

Victor Hugo Carlquist da Silva

*Desenvolvimento de um Algoritmo
Genético Otimizado para a solução do
problema dos Múltiplos Caixeiros
Viajantes ($mTSP$)*

Campos do Jordão, SP

27 de maio de 2014

Victor Hugo Carlquist da Silva

*Desenvolvimento de um Algoritmo
Genético Otimizado para a solução do
problema dos Múltiplos Caixeiros
Viajantes ($mTSP$)*

Proposta de Trabalho de Conclusão de Curso
do 5º semestre do curso Tecnologia em Aná-
lise e Desenvolvimento de Sistemas na maté-
ria Projeto Integrado I.

Orientador:
Prof. Me. André Malvezzi Lopes

Co-orientador:
Prof. Dr. Silvio Alexandre de Araujo

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE SÃO PAULO –
campus CAMPOS DO JORDÃO

Campos do Jordão, SP

27 de maio de 2014

VICTOR HUGO CARLQUIST DA SILVA

Desenvolvimento de um Algoritmo Genético Otimizado para a solução do problema dos Múltiplos Caixeiros Viajantes (*mTSP*)

Proposta de Trabalho de Conclusão de Curso do 5º semestre do curso Tecnologia em Análise e Desenvolvimento de Sistemas na matéria Projeto Integrado I.

BANCA EXAMINADORA

10 de junho de 2014

Prof. Alisson Ribeiro
Instituto Federal de Educação, Ciência e
Tecnologia

Profª. Dra. Thalita Biazzuz Veronese
Instituto Federal de Educação, Ciência e
Tecnologia

Prof. Paulo Giovani de Faria Zeferino
Instituto Federal de Educação, Ciência e
Tecnologia

LOCAL

Instituto Federal de Educação, Ciência e Tecnologia de São Paulo – IFSP
Campus Campos do Jordão
Campos do Jordão, SP

Sumário

Lista de Figuras

Resumo

Abstract

1	Introdução	p. 8
2	Metodologia	p. 9
3	Problema do Caixeiro Viajante - TSP)	p. 10
3.1	Problema dos Múltiplos Caixeiros Viajantes	p. 11
4	Algoritmos genéticos	p. 12
4.1	Operadores de cruzamento	p. 13
4.1.1	Cruzamento de Mapeamento Parcial - PMX	p. 13
4.2	Operador de mutação	p. 14
4.3	Algoritmo Genético Híbrido	p. 14
5	Estado da Arte	p. 15
6	Algoritmo <i>Nearest-Neighbor</i> (NN)	p. 16
7	Desenvolvimento	p. 17
7.1	Estrutura do cromossomo (indivíduo) - <i>two-part</i>	p. 17
7.2	Avaliação do Indivíduo (<i>fitness</i>)	p. 18

7.3	Cruzamento	p. 18
7.4	Mutação	p. 18
7.5	Parâmetros	p. 18
7.6	Desempenho	p. 19
8	Conclusão	p. 26
Referências Bibliográficas		p. 27
Apêndice A – Código Fonte - main.cpp		p. 28
Apêndice B – Código Fonte - ga.h		p. 31
Apêndice C – Código Fonte - ga.cpp		p. 33

Lista de Figuras

4.1	PMX - cruzamento	p. 13
4.2	PMX - preenchimento	p. 14
7.1	Cromossomo - <i>two-part</i>	p. 17
7.2	Resultado utilizando o algoritmo NN e AG - 4 caixeiros.	p. 20
7.3	Resultado utilizando o AG tradicional com 4 caixeiros.	p. 21
7.4	Resultado utilizando o AG tradicional com 1 caixeiro.	p. 23
7.5	Resultado utilizando o AG híbrido com 1 caixeiro.	p. 23
7.6	Resultado TSPLIB.	p. 24
7.7	Resultado utilizando AG tradicional com 4 caixeiros (Argentina). . .	p. 25
7.8	Resultado utilizando AG Híbrido com 4 caixeiros (Argentina). . . .	p. 25

Resumo

A alta complexidade para definir a melhor rota para diversos veículos estimula o desenvolvimento de novos algoritmos computacionais para resolver este problema. Este problema pode ser definido como o problema dos múltiplos caixeiros viajantes (*Multiple Traveling Salesman Problem* (MTSP)). Portanto, foi proposto um algoritmo Algoritmo Genético otimizado, dividindo o espaço dos objetivos para otimizar o cruzamento dos indivíduos, diminuindo assim o número de gerações necessárias para conseguir uma rota ótimo, ou quase ótima.

Abstract

The high complexity to define the best route for multiple vehicles motivate a development for a new algorithm to solve this problem. This problem is known as multiple Traveling Salesman Problem (*mTSP*). It was created a hybrid algorithm using Genetic Algorithm with Dijkstra's Algorithm. Executing and analysing results of the software, it shows very efficient to solve this problem.

1 *Introdução*

Hoje em dia o transporte veicular tornou-se algo de suma importância, impactando tanto positivamente quanto negativamente na sociedade e no meio ambiente, principalmente na economia mundial.

Apesar de agilizar o transporte de pessoas e de mercadorias, os veículos também geram despesas com combustível, manutenção, etc. Se um veículo percorrer uma menor rota, a empresa diminui custos, com, por exemplo, combustível, manutenção e tempo de entrega.

Cada vez as cidades estão ficando maiores e a complexidade rodoviária cresce, dificultando a análise da melhor rota a se percorrer. Esse problema tende a ficar mais complexo conforme aumenta a quantidade de veículos que a empresa possui. Com isso, surge a necessidade de criar *softwares* cada vez mais robustos que resolvam o problema de roteirização de veículos, encontrando o melhor caminho para chegar aos seus objetivos.

2 Metodologia

A metodologia utilizada para o desenvolvimento desta pesquisa é de natureza aplicada e exploratória, sendo baseada em referências bibliográficas e em experimentos. Para os experimentos e comparações, será utilizado a base de testes chamada TSPLIB desenvolvida para o Problema do Caixeiro Viajante (TSP), disponível em <http://www.math.uwaterloo.ca/tsp/world/countries.html>, já que não foi possível encontrar uma base de testes específica para o MTSP. Esta base se baseia em informações geográficas do *National Imagery and Mapping Agency* (RESEARCH, 2014).

O objetivo é desenvolver um novo algoritmo, que encontre a rota ótima, ou quase ótima, para múltiplos caixeiros viajantes, consumindo menos recurso computacional, mais especificamente o tempo de execução. O experimento será empírico, utilizando diversas configurações de cenário, comparando o tamanho das rotas e o tempo de execução no Algoritmo Genético “tradicional” com o algoritmo proposto.

3 Problema do Caixeiro Viajante - TSP)

O problema do Caixeiro Viajante (*Traveling Salesman Problem* - TSP) consiste em estabelecer uma rota que passe por cada vértice do grafo apenas uma vez e retorne ao vértice de partida. O número de rotas possíveis pode ser expressa por $(n - 1)!$, sendo n o número de pontos. O problema TSP é classificado como *NP-Hard*¹, ou seja, não existe algoritmo com limitação polinomial capaz de resolvê-lo (BENEVIDES et al., 2012).

Este problema pode ser formulado da seguinte forma:

$$\min \sum_{i=1}^n \sum_{j=1}^n c_{ij} x_{ij} \quad (3.1)$$

então

$$\sum_{i=1}^n x_{ij} = 1 \quad j = 1, \dots, n \quad (3.2)$$

$$\sum_{j=1}^n x_{ij} = 1 \quad i = 1, \dots, n \quad (3.3)$$

$$\{(i, j) | i, j = 2, \dots, n; x_{ij} = 1\} \text{ não contem sub-rotas}, \quad (3.4)$$

$$x_{ij} \in \{0, 1\} \forall i, j = 1, \dots, n \quad (3.5)$$

Para um grafo $G = (V, A)$, onde V é o conjunto de vértices e A é o conjunto de arestas, sendo $C = (C_{ij})$ a matriz que corresponde ao custo associado com A . A matriz C é simétrica caso $c_{ij} = c_{ji}, \forall (i, j) \in A$ e assimétrica caso contrário. A variável x_{ij} é binária, usado para indicar se a aresta foi usada na rota. As equações 3.2 e 3.3, criam uma restrição, para que haja um caminho de entrada e um caminho de saída para cada cidade. Já a equação 3.4 previne sub-rotas, ou seja, não permite um grafo desconexo (CARTER, 2003).

