U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

MarketOnWheels:

Supermercado Ambulante

Parte 2

Miguel Norberto Costa Freitas - up201906159
Joana Teixeira Mesquita - up201907878
Diogo Miguel Chaves dos Santos Antunes Pereira – up201906422

Mestrado Integrado em Engenharia Informática e de Computação
Conceção e analise de algoritmos
Turma 5 - Grupo 2

23 de maio de 2021

Contents

Cenários Implementados

Os três cenários propostos na primeira parte do relatório foram implementados:

Cenário 1 - a carrinha de capacidade ilimitada (cenário mais computacionalmente caro pois um único carro tem de visitar todos os clientes viáveis)

Cenário 2 - cenário de uma carrinha, mas com capacidade limitada.

Cenário 3 - múltiplas carrinhas em ação com capacidade limitada.

Análise de Conectividade

Um grafo não dirigido é conexo se e só se uma pesquisa em profundidade, a começar em qualquer vértice, visita todos os vértices do grafo.

Para verificar a conectividade de um grafo é verificada a quantidade de pontos de articulação (como exemplificado nas Figuras 1 e 2). Se um grafo for conexo, ou seja, uma pesquisa em profundidade a partir de qualquer vértice permite chegar a todos os outros (o grafo é cíclico), este apenas vai possuir um único ponto de articulação. Pelo contrário, em grafos não conexos pode haver um grande número de pontos de articulação.

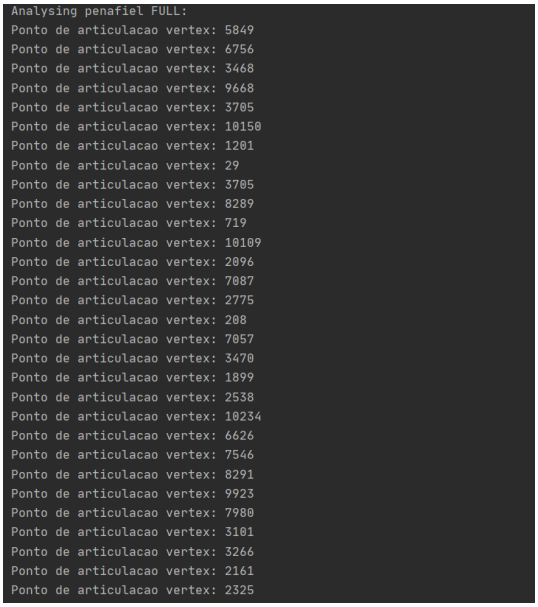

Figura 1 - Resultado para penafiel_full_nodes.txt

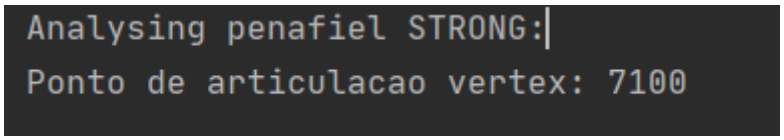

Figura 2 - Resultado para penafiel_strong_nodes.txt

Assim sendo, o grafo obtido a partir do ficheiro Penafiel_full_nodes.txt não é conexo, visto que possui diversos pontos de articulação. Isto deve-se ao facto de estes nodes representarem locais na cidade de Penafiel, e nem todos os locais dentro da cidade estarem ligados por estradas na cidade real e por isso não terão conexões no nosso grafo. Pelo contrário, o grafo obtido a partir do ficheiro Penafiel_strong_nodes.txt que utilizamos é conexo, com um único ponto de articulação devido ao facto de já terem sido retirados vários nodes inacessíveis, que não seriam úteis nem convenientes para nós, considerando que não podemos fazer entregas nestes.

