U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

MarketOnWheels: Supermercado Ambulante

Tema 3

Concepção e Análise de Algoritmos 2020/2021

2MIEIC07_G3

Adriano Soares up201904873@up.pt
Francisco Cerqueira up201905337@up.pt
Vasco Alves up201808031@up.pt

Índice

Descrição do Problema

Uma empresa pretende implementar um sistema de entregas de compras ao domicílio, tendo como objetivo minimizar a distância percorrida por cada veículo de entrega desde a sede da empresa até aos centros de distribuição de produtos e novo regresso à sede, passando pelos pontos de entrega. Decidimos, então, dividir o problema em duas fases.

1ª Fase: Um veículo de capacidade ilimitada

Na 1ª fase, é utilizado apenas um veículo para todas as encomendas, considerando que o mesmo tem capacidade ilimitada. Apenas iremos calcular a menor rota passando pelos diversos fornecedores de produtos e clientes.

É importante referir que previamente devemos verificar a conectividade do grafo, de modo a avaliar o caráter fortemente conexo que este deve possuir.

2ª Fase: Vários veículos de capacidade limitada

Na 2ª fase é utilizada uma frota de veículos para a distribuição das diversas encomendas, tendo cada veículo um limite de capacidade. O procedimento será distribuir as diversas encomendas pelos vários veículos, maximizando a capacidade de encomendas por veículo, de forma a reduzir o espaço não reservado / desprezado e o número de veículos em utilização.

Após este pré-processamento, conseguimos reduzir o problema ao problema imposto na primeira fase, na qual cada veículo deverá minimizar a respetiva rota.

Formalização do Problema

Dados de Entrada

V[] - sequência de veículos na central, sendo V[i] o seu i-ésimo elemento, que é caracterizado por:

- Capacidade máxima do veículo (na 1ª fase do problema será ∞).

E[] - sequência de encomendas, sendo E[i] o seu i-ésimo elemento. Cada uma é caracterizada por:

- Sequência de produtos e respetivas quantidades;
- Morada do cliente que realizou a encomenda.

F[] - sequência de fornecedores, sendo F[i] o seu i-ésimo elemento. Cada um é caracterizado por:

- Sequência de produtos e respetivos stocks;
- Morada.

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

- V - vértices representativos de pontos de uma certa cidade:
  - ID;
  - tag - Identificador do tipo de vértice (Fornecedor, Cliente ou nenhum);
  - outgoing $\subseteq$ E.
- E - arestas representativas de vias de comunicação:
  - ID;
  - W - peso da aresta, a distância entre dois vértices;
  - dest $\in$ Vi - vértice de destino.

S $\in$ Vi - vértice inicial (Sede da Empresa)

Dados de Saída

Gf = (Vf, Ef) - grafo dirigido pesado, tendo Vf e Ef os mesmos atributos que Vi e Ei.

R[] - sequência ordenada de veículos usados, sendo R[i] o i-ésimo elemento. Cada elemento é caracterizado por:

- Ev[] - sequência de encomendas atribuídas ao veículo;
- P[] - percurso, sequência ordenada de arestas a visitar.

Restrições

Sobre dados de Entrada

- $\forall$ v $\in$ V, capacidade > 0;
- $\forall$ e $\in$ E, quantidade de produtos > 0 $\cap$ morada $\in$ Vi;
- $\forall$ f $\in$ F, morada $\in$ Vi;
- $\forall$ v $\in$ Vi, ID $\geq$ 0 $\cap$ tag $\geq$ 0 $\cap$ vértices *outgoing* > 0 (Fortemente Conexo);
- $\forall$ e $\in$ Ei, ID $\geq$ 0 $\cap$ W $\geq$ 0 $\cap$ dest $\in$ Vi.

