ALGORITMOS DE OTIMIZAÇÃO PARA O PROBLEMA DE ROTEAMENTO DE VEÍCULOS

Aluno: Rodrigo Rosatti Galvão

RA: 176939

Orientador: Luis Augusto Angelotti Meira

Vigência: Fevereiro-2016 a Fevereiro-2017

30 de março de 2017

Faculdade e Tecnologia – Universidade Estadual de Campinas

R. Paschoal Marmo, 1888 - CEP:13484-332 - Jd. Nova Itália - Limeira, SP

Resumo

Este projeto se propôs a desenvolver algoritmos para o problema de roteamento de veículos numa malha viária. Quando um veículo recebe um conjunto de pontos de entrega, ele deve utilizar uma rota otimizada, minimizando a distância percorrida ou o tempo total. Trata-se do clássico problema do Caixeiro Viajante (TSP). Entretanto, havendo mais de um veículo, o problema passa a ter duas etapas: *(i)* particionar os pontos de entrega e *(ii)* para cada conjunto resolver um caixeiro viajante. O objetivo é encontrar um mínimo global em relação às duas etapas. Durante o projeto, o aluno estudou e implementou diversos algoritmos para o TSP, como a 2-aproximação baseado na árvore geradora mínima, a 1.5-aproximação baseado no algoritmo Christofides e também uma estratégia de mínimo local baseado em *single-swap*. Como etapas intermediárias, o aluno resolveu o problema da MST usando o algoritmo de Kruskal, o problema do Emparelhamento Mínino, usando um algorimto de *single-swap*. Também implementou um pacote de Grafos com diversas operações e um pacote de visualização que permite visualizar a instâncias e as soluções. Trabalhou com técnicas de depuração e *profiling*. O trabalho iniciou com instâncias de 35 vértices e foi progressivamente aumentando para 50, 100 e 200 vértices. Um dos objetivos era executar instâncias com 28 mil vértices, porém a maior instância que conseguimos excutar possui 6417 vértices. Durante a maior parte do projeto o aluno se dedicou ao TSP, que é uma etapa intermediária do VRP. Os ultimos dois meses do projeto foram dedicados para estudar sobre heurísticas e algoritmos para solução do VRP. Dentre os algoritmos estudados, foi escolhido o *Clarke & Wright's Savings Algorithm*, por se tratar de uma das heurísticas mais conhecidas para o VRP. Esta pesquisa foi contemplada com uma bosa de iniciação científica da FAPESP.

1 Introdução

O Problema do Caixeiro Viajante (TSP) é um dos mais estudados da otimização combinatória [2, 3, 10, 18]. Uma generalização do TSP é o Problema do Roteamento de Veículos também conhecido como *Vehicle Routing Problem–VRP* [22]. Neste problema, deseja-se encontrar k ciclos que cubram todos os pontos de entrega (clientes) e que possuam um vértice especial em comum (depósito). O VRP possui requisitos adicionais de acordo com a situação.

Uma das primeiras aparições do VRP foi em 1959 [12] sob o nome de *Truck Dispaching Problem*, uma generalização do Problema do Caixeiro Viajante *(TSP)*. O nome do VRP aparece em 1976 no tabalho de Christophides [8]. Christophides define VRP como um nome genérico dado a uma classe de problemas que envolvem visitar "clientes" com veículos.

Situações distintas levam a diversas variantes do problema. Assumindo-se capacidade no veículo, temos o *Capacitated-VRP (CVRP)* [15]. Definindo-se janelas temporais de entrega, temos o *VRP with Time Windows (VRPTW)* [17]. Havendo mais de um depósito, temos o *Multi-Depot VRP (MDVRP)* [21]. Outras variantes são facilmente encontradas na literatura.

O Problema do Roteamento de Veículos (*Vehicle Routing Problem*–VRP) modela diversas situações reais, como o roteamento de veículos para atender chamados ou ocorrências, definição de rotas para carteiros, definição de rotas para coleta de lixo, definição de itinerários de ônibus fretados e vans, entre muitas outras. Ele é um problema da classe NP-Difícil [11], o que implica não existir algoritmos eficientes para resolvê-lo na exatidão a menos que $P = NP$.

Este projeto se propôs a estudar técnicas de otimização e projetar algoritmos para o VRP.

Seja $w(v_i, v_j)$ o custo do menor caminho entre dois vértices v_i e v_j em um grafo $G(V, E)$. O custo pode ser a distância ou tempo gasto por um veículo entre os pontos v_i e v_j.

Definição 1.1 (Problema do Roteamento de Veículos–Variante 1 (VRP 1))**.** *Dados um Grafo $G(V, E)$, uma função custo $w : V \times V \to Q^+$, um conjunto pontos de entrega $S \subseteq V$, um depósito $d \in V$ e um número de veículos $k \in \mathbb{N}$, encontre uma partição de S em k rotas $\{R_1, \ldots, R_k\}$ tal que que o custo da soma das rotas seja minimizado, onde o custo de uma rota $R = (r_1, \ldots, r_n)$ é dado por*

$$w(d, r_1) + \sum_{i=1}^{n-1} w(r_i, r_{i+1}). + w(r_n, d)$$

2 Materiais e Métodos

O projeto foi desenvolvido na linguagem Java. O aluno utilizou a IDE Netbeans e o sistema de controle de versão SVN oferecido gratuitamente pela FT-UNICAMP. O aluno teve à disposição computadores e notebooks para desenvolver a pesquisa.

