

**INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE
SÃO PAULO**

VICTOR HUGO CARLQUIST DA SILVA

**DESENVOLVIMENTO DE UM ALGORITMO GENÉTICO HÍBRIDO
PARA A SOLUÇÃO DO PROBLEMA DOS MÚLTIPLOS CAIXEIROS
VIAJANTES (*MTSP*)**

CAMPOS DO JORDÃO

2015

VICTOR HUGO CARLQUIST DA SILVA

**DESENVOLVIMENTO DE UM ALGORITMO GENÉTICO HÍBRIDO
PARA A SOLUÇÃO DO PROBLEMA DOS MÚLTIPLOS CAIXEIROS
VIAJANTES (*MTSP*)**

Trabalho de Conclusão de Curso apresentado ao Instituto Federal de Educação, Ciência e Tecnologia - IFSP - *Campus Campos do Jordão*, como parte das exigências para obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas.

Orientador: Prof. Dr. Helton Hugo de Carvalho Júnior

Co-orientador: Prof. Me. André Malvezzi Lopes
Co-orientador: Prof. Dr. Silvio Alexandre de Araujo

CAMPOS DO JORDÃO
2015

Si586i Silva, Victor Hugo Carlquist da
Desenvolvimento de um algoritmo genético híbrido para a solução do
problema dos múltiplos caixeiros viajantes (*mTSP*) / Victor Hugo Carlquist
da Silva. – Campos do Jordão, 2015.
53 p. : il.

Orientador: Prof. Dr. Helton Hugo de Carvalho Júnior

Trabalho de Conclusão de Curso (Graduação em Tecnologia em
Análise e Desenvolvimento de Sistemas) – Instituto Federal de Educação,
Ciência e Tecnologia de São Paulo.

1. Problema dos múltiplos caixeiros viajantes. 2. Algoritmo genético.
3. Algoritmo híbrido. 4. Otimização. I. Carvalho Junior, Helton Hugo de,
orientador. II. Título

CDD: 511.6

VICTOR HUGO CARLQUIST DA SILVA

**DESENVOLVIMENTO DE UM ALGORITMO GENÉTICO HÍBRIDO
PARA A SOLUÇÃO DO PROBLEMA DOS MÚLTIPLOS CAIXEIROS
VIAJANTES (MTSP)**

Trabalho de Conclusão de Curso apresentado ao Instituto Federal de Educação, Ciência e Tecnologia - IFSP - *Campus Campos do Jordão*, como parte das exigências para obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas.

BANCA EXAMINADORA
7 de janeiro de 2015

**Orientador: Prof. Dr. Helton Hugo de
Carvalho Júnior**

**Co-orientador: Prof. Me. André Malvezzi
Lopes**

Prof. Esp. Paulo Giovani de Faria Zeferino

LOCAL

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

AGRADECIMENTOS

A Deus por ter me concedido a vida e por ter colocado em meu caminho pessoas incríveis.

Aos meus pais e aos meus dois irmãos que sempre me deram apoio e incentivo por todos esses anos.

Agradeço ao meu orientador Prof. Dr. Helton Hugo de Carvalho Júnior e aos meus co-orientadores Prof. Dr. Silvio Alexandre de Araujo e, em especial, o co-orientador Prof. Me. André Malvezzi Lopes, por ter compartilhado e dedicado seus conhecimentos, paciência e tempo a mim.

Agradeço a todos os meus amigos que sempre estiveram dispostos a ajudar.

Também agradeço aos professores desta instituição, que, sem dúvida, proveram uma base sólida de conhecimento, no qual levarei por toda a minha vida.

Também agradeço a todos os funcionários, que contribuíram, de forma direta ou indireta, para a minha formação.

RESUMO

Neste trabalho é abordado o desenvolvimento de um algoritmo que consiga gerar uma solução para o Problema dos Múltiplos Caixeiros Viajantes. Foi desenvolvido um algoritmo híbrido, utilizando o Algoritmo Genético com o algoritmo *Nearest Neighbor*, que foi utilizado para gerar a população inicial, obtendo um bom desempenho no tempo de execução e gerando uma solução para o Problema dos Múltiplos Caixeiros Viajantes. Todos os testes para validar o algoritmo foram comparados com os resultados da base de testes TSPLIB.

Palavras-chave: Problema dos múltiplos caixeiros viajantes, algoritmo genético, algoritmo híbrido, otimização.

ABSTRACT

This work shows the development of the algorithm that is able to generate a efficient solution for *Multiple Traveling Salesman Problem* (mTSP). An algorithm was developed a hybrid algorithm that use the Genetic Algorithm with *Nearest Neighbor* algorithm, this last algorithm cited, generates the initial population for Genetic Algorithm. This hybrid algorithm gets a great performance at execution time and it can generate a good solution to solve *Multiple Traveling Salesman Problem*. To valid the algorithm, the tests were compared against TSPLIB's results.

Key-words: Multiple traveling salesman problem, genetic algorithm, hybrid algorithm, optimization.

LISTA DE ILUSTRAÇÕES

Figura 1 – PMX - cruzamento	15
Figura 2 – PMX - preenchimento	16
Figura 3 – Evolução da rota utilizando AG com 2000 pontos.	19
Figura 4 – Evolução da rota com o algoritmo híbrido (GANN).	20
Figura 5 – Cromossomo representado pelo método <i>two-part</i>	22
Figura 6 – Cromossomo - <i>two-part</i>	22
Figura 7 – TSPLIB - Imagem gerada pelo algoritmo LKH, Argentina.	25
Figura 8 – Teste utilizando o Algoritmo Genético - Argentina.	27
Figura 9 – Teste utilizando o Algoritmo <i>Nearest-Neighbor</i> - Argentina.	27
Figura 10 – Teste utilizando o Algoritmo Genético - China.	28
Figura 11 – Teste utilizando o GANN - China.	28
Figura 12 – Resultado do teste PR2392 gerado por GANN.	29
Figura 13 – Evolução da rota com diversos caixeiros viajantes (GANN).	30
Figura 14 – Evolução da rota com diversos viajantes (GA).	31
Figura 15 – Desempenho do algoritmo utilizando <i>threads</i>	32

SUMÁRIO

1	INTRODUÇÃO	10
2	METODOLOGIA	11
3	PROBLEMA DO CAIXEIRO VIAJANTE - TSP	12
3.1	Problema dos Múltiplos Caixeiros Viajantes	12
4	ALGORITMOS EVOLUTIVOS	14
4.1	Algoritmo Genético	14
4.1.1	Operadores de cruzamento	15
4.1.1.1	Cruzamento de Mapeamento Parcial - PMX	15
4.1.2	Operador de mutação	16
4.1.3	Algoritmo Genético Híbrido	16
4.1.4	Algoritmo <i>Nearest-Neighbor</i> (NN)	17
5	ESTADO DA ARTE	18
6	DESENVOLVIMENTO	19
6.1	Parâmetros	20
6.2	<i>Nearest-Neighbor</i> modificado	21
6.3	Estrutura do cromossomo (indivíduo) - <i>two-part</i>	22
6.4	Avaliação do Indivíduo (<i>fitness</i>)	23
6.5	Cruzamento	23
6.6	Mutação	24
6.7	Desempenho	24
6.7.1	Requisitos de Sistema	24
6.7.2	Resultados do Problema do Caixeiro Viajante (TSP)	25
6.7.2.1	GANN <i>versus</i> LKH	29
6.7.3	Resultados do Problema dos Múltiplos Caixeiros Viajantes (mTSP)	30
6.7.3.1	Múltiplos Caixeiros - GANN	30
6.7.3.2	Múltiplos Caixeiros - GA	31
6.7.4	Threads	31
6.7.4.1	Threads (GANN)	32
7	CONCLUSÃO E TRABALHOS FUTUROS	33
	Referências	34

APÊNDICE A – CÓDIGO FONTE - MAIN.CPP	35
APÊNDICE B – CÓDIGO FONTE - GA.H	41
APÊNDICE C – CÓDIGO FONTE - GA.CPP	43

1 INTRODUÇÃO

A importância do transporte veicular nos dias de hoje causa impactos positivos e negativos no meio ambiente e na sociedade, principalmente na economia mundial (DIAS; KUWAHARA, 2009).

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

O crescimento das cidades e da complexidade rodoviária dificulta 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* que sejam mais rápidos e precisos em resolver o problema de roteirização de veículos, encontrando o melhor caminho para os veículos percorrerem, realizando suas entregas nestes pontos já preestabelecidos.

2 METODOLOGIA

O objetivo deste trabalho é desenvolver um novo algoritmo, ou otimizar um algoritmo, para encontrar a rota ótima¹ ou eficiente, para o problema dos múltiplos caixeiros viajantes, consumindo menos recurso computacional, mais especificamente o tempo de execução.

Antes de se iniciar o desenvolvimento do algoritmo híbrido, foi realizado o levantamento bibliográfico sobre o assunto e debatido como otimizá-lo com o orientador do projeto.

Foi utilizada a base de testes TSPLIB² desenvolvida para o Problema do Caixeiro Viajante (TSP), já que não foi possível encontrar uma base de testes específica para o Problema dos Múltiplos Caixeiros Viajantes (mTSP). Esta base de testes foi utilizada como referência para medir o desempenho das rotas e o tempo de execução do algoritmo híbrido desenvolvido³ (RESEARCH, 2014).

Para exemplificar, foi escolhido um teste da base TSPLIB que já possui resultados de outros algoritmos, como a distância encontrada e o tempo que o algoritmo levou para calcular a rota. O teste foi submetido ao algoritmo híbrido e os resultados que ele gerou foram comparados com os resultados da base TSPLIB.