“Data Structures” Utilizadas

Provider, Client e DeliveryCar

```
std::unordered_map<std::string, int> order;
std::unordered_map<std::string, int> stock;
std::unordered_map<std::string, int> shoppingList;
```

Estas três classes criadas por nós guardam o seu stock, order e shoppingList num std::unordered_map<std::string,int> que têm um tempo de inserção de O(1) em que a string representa o nome de um produto e o int representa a sua quantidade:

- A estrutura stock do Provider guarda os produtos que existem neste, para serem recolhidos pelo DeliveryCar;
- A estrutura order do Client aqueles que o cliente quer;
- A estrutura shoppingList do DeliveryCar os produtos que ele precisa de recolher nos Providers para poder fazer a entrega aos clientes definidos.

Para além disso o Provider possui ainda um booleano visited que será TRUE se o Provider tiver sido visitado pelo DeliveryCar. Por sua vez o DeliveryCar possui um vetor com os Ids dos nodes dos Clients, a que vai realizar entregas, uma capacidade, um ID e uma lista dos nodes visitados, que no fim da execução terão o caminho calculado.

Algoritmos Implementados - Análise Teórica e Espacial

Brute Force

Este foi o primeiro algoritmo que implementamos e é o que apresenta piores resultados em termos de utilização de memória e consumo de tempo, com o aumento do número de clientes viáveis. É a implementação do pseudocódigo submetido na primeira entrega do projeto que funciona da seguinte maneira (para mais detalhe sobre este algoritmo em específico pode verificar a primeira parte do relatório):

PARA CADA CARRO DE ENTREGA:

1) Seleciona as combinações de clientes válidas. (Uma combinação de clientes é válida quando existem produtos suficientes nas fábricas do grafo para realizar todas as suas entregas e quando o veículo é capaz de armazenar os produtos destes clientes);

2) Seleciona as melhores combinações de clientes. (Uma combinação de clientes é melhor que outra quando possui mais clientes. Para o mesmo número de clientes o desempate é realizado escolhendo a combinação que aproveita melhor a capacidade do veículo);

3) Caso haja um empate entre várias combinações, o algoritmo tenta cada uma delas e determina a melhor, ou seja, aquela que permite ao veículo percorrer a menor distância;

4) Determinada a melhor combinação de clientes para este carro de entregas, falta apenas eliminar os produtos que esta combinação irá consumir antes de repetir estes passos para o próximo carro, bem como "eliminar" estes clientes para que próximas combinações não os incluam.

Notas:

Este algoritmo atinge o seu limite com um input size de cerca de 40 clientes. Input sizes superiores a este valor provaram-se demasiado pesados para um computador de 16GB de RAM ficando a correr por mais de 20 minutos e eventualmente terminando com uma exceção de bad_alloc(). Para computadores de 8GB de RAM não é sequer possível correr este algoritmo para 20 clientes podendo mesmo forçar o sistema operativo a encerrar antes de dar a exceção bad_alloc(). Com esta quantidade de clientes, calcular todas as permutações e armazená-las sem qualquer tipo de "pruning" é demasiado pesado, o que é

comum para algoritmos do tipo brute force (funcionam bem para números pequenos de inputs, mas começam a dar problemas com números elevados).

Devido à necessidade de calcular permutações, que são o "bottleneck" deste algoritmo, a sua complexidade temporal acaba por ser O(n^3).

Evolução temporal

	2 Clientes	**5 Clientes**	**10 Clientes**	**20 Clientes**
1º Caso (um carro com capacidade ilimitada)	62886 ms	33733 ms	662133 ms	**Erro**: Bad_alloc
2º Caso (um carro com capacidade limitada – 10)	8100 ms	98874 ms	547874 ms	**Erro**: Bad_alloc
3º Caso (dois carros com capacidade limitada – 10)	81811 ms	89938 ms	725874 ms	**Erro**: Bad_alloc

Evolução espacial

A evolução espacial deste algoritmo é então O(n!), devido à necessidade de armazenar permutações.

Backtracking Recursive

Insatisfeitos com a performance do algoritmo de brute force decidimos tentar usar backtracking. Um dos grandes benefícios deste novo algoritmo foi não termos de guardar em memória todas as permutações de Clients e Providers, já que os nodes são verificados recursivamente, o que nos permite aceitar um input set de maiores dimensões que o brute force.