Utilizamos a instância Brasil28k, que se refere à um conjunto com 28.142 localidades brasileiras e suas respectivas latitudes e longitudes. Trata-se de uma instância de origem incerta, encontrada na internet no ano de 2010. Ela contempla os 5.570 municípios brasileiros bem como localidades internas aos municípios.

Mesmo tendo sua origem desconhecida, a instância Brasil28k tem sido utilizada regularmente nos cursos de programação avançados, servindo de caso de teste para algoritmos em grafos. Veja a Figura 1.

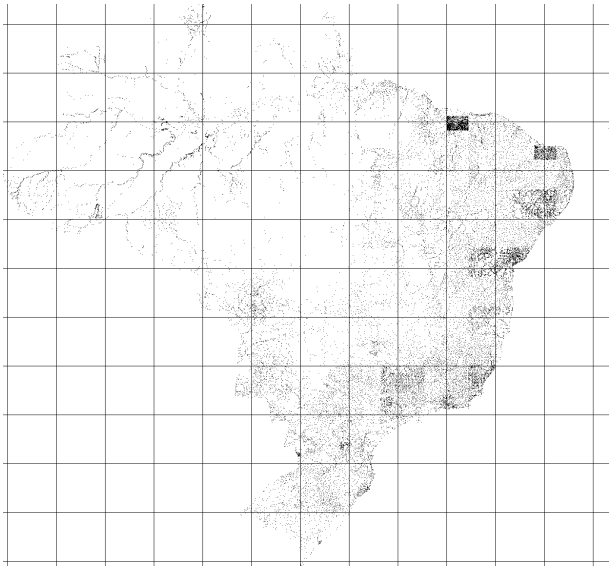

Figura 1: Instância Brasil28k. Cada pixel preto corresponde a uma das 28.142 localidades.

O aluno utilizou algoritmos de aproximação para o TSP, como a 2-aproximação a partir da árvore geradora mínima e o algoritmo 1.5-aproximado de Christofides [5]. Para o VRP, foi implementado o *Clarke & Wright's Savings Algorithm*[1].

3 Cronograma Inicial

Este projeto se propôs a estudar e desenvolver algoritmos para o VRP.

O aluno Rodrigo Rosatti Galvão tinha intenção de entrar em contato com uma empresa de

transportes, coleta de lixo ou que atenda chamados no cliente para obter instâncias reais. Tal atividade não foi realizada, devido a falta de tempo referente à execução do projeto.

A proposta de solução para o problema consistia em cinco etapas:

1. Estudar e Implementar Árvore Geradora Mínima e Emparelhamento Mínimo. Essa etapa é um pré-processamento para a resolução do TSP, uma vez que tais algoritmos fornecem limitantes para o problema.

2. Seleção e Criação de Instâncias: O aluno deverá buscar *benchmarks* para o VRP. Também irá modelar situações reais que podem ser otimizadas pelo VRP.

3. Implementações: Implementar o Algoritmo Força Bruta. Implementação do *branch-and-bound*. Implementar heurísticas para a resolução do problema, como algoritmos genéticos e bioinspirados, entre outros. Obtenção de limitantes superiores e inferiores para as instâncias.

4. Análise dos Resultados: Executar experimentos computacionais.

5. Escrita. Ao final da iniciação científica será gerado um relatório de atividades. É desejável que o trabalho se transforme num poster e seja submetido a um congresso de iniciação científica.

Na Tabela 1 há o cronograma esperado para o cumprimento destas atividades.

Etapa	Trimestre			
	01	02	03	04
1	✓	✓		
2	✓	✓		
3		✓	✓	
4		✓	✓	
5		✓		✓

Tabela 1: Cronograma das atividades previstas

4 Resultados

O problema do TSP e do VRP estão intimamente ligados. A parte inicial do projeto foi dedicada para implementação de soluções para o TSP. Tais soluções exigiram a maior parte de tempo dedicado ao projeto, visto que era uma etapa que teriamos que passar para depois começar a implementar soluções para o VRP.

4.1 Estudar e Implementar Árvore Geradora Mínima ✓

Foi implementado o algoritmo de Kruskal. Foi necessário estudo de estruturas de dados como o Union-Find. Foi desenvolvido uma classe *Graph* inicialmente simples. Tal classe foi ganhando métodos ao longo do tempo. Também foi criada uma classe *Draw* para visualização do resultado. A figura 2 contém árvore geradora mínima criada neste momento.

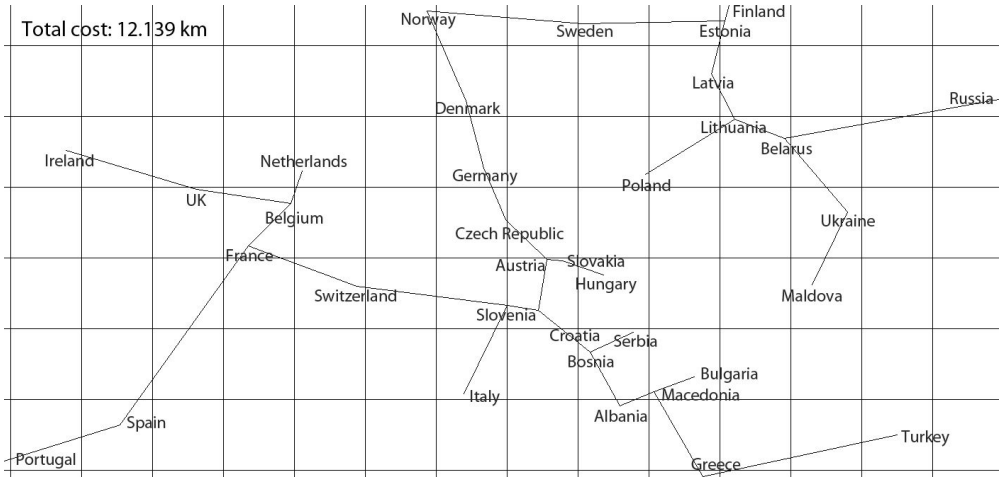

Figura 2: MST gerada a partir das capitais da Europa representadas em coordenadas geográficas.