Após esses testes foram utilizadas diversas configurações de cenário, modificando os parâmetros do Algoritmo Genético, que serão explicados neste trabalho, comparando o tamanho das rotas e o tempo de execução do Algoritmo Genético “tradicional” com o algoritmo híbrido para verificar a eficiência e eficácia do algoritmo desenvolvido.

¹ Quando a rota gerada por um algoritmo é o menor caminho possível, essa rota é considerada uma “rota ótima”.

² Disponível em <<http://www.math.uwaterloo.ca/tsp/world/countries.html>>.

³ O TSPLIB se baseia em informações geográficas do *National Imagery and Mapping Agency*.

3 PROBLEMA DO CAIXEIRO VIAJANTE - TSP

O Problema do Caixeiro Viajante (*Traveling Salesman Problem* - TSP) consiste em estabelecer uma rota para um **único** caixeiro, passando por cada vértice do grafo apenas uma vez e retornando ao vértice de partida. O número de rotas possíveis pode ser expresso por $(n - 1)!$, sendo n o número de pontos (vértices). O problema TSP é classificado como *NP-Hard* (ARNBORG; PROSKUROWSKI, 1989), 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

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

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

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

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

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, usada 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)

3.1 Problema dos Múltiplos Caixeiros Viajantes

O problema dos Múltiplos Caixeiros Viajantes (*Multiple Traveling Salesman Problem* - *mTSP*) é uma extensão do TSP citado na sessão anterior. Neste problema há mais de um caixeiro que precisa visitar um conjunto de vértices, sendo que não é permitido um caixeiro visitar um vértice que outro caixeiro já visitou, estabelecendo várias rotas, uma para cada caixeiro viajante.

Sendo assim, o Problema dos Múltiplos Caixeiros Viajantes torna-se mais complexo conforme o número de caixeiros aumenta, porque é necessário distribuir os vértices (cidades) da melhor forma possível para cada caixeiro, tentando reduzir o tamanho das rotas (CARTER, 2003).

O algoritmo híbrido desenvolvido leva em consideração o tamanho da rota global, mas não leva em consideração o tamanho da rota de cada caixeiro, ou seja, o algoritmo híbrido não tenta igualar o tamanho das rotas entre os caixeiros.

4 ALGORITMOS EVOLUTIVOS

Os Algoritmos Evolutivos (AE) tentam reproduzir a evolução natural para atingir uma solução ótima. Uma população de soluções é utilizada e modificada a cada iteração, assim o resultado final também é uma população de soluções. Se um problema de otimização tem apenas uma solução ótima, então tenta-se buscar que a população converja para este único indivíduo ótimo. Caso um problema de otimização possua múltiplas soluções eficientes, como no caso dos Algoritmos Evolutivos Multiobjetivo, os Algoritmos Evolutivos podem ser usados para encontrar a população final com diversas soluções eficientes (LOPES, 2009).

Neste trabalho tenta-se encontrar a melhor solução, ou seja, a que possui a menor rota possível para resolver o Problema dos Múltiplos Caixeiros Viajantes.

4.1 Algoritmo Genético

Segundo Correia (2003), os Algoritmos Genéticos (AGs) são técnicas de procura e otimização baseadas 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* (HOLLAND, 1975). Hoje, este livro é considerado um dos mais importantes sobre Algoritmos Genéticos (CARVALHO, 2013).

No Algoritmo Genético, o cromossomo, também chamado de indivíduo, é representado por um conjunto de genes que armazena uma possível solução de um problema. Cada gene possui um valor que representa um vértice do grafo, sendo assim, os indivíduos são cruzados gerando novos indivíduos com sequência de genes (vértices) diferentes. Conforme a população cresce, surgem indivíduos cada vez mais aptos, ou seja, armazenam em seus cromossomos uma solução para o problema cada vez melhor, sendo que um deles será o mais apto, contendo no seu cromossomo a solução do problema, portanto, este indivíduo terá a melhor sequência de vértices entre os indivíduos da população, pelo qual o caixeiro deverá passar.

O Algoritmo Genético possui alguns parâmetros importantes para configurar a evolução dos indivíduos, por exemplo, o tamanho da população, que indica quantos indivíduos existirão para realizar o cruzamento. Se o número da população for pequeno deixará as rotas ruins (muito grandes), pois terá um pequeno conjunto para a busca da solução do problema. Já uma população muito grande pode afetar o desempenho do algoritmo. Também existe 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. É preciso salientar que estes parâmetros são pertinentes ao algoritmo e não ao modelo matemático.

4.1.1 Operadores de cruzamento

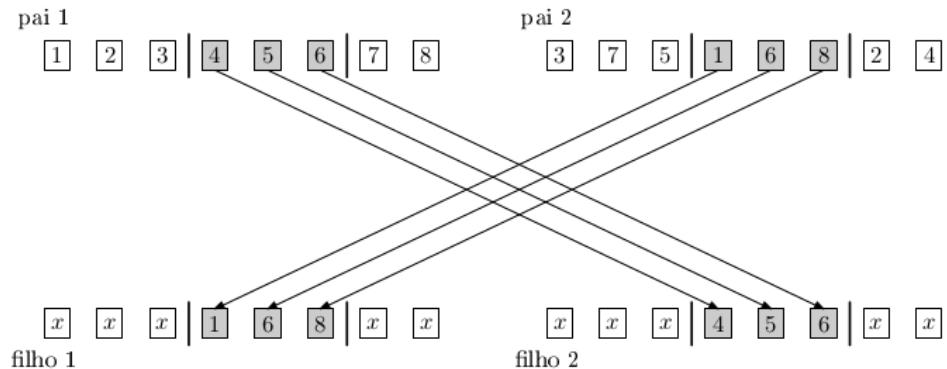
Os operadores de cruzamento definem como ocorrerá o cruzamento entre dois indivíduos, ou seja, como será feita a troca de uma sequência de genes entre dois indivíduos, gerando novos indivíduos na população (MALAQUIAS, 2006).

No algoritmo híbrido foi utilizado o Cruzamento de Mapeamento Parcial.

4.1.1.1 Cruzamento de Mapeamento Parcial - PMX

O operador de Cruzamento de Mapeamento Parcial (*Partially-mapped crossover* - PMX) escolhe dois indivíduos aleatoriamente e copia três genes seguidos, que são selecionados aleatoriamente, do pai para o filho, e completa o restante do cromossomo com os genes do outro pai, como a **figura 1** ilustra.

Figura 1 – PMX - cruzamento

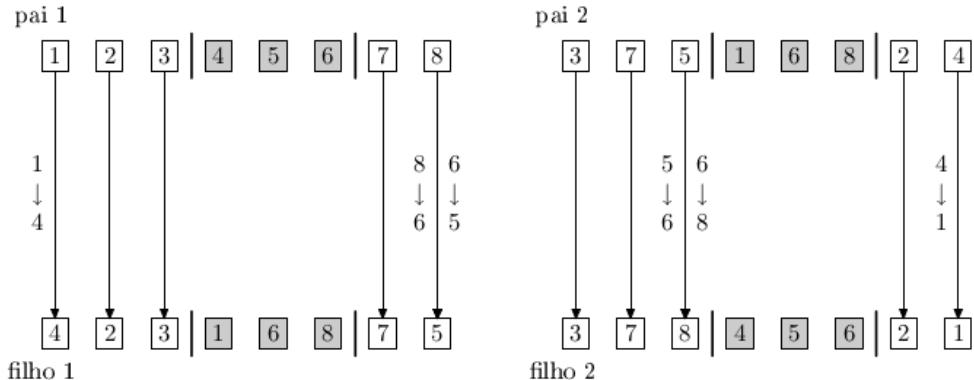


Fonte:(MALAQUIAS, 2006).

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

Por exemplo, como o “filho 1” já possui o valor 1, então é pesquisada qual a posição que o valor 1 está neste vetor, no caso da **figura 2** o valor 1 está na posição 4, então o valor na posição 4 do vetor do “pai 1” é copiado para a posição 1 do vetor “filho 1”. No outro exemplo na mesma figura, na terceira posição do vetor do “filho 2”, o número 5 já existe na posição 5, sendo assim, o valor que está na posição 5 no vetor “pai 2” é o valor 6, mas este valor também existe do vetor “filho 2”,

Figura 2 – PMX - preenchimento



Fonte:(MALAQUIAS, 2006).

portanto é pesquisada a posição do valor “filho 2”, que neste caso é a posição 6, então copia-se o valor 8 que está na posição 6 do vetor “pai 2” para a terceira posição do vetor “filho 2”. Este processo descrito acima é repetido para todos os outros elementos vazios¹ dos filhos.

4.1.2 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.

4.1.3 Algoritmo Genético Híbrido

Os Algoritmos Genéticos possuem o objetivo de serem robustos, ou seja, são eficientes nas soluções de problemas com complexidade *NP-HARD*, mas, estes algoritmos têm dificuldade em encontrar o caminho ótimo. Para solucionar esse problema foram criados os Algoritmos Genéticos Híbridos.

Os Algoritmos Genéticos Híbridos consistem em utilizar um outro algoritmo em conjunto com o Algoritmo Genético, produzindo algoritmos eficientes na prática (DAVENDRA, 2010).

¹ Os valores vazios são representados pelos caracteres “x” na figura 1.

4.1.4 Algoritmo *Nearest-Neighbor* (NN)

O Algoritmo *Nearest-Neighbor* (NN) é classificado como um algoritmo guloso².

O NN é um algoritmo simples de implementação, sua única tarefa é selecionar um vértice e verificar qual vértice ao seu redor está mais próximo, e colocá-lo como próximo vértice a ser visitado. Após isso, esse passo é repetido para o vértice escolhido.

O algoritmo NN possui uma complexidade $O(\frac{n^2-n}{2})$, pois faz a comparação com todos os vértices ainda não visitados (MIYAZAWA, 2002).

