24/04/2020

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

CONCEPÇÃO E ANÁLISE DE ALGORITMOS

2019/2020

MEAT WAGONS

TRANSPORTE DE PRISIONEIROS

T6 | GRUPO 6

TRABALHO REALIZADO POR:

BERNARDO RAMALHO – UP201704334

FRANCISCO GONÇALVES – UP201704790

MARTIM SILVA – UP201705205

ÍNDICE

DESCRIÇÃO DO PROBLEMA

Os transportes de prisioneiros entre prisões, esquadras e tribunais são feitos usando carrinhas e camionetas adaptadas para o serviço. A empresa MeatWagons é uma empresa de segurança privada que realiza os transportes de prisioneiros entre quaisquer pontos de interesse definidos (prisões, esquadras e tribunais), disponibilizando um certo número de carrinhas para esse efeito.

Neste trabalho, pretende-se implementar um serviço, no qual sejam feitos um conjunto de pedidos de receção e entrega de prisioneiros a diversos pontos de interesse. Estes pedidos devem ser agrupados de forma a garantir otimização do percurso das carrinhas. O problema será decomposto em três iterações.

Tal como foi discutido com o professor, os prisioneiros estarão em locais distintos, mas assim que recolhidos irão todos para o mesmo local. Todas as carrinhas partem de uma central, fazem a receção e entrega dos diferentes prisioneiros e voltam sempre à central no final da entrega.

1ª Iteração - Receber diversos pedidos utilizando 1 carrinha de capacidade = 1

Esta iteração servirá para testar os algoritmos do caminho mais curto entre 2 pontos. Nesta iteração iremos implementar todos os algoritmos (e suas melhorias) dos algoritmos do caminho mais curto. Como a carrinha é de capacidade 1, nesta fase ainda não haverá a preocupação de agrupar pedidos de diferentes prisioneiros. É importante referir que, cada pedido só será realizado, se for determinado que existe um caminho de ida e de volta, visto que a carrinha precisará de regressar à sua central. Além disso, poderá haver obras nas vias públicas e zonas inacessíveis, por isso, estas arestas também devem ser ignoradas no processamento. A ordem de entrega, nesta fase, será por ordem dos tempos de receção dos prisioneiros.

2ª Iteração - Receber diversos pedidos utilizando 1 carrinha de capacidade > 1

Nesta fase, devem ser agrupados os pedidos para garantir que a mínima distância possível seja percorrida e que o mínimo de carrinhas sejam utilizadas. O objetivo será então ter sempre que possível as carrinhas cheias, de modo a maximizar o potencial lucro da empresa. Assim, a carrinha deverá passar por uma série de diferentes pontos de recolha, entregar os prisioneiros no mesmo local e depois, regressar à central. Isto garante que seja alcançado o número mínimo de viagens possíveis.

3ª Iteração - Receber diversos pedidos utilizando várias carrinhas de diferentes capacidades

Na 3ª iteração, é usada uma frota de carrinhas de transporte de prisioneiros, com diversas capacidades. Isto implica diversas deslocações e diferentes combinações possíveis de agrupamentos de pedidos de entrega de prisioneiros, O sistema deve otimizar o agrupamento dos pedidos. Qualquer pedido, depois de concluído, deve voltar à central. Para o agrupamento dos pedidos deverão ser consideradas as zonas da localização de cada pedido.

FORMALIZAÇÃO DO PROBLEMA

Dados de entrada

- Gi = (V, E) → Grafo dirigido pesado constituído por:
    - V → Conjunto de vértices que representam pontos de recolha e entrega de prisioneiros. Cada vértice, por sua vez, é constituído por:
        - id → identificador do vértice
        - position → vetor que contém as coordenadas geográficas do local;
        - type → indica que tipo de local o vértice representa;
        - edges ⊆ E → subconjunto de E. Todos os caminhos que saem do vértice.
        - outros dados, utilizados nos diferentes algoritmos (valor heurístico, índice da file de prioridade mutável, *path*, *path* invertido, ...)
    - E → Conjunto de arestas, cada uma representando as estradas que ligam cada ponto de recolha e entrega. Cada aresta, por sua vez, é constituída por:
        - id → identificador da aresta
        - dest ∈ V → apontador para o vértice de destino;
        - peso → peso da aresta
- P → Lista de prisioneiros que têm de ser entregues/recolhidos. Cada prisioneiro P(i) tem a seguinte informação:
    - PR ∈ V → Vértice que contém a informação do sítio de recolha;
    - time → Tempo para recolha do prisioneiro
- C → Lista de carrinhas que existem e podem ser usadas. Cada carrinha C(i) tem a seguinte informação:

