Universidade Federal Fluminense
Instituto de Ciência e Tecnologia
Bacharelado em Ciência da Computação

ANA CAROLINA CLIVATTI FERRONATO

Um estudo de heurísticas para variações do Problema do Caixeiro Viajante

Rio das Ostras - RJ
2017

ANA CAROLINA CLIVATTI FERRONATO

UM ESTUDO DE HEURÍSTICAS PARA VARIAÇÕES DO PROBLEMA DO CAIXEIRO VIAJANTE

Monografia apresentada ao Curso de Bacharelado em Ciência da Computação do Instituto de Ciência e Tecnologia da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Bacharel.

Orientador: Profa. Dra. MAISE DANTAS DA SILVA

Rio das Ostras - RJ

2017

ANA CAROLINA CLIVATTI FERRONATO

UM ESTUDO DE HEURÍSTICAS PARA VARIAÇÕES DO PROBLEMA DO CAIXEIRO VIAJANTE

Monografia apresentada ao Curso de Bacharelado em Ciência da Computação do Instituto de Ciência e Tecnologia da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Bacharel.

Aprovada em JULHO de 2017.

BANCA EXAMINADORA

Profa. Dra. MAISE DANTAS DA SILVA - Orientador
UFF

Prof. Dr. DANILO ARTIGAS DA ROCHA
UFF

Prof. Dr. ANDRÉ RENATO VILLELA DA SILVA
UFF

Rio das Ostras - RJ
2017

Aos meus pais, Marcos e Rose.

Se fui capaz de ver mais longe,
é porque me apoiei em ombros de gigantes.
ISAAC NEWTON

Agradecimentos

Agradeço primeiramente aos meus pais, Marcos e Rose, que são a minha base e a razão de eu conseguir chegar até esta etapa tão sonhada da minha vida. Agradeço também a eles também que sempre me deram apoio e incentivo nas horas difíceis, de desânimo e cansaço.

Agradeço minhas irmãs Gabriela e Fernanda, que sempre fizeram entender que o futuro é feito a partir da constante dedicação no presente. Sempre dispostas a me ajudar conforme fosse possível, seja em correções ortográficas ou apenas por lerem minhas seções recém escritas. Agradeço a confiança depositada em mim durante todo esse processo, e a ansiedade de comemorar comigo mais um objetivo alcançado.

Agradeço também a minha orientadora Maise Dantas, pelo suporte no tempo que lhe coube, pela dedicação ainda que distante no início do estudo, sendo dedicada e que com sua sabedoria soube dirigir-me os passos e os pensamentos para o alcance de meus objetivos. Agradeço a paciência e os conselhos sábios nos momentos de desespero.

Agradeço ainda ao meu primo Marcos Clivatti Freitag, que sempre foi uma inspiração para meus estudos, me apresentando seus trabalhos e monografias desde a minha infância. Agradeço muito por todo o apoio, desde o início da graduação, por todos os conselhos e direcionamentos em todos os momentos em que precisei.

Não poderia deixar de agradecer o meu namorado, Leonardo Neves Gall, que se fez presente e parceiro durante a escrita deste trabalho. Agradeço a disponibilidade de passar finais de semanas inteiros em frente ao computador me apoiando, incentivando e principalmente acreditando em mim quando, em algumas vezes, eu mesma já não conseguia.

Agradeço a meus amigos, Mariana Magalhães e Raphael Sampaio, que por tanto tempo tiveram seus convites adiados, e até mesmo recusados, por ter que me dedicar a realização deste trabalho. Agradeço por me entenderem e não me abandonarem, esses convites sempre serviram de incentivo para concluir esta graduação.

Finalmente, gostaria de agradecer a FAPERJ pelo apoio financeiro.

Lista de Figuras

Lista de Tabelas

Lista de abreviaturas

1. ALSP - Procedimento de Busca Local Adaptável.
2. CAH - Heurística de Adição de Mercadorias.
3. EA - Algoritmo Evolucionário.
4. EA+ALSP - Algoritmo Híbrido Evolucionário e de Busca Local Adaptável.
5. LRC - Lista Restrita de Candidatos.
6. MA1 - Algoritmo Memético (1^a versão).
7. MA2 - Algoritmo Memético (2^a versão).
8. MAH - Heurística de Adição de Mercados.
9. PCV - Problema do Caixeiro Viajante.
10. PCVc - Problema do Caixeiro Viajante Comprador.
11. PCVa - Problema do Caixeiro Viajante Alugador.
12. PL - Programa Linear.
13. TA - Algoritmo Transgenético.
14. UFLP - Problema de Localização de Instalações Não-Capacitado.
15. VND - Busca Local de Descida.

Sumário

Resumo

O Problema do Caixeiro Viajante (PCV) é um problema NP-díficil aplicável a vários problemas reais e atuais, além de ser um dos problemas de otimização mais estudados na área. Partindo do interesse existente neste problema e da aplicabilidade de suas soluções, este trabalho tem por objetivo estudar duas variantes do PCV, o Problema do Caixeiro Viajante Comprador (PCVc) e o Problema do Caixeiro Viajante Alugador (PCVa), apresentar diferentes abordagens e heurísticas existentes na literatura, descrever os melhores algoritmos conhecidos e então comparar os resultados obtidos por estes. Este trabalho está organizado em 3 (três) partes. Na parte 1, será abordado o problema da pesquisa, suas questões investigativas, objetivos gerais e específicos e a justificativa do projeto. Na parte 2, será abordada a variante PCVc, com o foco da pesquisa na heurística de adição de mercados e na heurística de busca local. Na parte 3, será abordada a variante PCVa, com o foco da pesquisa no algoritmo memético e no algoritmo híbrido EA+ALSP.

Palavras-chave: Problema do Caixeiro Viajante; Caixeiro Viajante Comprador; Caixeiro Viajante Alugador; Heurísticas; NP-completude.

Abstract

The Traveling Salesman Problem (TSP) is an NP-hard problem applicable to several real and current problems, besides being one of the most studied optimization problems in the area. Based on the interest in this problem and the applicability of its solutions, this work aims to study two variants of TSP, Traveler Purchaser Problem (TPP) and Cars Renter Salesman Problem (CaRS), present different approaches and existing heuristics, describe the best known algorithms and then compare their results. This paper is organized in 3 (three) parts. In Part 1, the research problem, its investigative issues, general and specific objectives and the justification of the project will be addressed. In part 2, the TPP variant will be discussed, with the focus on the market addition heuristics and local search heuristics. In part 3, the CaRS variant will be discussed, with the focus on the memetic algorithm and the hybrid algorithm EA + ALSP.

Keywords: Traveling Salesman Problem; Traveling Purchaser Problem; Cars Renter Salesman; Heuristics; NP-completeness.

Capítulo 1

Introdução

A economia está inerente na vida das pessoas, podendo estar relacionada à economia de tempo ou recursos, por exemplo. Dessa forma, é cada vez mais necessária a otimização e melhoria das atividades do ser humano, sejam elas em sua rotina de trabalho ou até mesmo em suas tarefas domésticas. É neste âmbito que enquadram-se os problemas de transporte, que têm sido usados para modelar vários contextos de aplicação e são computacionalmente desafiadores. Um exemplo muito estudado neste aspecto é o Problema do Caixeiro Viajante, que pode tratar, ao mesmo tempo, da seleção de fornecedores, da otimização do plano de compras e das decisões de roteamento do comprador.