¹(ARNBORG; PROSKUROWSKI, 1989)

3.1 Problema dos Múltiplos Caixeiros Viajantes

O problema dos Múltiplos Caixeiros Viajantes (*Multiple Traveling Salesman Problem - mTSP*) consiste em estabelecer várias rotas, uma para cada caixeiro , sendo que cada um pode estar em lugares diferentes, passando por cada vértice do grafo apenas uma vez e retornando ao seu vértice de partida.

4 Algoritmos genéticos

Segundo Correia (2003), os Algoritmos Genéticos (AG) são técnicas de procura e otimização baseados em mecanismos de seleção natural.

Nas décadas de 60 e 70, John Holland e seus colegas da Universidade de Michigan criaram modelos para estudar o processo de adaptação dos seres vivos. Holland realizou diversas pesquisas e em 1975 publicou o seu livro intitulado *Adaptation in Natural and Artificial System*. Hoje, este livro é considerado um dos mais importantes sobre Algoritmos Genéticos (CARVALHO, 2013).

No Algoritmo Genético, o cromossomo é uma estrutura de dados (um conjunto de genes) que armazena uma possível solução de um problema. Sendo que cada indivíduo tem um cromossomo, os indivíduos são cruzados gerando novos indivíduos, conforme a população cresce, surgem indivíduos cada vez mais aptos, sendo que um deles será o mais apto, contendo no seu cromossomo a solução do problema.

O Algoritmo Genético possui alguns parâmetros importantes para configurar a evolução dos indivíduos, tais como, o tamanho da população, que indica quantos indivíduos existirão para realizar o cruzamento. Se o número da população for pequena deixará o desempenho da solução lenta, pois terá um pequeno conjunto para a busca de solução do problema. Já uma população muito grande pode afetar o desempenho do algoritmo. O outro parâmetro é a **taxa de cruzamento**, que controla as chances de ocorrer um cruzamento, gerando novos indivíduos. E por último, a **taxa de mutação**, que define as chances de um cromossomo sofrer mutação. Uma alta taxa de mutação irá deixar o algoritmo aleatório, mas com uma baixa taxa previne que os indivíduos sejam sempre os mesmos.

4.1 Operadores de cruzamento

Os operadores de cruzamento definem como ocorrerá o cruzamento entre dois indivíduos (MALAQUIAS, 2006).

Neste trabalho será abordado o Cruzamento de Mapeamento Parcial, que foi utilizado no software.

4.1.1 Cruzamento de Mapeamento Parcial - PMX

O operador de Cruzamento de Mapeamento Parcial (*Partially-mapped crossover - PMX*) seleciona e copia três genes do pai para um filho e completa o restante cromossomo com os genes do outro pai, como a **figura 4.1** demonstra.

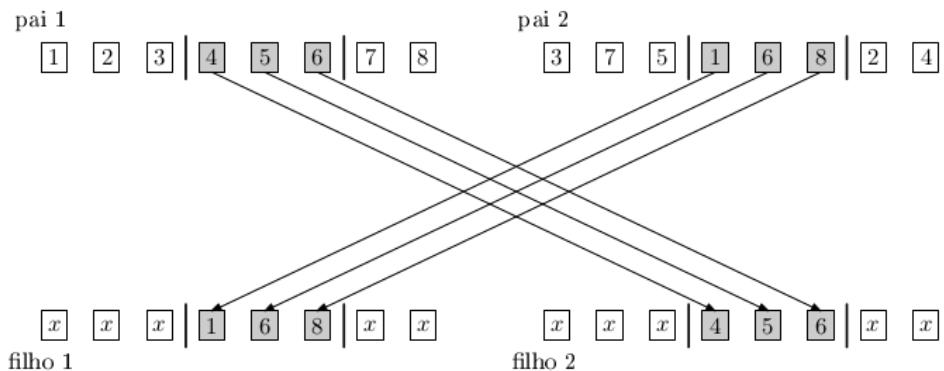


Figura 4.1: PMX - cruzamento

Fonte:(MALAQUIAS, 2006).

Caso o cromossomo já possua o mesmo número, é escolhido outro número com o mesmo índice do pai que não esteja no cromossomo (**figura 4.2**):

Por exemplo, como o “filho 1” já possui o número 1, então pesquisa-se em qual posição do número 1 está neste vetor, no caso da **figura 4.2** o número 1 está na posição 4, então copia-se o número que está na posição 4 no vetor do “pai 1” para o filho. Já no outro exemplo na mesma figura, na terceira posição do vetor do “filho 2”, tenta-se preencher com o número 5, mas, como o número 5 já existe no vetor do “filho 2”, então pega-se o valor na mesma posição que está o número 5, que no caso é o número 6, mas o número 6 também já existe, e sua posição remete ao número 8 na sexta posição, e finalmente é preenchido a terceira posição. Este processo descrito acima é repetido para todos os outros elementos vazios dos filhos.

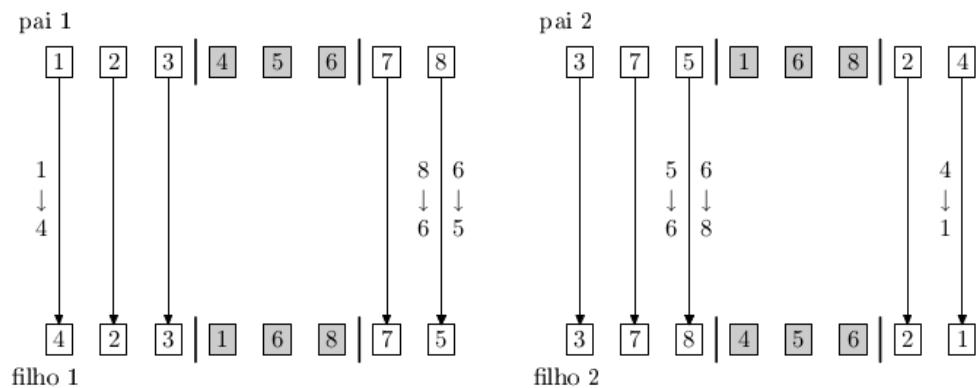


Figura 4.2: PMX - preenchimento

Fonte:(MALAQUIAS, 2006).

4.2 Operador de mutação

O operador de mutação define como será realizada a mutação de um cromossomo, impedindo que o programa sempre gere os mesmos indivíduos (cromossomos) (MALAQUIAS, 2006).

Foi implementado no algoritmo proposto o operador de mutação por troca *exchange mutation* (EM). Ele seleciona dois genes, aleatoriamente, do cromossomo e os troca de posição.

4.3 Algoritmo Genético Híbrido

Os Algoritmos Genéticos possuem o objetivo de serem robusto, mas, estes algoritmos têm dificuldade em encontrar o caminho ótimo, devido ao processo de avaliação do cromossomo e realização de bons cruzamentos. Para solucionar esse problema foi criado os Algoritmos Genéticos Híbridos.

Os Algoritmos Genéticos Híbridos consistem em utilizar um outro método em conjunto com o Algoritmo Genético, produzindo algoritmos eficientes na prática.

5 Estado da Arte

Existem diversos trabalhos sobre a utilização de Algoritmos Genéticos na resolução do problema do TSP. Segundo Pacheco e Fukasawa (2010), que criou uma implementação para resolver este problema, mostrou que os Algoritmos Genéticos não são eficientes na resolução do TSP em comparação com métodos exatos. Neste trabalho o AG levou mais tempo e não encontrou a solução ótima em comparação com algoritmos exatos.

A utilização de AGs mostrou-se mais eficiente na resolução do *mTSP* do que na resolução do TSP, levando-se em conta um maior tempo de processamento para o caso mais complexo.

A solução apresentada por Belfiore (2006) propõe resolver os problemas de roteirização de veículos com entregas fracionadas, problema clássico de roteirização de veículos e com frota heterogênea criando o algoritmo de roteirização de veículos com frota heterogênea, restrições de janelas de tempo e entregas fracionadas(*Heterogeneous Fleet Vehicle Routing Problem with Time Windows and Split Deliveries* (HFVRPTWSD)) utilizando Algoritmo Genético (AG).

Na proposta apresentada por Sedighpour et al. (2011) para a resolução do *mTSP* com um depósito, foi criado um único cromossomo utilizando o método *two-part*, que será explicado na próxima seção. Este método mostrou-se muito eficiente.

O Wu (2007) mostra que é possível calcular as rotas de múltiplos veículos utilizando AG para igualar o tempo de espera de encomendas de clientes, sendo que a variável “menor tempo da rota” não é levada em consideração.