Este algoritmo é utilizado na geração da população inicial, sendo que a quantidade de indivíduos gerados por este algoritmo é definido por um parâmetro chamado **AmountPopulationWithNN**. O valor deste parâmetro deve ser menor ou igual ao valor do parâmetro **InitialPopulation**, caso o valor seja menor, o restante dos indivíduos da população inicial será gerado de forma aleatória, se o valor for igual, toda a população inicial será gerada pelo algoritmo *Nearest-Neighbor*.

² Um algoritmo guloso preza pela menor rota localmente, sem se preocupar com o desempenho da rota globalmente.

5 ESTADO DA ARTE

Existem diversos trabalhos sobre a utilização de Algoritmos Genéticos na resolução do problema do TSP. 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, Yousefikhoshbakht e Darani (2011) para a resolução do *mTSP* com um depósito, foi criado um único cromossomo utilizando o método *two-part*, que será explicado no próximo capítulo. Este método mostrou-se muito eficiente.

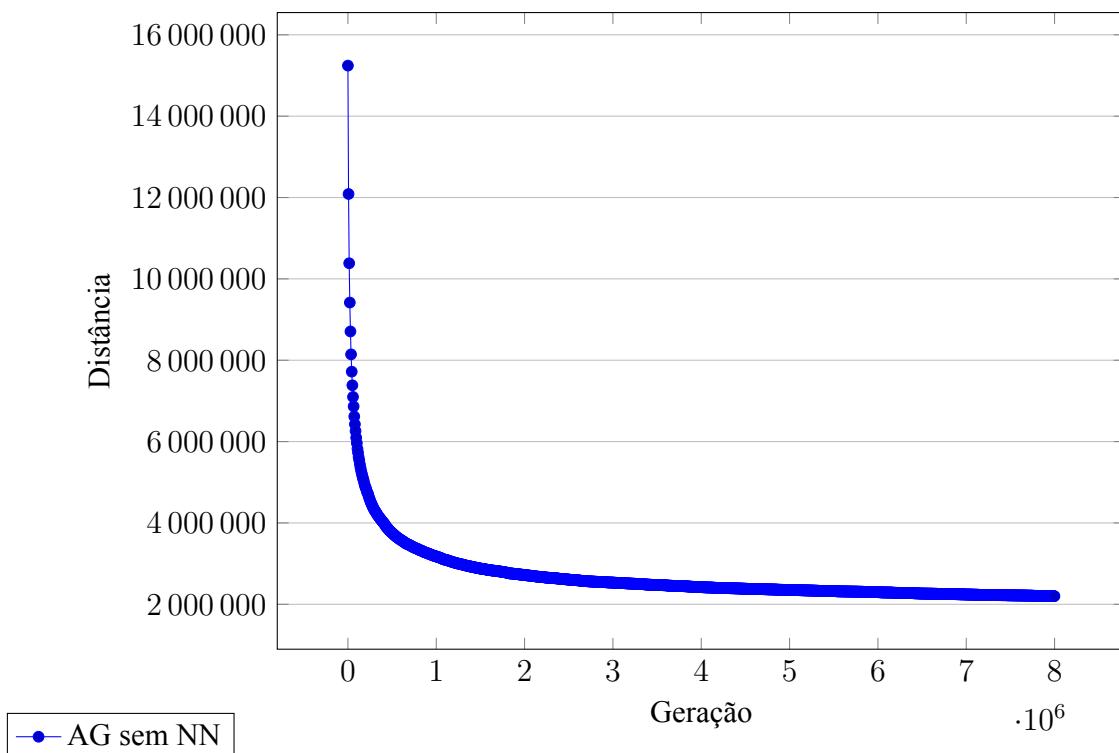
Wu (2007) mostra que é possível calcular as rotas de múltiplos veículos utilizando AG para igualar o tempo de espera de encomendas de clientes, sendo que a variável “menor tempo da rota” não é levada em consideração.

6 DESENVOLVIMENTO

O Algoritmo Genético Híbrido (GANN - *Genetic Algorithm with Nearest Neighbor*) proposto utiliza os conceitos do Algoritmo Genético tradicional em conjunto com o algoritmo *Nearest-Neighbor* (NN).

O desenvolvimento do Algoritmo Híbrido foi necessário pois, como ilustra a **figura 3**, a população do Algoritmo Genético está convergindo para um indivíduo que não é eficiente o suficiente, comparando com os resultados da base de teste TSPLIB.

Figura 3 – Evolução da rota utilizando AG com 2000 pontos.



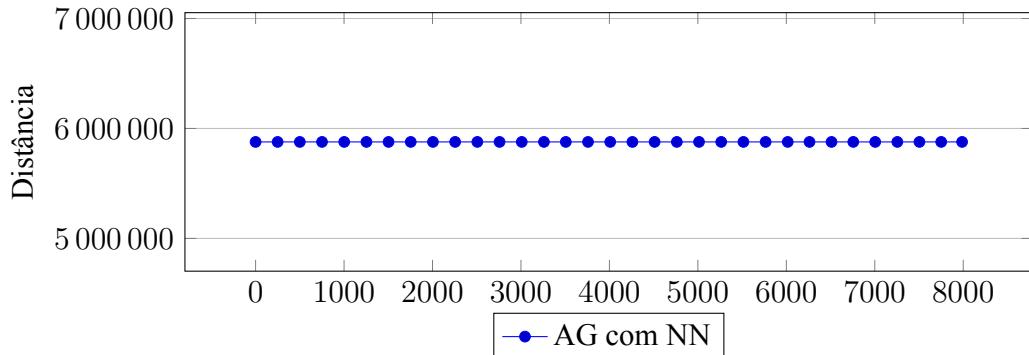
Fonte: Do autor.

Portanto, com a utilização do algoritmo *Nearest-Neighbor*, pode-se criar uma boa população inicial, contudo, após a criação desses indivíduos, o Algoritmo Híbrido (Algoritmo Genérico com o *Nearest-Neighbor*) não consegue evoluir a população, como a **figura 4** ilustra.

O algoritmo *Nearest-Neighbor* é utilizado para gerar a quantidade de indivíduos na população inicial definida pelo parâmetro *AmountPopulationWithNN*. Para calcular o número de indivíduos que serão geradas aleatoriamente é utilizado este cálculo:

$$\text{Número De Individuos Aleatorios} = \text{InitialPopulation} - \text{AmountPopulationWithNN} \quad (6.1)$$

Figura 4 – Evolução da rota com o algoritmo híbrido (GANN).



Fonte: Do autor.

Sendo que a variável *InitialPopulation* na **equação 6.1** é um parâmetro do algoritmo que armazena a quantidade total de indivíduos que a população inicial deve conter antes do processo de evolução ocorrer, como o cruzamento e a mutação.

6.1 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 GANN 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* for *true*,
- **InitialPopulationWithNN:** Se for igual a *true*, um indivíduo da população inicial será gerado utilizando o algoritmo NN;
- **AmountPopulationWithNN:** Define o número de indivíduos que serão gerados na população inicial 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 trocados com outros caixeiros. Se for igual a *false*, os pontos poderão ser trocados entre os caixeiros;
- **NNsizePart:** Define o tamanho da área em que os pontos serão gerados, por exemplo, se está variável possuir o valor 100, então será criado um espaço 100x100;

- **NNnPart:** Define o número de áreas em que o espaço será dividido para a utilização do algoritmo NN;
- **AmountMutation:** Define quantos pares de vértices serão trocados por uma mutação.
- **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;
- **AmountThread:** Define o número de *threads* para processar as áreas do NN.

6.2 Nearest-Neighbor modificado

O algoritmo *Nearest-Neighbor* (NN) possui uma alta complexidade. Para resolver este problema, o plano cartesiano pode ser dividido em inúmeras áreas de tamanhos iguais, sendo que o número de áreas é definido pelo parâmetro *NNnPart*, sendo sempre $NNnPart > 0$. Com isso, pode-se aplicar o NN em cada área, sendo assim, os pontos de uma área não podem ser comparados com os pontos de outras áreas reduzindo assim a complexidade para, na melhor hipótese, aproximadamente, $O(\frac{n^2-kn}{2k^2}k)$, sendo n o número total de pontos e k o número de áreas, e na pior hipótese $O(\frac{n^2-n}{2})$, quando todos os pontos estão em apenas uma área.

O tamanho do plano cartesiano é definido pelo parâmetro *NNsizePart*, utilizando o NN, o plano cartesiano será dividido em linhas e colunas, sendo que cada área será de tamanho igual, por exemplo, se o parâmetro *NNnPart* = 4, então o plano cartesiano com os vértices será dividido em 4 linhas e 4 colunas.

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

Cada cromossomo representa uma possível solução para o problema, então, para definir a estrutura do cromossomo do algoritmo foi utilizado o método *two-part*.

O *two-part* consiste em dividir o cromossomo em duas partes, uma parte armazena as informações da rota e a outra a informação dos caixeiros. Considere o cromossomo da **figura 5**.

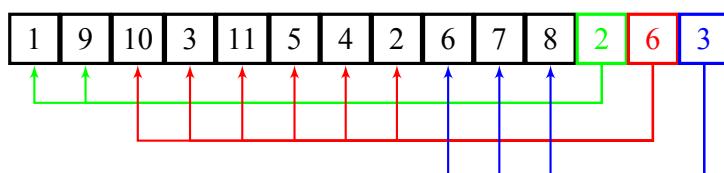
Figura 5 – Cromossomo representado pelo método *two-part*.

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

Fonte: Do autor.

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 6**).

Figura 6 – Cromossomo - *two-part*.



Fonte: Do autor.

No algoritmo híbrido desenvolvido neste trabalho, o número de pontos serão distribuídos entre os caixeiros de forma igual caso o indivíduo seja gerado aleatoriamente, se o indivíduo foi gerado pelo algoritmo *Nearest Neighbor* modificado, as áreas que este algoritmo gerou, explicado na seção 6.2, serão distribuídas de forma igual, permitindo que cada caixeiro fique responsável por visitar uma mesma quantidade de áreas que os outros.