Sobre dados de Saída

- No grafo Gf:
  - $\forall$ vf $\in$ Vf, $\exists$ vi $\in$ Vi tal que *vi* e *vf* têm os mesmos valores para todos os atributos
  - $\forall$ ef $\in$ Ef, $\exists$ ei $\in$ Ei tal que *ei* e *ef* têm os mesmos valores para todos os atributos.
- |R| $\leq$ |V|, não se podem utilizar mais veículos do que aqueles que estão disponíveis;
- $(\sum_{v \in R} |Ev|) \leq |E|$, o número total de encomendas distribuídas pelos veículos não pode ser superior ao número de encomendas inicial. Pode não ser possível entregar todas as encomendas devido às limitações de veículos ou stock.
- P[0] $\in$ S.outgoing, o veículo tem que partir da sede da empresa;
- $\exists$ v $\in$ R, $\exists$ e $\in$ Ev, P[|P|-1].dest = S, o veículo termina o percurso na sede;
- $\forall$ e $\in$ P, $\exists$ ei $\in$ Ei tal que *e* e *ei* têm os mesmos valores para todos os atributos.

Funções Objetivo

Pretende-se otimizar o problema, minimizando o número de veículos utilizados e a distância total percorrida por todos eles, conseguindo entregar todas as encomendas. Assim, a otimização do problema passa por minimizar as seguintes funções:

1. |R|, o número de veículos utilizado, maximizando o número de encomendas que pode levar;

2. $\sum_{v \in R} \sum_{e \in v.P} e.dist$, a distância percorrida total.

Perspetiva de Solução

Preparação dos ficheiros de entrada

Nesta primeira fase, são lidos os ficheiros de nodes e edges fornecidos pelos professores, povoando o grafo por nós definido na secção de dados de entrada e os ficheiros que contém outras informações relevantes, como veículos, encomendas, clientes e fornecedores. Os nós contêm apontadores para *Cliente* e *Fornecedor* de forma a facilitar a identificação dos mesmos sem percorrer as respetivas sequências fornecidas. Cada nó contém uma sequência de arestas *incoming* e *outgoing*, de modo a facilitar o processo de inversão de grafo, que foi necessária na verificação do carácter fortemente conexo do grafo.

Análise da conectividade do grafo

De modo a garantir a existência de um caminho de regresso para cada percurso calculado, eliminando a possibilidade de inviabilizar o algoritmo por nós utilizado, o grafo tem que ter uma componente fortemente conexa. Assim, utilizamos o método fornecido nas aulas teóricas[1] aplicado a um grafo dirigido pesado:

1. Pesquisa em profundidade no grafo G para determinar a floresta de expansão, numerando vértices em pós-ordem;
2. Inverter todas as arestas de G, resultando num grafo Gr;
3. Segunda pesquisa em profundidade, em Gr, começando sempre pelo vértice de numeração mais alta ainda não visitado;
4. Cada árvore obtida é um componente fortemente conexo, i.e., a partir de um qualquer dos nós pode chegar-se a todos os outros.

Após o cálculo das componentes fortemente conexas do grafo inicial, utilizamos apenas aquela que contém a sede de veículos da empresa (podendo esta ser única no caso óptimo), filtrando os fornecedores e clientes/encomendas contidas na mesma.

```
DFS(g):    // Algoritmo de Pesquisa em Profundidade
    res <- empty
    for each v ∈ V
        visited(v) <- false
    for each v ∈ V
        if not visited(v)
            DFS-VISIT(v, res)
    return res

DFS-VISIT(v, res):
    visited(v) <- true
    for each w ∈ outgoing(v)
        if not visited(w)
            DFS-VISIT(w, res)
    INSERT(res, v)

DFS-VISIT-REVERSE(v, res):
    visited(v) <- true
    for each w ∈ ingoing(v)
        if not visited(w)
            DFS-VISIT(w, res)
    INSERT(res, v)

analyzeGraphConnectivity(g, centerID):
        forest <- DFS(g)
        for each v ∈ V
            visited(v) <- false
            strong(v) <- false
        res <- empty
        for each v ∈ forest
            if not visited(v)
                res <- clear
                DFS-VISIT-REVERSE(v, res)
                if vertex(centerID) in res then
                    break
        for each v ∈ res
            strong(v) <- true
```