O Problema do Caixeiro Viajante (PCV) consiste em determinar o menor caminho (ou rota) para percorrer determinado número de cidades, passando por cada uma exatamente uma vez, e voltando para a cidade inicial. Este problema pode ser modelado por um grafo completo $G = (V, E)$, onde $V = \{v_1, \cdots, v_n\}$ representa o conjunto de vértices e $E = \{e_1, \cdots, e_m\}$, o conjunto de arestas, e buscamos um ciclo hamiltoniano mínimo (cuja soma dos pesos das arestas seja o menor possível).

O contexto de ciclo hamiltoniano mínimo como um problema de aplicação foi citado pela primeira vez nos anos 1920, por Karl Menger, em Viena. Somente após seminários posteriores, o problema passaria a ser referido como Caixeiro Viajante. Ainda assim, não há informações concretas quanto ao início dos estudos [1].

Conforme definido por [2], o PCV apresenta de uma maneira concreta e simples a velocidade de crescimento da função fatorial e pode ser utilizado como "unidade métrica" para dimensionar a complexidade computacional de diversos problemas combinatórios que possuem tantas soluções aceitáveis quanto o problema em questão.

O PCV foi inspirado na necessidade dos vendedores em realizarem entregas em diversas cidades percorrendo o menor caminho possível, reduzindo assim o tempo necessário para a viagem e os possíveis custos com transporte e combustível. Ele pode ser reescrito em um significativo número de variantes. A motivação das variantes para este problema decorre, na maioria dos casos, de aplicações práticas. Normalmente, estas variantes possuem ligações com outros problemas de otimização combinatória. De início, não houve uma sistematização na literatura. Sendo assim, uma mesma variante pode ocorrer nomeada de diferentes maneiras.

1.1 Objetivo geral

Neste trabalho, buscamos analisar e comparar algumas heurísticas aplicadas às seguintes variantes do PCV: Problema do Caixeiro Viajante Comprador (PCVc) e Problema do Caixeiro Viajante Alugador (PCVa), avaliando os resultados computacionais existentes na literatura.

1.2 Objetivos específicos

Diante da NP-dificuldade do PCV e das duas variações estudadas, foi realizado um levantamento das heurísticas mais atuais e estudadas para estas variações, comparando seus resultados computacionais e realizando um estudo de quais heurísticas apresentam significativas chances de melhoria e evolução.

1.3 Bibliometria da pesquisa

Partindo das informações descritas acima, uma bibliometria de pesquisa foi realizada na base *Scopus*[1], utilizando a seguinte estrutura de termos:

Termos	Tesauros
Problema do Caixeiro Viajante	PCV / Traveling Salesman Problem
Heurística	Heuristics
Variantes	Variants

Tabela 1.1: Bibliometria da Pesquisa - Termos e Tesauros. FONTE: O autor.

O termo "Traveling Salesman Problem", utilizado como termo de busca, resultou em 19.839 trabalhos publicados. Quando refinado para ("Traveling Salesman Problem" AND "Heuristics"), foram obtidos 2.486 resultados. Já a busca por ("Traveling Salesman Problem" AND "Variants") retornou 668, e ("Variants" AND "Heuristics") 3.582 resultados. Por fim, a união dos termos em uma única busca ("Traveling Salesman Problem" AND "Heuristics" AND "Variants") resultou em 193 publicações, conforme demonstrado na Figura 1.1, representada por um diagrama de Venn.

[1] https://www.scopus.com

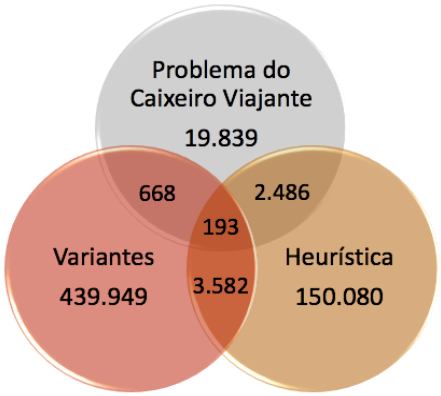

Figura 1.1: Diagrama de Venn - Bibliometria da Pesquisa. FONTE: O autor.

Ainda dentro da base *Scopus*, devido ao grande número de publicações, a busca foi restringida aos trabalhos publicados nos últimos dez anos (de 2007 a 2016). A pesquisa não passou por restrição quanto ao tipo de trabalho; artigos, dissertações, teses e livros foram considerados. Conforme ilustrado na Figura 1.2, é possível verificar o crescente interesse no estudo do PCV.

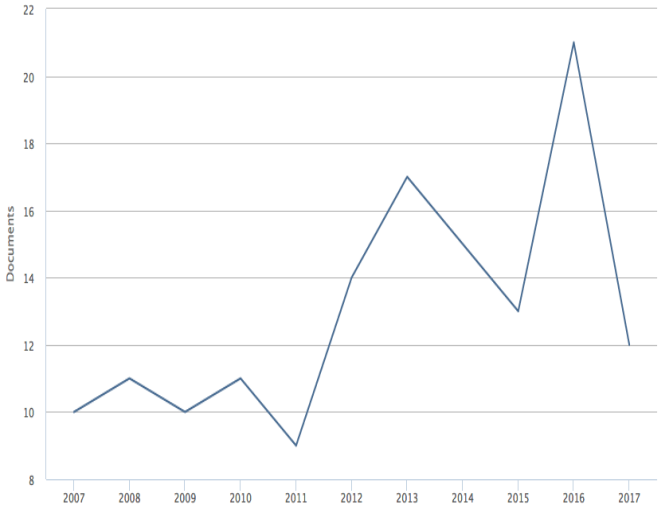

Figura 1.2: Bibliometria - Documentos publicados por ano. FONTE: O autor.

Após a busca ser realizada, uma análise foi feita a fim de identificar quais variantes do PCV estão sendo mais estudadas/publicadas e quais variantes possuem publicações mais atuais na literatura. O passo seguinte consistiu em identificar quais os principais autores que publicaram trabalhos sobre o tema, conforme visto na Figura 1.3.

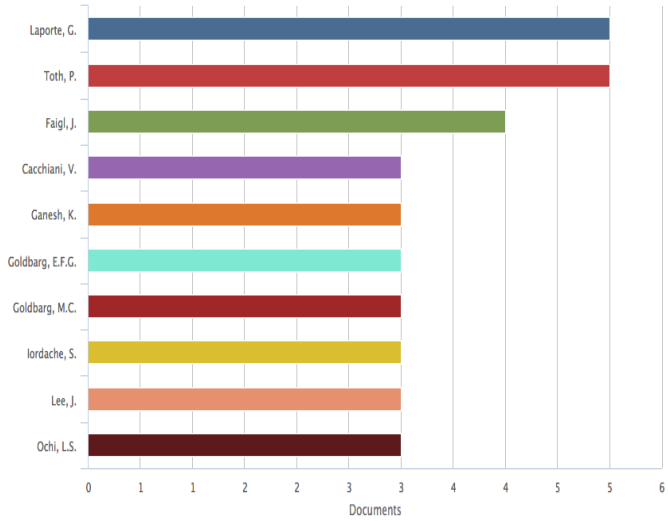

Figura 1.3: Bibliometria - Documentos publicados por autor. FONTE: O autor.