4.2 Estudar e Implementar Emparelhamento Mínimo ✓

O emparelhamento mínimo é uma etapa na obtenção da 1.5-aproximação para o TSP pelo algoritmo de Christofides.

Emparelhamento mínimo é um problema complexo. Para simplificar, utilizamos soluções heurísticas. o primeiro algoritmo para emparelhamento mínimo consisitiu em gerar milhares/milhões de emparelhamentos aleatórios e selecionar o menor. O critério de parada consistia em limitar o número de iterações a n^3.

Esta estratégia funcionou para instâncias pequenas, porém para instâncias grandes o consumo de tempo tornou-se proibitivo. Decidimos, então, utilizar a heurísitca de escalada da montanha. A cada solução, varremos a vizinhança e saltamos para uma solução que melhora a função objetivo (*fo*).

Considere um sequencia de m pares de vértices $((v_1, u_1), \ldots, (v_m, u_m))$. O conceito de vizinhança utilizado foi o *single-swap*. Escolhemos dois pares (u, v) e (u', v') e tentamos rearranjá-los de maneira a melhorar a *fo*.

Para facitar a varredura da vizinhança, criamos uma máquina de estados que, dado um conjunto S de n elementos, a máquina de estados passa por todas as $\binom{n}{2}$ combinações de S 2 a 2.

4.3 Seleção e Criação de Instâncias ✓

A primeira instância consistiu nas capitais da Europa. O aluno buscou a latitude e a longitude das principais capitais da Europa. O cálculo das distâncias foi feito por meio de coordendas geográficas. Veja figura 2.

A segunda instância utilizada foi um conjunto de aproximadamente 28 mil coordenadas geográficas do brasil (Brasil28k). Veja a Figura 1.

Tal instância foi trabalhada em partes. O aluno aplicou filtros como *(i)* localidades por estado, como São Paulo; *(ii)* localidades por região: sul, sudeste, centro-oeste, norte e nordeste.

Dentro destes filtros, o aluno selecinou uma quantidade limitada de localidades, como, por exemplo: primeiras 50, 100, 200, 400 e 800 localidades.

A figura 3 contém a MST das localidades do estado de São Paulo.

Figura 3: MST gerada a partir das 1719 localidades do estado de são paulo presentes na Instância Brasil28k.

4.4 Implementações (parcial)

A parte inicial do projeto foi dedicada para estudar e implementar o algoritmo 1.5-aproximado proposto por Christofides [6, 5]. O algoritmo de Christofides usa o algoritmo da árvore geradora mínima e o algoritmo do emparelhamento mínimo como sub-rotina. A figura 4 representa o

resultado inicial que tivemos à partir da implementação do algoritmo.

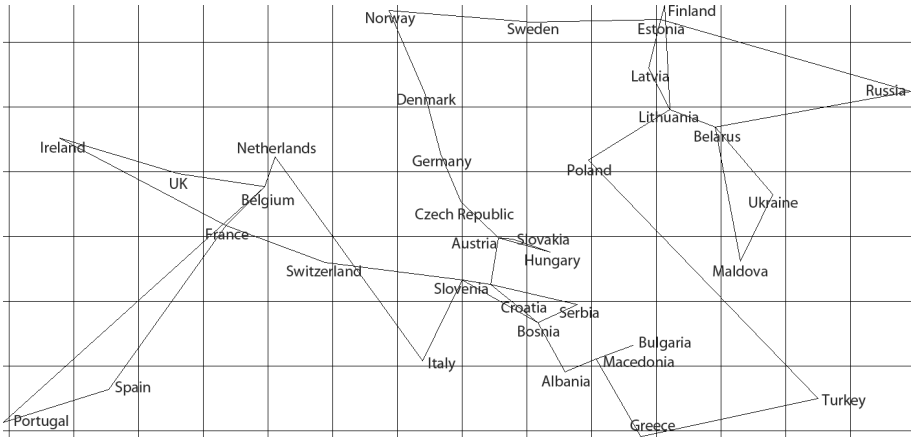

Figura 4: Resultado obtido à partir da primeira implementação do algoritmo de Christofides.

Durante a implementação Christofides, aperfeiçoamos a classe *Graph.java*, substituindo a matriz de adjacência por um *hash*. O *hash* é mais econômico em termos de memória e permite acesso em tempo constante, ou seja O(1). Criamos uma classe para visualização dos resultados, chamada *DrawPNG.java*. Tal classe permite a visualização de vértices, arestas, rotas e do grafo completo.

Uma das etapas do algoritmo de Christofides envolve a separação dos vértices de grau ímpar presentes no grafo. Para tanto, implementamos um algoritmo que calcula o sub-grafo induzido. Depois, foi necessário calcular o emparelhamento mínimo neste subgrafo.

Para isso, criamos a classe *HeuristicMinMatching.java*. Esta classe faz uma busca local baseada em *single swap*. Toda vez que é encontrado uma melhoria no empalhamento, a troca é feita. Decidimos usar a heurística de *single swap* por ser mais simples que o algoritmo do emparelhamento mínimo. O emparelhamento mínimo/minimal é adicionado à MST.

