U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Recolha de Lixos & Resíduos

Concepção e Análise de Algoritmos

Elaborado por:

Turma 4 - Grupo D

Diogo Moura – up201304068 - up2013049068@fe.up.pt

Pedro Silva – up201306032- up201306032@fe.up.pt

Sérgio Domingues – up201304367- up201304367@fe.up.pt

23 de Abril de 2015

Índice:

Introdução:

No âmbito da Unidade Curricular de Concepção e Análise de Algoritmos foi nos apresentado um problema, o qual deveríamos resolver da forma mais optimizada e eficaz possível.

No entanto para um melhor enquadramento e entendimento da situação em estudo o nosso grupo optou por fazer uma formalização e posterior definição do mesmo.

Assim sendo, este relatório irá expor o contexto do problema em que estamos inseridos através de esquemas,diagramas e métodos auxiliares como imagens e definições de conceitos ainda não explorados pelo leitor.

Após a contextualização do tema procederemos à especificação da estratégia de resolução adoptada para solução do problema, bem como a demostração dos algoritmos e métodos utilizados.

Descrição do problema

O objetivo do presente trabalho é construir um programa que consiga implementar uma solução em forma de grafos, determinando um percurso.

O projeto deverá ser composto por duas versões. A primeira tenciona determinar o menor caminho possível desde a central até ao ponto de recolha passando por todos os caixotes prioritários.

A segunda versão deverá construir um caminho que deva cumprir os requisitos acima enunciados mas com a adição de vários camiões que podem transportar uma quantidade de lixo limitada.

Formalização do Problema:

Temos como objectivo obter rotas de recolha de lixo, optimizadas, numa “smart city”. Os contentores distribuídos pela cidade possuem sensores de capacidade que são actualizados convenientemente. Considera-se como um caixote de lixo prioritário um caixote que esteja com uma taxa de ocupação superior a 70%.
Deparamo-nos então com um problema dividido em duas partes:

1) Primeira abordagem: consiste na fase inicial em que apenas é considerado um único camião de recolha de lixo, com capacidade ilimitada. O objectivo desta abordagem consiste em encontrar o caminho mais curto entre a Central (ponto de partida do camião) e o Ponto de Recolha de lixo passando por todos os caixotes prioritários.

Analisando esta abordagem conclui-se que o problema pode ter várias soluções óptimas, contudo não existe uma forma rápida e eficiente de a obter, para além disto é de salientar que, quanto maior for a dimensão do problema, maior será a complexidade (espaço-temporal) associada, deste modo o seu tempo pode facilmente escalar, apesar de serem usadas eurísticas já bastante estudadas e discutidas no âmbito da algoritmia, nomeadamente nos casos de problemas **NP-complete** (“non deterministic polynomial time”).

2) Numa segunda fase do problema, considera-se que podem existir vários camiões com capacidade limitada. Nesta parte tentar-se-á optimizar, para além do percurso, o número de camiões usados, procurando-se utilizar o menor número possível de camiões. Em alguns algoritmos ter-se-á em conta a quantidade de lixo presente nos contentores prioritários. Outros apenas terão em conta a distância entre os mesmos.

Tal como foi referido no relatório intercalar, foram usados grafos do tipo dirigido como estrutura de dados responsável por armazenar a informação relativa aos

contentores, na qual cada vértice representa um contentor e cada aresta uma ligação entre dois vértices (contentores) a qual armazena o peso entre dois vértices.

Em relação aos contentores optamos por introduzir variáveis que armazenam a quantidade máxima e a quantidade atual de lixo no mesmo. Optamos também por introduzir uma variável limite que simboliza a percentagem limite a partir da qual um contentor é considerado prioritário tal como é simbolizado na imagem apresentada abaixo. Cada vértice(contentor) possui também um id que varia de vértice para vértice e uma variável do tipo *string* correspondente ao nome rua na qual o mesmo está localizado.

```
class Contentor {
private:
    static unsigned int nextId;
    unsigned int id;
    string rua;
    unsigned int quantidadeLixo, quantidadeMaxima;
    static constexpr double limite = 0.7;
```

Para verificar se um contentor é prioritário é usado um método de nome isPrioritario() que através do quociente entre a sua quantidade de lixo atual e a sua quantidade máxima nos indica se está ultrapassada ou não a barreira dos 70% (valor predefinido) de capacidade.
Para uma melhor ilustração do caso acima explicado fica aqui uma imagem da implementação da função acima referida:

```
bool Contentor::isPrioritario() {
    if(quantidadeMaxima==0){
        return false;
    }

    if(((double)quantidadeLixo/quantidadeMaxima)>limite){
        return true;

    }else return false;
}
```