6.4 Avaliação do Indivíduo (*fitness*)

Para cada novo indivíduo gerado, é calculada a distância dos pontos da rota, por meio da distância euclidiana bidimensional que pode ser definida pela **equação 6.2**.

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \quad (6.2)$$

Sendo que a variável d é a distância entre dois vértices do grafo, o valor d se torna o valor do peso da aresta que liga os dois vértices.

O cálculo da distância total (dt) é dado por:

$$d = \sum_{j=1}^c \sum_{i=1}^{a_j-1} \sqrt{(vcx_{ji} - vcx_{ji+1})^2 + (vcy_{ji} - vcy_{ji+1})^2} \quad (6.3)$$

$$dt = d + \sum_{i=1}^c (\sqrt{(depX - vcx_{i1})^2 + (depY - vcy_{i1})^2} + \sqrt{(depX - vcx_{iq})^2 + (depY - vcy_{iq})^2}) \quad (6.4)$$

Considerando que a é um vetor que representa o número total de vértices de cada caixeiro, sem considerar as arestas que estão vinculadas ao depósito, vcx e vcy são matrizes de duas dimensões, sendo que o primeiro índice representa o caixeiro e o segundo índice representa o vértice pertencente ao caixeiro do primeiro índice, que armazenam os valores de x e y respectivamente. As variáveis $depX$ e $depY$ armazenam as posições do depósito, onde os caixeiros irão iniciar seus trajetos, assim:

$$vcx, vcy, depX, depY \in \mathbb{R} \quad (6.5)$$

A variável c representa o número de caixeiros viajantes, e a variável q representa o valor do último vértice do caixeiro.

Na **equação 6.3** é calculada a distância das conexões entre os vértices, não considerando o depósito. Já na **equação 6.4** é adicionada a distância do depósito para o primeiro e o último vértice da rota de cada caixeiro, finalizando o cálculo da distância total.

O algoritmo considera o melhor indivíduo aquele que tiver a menor distância total.

6.5 Cruzamento

O programa proposto utiliza o operador de cruzamento PMX, descrito no capítulo (4.1.1.1).

A cada iteração (geração) é realizado um cruzamento, para isso são selecionados, aleatoriamente, dois cromossomos da população. No cruzamento são gerados dois novos indivíduos, esses indivíduos serão avalizados pela função *fitness* e inseridos na população.

6.6 Mutação

O programa proposto utiliza o operador de mutação EM (*Exchange Mutation*), descrito no capítulo 4.1.2. A chance do cromossomo sofrer mutação é definida no parâmetro *RateMutation*, sendo que $\{RateMutation \in \mathbb{R} \mid 0 \leq RateMutation \leq 1\}$. O parâmetro *AmountMutation*, considerando que $\{AmountMutation \in \mathbb{N}^*\}$, estabelece quantos pares de genes (vértices) do cromossomo serão trocados.

6.7 Desempenho

Para mensurar o desempenho do algoritmo foram realizados diversos testes com parâmetros diferentes, levando em consideração o tamanho da rota e o tempo de execução. A base de testes TSPLIB foi utilizada apenas para validar o algoritmo, pois esta base apenas contém testes para o problema do caixeiro viajante, sendo que o algoritmo proposto tenta solucionar o problema dos múltiplos caixeiros viajantes.

6.7.1 Requisitos de Sistema

O algoritmo foi implementado utilizando a linguagem C++ e compilado com o compilador GCC versão 4.8, utilizando os parâmetros *-pthread -O3 -std=c++11*. Os resultados gráficos das rotas foram gerados pela biblioteca BITMAPIMG¹. Foi desenvolvida esta biblioteca, pois o *software gnuplot* utilizado inicialmente para gerar as figuras não tinha um bom desempenho para plotar mais de 100.000 pontos, e os testes estavam consumindo muito tempo para plotar a imagem dos resultados e, também, não foi encontrada uma biblioteca para plotar as figuras de uma maneira simples. Portanto, foi desenvolvida a biblioteca BITMAPIMG com o objetivo de plotar linhas e

¹ A biblioteca BITMAPIMG foi desenvolvida pelo autor e está disponível em <<https://github.com/VictorCarlquist/BITMAPIMG>>.

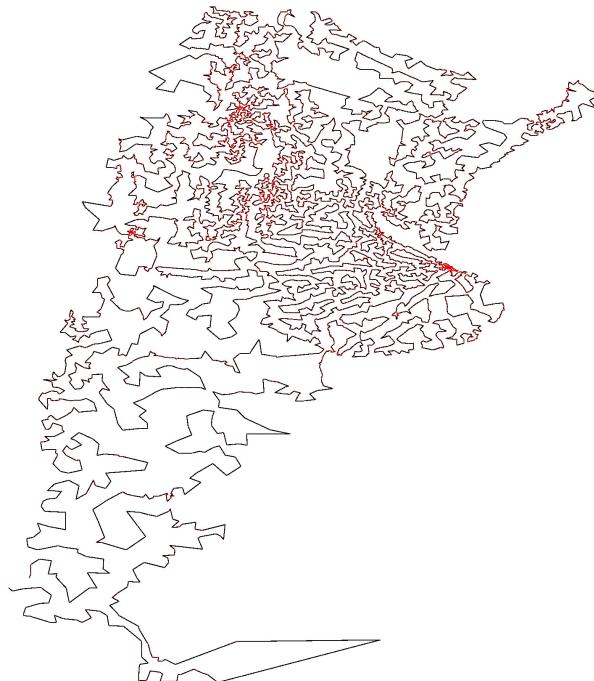
pontos de maneira simples, com isso foi possível gerar as figuras mais rapidamente.

O computador que executou os testes possui processador Intel®Core™ i5 2,4GHz, 4GB de memória RAM. É necessário ressaltar que os tempos de execuções entre os resultados da base de testes TSPLIB e o algoritmo desenvolvido pelo autor possuem discrepâncias, pois as arquiteturas dos processadores são diferentes, exceto no teste PR2392, portanto, os testes entre esses resultados são utilizados para, apenas, validar o algoritmo.

6.7.2 Resultados do Problema do Caixeiro Viajante (TSP)

É necessário medir o desempenho do Algoritmo Híbrido contra o Problema do Caixeiro Viajante, pois com os resultados é possível se ter uma base de comparação para constatar se o Algoritmo Híbrido está sendo executado corretamente, já que não foi possível encontrar nenhuma base de testes para o problema dos múltiplos caixeiros viajantes.

Figura 7 – TSPLIB - Imagem gerada pelo algoritmo LKH, Argentina.



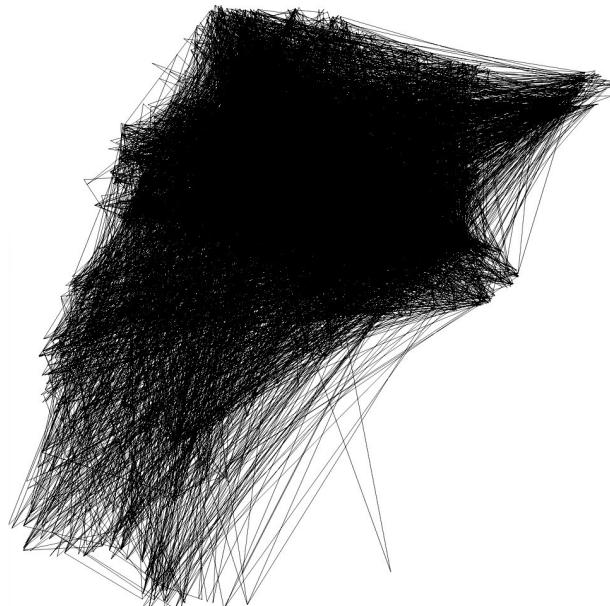
Fonte: TSPLIB.

Foram utilizados três testes da base de dados do *TSPLIB*. Os testes escolhidos contém todas as cidades da Argentina (9152 cidades), China (71.009 cidades) e um teste denominado PR2392 (2392 pontos) fornecido pela *Tektronics Incorporated* e armazenado no *TSPLIB*.

A rota ótima para o Problema do Caixeiro Viajante (TSP) utilizando a Argentina é de 837.479 km (**figura 7**). O melhor algoritmo para este problema encontrou a rota ótima em 24.301 segundos

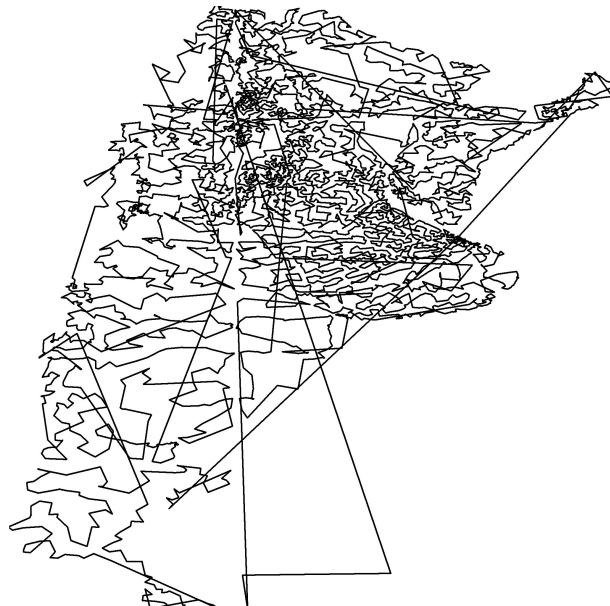
(aproximadamente 6 horas e 45 minutos) sendo executado em uma arquitetura Sun Ultra 80 450 MHz.

Figura 8 – Teste utilizando o Algoritmo Genético - Argentina.