Um dos últimos passos ao se implementar o algoritmo de Christofides é encontrar o caminho Euleriano. O caminho Euleriano tem como objetivo percorrer todas as arestas de um grafo, de modo a passar exatamente uma única vez por cada aresta. Para isso, implementamos a classe *EulerianGraph.java*, responsável por realizar os algoritmos de caminho euleriano seguido da etapa final, chamada *shortcut* [5].

O caminho euleriano é uma etapa do algoritmo de Christofides. Observamos que o caminho Euleriano em questão é feito em um multi-grafo. Então, criamos uma nova classe chamada *UnweightedMultiGraph.java* que funciona como um *wrapper* da classe *Graph.java*, permitindo que se possa utilizar os métodos já existentes dessa classe e modificar e/ou adicionar novas operações específicas para trabalhar com multigrafo.

O Algoritmo de Christofides é 1.5-aproximado. Ao analisar o resultado, foi percebido muitas arestas cruzadas. Num grafo no plano Euclidiano, cruzamento de arestas sempre podem ser removidos, diminuindo o custo da rota. Decidimos, então, executar uma técnica de *single-swap* para melhorar a rota vinda do algoritmo.

A implementação que fizemos para a técnica de *single-swap* pode ser classificada como escalada da montanha (*hill climbing*). Este conceito tenta maximizar ou minimizar determinada função à partir de uma análise de vizinhança. No nosso caso estávamos querendo minimizar o comprimento da rota. Então implementamos a função para o *swap* da seguinte maneira:

1. Percorremos os elementos da rota e realizamos a troca de posições entre duas arestas.

2. Calculamos o custo dessa nova rota. Se esse custo for maior, então mantemos o custo anterior e continuamos o processo de *swap*. Caso contrário, substituímos a antiga rota pela nova e começamos a varrer a vizinhança novamente.

3. Realizamos essas etapas até que não haja mais nenhuma troca de posições possível.

O Christofides somado ao *single-swap* obteve resultados satisfatórios. Sem cruzamento de arestas e dentro do 1.5 aproximado. Atualmente, estamos rodando esse algoritmo com uma instância de 6417 pontos, referente à região Sudeste do Brasil. A figura 7 mostra o resultado obtido para essa instância, após a implementação do algoritmo de Christofides e do algoritmo de *single-swap*.

Uma ferramenta utilizada durante o desenvolvimento do projeto foi o *Profiler* do Java, que fornece informações importantes relacionadas ao comportamento da aplicação durante seu tempo de execução. A IDE Netbeans oferece cinco recursos associados com essa ferramenta: *telemetry, methods, objects, threads* e *locks*. No nosso projeto, utilizamos apenas as opções *Telemetry* e *Methods*.

A *Telemetry* apresenta gráficos referentes ao uso de CPU, memória, *garbage collection, threads* e classes. Essas informações são muito interessantes, pois à partir delas é possível observar em que momentos a aplicação está consumindo mais CPU e memória e assim, poder analisar com mais clareza o comportamento do sistema nesses momentos.

A opção *Methods* mostra o tempo consumido por cada método presente na aplicação. Essas informações são úteis para observar quais métodos estão consumindo um tempo exagerado de execução. Isso auxilia na detecção e remoção de gargalos no código.

A análise do sistema utilizando o *Profiler* foi importante para encontrar determinadas situações que estavam levando o programa a certos problemas de desempenho, tanto com relação à tempo de execução como consumo de memória. À partir dessas observações conseguimos implementar novas soluções e assim melhorar o desempenho do nosso algoritmo.

Instância	Vértices	Tempo (min)	Memória (GB)
Centro-Oeste	2786	11	1,2
Sul	2953	10	0,9
Norte	4132	27	1,5
Sudeste	6417	108	2,3

Tabela 2: Análise das regiões do Brasil com relação ao tempo e uso de memória.

A tabela 2 apresenta o resultado obtido à partir da análise das instâncias das regiões Centro-Oeste, Sul, Norte e Sudeste. A figura 5 representa a relação entre a quantidade de vértices com o tempo de execução, enquanto a figura 6 representa a análise com base no consumo de memória durante a execução do programa.

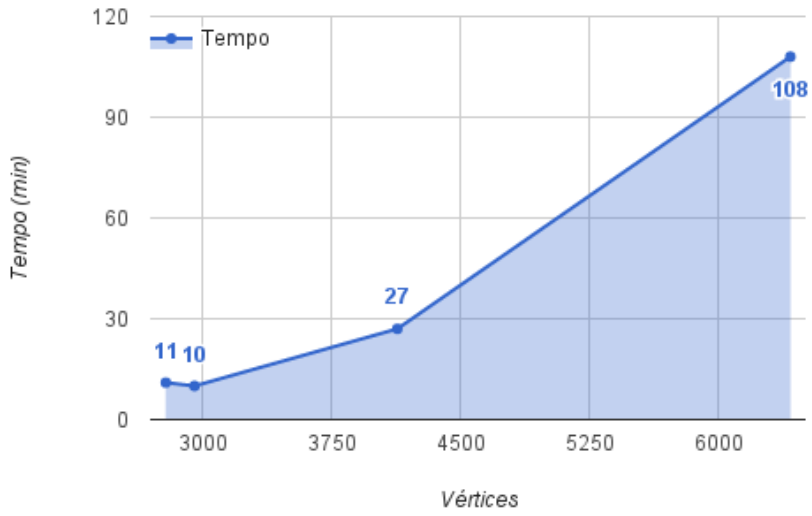

Figura 5: Relação entre vértices e tempo de execução consumido.