6 Algoritmo Nearest-Neighbor (NN)

O Algoritmo *Nearest-Neighbor* (NN) é classificado como um algoritmo guloso, ou seja, ele preza pela menor rota localmente, sem se preocupar com o desempenho da rota globalmente. O NN é um algoritmo simples de se implementar, sua única tarefa é verificar qual dos pontos ao seu redor esta mais próximo e escolhê-lo para a rota, após esse passo é repetido o mesmo processo para este ponto escolhido. Mas há um problema, este algoritmo possui uma complexidade muito alta, $O(n!)$, pois faz a comparação com todos os pontos ainda não visitados, sendo inviável sua utilização para muitos pontos (MIYAZAWA, 2002).

Para resolver este problema, o plano cartesiano foi dividido em 4 áreas de tamanhos iguais, sendo que o número de áreas pode ser alterado, com isso, pode-se aplicar o NN em cada área reduzindo assim a complexidade para, na melhor hipótese, $(\frac{n!}{k})k$, sendo n o número total de pontos e k o número de áreas e na pior hipótese $n!$.

O número de áreas sempre deve ser par, e o plano cartesiano sempre terá duas linhas, por exemplo, caso o número de áreas for definido em 8 partes, então a primeira linha do plano cartesiano terá 4 áreas de tamanhos iguais e a segundo linha também terá 4 áreas de tamanhos iguais totalizando 8 áreas.

7 Desenvolvimento

O Algoritmo Genético Híbrido proposto, utiliza os conceitos do Algoritmo Genético tradicional em conjunto com o algoritmo *Nearest-Neighbor* (NN). O algoritmo *Nearest-Neighbor* é utilizado para gerar apenas um indivíduo na população inicial, os outros indivíduos são gerados aleatoriamente.

7.1 Estrutura do cromossomo (indivíduo) - *two-part*

O método utilizado para definir a estrutura do cromossomo do programa foi o *two-part*. O *two-part* consistem em dividir o cromossomo em duas partes, uma parte armazena as informações da rota e a outra a informação dos caixeiros. Considere o cromossomo a seguir:

| 1 | 9 | 10 | 3 | 11 | 5 | 4 | 2 | 6 | 7 | 8 || 2 | 6 | 3 |

Os três últimos genes representam os caixeiros. Neste exemplo foi definido o ponto de partida como sendo o 0. Seguindo este raciocínio, o primeiro caixeiro terá que sair do ponto 0 e visitar dois pontos, ou seja, os pontos 1 e 9 e retornar ao ponto 0, já o segundo caixeiro terá que sair do ponto 0 e visitar seis pontos, os pontos 10, 3, 11, 5, 4 e 2 e retornar ao ponto 0 e o último caixeiro terá que sair do ponto 0 e visitar três pontos, os pontos 6, 7 e 8 e retornar ao ponto 0. (**figura 7.1**).

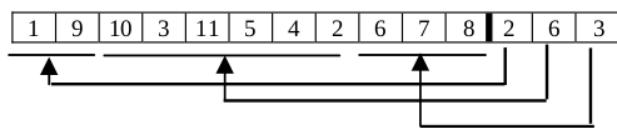


Figura 7.1: Cromossomo - *two-part*

Fonte: (SEDIGHPOUR et al., 2011).

7.2 Avaliação do Indivíduo (*fitness*)

Para cada novo indivíduo gerado, é calculado a distância dos pontos da rota, por meio da distância euclidiana, de cada caixeiro e somando-as, gerando o tamanho total da rota, quanto menor este valor, melhor é o indivíduo.

7.3 Cruzamento

O programa proposto utiliza o operador de cruzamento PMX, descrito no capítulo (4.1.1).

A cada iteração é realizado um cruzamento, para isso é selecionado, aleatoriamente, dois cromossomos da população. No cruzamento é gerado dois novos indivíduos, esses indivíduos serão avalizados pela função *fitness* e inseridos na população conforme sua avaliação.

7.4 Mutação

O programa proposto utiliza o operador de mutação EM (*Exchange Mutation*), descrito no capítulo 4.2. O indivíduo é escolhido aleatoriamente para receber a mutação.

7.5 Parâmetros

Segundo Carvalho (2013) os parâmetros são importantes para analisar o comportamento do algoritmo e ajustá-lo para suprir as necessidades do problema.

No algoritmo proposto foram implementados os seguintes parâmetros:

- **MaxPopulation:** Define o número máximo da população;
- **InitialPopulation:** Define a quantidade inicial de indivíduos dentro da população, as rotas destes indivíduos serão gerados aleatoriamente. Caso o parâmetro *InitialPopulationWithNN* estiver ativado, um indivíduo será gerado utilizando o algoritmo NN;
- **InitialPopulationWithNN:** Se for igual a *true* um indivíduo da população inicial será gerado utilizando o algoritmo NN;

- **MutationRouteItself:** Se for igual a *true* a mutação ocorrerá dentro da rota de um caixeiro viajante, ou seja, os pontos da rota de um caixeiro não poderão ser trocadas com outros caixeiros. Se for igual a *false* os pontos poderão ser trocados entre os caixeiros;
- **NNsizePart:** Define o tamanho da área que os pontos serão gerados, por exemplo, se está variável possuir o valor 100, então será criada um espaço 100x100;
- **NNnPart:** Define o número de áreas que o espaço será dividido para a utilização do algoritmo NN;
- **RateMutation:** Define a probabilidade de um indivíduo da população sofrer mutação;
- **RateGeneration:** Define a chance de adicionar ou substituir novos indivíduos na população, por exemplo, se o parâmetro *MaxPopulation* for maior que o valor do parâmetro *Initial Population*, então os dois novos indivíduos que serão gerados por meio do cruzamento terão chances de substituir um indivíduo da população, mesmo que o número máximo da população não tenha sido atingido, ou serão adicionados sem substituir nenhum indivíduo. Após a população atingir o número máximo permitido pelo parâmetro *MaxPopulation*, apenas ocorrerá substituição dos piores indivíduos;
- **RateSalesmanMutation:** Define a probabilidade de mutação no número de pontos que cada caixeiro irá visitar, ou seja, irá afetar a segunda parte do cromossomo (*two-part*);
- **Generation:** Define o número de iterações que o programa realizará;
- **Deposit:** Define qual ponto será o depósito no qual os caixeiros irão sair;
- **Salesman:** Define o número de caixeiros viajantes;
- **SaveBetterChromo:** Se for igual a *true*, o melhor indivíduo não poderá sofrer mutação;

7.6 Desempenho

Para mensurar o desempenho do algoritmo foram realizados diversos teste com parâmetros diferentes, levando em consideração o tamanho da rota e o tempo de execução.

O algoritmo foi implementado utilizando a linguagem C++ e os gráficos foram gerados pelo programa *gnuplot*. O computador que executou os testes possuía processador Intel®Core™ i5 2,4Hz, 4GB de memória RAM.

Foi gerado um cenário aleatoriamente com 600 pontos, executando 9000 gerações com 4 caixeiros, para demonstrar o desempenho do Algoritmo Genético tradicional e o desempenho do Algoritmo Híbrido.

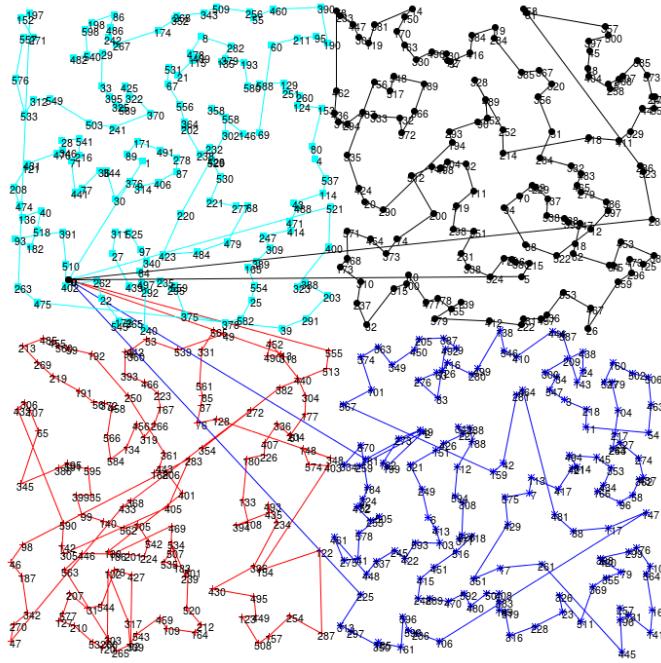


Figura 7.2: Resultado utilizando o algoritmo NN e AG - 4 caixeiros.