Baseado na Figura 1.3, iniciou-se uma busca detalhada dentre os autores para identificar então quais eram os documentos mais relevantes para o proposto trabalho. A busca foi realizada em princípio pela avaliação dos resumos disponíveis na própria base *Scopus*. Esta possibilita também uma análise dos países que mais estudam/publicam na área da pesquisa, conforme demonstrado na Figura 1.4.

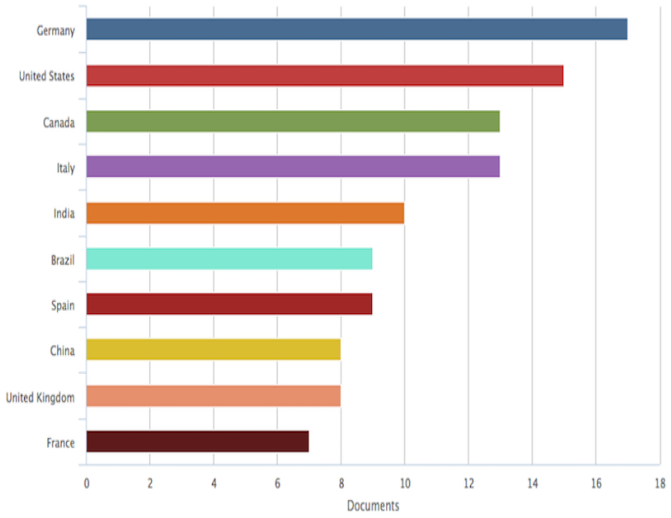

Figura 1.4: Bibliometria - Documentos publicados por país. FONTE: O autor.

Como pode ser verificado na Figura 1.5, que representa as diversas áreas onde o PCV é estudado, conclui-se que este problema é de grande interesse internacional.

Ainda na base *Scopus*, realizou-se uma busca a partir dos termos "Optimization" e "Optimization Problem", onde obteve-se um total de 81.540 trabalhos publicados nos últimos 10 anos. Em seguida, os termos relacionados ao PCV e suas variantes, PCVc e PCVa foram adicionados resultando em 9.156 trabalhos publicados no mesmo período. A partir destes resultados, pode-se comprovar a relevância do

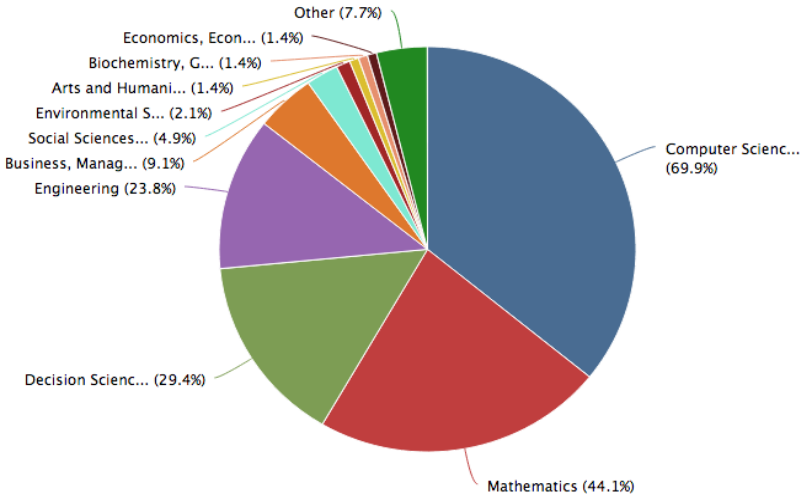

Figura 1.5: Bibliometria - Documentos publicados por área. FONTE: O autor.

estudo do PCV dentro da área de otimização. A Figura 1.6 apresenta os dados obtidos nesta busca, o que garante a relevância deste trabalho.

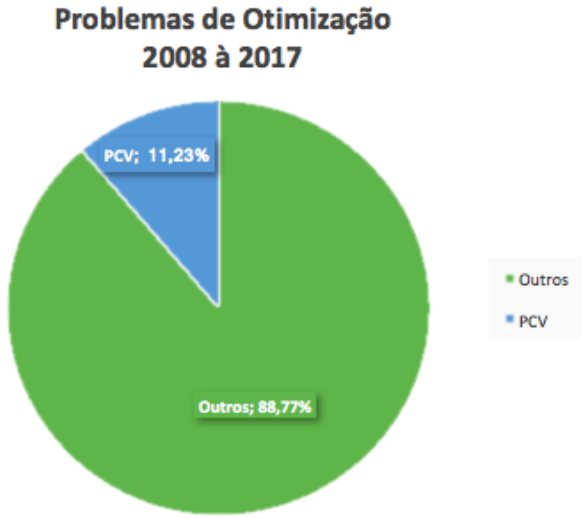

Figura 1.6: Porcentagem de trabalhos relacionados ao PCV e às duas variações estudadas.

O presente trabalho está dividido em cinco capítulos. No Capítulo 2, são apresentadas as definições relacionadas ao problema, assim como a revisão de literatura do PCV. No Capítulo 3, é apresentada a revisão de literatura do Problema do Caixeiro Viajante Comprador (PCVc), algumas heurísticas aplicadas ao problema, e seus algoritmos, assim como resultados computacionais para o problema. No Capítulo 4, abordamos o Problema do Caixeiro Viajante Alugador (PCVa), revisando sua literatura e resultados obtidos. No Capítulo 5, é apresentada uma comparação entre os resultados e as variantes estudadas e as conclusões do trabalho realizado.

Capítulo 2

Definições e nomenclatura

Neste capítulo, são apresentadas as definições e nomenclaturas adotadas ao longo do trabalho. Outras definições utilizadas podem ser encontradas em [3].

2.1 Definições e nomenclatura básica

Segundo Davendra [4], a teoria da complexidade computacional é um ramo de estudo que envolve tanto a área da computação quanto as áreas das matemáticas, o que é habitualmente relacionado com a classificação de problemas computacionais com suas dificuldades intrínsecas. Um dos principais problemas que ainda está com resolução em aberto é a questão de problemas P *versus* NP, que serão definidos na seção seguinte. A classe de problemas NP-Difíceis contém problemas muito importantes que, entretanto, não possuem nenhum algoritmo eficiente conhecido. Tal afirmação é constatada em Karp [5]. O Problema do Caixeiro Viajante (PCV) é um destes problemas, sendo considerado o problema mais intensamente estudado na matemática computacional por Ausiello *et al.* em [6].