Técnicas de Conceção de Algoritmos

Utilizamos o algoritmo **Nearest Neighbor**: a partir da sede da empresa, priorizamos a pesquisa por fornecedores viáveis, podendo, durante o percurso, encontrar clientes cuja encomenda esteja concluída, caso possua todos os produtos necessários. A cada ponto de interesse viável encontrado, *Fornecedores* e *Clientes*, restauramos o grafo para o seu estado inicial, permitindo a busca em sentidos já visitados anteriormente. O algoritmo termina quando todas as encomendas possíveis forem entregues.

```
nearestNeighbor(g, V, E, F, s):
PATH <- empty

while not deliverAllOrders do

   for each v ∈ V do
      dist(v) <- inf
      path(v) <- NULL
      visited(v) <- false

   dist(s) <- 0
   Q <- empty
   INSERT(Q, (s, 0))

   while Q not empty do
      v <- EXTRACT-MIN(Q)

      if isViableProvider(v) then
         supplyProducts()
         savePath()
         s = v
         break
      if isViableClient(v) then
         deliverProducts()
         savePath()
         s = v
         break

      for each w ∈ outgoing(v) do
         if dist(w) > dist(v) + weight(v,w) then
            dist(w) <- dist(v) + weight(v,w)
            path(w) <- v
            if !visited(dest(w)) then
               visited(dest(w)) <- true
               INSERT(Q, (w, dest(w))
            else
               DECREASE-KEY(Q, (w, dest(w))
  return PATH
```

```
// Pre-processing
analyzeGraphConnectivity()
filterValidOrders() // Ignore orders with insufficient stock
distributeValidOrdersToVehicles()

for each vehicle ∈ vehicles do
   nearestNeighbor()
```

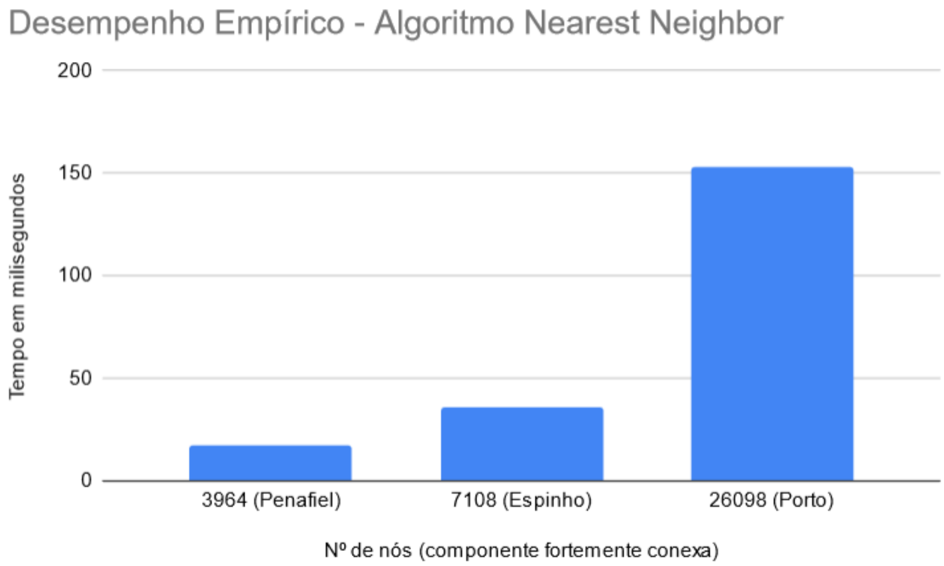
Para o regresso dos veículos à sede da empresa, ponderamos utilizar o algoritmo de Dijkstra Bidirecional, dado que sabemos o ponto inicial, o último cliente, e o ponto final, a sede da empresa. No entanto, não fomos capazes de o implementar e acabamos por utilizar o algoritmo Dijkstra.

```
bidirectionalDijkstraShortestPath(g, s, t):
    PATH <- empty
    for each v ∈ V do
       dist(v) <- inf
       path(v) <- NULL
       visited(v) <- false
       visitedReverse(v) <- false
    dist(s) <- 0
    dist(t) <- 0
    Q <- empty
    Qr <- empty
    INSERT(Q, (s, 0))
    INSERT(Qr, (t, 0))
    done <- false

    while Q not empty and Qr not empty and not done do
       v <- EXTRACT-MIN(Q)
       u <- EXTRACT-MIN(Qr)
       for each w ∈ outgoing(v) do
          if dist(w) > dist(v) + weight(v,w) then
              dist(w) <- dist(v) + weight(v,w)
              path(w) <- w
              if visitedReverse(dest(w)) then
                 done <- true
              else if !visited(dest(w)) then
                 visited(dest(w)) <- true
                 INSERT(Q, (w, dest(w))
              else
                 DECREASE-KEY(Q, (w, dest(w))
       for each w ∈ incoming(u) do
          if dist(w) > dist(u) + weight(u,w) then
              dist(w) <- dist(u) + weight(u,w)
              path(w) <- w
              if visited(orig(w)) then
                 done <- true
              else if !visitedReverse(orig(w)) then
                 visitedReverse(orig(w)) <- true
                 INSERT(Qr, (w, dest(w))
              else
                 DECREASE-KEY(Qr, (w, dest(w))

    savePath(t, path)
    return PATH
```

