UNIVERSIDADE DE CAXIAS DO SUL
ÁREA DO CONHECIMENTO DE CIÊNCIAS EXATAS E ENGENHARIAS

CÉSAR AUGUSTO GRAEFF

SOLUÇÃO PARALELA PARA UM SISTEMA DE ROTEIRIZAÇÃO UTILIZANDO O PROBLEMA DO CAIXEIRO VIAJANTE

CAXIAS DO SUL

2020

CÉSAR AUGUSTO GRAEFF

SOLUÇÃO PARALELA PARA UM SISTEMA DE ROTEIRIZAÇÃO UTILIZANDO O PROBLEMA DO CAIXEIRO VIAJANTE

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Bacharel em Ciência da Computação na Área do Conhecimento de Ciências Exatas e Engenharias da Universidade de Caxias do Sul.

Orientador: Prof. Dr. André Luis Martinotto

CAXIAS DO SUL

2020

CÉSAR AUGUSTO GRAEFF

Solução paralela para um sistema de roteirização utilizando o problema do caixeiro viajante

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Bacharel em Ciência da Computação na Área do Conhecimento de Ciências Exatas e Engenharias da Universidade de Caxias do Sul.

Aprovado em 16/12/2020

BANCA EXAMINADORA

_____

Prof. Dr. André Luis Martinotto
Universidade de Caxias do Sul - UCS

_____

Prof. Dr. Ricardo Vargas Dorneles
Universidade de Caxias do Sul - UCS

_____

Profa. Dra. Helena Graziottin Ribeiro
Universidade de Caxias do Sul - UCS

RESUMO

Este trabalho tem por principal objetivo o estudo de heurísticas para a solução do Problema do Caixeiro Viajante, em especial para a resolução do Problema do Caixeiro Viajante com Janelas de Tempo. O problema do Caixeiro Viajante possui complexidade NP-Difícil, sendo inviável a utilização de força bruta para a solução de grafos que apresentam um grande número de vértices. Deste modo, são utilizadas heurísticas que procuram reduzir significativamente o tempo de execução, porém não garantindo a solução ótima para o problema. Para o desenvolvimento deste trabalho optou-se pela utilização da heurística GENIUS. A implementação foi desenvolvida utilizando a linguagem de programação *C* e a biblioteca de *threads* OpenMP, de forma a explorar o paralelismo em arquiteturas com múltiplos núcleos de processamento. Para os testes foi utilizado o pacote de grafos proposto por Dumas em 1995. Já para a avaliação do desempenho foram calculados o *speedup* e a eficiência do algoritmo implementado. Os resultados obtidos através deste trabalho apresentaram um *speedup* de até 2.48 e uma eficiência de até 62% ao utilizar 4 núcleos de processamento.

Palavras-chave: Caixeiro Viajante. Janelas de Tempo. GENIUS. OpenMp

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ALGORITMOS

LISTA DE ABREVIATURAS E SIGLAS

PCV	*Problema do Caixeiro Viajante*
PCVJT	*Problema do Caixeiro Viajante com Janelas de Tempo*
SISD	*Single Instruction, Single Data*
SIMD	*Single Instruction, Multiple Data*
MISD	*Multiple Instruction, Single Data*
MIMD	*Multiple Instruction, Multiple Data*
GPU	*Graphics Processing Unit*
OpenMP	*Open Multi-Processing*
GENI	*Generalized Insertion*
US	*Unstringing and Stringing*
API	*Application Programming Interface*

SUMÁRIO

1 INTRODUÇÃO

Devido a expansão dos *e-commerces*, a área logística apresentou um aumento na demanda de serviços, provocando um aumento nos preços e no tempo de entrega. Para uma redução no custo final do produto e no prazo de entrega, torna-se necessário que as empresas do ramo sejam mais eficazes, melhorando seus processos e diminuindo seus custos (JUNIOR, 2018).

Um destes processos envolve a definição dos percursos a serem realizados pelos veículos responsáveis pela entrega. Frequentemente, essa definição é feita de forma manual, tornando esse processo muito demorado e desgastante (STABELINE, 2019). Desta forma, a partir da utilização de um sistema de roteirização, consegue-se um aumento da eficiência de entrega e, consequentemente, uma redução nos custos deste serviço (STABELINE, 2019).

Um sistema de roteirização é responsável pela definição de forma automática do percurso e das rotas a serem seguidas pelos veículos de entrega. Esse tipo de sistema gera a rota a ser percorrida pelo veículo de acordo com as entregas que devem ser efetuadas. Este tipo de problema pode ser modelado utilizando grafos, de modo que os vértices representam os pontos de entrega e as arestas representam os caminhos entre os clientes, utilizando o problema do caixeiro viajante para a sua solução (GOLDBARG, 2012).

Para resolver o problema proposto foi utilizada uma variação do problema do caixeiro viajante, o problema do caixeiro viajante com janelas de tempo (REBOUÇAS, 2016). Nesta variação do problema são definidos o horário inicial e final da entrega, assim como o tempo de entrega para cada um dos clientes. Desse modo os clientes receberão a entrega dentro do horário em que se encontram disponíveis.

O problema do Caixeiro Viajante com janelas de tempo é um problema NP-Difícil, apresentando uma alto custo computacional para a solução do problema para grafos com um grande número de vértices. Devido a alta complexidade do problema e, consequentemente, alto tempo de execução, neste trabalho foi desenvolvida uma solução paralela para o problema do caixeiro viajante com janelas de tempo, de forma a aproveitar o paralelismo de arquiteturas MIMD (*Multiple Instruction, Multiple Data*) (FLYNN, 1972).

A implementação foi desenvolvida utilizando a heurística GENIUS e definindo um intervalo (horário inicial e final) para cada um dos pontos de entrega. A paralelização foi realizada utilizando a biblioteca de *threads* OpenMP (*Open Multi-Processing*) (QUINN, 2003), de forma a explorar o paralelismo em computadores com múltiplos núcleos de processamento, também conhecidos como *multicore*.

1.1 OBJETIVOS DO TRABALHO

O objetivo geral do trabalho foi desenvolver uma implementação paralela para o problema da roteirização, encontrando o melhor caminho entre todos os locais de entrega, passando obrigatoriamente uma vez por cada local, partindo e retornando do mesmo depósito e obedecendo aos horários de entrega. Os seguintes objetivos específicos foram realizados para que o objetivo final fosse atingido:

- Definição da heurística a ser utilizada no desenvolvimento do sistema de roteirização.
- Implementação de uma versão sequencial do sistema de roteirização.
- Paralelização do sistema utilizando a biblioteca de *threads* OpenMP.
- Testes comparativos entre o tempo de execução da versão sequencial e da versão paralela.

1.2 ESTRUTURA DO TRABALHO

O presente trabalho está organizado da seguinte forma:

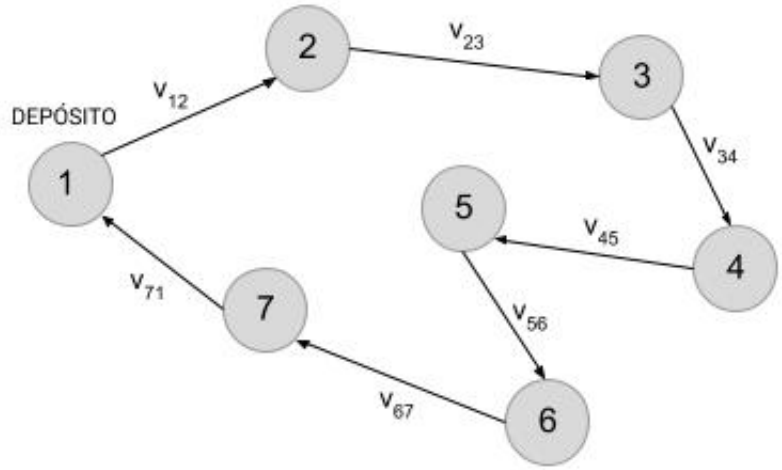
- Neste primeiro capítulo foi apresentado o contexto deste trabalho, as suas motivações e os seus principais objetivos.
- No Capítulo 2 é apresentado o problema do caixeiro viajante, e o problema do caixeiro viajante com janelas de tempo. Por fim, foram apresentadas três heurística utilizadas para a solução do problema do caixeiro viajante com janelas de tempo.
- No Capítulo 3 é apresentado o algoritmo GENIUS, que será utilizado para o desenvolvimento deste trabalho.
- No Capítulo 4 é apresentada uma breve introdução sobre arquiteturas paralelas. Após, é apresentada a técnica que será utilizada na paralelização do problema do caixeiro viajante.
- No Capítulo 5 é apresentado o desenvolvimento da aplicação, descrevendo as principais etapas e estruturas de dados utilizadas. Além disso, são apresentados os resultados obtidos.
- No Capítulo 6 são apresentadas as considerações finais do trabalho e sugestões de trabalhos futuros.

2 PROBLEMA DO CAIXEIRO VIAJANTE

Considerando o percurso de um comercial, que necessita visitar diversos clientes em cidades onde atua. Esse representante busca realizar o caminho mais curto de forma a visitar todas as cidades de uma única vez e retornando a sua cidade de origem minimizando as despesas de suas viagens. Esse problema pode ser modelado como o problema do caixeiro viajante (PCV) (BOAVENTURA; JURKIEWICZ, 2019).

O problema do caixeiro viajante é um problema clássico da computação, sendo um dos problemas de análise combinatória mais pesquisados e possuindo diversas aplicações práticas. Esse consiste em encontrar um ciclo hamiltoniano [1] de custo mínimo em um grafo G = (N, M), onde N = 1, ..., n é o conjunto de vértices representando as cidades que precisam ser visitados, e M = 1, . . . , m é o conjunto de arestas que representam as distâncias entre as cidades. Desta forma, o problema do caixeiro viajante consiste basicamente em determinar a menor rota a ser percorrida de forma que todos os vértices do grafo sejam visitados uma única vez e retornando ao vértice de origem, como pode ser visualizado na Figura 1, onde a rota inicia-se no depósito e retorna ao mesmo após passar uma única vez por todos os vértices (GOLDBARG, 2012).

Figura 1 – Representação do problema do caixeiro viajante



Fonte: do autor

[1] Um ciclo hamiltoniano é um caminho que permite visitar todos os vértices de um grafo passando obrigatoriamente uma única vez por cada um dos vértices (GOLDBARG, 2012)

O PCV encontra-se na classe dos problemas NP-Difícil, apresentando uma complexidade fatorial em relação ao número de vértices (GOLDBARG, 2012). Assim, em grafos com um grande número de vértices, torna-se inviável a utilização de uma técnica de força bruta [2]. A utilização de uma técnica de força bruta permitiria obter a solução ótima do problema, porém a sua utilização é impraticável devido ao elevado número de caminhos a serem testados, no caso do caixeiro viajante são $(n-1)!$ caminhos possíveis (NETTO; JURKIEWICZ, 2017). Na Tabela 1 são apresentados alguns exemplos com o números de caminhos a serem testados em função do número de cidades.

Tabela 1 – Nª de Caminhos vs Nº de Cidades

Nº de Cidades	Nº de Caminhos Possíveis
5	12
10	181.440
25	3.1022×10^{23}
50	3.0114×10^{62}
100	4.6663×10^{155}

Fonte: do autor

Desde modo, torna-se inviável utilizar uma técnica de força bruta para obter a solução exata do problema em instâncias com um grande número de cidades, mesmo que sejam utilizadas técnicas de paralelização. De forma a resolver este problema podem ser utilizadas heurísticas, que reduzem significativamente o tempo de execução, porém não garantem alcançar a solução ótima (NETTO; JURKIEWICZ, 2017).