VRP

Após implementação do algoritmo de Christofides, foi realizado um estudo mais aprofundado à respeito do VRP e suas variantes. Para isso, foi utilizado o livro *Bio-inspired Algorithms for the Vehicle Routing Problem*[14].

A proposta inicial pretendia implementar algoritmos como o Força Bruta, *branch-and-bound*, algoritmos genéticos e bioinspirados. Porém, a implementação de soluções para o TSP consumiu a

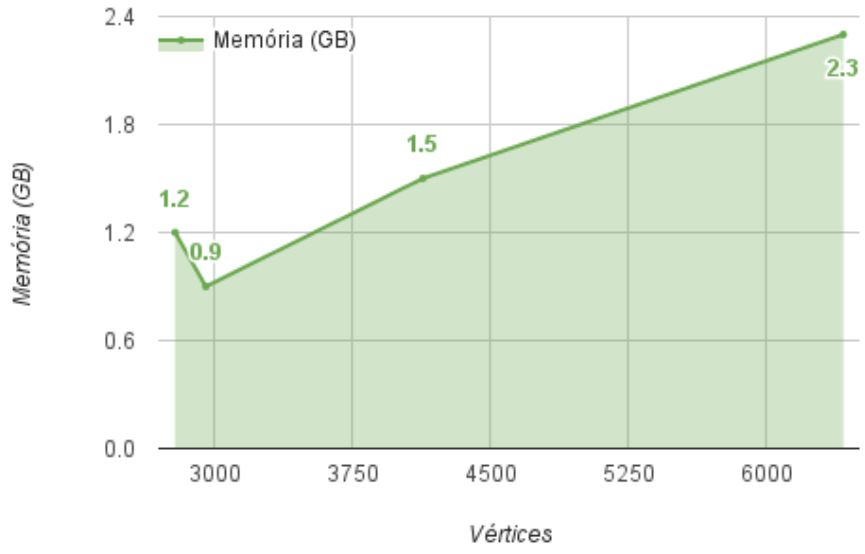

Figura 6: Relação entre vértices e uso de memória durante execução do programa.

maior parte do tempo, deixando o tempo e cronograma do projeto mais apertado. Por conta disso, optamos por dedicar esse tempo para estudar um pouco mais sobre o VRP e implementar o *Clarke & Wright's Savings Algorithm*[1].

Durante esse período, foi estudado sobre algumas variantes do VRP, como o VRPTW (*Vehicle Routing Problem with Time Windows*) onde uma janela de tempo é associada a cada localização, VRPPD (*Vehicle Routing Problem with Pickup and Delivery*) onde cada cliente pode estar associado a dois tipos de demanda: demanda de captação e demanda de entrega. E o CVRP (*Capacitated Vehicle Routing Problem*), onde uma frota fixa de veículos de entrega com capacidade uniforme deve atender as demandas de clientes que possuem um depósito em comum.

Além disso, boa parte desse período de estudos foi dedicado para entender as diferentes heurísticas e algoritmos que podem ser aplicados para resolver esse tipo de problema. Dentre essas heurísticas podemos citar dois grandes grupos: *Classical Heuristics* e *Metaheuristics*.

Dentro do primeiro grupo encontramos heurísticas de construção, que apresentam uma solução rápida e com resultado razoável, e heurísticas de melhoria que, como o próprio nome diz, procura melhorar a solução através de algoritmos de busca que podem envolver movimento e troca de vértices.

As Metaheurísticas apresentam uma grande variedade de soluções/algoritmos para resolver o VRP. Essas soluções apresentam melhores resultados se comparado com as heurísticas clássicas,

Figura 7: Resultado obtido após execução do algoritmo de Christofides seguido do *single-swap*.

visto que elas permitem estratégias globais de busca, podendo ser adaptadas a uma classe particular de problemas. Algumas metaheurísticas possuem um alto nível de abstração com relação à processos observados na natureza. Dentre essas metaheurísticas, as mais conhecidas são: *Tabu Search*, *Genetic Algorithm*, *Neural Networks* e *Ant Colony*.

Tabu Search: restringe áreas recentemente visitadas do "espaço de pesquisa", com o objetivo de manter uma memória de curto prazo referente aos movimentos recentes e prevenir que futuros movimentos desfaçam essas mudanças.

Genetic Algorithm: utiliza conceitos de processos genéticos encontrados na natureza e os aplica em problemas de otimização combinatória. De forma resumida, uma população de soluções evolui com o passar de "gerações", que ocorrem à partir de processos como *selection*, *crossover* e *mutation*.

Neural Networks: é um campo da Inteligência Artificial que usa estrutura de dados e algoritmos relacionados à aprendizado e classificação de dados, de forma a tentar simular processos parecidos que ocorrem no cérebro humano. Para isso, o algoritmo utiliza o conceito de *neurons* que funci-

ona de forma parecida com os neurônios do nosso cérebro. Esses neurônios podem aprender por experiência e, ao receber um conjunto de dados, eles conseguem se auto-ajustar para produzir o resultado desejado.

Ant Colony: é uma técnica que se baseia no comportamento real de uma colônia de formigas. A idea principal dessa heurística envolve o conceito de *pheromone*, que é uma substância produzida pelas formigas que serve para marcar o caminho onde elas podem encontrar comida de melhor qualidade. Essa heurística aplicada ao problema do VRP, relaciona cada formiga com uma localização inicial no grafo e cada aresta recebe uma quantidade inicial de *pheromone*. A cada iteração, uma formiga irá construir uma rota e a quantidade de *pheromone* em cada aresta é atualizado. No final, a melhor rota é construída com base no valor de *pheromone* encontrado nas arestas.