No PCV, assume-se que o caixeiro tem que visitar um dado número de cidades, iniciando e finalizando seu trajeto em uma mesma cidade. Como solução para o problema, em sua versão de otimização, busca-se uma viagem cujo comprimento do caminho seja o menor possível. Conforme o número de cidades aumenta, a determinação da rota ótima (neste caso, um ciclo hamiltoniano) torna-se complexa o suficiente para dificultar o controle sobre uma determinada solução. As variações deste problema encontradas na literatura são comumente também classificadas como NP-difíceis.

Uma das mais atuais e conhecidas abordagens para a solução do PCV é a aplicação de *algoritmos evolucionários*. Estes algoritmos são frequentemente baseados na ocorrência de fenômenos naturais encontrados no dia a dia, que podem ser utilizados como modelos para a criação de algoritmos. Há, hoje, na literatura, um substancial número de algoritmos evolucionários, tais como *algoritmos genéticos*, *inteligência de enxames*, *otimização em nuvem de partículas* e o SOMA (*Self Organising Migrating Algorithm*). Existem também algoritmos baseados em formulações matemáticas, como evolução diferencial (DE - *diferential evolution*), busca dispersa (BD - *Scatter Search*) e *busca tabu*, os quais têm sido comprovados como algoritmos robustos e de grande utilidade. Mais informações a respeito destes algoritmos

podem ser encontradas em [7].

2.2 Grafos

De acordo com [8], "Uma ampla variedade de problemas pode ser expressa com clareza e precisão na linguagem pictórica, concisa, dos grafos". Na matemática, há uma área de pesquisa que estuda objetos combinatórios, definida como Teoria dos Grafos. Os grafos são boas ferramentas para a resolução de diversos problemas nos ramos da computação, matemática, engenharia e da indústria, por exemplo. Eles facilitam a resolução de vários problemas, devido a sua representação gráfica permitir uma melhor assimilação e contextualização de detalhes contidos em suas definições.

Muitos problemas combinatórios existentes tornaram-se de extrema importância na matemática, devido às suas aplicações práticas e também por serem importantes desafios intelectuais aos estudiosos da área. Os problemas da teoria de grafos envolvem questões de complexidade computacional, devido à motivação algorítmica existente nos mesmos.

Definicão 1. *[3] Um grafo G(V,E) é um par ordenado formado por um conjunto finito não-vazio V e um conjunto E de pares não-ordenados de elementos distintos de V. Os elementos de V são os vértices e os de E são as arestas de G, respectivamente. Cada aresta e ∈ E será denotada pelo par de vértices e = (v,w) que a forma.*

Definicão 2. *[3] Um grafo é definido como orientado quando o par de vértices $e = (v, w)$ não possui relação de simetria com $e' = (w, v)$. Ainda assim, os vértices v e w são ditos adjacentes por possuírem uma aresta incidente a eles.*

Conforme apresentado em [3], em um grafo orientado (ou direcionado), as arestas são chamadas de *arcos*. Uma sequência de arestas adjacentes que geram a ligação entre dois vértices é dita *trilha*. Uma *trilha* é um *passeio* quando não passa duas vezes pelo mesmo arco e um *caminho* se não passa duas vezes pelo mesmo vértice. Por fim, um caminho pode ser definido como fechado se o vértice inicial é o mesmo que final.

Em um grafo direcionado, toda aresta (v, w) é *divergente* de v e *convergente* à w; ou seja, a aresta detém uma única direção de v para w.

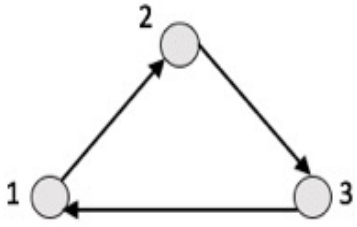

Figura 2.1: Um Grafo G(V,E) orientado.

Como em um *caminho* não há repetição de vértices, um caminho hamiltoniano é composto por um caminho em um grafo G que passa por todos os vértices de G exatamente uma vez. Um caminho fechado é denotado por *ciclo*. Logo, um grafo ciclo é um grafo com n vértices formado por apenas um ciclo passando por todos os vértices.

Os caminhos no estudo de grafos podem ser apresentados de diversas maneiras, sendo o caminho mais curto um dos assuntos mais importantes, por fazer parte de diversos problemas em grafos e por suas aplicações diretas a situações do dia-a-dia dos mais diferentes tipos de problemas e áreas de atuação.

Um resultado importante sobre Ciclos Hamiltonianos é dado por Karp, na década de 1970 (apud [9]): "O problema de decisão associado a determinação de ciclos e caminhos hamiltonianos em grafos sem propriedades particulares é NP-Completo."

No presente trabalho, utilizaremos a família de grafos hamiltonianos, que são aqueles que admitem um ciclo hamiltoniano. Um ciclo hamiltoniano é um ciclo que passa por todos os vértices do grafo exatamente uma vez e assim, um grafo é hamiltoniano se possui ciclo hamiltoniano, de acordo com [3].

2.3 Complexidade de problemas

Segundo Benoit, Robert e Vivien [10], Máquinas de Turing são essenciais no estudo da complexidade de um problema (ou algoritmo), já que, a partir destas, podemos acessar a complexidade de um problema e então definir seu tamanho e o número de intervalos de tempo que são necessários para resolvê-lo. O tamanho de um problema é dado pelo número de posições consecutivas utilizadas para armazenar seus dados na fita da máquina. O número de intervalos de tempo é o número de movimentos realizados antes que a máquina de Turing finalize a execução do programa, tendo em conta os dados inicialmente armazenados em sua fita.

De acordo com o lema *The composition of two polynomials is a polynomial* (A composição de dois polinômios é um polinômio) apresentado por [10], os valores de interesse (tamanho e tempo) podem ser definidos como um fator polinomial, já que, de um ponto de vista de classes de complexidade, os valores n, n^2, e até $n^{35} + n^{23} + 53$ pertencem a uma mesma classe, sendo todos valores polinomiais em n. Portanto, não há diferença se uma operação de um algoritmo custar n^2 ou $n^{35} + n^{23} + 53$ intervalos de tempo em uma Máquina de Turing.

A complexidade de um algoritmo é definida a partir do esforço (geralmente o tempo ou memória) utilizado para executá-lo e obter uma solução para um problema em particular. A dificuldade de um problema é definida de acordo com o número de recursos computacionais que são minimamente necessários para solucionar o problema. Consequentemente, a dificuldade de um problema está diretamente relacionada às complexidades dos algoritmos que o resolvem, com os quais podemos encontrar os limites inferiores e superiores na dificuldade do problema. O limite superior de complexidade de um problema faz referência ao melhor algoritmo existente que o resolve, enquanto o limite inferior faz referência a um resultado teórico que determine qual é a melhor complexidade possível que um algoritmo pode alcançar.

De acordo com [11] a notação mais utilizada para a medição de algoritmos é a O, que limita superiormente uma função de forma assintótica, por exemplo, a que determina a complexidade de um

algoritmo.

Definicão 3. *[11] Notação O - A função $C(n)$ é $O(F(n))$ se existem constantes positivas c e n_0 tais que $C(n) \leq c.F(n)$ quando $n \geq n_0$.*

A partir da Definição 3, pode-se dizer que, caso o espaço utilizado pela execução de um algoritmo seja $O(n^2)$, por exemplo, então a quantidade de memória utilizada é no máximo uma função quadrática da entrada.