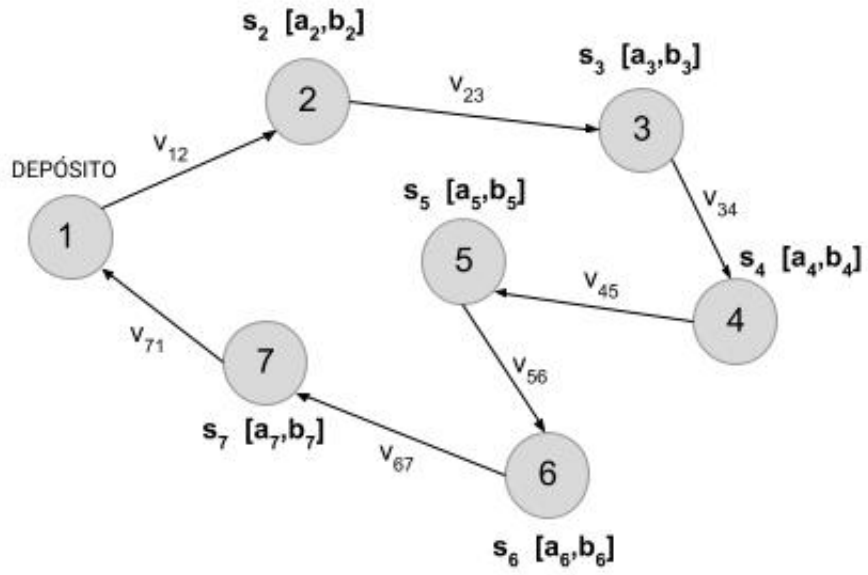
2.1 CAIXEIRO VIAJANTE COM JANELAS DE TEMPO

Existem algumas variações do problema do caixeiro viajante para aplicações práticas, como por exemplo, caixeiro viajante com janelas de tempo (PCVJT). O PCVJT consiste em definir para cada vértice do grafo um intervalo de tempo a ser atingido, ou seja, cada um dos clientes (vértices) terá um intervalo de tempo com o horário inicial de entrega e o horário final de entrega, sendo que a tarefa do PCVJT consiste em encontrar uma rota que obedeça estes intervalos de tempo. Essa variante do algoritmo pode ser utilizada para problemas práticos, como a roteirização de veículos (REBOUÇAS, 2016).

[2] Um algoritmo de busca por força bruta realiza a análise de todas as possíveis soluções de um determinado problema, encontrando assim a solução ótima se ela existir (NETTO; JURKIEWICZ, 2017)

No caso roteirização de veículos tem-se que um veículo irá deixar o depósito em determinado horário possuindo uma série de entregas a serem feitas e retornando ao depósito após concluí-las. Neste caso, os vértices do grafo representam os clientes e para cada vértice é definindo um intervalo de tempo preestabelecido para ser atingido. As arestas do grafo representam a distância entre os clientes e o tempo de deslocamento. Ou seja, o PCVJT consiste no problema do caixeiro viajante com a adição de restrições de horários de atendimento para cada cliente (REBOUÇAS, 2016). Na Figura 2 tem-se uma representação de como esse problema pode ser modelado em forma de um grafo onde: v_{ij} corresponde ao tempo de deslocamento entre o cliente *i* e o cliente *j*; s_i corresponde ao tempo de atendimento do cliente *i*; e a_i e b_i correspondem ao tempo mínimo e máximo, respectivamente, para o atendimento do cliente *i* (REBOUÇAS, 2016).

Figura 2 – Representação do problema do caixeiro viajante com janelas de tempo



Fonte: do autor

2.2 HEURÍSTICAS PARA A SOLUÇÃO DO PROBLEMA DO PCVJT

Um algoritmo normalmente é construído de forma a encontrar a solução ótima para um determinado problema. Porém, no caso de algoritmos com complexidade elevada, como por exemplo o PCVJT, o tempo de execução seria extremamente elevado, tornando a sua execução inviável. Nestes casos podem ser utilizadas heurísticas que apresentam um tempo menor de execução, e que fornecem soluções boas para o problema, mas não necessariamente a solução ótima (PEARL, 1984). Em geral heurísticas para o PCVJT podem ser divididas em duas categorias (REBOUÇAS, 2016):

- Heurísticas de Construção: nesta categoria as soluções são obtidas com a construção de um novo grafo, inserindo as arestas de forma a encontrar uma solução viável. Como exemplo de heurísticas que utilizam esse técnica pode-se citar o método guloso (REBOUÇAS, 2016).

- Heurísticas de Melhoramento: nesta categoria as soluções são obtidas manipulando o grafo inserindo, removendo ou trocando arestas de forma a encontrar uma solução viável. Como exemplo de heurísticas dessa categoria pode-se citar as heurísticas de busca local (REBOUÇAS, 2016).

2.2.1 Principais Heurísticas de Solução

Nesta seção foi realizada uma análise das principais heurísticas utilizadas para o desenvolvimento do Problema do Caixeiro Viajante com Janelas de Tempo (PCVJT). Mais especificamente serão apresentados três diferentes métodos: GENIUS, Método da Busca em Vizinhança Variável (VNS do inglês *Variable Neighborhood Search*) e Algoritmos Genéticos.

2.2.1.1 Heurística GENIUS

A heurística GENIUS foi proposta inicialmente por Gendreau em 1992, sendo utilizada originalmente para a solução do problema do caixeiro viajante. Ela se divide em duas fases: a fase GENI (*Generalized Insertion*), de construção, baseada em métodos de inserção e a fase US (*Unstringing and Stringing*), de melhoramento, onde busca-se uma solução melhorada a partir de uma solução pré-existente (GENDREAU; HERTZ; LAPORTE, 1992). O pseudo-código da heurística GENIUS pode ser visualizado no Algoritmo 1.

Algoritmo 1: Pseudo-Código da Heurística GENIUS

Input: V
Output: s
$s_0 \leftarrow GENI(V)$;
$s \leftarrow US(s_o)$;

Na fase GENI um novo vértice v é inserido entre dois vértices já pertencentes a rota s em cada iteração. A sua construção inicia a partir de três vértices escolhidos aleatoriamente e o novo vértice é inserido no ponto da rota com menor custo de inserção. O custo de inserção é calculado a partir da soma da distância entre os vértices da rota. A inserção de novos vértices ocorre até que todos os vértices do conjunto V tenham sido inseridos na rota (SILVA, 2012).

Já na segunda fase, a fase US, a solução gerada pela etapa GENI é melhorada. Para isso, em cada iteração, um vértice pertencente a rota é removido e re-inserido na rota por meio do método de inserção da fase GENI, caso a nova rota tenha um custo menor que a rota atual ela é substituída. O algoritmo é finalizado após percorrer todos os vértices da rota sem que haja uma melhoria no custo da rota gerada (SILVA, 2012).

Em 1998, Gendreau propôs uma versão da heurística para sua utilização com o Problema do Caixeiro Viajante com Janelas de Tempo, sendo necessárias, para isto, algumas modificações (GENDREAU *et al.*, 1998). Diferentemente do PCV onde em caso de grafos não direcionados a solução inversa também é possível, no PCVJT o grafo necessita ser direcionado, visto que a solução inversa poderia ocasionar em uma violação das janelas de tempo. Assim, na fase GENI os vértices não são inseridos de forma aleatória e sim de acordo com o comprimento de suas janelas de tempo, sendo que os vértices com menor comprimento da janela de tempo são inseridos inicialmente (REBOUÇAS, 2016).

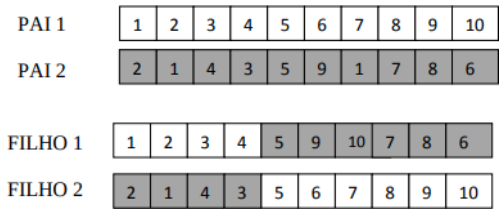
2.2.1.2 Algoritmo Genético

Os algoritmos genéticos são métodos evolutivos de otimização que baseiam-se no princípio da seleção natural proposto por Charles Darwin na qual somente os indivíduos mais aptos sobrevivem para uma próxima geração. Esses algoritmos utilizam dos princípios da teoria da genética populacional, onde a população é submetida a três operadores: seleção, *crossover* (ou cruzamento) e mutação (REINA, 2012).

Um algoritmo genético inicia-se com uma população de N indivíduos, onde cada indivíduo é representado por um cromossomo que representa uma possível solução para o problema. A estrutura definida para o cromossomo deve ser capaz de representar todas as possíveis soluções do problema (SILVA, 2016).

A cada iteração uma nova geração é criada, sendo que os melhores indivíduos são selecionados para o cruzamento. Para a seleção dos melhores indivíduos, devem ser definidos os critérios para a avaliação dos indivíduos. No caso do PCVJT o critério a ser considerado é a rota de menor custo e que obedece as janelas de tempo do problema (SILVA, 2016). No cruzamento ocorre a combinação entre esses indivíduos, permitindo que as melhores características continuem presentes nas próximas gerações. Na Figura 3 pode ser visualizado o processo de geração de dois novos indivíduos, que são gerados a partir da combinação de dois indivíduos da geração atual.

Figura 3 – Crossover em Algoritmos Genéticos



Fonte: SILVA (2016)

Após o cruzamento, aplica-se a mutação sobre um ou mais indivíduos de forma a trazer uma maior diversidade para a população (SILVA, 2016). Neste caso, um ou mais genes são selecionados aleatoriamente do cromossomo e são modificados, passando a representar uma nova solução para o problema, garantindo assim uma maior diversidade da população e evitando que o algoritmo venha a convergir prematuramente em um mínimo local (SILVA, 2016).

No caso do problema do caixeiro viajante com janelas de tempo o algoritmo é encerrado ao executar uma quantidade de gerações definida ou então ao convergir para um resultado, ou seja, quando não é possível obter uma solução com um menor custo (SILVA, 2016).

2.2.1.3 Método da Busca em Vizinhança Variável

O método da busca em vizinhança variável foi proposto por Mladenovic e Hansen em 1997 (MLADENOVIR, 1997), sendo utilizado para resolver problemas de otimização combinatória por meio da troca de vizinhanças combinada com uma busca local (REBOUÇAS, 2016).

A troca de vizinhança possibilita a geração de soluções mais eficientes uma vez que um mínimo local de uma vizinhança pode ter valor melhor que um mínimo local das outras. Dessa forma, aumentam-se as chances de encontrar um mínimo global e se aproximar da solução do problema (REBOUÇAS, 2016).

No VNS, parte-se de uma solução inicial e a cada iteração é realizada a seleção de uma vizinhança s' pertencente a solução atual s, sendo que esta vizinhança é submetida a um procedimento de busca local. No procedimento de busca local, a vizinhança s' é perturbada[3] um número determinado de vezes procurando melhorar a solução. Caso a solução gerada seja melhor que a atual ela é substituída. Uma representação do pseudo-código da heurística VNS pode ser visualizada no Algoritmo 2.

Algoritmo 2: Pseudo-Código da Heurística VNS

Input: V
Output: s
$n \leftarrow$ Número de vizinhanças;
$k \leftarrow 1$;
while $k <= n$ **do**
  selecione um vizinho s' na vizinhança de k;
  aplique o método de busca local em s' obtendo um ótimo local s'';
  if $s \leftarrow s''$;
  $k \leftarrow 1$;
  then s'' melhor que s
  $k \leftarrow k + 1$;

[3] A perturbação ocorre por meio da troca de vizinhanças de forma a embaralhar a solução atual em busca de uma melhor solução. A perturbação pode ser tendenciosa ou aleatória.

2.3 DEFINIÇÃO DA HEURÍSTICA

Para o desenvolvimento deste trabalho optou-se pela utilização da heurística GENIUS, que foi originalmente proposta por GENDREAU; HERTZ; LAPORTE (1992) e, posteriormente, alterada de forma a trabalhar com a inserção de janelas de tempo por GENDREAU *et al.*. Optou-se pela utilização da heurística GENIUS, pois essa é uma das heurísticas mais utilizadas na literatura para a resolução do Problema do Caixeiro Viajante, sendo uma das que apresenta melhor desempenho para a resolução do PCV (MLADENOVIR, 1997).