Fonte: Do autor.

Figura 9 – Teste utilizando o Algoritmo *Nearest-Neighbor* - Argentina.

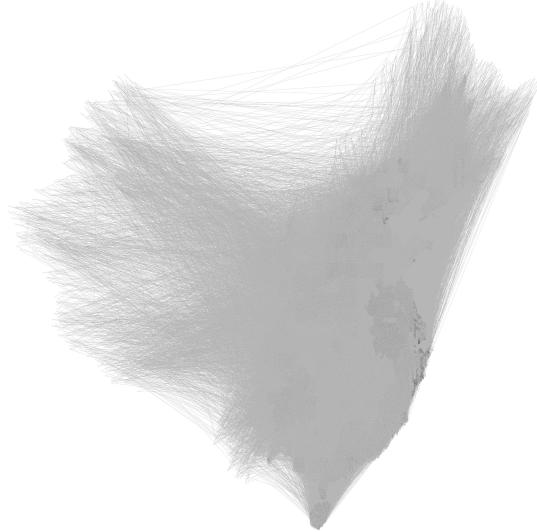


Fonte: Do autor.

Executando o teste com as cidades da Argentina utilizando o Algoritmo Genético “tradicional”, foi gerada uma rota com tamanho de 67.889.725,27 km (**figura 8**) e, com o Algoritmo Híbrido (GANN), gerou uma rota com tamanho de 1.182.405,53 km (**figura 9**), ambos com apenas um caixeiro. Isso demonstra que o algoritmo desenvolvido pode ser utilizado para se ter uma solução rápida, mas com uma qualidade menor; no Problema do Caixeiro Viajante, a rota gerada foi 41,18% maior, tendo um tempo de execução de 13,41 segundos. Utilizando apenas o *Nearest-Neighbor*

modificado, o tempo de execução é reduzido para, aproximadamente, 1,16 segundo com o mesmo tamanho de 1.182.405,53 km.

Figura 10 – Teste utilizando o Algoritmo Genético - China.



Fonte: Do autor.

Os teste com as cidades da China utilizando apenas o Algoritmo Genético resultaram em uma rota de 804.136.489,44 km (**figura 10**), executado em 97,53 segundos com 8000 gerações, e utilizando o GANN, o tamanho da rota foi de 5.884.954,34 km, executado em 75,93 segundos (**figura 11**). Uma rota ótima já encontrada, segundo o TSPLIB, é de 4.566.563 km com o tempo de execução de 72.945 segundos utilizando um processador Sun Ultra 80 450 MHz.

Figura 11 – Teste utilizando o GANN - China.

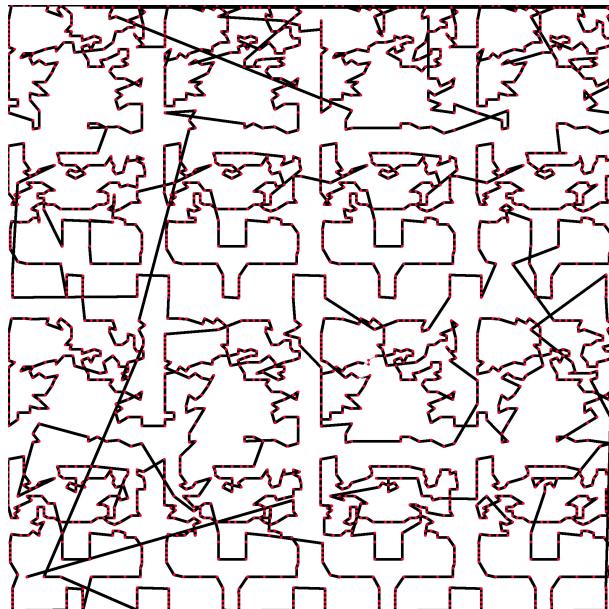


Fonte: o autor.

6.7.2.1 GANN versus LKH

O melhor algoritmo que é utilizado para gerar as menores rotas do TSPLIB é o LKH (Lin-Kernighan). Este algoritmo é especializado em *exchanges* (trocas), ou seja, são realizadas diversas trocas de arestas entre os vértices, obtendo uma rota mais eficiente, demonstrando ser um bom algoritmo para resolver o Problema do Caixeiro Viajante (HELSGAUN, 2007).

Figura 12 – Resultado do teste PR2392 gerado por GANN.



Fonte: Do autor.

Foi executado o teste PR2392 (2392 pontos) utilizando o LKH e o GANN, sendo que ambos foram executados no mesmo computador². Neste teste o LKH gerou uma rota com 378.032 km em 0,70 segundos, já o algoritmo GANN gerou 481.994,76 km em 0,30 segundos (**figura 12**). O algoritmo GANN, apesar de não gerar a melhor rota, é eficiente em encontrar uma solução aceitável em um menor tempo.

² Processador Intel®Core™ i5 2,4 Hz com 4GB de memória RAM.

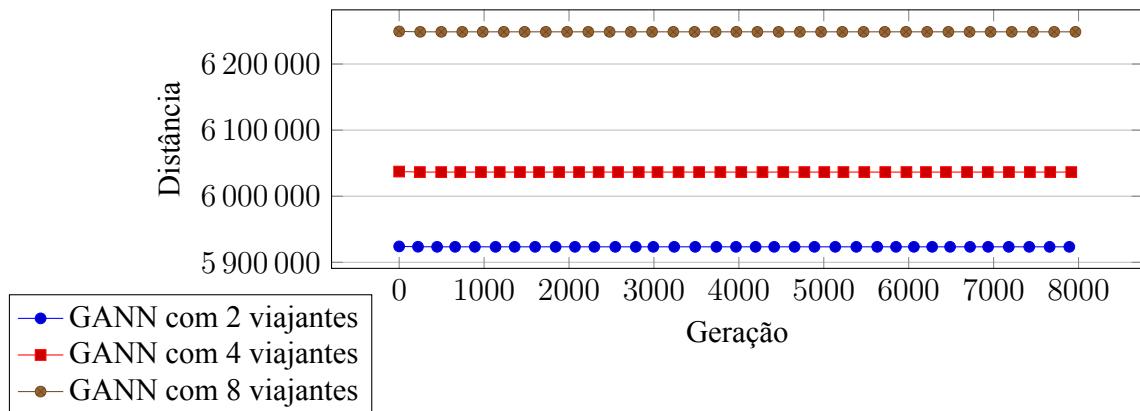
6.7.3 Resultados do Problema dos Múltiplos Caixeiros Viajantes (mTSP)

Nas próximas seções serão demonstrados os resultados que o algoritmo híbrido desenvolvido obteve em gerar uma solução para o Problema dos Múltiplos Caixeiros Viajantes.

6.7.3.1 Múltiplos Caixeiros - GANN

O Algoritmo Híbrido mostrou-se eficiente na solução do Problema dos Múltiplos Caixeiros Viajantes (mTSP), pois o tempo de execução se manteve o mesmo com diversos caixeiros, em média 175 segundos.

Figura 13 – Evolução da rota com diversos caixeiros viajantes (GANN).



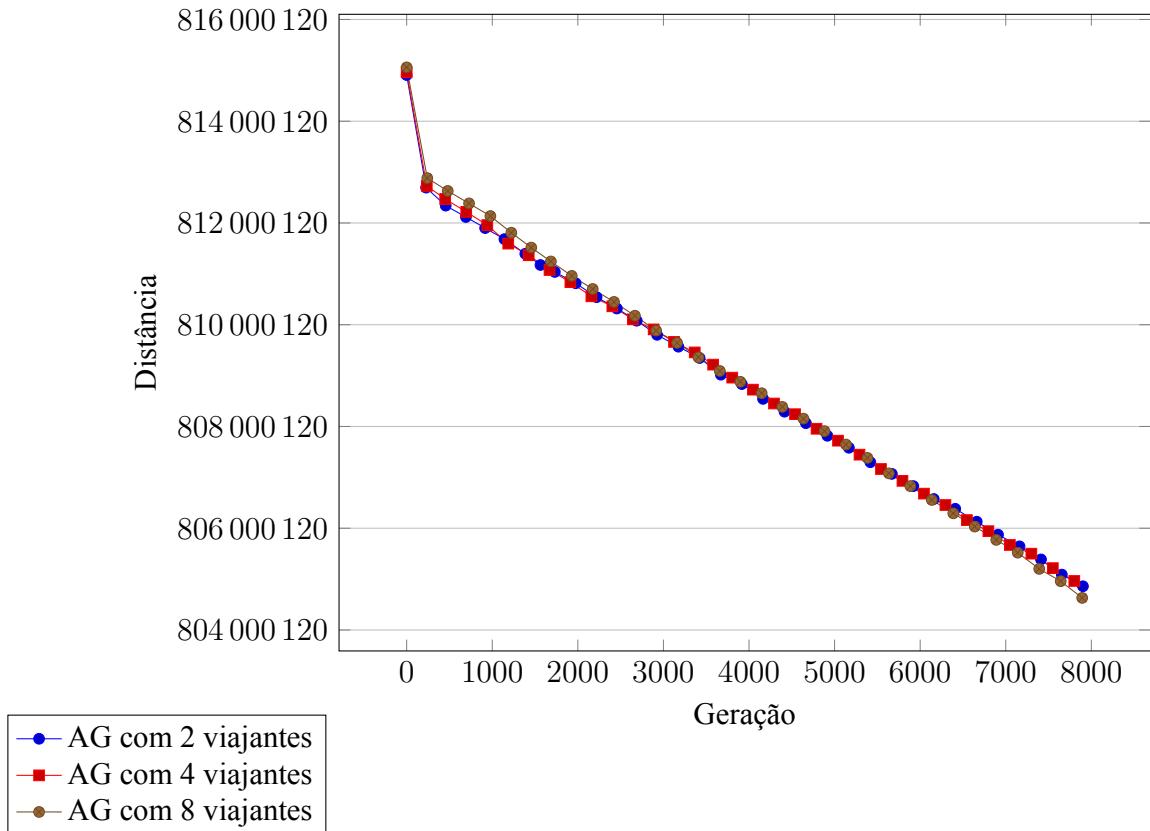
Fonte: Do autor.

