Faculdade de Engenharia da Universidade do Porto

Projeto de Concepção e Análise de Algoritmos:

EcoPonto: recolha de lixo seletiva (tema 4) - Parte 1

27 de abril de 2016

Grupo B, Turma 6:

Gonçalo da Mota Laranjeira Torres Leão	up201406036@fe.up.pt
Francisco Tomé Neto Queirós	up201404326@fe.up.pt
Eduardo Miguel Bastos Leite	gei12068@fe.up.pt

Índice

- Descrição sucinta do problema
  - 1ª iteração: recolha não-seletiva com camião de capacidade ilimitada
  - 2ª iteração: recolha seletiva com camião de capacidade ilimitada
  - 3ª iteração: recolha seletiva com camião de capacidade limitada
- Formalização do problema
  - Dados de entrada
  - Dados de saída
  - Restrições
    - Sobre os dados de entrada
    - Sobre os dados de saída
  - Funções objetivo
- Descrição da solução
  - Estruturas de dados
    - Representação de um grafo genérico
    - Representação de um grafo concreto ao problema
    - Gestor de resíduos
  - Algoritmos implementados
    - Análise da conetividade
    - Cálculo das menores distâncias
    - Geração do caminho entre dois vértices
    - Geração do caminho desde um vértice até a um vértice de chegada, passando por pontos de interesse
    - Recolha de um tipo de lixo por camião
- Casos de utilização
- Principais dificuldades encontradas no desenvolvimento do trabalho
- Esforço dedicado por cada elemento do grupo
  - Desenvolvimento da interface
  - Desenvolvimento do código
  - Desenvolvimento do relatório
- Bibliografia

1. Descrição sucinta do problema

Um centro de tratamento de lixo planeia implementar um sistema de recolha inteligente, onde as rotas dos camiões são determinadas de forma a minimizar o percurso efetuado por cada veículo.

Espalhados pela cidade, estão vários contentores de lixo com diferentes níveis de acumulação de resíduos.

Este problema pode ser decomposto em três iterações.

1.1. 1ª iteração: recolha não-seletiva com camião de capacidade ilimitada

Numa primeira fase, despreza-se o limite de capacidade de um camião. Assim, o objetivo trata-se simplesmente de determinar a rota mais curta que começa na central de recolha, passe por todos os contentores de lixo com um nível de acumulação de resíduos acima de um limite e que termine num dos centros de tratamento de lixo, onde este é depositado. É necessário apenas um camião para efetuar toda a recolha de lixo.

Aproveita-se para esclarecer que só é interessante recolher o lixo de um caixote a partir de uma certa taxa de ocupação pois há um custo associado à recolha.

É importante de notar que a recolha só pode ser efetuada se existirem caminhos que liguem todos os pontos de interesse (central, caixotes de lixo e pelo menos uma estação de tratamento de lixo), dois a dois, e em ambos os sentidos. Por outras palavras, todos os pontos de interesse devem fazer parte do mesmo componente fortemente conexo do grafo. Esta necessidade advém do facto que, da central, o camião deve conseguir alcançar todos os caixotes cujo lixo pretende recolher e uma central de tratamento, e, depois, deve poder conseguir regressar à central (o caminho de regresso não é do nosso interesse, mas temos de garantir que existe). Assim, torna-se imperativo efetuar uma análise da conetividade do grafo.

Certas vias de comunicação podem não poder ser usadas pelo camião para o seu percurso por diversos motivos (por exemplo, devido a obras ou devido às ruas serem demasiado estreitas para o camião). Assim, será necessário desprezar certas arestas durante o processamento do grafo.

1.2. 2ª iteração: recolha seletiva com camião de capacidade ilimitada

Numa segunda fase, passa-se a diferenciar os contentores segundo o tipo de resíduo em ecopontos: papel (contentores azuis), vidro (contentores verdes), plástico (contentores amarelos) e lixo comum (contentores pretos).

Cada camião só pode recolher um certo tipo de resíduo, pelo que serão necessários, no mínimo, quatro camiões para efetuar a recolha de lixo, e, consequentemente, será necessário definir pelo menos quatro rotas de recolha (em vez de uma só).

1.3. 3ª iteração: recolha seletiva com camião de capacidade limitada

Numa terceira fase, passa-se a considerar que os camiões têm capacidade limitada, ou seja, só podem recolher até uma certa quantidade de lixo. Este limite pode ser devido ao peso do lixo ou ao volume ocupado por este, por exemplo. Assim, se a quantidade total de lixo de um certo tipo ultrapassar a quantidade máxima suportada por um camião, passa a ser necessário, obrigatoriamente, mais do que um camião para efetuar a recolha desse tipo de lixo.

Como consequência desta nova restrição do problema, a solução ótima passa pela minimização de dois parâmetros: a distância total percorrida pelos camiões e o número de camiões necessários para efetuar toda a recolha de lixo. Destes dois parâmetros, dar-se-á prioridade ao segundo critério, ou seja, ao de minimizar o número de camiões usados.

Convém salientar que não é possível efetuar a recolha de parte dos conteúdos de um caixote de lixo: qualquer recolha remove todo o lixo de um contentor. Logo, se um camião já não tiver capacidade remanescente suficiente para recolher um contentor, será necessário outro camião para recolher o lixo.

2. Formalização do problema

2.1. Dados de entrada

type - tipo de lixo que se pretende recolher (papel, vidro, plástico…)
R_{max} - taxa de ocupação máxima de um caixote de lixo, a partir do qual é interessante recolher o seu lixo.
Ci - sequência de camiões da central, sendo Ci(i) o seu i-ésimo elemento. Cada um é caracterizado por:

- cap - capacidade total do camião (na 1ª e 2ª iteração do problema, cap = ∞).

Gi = (Vi, Ei) - grafo dirigido pesado, composto por:

- V - vértices (que representam pontos da cidade) com:
    - W - quantidade de lixo no vértice (0 se o vértice não tem um caixote) (W vem de *waste*)
    - W_{max} - quantidade máxima de lixo do vértice (0 se o vértice não tem um caixote)
    - $R = W/W_{max}$ - taxa de ocupação de lixo do vértice (0 se o vértice não tem um caixote)
    - Adj ⊆ E - conjunto de arestas que partem do vértice
- E - arestas (que representam vias de comunicação) com:
    - w - peso da aresta (representa a distância entre os dois vértices que a delimitam)
    - ID - identificador único de uma aresta
    - dest ∈ Vi - vértice de destino

S ∈ Vi - vértice inicial (central de camiões)
T ⊆ Vi - vértices finais (estações de tratamento do lixo)

2.2. Dados de saída

Gf = (Vf,Ef) grafo dirigido pesado, tendo Vf e Ef os mesmos atributos que Vi e Ei.
Cf- sequência ordenada de todos os camiões usados, sendo Cf(i) o seu i-ésimo elemento.
Cada um tem os seguintes valores:

- cap - capacidade do camião usado
- $P = \{e \in Ei \mid 1 \leq j \leq |P|\}$ - sequência ordenada (com repetidos) de arestas a visitar, sendo e_j o seu j-ésimo elemento.