A heurística GENIUS em sua forma original ou com pequenas adaptações, possui grande utilização em problemas práticos relacionados a resolução do Problema do Caixeiro Viajante com Janelas de Tempo. De fato, como exemplo da utilização da heurística GENIUS em problemas de PCVJT pode-se citar os trabalhos desenvolvidos por REBOUÇAS (2016), GENDREAU *et al.* (1998) e PESANT; GENDREAU; ROUSSEAU (2006)

No trabalho desenvolvido GENDREAU *et al.* (1998) um algoritmo baseado na heurística GENIUS é utilizado para a solução do problema do roteamento de veículos com janelas de tempo. Nesse é utilizada a fase construtiva GENI juntamente com a operação de retrocesso para gerar um caminho inicial.

Já no trabalho desenvolvido por REBOUÇAS (2016) a heurística GENIUS é utilizada para resolver o problema de roteamento de veículos com coleta de prêmios e janelas de tempo. Além da heurística GENIUS, foi utilizada uma meta-heurística VNS (*Variable Neighborhood Search*) (MLADENOVIR, 1997) para o refinamento da solução.

Por fim, no trabalho proposto por PESANT; GENDREAU; ROUSSEAU (2006) é apresentado um algoritmo que utiliza a heurística GENIUS, juntamente com um modelo de programação por restrições, para resolver o problema do roteamento de veículos, utilizando restrições para janelas de tempo.

No próximo capítulo é realizada uma apresentação mais detalhada da heurística GENIUS, bem com a utilização dessa para a solução do Problema do Caixeiro Viajante com Janelas de Tempo.

3 HEURÍSTICA GENIUS

A heurística GENIUS se divide em duas fases: a fase GENI (*Generalized Insertion*), que é uma fase construtiva e que baseia-se nos métodos de inserção; e a fase US (*Unstringing and Stringing*), que é uma fase de melhoramento onde busca-se melhorar a solução já existente através da troca de arestas (GENDREAU; HERTZ; LAPORTE, 1992). A solução do problema do PCV, utilizando o a heurística GENIUS, pode ser modelada considerando as seguintes variáveis:

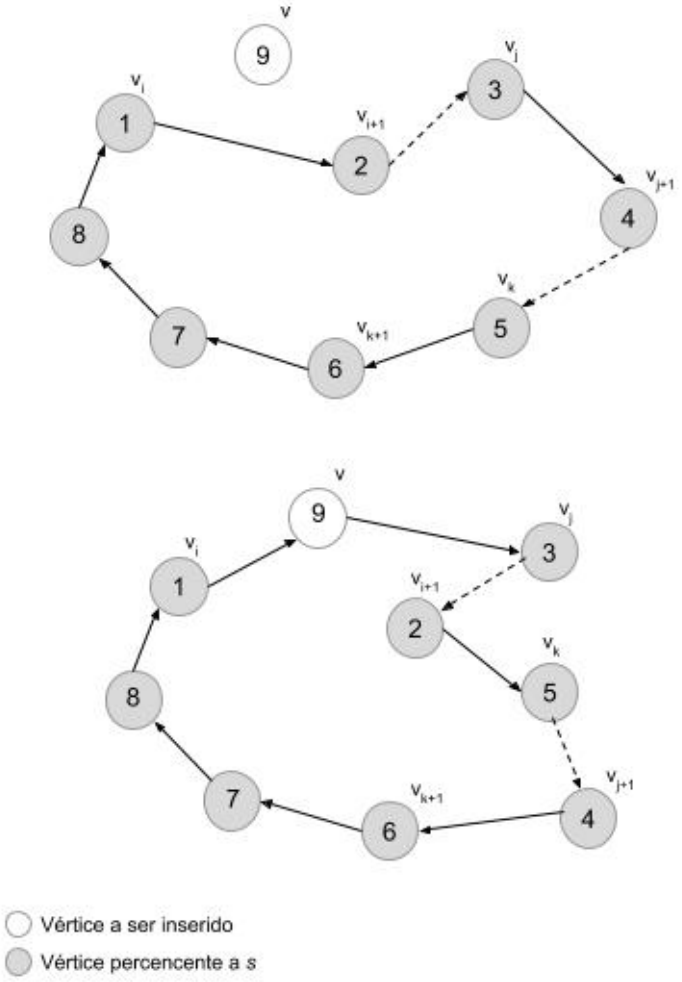
- V: conjunto com todos os vértices;
- V^+: conjunto dos vértices já inseridos;
- V^-: conjunto dos vértices não inseridos;
- *v*: vértice a ser inserido ou removido;
- v_k: vértice presente no caminho entre v_j e v_i;
- v_l: vértice presente no caminho entre v_i e v_j;
- v_{h-1}: vértice que antecede o vértice v_h, considerando o sentido do caminho;
- v_{h+1}: vértice que sucede o vértice v_h, considerando o sentido do caminho;
- $N_p(v)$: vizinhança do vértice v, formada pelos p vértices mais próximos de v.
- *s*: solução completa ou parcial do problema.
- $\bar{s}$: solução completa ou parcial do problema no sentido inverso.

3.1 FASE GENI - *GENERALIZED INSERTION*

Na fase de construtiva GENI, um novo vértice *v* é inserido entre dois vértices v_i e v_j a cada iteração. Os vértices v_i e v_j, não precisam ser necessariamente adjacentes entre si, mas tornam-se adjacentes a *v* após a inserção da aresta (SILVA, 2012). Existem duas formas de inserção pelo algoritmo GENI: a inserção Tipo I e a inserção Tipo II.

Na inserção de Tipo I, como pode ser observado na Figura 4, um vértice $v \in V^-$ é adicionado a rota *s* por meio da remoção dos arcos (v_i, v_{i+1}), (v_j, v_{j+1}), (v_k, v_{k+1}) e inserção dos arcos (v_i, v), (v, v_j), (v_{i+1}, v_k) e (v_{j+1}, v_{k+1}). O vértice v_k deve ser um vértice pertencente a V^+ e inserido no caminho entre v_j e v_i. Para isto é necessário que exista pelo menos um vértice entre v_j e v_i. Após a inserção do vértice, os caminhos (v_{i+1}, ... v_j) e (v_{j+1}, v_k) tem seus sentidos invertidos (SILVA, 2012).

Figura 4 – Representação da Inserção Tipo I da Heurística GENI



Fonte: do autor

No Algoritmo 3, tem-se o pseudocódigo da função para uma inserção do Tipo I.

Algoritmo 3: GENI - Inserção Tipo I

Input: s, v, v_i, v_j e v_k

Output: s'

s' $\leftarrow$ *s*;

if $v_k \neq v_i$ *and* $v_k \neq v_j$ **then**

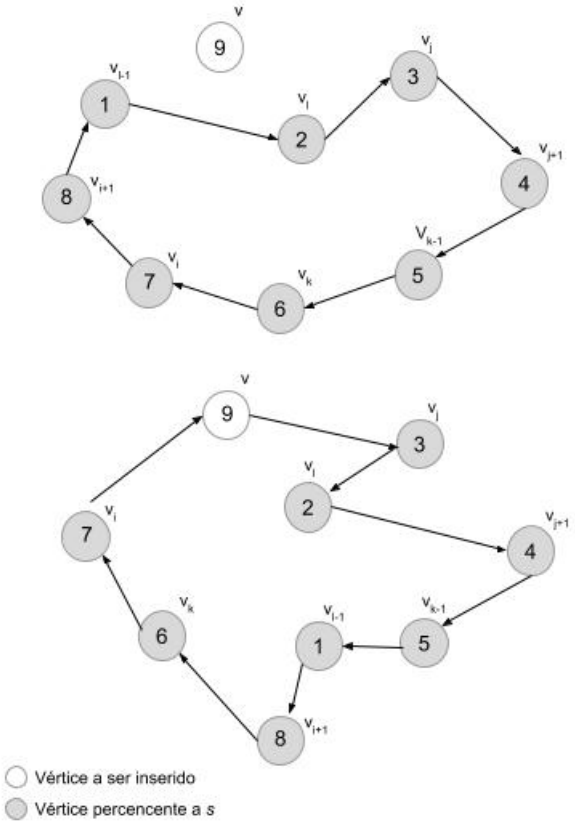
    remova os arcos (v_i, v_{i+1}), (v_j, v_{j+1}) e (v_k, v_{k+1}) de *s'*;

    insira os arcos (v_i, v), (v, v_j), (v_{i+1}, v_k) e (v_{j+1}, v_{k+1}) em *s'*;

    inverta o sentido dos caminhos entre $(v_{i+1}, ... v_j)$ e (v_{j+1}, v_k);

Na Figura 5 tem-se um exemplo de uma inserção do Tipo II. Nessa um vértice $v \in V^-$ é adicionado a rota *s* por meio da remoção dos arcos (v_i, v_{i+1}), (v_j, v_{j+1}), (v_{k-1}, v_k), (v_{l-1}, v_l) e inserção dos arcos (v_i, v), (v, v_j), (v_l, v_{j+1}), (v_{k-1}, v_{l-1}) e (v_{i+1}, v_k). O vértice v_k deve ser um vértice pertencente a V^+ e inserido no caminho entre v_j e v_i, e o vértice v_l deve ser um vértice pertencente a V^+ e inserido no caminho entre v_i e v_j. Para isto é necessário que exista pelo menos um vértice entre v_j e v_i e um vértice entre v_i e v_j. Após a inserção do vértice os caminhos $(v_{i+1}, ..., v_{l-1})$ e (v_l, v_j) tem seus sentidos invertidos (SILVA, 2012).

Figura 5 – Representação da Inserção Tipo II da Heurística GENI



Fonte: do autor

No Algoritmo 4, tem-se o pseudocódigo da função para uma inserção do Tipo II.

Algoritmo 4: GENI - Inserção Tipo II

Input: s, v, v_i, v_j, v_k e v_l
Output: s'
s' $\leftarrow$ *s*;
if $v_k \neq v_j$ *and* $v_k \neq v_{j+1}$ *and* $v_l \neq v_i$ *and* $v_l \neq v_{i+1}$ **then**
    remova os arcos (v_i, v_{i+1}), (v_j, v_{j+1}), (v_{k-1}, v_k) e (v_{l-1}, v_l) de *s'*;
    insira os arcos (v_i, v), (v, v_j), (v_l, v_{j+1}), (v_{k-1}, v_{l-1}) e (v_{i+1}, v_k) em *s'*;
    inverta o sentido dos caminhos entre $(v_{i+1}, ..., v_{l-1})$ e (v_l, v_j);

A heurística GENI é iniciada com a construção de uma subrota s contendo três vértices selecionados aleatoriamente pertencentes a V^-. Um novo vértice é inserido a cada iteração por meio das inserções de Tipo I e II e levando em consideração os dois sentidos da rota s e $\bar{s}$. Neste caso, considera-se todas as combinações possíveis de v_i e $v_j \in N_p(v)$, $v_k \in N_p(v_{i+1})$ e $v_l \in N_p(v_{j+1})$, dessa forma, o vértice v é inserido considerando a combinação com menor custo de inserção. O custo de inserção $f(s)$ é calculado pela soma das distâncias entre os vértices de s. No Algortimo 5 tem-se o pseudocódigo da inserção GENI.

Algoritmo 5: Inserção GENI

Input: s, v
Output: $s*$
for $s' \in \{s, \bar{s}\}$ **do**
  for v_i, $v_j \in N_p(v)$ **do**
    for $v_k \in N_p(v_{i+1})$ **do**
      $s'' \leftarrow$ *InsercaoTipoI*(s', v, v_i, v_j, v_k);
      if $f(s'') < f(s*)$ **then**
        $s* \leftarrow s''$;
      for $v_l \in N_p(v_{j+1})$ **do**
        $s'' \leftarrow$ *InsercaoTipoII*(s', v, v_i, v_j, v_k, v_l);
        if $f(s'') < f(s*)$ **then**
          $s* \leftarrow s''$;

Como pode ser observado no pseudocódigo do Algoritmo 6, a inserção de novos vértices ocorre até que todos os vértices do conjunto V^- tenham sido inseridos na rota s, ou seja, $V^+ = V$ (SILVA, 2012).