O seu funcionamento está clarificado em pseudocódigo apresentado abaixo, mas a maneira como opera é criando apenas os caminhos que são válidos, visto que quando encontra um cliente como primeiro node, tenta visitar um cliente para o qual não tem produtos, ou chega à conclusão que o caminho realizado até este ponto já leva mais tempo que o melhor calculado até agora ele para de calcular esse caminho dando backtracking e tentando um diferente. Isto permite-nos evitar a exceção de bad_alloc() já que uma quantidade significativa das subárvores de procura não têm de ser procuradas até ao final, o que também nos impede de atingir a maximum recursion depth. Se o algoritmo chegar ao final de uma árvore e não tiver percorrido todos os clientes a que o carro está associado, este é declarado inválido, já que, apesar de este caminho ter sido mais curto, o nosso objetivo principal é minimizar o número de carros necessários, o que implica maximizar o número de clientes a que cada um entrega.

A complexidade temporal deste algoritmo acaba por ser, no pior dos casos, O(n!) (caso em que TODAS as permutações têm de ser percorridas até ao fim) mas na realidade este cenário é extremamente raro e a maior parte das subárvores de pesquisa são cortadas.

É de notar que este N não inclui todos os clientes e fornecedores do grafo, apenas aqueles que foram selecionados num passo de pré-processamento para o carro atual em específico, o que lhe permite alcançar melhores resultados que o brute force.

Evolução temporal

	2 Clientes	**5 Clientes**	**10 Clientes**	**20 Clientes**
1º Caso (um carro com capacidade ilimitada)	40265 ms	130478 ms	170060 ms	**+21 minutos**
2º Caso (um carro com capacidade limitada – 10)	6621 ms	19039 ms	55610 ms	14677 ms
3º Caso (dois carros com capacidade limitada – 10)	25455 ms	21152 ms	52085 ms	54185 ms

Na realidade, estes resultados não são indicativos do "scaling" do algoritmo quando aumentamos o número de clientes, isto deve-se ao facto, de que, devido ao pré-processamento feito o tempo levado é influenciado não pelo número total de clientes do grafo, mas sim pelo número de clientes viáveis. Estes dados são então mais uteis em comparação com os dos outros algoritmos do que sozinhos, visto que todos os algoritmos foram testados para os mesmos grafos.

Houve uma tentativa de testar para o número de clientes viáveis, mas tornava-se muito difícil de criar testes com diversos números de clientes viáveis.

Evolução espacial

O algoritmo mantém-se mais ou menos consistente em termos de utilização de memória para os diferentes números de clientes, tendo uma evolução espacial de O(n).

Nearest Neighbour Heuristica

Testamos ainda um algoritmo com a heurística do Nearest Neighbour que consiste muito simplesmente em viajar para o node mais próximo do atual. A filosofia desta heurística, tal como muitos outros algoritmos gulosos, é encontrar a melhor solução global através de sucessivas soluções ótimas locais.

Naturalmente, esta heurística, à semelhança de muitos algoritmos gulosos, possui situações em que não devolve a solução ótima.

Para N pontos distribuídos aleatoriamente num plano, este algoritmo, em média, retorna um caminho com distâncias 25% superiores ao menor caminho possível, o que nós também verificámos, contudo, este algoritmo permite-nos resolver este problema num tempo menor que a solução com backtracking e brute force.

Para implementar esta heurística no nosso projeto apenas foi necessário adaptar este conceito para que incorpora-se as várias restrições do nosso enunciado. Assim o algoritmo final comporta-se da seguinte maneira:

1 - Calcula qual é o Provider mais próximo do Node atual que contém produtos que o Carro de Entregas vai precisar de recolher.

2 - Computa a lista de Clientes que podem receber uma entrega (um carro pode receber uma entrega quando os produtos atualmente no carro já são suficientes para cobrir os produtos que o cliente pediu).

3 - Tendo estas duas informações, o algoritmo move-se para o Provider (calculado no passo 1) ou para o Client que já pode receber a sua entrega, dependendo de qual se encontra mais próximo do node atual.

4 – Repete para todos os nodes até ter efetuado as entregas a todos os clientes que lhe foram indicados.