Os camiões seguem uma linha de implementação semelhante à dos contentores, a nível de atributos, pois também contêm uma quantidade de lixo actual e uma capacidade máxima de lixo que podem transportar. Cada camião contém também o seu próprio id o que facilita a distinção entre os mesmos para além disso cada camião armazena em registo todos os contentores por onde passou.
Vector foi a estrutura de dados escolhida para representar a rota de um Camião tal como indicado na imagem abaixo.

```
class Camiao {
private:
    static int capacidadeMaxima;
    static unsigned int nextId;
    unsigned int id;
    unsigned int quantidadeLixo;
    unsigned int distanciaPercorrida;
    vector<Contentor> rota;
```

Dados:

Construção de um grafo dirigido, G=(V,E), de contentores em que:

Vértices(V): corresponde ao conjunto de todos os contentores

Arestas(E): corresponde à distância entre todos os pares de contentores.

O grafo G é fortemente conexo pois cada vértice estará ligado aos restantes.
Seja o número de vértices e arestas igual à seguinte expressão:

$$e = v^2 - v$$

Sendo e o número de arestas e v o número de vértices tal que $e, v \in \mathbb{N}$.

Uso de ficheiros de texto (extensão *.txt*) para:

- Contentores;
- Adjacências;

Também usamos uma estrutura de dados do tipo **vector** para representar o conjunto de camiões que a Central de Recolha de lixo possui.

Input:

Os inputs para a execução do programa serão feitos através de informação descarregada de ficheiros .txt que contêm a informação de caixotes e adjacência de vértices.
Este processo será executado na classe LoadGraph que contém respectivamente 2 funções:

- LoadContentores() - Função responsável por transformar a informação de cada contentor presente no caixotes.txt de contentores num objecto da classe contentor sendo este posteriormente adicionado como vértice do nosso grafo;

- LoadAdjacentes() - Função responsável por transformar a informação de cada adjacência presente no ficheiro adj.txt numa arresta do nosso grafo de trabalho sendo depois esta adjacência adicionada ao nosso grafo de trabalho;

Para uma melhor ilustração da composição dos ficheiros txt deixamos em anexo algumas imagens que esperamos que ajudem numa melhor compreensão do processo:

Adj.txt

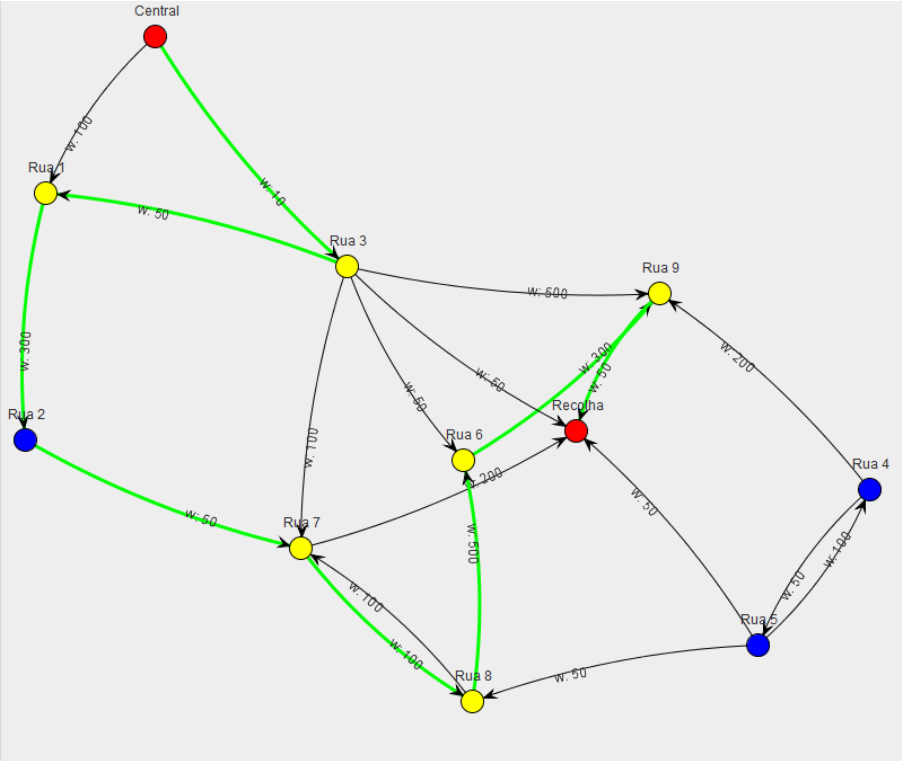
```
0,1,100
0,3,10
1,2,300
5,10,50
3,1,50
2,7,50
3,6,50
9,10,50
3,10,50
8,7,100
4,5,50
5,8,50
6,9,300
7,10,200
7,8,100
3,7,100
5,4,100
3,9,500
4,9,200
8,6,500
```