Algoritmo 6: Heurística GENI

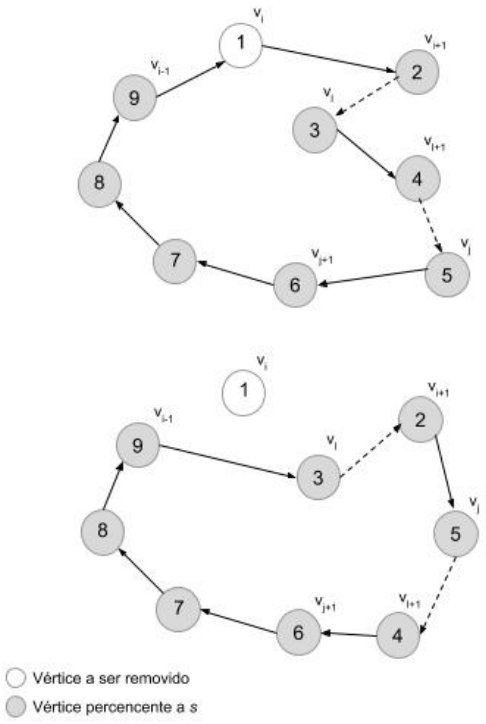
Input: V
Output: s
$s \leftarrow$ Ø; $V^- \leftarrow V$;
while $|V^-| > 0$ **do**
  selecione aleatoriamente um vértice $v \in V^-$;
  $s \leftarrow$ *InsercaoGENI*(s,v);
  $V^- = V^- \setminus \{v\}$;

3.2 FASE US - *UNSTRINGING AND STRINGING*

Na fase de melhoramento US, um vértice é removido e re-adicionado em outra posição da rota a cada iteração. Existem duas formas de remoção de vértices pelo algoritmo US: a remoção Tipo I e a remoção Tipo II. Após a remoção do vértice a re-inserção é realizada pelas duas formas de inserção da fase GENI (SILVA, 2012).

Como pode ser observado na Figura 6, na remoção de Tipo I um vértice $v_i \epsilon V^+$ é removido por meio da remoção dos arcos (v_{i-1}, v_i), (v_i, v_{i+1}), (v_j, v_{j+1}), (v_l, v_{l+1}) e inserção dos arcos (v_{i-1}, v_l), (v_{i+1}, v_j), (v_{l+1}, v_{j+1}). Após a remoção os caminhos (v_{i+1}, ... v_l) e (v_{l+1}, v_j) tem seus sentidos invertidos (SILVA, 2012).

Figura 6 – Representação da Remoção Tipo I da fase US



Fonte: do autor

No Algoritmo 7 tem-se o pseudocódigo é de uma remoção do Tipo I.

Algoritmo 7: US - Remoção Tipo I

Input: s, v_i, v_j e v_k

Output: s'

s' $\leftarrow$ *s*;

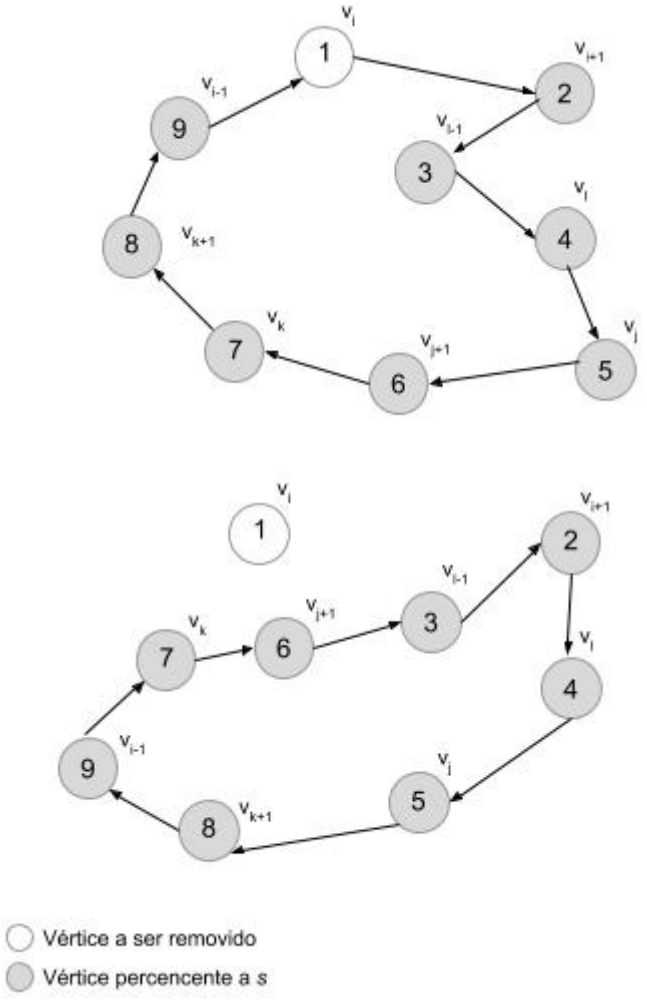
remova os arcos (v_{i-1}, v_i), (v_i, v_{i+1}), (v_j, v_{j+1}) e (v_l, v_{l+1}) de *s'*;

insira os arcos (v_{i-1}, v_l), (v_{i+1}, v_j), (v_{l+1}, v_{j+1}) em *s'*;

inverta o sentido dos caminhos entre (v_{i+1}, ... v_l) e (v_{l+1}, v_j);

Como pode ser observado na Figura 7, na remoção de Tipo II um vértice $v_i \in V^+$ é removido por meio da remoção dos arcos (v_{i-1}, v_i), (v_i, v_{i+1}), (v_j, v_{j+1}), (v_{l-1}, v_l), (v_k, v_{k+1}) e inserção dos arcos (v_{i-1}, v_k), (v_{j+1}, v_{l-1}), (v_{i+1}, v_l), (v_j, v_{k+1}). Após a remoção os caminhos $(v_j$, ... $v_k)$, (v_{i+1}, v_l) tem seus sentidos invertidos (SILVA, 2012).

Figura 7 – Representação da Remoção Tipo II da fase US



Fonte: do autor

No Algoritmo 8 tem-se o pseudocódigo da função de remoção do tipo II.

Algoritmo 8: US - Remoção Tipo II

Input: s, v, v_i, v_j, v_k e v_l

Output: s'

s' $\leftarrow$ *s*;

remova os arcos (v_{i-1}, v_i), (v_i, v_{i+1}), (v_j, v_{j+1}), (v_{l-1}, v_l) e (v_k, v_{k+1}) de *s'*;

insira os arcos (v_{i-1}, v_k), (v_{j+1}, v_{l-1}), (v_{i+1}, v_l) e (v_j, v_{k+1}) em *s'*;

inverta o sentido dos caminhos entre $(v_j$, ... $v_k)$ e (v_{i+1}, v_l);

A heurística US tem início a partir de uma rota gerado pela fase GENI e todos os vértices da rota s são percorridos. A cada iteração um vértice é removido e reinserido na rota por meio das remoções de Tipo I e II (SILVA, 2012). O pseudocódigo é demonstrado no Algoritmo 9, p_1 representa o primeiro vértice da rota s e p_n o número de vértice da rota s, considere também $v(p, s)$, como sendo o vértice que se encontra na posição p da rota s.

Algoritmo 9: Heurística US

Input: s
Output: $s*$
$s* \leftarrow s$;
$p_z \leftarrow p_1$;
while $p_z \leq p_n$ **do**
    $v \leftarrow v(p_z, s)$;
    $s' \leftarrow RemocaoUS(v, s)$;
    $s'' \leftarrow InsercaoGENI(v, s')$;
    if *f(s") < f(s*)* **then**
        $s* \leftarrow s''$;
        $p_z \leftarrow p_1$;
    else
        $p_z \leftarrow p_z + 1$;

Já no Algoritmo 10 é demostrado o pseudocódigo da remoção de vértices.

Algoritmo 10: Remoção US

Input: v_i, s
Output: $s*$
$s* \leftarrow s$;
for $s' \in \{s, \bar{s}\}$ **do**
    for $v_k \in N_p(v_{i-1})$ **do**
        for $v_j \in N_p(v_{k+1})$ **do**
            $s'' \leftarrow$ *RemocaoTipoI*(s', v_i, v_j, v_k);
            if $f(s'') < f(s*)$ **then**
                $s* \leftarrow s''$;
            for $v_l \in N_p(v_{i+1})$ **do**
                $s'' \leftarrow$ *RemocaoTipoII*(s', v_i, v_j, v_k, v_l);
                if $f(s'') < f(s*)$ **then**
                    $s* \leftarrow s''$;

A heurística GENIUS é formada pela junção das duas fases GENI e US. Primeiramente, a fase GENI e executada e é realizada a construção da rota inicial. A rota resultante é refinada pela fase US, através da a remoção e reinserção de vértices com o intuito de melhorar a rota inicial (SILVA, 2012). O pseudocódigo é da heurística GENIUS é apresentada no Algoritmo 11.

Algoritmo 11: Heurística GENIUS

Input: V
Output: s
$s_0 \leftarrow GENI(V)$;
$s \leftarrow US(s_o)$;

3.3 GENIUS PARA PCVJT

A heurística GENIUS necessita de adaptações para o problema do PCVJT. Diferentemente do PCV, onde a rota inversa também é uma solução possível e possui mesmo tempo de entrega, no PCVJT a solução inversa pode acarretar na violação das janelas de tempo, não sendo uma solução factível. Desse modo, a solução do caminho deve ser uma solução direcionada, ou seja, no PCVJT a rota inicia em um vértice v_1 e finaliza em um vértice v_{n+1}, sendo que v_1 deve sempre ser o ponto de partida e v_{n+1} o ponto de chegada (REBOUÇAS, 2016).

Para resolver o problema da violação das janelas de tempo e produzir soluções factíveis substitui-se a vizinhança $N(p)$ da definição da heurística GENIUS por duas vizinhanças direcionadas (REBOUÇAS, 2016), onde:

- $N_p^+(v)$: vizinhos a direita de v, contendo os p vértices mais próximos de v já inseridos no caminho.
- $N_p^-(v)$: vizinhos a esquerda de v, contendo os p vértices mais próximos de v já inseridos no caminho.

No caso do PCVJT o processo de inserção de um vértice no caminho é mais restritivo, já que as janelas de tempo devem ser obedecidas e a inserção de um vértice v tem impacto em todos vizinhos pertencentes a $N_p^+(v)$. Devido ao processo mais restritivo de inserção, em alguns estágios do algoritmo, pode ser impossível a inserção de um novo vértice, podendo ser aplicado um retrocesso. Nos piores casos a inserção de um vértice pode ser impossível e o algoritmo é finalizado sem que alguns vértices sejam inseridos (REBOUÇAS, 2016).

No problema do PCV a definição de vizinhança é baseada somente na distância. No caso do PCVJT, o cálculo da vizinhança é realizado de forma a levar em consideração tanto as janelas de tempo, bem como a distância entre os vértices. Para isso é realizado o cálculo da proximidade entre as janelas de tempo, representado por r_{ij}, através da Equação 3.1 , onde: T_{ij} representa o tempo de percurso entre os vértices v_i e v_j, aqui representado como a distância entre os vértices

v_i e v_j; a_i que representa o início da janela de tempo de um vértice i; e i que corresponde ao fim da janela de tempo de um vértice v_i (REBOUÇAS, 2016).

$$r_{ij} = min\{b_j, b_i + T_{ij}\} - max\{a_j, a_i + T_{ij}\} \tag{3.1}$$

Para a definição da vizinhança $N_p^+(v)$, são definidos os p_1 sucessores de v já inseridos e estão mais próximos a T_{ij}, e os p_2 sucessores de v já inseridos e que estão mais próximos a r_{ij}. Assim, o valor p pode ser definido como sendo a soma de $p_1 + p_2$. Já para a vizinhança $N_p^-(v)$, são considerados os p_1 predecessores de v já inseridos e que estão mais próximos a T_{ij}; e os p_2 predecessores de v já inseridos e que estão mais próximos em relação a r_{ij}. Assim o valor de p pode ser definido como sendo a soma de $p_1 + p_2$ (REBOUÇAS, 2016).

