U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Projeto de Concepção e Análise de Algoritmos

PapaRica: Distribuição de refeições prontas

Parte 1

André de Jesus Fernandes Flores – up201907001

Diogo Luís Araújo de Faria – up201907014

Tiago André Batista Rodrigues – up201906807

Índice

1. Descrição do tema

A **PapaRica** é uma empresa de confeção e distribuição de refeições prontas a consumir. Para tal, existe uma frota de veículos encarregue das entregas ao cliente, sendo que as recolhe diariamente em Vila do Conde para as distribuir na área metropolitana do Porto.

As encomendas são distribuídas em cabazes, sendo identificadas com a informação relevante acerca do destino e conteúdo.

O trabalho consiste em implementar um sistema que calcule os trajetos ótimos para a realização das entregas.

Numa primeira fase vai-se considerar a existência de um único veículo de entrega com capacidade ilimitada, sendo que se expande numa segunda fase a uma frota de veículos de diferentes capacidades e tipos.

Um exemplo de um trajeto de um veículo pode ser:

Sede -> Cliente 1 -> … -> Cliente n -> Sede

Para uma entrega poder ser realizada tem de existir pelo menos um trajeto possível que permita sair e retornar à Sede passando por todos os Clientes, tendo em consideração a existência de obras públicas que podem, em certas situações, impossibilitar a entrega de certas encomendas.

2. Formalização do problema

2.1. Dados de entrada

- Vi – Sequência de veículos disponíveis para utilização, sendo Vi[n] o n-ésimo elemento e caracterizado por:
  - type – Tipo de veículo (numa fase inicial = 'heavy');
  - cap – Capacidade (numa fase inicial = ∞).
- Ci – Sequência de cabazes para distribuição, sendo Ci[n] o n-ésimo elemento e caracterizado por:
  - clientName – Nome do destinatário;
  - packageNumber – Número de embalagens contidas no cabaz;
  - invoiceNumber – Número da fatura;
  - destAddress – Vértice de destino.
- Gi = (Ni, Ei) – Grafo dirigido pesado inicial, composto por:
  - N – Vértices do grafo, que representam pontos da cidade, caracterizados por:
    - type – Tipo de vértice;
    - address – Endereço;
    - adj ⊆ E – Conjunto de arestas que se iniciam do vértice.
  - E – Arestas do grafo, que representam o caminho entre 2 vértices, caracterizados por:
    - weight – Peso da aresta, que representa a distância;
    - dest ∈ Ni – Vértice de destino;
    - ID – Identificador de aresta.
- S ∈ Ni – Vértice que representa a Sede, de onde o veículo sai e retorna.

2.2. Dados de Saída

- Gf = (Nf, Ef) – Grafo dirigido pesado final, sendo que Nf = Ni e Ef = Ei.
- Vf – Sequência de veículos utilizados em entregas, sendo Vf[n] o n-ésimo elemento. Cada um é caracterizado por:
  - T – Sequência de vértices ordenados por ordem de passagem (com possibilidade de repetidos);
  - B – Sequência de cabazes para entrega pelo veículo ordenada pela ordem de entrega;
  - cap – Capacidade ocupada do veículo.

2.3. Restrições

2.3.1. Restrições sobre os dados de entrada

- ∀ n ∈ [0, |Vi| - 1]:
  - type(Vi[n]) = 'light' ∨ 'heavy' ∨ 'motorcycles' – tipos de veículos têm de ser carros ligeiros ('light'), carros pesados ('heavy') ou motociclos ('motorcycles');
  - cap(Vi[n]) >= 0 – capacidade têm de ser maior ou igual a zero, visto que se trata de uma quantidade de embalagens.
- ∀ n ∈ [0, |Ci| - 1]:
  - packageNumber(Ci[n]) >= 1 – número de embalagens tem de ser positivo, visto que um cabaz tem de ter pelo menos uma embalagem;
  - invoiceNumber(Ci[n]) >= 1 – número de fatura tem de ser maior ou igual a um;
  - destAddress(Ci[n]) ∈ Ni – vértice de destino tem de existir no conjunto de vértices do grafo.
- ∀ n ∈ Ni, type(n) = 'HQ' ∨ 'destiny' ∨ 'intermediate'.
- ∀ e ∈ Ei:
  - weight(e) >= 0 – arestas têm de ter um peso igual ou maior que zero visto que este corresponde à distância;
  - ID(e) >= 0 ∧ $\{\forall e1, e2 \in Ei \mid ID(e1) = ID(e2) \rightarrow e1 = e2\}$ – identificador de uma aresta tem de ser maior ou igual a zero e único para cada uma;
  - Deve ser utilizável por veículos.
- type(S) = 'HQ' – tipo de sede deve ser 'HQ'.