A notação Ω é utilizada para a especificação do limite inferior de um problema, como é apresentado na Definição 4.

Definicão 4. *[11] Notação Ω - A função $C(n)$ é $\Omega(F(n))$ se existem constantes positivas c e n_0 tais que $C(n) \geq c.F(n)$ quando $n \geq n_0$.*

Ou seja, a partir de um valor n_0, o custo $C(n)$ é maior que $F(n)$ multiplicado por um fator constante. Dessa forma, assintoticamente, $C(n)$ cresce mais rapidamente que $F(n)$. Sendo assim, caso exista um algoritmo que resolva um problema, podem ser definidos um limite superior e/ou um inferior para tal, sendo possível, então, ter a certeza de uma solução, ainda que esta não seja ótima para o problema (um algoritmo de uma solução é *ótimo* quando a sua complexidade é igual ao limite inferior obtido). A cada melhoria realizada no algoritmo, reduzimos o limite superior da solução, o que nos proporciona versões cada vez mais próximas da solução ótima do problema. Encontrar o limite inferior de um problema é uma tarefa mais complexa de ser realizada, pois devemos provar que não é possível desenvolver um algoritmo que solucione o mesmo problema realizando menos esforço que o apresentado. Segundo Rothlauf, F [12], um problema pode ser definido como *fechado* quando os limites inferior e superior são da mesma ordem de grandeza.

Papadimitriou, C. *et al.* [13] define que problemas de decisão podem ser categorizados em diferentes grupos, fundamentados em suas dificuldades. A definição da complexidade de um problema apoia-se no tamanho do problema por meio do cálculo da complexidade de tempo e de espaço, onde a complexidade de tempo define quantas interações ou passos são necessários para se obter a solução (quanto mais difíceis os problemas, mais tempo é necessário para solucioná-lo), e a complexidade de espaço descreve a quantidade de espaço (normalmente a memória de um computador) que é necessária para resolver o problema. Da mesma maneira que ocorre com a complexidade de tempo, quanto maior o espaço utilizado, maior a dificuldade do problema.

Uma classe de complexidade é um grupo de problemas computacionais onde a quantidade de recursos computacionais necessários para resolvê-los possui o mesmo comportamento assintótico. Os limites dependem do tamanho do problema, também chamado de *tamanho da entrada* do problema. Nas subseções seguintes serão apresentadas as definições das classes de complexidade P, NP e NP-difícil, segundo as definições apresentadas em [12].

2.3.1 Classe de complexidade P

A classe de complexidade P, com P fazendo referência ao termo polinomial, é definida como o conjunto de problemas de decisão que podem ser solucionados por um algoritmo de complexidade de pior

caso polinomial. O tempo necessário para solucionar um problema de decisão em P é assintoticamente limitado (para $n > n_0$) por uma função polinomial $O(n^k)$. Para todos os problemas em P, há um algoritmo que soluciona qualquer instância do problema em um tempo $O(n^k)$, para alguma constante k.

2.3.2 Classe de complexidade NP

A classe de complexidade NP, com NP fazendo referência ao termo tempo polinomial não-determinístico, reúne o conjunto de problemas de decisão onde uma solução afirmativa, válida, ou simplesmente uma solução *'sim'* para o problema pode ser verificada em tempo polinomial. Ou seja, a representação formal da solução x (certificado *'sim'*) e o tempo necessário para verificar sua veracidade (se a solução *'sim'* é válida) deve ser polinomialmente limitada (ou simplesmente polinomial).

Todos os problemas na classe NP têm por propriedade a garantia de que suas soluções *'sim'* podem ser efetivamente verificadas. Porém, a definição da classe NP não apresenta informações sobre o tempo necessário para verificar soluções não válidas, ou simplesmente soluções *'não'*. Além disso, um problema NP pode não ser necessariamente solucionado em tempo polinomial. Para problemas que não fazem parte da classe NP, até mesmo verificar se a solução é realmente válida pode ser extremamente difícil e precisar de cálculos em tempo exponencial.

Dentro da classe NP, há uma subclasse de problemas definidos como NP-Completos.

Definicão 5. *[14] Um problema de decisão π é NP-Completo se:*

1. $\pi \subset NP$, *e*
2. *Todo problema em NP pode ser reduzido para π em tempo polinomial.*

2.3.3 Classe de complexidade NP-difícil

Um problema π é definido como NP-difícil se todo problema em NP pode ser reduzido polinomialmente a π. Segundo Rothlauf, F.[12], um problema π é redutível em tempo polinomial a um problema π', $(\pi' \neq \pi)$ se, e somente se, há uma transformação que modifique qualquer instância arbitrária x de π em uma instância x' de π' em tempo polinomial, tal que x seja uma instância *'sim'*, para π se, e somente se, x' for uma instância *'sim'* para π'. Um problema π é redutível a um outro problema π' se o π' possuir a mesma dificuldade (complexidade) ou for mais difícil que π. Logo, problemas NP-difíceis são pelo menos tão difíceis quanto quaisquer outros problemas da classe NP. Assim, problemas NP-difíceis não necessariamente pertencem à classe NP.

2.4 O Problema do Caixeiro Viajante e suas variantes

Conforme apresentado em [15], problemas matemáticos relacionados ao Problema do Caixeiro Viajante foram estudados nos anos de 1800 pelo matemático irlandês Sir William Rowam Hamilton e pelo britânico Thomas Penyngton Kirkman. A Figura 2.2 é uma fotografia do Jogo Icosiano (*Hamilton's Icosian Game*), publicada em [16]. O objetivo do jogo consiste em visitar 20 cidades, de forma que cada

cidade seja visitada uma única vez, retornando, então, à cidade inicial. O formato do tabuleiro, modelado como um grafo dodecaedro regular, está ilustrado na Figura 2.3. Uma possível solução para este jogo é a ordem de visitas ilustrada na Figura 2.4.

Figura 2.2: Jogo Icosiano de Hamilton.

O Problema do Caixeiro Viajante (cuja entrada é um conjunto de cidades e o custo de viagem referente a cada par deste conjunto) consiste em encontrar um caminho menos custoso para visitar todas as cidades do conjunto e retornar ao ponto inicial, cidade de partida. No PCV original, os custos de viagem entre as cidades são *simétricos*, ou seja, realizar a viagem de uma cidade A para uma cidade B custa exatamente o mesmo que realizar a viagem de B para A. Quando estes custos são diferentes, dizemos que temos um problema *assimétrico*. Este, em geral, é mais difícil de resolver, de acordo com [9].

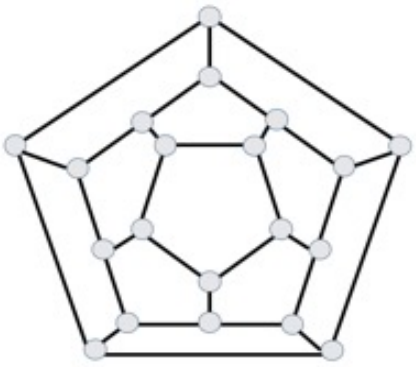

Figura 2.3: Grafo dodecaedro regular.