Implementação de *Savings Heuristic*

Após esse período de estudos, foi discutido sobre todo o conteúdo visto e decidimos começar com a implementação de uma heurística de construção. A heurística escolhida foi a *Savings Heuristic*, que é uma das abordagens mais conhecidas para solucionar o problema do VRP. Mais especificamente, utilizamos o *Clarke & Wright's Savings Algorithm*[1]. Apesar de não se ter certeza que será apresentado uma solução ótima, este algoritmo frequentemente apresenta uma boa solução para o problema. Que pode posteriormente ser melhorada com o uso de uma heurística de melhoria.

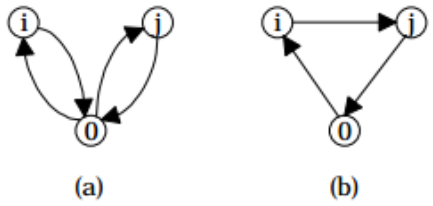

Figura 8: Ilustração do conceito de *Savings*

O conceito de *savings* refere-se ao custo da união entre duas rotas em apenas uma única rota. O custo de transporte da Figura 8a pode ser representado por:

$$D_a = C_{0i} + C_{i0} + C_{0j} + C_{j0}$$

Após unir as duas rotas, como mostra a Figura 8b, temos o seguinte custo:

$$D_b = C_{0i} + C_{ij} + C_{j0}$$

Dessa forma, o calculo do *savings* é dado por:

$$S_{ij} = D_a - D_b = C_{i0} + C_{0j} - C_{ij}$$

Quanto maior o valor do *savings*, maior é a "atração" entre os pontos i e j, ou seja, a possibilidade de se visitar os pontos i e j sequencialmente na mesma rota é mais interessante em termo de custo. O algoritmo de *savings* pode ser implementado em duas versões diferentes: sequencial e paralela. Na versão sequencial, apenas uma rota é criada por vez enquanto na versão paralela, mais de uma rota é criada por vez.

Para implementação do algoritmo, seguimos as seguintes etapas:

1. Calcular o valor do *savings* para todos os pares de clientes
2. Ordenar os *savings* em ordem decrescente
3. Um par de pontos é considerado por vez (topo da lista)
4. Quando um par de pontos i-j é considerado, as duas rotas que visitam i e j são combinadas, se:
   (a) essa união não destrua a conexão previamente estabelecida entre dois clientes
   (b) a demanda total não exceda a capacidade do veículo
5. Na versão sequencial, a iteração começa novamente do início da lista.

Para esse projeto foi feito algumas adaptações de forma a poder utilizar essa heurística. Uma nova classe foi criada para tratar do conceito de rota. Nesse caso, não conseguimos reutilizar a classe *Route.java* criada anteriormente, pois tivemos que trabalhar com algumas características mais específicas para essa heurística. Nessa nova classe, utilizamos a estrutura de dados *ArrayDeque*, que usa o conceito de fila de duas pontas, o que permite explorar de forma mais fácil o início e fim de uma lista. Usamos também uma variável para controlar a demanda total da rota. Por esse motivo, a classe *SavingsHeuristicRoute.java* foi criada.

Criamos também a classe *SavingsHeuristic.java*, que contém todo o algoritmo necessário para trabalharmos com ambas as versões do algoritmo de *savings*. Dentre as adaptações que tivemos que fazer, podemos citar a geração randômica de valores referente às demandas dos clientes. Tivemos que fazer isso, pois estamos usando instâncias referentes à localizações geográficas, e não à clientes. Dessa forma, cada localização representa um cliente, que por si só, possui um valor de demanda.

Para gerar esse valor, criamos uma regra simples, onde esse valor será no máximo 40% e no mínimo 20% da capacidade do veículo. A capacidade do veículo é passada como parâmetro no

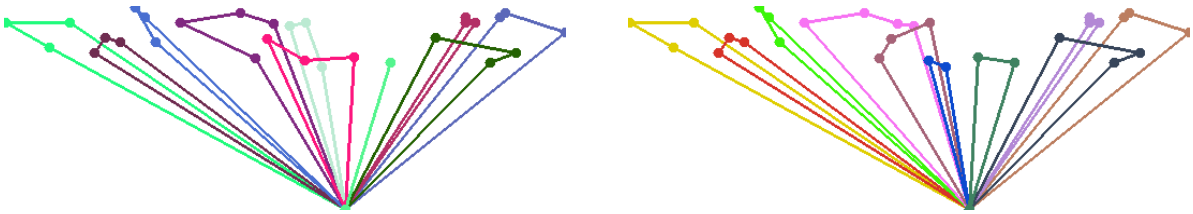

Figura 9: Versão sequencial e paralela do algoritmo de *savings* com 30 localizações.

momento da criação de um objeto dessa classe. De forma a obter acesso mais rápido ao valor de demandas, optamos por utilizar a estrutura de dados *HashMap*. Além disso, consideramos o depósito como sendo o primeiro elemento da lista de localizações. E implementamos o conceito de *saving* como uma classe à parte, chamada *Saving.java*, que contém os vértices i e j, além do próprio valor de *saving*.

A primeira etapa da implementação consiste no cálculo do valor dos *savings*. Para realizar esse cálculo, usamos o conceito de máquina de estados novamente. Logo após esse cálculo, ordenamos a lista de *savings* em ordem decrescente, como pede a etapa 2. As etapas seguintes diferenciam-se um pouco de acordo com a versão do algoritmo de *savings*. Na versão sequencial, o laço de repetição tem como condição de parada a não existência de vértices na lista. A cada iteração, uma rota é criada respeitando as condições estabelecidas nas etapas 4a e 4b. No final, a solução é representada por uma lista de rotas, onde cada rota corresponde a um conjunto de clientes que foram visitados por um veículo.