A inserção de um novo vértice no caminho deve ser factível, para tanto é calculado o tempo crítico de partida z_i para cada um dos vértices já inseridos no caminho, sendo computados do último para o primeiro, iniciando em v_{n+1}. O valor de z_i corresponde ao maior tempo para qual se pode partir do vértice v_i sem que os próximos vértices do caminho sejam afetados. O cálculo de z pode ser realizado através de Equação 3.2.

$$\begin{cases} z_{n+1} = b_{n+1} \\ z_i = min\{b_i, z_{i+1} - T_{i,i+1}\} \end{cases} \tag{3.2}$$

A inserção de um novo vértice no caminho pode afetar a ordem de diversos vértices. Assim, considerando-se que após a inserção tem-se o caminho $(v_1, ..., v_a, ..., v_b, ..., v_{n+1})$, onde, $(v_a, ..., v_b)$ representa a parte modificada pela inserção, é necessário que para cada vértice do caminho o tempo de chegada t_i seja menor ou igual a b_i, e para cada vértice v_i após v_b a Equação 3.3 seja atendida.

$$t_{i+1} = t_i + T_{i,i+1} \leq z_{i+1} \tag{3.3}$$

No PCVJT os vértices não são inseridos aleatoriamente como acontece no PCV. Neste caso, são utilizados critérios de forma a ordenar os nodos em ordem decrescente de dificuldade de inserção. Os critérios mais utilizados são: ordem crescente do intervalo das janelas de tempo; ordem crescente de b_i; e ordem crescente de a_i.

De acordo com Gendreau (1998) o critério que melhor atende o problema é o que utiliza o intervalo das janelas de tempo. A cada inserção, pelo método GENI, o primeiro vértice inserido de forma factível é alocado na posição que gerar um menor custo para o caminho. Quando nenhum vértice pode ser inserido no caminho uma técnica de retrocesso é aplicada (REBOUÇAS, 2016).

A aplicação do retrocesso consiste em remover um vértice v_i do caminho, começando pelo segundo vértice do caminho e colocando-o no final da lista de vértices não inseridos. Após

a remoção do vértice, cada um dos vértices não inseridos e diferente do removido é inserido no caminho. Caso a inserção seja factível, ela é realizada e a heurística é retomada, Já no caso de nenhuma das inserções ser factível o processo de retrocesso é repetido para o próximo sucessor de v_i, até atingir o vértice final v_{n+1}. Um valor θ_i é incrementado a cada reinserção do vértice v_i na lista de vértices não inseridos, sendo que o processo é terminado após um limite superior θ ser atingido (REBOUÇAS, 2016).

Após concluída a fase GENI de inserções e um caminho factível ser encontrado, um procedimento de pós-otimização baseado em US é realizado. O processo é realizado repetidamente até o momento em que não ocorre mais nenhuma melhoria no caminho. Para sua utilização no PCVJT o algoritmo US também deve ser adaptado, respeitando sempre as janelas de tempo durante a inserção, ou seja, para cada vértice do percurso t_i deve ser menor ou igual à b_i.

No Algoritmo 12 tem-se o pseudocódigo da heurística GENIUS para PCVJT.

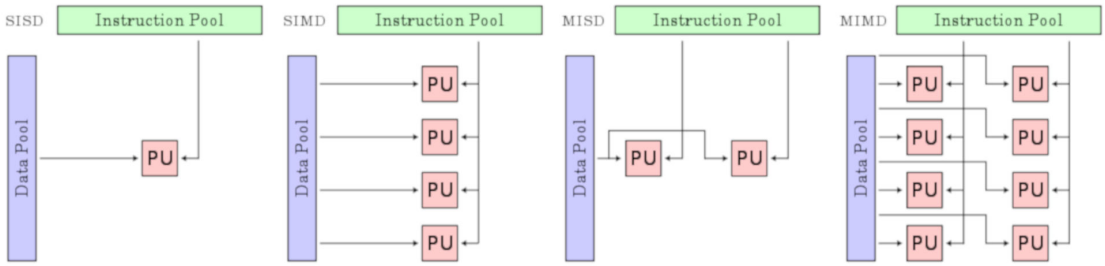
Algoritmo 12: Heurística GENIUS para PCVJT

Input: V, s
Output: $s*$
$S \leftarrow$ Todos os vértices de $V \backslash \{v_1, v_{n+1}\}$ ordenados pelo comprimento das janelas de tempo
$k \leftarrow 1$
$s \leftarrow |S|$
while $s \neq \varnothing$ **do**
  $v \leftarrow S(k)$
  $s* \leftarrow$ *InsercaoGENI*(s,v);
  if *não houve inserção ou inserção é inválida* **then**
    $k \leftarrow k + 1$
    if $k = s + 1$ **then**
      Remova o vértice v_i logo após o depósito
      Insira o vértice v_i no final da lista S
      $\theta_{vi} \leftarrow \theta_{vi} + 1$
      if $\theta_{vi} = \theta$ **then**
        Pare, não pôde ser identificada solução factível
      $k \leftarrow k + 1$
  $s \leftarrow s*$
  Atualize as vizinhanças de S
  $k \leftarrow 1$
  $s \leftarrow |S|$
$k \leftarrow 2$
while $k \leq n$ **do**
  $s* \leftarrow US(s_o)$
  if *Há solução melhor e solução é válida* **then**
    $s \leftarrow s*$
    $k \leftarrow 2$
  $k \leftarrow k + 1$

4 PROGRAMAÇÃO PARALELA

A paralelização é uma técnica que permite que múltiplas unidades de processamento trabalhem em conjunto de forma a resolver um problema. Ou seja, a programação paralela consiste em dividir um problema em tarefas menores de forma que essas possam ser executadas paralelamente e em unidades de processamento diferentes, diminuindo assim o tempo total de execução (FOSTER, 1995).

Figura 8 – Classificação segundo Flynn



Existem várias formas de organizar os processadores, e com base nisso, Flynn criou a Taxonomia de Flynn. De acordo com o fluxo de instruções e o fluxo de dados, as arquiteturas de computadores podem ser classificadas em quatro categorias (Figura 9) (FLYNN, 1972):

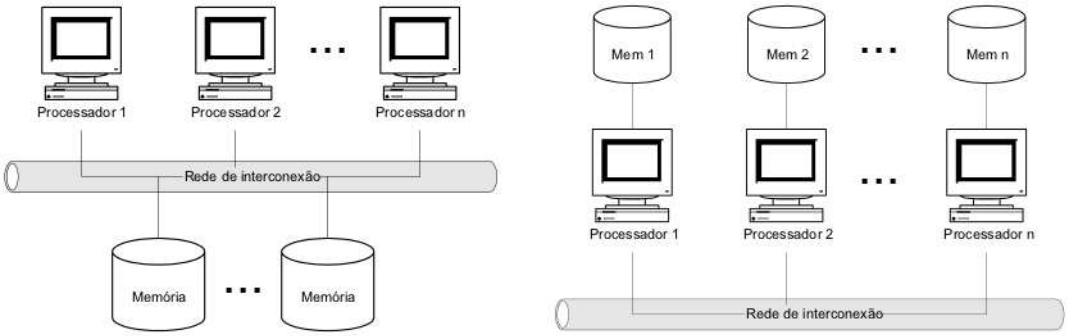
- **SISD** (*Single Instruction, Single Data*): um fluxo único de instruções sobre um único fluxo de dados. Neste caso não há paralelismo, sendo que nesta categoria encontram-se os computadores que possuem processadores com um único *core*.
- **SIMD** (*Single Instruction, Multiple Data*): um fluxo único de instruções sobre múltiplos fluxos de dados. Neste caso ocorre o paralelismo de dados. Um exemplo nesta categoria são os processadores vetoriais e matriciais, como as GPUs (*Graphics Processing Unit*, ou Unidade de Processamento Gráfico) [1].
- **MISD** (*Multiple Instruction, Single Data*): múltiplos fluxos de instruções sobre um único fluxo de dados. Neste caso ocorre o paralelismo de instruções, porém atualmente não existem arquiteturas que representem essa categoria.

[1] Unidade de processamento gráfico que trabalha com uma grande quantidade de dados em paralelo, não sendo adequadas para dados em série. Antigamente, as GPUs eram utilizadas principalmente para processamentos gráficos e de imagem, porém atualmente estão sendo utilizadas para a execução de aplicações em geral (TANENBAUM, 2010)

- **MIMD** (*Multiple Instruction, Multiple Data*): múltiplos fluxos de instruções sobre múltiplos fluxo de dados. Neste caso ocorre o paralelismo de dados e de instruções. Nesta categoria encontram-se a maioria dos computadores atuais que possuem múltiplos *cores*. Como exemplo de arquiteturas que encontram-se nesta categoria pode-se citar os super computadores, *clusters* de computadores [2] e computadores multiprocessados.

A categoria MIMD, pode ser subdividida ainda de acordo com a organização da memória, sendo que essa subdivisão, geralmente, é baseada no compartilhamento ou não da memória principal entre os processadores (FLYNN, 1972). Como pode ser observado na Figura 9 a categoria MIMD pode ser subdividida ainda em arquiteturas: com memória compartilhada e com memória distribuída.

Figura 9 – Memória Compartilhada x Memória Distribuída



Fonte: MIRANDA; PORSANI (2011)

4.1 ARQUITETURAS COM MEMÓRIA COMPARTILHADA

Em arquiteturas com memória compartilhada, a mesma área de memória é acessada por todos os processadores por meio de um barramento. Assim, os processadores podem realizar a troca de informações através dessa área de memória comum, possuindo um menor custo de comunicação. Porém, existem limites físicos já que o mesmo barramento é utilizado para a comunicação entre todos os processadores [3], assim ocorre uma disputa por esse barramento e esse torna-se um gargalo com o aumento no número de processadores. Como exemplo de arquiteturas que encontram-se nessa categoria pode-se citar os computadores multiprocessados e os computadores com processadores *multicore* (TANENBAUM, 2010).

[2] É uma arquitetura que tem por objetivo combinar diversos computadores de modo que eles trabalhem em conjunto, aumentando assim, a capacidade de processamento (TANENBAUM, 2010)

[3] Quando um ou mais processadores utilizam do mesmo barramento para comunicação acontece a chamada disputa de barramento. O barramento só pode ser utilizado por um único processador ao mesmo tempo. Em caso de aplicações com alto número de operações de acesso a memória o desempenho utilizando múltiplos processadores pode ser reduzido, já que um processador deverá aguardar a liberação do barramento para realizar a operação e continuar a execução.

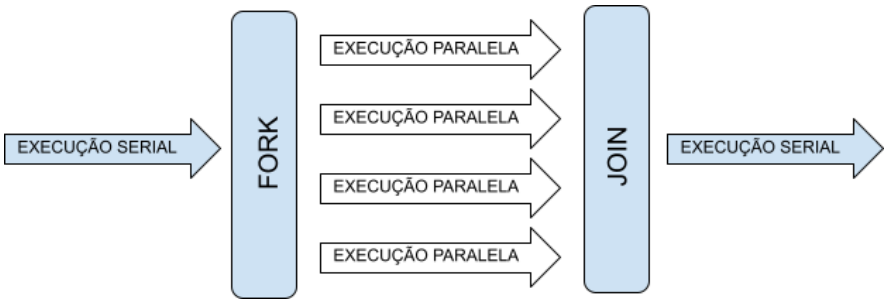
No caso de arquiteturas com memória compartilhada, uma das formas mais simples para desenvolvimento de programas paralelos é a utilização de *multithreading*. As *threads* possuem uma forma de funcionamento semelhante a de um processo, exceto pelo fato de todas as *threads* de um processo compartilharem do mesmo endereçamento de memória, facilitando assim a troca de informações entre as *threads*. Além disso, cada *thread* pode ser executada por um processador, desta forma, o processamento pode ser realizado de forma paralela. (TANENBAUM, 2010).