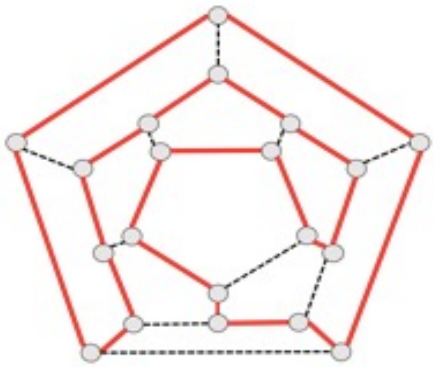

Figura 2.4: Um Ciclo Hamiltoniano.

O PCV é um dos problemas mais estudados na área da matemática computacional e, ainda que seu enunciado seja simples, não há um algoritmo eficiente de solução para seu caso geral. A complexidade da solução ótima deste problema ainda não é conhecida, apesar de vir sendo estudado por mais de cinquenta anos. Soluções para casos específicos vem sendo desenvolvidas nas mais diversas áreas da otimização

combinatória.

Em [17], o PCV é apresentado como a permutação de n cidades, o que possibilita $n!$ permutações diferentes. Para o PCV simétrico, cada rota possui duas maneiras de representação. Por esse motivo, o tamanho do seu espaço de busca é

$$S = \frac{n!}{2n} = \frac{(n-1)!}{2}$$

É simples calcularmos o número de diferentes viagens entre as n cidades do conjunto. Dada a cidade inicial, temos $n-1$ possíveis escolhas para a segunda cidade, $n-2$ possíveis escolhas para a terceira cidade, e assim por diante. Ao multiplicarmos essas possíveis escolhas, temos

$$(n-1)! = (n-1) * (n-2) * (n-3) * \ldots * 3 * 2 * 1.$$

Desde que o custo entre as cidades não dependa da direção da viagem, devemos dividir este número por 2, para calcularmos apenas uma direção de viagem, obtendo então $\frac{(n-1)!}{2}$. Como este valor cresce muito rapidamente à medida em que n aumenta, é descartada a possibilidade de conferir individualmente todas as viagens.

O PCV pode ser reescrito em um significativo número de variações. A motivação de cada variação decorre, na maioria dos casos, de aplicações práticas. Normalmente, estas variações possuem ligações com outros problemas de otimização combinatória. Não há uma sistematização espontânea na literatura; sendo assim, uma variação pode ser nomeada de diferentes maneiras.

Este projeto tem por objetivo estudar duas variações do PCV (estas serão apresentadas nos capítulos seguintes) existentes na literatura, detalhando-as e aproximando-as de aplicações do dia-a-dia.

2.5 Algoritmos aproximativos e heurísticos

2.5.1 Algoritmos aproximativos

Uma definição informal de algoritmo aproximativo é um algoritmo para um problema de otimização que provê uma solução obtida em tempo viável, ou seja, em tempo polinomial, cuja qualidade não difere muito (substancialmente) da qualidade da solução ótima. Os algoritmos aproximativos (especialmente os determinísticos aproximativos) foram introduzidos por [18], e são algoritmos polinomiais que buscam reduzir no mínimo possível a qualidade da solução obtida por outros métodos, e ao mesmo tempo ganhando o máximo possível em eficiência (tempo polinomial). Em [19], é observado que a busca do equilíbrio entre estas situações conflitantes é o grande paradigma dos algoritmos aproximativos.

Em um período anterior ao surgimento dos algoritmos aproximativos, a análise do desempenho dos métodos heurísticos era baseada na execução de um conjunto finito de instâncias. Em seguida, os resultados da heurística eram comparados com os resultados de outras heurísticas para o mesmo conjunto de instâncias.

Em [20], é apresentada a definição de algoritmos aproximativos. Consideremos um problema de otimização, onde $val(S) \geq 0$ para toda solução viável S e para qualquer instância do problema. Sejam A

um algoritmo e I uma instância viável para o problema, tal que A resulta em uma solução viável $A(I)$ de I, para toda instância I do problema. Caso o problema seja de minimização e

$$opt(I) \leq val(A(I)) \leq \alpha opt(I)$$

para toda instância I, pode-se dizer então que A é uma α-aproximação para o problema em questão. O termo α é um valor que pode depender de I. Dessa maneira, diz-se que α é uma razão de aproximação (*approximation factor*) do algoritmo. Como o exemplo acima é de um problema de minimização, temos que $\alpha \geq 1$. Se o problema for de maximização, deve-se refazer a definição com

$$\alpha\ opt(I) \leq val(A(I)) \leq opt(I)$$

Neste caso, temos que $0 < \alpha \leq 1$.

Em [20], ainda é definido que "um algoritmo de aproximação é uma α-aproximação para algum α. Uma $1 - aproximacao$ para um problema de otimização é um algoritmo exato para o problema". Pode-se observar então que um algoritmo A é uma α-aproximação para determinado problema de minimização (ou maximização) se α for uma delimitação superior (ou inferior) para a razão entre o $val(A(I))$ e o $opt(I)$ para uma instância qualquer I do problema. Ainda em [20], temos também que, a cada instância I de um determinado problema, associa-se um número natural $\langle I \rangle$, que é definido como tamanho da instância. Uma possível analogia é de que instâncias e soluções viáveis são cadeias de caracteres. Dessa forma, $\langle I \rangle$ pode ser entendido como sendo o comprimento da cadeia de caracteres I.

Por fim, um algoritmo A para o problema é polinomial caso exista um polinômio p onde o consumo de tempo do algoritmo é delimitado por $p(\langle I \rangle)$ para cada instância I. O autor [20] conclui então que "o conceito de algoritmos aproximativos polinomiais deve ser entendido como uma formalização da ideia de algoritmo eficiente".

2.5.2 Algoritmos heurísticos

O conceito de heurísticas é definido em [21] como a "arte de inventar". Dentro da programação, diz-se que um algoritmo é heurístico quando a sua solução não é determinada de forma direta, mas através de testes. Este método consiste em gerar candidatos a possíveis soluções de acordo com um dado padrão. Em seguida, os candidatos são submetidos a testes de acordo com um critério que caracteriza a solução. Caso um candidato não seja aceito, gera-se outro e os passos ou medidas tomadas com o candidato anterior não são considerados. Ou seja, existe um retrocesso para começar a se gerar um novo candidato. Os algoritmos baseados na técnica de *backtracking* partem deste princípio.

Segundo [22], algoritmos heurísticos são uma boa escolha quando há instâncias muito grandes do problema, ou quando há pouco tempo disponível para a resolução. Os métodos heurísticos são, por diversas vezes, definidos como *busca cega*, por não envolverem implementações de conhecimentos especializados. Por exemplo, um método heurístico para a solução de uma equação de segundo grau não utilizaria a fórmula de Bhaskara. Assim, o objetivo destes métodos não é necessariamente obter uma solução ótima do problema. Como é apresentado em [23], toma-se como ponto de partida uma

solução viável e então realizam-se sucessivas aproximações direcionadas a um ponto ótimo. Ao final deste processo, obtêm-se as melhores soluções possíveis para os problemas, e não necessariamente soluções ótimas, exatas e definitivas.