2.3. Restrições

A. Sobre os dados de entrada

- ∀ i ∈ [1 ; |Cf|], cap(Cf[i]) > 0, dado que uma capacidade representa um peso ou volume.
- Pelos mesmos motivos: ∀ v ∈ Vi, $W \geq 0$, $W_{max} \geq 0$.

- $0 < R_{max} \leq 1$, pois R_{max} representa uma taxa de ocupação (é de notar que o intervalo de R_{max} é aberto no seu limite inferior senão estar-se-ia a recolher o lixo de contentores vazios).
- Pelos mesmos motivos: $\forall v \in Vi$, $0 \leq R \leq 1$ (aqui R pode ser 0, se o vértice não corresponder ao um caixote desse tipo de lixo, ou o caixote estiver vazio).
- $\forall e \in Ei$, $w > 0$, dado que o peso de uma aresta representa uma distância entre pontos de um mapa.
- Seja o conjunto $L = \{v \in Vi \mid v = S \text{ ou } v \in T \text{ ou } W(v) > 0\}$. É preciso que todos os elementos de L façam parte do mesmo componente fortemente conexo de um grafo. Segundo Skiena *(Skiena 2008)*, um componente fortemente conexo (CFC) é uma partição do conjunto de vértices V de um grafo G = (Vi,Ei), tal que existem caminhos dirigidos que liguem todos os pares de vértices distintos da partição. Assim, se um vértice pertencesse a dois CFC's, então seria possível transitar de qualquer vértice de um dos CFC's para qualquer vértice do outro CFC usando o vértice que têm um comum, pelo que todos os vértices de ambos os CFC formam um único CFC. É por esta razão que os CFC's particionam o grafo em conjuntos disjuntos de vértices.
- $\forall e \in Ei$, e deve ser utilizável pelo camião. Senão, não é incluída no grafo Gi.

B. <u>Sobre os dados de saída</u>

No grafo Gf:

- $\forall vf \in Vf$, $\exists vi \in Vi$ tal que vi e vf têm os mesmos valores para todos os atributos, exceto eventualmente W e R que podem ser 0 (se o lixo tiver sido recolhido).
- $\forall ef \in Ef$, $\exists ei \in Ei$ tal que ei e ef têm os mesmos valores para todos os atributos.

Na sequência Cf:

- $\forall c \in Cf$: $cap \geq \sum_{v \in V} W(v)*u$, onde

  u = 0 se o lixo já foi recolhido (pelo camião c, ou por outro camião que venha antes da sequência ordenada Cf) ou se não relevante ...

  u = 1 caso contrário.
- $|Cf| \leq |Ci|$, pois não se podem usar mais camiões do que os que estão disponíveis.
- Se $|Cf| < |Ci|$, então $\forall v \in Vf$, $W(v) = 0$ logo $R(v) = 0$, pois, se nem todos os camiões foram usados, foi porque foi possível recolher todo o lixo. Caso $|Cf| = |Ci|$, então pode não ter sido possível recolher todo o lixo.

Na sequência P:

- Seja e_1 o primeiro elemento de P. É preciso que $e_1 \in Adj(S)$, pois o camião parte da central.
- Seja $e_{|P|}$ o último elemento de P. É preciso que $dest(e_{|P|}) \in T$, pois o camião termina o percurso numa estação de tratamento do lixo.

2.4. Funções objetivo

A solução ótima do problema passa por minimizar o número de camiões usados e a distância total percorrida por todos os camiões, de forma a que todo o lixo seja recolhido (acima de uma certa taxa de ocupação dos contentores). Logo, a solução ótima passa pela minimização das duas respetivas funções:

$f = |C|$

$g = \sum_{c \in C} [\sum_{e \in P} w(e)]$

Tal como referido na descrição do problema, irá ser privilegiada a minimização da função f sobre a da função g.

3. Descrição da solução

3.1. Estruturas de dados

A. Representação de um grafo genérico

Para a estrutura de dados do grafo (definida no ficheiro "Graph.h"), foram usados dois *templates*, V e E, que representam, respetivamente, a "informação não trivial" de vértices e de arestas do grafo. Entende-se por "informação não trivial" toda a informação que não é inerente à estrutura de dados do grafo, como por exemplo o nome de um vértice (se este representar um ponto de uma cidade, por exemplo) ou de uma aresta (se este representar o nome de uma rua, por exemplo), ou um inteiro positivo que serve para identificar uma destas duas estruturas de dados.

Um vértice genérico (definido no ficheiro "VertexEdgePath.h") contém:

- info: uma variável com informação "não-trivial" (do tipo genérico V). É necessário que o valor de V seja distinto para vértices distintos do grafo, para que se possa identificar, de forma única todos os vértices do grafo.
- adj: um *unordered_map* (estrutura de dados da biblioteca *Standard Template Library*, *STL*, para C++) de apontadores para arestas usando como chave o valor da informação info, do tipo E, da aresta (que serve para a identificar de forma única, como veremos na descrição do conteúdo de um vértice genérico).

Uma aresta genérica (também definida em "VertexEdgePath.h") contém:

- info: uma variável com a informação "não-trivial" (do tipo genérico E). É necessário que o valor de E seja distinto para arestas distintas do grafo, para que se possa identificar, de forma única todas as arestas do grafo.
- dest: um apontador para o vértice de destino.
- weight: o peso da aresta (um *double*, que deverá ser positivo, pois representa uma distância).

Um grafo genérico, em si, contém um *unordered_map*, chamado vertexSet, de apontadores para todos os vértices do grafo. Esta estrutura de dados usa como chave o valor da informação info, do tipo V, do vértice.

A vantagem de usar *unordered_maps* para guardar os vértices e arestas do grafo traz a vantagem da pesquisa, inserção e remoção de elementos de efetuar em tempo constante, O(1), no caso médio, contrariamente a estruturas de dados mais convencionais como vetores, para o qual as operações de pesquisa e remoção de um elemento se realizam em tempo linear, O(n). No pior caso, se a estrutura estiver muito preenchida, estas operações podem se realizar com complexidade temporal O(n). Contudo tratam- se de casos pontuais, pois quando a estrutura fica completamente preenchida, é feito um *re-hash* para alocar mais

espaço. Após efetuar um *re-hash*, estas operações tendem a decorrer um tempo constante, em média.
Por se usar *unordered_maps*, passa a ser necessário identificar os vértices e arestas por uma chave. Ora, isto não constitui um problema pois seria necessário, de qualquer forma, ser possível aceder a qualquer elemento de um grafo através de uma chave, para que estes possam ser manipulados (por exemplo, para alterar os seus atributos, como o peso de uma aresta).

Na realidade, foram usados mais atributos, tanto para o grafo, como para vértices e arestas. Contudo, como se tratam de atributos específicos para certos algoritmos, abordar-lhes-emos oportunamente quando falarmos dos algoritmos empregues para a resolução do problema.