Como a **figura 13** apresenta, o tamanho da rota aumenta conforme o número de caixeiros também aumenta, porque é necessário adicionar um caminho de saída do depósito e um outro caminho de entrada para cada caixeiro, portanto, quanto maior for o número de caixeiros, mais caminhos de entrada e saída serão adicionados, influenciando no tamanho da rota.

6.7.3.2 Múltiplos Caixeiros - GA

Foram realizados alguns testes para verificar o comportamento do Algoritmo Genético em resolver o Problema dos Múltiplos Caixeiros Viajantes.

Figura 14 – Evolução da rota com diversos viajantes (GA).



Fonte: Do autor.

A **figura 14** ilustra que, apesar de que cada teste possui um número de caixeiros diferentes, todos os resultados tem um comportamento semelhante. Vale salientar que o resultado com 8 caixeiros iniciou-se com a maior rota e terminou com o melhor resultado das 3 soluções expostas nesta figura.

6.7.4 Threads

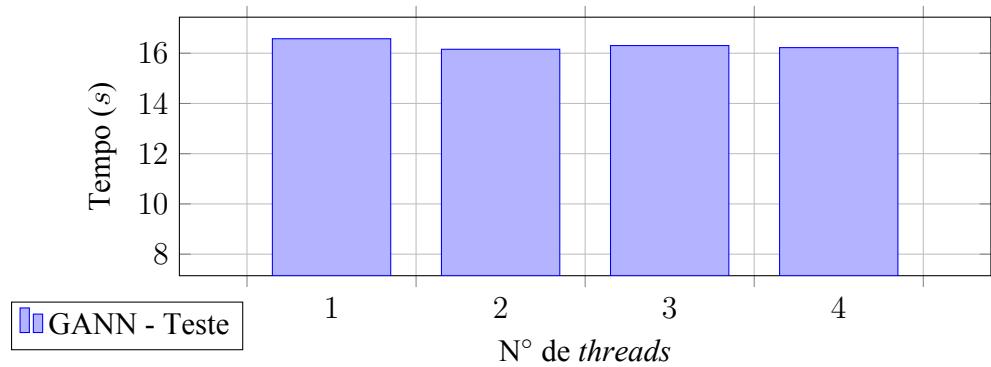
Uma *thread* possui um ID, um contador de programa, um conjunto de registradores e uma pilha. Também compartilha com outras *threads* em um mesmo processo a seção de código, seção de dados e outros recursos do sistema operacional (SO), como arquivos abertos e sinais. O processo tendo mais de uma *thread* significa que é possível realizar mais de uma tarefa ao mesmo tempo (SILBERSCHATZ, 2008).

6.7.4.1 Threads (GANN)

Foram implementados *threads* para otimizar o tempo de execução do *Nearest-Neighbor*.

O parâmetro *NNnPart* define a quantidade de áreas em que o espaço será dividido, essas áreas serão distribuídas igualmente para cada *thread*, dividindo a carga de processamento entre os núcleos, caso o *hardware* tenha mais que um processador ou núcleo (*core*).

Figura 15 – Desempenho do algoritmo utilizando *threads*.



Fonte: Do autor.

O algoritmo *Nearest Neighbor* divide o plano cartesiano em diversas áreas, assim, cada área pode ser processada por uma *thread*, podendo aumentar o desempenho do algoritmo na geração da população inicial. Como a **figura 15** demonstra, nos 4 testes realizados, a utilização das *threads* não teve impacto no tempo de execução, pois a implementação do algoritmo está tendo um falso paralelismo que pode estar sendo causado pelo acesso à memória RAM (*Random Access Memory*) pelas múltiplas *threads*, mas a razão deste resultado precisará ser investigada em um trabalho futuro, pois não é um problema trivial.

7 CONCLUSÃO E TRABALHOS FUTUROS

O Algoritmo Híbrido desenvolvido apresentou um bom tempo de execução para gerar uma solução para o problema dos múltiplos caixeiros. Pode-se notar nas imagens das rotas que ainda é possível gerar rotas menores, mostrando que o algoritmo ainda pode ser otimizado.

Como apresentado nesta pesquisa, o Algoritmo Genético mostrou-se eficiente em uma população gerada aleatoriamente, mas com a utilização do algoritmo *Nearest Neighbor* modificado, para gerar a população inicial, o Algoritmo Genético não consegue evoluir a população com eficácia, pois são necessárias muitas gerações para haver uma melhoria na rota.

De fato, com este algoritmo é possível obter uma noção do tamanho da rota, podendo ser utilizado para comparar grafos diferentes e auxiliar na escolha do grafo que possui a menor rota.

Apesar de não ser o objetivo do algoritmo resolver o problema de um caixeiro viajante, ele também se mostrou eficiente em encontrar uma rota com tempo de execução pequeno.

Portanto, este trabalho demonstra que um algoritmo híbrido pode ser utilizado para solucionar o Problema dos Múltiplos Caixeiros Viajantes.

Como trabalho futuro, poderiam ser adicionadas novas estratégias para gerar a população inicial e realização dos cruzamentos dos cromossomos. Com essas implementações a probabilidade do Algoritmo Genético evoluir a população poderia aumentar.

Uma segunda abordagem para otimizar a solução poderia ser adicionar novas análises ao algoritmo *Nearest Neighbor*, fazendo com que a análise de seus vizinhos seja melhor gerenciada, podendo eliminar o Algoritmo Genético.

A implementação de *threads* utilizando GPUs (*Graphics Processing Unit*) poderia deixar mais rápido o algoritmo, já que cada geração do algoritmo genético ou a otimização de cada parte (área) do *Nearest Neighbor* pode ocorrer separadamente.

REFERÊNCIAS

- ARNBORG, S.; PROSKUROWSKI, A. Linear time algorithms for np-hard problems restricted to partial k-trees. *Elsevier Science Publishers*, Março 1989. 12
- 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. 18
- 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. 12
- 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. 12, 13
- CARVALHO, A. P. de Leon F. de. *Algoritmos Genéticos*. 2013. Disponível em: <<http://www2.icmc.usp.br/~andre/research/genetic/>>. 14, 20
- CORREIA, M. Algoritmos genéticos. *dosalgarves*, p. 36–43, junho 2003. 14
- DAVENDRA, D. *Traveling Salesman Problem, Theory, and Applications*. [S.l.]: InTech, 2010. 16
- DIAS, I. P. da S.; KUWAHARA, M. Y. Sistema de transporte público urbano da rmsp e seus impactos ambientais. *Jovens Pesquisadores*, Janeiro 2009. 10
- HELSGAUN, K. *An Effective Implementation of K-opt Moves for the Lin-Kernighan TSP Heuristic*. [S.l.]: Roskilde University, 2007. 29
- HOLLAND, J. H. *Adaptation in Natural and Artificial Systems*. [S.l.]: University of Michigan Press, 1975. 14
- LOPES, A. M. *Uma Abordagem Multiobjetivo para o Problema de Corte de Estoque Unidimensional*. Dissertação (Mestrado) — Universidade Estadual Paulista, 2009. 14
- MALAQUIAS, N. G. L. *Uso dos algoritmos genéticos para a otimização de rotas de distribuição*. Dissertação (Mestrado) — Universidade Federal de Uberlândia, 2006. 15, 16
- MIYAZAWA, F. K. *Otimização*. 2002. Disponível em: <<http://www.ic.unicamp.br/~fkm/lectures/otimo.pdf>>. 17
- RESEARCH, O. of N. *The Traveling Salesman Problem*. 2014. Disponível em: <<http://www.math.uwaterloo.ca/tsp/index.html>>. 11
- 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. 18
- SILBERSCHATZ, A. *Sistemas operacionais com Java*. [S.l.]: Elsevier, 2008. 31
- WU, L. *O problema de roteirização periódica de veículos*. Tese (Doutorado) — Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Transportes., 2007. 18

APÊNDICE A – CÓDIGO FONTE - MAIN.CPP

```

/*
 * Author: Victor Carlquist
 * Date: 22/09/2014
 * IFSP - Campos do Jordão
 * e-mail: victorcarlquist@gmail.com
 */

#include <iostream>
#include <cstdlib>
#include <cctime>
#include <fstream>
#include <cstring>
#include <search.h>
#include <algorithm>

#include <float.h>
#include <cmath>

#define BMP_WIDTH 2048
#define BMP_HEIGHT 2048
#define TOTAL_COLOR 14
// Paleta de cores em RGB.
uint8_t COLORS[TOTAL_COLOR][3] = {
    {255, 0, 0},
    {0, 0, 0},
    {0, 0, 255},
    {255,255,0},
    {0,255,0},
    {255, 0, 255},
    {0, 255, 255},
    {0, 255, 255},
    {144, 238, 144},
    {139, 0, 0},
    {139, 0, 139},
    {0, 139, 139},
    {0, 0, 139},
    {79, 79, 79}
};
extern "C"{
    #include "bitmap.h"
}
#include "ga.h"

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 path)
{
    string line;
    ifstream myfile (path);

    if (myfile.is_open()) {
        vector<string> c = split(line, " ");
        while (getline (myfile, line)) {
            c = split(line, " ");
            //cout << c[0] << " " << c[1] << " " << c[2] << "\n";
            //if (c[0] != "7179")
                world->setCity(atof(c[1].c_str()), atof(c[2].c_str()), atoi(c[0].c_str()));
        }
        myfile.close();
        cout << "Loaded file.\n";
    }
    else
        cout << "Unable to open file";
}

void loadCityWorld(GA *world)
{
    string line;
    ifstream myfile ("../world.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())+200, atof(c[2].c_str())+200, atoi(c[0].c_str()));
        }
        myfile.close();
        cout << "Loaded file.\n";
    }
    else
        cout << "Unable to open file";
}

int main(int argc, char *argv[])
{
    srand(11111);

    clock_t begin, end;
    double time_spent;

    begin = clock();
    if (argc == 2) {
        if (strcmp(argv[1], "-h") == 0) {
            std::cout << "Parameter\n"