Portanto, metódos heurísticos podem ser resumidos como uma busca empírica e contínua, que encontram vários ótimos locais, onde o resultado é o melhor que se pode encontrar mediante determinadas condições.

Capítulo 3

O problema do Caixeiro Viajante Comprador

3.1 Introdução

Sgundo [24], o Problema do Caixeiro Viajante Comprador (PCVc) – conhecido como *Traveling Purchaser Problem (TPP)*, atraiu a atenção de pesquisadores das áreas de otimização combinatória e profissionais, devido a sua dupla natureza de problema de aquisição e de transporte. O PCVc consiste em um agente (caixeiro comprador) que deve visitar um conjunto de pontos de venda, ou lojas, a fim de satisfazer a exigência de demanda de custo mínimo para determinados produtos. O custo é constituído por dois elementos: o custo de viagem entre dois pontos de venda e o custo de compra de cada produto. Em [25], o PCVc é citado como um problema comum, enfrentado por compradores em geral, mas que também possui aplicações na área de planejamento (ou agendamento) de produção.

No ano de 1981, foi apresentada a primeira generalização do PCV, correspondente ao PCVc. A versão não direcionada deste problema foi definida por [26]. Uma definição completa é apresentada em [9]:

Definicão 6. *[9] Considerando uma residência v_0, um conjunto de mercados $M = (v_1, v_2, \ldots, v_n)$ e um conjunto de produtos $K = (f_1, f_2, \ldots, f_n)$, o Caixeiro Comprador pode ser representado por um grafo $G = (V, E)$ não direcionado simples, tal que $V = \{v_0\} \cup M$ é o conjunto de vértices e $E = \{(v_i, v_j) : v_i, v_j \in V, i < j\}$, o conjunto de arestas. Todos os produtos f_k são associados a uma demanda d_k existente em um subconjunto de mercados $M_k \subseteq M$, sendo b_{ki} o preço do produto f_k no mercado v_i e c_{ij} o custo de viagem entre v_i e v_j.*

O problema consiste, então, em determinar um caminho em G começando e terminando em v_0, passando pelos vértices necessários para adquirir os produtos $f_k, k = 1, \ldots, n$, que atenderão à procura d_k, minimizando o custo de compra dos produtos juntamente com o custo do deslocamento no grafo. Nesta variante, não é necessária a visita a todos os vértices. Em [9], é apresentado um exemplo onde um mercado não é inserido no ciclo final do caixeiro comprador, como ilustrado na Figura 3.2.

A partir destas definições, temos por q_{ki} a quantidade do produto f_k disponível no mercado v_i. Sendo assim, existem duas situações possíveis:

1. $d_k \geq q_{ki} > 0$, sendo $d_k \geq 1, \forall\, \mathrm{f}_k \in K$ e $\forall\, \mathrm{v}_i \in P_k$ – denominando *Problema do Caixeiro Comprador Capacitado*, onde a demanda pode ser satisfeita após a visita a mais de um mercado, já que $d_k \geq q_{ki}$.

2. $d_k = q_{ki} = 1$ – denominando *Problema do Caixeiro Comprador Não-Capacitado*, onde supõe-se que, se um produto estiver disponível em um dado mercado, sua quantidade será suficiente para satisfazer à demanda. Nesse caso, a demanda é de uma unidade por produto.

A Figura 3.1 [9] ilustra um exemplo do PCVc, com o custo correspondente a cada par de vértices.

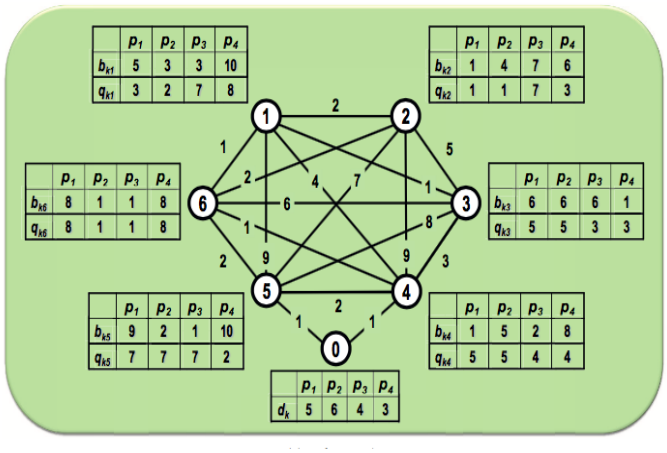

Figura 3.1: Ciclo do Problema do Caixeiro Comprador.

Fonte: [9].

As tabelas apresentadas juntamente aos vértices apresentam as informações relacionadas à demanda d_{ki}, produto p_i, preço de cada produto b_{ki} e a quantidade disponível do produto q_{ki} no mercado, ou vértice, correspondente. A tabela referente ao vértice 0 corresponde à demanda dos produtos no início do problema, ou seja, o total de cada produto que deverá ter sido adquirido ao fim do ciclo de compra.

A partir do grafo apresentado na Figura 3.1, inicia-se o cálculo para obter a solução ótima, combinando os menores custos de viagem e os menores custos individuais dos produtos. Na organização do PCVc apresentada na Figura 3.1, serão realizadas as seguintes compras:

Produto	Quantidade	Vértice	Custo Unitário	Custo
P_1	5	4	1	5
P_2	1	6	1	1
P_2	2	1	3	6
P_2	3	4	5	15
P_3	4	5	1	4
P_4	3	3	1	3
			Custo total	34

Tabela 3.1: Cálculo para a solução do PCVc.

Por sua vez, o custo total do ciclo é calculado pela soma dos custos de viagens entre as cidades. Ou seja:

Vértice de Partida	Vértice de Destino	Custo
V_0	V_4	1
V_4	V_3	3
V_3	V_1	1
V_1	V_6	1
V_6	V_5	2
V_5	V_0	1
	Custo total	9

Tabela 3.2: Cálculo para o custo de viagem da Solução do PCVc.

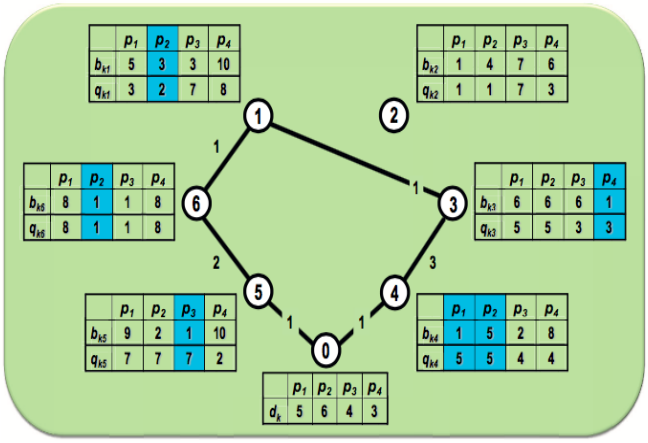

Figura 3.2: Solução do Problema do Caixeiro Comprador.

Fonte: [9].