O algoritmo acaba por ter uma complexidade temporal de O (n * O (E + V. log(V))), devido à necessidade de calcular as distâncias entre todos os nodes possíveis de visitar, no pior caso. No melhor caso possível, em que todas as distâncias já foram calculadas, a complexidade temporal acaba por ser O (n * log(N)). Isto é possível utilizando um algoritmo de Path finding que possui “memoization”, que guarda os resultados num map (em que o “lookup” time é O(log(n))) ou numa outra estrutura com um “lookup” time semelhante.

Nota:

O algoritmo de Nearest Neighbour apenas corre após saber a que clientes o carro em questão irá ter de realizar entregas, ou seja, o carro já sabe, previamente, a lista de produtos que tem de ir buscar.

Evolução temporal

	2 Clientes	**5 Clientes**	**10 Clientes**	**20 Clientes**
1º Caso (um carro com capacidade ilimitada)	10366 ms	10111 ms	10659 ms	15707 ms
2º Caso (um carro com capacidade limitada – 10)	3836 ms	3523 ms	4724 ms	7232 ms
3º Caso (dois carros com capacidade limitada – 10)	7839 ms	6601 ms	8424 ms	10543 ms

Mais uma vez, devido ao pré-processamento realizado e ao facto de já ser dado ao algoritmo os clientes a visitar, estes resultados são menos indicativos do "scalling" do algoritmo para elevados inputs do que se tivéssemos o número de clientes viáveis. Assim, são mais interessantes em conjunto com os dos restantes algoritmos.

Evolução espacial

Este algoritmo tem uma complexidade de O(n) no pior caso em que n são os nodes que o algoritmo tem de visitar.

Cálculo de distâncias entre nodes

Inicialmente, todos os algoritmos utilizados usavam o algoritmo de Path Finding Dijkstra, no entanto, os algoritmos de Brute Force e Nearest Neighbour Heuristic foram modificados para funcionarem com dois algoritmos de path finding, Dijkstra com "memoization" e Floyd-Warshall de forma a melhorar os seus tempos. Estes apenas não foram aplicados ao backtracking recursive devido a falta de tempo, mas também poderiam ser utilizados neste.

No nosso caso em que foram consideradas estas duas soluções, verificou-se que o algoritmo estava a perder bastante tempo a calcular a distância mínima entre conjuntos de nodes, que já tinha sido calculada anteriormente. Assim, surgiu a necessidade de adaptar o algoritmo de Dijkstra e combiná-lo com programação dinâmica para criar um Dijkstra com "memoization", que apenas realiza os cálculos se verificar que ainda não os realizou anteriormente, caso contrário simplesmente lê o valor previamente calculado.

Adicionalmente, para resolução deste problema tentamos usar o algoritmo de Floyd-Warshall, um dos grandes exemplos da programação dinâmica, que opera calculando todos os menores caminhos entre todos os Nodes do grafo no início da execução. No entanto, para os grafos fornecidos, que continham entre 15 mil e 20 mil nodes, este passo de pré-processamento é naturalmente computacionalmente caro, levando, em média, cerca de 20 minutos nos nossos computadores pessoais. Face a esta realidade, tentámos evitar este passo e substituí-lo por uma leitura de um ficheiro ".txt", onde seriam armazenados os dados relevantes para o Floyd Warshall após a primeira execução deste, evitando assim o pré-processamento inicial quando o programa corresse novamente. Contudo, devido à dimensão do grafo e à lentidão de input/output streams em C++, este passo demorava mais tempo a executar do que simplesmente realizar o pré-processamento novamente, daí que foi retirado da versão final do código.

Dijkstra Memoized

A sua ordem de complexidade é:

O (|V|^2)

```
std::map< std::pair<std::string , std::string>,
std::pair<Vertex<T>*, double> > dijkstraMemoization;
```

Para realizar a "memoization" do Dijkstra recorreu-se a um std::map cuja key é um par de duas strings que funcionam como um ID <idNodeInicial, idNodeFinal> e cujo valor é outro par constituído por um apontador para Vértice (seria o valor do campo PATH da classe Vertex utilizada nas aulas práticas) e um double que seria a distância mínima entre Node Inicial e Node Final.