A maioria dos sistemas operacionais baseados em UNIX possuem uma biblioteca de *threads* chamada de *POSIX Threads*, onde são definidos padrões para a criação e a manipulação de *threads* utilizando a linguagem C. A biblioteca *POSIX Threads* possui a definição de mais de 100 funções, sendo elas para a criação, manipulação, sincronização, *mutex*, semáforos, entre outros (BUTENHOF, 1997).

Recentemente, a biblioteca OpenMP (*Open Multi-Processing*) vem sendo utilizada como uma padrão para desenvolvimento de aplicações paralelas para ambientes de memória compartilhada. Essa API (*Application Programming Interface*) permite a paralelização de aplicações por meio de diretivas de compilação e funções. Existem implementações desta biblioteca para sua utilização com C, C++ e Fortran.

O OpenMP baseia-se no modelo *fork/join*, a execução inicia sequencialmente pela *thread* principal executando as tarefas sequenciais. Nas partes onde devem ser executadas de forma paralela a *thread* principal cria *threads* adicionais para a execução. Após a conclusão da execução dessas *threads* ocorre o *join*, onde todas as *threads* são finalizadas e a execução continua na *thread* principal de forma sequencial (QUINN, 2003).

Figura 10 – Representação do modelo *fork/join*



Fonte: do autor

Neste trabalho será realizada uma implementação paralela da heurística do caixeiro viajante com janelas de tempo. Essa será desenvolvida de forma a explorar o paralelismo em computadores com processadores *multi-core*. Para esse desenvolvimento será utilizada a biblioteca *OpenMp* (QUINN, 2003). Optou-se pela utilização do OpenMp pois os computadores multi-cores são a arquitetura mais popular, presente em quase todos os computadores, sendo a biblioteca mais utilizada atualmente (MURAROLLI, 2015)

4.2 ARQUITETURAS COM MEMÓRIA DISTRIBUÍDA

Em arquiteturas com memória distribuída, cada processador possui uma memória exclusiva que não pode ser acessada por outros processadores de forma direta. A comunicação entre os processos é realizada por meio de troca de mensagens, sendo que este processo é mais lento quando comparado a sistemas com memória compartilhada. Um dos fatores que contribui para a redução da velocidade de comunicação é a latência de rede. A latência de rede é o tempo necessário para que uma mensagem chegue ao seu receptor, ou seja, o tempo para um pacote de dados ir de um ponto a outro. Além disso, a comunicação entre os processos fica a cargo do programador, tornando a programação neste tipo de arquitetura mais complexa (TANENBAUM, 2010).

Neste tipo de arquitetura não existem limites para a quantidade de processadores que podem ser utilizados, já que diferentemente da memória compartilhada, não existe um barramento compartilhado para o acesso a memória. Esta é uma das vantagens das arquiteturas de memória distribuída quando comparadas a arquiteturas de memória compartilhada. Um exemplo deste tipo de arquitetura são os *clusters* de computadores.

Para implementação de sistemas paralelos com memória distribuída são utilizadas diversas bibliotecas para a comunicação e sincronização entre os processos, sendo que entre essas bibliotecas pode-se destacar o PVM (*Parallel Virtual Machine*) e o MPI (*Message Passing Interface*).

O PVM é uma biblioteca que permite que uma rede de computadores possa ser programada de forma a criar uma única máquina virtual para a execução de aplicações em paralelo. O PVM utiliza processos Unix, dessa forma, cada tarefa corresponde a um processo. Cada computador da rede possui uma instância do PVM e a comunicação é realizada por meio de troca de mensagens (TANENBAUM, 2013).

O MPI é uma biblioteca que suporta diversas operações de comunicação para arquiteturas com memória distribuída. As funções disponíveis possibilitam o envio e o recebimento de mensagens entre processos, além de operações de comunicação coletiva com, por exemplo, barreiras, agregação, *broadcast* e espalha/reúne, etc.. As mensagens suportam diversos tipos de dados, como números inteiros, números decimais de ponto flutuante, caracteres, entre outros, além de tipos que derivam desses dados (TANENBAUM, 2013).

4.3 AVALIAÇÃO DE DESEMPENHO DE PROGRAMAS PARALELOS

A avaliação de desempenho de aplicações paralelas inclui diversos fatores, não se limitando ao tempo de execução. De fato, o desempenho de uma aplicação deve levar em consideração a escalabilidade, os mecanismos de geração, armazenamento e transmissão de dados, entre outros fatores (FOSTER, 1995).

A forma mais comum de avaliação de sistemas paralelos leva em consideração o tempo de execução. O tempo de execução é o intervalo entre o início da execução pelo primeiro processador e o momento da finalização da execução pelo último processador, sendo que esse pode ser decomposto em três partes (FOSTER, 1995):

- **Tempo de Computação**: é o tempo em que o processo está sendo efetivamente executado pelo processador. O tempo de execução depende do tamanho do problema e da arquitetura utilizada, variando de acordo com o número de processadores e sistema de memória utilizado (compartilhada ou distribuída). Desta forma, não se pode garantir que o tempo de execução continuará o mesmo com o aumento no número de processadores.

- **Tempo de Comunicação**: é o tempo em que uma tarefa passa efetuando a troca de informações. A comunicação entre processos pode ser dividida entre dois tipos: intra-processador ou inter-processadores. Na comunicação intra-processador, as tarefas estão sendo executadas em vários processadores que compartilham da mesma memória, sendo esta forma de comunicação é mais rápida. Já na comunicação inter-processadores o tempo é calculado pela soma do tempo de inicialização da comunicação e o tempo de transferência. O tempo de transferência é determinado pela largura de banda e pela latência do canal, podendo variar de acordo com a tecnologia de rede utilizada.

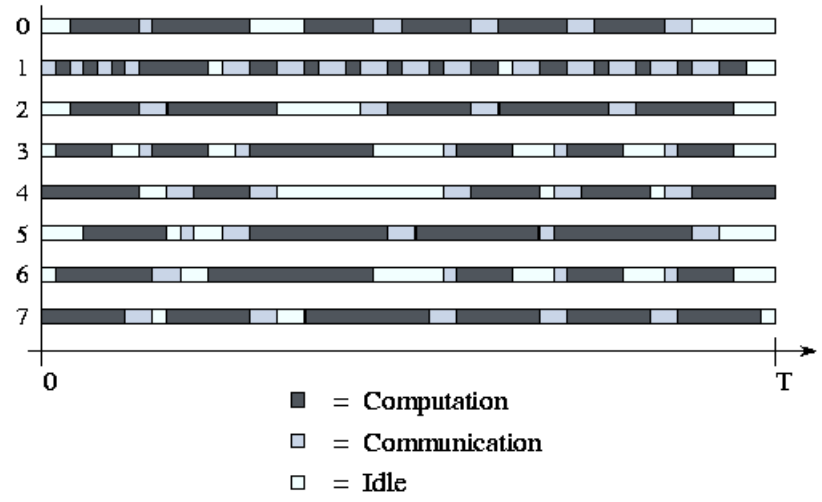
- **Tempo Ocioso**: é o tempo em que o processador não está realizando nenhum tipo de processamento. Uma das causas mais comuns são a espera de dados remotos que precisam ser comunicados. O tempo ocioso pode ser contornado com a realização de outras tarefas enquanto o processador está aguardando os dados.

Na Figura 11 tem-se uma representação gráfica de como é composto o tempo de execução de um programa paralelo. Desta forma, o tempo de execução pode ser calculado através da Equação 4.1.

$$T = T_{comp} + T_{comn} + T_{ocioso} \tag{4.1}$$

Para a avaliação do desempenho relativo ao tempo de execução da aplicação são utilizadas diversas métricas, entre elas as mais utilizadas são o *Speedup*, a Eficiência e a Escalabilidade, elas são descritas nas seções a seguir.

Figura 11 – Representação do tempo de execução



Fonte: FOSTER (1995)

Na Equação 4.2 tem-se a expressão para o cálculo do *Speedup*. Esse corresponde ao aumento de velocidade observado quando se executa um determinado processo em P processadores em relação à execução deste processo em 1 processador. Desta forma, tem-se que o *Speedup* é dado pela relação entre o tempo de execução sequencial (T_1), executando em somente um processador, e o tempo de execução em P processadores (T_p) (FOSTER, 1995). Assim, quanto mais próximo de P o valor do *Speedup*, maior é o desempenho do programa paralelo.

$$S = \frac{T_1}{T_p} \tag{4.2}$$

A eficiência é obtida através da Equação 4.3 e corresponde a relação entre o *speedup* (S) e o número de processadores (P). A eficiência corresponde ao grau de aproveitamento dos recursos computacionais disponíveis, sendo que quanto mais perto de 1 maior será a eficiência do programa paralelo. A eficiência máxima é 1, ou seja, 100% de eficiência (FOSTER, 1995).

$$E = \frac{S}{P} \tag{4.3}$$

Por fim, uma outra medida importante é a escalabilidade de uma aplicação paralela. Uma aplicação é dita escalável quando a eficiência se mantém constante independentemente do número de processadores utilizados (FOSTER, 1995). Assim, quando maior a eficiência e a sua variação com o aumento do número de processadores utilizados, maior a escalabilidade da aplicação.

5 IMPLEMENTAÇÃO E RESULTADOS OBTIDOS

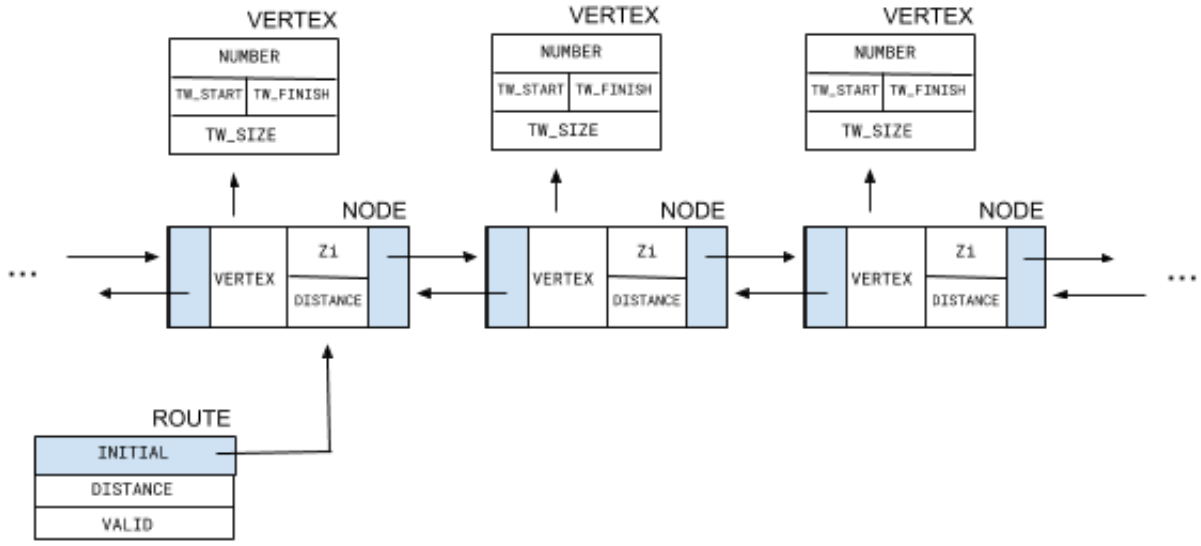
A heurística GENIUS foi implementada utilizando a linguagem de programação C e, após a aplicação foi paralelizada de forma a explorar o paralelismo em arquiteturas com processadores com múltiplos núcleos de processamento (*multicores*). Optou-se pela utilização desse tipo de arquitetura pois essa é a mais comum atualmente, estando presente na grande maioria dos computadores pessoais.

Para o desenvolvimento optou-se pela utilização da biblioteca de *threads OpenMp*, que é uma API baseada na utilização de diretivas de compilação e funções para o desenvolvimento de aplicações paralelas. Essa é a biblioteca de *threads* mais utilizada atualmente para o desenvolvimento de aplicações paralelas (MURAROLLI, 2015).