A implementação da versão paralela foi um pouco diferente, visto que teríamos que lidar com várias rotas sendo criadas e alteradas em uma única vez. Para facilitar a implementação, trabalhamos com uma estrutura de dados *HashMap*, onde estabelecemos uma ligação entre cada vértice e a rota em que ele estava presente. Dessa forma, ficou mais fácil manter o controle sobre todas as rotas sendo criadas e alteradas.

Para os resultados à seguir, foi utilizado 100 unidades como valor da capacidade do veículo. A Figura 9 mostra o resultado obtido ao rodar os algoritmos na versão sequencial e paralela, respectivamente, utilizando 30 localizações.

Em ambos os casos, foram geradas 10 rotas e o algoritmo rodou em menos de um segundo.

Ao utilizar uma instância de 200 localizações, ambas as versões apresentaram resultados muito parecidos. A versão sequencial gerou 64 rotas, enquanto a paralela gerou 65. O algoritmo executou em 1 segundo para ambos os casos. A Figura 10 mostra a imagem das soluções.

O último teste foi realizado com as localizações do estado de SP, referente à instância Brasil28k(1719 localidades). O resultado obtido ao rodar a versão sequencial, apresentou 556

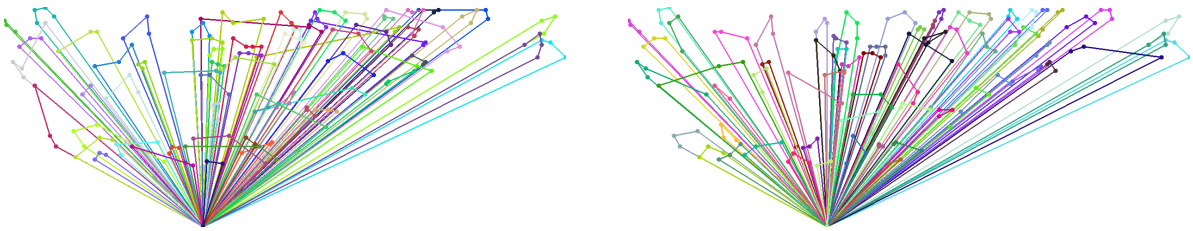

Figura 10: Versão sequencial e paralela do algoritmo de *savings* com 200 localizações.

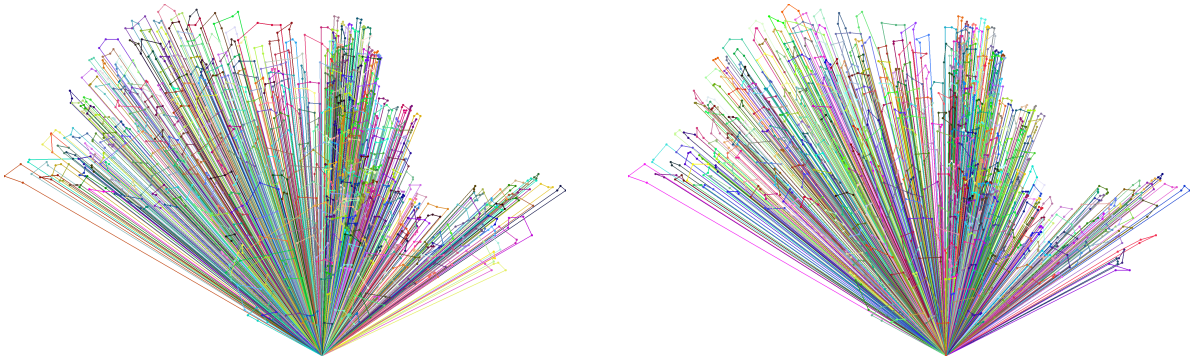

Figura 11: Versão sequencial e paralela do algoritmo de *savings* com 1719 localizações (instância do estado de São Paulo)

.

rotas e o algoritmo rodou em 1 minuto e 19 segundos. A versão paralela gerou 552 rotas e o algoritmo rodou em 5 segundos. A Figura 11 mostra a imagem das soluções.

Durante esse período de implementação, foi dedicado alguns dias apenas para estudar sobre Modularização e aplicar esse conceito na classe *SavingsHeuristic.java*. Algumas funções desta classe estavam com muitas linhas de código, começando a repetir trechos de código e consequentemente difícil de compreender em determinados pontos. Por esse motivo, fui aconselhado por meu Orientador a dedicar um tempo para entender e utilizar esse conceito. Como resultado a classe ficou mais organizada, com métodos menores e concisos. Além disso, este tipo de conhecimento pode ser utilizado em diversas outras situações, sejam elas em âmbito acadêmico, pessoal ou profissional.

5 Conclusões

Propôs-se neste Projeto de Pesquisa desenvolver algoritmos para o VRP e consequentemente, para o TSP, de modo a fazer uma análise comparativa entre a qualidade da solução e o tempo de processamento de cada abordagem. Dentre os principais algoritmos implementados, podemos citar a Árvore Geradora Mínima, *Christofides* e *Savings Heuristic*, além de outras heurísticas como o *hill climbing*, *single swap* e *minimal matching*.

Através dos testes realizados para o TSP com as regiões do Brasil pode-se observar que à medida que aumenta-se o número de vértices, o tempo de execução e o uso de memória aumentam consideravelmente, como era esperado. À partir desses valores é possível ter uma ideia dos recursos de hardware necessários para que se possa replicar os testes.