```

```

    "world.MaxPopulation      = 20;\n"
    "world.InitialPopulation  = 10;\n"
    "world.InitialPopulationWithNN = true;  utilizar o algortimo Nearest Neighbor\n"
    "world.AmountPopulationWithNN = 1;" 
    "world.MutationRouteItself = true;  realiza a mutação em cada rota de cada viajante , sem
        afetar a outra\n"
    "world.NNsizePart         = 382;  Arg: 75000 world:382 Tamanho de cada regiao para a divisao
        do Nearest neighbor\n"
    "world.NNnPart             = 40;  Arg: 4 World: 300 ou 40  Quantidade de regioes para a
        divisao do Nearest neighbor\n"
    "world.RateGeneration      = 0;  taxa de substituição dos individuos\n"
    "world.AmountMutation       = 1;  número randomico entre 0 e |AmountMutation|\n"
    "world.RateMutation          = 1;  quantidade de mutação\n"
    "world.RateSalesmanMutation = 0;  taxa de mutação\n"
    "world.Generation           = 5000;" 
    "world.Deposit              = 1;  O deposito está na cidade 1\n"
    "world.Salesman             = 10;\n"
    "world.SaveBetterChromo     = true;  Evita que o melhor individuo sofra mutação\n"
    "world.AmountThread          = 4;\n";
    pthread_exit(NULL);
}

pthread_exit(NULL);
}

GA world;
if (argc == 1) {
    world.MaxPopulation      = 20;
    world.InitialPopulation  = 10;
    // Utilizar o algortimo Nearest Neighbor.
    world.InitialPopulationWithNN = true;
    // Quantidade NN gerado na populacao inicial
    world.AmountPopulationWithNN = 1;
    // realiza a mutação em cada rota de cada viajante , sem afetar a outra
    world.MutationRouteItself = true;
    // pp: 16000 China: 150000 Arg: 75000 world:382
    // Tamanho de cada regiao para a divisao do Nearest neighbor
    world.NNsizePart         = 16000;
    // Arg: 4 World: 300 ou 40
    // Quantidade de regioes para a divisao do Nearest neighbor
    world.NNnPart             = 1;
    // taxa de substituição dos individuos
    world.RateGeneration      = 0;
    // número randomico entre 0 e |AmountMutation|
    world.AmountMutation       = 1;
    // taxa de mutação
    world.RateMutation          = 0.1;
    // taxa de mutação
    world.RateSalesmanMutation = 0;
    world.Generation           = 10;
    // O deposito está na cidade 1
    world.Deposit              = 1;
    world.Salesman             = 1;
    // Evita que o melhor individuo sofra mutação
    world.SaveBetterChromo     = true;
    // Numero de threads do NN
    world.AmountThread          = 1;
    loadCity(&world, "../pr2392.tsp");
} else {
    world.MaxPopulation      = std::atoi(argv[1]);
    world.InitialPopulation  = std::atoi(argv[2]);
}

```

```

world.InitialPopulationWithNN = (strcmp(argv[3], "true") == 0) ? true : false; // utilizar o
algoritmo Nearest Neighbor
world.AmountPopulationWithNN = std::atoi(argv[4]); // 
world.MutationRouteItself = (strcmp(argv[5], "true") == 0) ? true : false; // realiza a mutação
em cada rota de cada viajante, sem afetar a outra
world.NNsizePart = std::atoi(argv[6]); // Arg: 75000 // world:382 // Tamanho de
cada regiao para a divisao do Nearest neighbor
world.NNnPart = std::atoi(argv[7]); // Arg: 4 World: 300 ou 40 // Quantidade
de regioes para a divisao do Nearest neighbor
world.RateGeneration = std::atof(argv[8]); // taxa de substituição dos individuos
world.AmountMutation = std::atoi(argv[9]); // número randomico entre 0 e |
AmountMutation|
world.RateMutation = std::atof(argv[10]); // taxa de mutação
world.RateSalesmanMutation = std::atof(argv[11]); // taxa de mutação
world.Generation = std::atoi(argv[12]);
world.Deposit = std::atoi(argv[13]); // O deposito está na cidade 0(zero)
world.Salesman = std::atoi(argv[14]);
world.SaveBetterChromo = (strcmp(argv[15], "true") == 0) ? true : false; // Evita que o melhor
individuo sofra mutação
world.AmountThread = std::atoi(argv[16]);
loadCity(&world, argv[17]);
}

unsigned int i,j,k,m;
ofstream myfile;
GA::Chromossome *a;

/*
// Gera a configuração do arquivo para que o gnuplot seja chamado.
// O gnuplot não é mais utilizado, pois foi desenvolvido a biblioteca
// BITMAPIMG para gerar as figuração.
myfile.open("gnuplot.conf");
myfile << "set term png size 1000,1000 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();
*/
// Inicia a execução do algoritmo.
world.evolution();
end = clock();

```

```

cout << "Gerando Imagem...\n";
/*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();
for (j = 0;j < a->Salesman.size();++j) {
    cout << a->Salesman[j];
    if (j < a->Salesman.size()-1)
        cout << "-";
}
cout << "\nDist: " << a->dist << endl;

m = 0;

// BMPIMG
// Gera a figura do resultado.
bmp_pixel color = bmp_create_pixel(255,255,255);
BMPFILE *bmpFile = bmp_init_bmp(BMP_WIDTH,BMP_HEIGHT, color, BMP_SCALE_FIT_XY);
color = bmp_create_pixel(COLORS[1][0], COLORS[1][1], COLORS[1][2]);

GA::Point paux = world.cities[world.Deposit];
bmp_pixel colorD = bmp_create_pixel(COLORS[0][0], COLORS[0][1], COLORS[0][2]);
bmp_add_dot(bmpFile, paux.x, paux.y,0 , colorD);
for (j=0;j<a->Salesman.size();++j) {
    color = bmp_create_pixel(COLORS[j%TOTAL_COLOR+1][0], COLORS[ j%TOTAL_COLOR+1][1], COLORS[ j%TOTAL_COLOR
+1][2]);
    bmp_add_line(bmpFile,paux.x, paux.y, a->Gene[m].x, a->Gene[m].y, 1, color);
    for (k=0;k<a->Salesman[j];++k)
        bmp_add_line(bmpFile,paux.x, paux.y, a->Gene[m].x, a->Gene[m].y,1, color);
        bmp_add_dot(bmpFile, a->Gene[m].x, a->Gene[m].y, 1, colorD);
        paux = a->Gene[m];
        m++;
}
bmp_add_line(bmpFile,paux.x, paux.y, a->Gene[m-1].x, a->Gene[m-1].y, 1, color);
}

if (argc != 1)
    bmp_generate_bmp(bmpFile, argv[18]);
else
    bmp_generate_bmp(bmpFile, "teste.bmp");

// END BMPIMG

/*
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'");
*/
*/



char str[80];
std::strcat(str, "xdg-open ");

if (argc == 1)
    std::strcat(str, "teste.bmp");
else
    std::strcat(str, argv[18]);

//system(str);

time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
cout << "Tempo: " << time_spent << endl;
pthread_exit(NULL);
}

```

APÊNDICE B – CÓDIGO FONTE - GA.H

```

/*
 * Author: Victor Carlquist
 * Date:22/09/2014
 * IFSP - Campos do Jordão
 */

#ifndef GA_H
#define GA_H

#include <iostream>
#include <vector>
#include <CL/cl.hpp>

#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 Chromossome
    {
        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 AmountPopulationWithNN; // = 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 AmountMutation;     //      = 1;      // troca dois genes de posição
    unsigned int Generation;         //      = 1000; //
    unsigned int Deposit;             //      = 0;      // O deposito está na cidade 0(zero)
    unsigned int Salesman;            //      = 2;      // quantidade de viajantes
    unsigned int AmountThread;       // Número total de thread
    void addChromo(Chromossome *a);
    virtual void fitness(Chromossome *a);
};

```

```

virtual void crossover(Chromossome *f1, Chromossome *f2);
virtual void mutation(Chromossome *chromo); // EX
// muda o numero de objetivos que o viajante deve visitar
virtual void salesmanMutation(Chromossome *chromo);
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 ultimos elementos representa o caixeleiro 1 e 2, sendo que
 * o caixeleiro 1 ira visitar os dois primeiros elementos do vetor,
 * e o segundo caixeleiro ira visitar os três ultimos 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: Date:22/09/2014
 * 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>
#include <CL/cl.hpp>

pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;

// 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)
{
    register unsigned int i;
    const unsigned int max = vec->size();
    for (i=0;i<max;++i) {
        if ((*vec)[i].id == target.id)
            return i;
    }
    return vec->size();
}

/*

```

```

 * Adiciona um indivíduo na população
 */
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);
        }
    }
}

/*
 * Adiciona uma nova cidade (vértice)
 */
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);
}

/*
 * Calcula a distância entre dois pontos
 */
static double euclideanDistance(GA::Point *a, GA::Point *b)
{
    if (a == nullptr || b == nullptr)
        return 0;
    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 distâncias euclidianas
 * Menor distância , melhor é o individuo
 */
void GA::fitness(Chromossome *a)
{
    register unsigned int i = 0;
    unsigned int j, k;
    double dist = 0;