5.1 ESTRUTURAS DE DADOS UTILIZADAS

Para a representação dos grafos gerados foi utilizada uma lista duplamente encadeada para o armazenamento dos nodos, janelas de tempo e arcos. Cada nodo da lista possui referência ao seu vértice correspondente, a distância ao próximo vértice e seu tempo crítico de partida. De forma a representar uma rota a lista é circular, desse modo, o nodo inicial e final são o mesmo. Na Figura 12 tem-se uma representação da estrutura que foi utilizada no desenvolvimento deste trabalho.

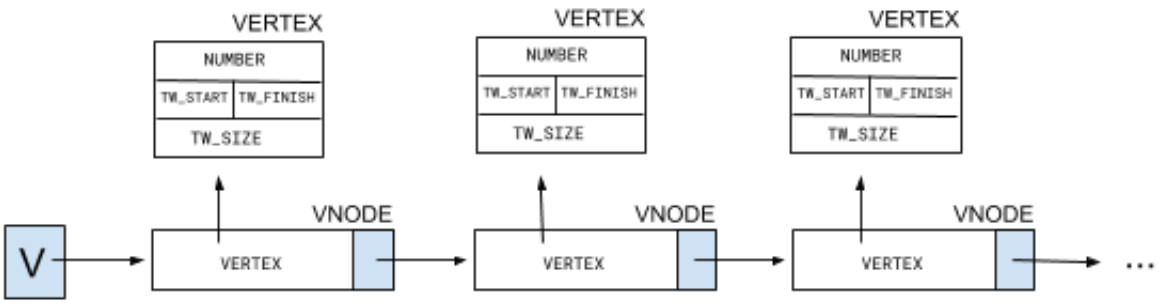
Figura 12 – Lista Duplamente Encadeada



Fonte: do autor

Para a representação dos vértices que ainda não foram adicionados à rota foi utilizada uma estrutura de fila implementada por meio de uma lista simplesmente encadeada, onde cada nodo aponta para o vértice que contém as informações do seu número, tamanho do intervalo das janelas de tempo, e valor inicial e final da janela de tempo. Na Figura 13 tem-se uma representação dessa estrutura.

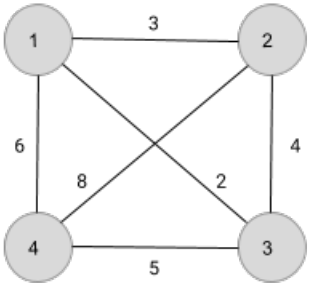
Figura 13 – Lista Simplesmente Encadeada



Fonte: do autor

Por fim, para a representação das distâncias entre os vértices foi utilizada uma matriz de ordem n, onde n corresponde ao número de vértices. Por exemplo, na Figura 14 tem-se um exemplo de um grafo com 4 vértices e sua respectiva matriz de distâncias. Por exemplo, o valor 4 da linha 3 e coluna 2 representa que a distância entre o vértice 3 e o vértice 2 é igual a 4.

Figura 14 – Matriz de Adjacências



	1	2	3	4
1	0	3	2	6
2	3	0	4	8
3	2	4	0	5
4	6	8	5	0

Fonte: do autor

5.2 DESCRIÇÃO DA IMPLEMENTAÇÃO PARALELA

A partir do perfilamento da aplicação observou-se que que o tempo de execução se concentra nas fases GENI e US, sendo que estas foram escolhidas para serem paralelizadas. Para o perfilamento da implementação foi utilizado a ferramenta *gprof* (ROBBINS, 2005). Na Figura 15 tem-se o resultados do perfilamento realizado. Como pode ser observado a fase GENI corresponde a 26.2% do tempo total de execução. Já a fase US corresponde a 73.4% do tempo total de execução.

Figura 15 – Perfilamento do código

```
                                                <spontaneous>
[3]     73.4    0.00    1.74                 US [3]
                0.21    1.53     274/372         GENI_Insertion [1]
                0.00    0.00     274/274         US_Removal [8]
-----------------------------------------------
                                                <spontaneous>
[4]     26.2    0.00    0.62                 GENI [4]
                0.07    0.55      98/372         GENI_Insertion [1]
                0.00    0.00       1/1           GENI_Initial [9]
```

Fonte: do autor

De forma a aproveitar o paralelismo em processadores *multi-core* foi realizada a paralelização da heurística *GENIUS* utilizando a biblioteca *OpenMp*. Para a paralelização foi utilizada a diretiva de paralelização *parallel*, que realiza a paralelização da seção de código de acordo com a quantidade de *threads* informadas (função *(*omp_set_num_threads).

Na fase GENI o vértice a ser inserido v_i é escolhido, e após os pivôs v_j são distribuídos de forma paralela, onde cada *thread* irá executar a inserção variando os pivôs utilizados. Ao final da execução de todas as *threads*, são comparadas as rotas geradas por cada inserção e a rota com menor custo é escolhida. O próximo vértice v_i é selecionado e o processo se repete. No Algoritmo 13 tem-se um pseudocódigo da paralelização da fase GENI. A função *(*omp_get_num_thread) é utilizada para retornar o número da *thread* atual.

Algoritmo 13: Paralelização da Inserção GENI para PCVJT

```
Input: s, v, nThreads
Output: s*
res[nThreads] ← s;
#pragma omp parallel
{
tid ← omp_get_thread_num()
for v_i ∈ N_p(v) do
    for v_j ∈N_p(v) do
        for v_k ∈ N_p(v_{i+1}) do
            s'' ← InsercaoTipoI(s', v, v_i, v_j, v_k);
            if f(s'') < f(res[tid]) then
                res[tid] ← s'';
            for v_l ∈ N_p(v_{j+1}) do
                s'' ← InsercaoTipoII(s', v, v_i, v_j, v_k, v_l);
                if f(s'') < f(res[tid]) then
                    res[tid] ← s'';
    v_i é incrementado em nThreads vezes;
}
for s' ∈ res do
    if f(s') < s* then
        s* ←s'
```

Já na fase US, o vértice a ser removido v_i é escolhido, e após os pivôs v_k são distribuídos de forma paralela entre as *threads*. Desta forma, cada *thread* irá executar a remoção variando os pivôs utilizados. Ao final da execução de todas as *threads*, são comparadas as rotas gerados por cada remoção e a rota de menor custo é escolhida. O próximo vértice v_i é selecionado e o processo de remoção se repete. No Algoritmo 14 tem-se um pseudocódigo da paralelização da fase US.

Algoritmo 14: Paralelização da Remoção US

Input: v_i, s, $nThreads$
Output: $s*$
res[nThreads] $\leftarrow s$;
#pragma omp parallel
{
tid $\leftarrow omp\_get\_thread\_num()$
for $v_k \in N_p(v_{i-1})$ **do**
  for $v_j \in N_p(v_{k+1})$ **do**
    $s'' \leftarrow$ ***RemocaoTipoI***(s', v_i, v_j, v_k);
    if $f(s'') < f(res[tid])$ **then**
      $res[tid] \leftarrow s''$;
    for $v_l \in N_p(v_{i+1})$ **do**
      $s'' \leftarrow$ ***RemocaoTipoII***(s', v_i, v_j, v_k, v_l);
      if $f(s'') < f(res[tid])$ **then**
        $res[tid] \leftarrow s''$;
  v_k é incrementado em *nThreads* vezes;
}
for $s' \in res$ **do**
  if $f(s') < s*$ **then**
    $s* \leftarrow s'$

5.3 TESTES E RESULTADOS OBTIDOS

Todos os testes foram realizados utilizando um computador com 20 GB de memória RAM, 500GB de disco SSD e um processador Intel Core i7 8550U, com 4 núcleos físicos e 8 *threads*, possuindo frequência de 1.8 Ghz por núcleo e podendo atingir até 4.0 Ghz em caso de ativação do *Turbo Boost* [1].

Para avaliar o desempenho da aplicação sequencial e paralela foram realizadas 5 tomadas de tempo para cada uma das instâncias selecionadas do problema, sendo que cada instância foi executada variando o número de *threads*. Para o cálculo do tempo de execução de cada instância, foi considerado o tempo médio das 5 execuções. A partir do tempo médio de execução foi realizado o cálculo do *speedup* (Equação 4.2) e o cálculo da eficiência (Equação 4.3).

[1] O *Turbo Boost* é uma tecnologia da Intel que acelera o desempenho do processador automaticamente nos picos de carga, fazendo com que o processador opere acima da frequência normal (INTEL, 2020)

5.3.1 GRAFOS UTILIZADOS PARA TESTES

Para o desenvolvimento e realização de testes foi utilizado o pacote de grafos proposto por DUMAS *et al.* (1995). Os grafos disponíveis neste pacote possuem tamanho variável entre 20 e 200 vértices, podendo ser divididos em 7 classes diferentes em relação ao número de vértices e 27 classes diferentes considerando em relação as janelas de tempo e número de vértices. Cada uma das 27 classes possui 5 instâncias, totalizando 135 grafos. Na Tabela 2 tem-se um resumo da quantidade de instâncias em relação aos seus respectivos número de vértices. Todos os grafos disponíveis possuem soluções ótimas para o problema (DUMAS *et al.*, 1995).

Tabela 2 – Instâncias para teste da heurística

Nº de Vértices	Nº de Instâncias
20	25
40	25
60	25
80	20
100	15
150	15
200	10

Fonte: do autor

Na Figura 16 tem-se um exemplo do formato de arquivo utilizado para a representação dos grafos. Na primeira linha é informado o número n de vértices do grafo. Nas n linhas subsequentes é informada a matriz de distâncias, onde na primeira linha são representadas as distâncias entre o vértice inicial e os demais vértices, e na primeira coluna são representadas as distâncias entre os demais vértices e o vértice inicial. Dessa forma, a matriz representa a distância entre dois vértices i e j. Nas últimas n linhas são definidas as janelas de tempo para cada vértice onde a primeira coluna representa o início da janela de tempo e a segunda coluna o final da janela de tempo. Por exemplo, na Figura 16, o número 6 na primeira linha representa o número de vértices, as linhas 2 a 7 representam a matriz de distâncias e as linhas 8 a 13 representam as janelas de tempo.

Figura 16 – Formato do Arquivo Utilizado

```
 1 6
 2 0 40 45 13 24 38
 3 40 0 10 37 27 24
 4 45 10 0 39 36 34
 5 13 37 39 0 30 41
 6 24 27 36 30 0 14
 7 38 24 34 41 14 0
 8 0         372
 9 198       242
10 278       316
11 59        105
12 150       202
13 244       253
```

Fonte: do autor

5.3.2 Validação da Implementação

Para validação da implementação os resultados obtidos foram comparados com os resultados apresentados por DUMAS *et al.* [2].

Na Tabela 3 são apresentados os resultados encontrados para diferentes instâncias do problema. Como pode ser observado, quando comparado em relação a distância total do percurso, ocorreram pequenas variações no resultado em instâncias de maior tamanho. Esta variação ocorre pois o resultado da implementação heurística está sendo comparado a solução ótima do problema. Desta forma, a solução heurística não garante a solução ótima como resultado, porém os resultados obtidos foram satisfatórios, possuindo uma baixa taxa de erro.