B. Representação de um grafo concreto ao problema

No caso concreto do nosso problema, os *templates* V e E foram implementados usando as classes VertexInfo e EdgeInfo, respetivamente.

A classe VertexInfo contém os seguintes atributos:

- nodeId: inteiro positivo que identifica, de forma única, um vértice.
- lat: latitude do vértice em radianos.
- lon: longitude do vértice em radianos.
- paper: quantidade de papel no vértice. Para ser o mais genérico possível, assume-se que cada vértice de um caixote de lixo de cada tipo. Se este não for usado, o valores associados ao lixo são postos a 0.
- glass: quantidade de vidro no vértice.
- plastic: quantidade de plástico no vértice.
- other: quantidade de lixo comum no vértice.
- maxPaper: quantidade máxima de papel no vértice. O valor de paper não pode exceder o valor este atributo.
- maxGlass: quantidade máxima de vidro no vértice.
- maxPlastic: quantidade máxima de plástico no vértice.
- maxOther: quantidade máxima de lixo comum no vértice.
- name: nome associado ao vértice. Este atributo pode corresponder, por exemplo, ao nome de uma praça ou de um ponto de interesse, na interseção entre arestas.

A classe EdgeInfo contém os seguintes atributos:

- ID: inteiro positivo que identifica, de forma única, um aresta.
- name: nome associado à aresta. Este atributo pode corresponder, por exemplo, ao nome de uma rua ou estrada.
- active: booleano que indica se uma aresta é utilizável pelo camião. As arestas inativas são mantidas no grafo, visto que a sua inatividade pode ser temporário (por exemplo, devido a obras). Assim, não é necessário remover e reinserir uma aresta sempre que se torne inativa, o que poupa tempo de execução.

C. Gestor de resíduos

Para uma gestão mais eficiente dos vértices com lixo e das centrais de tratamento de lixo, criou-se uma nova estrutura de dados chamada WasteManager, que encapsula o classe do grafo (esta classe foi implementada em WasteManager.h e WasteManager.cpp).

Esta classe tem como principais atributos:

- graph: grafo de base, com valores de info para vértices e arestas do tipo VertexInfo e EdgeInfo, respetivamente.
- criticalPaperRatio/criticalGlassRatio/criticalPlasticRatio/criticalOtherRatio: taxa de acumulação crítica para um certo tipo de lixo (papel, vidro, plástico e lixo comum, respetivamente), a partir da qual se deve recolher o lixo de um caixote.
- start: apontador para o vértice que corresponde à central.
- paper/glass/plastic/other: listas de vértices com lixo de um certo tipo (papel, vidro, plástico e lixo comum, respetivamente) que ultrapasse o limite crítico, ordenada pela quantidade de lixo, em ordem decrescente.
- sortedPaper/sortedGlass/sortedPlastic/sortedOther: booleanos que indicam se as respetivas listas de vértice com lixo estão ordenadas.
- ends: lista de vértices que correspondem a centros de tratamento de lixo.

Optou-se por guardar apontadores para a central para os centros de tratamento em vez de manter um atributo nesses vértice a indicar que desempenham estes papéis para poupar, de forma significativa, tempo de execução (a custo de espaço em memória, que mesmo assim é reduzido pois tratam-se de listas de apontadores e não dos objetos diretamente).

Usaram-se listas em vez de vetores pois, como veremos mais à frente, quando estas listas forem manipuladas, vai ser necessário iterar sobre todo o contentor, pelo que a eficiência de usar um vetor ou uma lista é a mesma. A grande vantagem de se usar listas é que permitem remoções de elementos em tempo O(1), contrariamente a vetores onde o custo é O(N), sendo N o comprimento do vetor.

A utilidade dos quatro booleanos “sorted” serve para indicar quais das listas de vértices não estão ordenadas, para evitar ter de se reordenar as listas “em caso de dúvida” quando for preciso as utilizar. Assim, sempre que for acrescentado um vértice a alguma das listas de vértices de caixotes a recolher (por se ter adicionado lixo de um certo tipo a um vértice), o booleano correspondente é posto a *false*. Quando a lista é reordenada, o booleano é posto a *true*. Na secção sobre o algoritmo de recolha de um tipo de lixo por camião, explica-se porque razão se pretende manter as listas ordenadas (a lista de centros de tratamento não precisa de estar ordenada).

3.2. Algoritmos implementados

É de notar que, como é óbvio, as arestas inativas são ignoradas pelo grafo. Ao longo dos algoritmos que iremos descrever de seguida, quando se refere a arestas, refere-se unicamente a arestas ativas, que podem ser usadas pelos camiões.

A. Análise da conetividade

Para determinar os componentes fortemente conexos (CFC's) do grafo, foi implementado o algoritmo proposto pelos docentes na aula teórica sobre conetividade (Rossetti and Rocha 2015/2016).

Em primeiro lugar, é feita uma pesquisa em profundidade no grafo, onde os vértices são numerados em pós-ordem, ou seja, um vértice tem sempre maior índice que todos os seus descendentes na floresta de expansão resultante. Caso hajam várias árvores na floresta em expansão, o índice de qualquer vértice da (i+1)-ésima árvore visitada é maior que o índice do qualquer vértice da i-ésima árvore. Para poupar tempo na consulta dos índices, em vez de se guardar em cada vértice o seu índice, é criado um vetor ordenado de apontadores para vértices na classe Graph, chamado invOrder, onde o primeiro elemento corresponde ao vértice de menor índice, e o último, ao de maior índice.

A complexidade deste primeiro passo é O(|V| + |E|), sendo V o conjunto de vértices e E o conjunto de arestas do grafo.

Em segundo lugar, invertem-se todos os vértices. O algoritmo é descrito de seguida em pseudocódigo:

```
1. for each(v: V)
       a. v.removeInvAdj();
2. for each(v: V)
       a. for each(e: adj(v))
              i.  x = e.getDest(v)
              ii. if(!x.getInvAdj.find(v) )
                      1. x.addToInvAdj(v)
```

Para poupar espaço (e também tempo), em vez de se criar um grafo auxiliar, ou de alterar a lista de arestas a sair de cada vértice, é criado um *unordered_map* de apontadores para vértices (usando como chave a informação do vértice de destino), chamado invAdj, em cada vértice, que identifica as adjacências de cada vértice no grafo Gr, resultante da inversão de todas as arestas. É de notar que, caso hajam várias arestas com o mesmo sentido a conectar dois vértices (eventualmente com diferentes pesos), só é necessário acrescentar a invAdj uma única aresta, visto que o que nos interessa é a existência, ou não, de arestas entre cada

par de vértices, e não o peso destes. Aliás, ter mais que uma aresta a ligar dois vértices não só seria redundante, como também pioraria o desempenho temporal do algoritmo, que se for usar esta estrutura de dados. Assim, faz sentido usar um *unordered_map* pois permite identificar em tempo linear se já existe uma aresta a ligar dois pares de vértices em tempo constante.