Contentores.txt

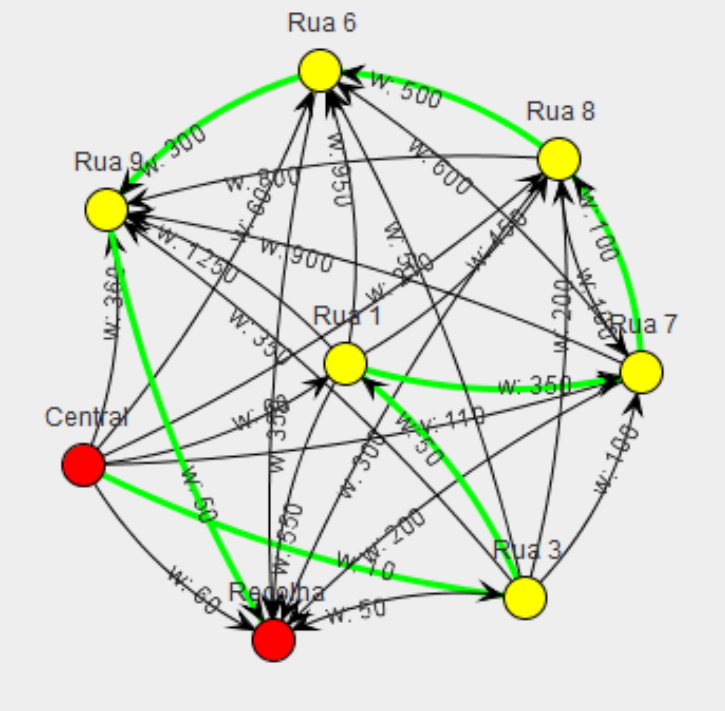
```
11
0,Central,0,0
1,Rua 1,800,1000
2,Rua 2,500,1000
3,Rua 3,2523,3000
4,Rua 4,432,3000
5,Rua 5,1543,3000
6,Rua 6,2456,3000
7,Rua 7,4665,5000
8,Rua 8,2450,3000
9,Rua 9,2550,3000
10,Recolha,0,0
```

Output:

Como output final do nosso trabalho optamos por mostrar primeiramente o nosso grafo original de trabalho com todas as rotas possíveis porém sinalizando a melhor rota a verde,para além disso decidimos por o ponto inicial e final a vermelho e os caixotes prioritários sinalizados a amarelo e os não prioritários a azul tal como acontece na imagem presente abaixo:



Em ambas as abordagens procedemos ao pré-processamento do grafo representativo dos dados lidos, ao grafo filtrado, onde são excluídos os grafos não prioritários, damos o nome de grafo de trabalho (**WorkingGraph**).Este grafo de trabalho é obtido quando o método **newWorkingGraph()** é invocado. A representação final do mesmo é a seguinte:



Restrições:

De modo a optimizar o mais possível o nosso trabalho optamos por definir um limite de 1000 contentores por região sendo que:

$$v \leq 1000$$

Por outro lado e tal como referido anteriormente numa segunda fase do projeto cada camião terá uma capacidade limite de lixo, que poderá transportar.

Também admitimos que existe um caminho direto de cada caixote para o centro de recolha e o primeiro ponto do ficheiro de contentores é sempre a central e o último a recolha.

Resultados Esperados:

No final do nosso trabalho esperamos obter uma solução óptima para a rota dos camiões (ou do camião no caso da primeira abordagem) tendo em conta a distância entre os contentores e a sua quantidade de lixo. Serão comparados vários resultados obtidos pelos algoritmos usados.

Descrição da Solução:

A nossa estratégia de resolução para os problemas consiste em inicialmente extrair os dados do ficheiro de input e com eles criar o grafo. Este será organizado para que cada vértice represente um caixote, cada aresta represente a distância entre dois caixotes.