Floyd Warshall

A sua ordem de complexidade é:

O (|V|^3)

```
std::vector<std::vector<double>> distMin;
std::vector<std::vector<Vertex<T>*>> predecessores;
```

Para a realização do algoritmo de Floyd Warshall foram criadas duas novas estruturas na classe Graph:

- Um vetor de vetores de doubles, que está organizado para que distMin[indiceOrigem][indiceFinal] retorne a distância mínima entre esses dois nodes.
- Um vetor de vetores com apontadores para vértices, que existe para auxiliar a descoberta do path mínimo entre dois nodes.

Pseudo-código

Floyd Warshall

```
Input: string city

	distMin.clear()
	predecessores.clear()
	distMin <- std::vector<std::vector<double>>(getNumVertex(),
std::vector<double>(getNumVertex(), INF))

	predecessores <- std::vector<std::vector<Vertex<T>*>>(getNumVertex(),
std::vector<Vertex<T>*>(getNumVertex(), NULL))

	for i <- 0 to getNumVertex() do
		for j <- 0 to getNumVertex() do
			if i == j then // Elementos da diagonal da Matriz a 0
				distMin[i][j] <- 0
			end if
			else then
				for edge in vertexSet[i]->adj do
					if edge.dest->getInfo() == vertexSet[j]->info then
						distMin[i][j] <- edge.weight;
						predecessores[i][j] <- vertexSet[i];
				end for
			end else
		end for
	end for
	for k <- 0 to getNumVertex() do
		for l <- 0 to getNumVertex() do
			for m <- 0 to getNumVertex() do
				if distMin[l][m] > distMin[l][k] + distMin[k][m] then
					distMin[l][m] <- distMin[l][k] + distMin[k][m]
					predecessores[l][m] <- predecessores[k][m]
				end if
			end for
		end for
	end for
```

Dijkstra

```
Input: const T origin

	for vertex in getVertexSet() do
		vertex->dist <- 9999
		vertex->path <- NULL
	end for
	vertexSource <- findVertex(origin)
	vertexSource->dist <- 0
	vertexSource->path <- NULL

	MutablePriorityQueue<Vertex<T>> q
	q.insert(vertexSource)
	while not q.empty() do
		vertexCurrent <- q.extractMin()
		for edge in vertexCurrent->adj do
			if edge.dest->dist > vertexCurrent->dist + edge.weight then
				edge.dest->dist <- vertexCurrent->dist + edge.weight
				edge.dest->path = vertexCurrent
				result <-
dijkstraMemoization.insert(std::make_pair(std::make_pair(origin.getId(), edge.dest-
>info.getId()), std::make_pair(vertexCurrent,edge.dest->dist)))
				if not result.second then
					result.first->second <- vertexCurrent,edge.dest->dist
```

```
                        end if
                        q.insert(edge.dest)
                end if
        end for
end while
```