Com uma avaliação preliminar, podemos deduzir que a complexidade temporal seja $O((|E| + |F|)(|Ei| + |Vi|) \, log(|Vi|))$, pois é visível que o algoritmo é proporcional ao número de vezes que o algoritmo é restaurado, que pode ser aproximado pela soma do número de fornecedores e de encomendas $(|E| + |F|)$, e à complexidade do algoritmo de Dijkstra $((Ei + Vi) \, log(Vi))$. Dado que as encomendas são previamente distribuídas pelos veículos, o primeiro ciclo torna-se insignificante no cálculo desta complexidade. Foi possível observar os seguintes resultados obtidos através de métodos empíricos:



Quanto à complexidade espacial, o algoritmo necessita de armazenar espaço para o grafo $O(|Ei| + |Vi|)$ e para o caminho ótimo, como no pior dos casos a algoritmo utilizaria $O(|Ei||)$ a cada vez que o algoritmo reinicia e como foi dito acima que o número de vezes que o algoritmo é restaurado pode ser aproximado pela soma do número de fornecedores e de encomendas $(|E| + |F|)$, a complexidade espacial seria $O((|E| + |F|)(|Ei|))$, resultando numa complexidade espacial $O(|Ei| + |Vi| + (|E| + |F|)(|Ei|))$. Como as redes de transportes se aproximam de um grafo esparso, onde $|E| \sim \Theta(V)$, podemos aproximar a complexidade espacial a $O((|E| + |F|)(|Vi|))$.

É de notar que este algoritmo, apesar de ser mais eficiente que força bruta, de complexidade temporal fatorial, pode não retornar o caminho mais curto, sendo provável que retorne, em média, um caminho 25% maior do que o caminho ótimo[2]. A utilização de força bruta não é uma opção devido à quantidade de nós no grafo que nos irá ser fornecido.

Temos conhecimento da existência de um algoritmo potencialmente mais eficiente, A*, apresentado brevemente num dos slides das aulas teóricas.