- cap→ Indica o número total de passageiros que a carrinha consegue levar;

- D → Representa a central da empresa MeatWagons. Esta central tem a seguinte informação:
  - position → vetor que contem as coordenadas geográficas da central;
- M → Distância máxima entre dois pontos de interesse.
- I → Conjunto de vértices correspondentes aos pontos de interesse (tribunais, prisões, esquadras)

Dados de saída

- C → Lista de carrinhas que irão ser utilizadas para todos os trajetos. Cada carrinha (Cf) tem a seguinte informação:
  - Capacity → Indica o número total de passageiros que a carrinha consegue levar;
  - Pf ⊆ P → Lista de prisioneiros que viajam na carrinha em qualquer altura;
  - Va ⊆ V → Lista ordenada dos vértices pelos quais a carrinha passa na ida.
  - Vb ⊆ V → Lista ordenada dos vértices pelos quais a carrinha passa no regresso.
- Gf = (V, E) → O mesmo grafo dirigido pesado recebido no input

Restrições

Nos dados de entrada:

- ∀ V → Type(V) == “Tribunal” ∨ “Esquadra” ∨ “Prisão”;
- ∀ E → dist(E) > 0 (isto é, não há distâncias negativas);
- ∀ C → Capacity(C) > 0 (Não existem capacidades negativas);
- M > 0

Nos dados de saída:

- Gf = Gi, isto é, o grafo deverá ser igual ao fornecido no input;
- Ci >= Cf, isto é, no máximo pode-se utilizar o número de carrinhas disponíveis;
- ∀ c ∈ Cf: Capacity(c) <= P(c), isto é, o número de prisioneiros numa carrinha não pode ser maior que a sua capacidade;

- ∀ Cf → Capacity(Cf) > 0 (Não existem capacidades negativas);
- | Pf | <= | P | ;
- | Cf | <= | C | ;
- ∀c ∈ Cf:
    - Va(1) = D, isto é, todas as viagens começam na central;
    - Va(n) = I, isto é, no caminho de ida, a carrinha termina num ponto de interesse;
    - Vb(1) = Va(n), a carrinha começa o trajeto de ida depois da entrega;
    - Vb(n) = Va(1) = D, a carrinha acaba a viagem chegando à central

Função objetivo

A nossa solução ótima é aquela que minimizar o número de lugares vazios por carrinha, minimizando assim o número de carrinhas usadas. Também se pretende minimizar a distância total percorrida. Isto é pretende-se minimizar as seguintes funções (primeiro f, depois g):

- f = | CF |, isto é, minimizar o número de veículos utilizados.
- g = ∑c ∈ C (∑v ∈ Va dist(Vn, Vn + 1) + ∑v ∈ Vb dist(Vn, Vn + 1)), isto é, minimizar a distância total percorrida (distância de ida e de volta)

PERSPETIVA DE SOLUÇÃO

Pré-processamento

Pré Processamentos:

O grafo deve ser sujeito a um pré processamento ao nível do grafo, dos dados e dos percursos de duração mínima. Isto aumenta a eficiência temporal ao aplicar os algoritmos descritos abaixo e no decorrer do programa.

Pré-processamento do grafo:

Devem ser eliminadas as arestas que representam estradas onde estejam a decorrer obras nas vias públicas (zonas inacessíveis). Além disso, é necessário avaliar a conectividade do grafo, a fim de identificar pontos de recolha e de entrega com pouca acessibilidade. Assim, todos os vértices, incluídos pontos de interesse, que não pertencem ao mesmo componente fortemente conexo da central, devem ser removidos. Por último, podemos remover todos os vértices que não correspondem a pontos de interesse, ou que não correspondam a vértices que pertencem aos caminhos mais curtos dos pontos de interesse.

Pré-processamento dos dados:

Devem ser retirados todos os pedidos em que os vértices de receção dos prisioneiros não estejam contidos no grafo pré processado. Além disso, todos os pedidos que excedam o limite para cada veículo devem ser subdivididos em pedidos que respeitem este limite, retirando-se também o pedido "ilegal" original. Por último, os pedidos devem ser ordenados pelo tempo definido para recolha dos prisioneiros.