Os testes realizados para o VRP foram mais simples, pois implementamos apenas o algoritmo de savings devido ao tempo de execução do projeto ter chegado ao fim. Mesmo assim, conseguimos comparar as duas versões desse algoritmo: sequencial e paralela. Através dos resultados é possível observar que as instâncias menores, de 30 e 200 vértices, apresentaram uma solução razoável se comparado com o tempo de execução. Ambas as instâncias apresentaram resultados parecidos.

Já o teste realizado com a instância do estado de São Paulo, apresentou um resultado de comparação interessante com relação ao tempo de execução entre as duas versões do algoritmo de *savings*. Enquanto a versão sequencial executou em 1 minuto e 19 segundos, a versão paralela executou em apenas 5 segundos. À medida que a se aumenta o número de vértices, essa diferença de tempo de execução entre as duas versões também aumenta.

Há muitas coisas que ainda podem ser exploradas dentro desse problema. A aplicação de heurísticas mais complexas, como o Algoritmo Genético e Colônia de Formigas, por exemplo, é algo que pretendo explorar em trabalhos futuros.

6 Perspectivas de continuidade

O VRP é um problema interessante e desafiador, que abrange diversos assuntos e pode ser aplicado à diferentes situações. Seria muito difícil explorar todos esses detalhes minuciosamente, visto que a maior parte desse projeto foi dedicada para resolução do TSP.

Por conta desses motivos, eu quero continuar explorando esse problema à partir do Trabalho de Conclusão de Curso (TCC). Há muitas coisas que ainda posso explorar dentro desse problema, desde aplicações de heurísticas mais complexas, ou até mesmo otimizar os algoritmos que já implementamos.

Aqui estão algumas coisas que podem ser exploradas em trabalhos futuros:

- Tentar otimizar o algoritmo para funcionar com instâncias maiores
- Utilizar dados reais de empresas e aplicá-los aos algoritmos implementados nesse projeto
- Implementar metaheurísticas, como *Genetic Algorithm*, *Neural Networks* e *Ant colony*, para que se possa fazer uma comparação com os resultados encontrados nesse projeto

7 Apoio

Este projeto de Iniciação Científica teve apoio da Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), processo n°2016/08481-6.

8 Agradecimentos

Esse projeto de Iniciação Científica abriu minha mente para muitas coisas que eu nunca tive contato antes. Eu adquiri conhecimento sobre escrita acadêmica, ciência da computação, boas práticas de programação, algoritmos, até mesmo à respeito de dedicação, prazo, entre outras coisas. Boa parte desse conhecimento, dificilmente eu conseguiria adquirir apenas com as disciplinas da grade curricular de estudos.

Além disso, gostaria de agradecer a meu orientador, Luis Augusto Angelotti Meira, que foi extremamente atencioso durante toda a execução desse projeto de Iniciação Científica. Com ele, pude aprender desde conceitos específicos da área de Otimização até assuntos dos mais variados possíveis.

Referências

[1] Clarke & wright's savings algorithm. page 7, 1997.

[2] David Applegate, William Cook, and André Rohe. Chained lin-kernighan for large traveling salesman problems, 1999.

[3] David L Applegate. *The traveling salesman problem: a computational study*. Princeton University Press, 2006.

[4] M.H. Carvalho, M.R. Cerioli, R. Dahab, P. Feofiloff, C.G. Fernandes, C.E. Ferreira, K.S. Guimarães, F.K. Miyazawa, J.C. Pina J. Soares, and Y. Wabayashi. *Uma Introdução Sucinta a Algoritmos de Aproximação*. IMPA, Instituto de Matemática Pura e Aplicada, 2001.

[5] N. Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical Report 388, Carnegie-Mellon University, 1976.

[6] Nicos Christofides. The vehicle routing problem. *Revue française d'automatique, d'informatique et de recherche opérationnelle. Recherche opérationnelle*, 10(1):55–70, 1976.

[7] William Cook. *In pursuit of the traveling salesman: mathematics at the limits of computation*. Princeton University Press, 2012.

[8] P. Crescenzi and V. Kann. *A compendium of NP optimization problems*, 2000. http://www.nada.kth.se/˜viggo/wwwcompendium/.

[9] George B Dantzig and John H Ramser. The truck dispatching problem. *Management science*, 6(1):80–91, 1959.

[10] Jorge Tavares Francisco Baptista Pereira, editor. *Bio-inspired Algorithms for the Vehicle Routing Problem*, volume 161. Springer.

[11] Ricardo Fukasawa, Humberto Longo, Jens Lysgaard, Marcus Poggi de Aragão, Marcelo Reis, Eduardo Uchoa, and Renato F Werneck. Robust branch-and-cut-and-price for the capacitated vehicle routing problem. *Mathematical programming*, 106(3):491–511, 2006.

[12] Brian Kallehauge, Jesper Larsen, Oli BG Madsen, and Marius M Solomon. *Vehicle routing problem with time windows*. Springer, 2005.

[13] Eugene L Lawler, Jan Karel Lenstra, AHG Rinnooy Kan, and David B Shmoys. *The traveling salesman problem: a guided tour of combinatorial optimization*, volume 3. Wiley New York, 1985.

[14] Jacques Renaud, Gilbert Laporte, and Fayez F Boctor. A tabu search heuristic for the multi-depot vehicle routing problem. *Computers & Operations Research*, 23(3):229–235, 1996.

[15] Paolo Toth and Daniele Vigo. *The vehicle routing problem*. Siam, 2001.