Nearest Neighbour Adapted

```
Input: std::vector<Provider *> &providers, std::vector<Client *> &clients, Graph<Node>
&graph, PATH_FINDING_ALGO algo

        //get all the deliverable clients
        for *provider in providers do
                provider->setVisited(false)
        end for

        fillShoppingList(clientsToDeliverTo,clients)
        isClientDeliverableNext <- false
        deliverableClientID <- 0
        currNode <- 0
        totalCost <- 0
        closestDistance <- 0
        std::vector<Node> finalPath
        amountOfDeliverableClients <- 0


        finalPath.push_back(graph.getOriginNode())
        std::unordered_map<std::string, int> currCarryingCar
        std::pair<std::vector<Node>, double> intermediatePaths
        costToTravel <- 0


        while not clientsToDeliverTo.empty() do
                amountOfDeliverableClients <- 0
                closestDistance <- 99999999
                for clientID in clientsToDeliverTo do
                        costToTravel <- 0
                        if checkIfClientIsDeliverable(clients[clientID], currCarryingCar) then
                                amountOfDeliverableClients++
                                //check the distance between currentNode and client
                                if algo == DIJSKTRA then
                                        intermediatePaths <-
                                graph.getDijsktraPath(finalPath[currNode], *clients[clientID])
                                end if
                                else if algo == FLOYD_WARSHALL then
                                        intermediatePaths <-
                                graph.getfloydWarshallPath(finalPath[currNode],
                                *clients[clientID])
                                end else if
                                costToTravel <- intermediatePaths.second
                                //add the cost to travel in between these two nodes
                                if costToTravel < closestDistance then
                                        closestDistance <- costToTravel
                                        isClientDeliverableNext <- true
                                        deliverableClientID <- clientID
                                end if
                        end if
                end for

                //if all clients are already deliverable there's no point in visiting another
        provider
                if amountOfDeliverableClients != clientsToDeliverTo.size() then
                        //get the closest provider that has products relevant to the cause
                        for *provider in providers do
                                costToTravel <- 0
                                if checkIfProviderIsRelevant(provider, shoppingList) then
                                        //check the distance between currentNode and client
                                        if algo == DIJSKTRA then
                                                intermediatePaths <-
                                        graph.getDijsktraPath(finalPath[currNode], *provider)
                                        end if
                                        else if algo == FLOYD_WARSHALL then
```

```
                                        intermediatePaths <-
                                graph.getfloydWarshallPath(finalPath[currNode],
                                *provider)
                                end else if
                                costToTravel <- intermediatePaths.second
                                //add the cost to travel in between these two nodes
                                if costToTravel < closestDistance then
                                        closestDistance <- costToTravel
                                        isClientDeliverableNext <- false
                                        nearestProvider <- provider
                                end if
                        end if
                end for
        end if

        if not isClientDeliverableNext then
                loadCar(currCarryingCar, nearestProvider->getStock())
                finalPath.push_back(*nearestProvider)
                nearestProvider->setVisited(true)
        end if
        else then
                removeFromClients(deliverableClientID)
                finalPath.push_back(*clients[deliverableClientID])
                unloadMerch(currCarryingCar,clients[deliverableClientID])
        end else
        currNode++
        totalCost += closestDistance
end while



finalPath.push_back(graph.getOriginNode())
if algo == DIJSKTRA then
        intermediatePaths <- graph.getDijsktraPath(finalPath[currNode],
graph.getOriginNode())
end if
else if algo == FLOYD_WARSHALL then
        intermediatePaths <- graph.getfloydWarshallPath(finalPath[currNode],
graph.getOriginNode())
end else if
totalCost += intermediatePaths.second

return make_pair(finalPath,totalCost)
```

Backtracking Recursive

```
Input: std::vector<Provider *> &providers, std::vector<Client*> &clients, Graph<Node> &graph,
PATH_FINDING_ALGO algo

        fillShoppingList(clientsToDeliverTo, clients)
        std::vector<std::vector<int>> viableRoutes
        checkProviderCombinations(providers.size(), viableRoutes, providers)
        std::vector<Node> bestRoute

        std::vector<int> clientIds <- clientsToDeliverTo
        bestestDistance <- 9999999
        std::vector<int> bestestPath
        std::vector<int> Path
        for &provPerm in viableRoutes do
                Path <- {} //source node of the company
                for &prov in provPerm do
                        Path.push_back(stoi(providers[prov]->getId()))
                end for

                for &client in clientIds do
                        Path.push_back(stoi(clients[client]->getId()))
                end for
                std::vector<int> visitedNodes
                bestDistance <- 0
                numClients <- 0
                maxClients <- clientIds.size()
                std::vector<int> bestPath
```

```
        std::unordered_map<std::string, int> carStock
        findBest(visitedNodes, graph, Path, bestPath, bestDistance, 0, providers,
clients, carStock, numClients, maxClients)
        bestestDistance <- bestDistance + graph.getDijsktraPath(graph.getOriginNode(),
Node(to_string(bestPath[0]))).second
        bestestDistance <- bestDistance + graph.getDijsktraPath(graph.getOriginNode(),
Node(to_string(bestPath.back()))).second
        if bestDistance < bestestDistance then
                bestestDistance <- bestDistance
                bestestPath <- bestPath
        end if
end for
bestRoute.push_back(graph.getOriginNode())
for pathId in bestestPath do
        bestRoute.push_back(Node(to_string(pathId)))
end for
bestRoute.push_back(graph.getOriginNode())

return make_pair(bestRoute,bestestDistance)
```