Pré-processamento dos percursos de duração mínima:

Será necessário um algoritmo que permita saber a distância entre quaisquer pares de pontos de interesse, assim como o caminho entre eles. Tanto o algoritmo de Dijkstra como o de Floyd–Warshall podem ser utilizados nesta etapa. Os pontos de interesse considerados são os tribunais, esquadras e prisões. Teoricamente o algoritmo de Floyd–Warshall é melhor tanto no caso de o grafo ser denso, como se não o for, no entanto isso terá de ser testado ao desenvolver o projeto, e por isso optaremos mais tarde na escolha de um dos dois algoritmos.

Este pré-cálculo aumentaria, num elevado grau, a eficiência do programa, ao evitar ter de se calcular a distância mínima entre os vários pontos para cada novo percurso. Como é um cálculo muito custoso, pode representar uma menos-valia, no entanto também será avaliado o benefício do mesmo em termos práticos, com recurso a análises temporais.

Se o cálculo for muito exigente, será testado usar memória não volátil para guardar este pré-processamento, que seria feito apenas 1 vez e aumentar assim consideravelmente a eficiência e eficácia do algoritmo. Mesmo assim, sempre que o grafo fosse alterado, seria necessário recalcular com o algoritmo de Floyd–Warshall.

Inicialmente será testado o algoritmo de Dijkstra, e será guardado o caminho mais curto entre todos os pares de pontos de interesse. Será guardado também a distância total. Ambas as informações serão guardadas numa matriz quadrada.

Algoritmos

Algoritmo de Dijkstra

Este algoritmo será usado no cálculo do caminho mais curto entre todos os pares de vértices e entre dois vértices (*single source* e *all pairs*). É um algoritmo ganancioso usado nos casos de grafos dirigidos com pesos, para encontrar o caminho mais curto.

Cada vértice terá de guardar a duração mínima até ao nó de origem assim como o caminho, isto é, o vértice anterior pertencente ao caminho mais curto.

O algoritmo subdivide-se em preparação e pesquisa.

Iremos tentar aplicar várias melhorias a este algoritmo, descritas abaixo.

Este algoritmo foi escolhido devido à facilidade da sua implementação, estudado nas aulas práticas e teóricas, e com algumas otimizações pode ser muito eficiente e prático.

Em termos de eficiência, a fase de preparação tem O(|V|) e a fase da pesquisa O((|V|+|E|)*log(|V|)).

Na parte de preparação, é atribuído um valor da distância a cada vértice de infinito e o caminho mínimo a ser percorrido (πf) como null. Coloca-se também que a distância do vértice inicial a 0 (vértice de começo).

Para além disto, coloca-se o vértice de origem na fila de mínima prioridade. Na parte de pesquisa, é feito um ciclo que acaba até a fila de prioridade estar vazia. Primeiro, extrai-se o vértice com menor prioridade (vértice com menor distância). Em seguida, para cada vértice adjacente ao vértice extraído, verifica-se a distância atual até ele (inicialmente como infinito) é maior do que a distância usando o caminho do vértice extraído. Se o for, atualiza esse vértice com os valores do caminho menor, do novo valor da distância e a posição na fila de prioridade.

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)

RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2      v.d = u.d + w(u, v)
3      v.π = u

INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ G.V
2      v.d = ∞
3      v.π = NIL
4  s.d = 0
```

Single source, single destination:

A primeira otimização a ser feita, será no cálculo do caminho mais curto entre 2 pontos. Como o algoritmo tradicional de Dijkstra calcula o caminho mais curto a partir de um vértice inicial até todos os outros vértices, a melhoria a ser feita neste caso será de parar o ciclo quando o vértice extraído, da fila de prioridade, for igual ao vértice de destino pretendido.

Fila de prioridade dinâmica:

Nesta melhoria, o próximo vértice a processar continua a ser o da distância mínima (greedy), no entanto, não é necessariamente o mais antigo. Isto obriga a usar a fila de prioridade em vez de uma fila simples. Além disso, será necessário alterar as prioridades estabelecidas e por isso terá também de ser dinâmica. A cada passo do método, pode ser necessário realizar operações de extração, inserção ou decrease-key na fila de prioridade.

Pesquisa bidirecional:

Em caminhos curtos pode reduzir consideravelmente o tempo da pesquisa, e mesmo em caminhos mais longos a redução é sempre benéfica. Para isto, teremos de ter o grafo invertido, duas filas de prioridade, dois *arrays* de distâncias distintos e executar o algoritmo de Dijkstra nos dois sentidos, alternando consecutivamente entre os dois. Há um problema que se põe (*referência 1*) neste algoritmo. É que apesar de a condição de paragem ser quando um vértice é retirado das duas filas de prioridades este vértice pode não fazer parte do caminho mais curto. Para isso, assim que a condição

de paragem for estabelecida, teremos de percorrer ambas as filas de prioridade e ver qual dos caminhos leva ao caminho mais curto.

Pesquisa orientada:

Juntando ao que foi acima descrito, nesta etapa acrescenta-se uma função de heurística (função de avaliação), normalmente a distância euclidiana entre o vértice atual e o vértice de destino. Para aplicar esta melhoria, é preciso estar consciente do que é a desigualdade triangular. Esta função de avaliação evita que o algoritmo seja completamente ganancioso e através da heurística definida consegue "indicar" para que direção o algoritmo de Dijkstra deve prosseguir. O ganho conseguido ao usar este algoritmo é enorme e muito benéfico, principalmente nas zonas mais densas. Apesar disto, não consegue garantir exatamente que o caminho encontrado seja sempre o mais curto. A utilização, ou não, desta otimização será discutida na fase de implementação, com base a testes práticos.

Algoritmo de Floyd-Warshall:

Este algoritmo será usado no cálculo do caminho mais curto entre todos os pares de vértices. Utiliza conceitos de programação dinâmica. Baseia-se em matrizes de adjacências, uma com as distâncias mínimas e a outra matriz de predecessor no caminho mais curto (*path*).

Tem uma complexidade melhor (O(|V|^3)) ao algoritmo de Dijkstra, tanto se o grafo for denso, como se o grafo for pouco denso. Como este algoritmo exige um trabalho muito grande de pré-processamento será avaliado mais tarde se o iremos utilizar.

Estratégias de aplicação dos algoritmos

Cada uma das estratégias abaixo definidas corresponde a cada uma das iterações já previamente descritas. Relativamente aos pré-processamentos, todas devem fazer o mesmo para otimizar o programa. Uma vez que todas as iterações (exceto a primeira), muito similares no seu contexto, tratam do problema *Vehicle Routing Problem (VRP)*, uma generalização do problema *Travelling Salesman Problem (TSP)*, partilham o mesmo objetivo de minimizar a distância total percorrida.

Em todos os casos, o pré-processamento do grafo, resulta num grafo apenas com todos os vértices correspondentes a pontos de interesse, e vértices que correspondem aos caminhos mais curtos entre todos os pares de vértices de pontos de interesse.

Receber diversos pedidos utilizando 1 carrinha de capacidade 1

Na primeira iteração, com uma única carrinha, o problema reduz-se a encontrar o caminho mais curto entre a central, local de receber o prisioneiro e o local de destino final do passageiro.

Por último, regressar a central para ir buscar novo prisioneiro, pela ordem estabelecida de cada pedido.

Receber diversos pedidos utilizando 1 carrinha de capacidade maior que 1

Na segunda iteração, os pedidos passam a poder conter vários pontos para receber e apenas um para entregar prisioneiros, isto é, recolha de mais que um prisioneiro de diferentes locais e entrega no mesmo estabelecimento, usando o mesmo veículo. Nesta iteração estaremos a lidar com o problema do Caixeiro Viajante (*Travelling Salesman Problem*), uma vez que teremos de passar por todos os locais de interesse definidos com o menor custo total em termos de distância percorrida.

Nesta iteração a complexidade aumenta a um grau muito grande, por se tratar de um problema complexo (TSP). Temos soluções gananciosas como é o caso do *nearest neighbour.* Esta solução apesar de eficiente, não é exata. Baseia-se em escolher um vértice inicial aleatório e depois escolher o próximo vértice que for o mais perto do que escolheu inicialmente. Isto pode levar a problemas de sobreposição de caminhos o que faz não ser uma solução ótima (*referência 2*). Como alternativa seria utilizado o método de Held-Karp, brute force ou spanning trees. Estes algoritmos

teriam um custo muito elevado e serão explorados apenas no final. A solução inicial escolhida será então do *nearest neighbour.*

Receber diversos pedidos utilizando várias carrinhas de diferentes capacidades

Na terceira iteração há a possibilidade de ter um conjunto de veículos, de diferentes capacidades, e combiná-los com um conjunto de diversos pedidos. O algoritmo deve ser capaz de alocar o número ideal de prisioneiros com base nas capacidades dos veículos, o número de carrinhas para cada pedido (pode ser necessário subdividir pedidos se a capacidade máxima do maior veículo não for necessária) tendo em conta uma função de avaliação. Nesta fase o problema torna-se idêntico ao problema do *Vehicle Routing Problem (VRP)*, uma generalização do problema *Travelling Salesman Problem (TSP).*

Tendo carrinhas de diferentes capacidades, é necessário começar pelas carrinhas de maior capacidade para evitar desperdícios. Isto também mínima o número de veículos utilizados, reduzindo o custo da empresa. Para a escolha da carrinha a ser usada em seguinte foi necessário desenvolver um algoritmo ganancioso:

```
chooseWagon(n, W) // n = number of prisioners, W = available wagons
    sort(W) // sort by capacity (first the largest)
    w ← W.begin()
    while n > 0 & W.size > 0 do
        // until there is no space left in a wagon
        if n > w.capacity then
            n - w.capacity
            w -> U
            W.erase(w)
            w ← W.begin()
        // if there is a space left -> start by the smallest capacity wagon
        else
            w ← next(W)
            if w == W.end() || n > w.capacity then
                w ← retrieve(W)
                n - w.capacity
                w -> U
                W.erase(w)
```

De seguida, será aplicado o algoritmo para agrupar os melhores pedidos para cada capacidade de cada veículo e será aplicado o algoritmo do TSP para entregar todos os prisioneiros recolhidos no mesmo local.

Estratégias de agrupamentos de pedidos

A questão de agrupar os pedidos conforme as localizações é fulcral para a questão de otimizar e obter a máxima eficiência para os percursos. Estes agrupamentos devem ter como base o maior lucro da empresa, ou seja, maximizar a capacidade de cada carrinha para diminuir o número de viagens e diminuir a distância total percorrida. Neste caso, tratando-se de prisioneiros, os tempos de espera não serão o mais importante para a função objetivo, mas sim maximizar as capacidades das carrinhas em movimento.

As carrinhas partem da central completamente vazias. Sendo o objetivo preencher totalmente a capacidade de cada carrinha, serão procurados N (capacidade da carrinha) pedidos de receção de prisioneiros para preencher a carrinha com base nas prioridades e distâncias entre cada pedido.

Os pedidos serão ordenados com base nos tempos de recolha dos prisioneiros e com base na prioridade, podendo ser de prioridade 1, e nesse caso terão de ser obrigatoriamente preenchidos imediatamente. Pedidos com prioridade 2 não terão qualquer pressa adicional para a receção dos prisioneiros.

Depois de processados todos os pedidos de prioridade 1, o agrupamento entre os prisioneiros terá em conta a localização de cada prisioneiro e os tempos de recolha de cada passageiro. Deste modo, deve haver uma tentativa de agrupar os pedidos por zonas de receção. Para isso, será necessário estabelecer um valor máximo de tempo entre 2 pontos de recolha de prisioneiros. Este será o único caso onde as carrinhas poderão partir não cheias.

Análise da conectividade

Como foi mencionado na secção de perspetiva de solução e pré processamento dos grafos, deve ser feita uma remoção de arestas indisponíveis e vértices inalcançáveis, bem como uma análise de conectividade dos grafos. Nesta análise pretendemos distinguir o componente fortemente conexo (CFN) ao qual pertence a central, removendo vértices que não pertençam a este componente

Algoritmo Kosaraju

Para distinguir todos os CFN do grafo vamos utilizar o algoritmo de Kosaraju, que se traduz nos seguintes passos:

1. Fazer uma pesquisa em profundidade no grafo, numerando os vértices encontrados em pós ordem
2. Transpor o grafo (inverter o sentido de todas as arestas)
3. Realizar uma nova pesquisa em profundidade, dando prioridade aos vértices com índice maior, que estão ainda por visitar.

Após aplicar este algoritmo, todas as árvores resultantes representam CFNs. No entanto, para o nosso caso específico apenas é relevante o componente que contém a central, pelo que quando o CFN que contém este ponto for encontrado, podemos parar a execução do algoritmo, fazendo uma otimização relevante. Este método reduz-se a duas pesquisas em profundidade e também à transposição (inversão das arestas) do grafo, pelo que todo este algoritmo tem uma complexidade linear, já que cada uma destas operações tem uma complexidade O(V+E).

Algoritmo de Tarjan

Uma outra opção para determinar os CFNs dum grafo é o Algoritmo de Tarjan, que apesar de também ter uma complexidade linear, é mais eficiente, já que usa apenas uma pesquisa em profundidade.

Depth first search (DFS)