Utilizando este cenário foi executados os testes apresentados na **tabela 7.1**. Pode-se notar que utilizando algoritmo NN a distância da rota é bem reduzida se comparado ao algoritmo tradicional (Teste 4). O algoritmo tradicional gerou uma rota com tamanho de 273.202km (**figura 7.3**) e o Algoritmo Híbrido utilizando o *Nearest-Neighbor* gerou uma rota com tamanho de 26.629,6km (**figura 7.2**).

Foi utilizado um teste da base de dados do *TSPLIB*. O teste escolhido contém todas as cidades da Argentina (9152 cidades). A rota ótima para este problema é de 837.479km. O melhor algoritmo encontrou a rota ótima em 460.058 segundos (aproximadamente 5 dias) rodando em uma arquitetura EV6 Alpha 500 MHz. O teste utilizando o algoritmo tradicional gerou uma rota com tamanho de 73.298.400km (**figura 7.4**) e o Algoritmo Híbrido utilizando o *Nearest-Neighbor* gerou uma rota com tamanho de 1.120.070km (**figura 7.5**), ambos com apenas um caixeiro viajante. Isso demonstra que o algoritmo desenvolvido é eficiente para o problema com um único caixeiro viajante, pois conseguiu encontrar uma rota sa-

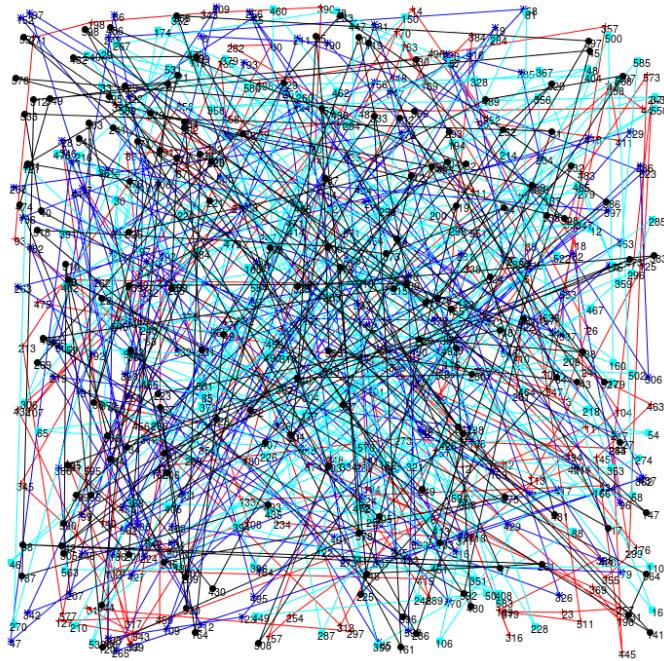


Figura 7.3: Resultado utilizando o AG tradicional com 4 caixeiros.

tisfatória em 708,913 segundos (aproximadamente 11,8 minutos) (**tabela 7.2**), também o resultado da rota gerado pelo Algoritmo Híbrido desenvolvido pelo autor (**figura 7.5**) e o resultado da rota da base de teste (**figura 7.6**) são bem semelhantes.

Também foram executados testes com 4 caixeiros viajantes. Os testes 3 (**figura 7.8**) e 4 (**figura 7.7**) da **tabela 7.2** demonstram que há uma grande diferença no tamanho das rotas geradas pelo Algoritmo Genético tradicional (73.338.600,00km) e o Algoritmo Genético Híbrido (1.145.840,00km).

	Teste 1	Teste 2	Teste 3	Teste 4
MaxPopulation	100	10	10	10
Initial Population	10	10	10	10
InitialPopulationWithNN	true	true	true	false
MutationRouteitself	true	true	true	true
NnsizePart	1000	1000	1000	1000
Nnnpart	4	4	4	4
RateMutation	0,8	0,8	0,8	0,8
RateGeneration	0,2	0,2	0,2	0,2
RateSalesmanMutation	0	0	0	0
Generation	9000	9000	10000	9000
Deposit	0	0	0	0
Salesman	4	4	4	4
SaveBetterChromo	false	false	false	false
Distance	26.652,3	26.645,3	26.629,6	273.202
Time (s)	62,0188	61,8399	68,8648	60,6324

Tabela 7.1: Teste com 600 pontos.

	Teste 1	Teste 2	Teste 3	Teste 4
MaxPopulation	4	4	4	4
Initial Population	2	2	2	2
InitialPopulationWithNN	true	false	false	true
MutationRouteitself	true	true	true	true
NnsizePart	71000	71000	71000	71000
Nnnpart	4	4	4	4
RateMutation	0,8	0,8	0,8	0,8
RateGeneration	0,6	0,6	0,6	0,6
RateSalesmanMutation	0	0	0	0
Generation	460	460	460	460
Deposit	0	0	0	0
Salesman	1	1	4	4
SaveBetterChromo	false	false	false	false
Distance	1.120.070,00	73.298.400,00	73.338.600,00	1.145.840,00
Time (s)	708,913	710,578	708,304	706,616

Tabela 7.2: Teste com 9152 pontos (Argentina).

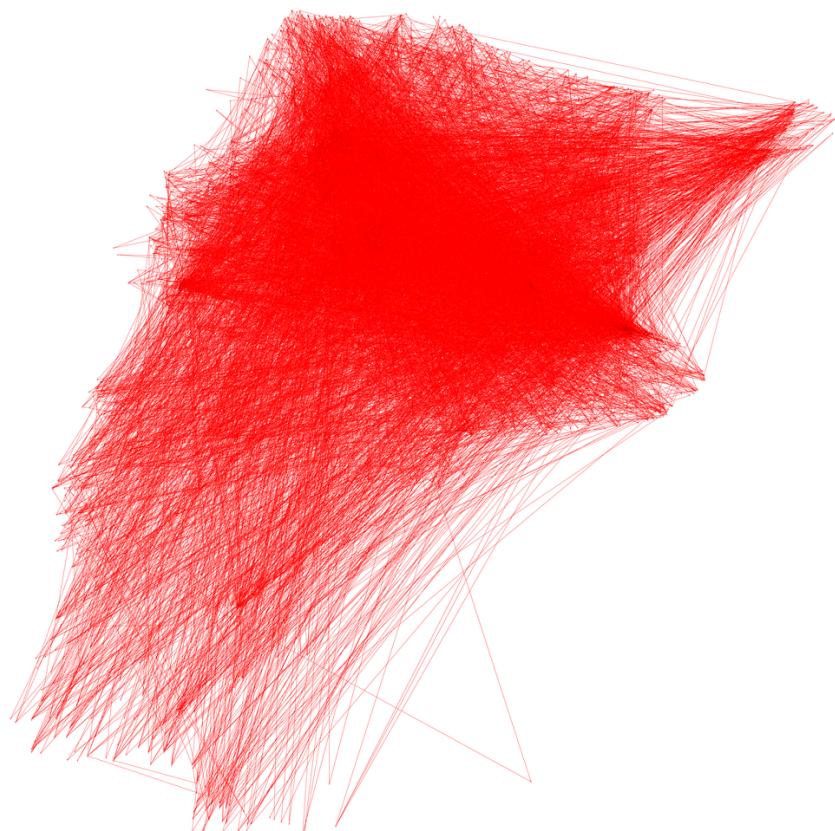


Figura 7.4: Resultado utilizando o AG tradicional com 1 caixeiro.



Figura 7.5: Resultado utilizando o AG híbrido com 1 caixeiro.