A complexidade deste segundo passo é O(|V|+|E|). O passo 1 do algoritmo tem complexidade O(|V|+|E|), pois a cada aresta do grafo Gr, de arestas invertidas, corresponde, em média uma aresta do grafo original G (assume-se que são raros os casos de haver múltiplas arestas a ligar o mesmo par de vértices; se não podermos assumir isto, então a complexidade é menor que O(|V|+|E|). O passo 2 do algoritmo também tem complexidade O(|V|+|E|) pois a pesquisa e adição de vértices a invAdj tem complexidade linear (por se usar um *unordered_map*).

O terceiro e último passo passa por uma nova pesquisa em profundidade, em que se usam as adjacências invertidas (por oposição às adjacências convencionais do grafo) de cada invAdj. Neste caso, quando se inicia a pesquisa numa nova árvore da floresta em expansão, inicia-se no vértice não visitado de numeração mais alta, ou seja, o mais próximo a contar do fim do vetor invOrder que ainda não foi visitado. Os vértices de cada árvore da floresta corresponde a um componente fortemente conexo (CFC) do grafo. Na nossa solução, foi criado um vetor de vetores de apontadores para vértices (vector<vector<Vertex*> >), chamado cfcs, em que cada "sub-vetor" corresponde a um CFC do grafo. Assim, basta criar e ir preenchendo um novo vetor para cada árvore obtida pela pesquisa em profundidade.

A complexidade deste terceiro passo é também O(|V| + |E|), sendo V o conjunto de vértices e E o conjunto de arestas do grafo.

Assim, globalmente, a solução implementada para este sub-problema tem complexidade O(|V| + |E|).

B. <u>Cálculo das menores distâncias</u>

Para a resolução do nosso problema em concreto, é necessário calcular as distâncias entre todos os pares de vértice. Esta necessidade será justificada mais à frente quando se abordar o algoritmo para encontrar o percurso mais curto do camião para a recolha do lixo.

No caso concreto do problema da recolha de lixo, dado que as arestas têm peso positivo por representarem distâncias, o algoritmo de Bellman-Ford não é interessante devido à sua complexidade temporal elevada no pior caso: O(|V|*|E|) para cada vértice de partida ou seja, O($|V|^2$*|E|) para encontrar todas as distâncias entre todos os pares de vértices.

Para resolver este sub-problema, implementou-se o algoritmo de Floyd-Warshall, de complexidade temporal O($|V|^3$) (Rossetti and Rocha 2015/2016), com uma pequena

mudança: em cada vértice, é guardado um *unordered_map* de "paths" onde se usa como chave a informação do vértice que lhe está associado.

Um "path" é uma nova estrutura de dados com três atributos:

- dest: apontador para o vértice que é possível visitar seguindo o path.
- next: apontador para a próxima aresta a visitar para ir desde o vértice que contém este path até dest, através do caminho de menor distância que liga os dois vértices.
- dist: distância total mínima que liga os dois vértices.

Convém salientar que um vértice não guarda o "path" (caminho) apenas para os vértices para o qual partilha uma aresta, mas sim para todos os vértices que são alcançáveis a partir deste ao percorrer várias arestas.
Por outra palavras, um vértice guarda um conjunto de "paths" de forma análoga a como uma cidade pode ter um conjunto de placas a apontar para outras cidades (dest) indicando a distância que separa as cidades (dist) se for tomada a rota apontada pelas placas (next).

Dado que se usa um *unordered_map* para guardar os "paths" (e esta estrutura de dados, tem, como já antes foi referido, complexidade temporal constante, O(1), para inserções e pesquisas), este passo adicional não afeta a complexidade temporal do algoritmo original, que continua a ser portanto $O(|V|^3)$.

Uma alternativa ao algoritmo de Floyd-Warshall é a aplicação do algoritmo de Dijkstra para cada vértice. A complexidade temporal global é $O(|V|*|E|*\log(|V|))$. O processamento adicional para gerar os "paths" não afeta a complexidade temporal do algoritmo.

Assim, para grafos esparsos (em que $|E| \sim |V|$), esta solução é mais eficiente que a de Floyd-Warshall, pois a complexidade de Dijkstra fica $O(|V|^2*\log(|V|))$ e a de Floyd-Warshall mantém-se $O(|V|^3)$.

Contudo, para grafos mais densos (em que $|E| \sim |V|^2$), é mais eficiente o algoritmo de Floyd-Warshall, de complexidade $O(|V|^3)$, que o de Dijkstra, de complexidade $O(|V|^3*\log(|V|))$.

Para demonstrar, de forma empírica, qual o melhor algoritmo para usar nas dois situações (grafo esparso vs. grafo denso), foram feitas duas pequenas experiências.

Na primeira experiência, foram gerados grafos completos de ordem n, Kn. Segundo Goodaire e Parmenter, um grafo é dito "completo" se cada par de vértices distintos (desprezam-se os anéis, arestas que ligam um vértice a ele mesmo) for adjacente (Goodaire and Parmenter 2006). É de notar que quando nos referimos ao grafo "completo" de ordem n, referimo-nos a grafos dirigidos. Assim, o grafo Kn, tem $|V| = n$ e $|E| = n*(n-1) = n^2 - n \sim n^2$, logo $|E| \sim |V|^2$, o que corresponde a um grafo denso.

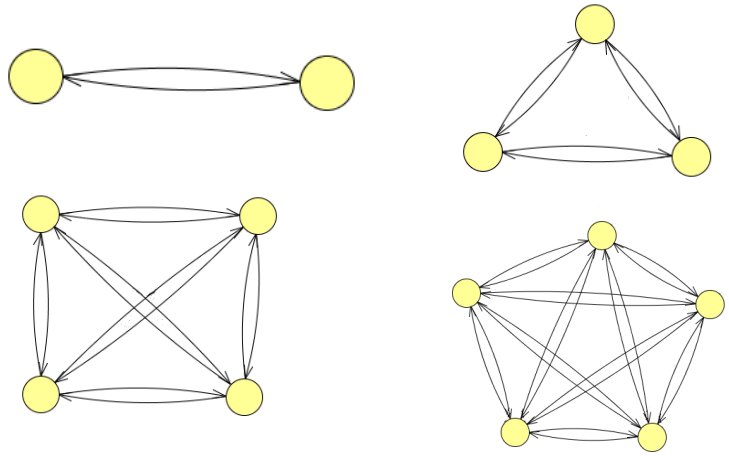

Figura 1: Grafos Kn para n = 2,3,4,5

De seguida, para os grafos Kn, com n entre 2 e 30, correu-se o algoritmo de Dijkstra para cada vértice e o algoritmo de Floyd-Warshall. Os resultados apresentam-se na imagem abaixo.

n	Tempo Dijsktra	Tempo Floyd-Warshall
2	0	0
3	0,001	0,001
4	0,005	0,003
5	0,009	0,005
6	0,014	0,008
7	0,021	0,013
8	0,03	0,02
9	0,036	0,023
10	0,06	0,034
11	0,071	0,037
12	0,072	0,039
13	0,085	0,049
14	0,112	0,06
15	0,139	0,072
16	0,156	0,098
17	0,175	0,111
18	0,227	0,163
19	0,247	0,151
20	0,28	0,169
21	0,32	0,2
22	0,254	0,239
23	0,401	0,25
24	0,477	0,297
25	0,502	0,337
26	0,565	0,358
27	0,619	0,409
28	0,682	0,451
29	0,768	0,489
30	0,851	0,565

