a complexidade temporal é O(|V|+|E|), sendo V o conjunto de vértices e E o conjunto de arestas.

No máximo, a profundidade de recursão e o tamanho da pilha podem ser N nós, sendo N o número total de vértices. Portanto, a complexidade espacial é O(|V|).

**Figura 6 – Pseudocódigo do Algoritmo de Tarjan (obtido de https://www.programming-algorithms.net/article/44220/Tarjan's-algorithm)**

# **Algortimo de Clarke e Wright**

# A abordagem mais conhecida para o problema de VRP (Vehicle Routing Problem) é o algoritmo de “economia” de Clarke e Wright. Considerando um depósito D e n pontos de demanda. Supondo que a solução inicial para o VRP é usar n veículos e despachar um veículo para cada um dos pontos de demanda. O comprimento por demanda é obviamente .

# Se utilizarmos um veículo para atender dois pontos diferentes, digamos i e j, em uma única viagem, teríamos algo como: .

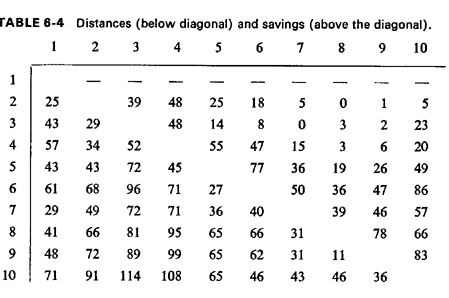
# A distância entre estes dois é reduzida em: . Esta redução é tratada como “economia” resultante da combinação dos pontos i e j.

Pode se descrever o algoritmo da seguinte forma:  
->Para simplificar, vai ser usado em alguns momentos um exemplo com os pontos {0,1,2,3}

* 1º passo: Caso não detenha um, calcular uma matrix nxn de distâncias entre dois pontos sendo que o 0 será o depósito.
* 2º passo: Com a matriz de custos já preparada, calcula-se a “economia” para cada conjunto de dois pontos, não incluindo o depósito (neste caso, seriam [{1,2},{1,3},{2,3}]). No nosso caso, usamos uma struct para guardar estes dados.
* 3º passo: Ordenar o conjunto de “economias” por ordem decrescente de custo.
* 4º passo: Ao longo dos veículos, percorrer a lista de “economias” e a cada economia verificar se um dos dois pontos já se encontra ou não no caminho do veículo.

Caso se encontrem os dois, o veículo ignora essa economia e elimina-a da lista  
Caso se encontre apenas um deles, adiciona-se ao caminho do veículo o novo ponto de demanda, de modo a estar ou antes ou depois do ponto encontrado.  
Caso não se encontre nenhum, adicionar ambos à lista caso esta esteja vazia, senão não adiciona, pois, neste caso, é suposto ele fazer um caminho sem voltar ao depósito.

Verificar antes de adicionar que está dentro das regulações da VRP, neste caso, não ultrapassa o peso do veículo.

* 5º passo: se sobraram pontos sem rota, terão de ser adicionadas rotas depósito-rota-depósito

**Figura 7- Exemplo de matriz de custos (pode ser apenas triângulo superior)**

//Falta análise

# Conclusão

Conclui-se que esta primeira parte de especificação e formalização do projeto constitui uma base sólida para a realização da implementação numa segunda fase.

O principal problema que se espera encontrar na parte do desenvolvimento do programa tem a ver com a “tradução” do mapa, uma vez que podem surgir diversas contrariedades neste ponto como, por exemplo, a equivalência entre as ruas (arestas do grafo) e o tempo que se demora a percorrer as mesmas (serão sempre valores relativos).

A contribuição para o trabalho foi igual por parte de todos membros na pesquisa e realização deste relatório.

# Bibliografia

* Slides das aulas teóricas da UC de Conceção de Análise de Algoritmos
* Cormen T. H., Leiserson C. E., Rivest R. L., Stein C. (2009). *Introduction to Algorithms* (Chapter 24).
* Weiss, Mark Allen (2007). *Data structures and algorithm analysis in C++*.
* Imagens ilustrativas dos algoritmos em grafo: Caminho mais curto, <https://qiao.github.io/PathFinding.js/visual/>
* A\* PathFinding para iniciantes por Patrick Lester ,https://web.archive.org/web/20061222182121/http://policyalmanac.org/games/aStarTutorial\_port.htm
* Heuristics from Amits thoughts on Pathfinding, <http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html#a-stars-use-of-the-heuristic>
* Program Algorithms, Tarjan’s-algorithm,   
  <https://www.programming-algorithms.net/article/44220/Tarjan's-algorithm>
* VRP Depósito Único, https://web.mit.edu/urban\_or\_book/www/book/chapter6/6.4.12.html