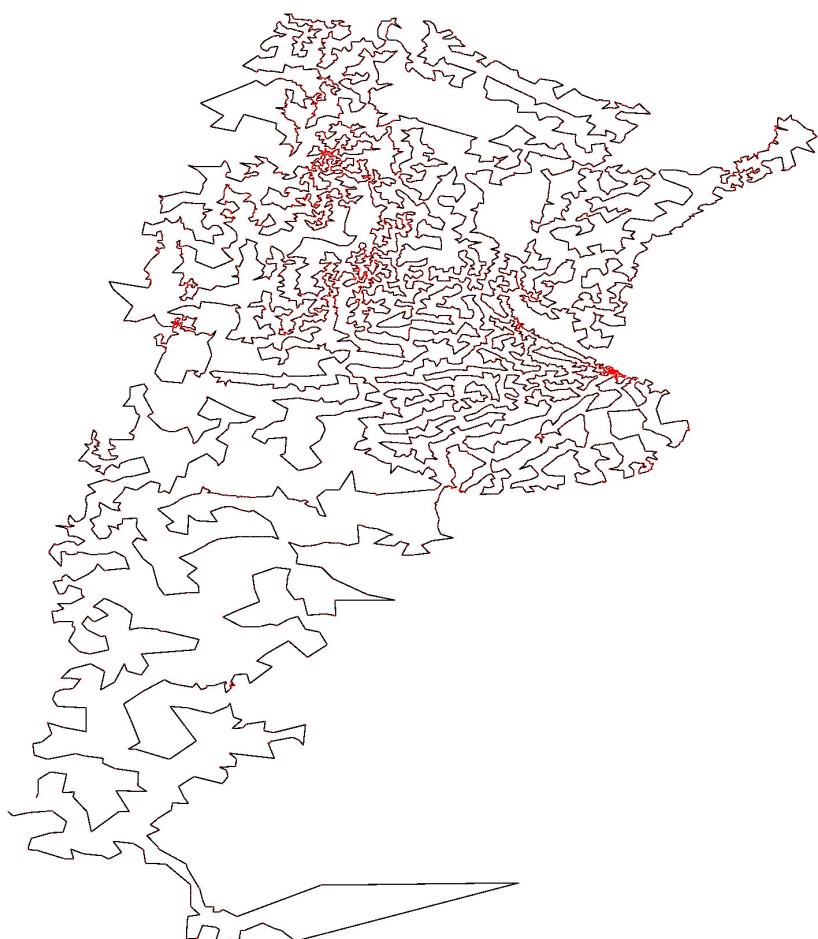


Figura 7.6: Resultado TSPLIB.

Fonte: (RESEARCH, 2014)

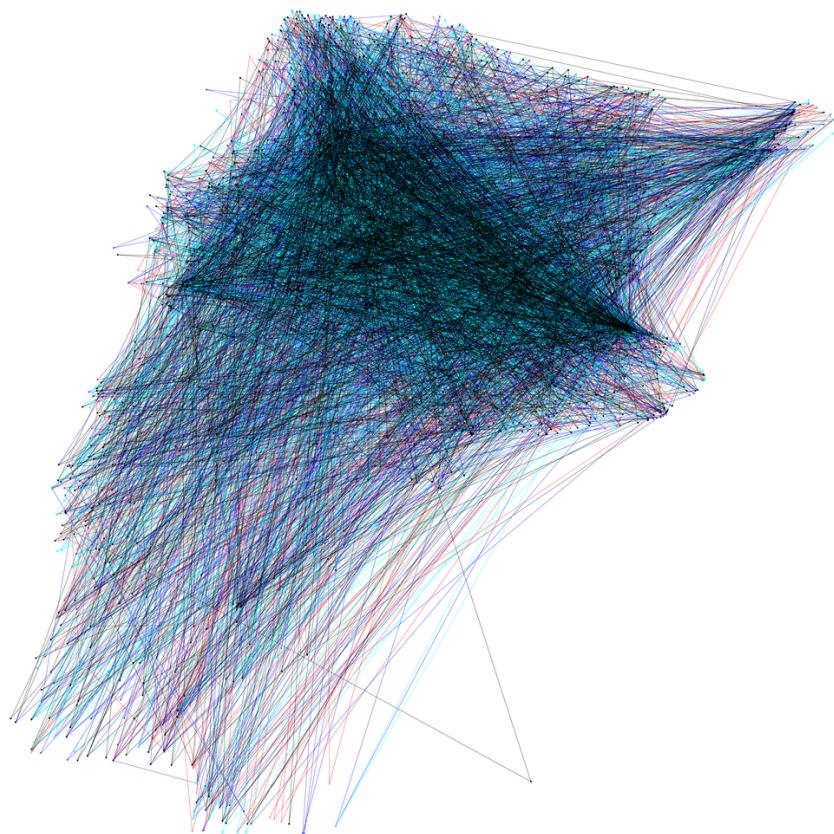


Figura 7.7: Resultado utilizando AG tradicional com 4 caixeiros (Argentina).

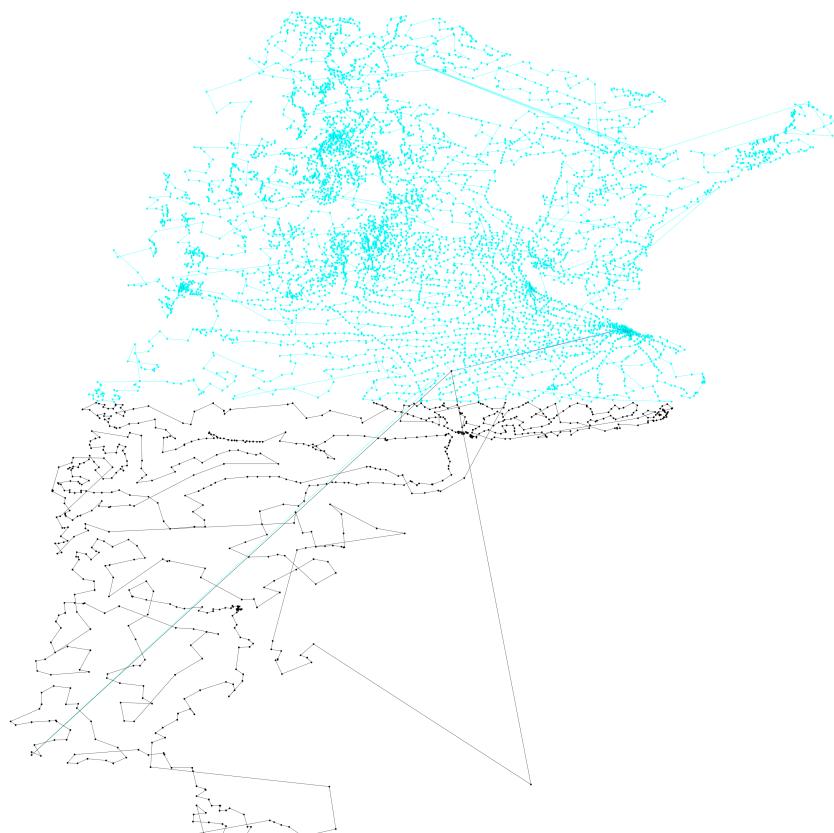


Figura 7.8: Resultado utilizando AG Híbrido com 4 caixeiros (Argentina).

8 Conclusão

Por meio desta pesquisa preliminar pode-se concluir que o Algoritmo Genético Híbrido desenvolvido é eficiente e economiza um tempo significativo de execução, mas ainda, não gera o resultado pretendido, pois suas rotas ainda são grandes comparadas com a base de teste *TSPLIB*.

O objetivo futuro é implementar diversos outros algoritmos para gerar a população inicial mais otimizada, já que o problema está neste, pois o indivíduo gerado pelo algoritmo *Nearest-Neighbor* é muito superior à qualquer outro indivíduo gerado de forma aleatória, limitando a evolução da população, já que qualquer cruzamento entre estes indivíduos tem uma probabilidade remota de gerar um filho melhor que o indivíduo gerado pelo *Nearest-Neighbor*.

Referências Bibliográficas

ARNBORG, S.; PROSKUROWSKI, A. Linear time algorithms for np-hard problems restricted to partial k-trees. *Elsevier Science Publishers*, Março 1989.

BELFIORE, P. P. *Scatter search para Problemas de Roteirização de Veículos com Frota Heterogênea, Janelas de Tempo e Entregas Fracionadas*. Tese (Doutorado) — Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Produção., 2006.

BENEVIDES, P. F. et al. Aplicação e análise de alguns procedimentos de construção de rota para o problema do caixeiro viajante. *Revista Ingeniería Industrial*, p. 17–25, 2012.

CARTER, A. E. *Design and Application of Genetic Algorithms for the Multiple Traveling Salesperson Assignment Problem*. Tese (Doutorado) — Virginia Polytechnic Institute and State University, 2003.

CARVALHO, A. P. de Leon F. de. *Algoritmos Genéticos*. 2013. Disponível em: <<http://www2.icmc.usp.br/~andre/research/genetic/>>.

CORREIA, M. Algoritmos genéticos. *dosalgarves*, p. 36–43, junho 2003.