No entanto, não possuímos o conhecimento necessário para a implementação deste. Contudo, a utilização de um pré-processamento aplicado aos pesos das arestas do grafo inicial, $w'_{uv} = w_{uv} - \pi_{ut} + \pi_{vt}$, aplicando o algoritmo de Dijkstra, é equivalente à utilização deste mesmo algoritmo, A*[3]. Dado que este pré-processamento depende tanto do vértice inicial como do final, e não possuímos conhecimento do nó final no algoritmo por nós aplicado ao pesquisarmos por *Fornecedores* e *Clientes*, não o utilizamos neste procedimento.

Estrutura do Código

Optamos por estruturar o código com os seguintes módulos:

- **graph**: Contém todos os ficheiros relacionados com grafos e os algoritmos principais aplicados nestes, bem como uma classe com o intuito de armazenar um caminho. [*Graph.h, MutablePriorityQueue.h, Node.h, Path.h*]
- **gui**: É responsável por utilizar a biblioteca GraphViewer, de modo a mostrar o grafo e os caminhos. [*GUI.h*]
- **market**: Todos os modelos necessários na implementação do programa estão incluídos neste módulo. [*Client.h, Order.h, Product.h, Stock.h, Supplier.h, Vehicle.h*]
- **menu**: Contém os ficheiros que são responsáveis pela gestão e implementação dos menus utilizados no programa. [*Menu.h*]
- **util**: Módulo onde estão incluídos os ficheiros com utilidades necessárias para os outros módulos, como receção e verificação de input e funções para calcular distâncias e entregar e recolher produtos. [*Input.h, Utils.h*]
- **application**: Classe responsável por ler, armazenar e tratar maior parte da informação, ligando quase todos os módulos anteriormente apresentados. [*Application.h*]

Casos de Utilização

O programa desenvolvido permite ao utilizador:

- Selecionar o nó da sede da empresa;
- Visualizar o mapa completo através do GraphViewer;
- Calcular o caminho ótimo tendo em conta os veículos inseridos no sistema, que será apresentado ao utilizador através do GraphViewer, podendo optar por selecionar um veículo com capacidade infinita ou vários veículos de capacidade limitada;
- Verificar a conectividade do grafo a partir da sede da empresa;
- Medir o desempenho do algoritmo aplicado;
- Navegar entre os mapas pré-definidos pela aplicação (Porto, Espinho, Penafiel, Grid, ...).

Dificuldades Sentidas

Não fomos capazes de implementar o algoritmos de Dijkstra Bidirecional. Criamos também um script para corrigir a estrutura dos ficheiros que contêm os nós e as arestas dos grafos grid fornecidos (os índices destes eram iniciados a 0, ao invés de 1, como é observado nos restantes mapas).

Conclusão

Este trabalho estimulou-nos a recorrer a grafos e a algoritmos de pesquisa nos mesmos, convertendo em prática os conteúdos lecionados. Concluímos que alcançamos os objetivos pretendidos, quer a nível individual, como coletivo, uma vez que cada elemento do grupo domina os temas doutrinados.

Distribuição de Tarefas

Todos os membros do grupo esforçaram-se igualmente na estratificação e resolução do problema.

[1] R. Rossetti, A. P. Rocha, L. Ferreira, J. P. Fernandes, F. Ramos, G. Leão (2021) *Algoritmos em Grafos: Conectividade*. Porto: FEUP, https://moodle.up.pt/pluginfile.php/164086/mod_label/intro/12.grafos7.pdf?time=1617281195155 [10 de abril de 2021].

[2] David S. Johnson, Lyle A. McGeoch (1995) *The Traveling Salesman Problem: A Case Study in Local Optimization*, https://www.cs.ubc.ca/~hutter/previous-earg/EmpAlgReadingGroup/TSP-JohMcg97.pdf [10 de abril de 2021].

[3] R. Rossetti, A. P. Rocha, L. Ferreira, J. P. Fernandes, F. Ramos, G. Leão (2021) *Algoritmos em Grafos: Caminho mais curto (Parte II)*. Porto: FEUP, https://moodle.up.pt/pluginfile.php/197873/mod_label/intro/09.grafos4.pdf [10 de abril de 2021].