Numa primeira fase do trabalho optamos por pré-processar o grafo inicial para extrair apenas os nós onde existem caixotes prioritários e todos os vértices ligados aos outros quando possível (Floyd-Warshall) formando assim um novo grafo de trabalho constituído apenas por nós com caixotes prioritários e pelo ponto de partida(Central) e ponto final(Recolha).

Descricao dos Algoritmos Utilizados

1º Parte

Floyd-Warshall:

A complexidade deste algoritmo não é constante pois pode variar para situações diferentes no entanto podemos afirmar que:

- No melhor caso possível a sua complexidade é de $O(V^3)$ sendo v o número de arestas do grafo;

- No pior caso possível a sua complexidade é de $O(V^3)$ sendo v o número de arestas do grafo;

Matematicamente, podemos descrever este algoritmo com a seguinte fórmula:

$$D_{ij}^{n} = \min(D_{ij}^{n-1}, D_{ik}^{n-1} + D_{kj}^{n-1})$$

Este algoritmo começa por comparar todos os caminhos possíveis entre pares de vértices retornando o caminho mais curto entre esses mesmos pares.
De seguida, aplica-se uma técnica de reconstrução de caminhos no grafo para substituir a aresta que liga o par de vértices pela aresta com menor peso neste caso.

Neste ponto é criado o nosso Working Graph, através da remoção de vértices e arestas que não são relevantes para a resolucao da 1 parte do problema. Ficamos assim apenas com os vértices(contentores) que precisam de ser esvaziados. Encontramos neste ponto da nossa resolucao o problema do caixeiro viajante(TSP).

Para a resolução deste problema aplicamos 3 algoritmos de complexidades diferentes, o Nearest Neighbour, Força Bruta através das permutações de todos os caminhos possíveis e Branch and Bound.

Nearest Neighbour

O Nearest Neighbour parte do ponto inicial, a central, e percorre todas as arrestas, calcula a que tem menor distancia e segue esse caminho. Faz este método até todas os vértices terem sido visitados e acaba no ponto de recolha.

```
void nearestNeighbour(Graph<Contentor> &grafo) {
    for (unsigned int i = 0; i < grafo.getNumVertex(); i++) {
        grafo.getVertexSet()[i]->setVisited(false);
    }
    Vertex<Contentor>* actual = grafo.getVertexSet()[0];
    Vertex<Contentor>* proximo;
    unsigned int pesoMinimo = INT_INFINITY;
    actual->setVisited(true);

    for (unsigned int i = 0; i < grafo.getNumVertex(); i++) {
        for (unsigned int j = 0; j < actual->getAdj().size(); j++) {
            if (actual->getAdj()[j].getWeight() < pesoMinimo && !actual->getAdj()[j].getDest()->isVisited()) {
                if ((actual->getAdj()[j].getDest()->getInfo().getQuantidadeMaxima() == 0) && i < grafo.getNumVertex() - 1) {
                    continue;
                } else {
                    pesoMinimo = actual->getAdj()[j].getWeight();
                    proximo = actual->getAdj()[j].getDest();
                }
            }
        }
        if (!(proximo->getInfo() == actual->getInfo())) {
            actual->path = proximo;
        }
        actual = proximo;
        actual->setVisited(true);
        pesoMinimo = INT_INFINITY;

    }

}
```

Este algoritmo é bastante rápido mas em tem pouca probabilidade de dar o melhor caminho possivel. Em certos casos pode mesmo dar o pior caminho.
No pior dos casos a complexidade do algoritmo é $O(v^2)$. em que o v representa número de vértices.

Brute-Force (Permutações)

Também optamos por implementar outra solução brute-force mas que neste caso recorresse a permutações de todos os caminhos possíveis. Neste é guardado o caminho mais curto e a suas respetiva distância. A cada permutação actualiza o caminho e a distancia mínima.

```
void brute_force(Graph<Contentor> &grafo,Camiao &c) {

    int sum = 0, min_dist = INT_INFINITY;
    int size = grafo.getNumVertex();

    int route[size], vert[size];

    for (int i = 0; i < size; i++) {
        vert[i] = i;
    }

    while (std::next_permutation(vert + 1, vert + size - 1)) {

        for (int i = 0; i < size - 1; i++) {

            if (grafo.getW()[vert[i]][vert[i + 1]] == INT_INFINITY)
                sum = INT_INFINITY;

            if (sum != INT_INFINITY)
                sum += grafo.getW()[vert[i]][vert[i + 1]];
            else break; //se der porcaria e por causa disto
        }

        if (sum < min_dist) {
            min_dist = sum;
            iguala_arrays(route, vert, size);
        }
        sum = 0;
    }

    for (unsigned int i = 0; i < size - 1; i++) {
        grafo.getVertexSet()[route[i]]->path = grafo.getVertexSet()[route[i + 1]];
        c.addContentor(grafo.getVertexSet()[route[i]]->getInfo());
    }
}
```

Em termos temporais este algoritmo segue uma complexidade do tipo:

$O(V!)$ sendo V o número de vértices.