MALAQUIAS, N. G. L. *Uso dos algoritmos genéticos para a otimização de rotas de distribuição*. Dissertação (Mestrado) — Universidade Federal de Uberlândia, 2006.

MIYAZAWA, F. K. *Otimização*. 2002. Disponível em: <<http://www.ic.unicamp.br/~fkm/lectures/otimo.pdf>>.

PACHECO, M. A.; FUKASAWA, R. Resolução do problema do entregador viajante. *Revista de Inteligência Computacional Aplicada*, n. 4, 2010.

RESEARCH, O. of N. *The Traveling Salesman Problem*. 2014. Disponível em: <<http://www.math.uwaterloo.ca/tsp/index.html>>.

SEDIGHPOUR, M.; YOUSEFIKHOSHBAKHT, M.; DARANI, N. M. An effective genetic algorithm for solving the multiple traveling salesman problem. *Journal of Optimization in Industrial Engineering*, p. 73–79, 2011.

WU, L. *O problema de roteirização periódica de veículos*. Tese (Doutorado) — Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Transportes., 2007.

APÊNDICE A – Código Fonte - main.cpp

```

/*
 * Author: Victor Carlquist
 * Date:24/04/2014
 * IFSP - Campos do Jordão
 * e-mail: victorcarlquist@gmail.com
 */

#include <iostream>
#include "ga.h"
#include <cstdlib>
#include <ctime>
#include <fstream>
#include <cstring>
#include <search.h>
#include <algorithm>

using namespace std;

vector<string> split(const string& s, const string& delim, const bool keep_empty = true) {
    vector<string> result;
    if (delim.empty()) {
        result.push_back(s);
        return result;
    }
    string::const_iterator substart = s.begin(), subend;
    while (true) {
        subend = search(substart, s.end(), delim.begin(), delim.end());
        string temp(substart, subend);
        if (keep_empty || !temp.empty()) {
            result.push_back(temp);
        }
        if (subend == s.end()) {
            break;
        }
        substart = subend + delim.size();
    }
    return result;
}

void loadCity(GA *world)
{
    string line;
    ifstream myfile ("../argentina.tsp");

    if (myfile.is_open())
    {
        vector<string> c = split(line, " ");
        while ( getline (myfile,line) )
        {
            c = split(line, " ");
            //cout<< c[0] << " " << c[1] << " " << c[2] << "\n";
            world->setCity(atof(c[1].c_str())+20000,atof(c[2].c_str())+20000,atoi(c[0].c_str()));
        }
        myfile.close();
        cout<< "Loaded file.\n";
    }
    else
}

```

```

        cout << "Unable to open file";
}

int main(void){
    srand(11111);

    clock_t begin, end;
    double time_spent;

    begin = clock();

    GA world;
    world.MaxPopulation      = 10;
    world.InitialPopulation  = 10;
    world.InitialPopulationWithNN = false; // utilizar o algortimo Nearest Neighbor
    world.MutationRouteItself = true; // realiza a mutação em cada rota de cada viajante, sem afetar a outra
    world.NNsizePart         = 1000; // Tamanho de cada regiao para a divisao do Nearest neighbor
    world.NNnPart             = 4; // Quantidade de regioes para a divisao do Nearest neighbor
    world.RateGeneration     = 0.8; // taxa de substituição dos individuos
    world.RateMutation       = 0.2; // taxa de mutação
    world.RateSalesmanMutation = 0; // taxa de mutação
    world.Generation          = 900;
    world.Deposit             = 0; // O deposito está na cidade 0(zero)
    world.Salesman            = 4;
    world.SaveBetterChromo   = false; //Evita que o melhor individuo sofra mutação

    unsigned int i,j,k,m;
    for(i=0;i<600;++i)
        world.setCity(rand()%world.NNsizePart,rand()%world.NNsizePart,i);
    //loadCity(&world);
/*
    world.setCity(5,5,0);
    world.setCity(1,5,1);
    world.setCity(2,6,2);
    world.setCity(3,2,3);
    world.setCity(4,3,4);
    world.setCity(6,7,5);
    world.setCity(2,5,6);
    world.setCity(2,3,7);
    world.setCity(3,1,8);
    world.setCity(4,4,9);
*/
    ofstream myfile;
    GA::Chromossome *a;

    myfile.open("gnuplot.conf");
    myfile << "set term pngcairo size 3500,3500 font \"Helvetica,10\"\n" \
              "set output 'output.png'\n" \
              "set size square\n" \
              "unset key; unset tics; unset border\n";
    if(world.Salesman == 1)
        myfile << "plot 'route0.txt' using 1:2 with linespoint, 'route0.txt' using 1:2:3 with labels font 'Helvetica,1' offset
                  0.7 notitle\n";
    else
        myfile << "plot 'route0.txt' using 1:2 with linespoint, 'route0.txt' using 1:2:3 with labels font 'Helvetica,1' offset
                  0.7 notitle, \\\n";
    for(j=1;j<world.Salesman;++j)
    {
        if(j != world.Salesman-1)
            myfile << "'route'"<< j << ".txt' using 1:2 with linespoint, 'route"<< j << ".txt' using 1:2:3 with labels font '
                  "Helvetica,1' offset 0.7 notitle, \\\n";
        else
            myfile << "'route'"<< j << ".txt' using 1:2 with linespoint, 'route"<< j << ".txt' using 1:2:3 with labels font '
                  "Helvetica,1' offset 0.7 notitle, \\\n";
    }
    //myfile << endl << "pause 1" << endl;
    //myfile << endl << "replot" << endl;
    //myfile << endl << "reread" << endl;
    myfile.close();

    world.evolution();
}

```

```

end = clock();

for(i = 0;i<world.Population.size();++i)
{
    for(j = 0;j < world.getRouteIndexMax();++j)
        cout << world.Population[i]->Gene[j].id << "-";
    cout << "|";
    for(j = 0;j < world.Population[i]->Salesman.size();++j)
        cout << world.Population[i]->Salesman[j] << "-";
    cout << "Dist: " << world.Population[i]->dist << endl;
}
a = world.Population.back();
m = 0;
for(j=0;j<a->Salesman.size();++j)
{
    string path = "route";
    path += std::to_string(j) + ".txt";
    myfile.open (path);
    myfile << "#Distancia total de todas as rotas: "<< a->dist << endl;
    myfile << world.cities[world.Deposit] << endl;
    for(k=0;k<a->Salesman[j];++k)
    {
        myfile << a->Gene[m] << endl;
        m++;
    }
    myfile << world.cities[world.Deposit] << endl;
    myfile.close();
}
system("gnuplot 'gnuplot.conf'");
system("xdg-open 'output.png'");

time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
cout << "Time: " << time_spent << endl;

return 0;
}

```

APÊNDICE B – Código Fonte - ga.h

```

/*
 * Author: Victor Carlquist
 * Date:24/04/2014
 * IFSP - Campos do Jordão
 */

#ifndef GA_H
#define GA_H

#include <iostream>
#include <vector>

#define INF 0xFFFFFFF

class GA
{
public:

    struct Point{
        int id;
        float x,y;
        friend std::ostream & operator<<(std::ostream &Str, Point const &v)
        {
            Str << v.x << " " << v.y << " " << v.id;
            return Str;
        }
    };
    struct Chromosome
    {
        std::vector<Point> Gene;
        std::vector<unsigned int> Salesman;
        double dist;
    };

    GA();
    ~GA() {};
    bool SaveBetterChromo;
    unsigned int MaxPopulation ; // = 1000;
    unsigned int InitialPopulation; // = 10;
    unsigned int NNsizePart;
    unsigned int NNNpart;
    bool InitialPopulationWithNN; // = 10;
    float RateGeneration; // = 0.02; // taxa de substituição dos individuos
    float RateMutation; // = 0.01; // taxa de mutação da rota
    bool MutationRouteItself;
    float RateSalesmanMutation; // = 0.02; // taxa de mutação do numero de objetivos por viajante
    unsigned int Generation; // = 1000; //
    unsigned int Deposit; // = 0; // O deposito está na cidade 0(zero)
    unsigned int Salesman; // = 2; // quantidade de viajantes
    void addChromo(Chromosome *a);
    virtual void fitness(Chromosome *a);
    virtual void crossover(Chromosome *f1, Chromosome *f2);
    virtual void mutation(Chromosome *chromo); // EX
    // muda o numero de objetivos que o viajante deve visitar
    virtual void salesmanMutation(Chromosome *chromo);
    static double euclideanDistance(Point *a, Point *b);
};