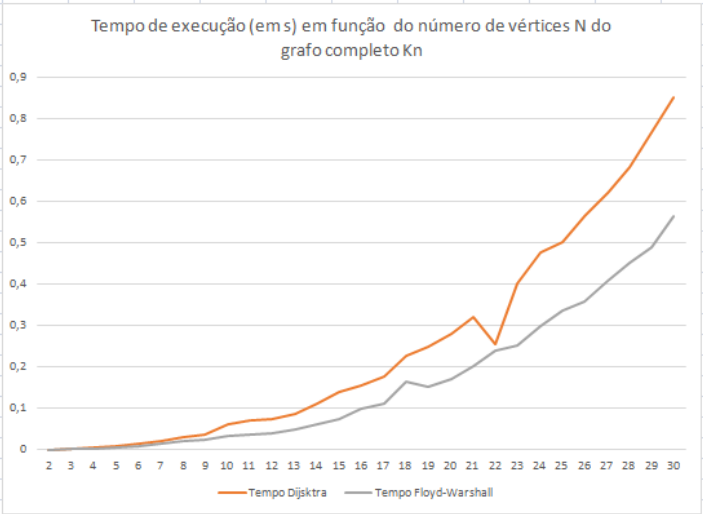

Figura 2: Medição do tempo de execução, de ambos os algoritmos para grafos Kn

Como se pode ver, o algoritmo de Floyd-Warshall é em média mais rápido que o de Dijksta para grafos densos. As diferenças entre os tempos de execução aumenta, à medida que n cresce pois o valor de |E| tem um crescimento maior (quadrático) que |V|.

É de notar que para n = 22, há uma quebra da curva: o tempo de execução do Dijkstra é menor que o esperado. Não se sabe ao certo a causa desta discrepância, mas ao efetuar de novo esta experiência, algumas vezes, houve quebras nas curvas para outros valores de n. Assim, em princípio, estes *outliers* são devidos às variações da utilização do CPU e da memória RAM, por outros processos, no momento da execução (causas externas à execução do programa e ao algoritmo em si).

Na segunda experiência, geraram-se listas simplesmente ligadas circulares, que correspondem a um caso particular de um grafo, que é uma estrutura de dados bastante genérica. Seja Ln uma lista simplesmente ligada circular com n elementos. Neste caso, |E| = |V| = n, o que corresponde a um grafo esparso.

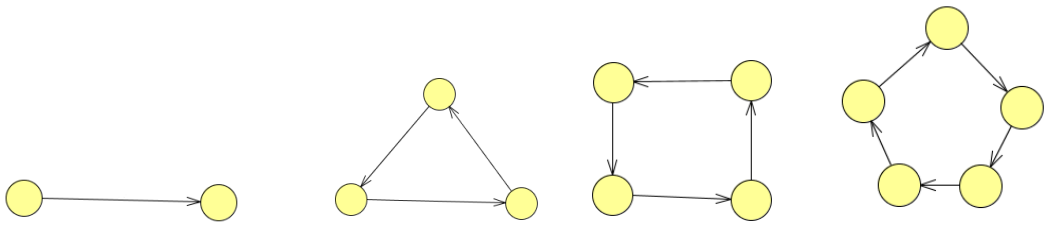

Figura 3: Grafos Ln para n = 2,3,4,5

De seguida, para os grafos Ln, com n entre 2 e 30, correu-se o algoritmo de Dijkstra para cada vértice e o algoritmo de Floyd-Warshall (como na primeira experiência). Os resultados apresentam-se na imagem abaixo.

n	Tempo Dijsktra	Tempo Floyd-Warshall
2	0,001	0
3	0,002	0
4	0,002	0,002
5	0,003	0,002
6	0,005	0,003
7	0,007	0,004
8	0,009	0,006
9	0,011	0,008
10	0,016	0,011
11	0,018	0,014
12	0,022	0,017
13	0,025	0,02
14	0,029	0,024
15	0,032	0,029
16	0,037	0,033
17	0,041	0,038
18	0,055	0,063
19	0,057	0,05
20	0,055	0,057
21	0,062	0,065
22	0,067	0,073
23	0,072	0,086
24	0,1	0,092
25	0,086	0,102
26	0,092	0,112
27	0,099	0,145
28	0,108	0,138
29	0,118	0,153
30	0,121	0,166

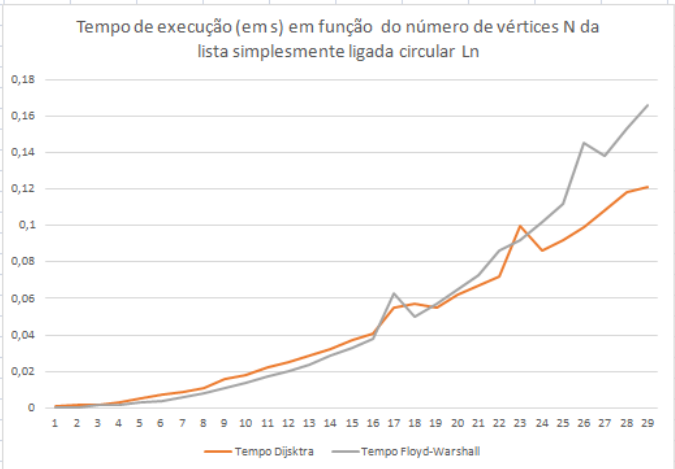

Figura 4: Medição do tempo de execução, de ambos os algoritmos para grafos Ln

Como se pode ver, neste caso, o algoritmo de Dijkstra, aplicado |V| vezes, é mais eficiente, em termos de tempo de execução, que o algoritmo de Floyd-Warshall, tal como tinha sido estabelecido em teoria. Também se pode constatar que o tempo de execução dos ambos os algoritmos sobre Ln, à medida que n cresce, é menor que para o grafo Kn, pois Ln tem menos arestas para processar que Kn (se n é pequeno, Kn e Ln diferem pouco em termos de número de arestas). Mais uma vez, os *outliers* ocorrem de forma pontual e irregular, o que sugere que não apresentam relevância em relação ao programa e ao algoritmo em si.

No cenário do nosso problema, as arestas representam ruas e estradas. Mais especificamente, representam vias de comunicação no interior de uma cidade (uma *smart city*). Neste caso, não é fácil de determinar se a cardinalidade de E é mais próxima de |V| ou de $|V|^2$, pois a grandeza de |E| depende do tipo de local (se é uma pequena aldeia, ou a baixa de uma cidade com ruas curtas e repleto de cruzamentos).