Em termos espaciais o algoritmo segue uma complexidade de: $O(1)$

Branch and Bound (Matriz de Custo Reduzido)

Dos dois algoritmos implementados em cima, um dá varias vezes o caminho não ótimo e outro é muito dispendioso, por isso optamos por uma eurística branch and bound para comparação de resultados.

Sobre esta eurística podemos afirmar que segue uma complexidade :

$O(V^2)$, sendo V o número de vértices do grafo.

Este método calcula as matriz reduzidas das subárvores do nó em questão e analisa qual dos bounds das varias subárvores é o menor. De seguida aplica-se o mesmo para as subárvores calculadas anteriormente. A cada iteração registamos o caminho e quando não existirem mais subárvores possíveis retornamos o caminho.

```
void branchBoundRec(Graph<Contentor> &grafo, vector<int> &path, int id, int** W) {

    Vertex<Contentor>* actual = grafo.getVertexSet()[id];

    if (path.size() == grafo.getNumVertex() - 1) {
        return;
    }

    int minBound = INT_MAX, novoId, idComparar;

    for (unsigned int i = 0; i < actual->getAdj().size(); i++) {
        idComparar = getVertexIndice(grafo, actual->getAdj()[i].getDest()->getInfo().getId());

        if (checkBounds(id, idComparar, W, grafo.getNumVertex()) < minBound && !visitedPath(path, idComparar)
                && idComparar != getVertexIndice(grafo, grafo.getVertexSet()[grafo.getNumVertex() - 1]->getInfo().getId())) {
            minBound = checkBounds(id, getVertexIndice(grafo, actual->getAdj()[i].getDest()->getInfo().getId()), W, grafo.getNumVertex());
            novoId = getVertexIndice(grafo, actual->getAdj()[i].getDest()->getInfo().getId());
        }
    }
    path.push_back(novoId);

    for (int i = 0; i < grafo.getNumVertex(); i++) {
        W[id][i] = INT_INFINITY;
    }

    for (int i = 0; i < grafo.getNumVertex(); i++) {
        W[i][novoId] = INT_INFINITY;
    }

    W[novoId][id] = INT_INFINITY;

    rowReduction(W, grafo.getNumVertex());
    colReduction(W, grafo.getNumVertex());

    branchBoundRec(grafo, path, novoId, W);

}
```