```

```

static unsigned int find(std::vector<Point> *vec, Point target);
std::vector<Point> NN(std::vector<Point>);

/*
 * Retorna a quantidade de cidades em uma rota
 */
unsigned int getRouteIndexMax()
{
    return (this->cities.size()-1);
}
void setCity(float x, float y, int id);

/* Cada individuo será representado utilizando o método two-part
 * (Para facilitar a implementação foi criado a classe Chromossome
 * para separa os tipos de variáveis. Nesta classe foi criado dois
 * vetores, um armazena a rota e o outro os viajantes)
 * | 1 | 2 | 3 | 4 | 5 / 2 | 3 |
 * Os dois últimos elementos representa o caixearo 1 e 2, sendo que
 * o caixearo 1 ira visitar os dois primeiros elementos do vetor,
 * e o segundo caixearo ira visitar os três últimos elementos do vetor.
 * A distância da rota será armazenada no ultimo indice do vetor:
 * | 1 | 2 | 3 | 4 | 5 / 2 | 3 || 44 |
 */
std::vector<Chromossome *> Population;
/*
 * Executa o algoritmo - nucleo (core)
 */
void evolution();
std::vector<Point> cities;

// Thread gnuplot
void saveGnuplot();
bool finish;
//  

private:  

/*
 * divide os objetivos em partes iguais para cada viajante
 * e gera a população inicial
 */
void prepare();
};

#endif // GA_H

```

APÊNDICE C – Código Fonte - ga.cpp

```

/*
 * Author: Victor Carlquist
 * Date:24/04/2014
 * IFSP - Campos do Jordão
 */

#include "ga.h"
#include <cmath>
#include <cstdlib>      /* srand, rand */
#include <ctime>
#include <algorithm>    // std::sort
#include <vector>
#include <iostream>
#include <fstream>
#include <cstring>
#include <thread>
#include <unistd.h>

//Ordem crescente
bool Csort (GA::Chromossome *i,GA::Chromossome *j) {
    return (i->dist < j->dist);
}

//Ordem decrescente
bool CsortM (GA::Chromossome *i,GA::Chromossome *j) {
    return (i->dist > j->dist);
}

// utilizado no shuffle das rotas iniciais
int myrandom (int i) { return std::rand()%i;}

GA::GA()
{
    std::srand(11111);
}

/*
 * retorna o indice do alvo no vetor
 */
unsigned int GA::find(std::vector<Point> *vec, Point target)
{
    unsigned int i;

    for(i=0;i<vec->size();++i)
    {
        if((*vec)[i].id == target.id)
            return i;
    }
    return vec->size();
}

void GA::addChromo(Chromossome *a)
{
    this->fitness(a);
    if(this->RateGeneration < ((float)rand() / (RAND_MAX)) && this->Population.size() < this->MaxPopulation)
    {
        this->Population.push_back(a);
    }
    else
    {
        if(this->Population.size() == 0)
            this->Population.push_back(a);
    }
}

```

```

        else{
            bool s = false;
            if(this->Population.back()->dist < a->dist)
                s = true;
            delete this->Population.back();
            this->Population.back() = a;
            if(!s)
                std::sort(this->Population.begin(),this->Population.end(),Csort);
        }
    }
}

void GA::setCity(float x, float y, int id)
{
    GA::Point p;
    p.id = id;
    p.x = x;
    p.y = y;
    this->cities.push_back(p);
}

double GA::euclideanDistance(Point *a, Point *b)
{
    float da = (a->x-b->x)*(a->x-b->x);
    float db = (a->y-b->y)*(a->y-b->y);

    return sqrt((double)(da+db));
}

/* fitness
 * Ira avaliar o individuo somando as distancias euclidianas
 * Menor distancia, melhor é o individuo
 */
void GA::fitness(Chromossome *a)
{
    unsigned int i = 0,j,k;
    double dist = 0;

    for(j=0;j<a->Salesman.size();++j)
    {
        dist += GA::euclideanDistance(&this->cities[this->Deposit],&a->Gene[i]);
        for(k=0;k<a->Salesman[j]-1;++k)
        {
            dist += GA::euclideanDistance(&a->Gene[i],&a->Gene[i+1]);
            i++;
        }
        dist += GA::euclideanDistance(&this->cities[this->Deposit],&a->Gene[i]);
        i++;
    }

    a->dist = dist;
}
/*
 * Crossover
 * O cruzamento usara o metodo PMX
 */
void GA::crossover(Chromossome *f1, Chromossome *f2)
{
    GA::Chromossome *s1, *s2;
    s1 = new GA::Chromossome;
    s2 = new GA::Chromossome;
    s1->Gene.reserve(this->getRouteIndexMax());
    s2->Gene.reserve(this->getRouteIndexMax());
    // armaze a quantidade de indices reservados para a rota,
    // usando apenas a primeira parte do cromossomo,
    // mas descontando 3 indices para o PMX funcionar
    unsigned int x = rand() % (this->getRouteIndexMax() - 2);

    int ind;
    unsigned int i;
    Point foo;
    foo.x = -1;
    foo.y = -1;
    foo.id = -1;
}

```

```

    for(i = 0; i<this->getRouteIndexMax(); ++i)
        s1->Gene.push_back(foo);
    for(i = 0; i<this->getRouteIndexMax(); ++i)
        s2->Gene.push_back(foo);

    for(i = x;i<(x+3);++i)
    {
        s1->Gene[i] = f2->Gene[i];
        s2->Gene[i] = f1->Gene[i];
    }

    //Evita que as cidades sejam repetidas dentro do cromossoma
    for(i = 0; i<this->getRouteIndexMax(); ++i)
    {
        if(s1->Gene[i].id != -1)
            continue;
        if(GA::find(&s1->Gene, f1->Gene[i]) == s1->Gene.size() )
        {
            s1->Gene[i] = f1->Gene[i];
        }else{
            ind = GA::find(&s1->Gene, f1->Gene[i]);
            while(GA::find(&s1->Gene, f1->Gene[ind]) != s1->Gene.size())
            {
                ind = GA::find(&s1->Gene, f1->Gene[ind]);
            }
            s1->Gene[i] = f1->Gene[ind];
        }
    }

    for(i = 0; i<f1->Salesman.size(); ++i)
        s1->Salesman.push_back(f1->Salesman[i]);
    for(i = 0; i<this->getRouteIndexMax(); ++i)
    {
        if(s2->Gene[i].id != -1)
            continue;
        if(GA::find(&s2->Gene, f2->Gene[i]) == s2->Gene.size() )
        {
            s2->Gene[i] = f2->Gene[i];
        }else{
            ind = GA::find(&s2->Gene, f2->Gene[i]);
            while(GA::find(&s2->Gene, f2->Gene[ind]) != s2->Gene.size())
            {
                ind = GA::find(&s2->Gene, f2->Gene[ind]);
            }
            s2->Gene[i] = f2->Gene[ind];
        }
    }

    for(i = 0; i<f2->Salesman.size(); ++i)
        s2->Salesman.push_back(f2->Salesman[i]);
    this->addChromo(s1);
    this->addChromo(s2);
}

/*
 * O operador de mutação é o EX
 * Troca dois elementos de lugar
 */
void GA::mutation(Chromosome *chromo)
{
    unsigned int a,b,offset,i;

    if(!this->MutationRouteItself)
    {
        a = rand() % this->getRouteIndexMax();
        do{
            b = rand() % this->getRouteIndexMax();
        }while(a == b);

        GA::Point x = chromo->Gene[a];
        chromo->Gene[a] = chromo->Gene[b];
        chromo->Gene[b] = x;

    }else{
        offset = 0;
    }
}