Assim, numa tentativa de usar cada um dos algoritmos (Dijkstra ou Floyd-Warshall) nas situações em que apresentam melhor desempenho, foi criada na classe Graph o atributo numEdges que guarda o número de arestas no grafo (para não ser necessário recalcular este valor sempre que necessário, o que teria um custo temporal O(|V+E|). Sempre que é acrescentado ou removido uma aresta do grafo, é incrementado ou decrementado esta variável, respetivamente. É também necessário ter o número de vértices, que pode ser obtido em tempo constante, O(1), usando o método .size() de *unordered_map*.

Tendo estes dois valores, |V| e |E|, definiu-se uma inequação que decide qual dos dois algoritmos usar, para gerar a estrutura de dados “paths” para vértice do grafo, no menor tempo possível:

se($|E| \geq |V|^2 / 2$)
{
  usar o algoritmo de Floyd-Warshall
}
senão
{
  usar o algoritmo de Dijkstra
}

Definiu-se, de forma empírica (não foi feito um estudo profundo para determinar o melhor valor de fronteira), que o valor de |E| que define a fronteira da decisão como sendo $|V|^2 / 2$, e não $|V|^2$ por duas razões. Por um lado, no caso concreto do nosso problema, é muito raro ter um vértice ligado a todos os outros vértices do grafo. Por outro lado, para um valor elevado de |V|, se |E| estiver próximo de $|V|^2 / 2$, o grafo resultante aproxima-se mais no caso de ser denso ($|E| \sim |V|^2$) que do caso de ser esparso ($|E| \sim |V|$).

C. Geração do caminho entre dois vértices

Graças ao esforço adicional de criar *unordered_maps* de "paths" para cada vértice no passo anterior, este novo subproblema torna-se bastante mais simples de resolver.

Entende-se por "geração do caminho" o problema de determinar a sequência ordenada de arestas a percorrer para ir de um vértice vi ∈ V a um vértice vf ∈ V.

O algoritmo para a resolução do subproblema é descrito abaixo, em pseudocódigo. Os valores de entrada são o vértice inicial vi, o vértice final vf e um vetor de apontadores para arestas, path, passado por referência, que guarda as arestas a percorrer para ir de vi a vf. O valores de saída é um booleano que indica se foi possível gerar o percurso de vi a vf.

1. if(vi ou vf não estão definidos)
    a. return false;
2. cur = vi; //sendo cur o vértice atual
3. while(cur != vf)
    a. paths = cur.getPaths();
    b. path = paths.find(vf); //procurar no *unordered_map* o path até o vértice de vf, partindo de cur
    c. if(path == null) //path não encontrado
        i. return false; //caso excecional em que os não há caminho de vi a vf
    d. next = path.getNext(); //próxima aresta a percorrer
    e. path.push_back(next); //vetor passado por referência que guarda o path (arestas) de vi a vf
    f. nextNode = next.getDest();
    g. cur = nextNode;
4. return true; //algoritmo terminou como esperado

É importante de notar que, na realidade, basta procurar vf em "paths" uma única vez (no início), pois se os "paths" foram corretamente gerados, dando um "passo" em direção a vf partindo de vi, vamos dar a um vértice que também apresenta um "path" até vf. Esta afirmação pode ser provada usando uma prova por indução, usando como caso base o vértice vf (a partir do qual se pode chegar a vf, usando zero arestas), e como passo indutivo um vértice va que tenha uma aresta que dê um vb, onde já foi provado que havia um caminho até vf.

Contudo, assumindo que o passos 3a e 3c são realizados em tempo O(1), e sabendo que a pesquisa de um elemento num *unordered_map* se realiza também com custo O(1), a complexidade temporal não aumenta por se realizar o teste a cada iteração. Esta redondância foi introduzida para tornar o algoritmo mais robusto e capaz de lidar com os casos em que os "paths" não tenham sido inicializados corretamente.

Os passos 1, 2 e 4 do algoritmo são executados uma única vez e têm complexidade temporal O(1).

Todos os passos no interior do ciclo *while* do passo 3 podem ser executados em tempo constante, dado que se usam *unordered_maps* e a inserção de um elemento do vetor também tem custo O(1) (assume-se também que os métodos *get* são executados em tempo constante).

No pior dos casos, ir de vi e vf implica passar por todos os vértices do grafo (só se passa uma única vez por cada um, porque senão o caminho teria um ciclo), pelo que o ciclo *while*, no pior caso e no caso médio, é executado |V| vezes.

Assim, a complexidade temporal deste algoritmo é O(|V|).

D. <u>Geração do caminho desde um vértice até a um vértice de chegada, passando por pontos de interesse</u>

Neste subproblema, assume-se que o vértice vi ∈ V, a sequência de vértice de chegada Vf ⊆ V e os pontos de interesse POIs ⊆ V fazem parte do mesmo componente fortemente conexo (CFC). A análise da conetividade do grafo já foi abordada.

O objetivo deste subproblema é determinar o conjunto de arestas a percorrer para, partindo de vi, passar por todos os vértices de POIs, e terminar num vértice qualquer de Vf.

O algoritmo usado para resolver este subproblema é descrito abaixo. Os dados de entrada são vi, POIs, Vf, e um vetor, path, passado por referência com a lista de arestas a percorrer.

1. if(vi = null || Vf.size() == 0)
    a. return false;
2. cur = vi; //sendo cur o vértice atual a ser processado
3. POIs_visited = 0; //numero de vértices de I visitados
4. for each(v: POIs)
    a. v.resetVisited();
5. while (POIs_visited < POIs.size())
    a. double min_dist = INF; //distância mínima desde cur ao POI não visitado mais próximo, o valor INF indica que ainda não foi processado nenhum POI não visitado.
    b. next_POI; //declaração do próximo POI a visitar
    c. for each(POI: POIs)
        i. if(!(POI.isVisited()))
            1. paths = cur.getPaths();
            2. path = paths.find(POI);

3. if(path == null) //path não encontrado
    a. return false; //caso excecional em que os não há caminho de cur a POI, pelo que não estão no mesmo CFC
4. dist = path.getDist();
5. if(dist < min_dist)
    a. min_dist = dist;
    b. next_POI = v;

d. next_POI->setVisited(true);
e. if(!findPath(cur, next_POI, path)) //algoritmo descrito na secção anterior para guardar em path o caminho de cur a next_POI
    i. return false;
f. cur = next_POI;
g. POIs_visited++;

//Determinar o elemento de Vf mais próximo de cur [...]

6. end = elemento de Vf mais próximo de cur
7. if(!findPath(cur, end, path)) //algoritmo descrito na secção anterior para guardar em path o caminho de cur a end
    a. return false;
8. return true;

É de notar que foi omitido o passo de determinar o elemento de Vf mais próximo de cur pois o algoritmo é o mesmo que os passos no interior do ciclo *while* do passo 5.

Os passos 1 a 3 são realizados em tempo O(1), e o passo 4, em tempo O(V), pois no pior caso POIs = V.
O passo 5 realiza-se em tempo $O(|V|^2)$. Com efeito, o ciclo *while* externo é executado aproximadamente |V| vezes (pior caso e caso médio). No pior caso, para uma iteração do passo 5 (ciclo *while*), o ciclo *for each* do passo 5c é executado |V| e todos os passos no seu interior podem ser realizados em tempo constante (a pesquisa num *unordered_map* dá se em tempo constante). Os passos 5d, 5f e 5g são executados em tempo constante, enquanto que o passo 5e (findPath) é executado com custo O(|V|), tal como foi visto no ponto anterior.