2.3.2. Restrições sobre os dados de saída

- ∀ $vi \in Vi$, $\exists\, vf \in Vf \mid vi = vf$ – os vértices iniciais e finais vão ser iguais.
- ∀ $ei \in Ei$, $\exists\, ef \in Ef \mid ei = ef$ – as arestas iniciais e finais vão ser iguais.
- Vf ⊆ Vi – os veículos utilizados têm de ser parte do conjunto de veículos disponíveis.
- ∀ v ∈ Vf:
  - T(v) ⊆ Vi – os vértices têm de fazer parte do conjunto de vertices iniciais;
  - B(v) ⊆ Ci – o conjunto de cabazes para entrega tem de fazer parte do conjunto de cabazes inicial;
  - cap(v) = $\sum_{k=0}^{|B|-1} packageNumber(B[k])$ – a capacidade utilizada de um veículo tem de ser igual ao número de embalagens totais em cabazes;
  - Capacidade utilizada do veículo tem ser menor ou igual à capacidade disponível do veículo.
- T[0] ∈ adj(S) ∧ dest(T[|T| - 1]) = S – o trajeto de todos os veículos começa e acaba na sede.

2.4. Funções objetivo

O objetivo do trabalho é encontrar o mínimo número de veículos para entregar as encomendas e o menor trajeto para cada veículo possível. Para tal é necessária a minimização de duas funções, sendo que uma se refere ao menor número de veículos e a outra ao menor trajeto possível:

- f = |Vf|
- g = $\sum_{v \in Vf} \sum_{e \in T} weight(e)$

Numa fase inicial devido à consideração de uma capacidade infinita de veículos, a minimização da função f é desnecessária, sendo que na fase seguinte se prioriza a sua minimização à função g.

3. Perspetiva de Solução

3.1. Primeira fase

Inicialmente vai ser considerada a existência de apenas um veículo com capacidade infinita para a distribuição das encomendas, sendo então apenas necessário encontrar o trajeto ótimo para o mesmo. Para tal, seguem-se as próximas etapas:

1. Remoção das arestas indesejáveis, que vão ser as inutilizáveis pelos veículos.
2. Pré-processamento em que se calculam as distâncias entre todos os pares de pontos.
3. Verificar se os pontos de interesse (Sede e clientes) pertencem a uma componente fortemente conexa.
4. Ordenação dos pontos de interesse.
5. Calcular a sequência de vértices a percorrer no trajeto.

3.2. Segunda fase

Numa segunda fase, já vão ser considerados vários veículos de diferentes tipos e capacidades, pelo que é necessária uma distribuição ótima das encomendas pelos veículos disponíveis antes do cálculo do trajeto ótimo. Para tal, são necessárias as seguintes etapas:

1. Remoção das arestas indesejáveis, que vão ser as inutilizáveis pelos veículos.
2. Pré-processamento em que se calculam as distâncias entre todos os pares de pontos.
3. Distribuir as encomendas pelos veículos disponíveis.
4. Verificar se os pontos de interesse (Sede e clientes) pertencem a uma componente fortemente conexa.
5. Ordenação de pontos de interesse para cada veículo, considerando os pontos de entrega específicos do trajeto.
6. Calcular a sequência de vértices a percorrer no trajeto de cada veículo.

3.3. Remoção das arestas indesejáveis

A remoção vai consistir em encontrar as arestas inutilizáveis por veículos, atribuindo ao seu peso o valor de infinito, utilizando uma pesquisa em profundidade, que ao percorrer todos os vértices os assinala como visitados e nos permite no final encontrar aqueles que tal não é possível.

Eficiência temporal: O(|N| + |E|); Eficiência espacial: O(|N|)

3.4. Distribuição das encomendas

Esta etapa vai ser realizado utilizando o algoritmo “First Fit Decreasing”[1], que ordena a sequência de cabazes por número de embalagens crescente e atribui sucessivamente a veículos, já ordenados pela sua capacidade, até que não existam mais cabazes para distribuição.

Eficiência temporal: O(|C| * log(|C|)); Eficiência espacial: O(|C|); Sendo |C| o número de cabazes;

3.5. Pré-processamento

Nesta etapa considerou-se a utilização de 2 diferentes algoritmos dependendo da densidade do grafo utilizado. Para um grafo esparso, utilizar-se-ia o algoritmo de Dijkstra repetidamente de forma a se obterem as distâncias entre cada par de ponto, enquanto que para um grafo denso, utilizar-se-ia o algoritmo de Floyd-Warshall.