2º parte

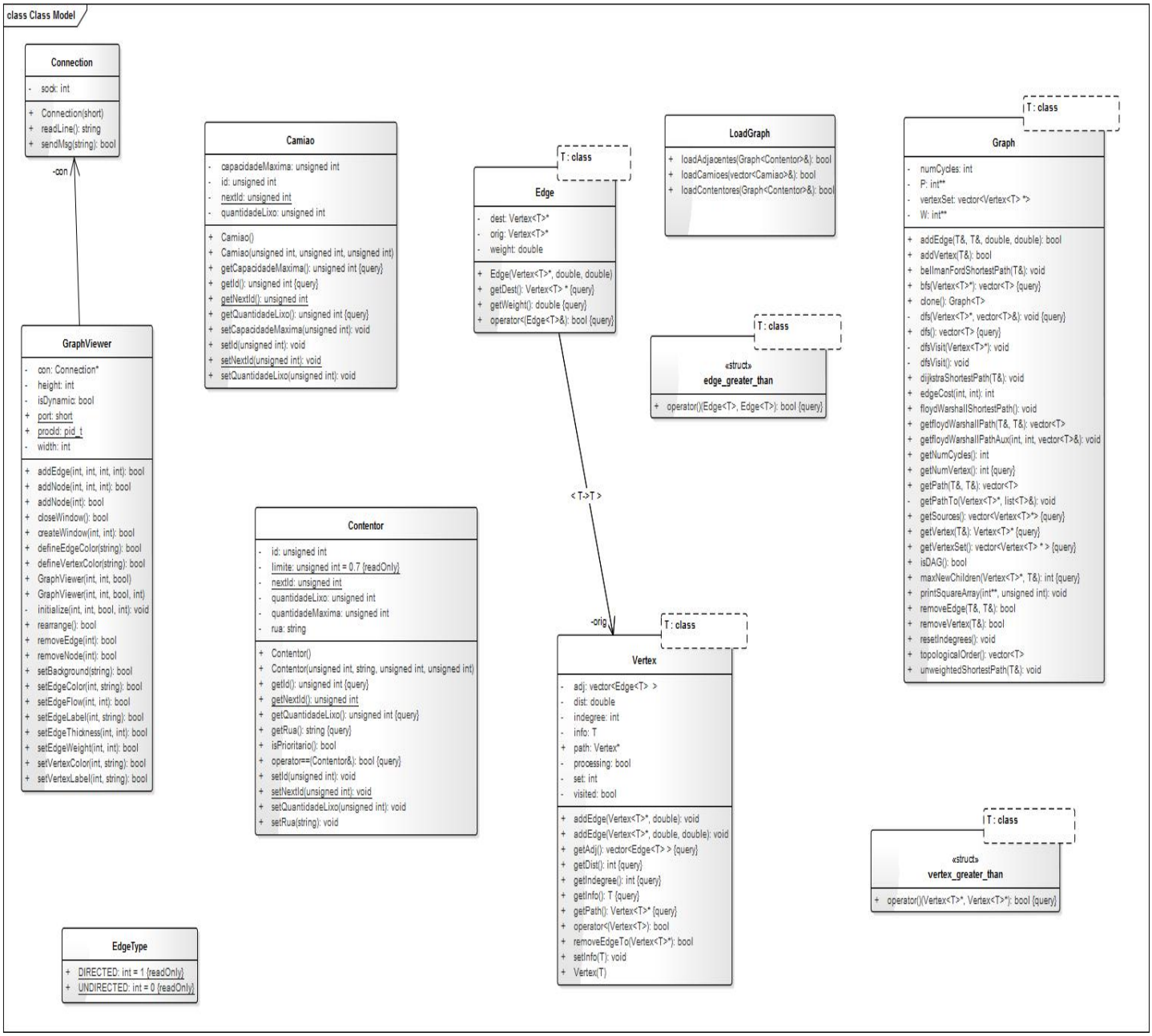
Nearest Neighbour Adaptado

Na 2ª fase do trabalho pretendemos para além de encontrar a melhor rota possível optimizar o número de camiões utilizados no processo para tal voltamos a aplicar o método nearestNeighbour só que desta vez aplicado aos vários camiões. O algoritmo executa para 1 camião com uma quantidade predefinida e vai apanhar a máxima quantidade de lixo possível. Este algoritmo é executado varias vezes até não existirem mais contentores prioritários.
No final é retornada a rota, a quantidade de lixo apanhada e a distância percorrida por cada camião.

Brute-Force Adaptado

Algoritmo semelhante ao da primeira parte do problema mas adaptado para suportar vários camiões, priorizando as rotas com maior quantidade de lixo.

Diagrama de Classes UML:



Lista de casos de utilização:

- Aplicação dos 3 algoritmos para resolução da primeira parte do trabalho
- Visualização do grafos com a rota óptima traçada
- Visualização do grafo de trabalho onde são aplicados os algoritmos
- Comparação de velocidade dos algoritmos da 1º parte.
- Aplicação dos algoritmos para a 2º parte.
- Visualização das rotas na consola assim como o lixo recolhido por cada camião e a distância percorrida.

Principais dificuldades no desenvolvimento do projeto:

Devido à extensa pesquisa e trabalho de preparação realizado anteriormente à implementação, as principais dificuldades que encontrámos resumem-se à escolha dos algoritmos a utilizar, associados com a complexidade do tema do trabalho escolhido. Na sua implementação, nomeadamente na implementação da eurística de resolução pelo método branch and bound, sentimos a maior dificuldade. Sentimos também algumas dificuldades na adaptação dos algoritmos para a segunda abordagem do problema.

Conclusões finais:

Antes de mais é necessário salientar que apesar das dificuldades todos os membros do grupo se envolveram ao máximo no desenvolvimento do projeto no entanto é necessário ressalvar que os elementos Diogo Moura e Sérgio Domingues tiveram um maior impacto no desenvolvimento dos algoritmos a aplicar no projeto enquanto que o membro Pedro Silva teve um maior impacto no desenvolvimento do relatório de apresentação final.

Em suma,podemos afirmar que todos os elementos do grupo estão contentes com o desenvolvimento do projeto pois conseguimos cumprir os objetivos a que nos propusemos com distinção.