Pelos mesmos motivos que os apresentados no parágrafo anterior, o passo 6, dá-se em tempo O(|V|).

O passo 7, um findPath, dá-se em tempo O(|V|), e o passo 8, em tempo constante.

Assim, esta solução tem complexidade $O(|V|^2)$.

Tal como se pode ver, é adotada uma heurística *greedy* para determinar o próximo ponto de interesse a visitar. Este subproblema não tem uma subestrutura ótima, pelo que a aplicação desta heurística não conduz sistematicamente ao melhor resultado. Vejamos um contraexemplo simples. Seja vi = q0, POIs = {q1,q2} e Vf = {q3}.

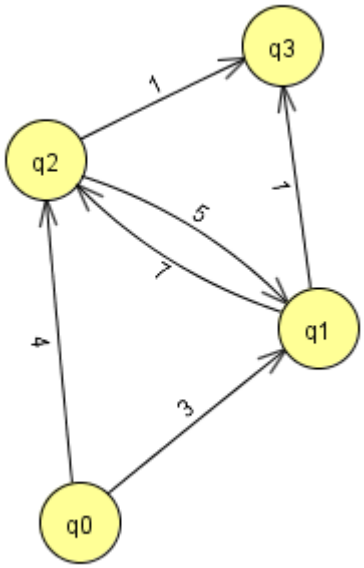

Figura 5: Grafo de contraexemplo à conjetura que o subproblema tem uma subestrutura ótima

Neste caso, é fácil de se constatar que o melhor percurso é começar em q0, ir para q2, de seguida para q1, e terminar em q3. A distância total percorrida seria 4 + 5 + 1 = 10. Ora, uma heurística *greedy* conduzir-nos-ia pelo percurso q0->q1->q2->q3, e a distância total seria 3 + 7 + 3 = 13. Mostra-se assim, com um contraexemplo (onde se desprezou o facto dos vértices não fazerem parte do mesmo CFC), que este problema não tem subestrutura ótima.

Ora, o contraexemplo anterior corresponde a um fenômeno raro (se não for impossível) no caso concreto do nosso problema numa *smart city*, pois, normalmente, há arestas de mesmo peso (ou de peso muito próximo) a ligar dois vértices em ambos os sentidos. Assim, a heurística *greedy*, apesar de não garantir a solução ótima, fornece uma boa aproximação desta com a vantagem de ter um custo temporal muito menor do que testar todas as sequências ordenadas de POIs a visitar (o que levaria a um custo fatorial O(|V|!) em termos de tempo, por se testar todas as permutações de elementos de POIs, e onde, no pior dos casos, POIs = V).

E. Recolha de um tipo de lixo por camião

Por fim, apresenta-se o algoritmo para gerar o conjunto arestas por onde passa um camião de capacidade cap para recolher o máximo possível de lixo de um tipo, partindo da central e acabando o percurso numa central de tratamento.

O algoritmo é apresentado abaixo em pseudocódigo. Os valores de entrada correspondem aos atributos da classe WasteManager. O valor de saída é um booleano que indica se a recolha de lixo foi efetuada com sucesso.

1. if (start == null || (ends.size() == 0))
    a. return false; //não foi definido
2. if (!updatedPaths)
    a. generatePaths(); //função com o algoritmo para gerar todos os "paths"
    b. updatedPaths = true;
3. if (!updatedCfcs)
    a. generateCfcs(); //função com o algoritmo para gerar todos os cfcs
    b. updatedPaths = true;
4. if (!sortedWaste)
    a. sortWaste();
    b. sortedWaste = true;
5. cfc = findCfc(start); //procurar o CFC onde está a central
   //Procurar os centros de tratamento que estão no CFC da central
6. for each (end: ends)
    a. if(end found in cfc)
        i. useful_ends.push_back(end); //guardar vetor de centrais de tratamento que fazem parte do CFC
7. if (useful_ends.size() == 0)
    a. return false;
8. used = 0; //capacidade do camião já usada
9. for each(vertex: wasteList) //wasteList é lista de vértices com lixo a recolher, ordenada de forma decrescente segunda a quantidade de lixo
    a. if (vertex.wasteRatio < criticalWasteRatio)
        i. wasteVector.erase(vertex);
        ii. continue;
    b. if (vertex.waste + used > cap)
        i. continue; //lixo excede capacidade remanescente do camião
    c. if(vertex found in cfc)
        i. used += vertex.waste;
        ii. vertex.waste = 0;
        iii. POIs.push_back(vertex); //adicionar vértice aos pontos de interesse (POIs) a visitar obrigatoriamente pelo camião

iv. wasteVector.erase(vertex);

10. if (POIs.size() == 0)
    a. return false;
11. graph.generatePOIPath(start, POIs, useful_ends, nodes, path); //chamar função que implementa o algortimo da secção anterior
12. return true;

Os passos 1, 10 e 12 têm complexidade O(1).
Tal como fora visto antes, o passo 2 (geração de "paths") tem custo $O(|V|^3)$ se for usado o Floyd-Warshall e custo O(|V|*|E|*log(|V|) se for usado o Dijkstra.
O passo 3 (análise de conetividade) tem custo O(|V|+|E|).
O passo 4 tem custo O(|V|*log(|V|)) pois é usado o método *sort* da estrutura de dados *list* da STL.
O passo 5 tem complexidade O(|V|) pois a estrutura de dados vector<vector<Vertex*> > tem exatamente |V| apontadores para vértices, quando somados os elementos de cada subsetor (pois todos os vértices pertencem a um componente fortemente conexo, mesmo que seja uma partição de V, que só inclua esse vértice)).
O ciclo *for each* do passo 6 é executado, no máximo |V| vezes (se todos vértices forem centros de tratamento). A pesquisa no passo 6a tem custo O(|V|), pois, no pior dos casos, todos os vértices de V fazem parte do mesmo CFC, no qual está incluído o vértice da central (apontado por start). Assim, este passo tem custo $O(|V|^2)$.
Os passos 7 e 8 têm custo O(1).
O ciclo *for each* do passo 9 é repetido, no pior dos casos |V| vezes (se wasteList tiver todos os vértices). Os passos 9a.i. e 9.c.iv têm custo O(1) pois a remoção de elemento de uma lista faz-se tempo constante. A inserção de um elemento num vetor ou uma lista (passo 9c.iii) faz-se em tempo O(1). O passo 9c (pesquisa em vetor) faz-se em tempo O(|V|), logo o passo 9 tem custo global $O(|V|^2)$.
O passo 11 tem complexidade $O(|V|^2)$.