Neste caso, considera-se que o grafo a utilizar da cidade metropolitana do Porto é um grafo denso e, por isso, utiliza-se o algoritmo de Floyd-Warshall.

Algorithm 1 Floyd-Warshall with path reconstruction

```
1:  dist ← |V| × |V| length array of minimum distances initialized to ∞
2:  next ← |V| × |V| length array of vertex indices initialized to null
3:  procedure FLOYD-WARSHALL(PATH RECONSTRUCTION)
4:      for each edge(u, v) do
5:          dist[u][v] ← w(u, v)
6:          next[u][v] ← v
7:      for k = 1 to |V| do
8:          for i = 1 to |V| do
9:              for j = 1 to |V| do
10:                 if dist[u][v] > dist[i][k] + dist[k][j] then
11:                     dist[u][v] ← dist[i][k] + dist[k][j]
12:                     next[i][j] ← next[i][k]
13: procedure GETPATH(u, v)
14:     if next[u][v] = null then
15:         return []
16:     path = [u]
17:     while u ≠ v do
18:         u ← next[u][v]
19:         path.append(u)
20:     return path
```

Eficiência temporal: $O(|N|^3)$; Eficiência espacial: $O(|N|^2)$;

3.6. Verificar se os pontos pertencem a um CFC

Este passo vai ser realizado utilizando o método lecionado na cadeira:

- Pesquisa em profundidade no grafo G determina floresta de expansão, numerando vértices em pós-ordem;
- Inverter todas as arestas de G;
- Segunda pesquisa em profundidade, em Gr, começando sempre pelo vértice de numeração mais alta ainda não visitado;
- Cada árvore obtida é um componente fortemente conexo.
- Percorrer cada CFC e verificar se têm todos os vértices do trajeto (Sede e clientes), sendo que se nenhum CFC for encontrado não vai existir um trajeto possível.

3.7. Ordenação dos pontos de interesse do trajeto

Para tal, utiliza-se o algoritmo de Nearest Neighbour[2] que, para cada vértice se calcula o vértice que se encontra a menor distância sucessivamente até se encontrar o trajeto final.

Este algoritmo é um método de construção heurística que apresenta um caminho aproximadamente 25% mais longo do que o caminho de menor distância exato. Tal facto é contrariado pelo facto de ser um algoritmo com boa eficiência temporal quando comparado com métodos exatos.

Algorithm 2 TSP by Nearest Neighbour

```
1: V ← vertices representing all destinations in the route
2: procedure NEAREST NEIGHBOUR(Vertex P)
3:     sortedVertices ← V \ {P}
4:     result = [P]
5:     while |sortedVertices| > 0 do
6:         sortedVertices.sortRelativeTo(P)
7:         P ← sortedVertices[0]
8:         sortedVertices.remove(P)
9:         result.append(P)
10:    result.append(result[0])
11:    return result
```

Eficiência temporal: O(|A^2|); Eficiência espacial: O(|A|); Sendo |A| o número de clientes

3.8. Calcular a sequência de vértices a percorrer

De forma a calcular esta sequência, utiliza-se sucessivamente um algoritmo, GetPath mencionado no algoritmo 1, para encontrar o caminho entre dois pontos sucessivamente entre cada ponto sucessivo dos vértices principais utilizando a array de vértices “next” calculada no algoritmo de Floyd-Warshall.

Eficiência temporal: O(|T|); Eficiência Espacial: O(|T|); Sendo |T| o número de vértices a percorrer no trajeto.

4. Conclusão

Com a preparação necessária para a elaboração deste relatório e consequente planeamento do projeto, adquirimos uma melhor compreensão do uso de grafos e algoritmos relacionados com os mesmos.

O trabalho foi dividido igualmente pelos 3 elementos do grupo.

5. Bibliografia

[1] Dósa, György. (2007). The tight bound of first fit decreasing bin-packing algorithm is FFD(I) ≤ 11/9 OPT(I) + 6/9. Lect Notes Comput Sci. 4614. 1-11. 10.1007/978-3-540-74450-4_1.

[2] Johnson, D. S.; McGeoch, L. A. (1997). “The Traveling Salesman Problem: A Case Study in Local Optimization” (PDF). In Aarts, E. H. L.; Lenstra, J.K. *Local Search In Combinatorial Optimization*. London: John Wiley and Sons Ltd. pp. 215-310