Perante a possibilidade de serem fornecidos mapas com grafos não-dirigidos vamos optar por implementar DFS, fazendo a pesquisa a partir da central e marcando os vértices como visitados ao longo do percurso. Resumidamente, o algoritmo:

1. Marca todos os vértices do grafo como não visitados;
2. Marca o vértice presente como visitado;
3. Caso o vértice de destino ainda não tenha sido visitado, visitá-lo recursivamente, fazendo uso do passo 2 e 3 na recursividade, para todas as arestas com origem no vértice presente.

O facto deste algoritmo ser recursivo garante que os vértices serão percorridos em profundidade ao invés de em largura, explorando cada ramo o mais fundo possível antes de realizar o backtracking. Como o vértice inicial será a central, apenas encontraremos vértices pertencentes ao mesmo CFN. A complexidade média deste algoritmo deve ser O(|V| + |E|), já que para marcar todos os vértices como não visitados o algoritmo tem de percorrer todos os vértices - |V| - e, já na fase de pesquisa, percorre todas as arestas de cada vértice encontrado - |E|.

CASOS DE UTILIZAÇÃO

A comunicação entre a aplicação e o utilizador será feita através de uma interface simples de texto. Esta interface será composta por um sistema de menus com várias opções, sendo as de mais destaque as seguintes:

- Importar um grafo através dum ficheiro de texto;
- Calcular o caminho mais curto entre quaisquer dois pontos de interesse;
- Calcular a rota ótima entre vários pontos;
- Mudar a posição da central;
- Adicionar, remover, alterar e listar pedidos;
- Listar veículos disponíveis na central e as respetivas capacidades;
- Adicionar e remover um veículo à central;
- Verificar se o grafo é fortemente conexo;
- Verificar conectividade entre todos os pontos de interesse.

CONCLUSÃO

Em geral, após análise detalhada deste problema, foi possível compreender com grande detalhe grande parte da matéria de grafos, nomeadamente cálculo do caminho mais curto, otimização do algoritmo de Dijkstra, possíveis resoluções para o problema do *Travelling Salesman Problem* e as suas aplicações ao *Vehicle Routing Problem*, nomeadamente a difícil escolha entre eficiência e soluções exatas.

Para além disso, o estudo dos algoritmos da análise da conectividade e do significado de componentes fortemente conexos e suas aplicações foi muito benéfica para uma compreensão mais geral dos problemas que nós, enquanto informáticos, teremos de enfrentar.

Cada membro do grupo dedicou um esforço mais ou menos similar a esta primeira entrega do projeto, tendo a divisão das tarefas sido feita da seguinte forma:

Martim Pinto da Silva

- Descrição do problema (relatório)
- Pré-processamentos (relatório)
- Algoritmo de Dijkstra (e variantes) (relatório)
- Algoritmo de Floyd-Warshall (relatório)
- Estratégias de aplicação dos algoritmos (relatório)
- Estratégias de agrupamentos de pedidos (relatório)
- Conclusão
- Bibliografia
- Implementação da interface
- Implementação de Dijkstra (all pairs & single destination)

Bernardo

- Casos de utilização (relatório)
- Formalização do problema (relatório)

Francisco

- Análise da conectividade (relatório)
- Capa

BIBLIOGRAFIA

Djikstra:

Speeding up Dijkstra (YouTube) - *referência 1*

Bidirectional Search (GeeksForGeeks)

Bidirectional Dijkstra - Advanced Shortest Paths

Bidirectional Dijkstra vs Dijkstra (Stack Exchange Forum)

The Shortest Path Problem - Bidirectional Dijkstra's / Alt / Reach (YouTube)

Dijkstra vs Bi-directional Dijkstra Algorithm on US Road Network (YouTube)

Dijkstra's Algorithm - Computerphile (YouTube)

Dijkstra Explained (YouTube)

Dijkstra's algorithm in 3 minutes — Review and example (YouTube)

TSP:

Traveling Salesman Problem Visualization (YouTube) - *referência 2*

Traveling Salesperson Problem (Wikipedia)

Travelling Salesman Problem | Dynamic Programming | Graph Theory (YouTube)

Nearest Neighbour Algorithm (Wikipedia)

Conectividade:

Strongly Connected Components (YouTube)

Tarjan's Strongly Connected Components algorithm | Graph Theory (YouTube)

Tarjan's strongly connected components algorithm (Wikipedia)

Strongly connected component (Wikipedia)