Find Best

```
Input: visitedNodes, Graph<Node>& graph, ids, &bestPath, &bestDistance, curDistance,
std::vector<Provider *> &providers, std::vector<Client *> &clients,
std::unordered_map<std::string, int>& carStock, &numClients, &maxClients

        std::pair<std::vector<Node>, double> intermediatePaths
        std::unordered_map<std::string, int> carStockCopy <- carStock
        for i <- 0 to ids.size() do
                itId <- find(visitedNodes.begin(),visitedNodes.end(),ids[i])
                if itId != visitedNodes.end() && not visitedNodes.empty()then
                        continue
                end if
                provider <- checkIfProvider(providers,ids[i])
                if provider != nullptr then
                        visitedNodes <- ids[i]
                        if visitedNodes.size() > 1 then
                                //check distance between this node and the last
                                intermediatePaths <-
                        graph.getDijsktraPath(Node(to_string(visitedNodes[visitedNodes.size() -
                        1])), Node(to_string(visitedNodes[visitedNodes.size() - 2])))
                                curDistance <- curDistance + intermediatePaths.second
                        end if

                        //add products to map
                        loadCar(carStock,provider->getStock())
                        if curDistance <= bestDistance then
                                findBest(visitedNodes,graph,ids,bestPath,bestDistance,
                                curDistance, providers,clients, carStock,numClients,maxClients)
                        end if

                        visitedNodes.pop_back()
                        carStock <- carStockCopy
                        if visitedNodes.size() >= 1 then
                                curDistance <- curDistance - intermediatePaths.second
                        end if
                end if
                else then
                        if visitedNodes.empty() then
                                continue
                        end if
                        else then
                                client <- checkIfClient(clients, ids[i])
                                if checkIfClientIsDeliverable(client,carStock) then
                                        numClients++
                                        visitedNodes <- ids[i]
                                        unloadMerch(carStock,client)

                                        intermediatePaths <-
                                        graph.getDijsktraPath(visitedNodes[visitedNodes.size() -
                                        1]),visitedNodes[visitedNodes.size() - 2])

                                        curDistance <- curDistance + intermediatePaths.second
```

```
                                if curDistance <= bestDistance then
                                                                findBest(visitedNodes,
                                        graph,ids,bestPath,bestDistance,curDistance,provi
                                        ders,clients, carStock,numClients,maxClients)

                                end if
                                numClients--
                                visitedNodes.pop_back()
                                carStock <- carStockCopy


                                if visitedNodes.size() >= 1 then
                                        curDistance <- curDistance -
                                        intermediatePaths.second
                                end if
                        end if
                end else
        end else
end for
if numClients != maxClients then
        return
end if
if bestPath.empty() then
        bestPath <- visitedNodes
        bestDistance <- curDistance
end if
else if curDistance < bestDistance then
        bestPath <- visitedNodes
        bestDistance <- curDistance
end else if
```

Biografia

- Gutina, Gregory; Yeob, Anders; Zverovich, Alexey (15 March 2002). "Traveling salesman should not be greedy: domination analysis of greedy-type heuristics for the TSP". Discrete Applied Mathematics. 117 (1–3): 81–86.
- Zverovitch, Alexei; Zhang, Weixiong; Yeo, Anders; McGeoch, Lyle A.; Gutin, Gregory; Johnson, David S. (2007), "Experimental Analysis of Heuristics for the ATSP", The Traveling Salesman Problem and Its Variations, Combinatorial Optimization, Springer, Boston, MA, pp. 445–487, CiteSeerX 10.1.1.24.2386,