```

```

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

/*
 * Crossover
 * O cruzamento usará 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());

    // Armazena a quantidade de índices reservados para a rota,
    // usando apenas a primeira parte do cromossomo,
    // mas descontando 3 índices para o PMX funcionar.
    unsigned int x = rand() % (this->getRouteIndexMax() - 2);

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

    // Os genes do novo indivíduo é inicializado com o valor -1.
    std::fill_n(s1->Gene.begin(),this->getRouteIndexMax(),foo);
    std::fill_n(s2->Gene.begin(),this->getRouteIndexMax(),foo);

    // Três genes do pai1 e do pai2 são passados para filho1 e filho2.
    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 cromossomo filho1.
    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];
    }

}

// A quantidade das rotas atribuídas para cada caixeiro do indivíduo pai1 é
// passado para o filho1.
for (i = 0; i<f1->Salesman.size(); ++i)
    s1->Salesman.push_back(f1->Salesman[i]);

// Evita que as cidades sejam repetidas dentro do cromossomo filho2.
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];
    }
}
// A quantidade das rotas atribuídas para cada caixeiro do indivíduo pai2 é
// passado para o filho2.
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 ou mais elementos de lugar
 */
void GA::mutation(Chromosome *chromo)
{
    unsigned int a,b;

    for (unsigned int k = 0; k < this->AmountMutation; ++k)
    {
        if (!this->MutationRouteItself) {
            a = rand() % this->getRouteIndexMax();
            b = rand() % this->getRouteIndexMax();

            GA::Point x = chromo->Gene[a];
            chromo->Gene[a] = chromo->Gene[b];
            chromo->Gene[b] = x;
        } else {
            // Troca as posicoes entre os pontos de um mesmo caixeiro
            unsigned int i, offset = 0;
            i = rand() % this->Salesman;

            for (unsigned int j = 0; j < i; ++j)
                offset+=chromo->Salesman[j];

            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;
    }
}

this->fitness(chromo);
}

/*
 * Muda a quantidade de vértices atribuídos para cada caixeiro.
 */
void GA::salesmanMutation(Chromossome *chromo)
{
    unsigned int a,b,c;

    std::vector<unsigned int> newSalesmans;

    // O '-1' é o desconto da cidade depósito.
    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 contém os pontos de uma região.
 */
std::vector<GA:: Point> GA::NN(std::vector<Point> citiesReg)
{
    unsigned int j, indClosest;
    double dist, closestDist;

    std::vector<Point> routeNN;
    routeNN.reserve(citiesReg.size());

    routeNN.push_back(citiesReg[0]);
    citiesReg.erase(citiesReg.begin());

    // Quando o vetor |citiesReg| estiver vazio, significa que todos os
    // vértices da região passada no parâmetro já foi colocado em ordem para
    // ser visitado.
    while (citiesReg.size() > 0) {
        j = 0;
        closestDist = std::numeric_limits<double>().max();
        while (j < citiesReg.size()) {
            if (routeNN.back().id != citiesReg[j].id) {
                // std::cout << citiesReg.size() << " " << j << "\n";
                dist = euclideanDistance(&routeNN.back(), &citiesReg[j]);
                if (closestDist > dist) {
                    closestDist = dist;

```

```

        indClosest = j;
    }
}
j++;
}
// std::cout << citiesReg.size() << std::endl;
routeNN.push_back(citiesReg[indClosest]);
citiesReg.erase(citiesReg.begin() + indClosest);
}

return routeNN;
}

/*
 * Estrutura utilizada para passar valores para as threads.
 */
typedef struct {
    std::vector<std::vector<GA::Point> > *PartNN;
    GA *global;
    int id;
    GA::Chromossome *a;
} argStruct;

/*
 * A função NN é chamada dentro de cada thread, assim, cada região é processado
 * por uma thread. Não se pode afirmar com certeza que cada thread será executado
 * em um núcleo ou processador ao mesmo tempo, pois quem faz essa escolha é o
 * Sistema Operacional.
 */
void *NNThread(void *arguments)
{
    argStruct *args = (argStruct *)arguments;
    GA::Chromossome *a = args->a;
    int auxMaxArea, maxArea = args->global->NNnPart * args->global->NNnPart / args->global->AmountThread;
    auxMaxArea = 0;
    if (args->id == args->global->AmountThread - 1)
        auxMaxArea = args->global->NNnPart * args->global->NNnPart % args->global->AmountThread;

    // A área de vértices a ser calculada pela thread é baseada no ID da
    // thread.
    for (int i = maxArea * args->id; i < maxArea * (args->id + 1) + auxMaxArea; ++i) {
        std::vector<GA::Point> *auxVec;
        auxVec = &(*args->PartNN)[i];
        if (!auxVec->empty()) {
            std::random_shuffle(auxVec->begin(), auxVec->end(), myrandom);
            *auxVec = args->global->NN(*auxVec);
            a->Gene.insert(a->Gene.end(), auxVec->begin(), auxVec->end());
        }
        // std::cout << "NN " << i << " id: " << args->id << " |: " << i << "\n";
    }

    pthread_exit(NULL);
}

/*
 * Gera a população 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 - this->InitialPopulationWithNN;
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 regiões , comparando os vizinhos que pertencem a mesma região do plano cartesiano
 */
// nPart , numero de regiões
// Ex: nPart = 4, divididos em 4 linhas e 4 colunas
// Ex: nPart = 8, divididos em 8 linhas e 8 colunas
if (!this->InitialPopulationWithNN)
    return;

unsigned int nPart = this->NNnPart;
float retSizeY = this->NNsizePart / nPart;
float retSizeX = this->NNsizePart / nPart;
float x, y;

for (uint32_t m = 0; m < this->AmountPopulationWithNN; ++m) {
    std::vector<std::vector<Point>> PartNN;
    for (i = 0;i < (nPart*nPart);++i) {
        std::vector<Point> lineReg;
        PartNN.push_back(lineReg);
    }

    // Separa os pontos em cada area.
    for (i=0;i<this->cities.size();++i) {
        x = y = 0;
        if (i == this->Deposit)
            continue;
        for (j = 0; j < (nPart*nPart); ++j) {
            if (this->cities[i].x >= x && this->cities[i].x < x+retSizeX
                && this->cities[i].y >= y && this->cities[i].y < y+retSizeY) {
                PartNN[j].push_back(this->cities[i]);
                break;
            }
        }
        if (j%nPart != 0 || j == 0)
    }
}

```

```

        x+=retSizeX;
    else {
        x = 0;
        y+=retSizeY;
    }
}

a = new GA::Chromossome();
pthread_t threads[this->AmountThread];
pthread_attr_t attr;
void *status;
int rc;

// Inicializa e seta uma thread joinable.
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

argStruct args[this->AmountThread];
for( i=0; i < this->AmountThread; i++ ){
    args[i].global = this;
    args[i].PartNN = &PartNN;
    args[i].id = i;
    args[i].a = new GA::Chromossome();
    // std::cout << "main() : creating thread , " << i << "endereco: "<< &args[i] << std::endl;
    rc = pthread_create(&threads[i], NULL, &NNThread, (void *)&args[i]);
    if (rc){
        // std::cout << "Error:unable to create thread , " << rc << std::endl;
        exit(-1);
    }
}

// Libera os atributo e fica aguardando outras threads.
pthread_attr_destroy(&attr);
for(i = 0; i < this->AmountThread; i++){
    rc = pthread_join(threads[i], &status);
    if (rc){
        // std::cout << "Error:unable to join , " << rc << std::endl;
        exit(-1);
    }
    // std::cout << "Main: completed thread id :" << i ;
    // std::cout << " exiting with status :" << status << std::endl;
}

// Junta todas as areas dentro de um indivíduo.
for(i = 0; i < this->AmountThread; i++) {
    a->Gene.insert(a->Gene.end(), args[i].a->Gene.begin(), args[i].a->Gene.end());
    delete args[i].a;
}

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

// Distribui as áreas para cada caixeiro.
i = (nPart*nPart)/this->Salesman;
k = 0;
if (i != 0)
    for (j=0;j<nPart*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 desmontar uma rota de um caixeiro pois
// foi preciso dar um objetivo para um caixeiro que nao 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++;
    if (j==this->Salesman)
        j = 0;
    for (k=0;k<this->Salesman;++k)
        if (a->Salesman[k] == 0 && max == 0)
            max++;
}

this->addChromo(a);
std::sort(this->Population.begin(),this->Population.end(),Csort);
// std::cout << "Adicionado: " << m << std::endl;
}
// Fim Nearest Neighbor
}
/*
 * Executa o núcleo do AG.
 * Neste método que são feitas as chamadas para a preparação, mutação,
 * cruzamento, avaliação (fitness), etc...
 */
void GA::evolution()
{
    // Gera a população inicial.
    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;
}

```

```

std::cout.setf(std::ios::fixed, std::ios::floatfield);
std::cout.setf(std::ios::showpoint);

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 << i << "," << this->Population[0]->dist << std::endl;
    }

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

    // Verifica se será realizado uma mutação.
    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));

    // Verifica se será realizado uma mutação na quantidade que cada
    // caixeleiro deve visitar.
    if (mr < this->RateSalesmanMutation) {
        b = rand()%this->Population.size();
        if (this->SaveBetterChromo) {
            if (b == 0)
                b = 1;
        }
        this->salesmanMutation(this->Population[b]);
    }

    // Seleciona dois indivíduos para fazer o cruzamento.
    a = rand() % this->Population.size();
    b = rand() % this->Population.size();
    this->crossover(this->Population[a], this->Population[b]);

}
this->finish = true;
//t1.join();

// Ordena a população, deixando o melhor indivíduo na posição 0.
std::sort(this->Population.begin(), this->Population.end(), CsortM);
}

/*
 * Plota a figura utilizando o software gnuplot
 */
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);
// }
}
```