Assim, o passo com maior custo (*bottleneck*) é o passo 2, de custo $O(|V|^3)$ ou O(|V|*|E|*log(|V|) dependendo do algoritmo usado para a geração dos "paths" (como foi visto, esta escolha é feita mediante o número de arestas do grafo), pelo que este custo corresponde à ordem de complexidade temporal de todo o algoritmo.

É de notar que são usados os booleanos updatedPaths e updatedCfcs (atributos da classe WasteManager) para evitar que se tenham de recalcular os "paths" e CFCs do grafo caso não tenha sido adicionado nenhum vértice ou nenhuma aresta ao grafo. Neste caso, a complexidade temporal do algoritmo passa a ser $O(|V|^2)$.

O passo 9a é efetuado para remover do vetor de vértices com lixo a recolher os caixotes com uma taxa de ocupação menor que o valor crítico. Estes vértices estão no vetor pois, inicialmente a sua taxa excedia o valor crítico, mas, após ter sido aumentada a capacidade

máxima de um caixote, a taxa ficou abaixo do valor limite. Optou-se por efetuar a remoção neste algoritmo e não ao alterar a capacidade máxima de um caixote, pois, dado que temos de iterar sobre todo a lista de vértices com lixo acima do valor crítico, a remoção faz-se em tempo O(1) (senão, seria preciso pesquisar o vértice na lista para o remover, o que teria custo linear em relação ao número de vértices do grafo, |V|).

Para concluir, este algoritmo adota uma heurística *greedy* ao optar por recolher os caixotes de lixo com maior quantidade de lixo. Isto garante que é minimizado o número de camiões necessário para recolher todo o lixo de um certo tipo da cidade (e garante portanto uma minimização da função objetivo f).

4. Casos de utilização

A aplicação desenvolvida permite simular todo o processo de depósito e posterior recolha de lixo dos contentores de uma cidade. Este lixo pode ser de um dos quatro seguintes tipos: papel, vidro, plástico ou lixo comum.
Em particular, a aplicação apresenta as seguintes funcionalidades:

- Definir a taxa de ocupação de um contentor a partir do qual vale a pena recolher o seu lixo.
- Carregar um grafo através dos três ficheiros .txt gerados pelo parser OSM2TXT quando aplicado sobre ficheiros de OpenStreetMap(.osm).
- Acrescentar vértices e arestas ao grafo, que são identificados por ID's (há ID's distintos para vértices e arestas).
- Remover vértices e arestas indicando o seu ID.
- Remover todas as arestas que vão de um certo vértice a outro (a indicar pelos ID's dos dois vértices extremos).
- Desativar e reativar arestas do grafo, para que não sejam compatibilizadas na pesquisa do percurso, sem ser necessário remover-las.
- Definir a capacidade máxima de cada vértice, para cada tipo de lixo, usando o ID do vértice.
- Acrescentar lixo de um certo tipo a qualquer vértice (usando o ID do vértice). Se não for possível acrescentar todo o lixo ao vértice por excesso da capacidade máxima do contentor, é indicado a quantidade de lixo remanescente.
- Obter a quantidade atual e máxima, e a taxa de ocupação de um certo tipo de lixo num vértice (usando o ID do vértice).
- Obter e modificar o nome de um vértice ou de uma aresta (usando o ID).
- Obter e modificar o peso de uma aresta (usando o ID).
- Definir o vértice que representa a central de camiões (onde começa o percurso de um camião) (usando o ID do vértice).
- Acrescentar e remover vértices da lista das estações de tratamento de lixo (onde acaba o percurso de um camião) (usando o ID do vértice).
- Recolher um certo tipo de lixo, indicando a capacidade do camião usado. Como resultado, são obtidas duas sequências ordenadas com os vértices e arestas visitadas pelo camião, desde a central a um centro de tratamento, passando pelo máximo de contentores possível. Se não houver lixo a recolher no componente fortemente conexo da central, ou se não houver centros de tratamento neste componente, não é efetuada nenhuma recolha e o utilizador é notificado disto.

5. Principais dificuldades encontradas no desenvolvimento do trabalho

A principal dificuldade ao longo do desenvolvimento do trabalho, foi a falta de tempo que houve, face aos objetivos elevados que tivemos para este trabalho.

Podemos dizer que este objetivo foi ultrapassado através do esforço colectivo.

Relacionada com a dificuldade anterior, também sentimos que foram necessárias várias revisões do código, até chegarmos às estruturas de dados e algoritmos ideais para a solução do problema.

Esta dificuldade também foi ultrapassada, com alguma pesquisa da nossa parte e discussão colectiva.

6. Esforço dedicado por cada elemento do grupo

O trabalho foi realizado em conjunto, utilizando a ferramenta *GIT* para sincronizar e organizar o código, sendo que por norma, todos os elementos do grupo se encontravam presentes na mesma chamada *SKYPE* nas alturas em que o trabalho estava a ser realizado.

Dito isto, não houve grandes disparidades de contribuição, cada um se focou mais nas áreas em que se sentia mais à vontade, sendo que o membro do grupo Gonçalo Leão, assumiu um certo grau de liderança e distribuiu tarefas pelos restantes membros, criação de funções, desenvolvimento da interface, obtenção de imagens e gráficos para a realização do relatório, etc. Houve um bom entendimento entre os membros do grupo e sentimos que ninguém teve uma contribuição inferior à dos restantes.

6.1. Desenvolvimento da interface

Maioritariamente realizada pelos membros, Eduardo Leite e Francisco Queirós, com intervenção do Gonçalo Leão.

6.2. Desenvolvimento do código

Realizada em conjunto, sendo que o membro Gonçalo Leão teve mais tempo a desenvolver as partes que necessitavam estruturas de dados e algoritmos que afetavam a eficiência do programa, mudando várias vezes partes do código de forma a atingir a maneira ideal e assegurando a sua qualidade atráves de *CUTE TESTS*. O membro Eduardo Leite teve uma participação mais ativa no aperfeiçoamento da forma como se liam e organizavam os dados lidos do *Parser* OSM2TXT assim como eles eram desenhados na ferramenta *JUNG* e o membro Francisco Queirós teve uma participação de destaque nos assuntos que diziam respeito aos erros causados pelo *GIT* e na resolução de erros causados pela falta de alguma configuração específica do *Visual Studio*, do *CUTE*, do *Parser*, etc.

6.3. Desenvolvimento do relatório

A escrita foi feita maioritariamente pelo membro Gonçalo Leão, sendo que o membro Eduardo Leite participou na discussão e revisão dos conteúdos e foi contribuindo para o conteúdo do mesmo, através da criação de componentes como gráficos e imagens. O membro Francisco foi participando na discussões que mais dúvidas geravam, com críticas construtivas e sugestões.

7. Bibliografia

Goodaire, Edgar G., and Michael M. Parmenter. 2006. *Discrete Mathematics with Graph Theory*. Prentice Hall.

Rossetti, Rosaldo, and Ana Paula Rocha. 2015/2016. “Algoritmos Em Grafos: Caminho Mais Curto”

———. 2015/2016. “Algoritmos Em Grafos: Conetividade”

Skiena, Steven S. 2008. *The Algorithm Design Manual*.