```

```

/*for(i=0;i<this->Salesman;++i)
{
    a = offset + rand() % chromo->Salesman[i];
    b = offset + rand() % chromo->Salesman[i];

    GA::Point x = chromo->Gene[a];
    chromo->Gene[a] = chromo->Gene[b];
    chromo->Gene[b] = x;

    offset+=chromo->Salesman[i];
}*/
i = rand() % this->Salesman;
a = offset + rand() % chromo->Salesman[i];
b = offset + rand() % chromo->Salesman[i];

GA::Point x = chromo->Gene[a];
chromo->Gene[a] = chromo->Gene[b];
chromo->Gene[b] = x;

offset+=chromo->Salesman[i];
}
this->fitness(chromo);
}

void GA::salesmanMutation(Chromossome *chromo)
{
    unsigned int a,b,c;

    std::vector<unsigned int> newSalesmans;
    //0 '-1' é o desconto da cidade deposito
    c = this->cities.size()-1-this->Salesman;
    for(b=0;b<this->Salesman-1;++b)
    {
        a = rand()%c+1;
        newSalesmans.push_back(a);
        c -= a-1;
    }
    c = 0;
    for(b=0;b<newSalesmans.size();++b)
        c+=newSalesmans[b];
    newSalesmans.push_back(this->cities.size()-1-c);
    chromo->Salesman = newSalesmans;

    this->fitness(chromo);
}

/*
 * É passado o vetor que contem os pontos de uma região
 */
std::vector<GA::Point> GA::NN(std::vector<GA::Point> citiesReg)
{
    unsigned int i,j,indClosest;
    double dist,closestDist;

    std::vector<Point> NN;

    j = i = 0;
    NN.push_back(citiesReg[0]);
    citiesReg.erase(citiesReg.begin());
    while(citiesReg.size()>0)
    {
        j =0;
        closestDist = 99999999;
        while(j<citiesReg.size())
        {
            if(NN.back().id != citiesReg.at(j).id)
            {
                dist = this->euclideanDistance(&NN.back(),&citiesReg.at(j));
                if(closestDist > dist)
                {
                    closestDist = dist;
                    indClosest = j;
                }
            }
        }
    }
}

```

```

        j++;
    }
    NN.push_back(citiesReg[indClosest]);
    citiesReg.erase(citiesReg.begin() + indClosest);
}

return NN;
}

/*
 *Gera a populaçao inicial
 */
void GA::prepare()
{
    unsigned int n = this->getRouteIndexMax() / this->Salesman;

    GA::Chromossome *a;
    unsigned int i,j,k,max;

    if(this->InitialPopulationWithNN)
        max = this->InitialPopulation - 1;
    else
        max = this->InitialPopulation;

    for(k=0;k<max;++k)
    {
        a = new GA::Chromossome();
        for(i=0;i<cities.size();++i)
        {
            if(i != this->Deposit)
            {
                a->Gene.reserve(this->getRouteIndexMax());
                a->Gene.push_back(cities[i]);
            }
        }
        std::random_shuffle(a->Gene.begin(), a->Gene.end(), myrandom );
        for(j=0;j<this->Salesman-1;++j)
            a->Salesman.push_back(n);
        a->Salesman.push_back(n+this->getRouteIndexMax()%this->Salesman);

        this->addChromo(a);
    }
    // Nearest Neighbor
    /*
     * Para deixar o algoritmo NN eficiente,o código abaixo divide o espaço
     * em regioes, comparando os vizinhos que pertencem a mesma região do plano cartesiano
     */
    // nPart, numero de regiões
    // nPart, o valor armazenado sempre será dividido por 2 e deve ser par
    // Ex: nPart = 4, divididos em 2 linhas e 2 colunas
    // Ex: nPart = 8, divididos em 2 linhas e 4 colunas
    if(this->InitialPopulationWithNN)
    {
        unsigned int nPart = this->NNnPart;
        unsigned int retSizeY = (this->NNsizePart+2)/2;
        unsigned int retSizeX = (this->NNsizePart+2)/(nPart/2);
        unsigned int x, y;
        std::vector<std::vector<Point>> PartNN;

        for(i=0;i<nPart;++i)
        {
            std::vector<Point> lineReg;
            PartNN.push_back(lineReg);
        }
        std::vector<Point> lineReg;
        for(i=0;i<this->cities.size();++i)
        {
            x = y = 0;
            if(i == this->Deposit)
                continue;
            for(j=0;j<nPart;++j)
            {

```

```

if(this->cities.at(i).x >= x && this->cities.at(i).x < x+retSizeX && this->cities.at(i).y >= y && this->cities
    .at(i).y < y+retSizeY)
{
    PartNN.at(j).push_back(this->cities.at(i));
}
if(j < (nPart/2)-1 || j > (nPart/2)-1)
{
    x+=retSizeX;
}else{
    if(j == (nPart/2)-1)
    {
        x = 0;
        y+=retSizeY;
    }
}
}

}

a= new GA::Chromossome();

for(i=0;i<nPart;++i)
{
    if(!PartNN[i].empty())
    {
        PartNN[i] = this->NN(PartNN[i]);
        a->Gene.insert(a->Gene.end(),PartNN[i].begin(),PartNN[i].end());
    }
}

for(j=0;j<this->Salesman;++j)
    a->Salesman.push_back(0);

i = nPart/this->Salesman;
k = 0;
if(i != 0)
    for(j=0;j<nPart;++j)
    {
        if(!PartNN[j].empty())
            a->Salesman[k]+=PartNN[j].size();
        if(((j+1)%i) == 0)
            k++;
    }
j = 0;
i = 0;
max = 0;
// se i != 0 entao eh preciso desontar uma rota de um caixeiro pois
// foi preciso dar um objetivo para um caixeiro que não tinha nenhum
for(k=0;k<this->Salesman;++k)
    if(a->Salesman[k] == 0)
        i++;
    else
        max+=a->Salesman[k];

max -= i;
max = this->cities.size()-1-max;
while(max != 0 || i != 0)
{
    if(i != 0)
    {
        if(a->Salesman[j] > 1)
        {
            a->Salesman[j]--;
            i--;
        }
    }
    if(max != 0 && a->Salesman[j] < (this->cities.size()-1-this->Salesman))
    {
        a->Salesman[j]++;
        max--;
    }
    j++;
}
// se i != 0 entao eh preciso desontar uma rota de um caixeiro pois
// foi preciso dar um objetivo para um caixeiro que não tinha nenhum

```

```

        j = 0;
    for(k=0;k<this->Salesman;++k)
        if(a->Salesman[k] == 0 && max == 0)
            max++;

    }
    //for(j=0;j<a->Gene.size();++j)
    //std::cout << a->Gene[j].id << " ";

    this->addChromo(a);
}
// Find Nearest Neighbor

}
/*
 * Executa o nucleo do AG
 */
void GA::evolution()
{
    this->prepare();

    this->finish = false;
    //std::thread t1(&GA::saveGnuplot,this);
    clock_t begin = clock();
    clock_t end = clock();
    double elapsed_secs;
    unsigned int i = 0;
    for(i=0;i<this->Generation;++i)
    {
        end = clock();
        elapsed_secs = double(end - begin) / CLOCKS_PER_SEC;
        if(elapsed_secs>3 || i == 0)
        {
            begin = clock();
            //this->saveGnuplot();
            std::cout << "Generation: " << i << std::endl;
        }

        float mr = ((float)rand() / (RAND_MAX));
        unsigned int a,b;

        if(mr < this->RateMutation)
        {
            b = rand()%this->Population.size();
            if(this->SaveBetterChromo)
                if(b == 0)
                    b = 1;
            this->mutation(this->Population[b]);
        }
        mr = ((float)rand() / (RAND_MAX));

        if(mr < this->RateSalesmanMutation)
        {
            b = rand()%this->Population.size();
            if(this->SaveBetterChromo)
                if(b == 0)
                    b = 1;
            this->salesmanMutation(this->Population[b]);
        }

        a = rand() % this->Population.size();
        //do{
        b = rand() % this->Population.size();
        //}while(a == b);
        this->crossover(this->Population.at(a),this->Population.at(b));

    }
    this->finish = true;
    //t1.join();
    std::sort(this->Population.begin(),this->Population.end(),CsortM);
}

```

```
void GA::saveGnuplot()
{
    std::ofstream myfile;
    GA::Chromossome *a;
    unsigned int m,j,k;
    system("xdg-open 'output.png'");
    // while(!this->finish && this->Population.size()>1)
    //{
    //    a = this->Population.front();
    //    m = 0;
    //    for(j=0;j<a->Salesman.size();++j)
    //    {
    //        std::string path = "route";
    //        path += std::to_string(j) + ".txt";
    //        myfile.open (path,std::fstream::out);
    //        if(myfile.is_open())
    //        {
    //            myfile << "#Distancia total de todas as rotas: "<< a->dist << std::endl;
    //            myfile << this->cities[this->Deposit] << std::endl;
    //            for(k=0;k<a->Salesman[j];++k)
    //            {
    //                myfile << a->Gene[m] << std::endl;
    //                m++;
    //            }
    //            myfile << this->cities[this->Deposit] << std::endl;
    //            myfile.close();
    //            //myfile.clear();
    //        }
    //    }
    //    system("gnuplot 'gnuplot.conf' &");
    //    //sleep(2);
    //}
}
```