Tabela 3 – Resultados Obtidos

Instância do Problema	Resultado Ótimo	Resultado Obtido	Erro
n20w20.001	378	378	0%
n20w40.002	333	333	0%
n20w60.005	338	340	0.6%
n20w80.004	304	304	0%
n20w100.001	237	237	0%
n40w20.004	404	404	0%
n40w40.002	461	461	0%
n40w60.003	408	411	0.7%
n40w80.005	344	344	0%
n40w100.001	429	445	3.7%
n60w20.001	551	551	0%
n60w40.003	603	603	0%
n60w60.004	571	572	0.1%
n60w80.003	550	550	0%
n60w100.005	451	454	0.6%
n80w20.004	615	615	0%
n80w40.002	618	620	0.3%
n80w60.004	619	619	0%
n80w80.003	589	591	0.3%
n100w20.001	738	738	0%
n100w40.002	653	653	0%
n100w60.005	661	661	0%
n150w20.002	864	865	0.1%
n150w40.003	727	729	0.3%
n150w60.005	840	845	0.6%
n200w20.002	973	974	0.1%
n200w40.004	980	984	0.4%

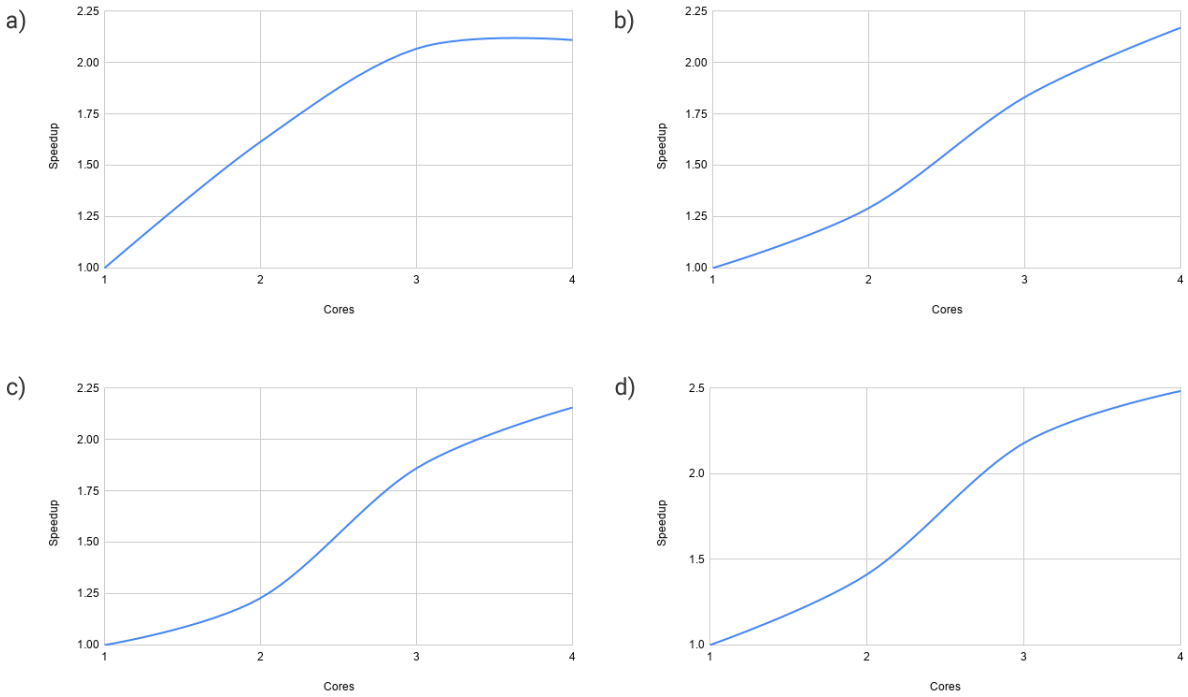
Fonte: do autor

[2] A solução ótima das instâncias podem ser acessada em: <https://homepages.dcc.ufmg.br/~rfsilva/tsptw/>

5.3.3 Avaliação de Desempenho da Implementação Paralela

No gráfico da Figura 17 a), tem-se o *speedup* obtido utilizando grafos com 20 vértices. Para grafos, com 20 vértices e 4 *threads* foi atigido, um *speedup* de $2,11$ e uma eficiência de 53%. Já no gráfico da Figura 17 b), tem-se o *speedup* para grafos com 60 vértices, sendo atingido um *speedup* de $2,17$ e eficiência de 54% Neste caso, observa-se que houve uma pequena melhora em relação aos grafos com 20 vértices. Na Figura 17 c) tem-se o *speedup* e a eficiência obtida para grafos com 100 vértices, sendo atingido um *speedup* de $2,15$ e eficiência de 54%. Por fim, na Figura 17, tem-se o *speedup* obtido para grafos com 200 vértices. Neste caso, tem-se um *speedup* de $2,48$ e eficiência de 62%.

Figura 17 – Avaliação de Desempenho



Fonte: do autor

Observa-se que o *speedup* e a eficiência da implementação desenvolvida aumentam com o aumento do tamanho das instâncias do problema. Isso ocorre, pois com o aumento do número de vértices maior é a quantidade de operações paralelas efetuadas, aumentando a eficiência da aplicação.

6 CONSIDERAÇÕES FINAIS

O Problema do Caixeiro Viajante é um problema clássico da área de grafos, sendo um dos problemas de análise combinatória mais pesquisados atualmente e possuindo diversas aplicações práticas. Podemos citar sua utilização na construção de placas de circuitos eletrônicos, na análise de cristais e no problema de roteirização de veículos. Esse se encontra na categoria NP-Difícil possuindo uma complexidade O(n!).

A variação do Caixeiro Viajante com Janelas de Tempo pode ser utilizada para a solução de diversos problemas da área logística como, por exemplo, a roteirização de veículos, melhoria da eficiência, redução de custos, entre outros. trazendo benefícios tanto para as empresas como para os clientes.

A forma mais simples de implementação do Caixeiro Viajante é por meio da técnica de força bruta que realiza a verificação de todos os caminhos possíveis. Devido a complexidade do problema, a utilização do método de força bruta se torna inviável com o aumento no número de clientes e número de caminhos. Desta forma, são utilizadas heurísticas para resolver o problema, uma vez que essas reduzem significativamente o tempo de execução, reduzindo o espaço de busca, porém não garantem que a solução encontrada seja a solução ótima (PEARL, 1984).

Dentre as heurísticas para a solução do Problema Caixeiro Viajante com Janelas de Tempo destaca-se a heurística GENIUS (SILVA, 2012). Essa foi inicialmente desenvolvida para a resolução do Problema do Caixeiro Viajante, porém, com algumas adaptações, é utilizada para a resolução do Problema do Caixeiro Viajante com Janelas de Tempo, de forma que a rota resultante obedeça as janelas de tempo de cada vértice (REBOUÇAS, 2016).

A implementação da heurística GENIUS foi desenvolvida de forma a explorar o paralelismo em arquiteturas com múltiplos núcleos de processamento, uma vez que esta arquitetura está presente na maioria dos computadores atuais (TANENBAUM, 2013). Para o desenvolvimento foi utilizada a biblioteca de *threads OpenMp* (QUINN, 2003), sendo esta a biblioteca de *threads* mais utilizada atualmente (MURAROLLI, 2015).

Para a realização dos testes foram utilizados os grafos propostos por DUMAS *et al.*(1995). Os grafos disponíveis possuem tamanho variando entre 20 a 200 vértices e possuem soluções ótimas. A partir de uma comparação dos resultados obtidos pela heurística GENIUS com as soluções ótimas, observou-se que a heurística produziu resultados satisfatórios, possuindo uma baixa taxa de erro.

Para avaliação do desempenho da aplicação foram utilizadas as métricas de *speedUp* e da eficiência. Observou-se que em instâncias com maior número de vértices houve um aumento do *speedup* e eficiência da aplicação quando comparado a instâncias de menor tamanho, sendo atingido um resultado satisfatório com *speedup* de 2.48x e eficiência de 0.62% ao utilizar de 4

cores em instâncias de 200 vértices.

6.1 TRABALHOS FUTUROS

Como possíveis trabalhos futuros sugere-se:

- Criação de uma interface gráfica para a visualização das rotas geradas;
- Otimização da versão paralela de forma a aumentar a eficiência do algoritmo;
- Realizar otimizações na fase de melhoramento do algoritmo GENIUS procurando aproximar a solução obtida da solução ótima em todos os casos.

REFERÊNCIAS

BOAVENTURA, P.; JURKIEWICZ, S. **Grafos: Introdução e Prática**. [S.l.]: Blucher, 2019. 192 p. ISBN 9788521215172.

BUTENHOF, D. R. **Programming with POSIX threads**. [S.l.]: Addison-Wesley, 1997. (Addison-Wesley professional computing series). ISBN 0201633922,9780201633924.

DUMAS, Y. *et al.* An optimal algorithm for the traveling salesman problem with time windows. **Operations Research**, INFORMS, v. 43, n. 2, p. 367–371, 1995. ISSN 0030364X, 15265463.

FLYNN, M. J. Some computer organizations and their effectiveness. **IEEE Trans. Comput.**, IEEE Computer Society, USA, v. 21, n. 9, p. 948–960, set. 1972. ISSN 0018-9340.

FOSTER, I. **Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering**. USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN 0201575949.

GENDREAU, M.; HERTZ, A.; LAPORTE, G. New insertion and postoptimization procedures for the traveling salesman problem. **Operations Research**, v. 40, p. 1086–1094, 12 1992.

GENDREAU, M. *et al.* A generalized insertion heuristic for the traveling salesman problem with time windows. **Operations Research**, v. 46, n. 3, p. 330–335, 1998.

GOLDBARG, M. **Grafos: Conceitos, algoritmos e aplicações**. [S.l.]: Elsevier, 2012. ISBN 9788535257168.

INTEL. **Tecnologia Intel® Turbo Boost 2.0**. 2020. Disponível em: <https://www.intel.com.br/content/www/br/pt/architecture-and-technology/turbo-boost/turbo-boost-technology.html>. Acesso em: 27 jun. 2020.

JUNIOR, J. P. **A interferência da logística no e-commerce**. 2018. Disponível em: <https://www.ecommercebrasil.com.br/artigos/interferencia-logistica-no-e-commerce/>. Acesso em: 27 jun. 2020.

MIRANDA, L.; PORSANI, M. **Implementação de algoritmo paralelo para inversão de dados geofísicos. TCC de Graduação em Geofísica, CPGG/IGEO/UFBA, 2011.** 2011.

MLADENOVIR, E. H. N. Variable neighborhood search. **Operations Research**, v. 40, p. 1097–1100, 11 1997.

MURAROLLI, P. L. **Inovações Tecnológicas nas Perspectivas Computacionais**. [S.l.]: Biblioteca24horas, 2015.

NETTO, P.; JURKIEWICZ, S. **Grafos - Introdução E Prática**. [S.l.]: EDGARD BLUCHER, 2017. ISBN 9788521204732.

PEARL, J. **Heuristics: Intelligent Search Strategies for Computer Problem Solving**. USA: Addison-Wesley Longman Publishing Co., Inc., 1984. ISBN 0201055945.

PESANT, G.; GENDREAU, M.; ROUSSEAU, J.-M. Genius-cp: A generic single-vehicle routing algorithm. In: ____. [S.l.: s.n.], 2006. v. 1330, p. 420–434.

QUINN, M. J. **Parallel Programming in C with MPI and OpenMP**. [S.l.]: McGraw-Hill Education Group, 2003. ISBN 0071232656.

REBOUÇAS, R. S. **Problema do Caixeiro Viajante com Coleta de Prêmios e Janelas de Tempo**. 2016.

REINA, C. D. **Roteirização de Veículos com Janelas de Tempo Utilizando Algoritmo Genético**. 2012.

ROBBINS, A. **Unix in a Nutshell, Fourth Edition**. 4. ed. [S.l.]: O'Reilly Media, Inc., 2005. ISBN 9780596100292.

SILVA, F. A. V. **UM ALGORITMO GENÉTICO PARA O PROBLEMA DE ROTEAMENTO DE VEÍCULOS COM JANELA DE TEMPO APLICADO NA DISTRIBUIÇÃO DE SERVIÇOS DE TELECOMUNICAÇÃO**. 2016.

SILVA, T. C. B. da. **GENILS-TS-CL-PR:Um algoritmo heurístico para resolução do Problema de Roteamento de Veículos com Coleta e Entrega Simultânea**. 2012.

STABELINE, D. **O que é roteirização e porque é fundamental para a sua empresa**. 2019. Disponível em: <https://blog.texaco.com.br/ursa/o-que-e-roteirizacao/>. Acesso em: 27 jun. 2020.

TANENBAUM. **Organização estruturada de computadores**. [S.l.]: PRENTICE HALL BRASIL, 2013. ISBN 9788581435398.

TANENBAUM, A. **Sistemas operacionais modernos**. [S.l.]: Prentice-Hall do Brasil, 2010. ISBN 9788576052371.