Portanto, ao somar os custos totais apresentados nas Tabelas 3.1 e 3.2 obtém-se o custo total da

solução ótima para o problema - 43, onde o custo total de viagem é 9 e o custo total de compra é 34. A Figura 3.2, extraída de [9] apresenta a solução ótima do problema, informando quais vértices foram visitados e quais produtos foram comprados em cada um.

3.2 NP-dificuldade do problema

Segundo [24], o maior interesse no PCVc se dá pelo fato de que ele desafiadoramente combina a seleção do fornecedor, a construção de roteamento e o planejamento de compra de produtos em busca de uma única solução. Ainda de acordo com [24], encontrar uma solução ótima para cada um dos subproblemas gerados não garante que alcancemos uma solução ótima para o PCVc.

As aplicações mais comuns para o PCVc são relativas a roteamento de veículos e armazenamento. O PCVc é NP-Díficil, uma vez que ele generaliza o PCV original e o Problema de Localização de Instalações Não-Capacitado (*Uncapacitated Facility Location Problem* - UFLP), que, segundo [28], é um problema bastante estudado na literatura, envolvendo custos fixos para a localização de instalações e custos de produção, transporte e distribuição, para satisfazer a demanda de determinadas regiões. O objetivo do UFLP é o de decidir onde localizar as instalações dos seus clientes, de forma a minimizar o custo total.

De acordo com [24], a afirmação acima pode ser comprovada a partir de duas reduções:

1. O PCV corresponde a um PCVc (variação do problema onde cada produto está disponível apenas em um único mercado, e cada mercado vende apenas um produto).

2. O UFLP pode ser entendido como um PCVc irrestrito, onde cada potencial local de instalação corresponde a um mercado, e cada cliente corresponde a um produto.

3.3 Resultados obtidos por diferentes abordagens

Esta seção apresenta os principais algoritmos e heurísticas disponíveis para o PCVc. A partir destes estudos, novas soluções e heurísticas vem sendo desenvolvidas para o problema. Segundo [24], todas as heurísticas construtivas para o PCVc se baseiam no conceito de poupar ou economizar, ou seja, buscar uma melhoria relacionada à diminuição total dos custos de compra, associada ao possível aumento dos custos de viagem, ao inserir um novo fornecedor em uma solução.

3.3.1 Ramesh (1981)

Este algoritmo foi formulado a partir da ideia de busca lexicográfica. É criada uma tabela contendo um alfabeto que representa todas as sequências possíveis de mercados. Em [29], o autor afirma que "o algoritmo procura por cada possível palavra (sequência de mercados) de acordo com a tabela do alfabeto e compara se a sequência corrente de mercados constitui uma solução praticável".

Por ser um dos primeiros algoritmos desenvolvidos para este problema, este algoritmo, em um primeiro momento, pode ser definido como um algoritmo exato; porém, funciona somente com um conjunto

muito pequeno de instâncias. Dessa forma, uma heurística pode ser derivada a partir deste algoritmo, interrompendo a busca na primeira solução viável. Em [26], o autor considera uma mudança no algoritmo, a fim de melhorar a solução obtida. Esta mudança consiste em examinar todos os mercados até que a solução seja melhor do que a solução corrente.

O algoritmo desenvolvido por Ramesh serviu de base para dois outros algoritmos desenvolvidos por [30], o *Algoritmo do vizinho-mais-próximo* (Next-Neighbor Search Algorithm) e o *Algoritmo do bloco-mais-próximo* (Next-Bloc Search Algorithm - NB-SH). O algoritmo NB-SH realiza a busca lexicográfica por blocos de palavras, ou seja, blocos de mercados, buscando a melhor solução do bloco atual para depois mover-se para o próximo bloco. Esse procedimento é repetido até que todos os blocos sejam percorridos e uma solução ótima seja alcançada. Por sua vez, o algoritmo do vizinho-mais-próximo inicia-se com o vizinho mais próximo ao vértice v_0, realiza-se a busca lexicográfica até que a melhor solução seja encontrada. Em seguida, visita-se o segundo vizinho-mais-próximo de v_0 e busca-se, da mesma forma, a melhor solução possível. O procedimento é realizado até que todos os vizinhos sejam avaliados.

3.3.2 Heurística de Economia Generalizada (1981)

A *Heurística de Economia Generalizada* (Generalized Saving Heuristic) foi desenvolvida por [31] e é um dos primeiros trabalhos com abordagem de algoritmos heurísticos. Esta heurística tem seu ponto de partida em um ciclo composto pelo ponto de partida v_0 e todos os mercados que possuírem o maior número de produtos com o menor custo. Em seguida, substitui-se no ciclo o mercado que viabilizar uma menor soma dos custos dos produtos. A solução ótima é alcançada quando não houver outra melhoria, ou economia, através da adição de novos mercados ao ciclo.

Em 1982, foi publicado um trabalho [32] com uma modificação desta heurística, de forma que a configuração inicial passou a ser composta por um conjunto de mercados que tenham os produtos requeridos pelo problema. A partir disso, os mercados que oferecem um custo mais elevado são retirados iterativamente até que não se possa realizar melhorias na solução obtida.

3.3.3 Metaheurística baseada em Busca Tabu (1996)

No ano de 1996, foram criados dois procedimentos heurísticos [33] que são utilizados na criação da solução inicial do PCVc - o *DROP-procedure* e o *ADD-procedure*. O primeiro inicia-se com uma possível solução contendo todos os mercados; o procedimento então passa a remover, a cada iteração, o mercado que resulte na melhor redução, de acordo com o objetivo da problema. O procedimento *ADD-procedure* funciona de maneira oposta; ele cria um caminho adicionando os mercados, respeitando o critério de economia definido (economia de produtos ou economia de viagem). Após a criação de um caminho viável, os demais mercados são adicionados de forma a não restarem melhorias possíveis para a solução do problema.

3.3.4 Algoritmo de Singh & Van Oudheusden (1997)

Nesta abordagem, os autores propõem um algoritmo de *branch-and-bound* que, como diz o nome, possui a ideia de dividir todos os caminhos possíveis em subconjuntos menores e calcular então os limites inferiores, levando sempre em conta a soma da compra dos produtos e a soma da viagem. O cálculo do limite inferior é realizado através do UFLP. Este algoritmo pode ser aplicado a instâncias assimétricas e simétricas, como é definido em [22].

3.3.5 Heurística de Adição de Mercadorias (1998)

Outra heurística proposta na literatura [30] é a *Heurística de Adição de Mercadorias* (Commodity Adding Heuristic - CAH). Nesta heurística, é definido que todos os mercados possuem todos os produtos existentes na configuração do problema. O procedimento realizado para a obtenção de uma solução cada vez mais próxima da ótima leva em consideração apenas o custo do produto, sem incluir no cálculo o custo da viagem.

O primeiro passo é criar uma lista com todos os produtos e, a partir do primeiro produto, construir uma solução que possua o menor custo para o mesmo. As iterações seguintes realizam o mesmo procedimento, selecionando os produtos seguintes, sempre preservando o menor custo. Os mercados podem ser adicionados a cada iteração, ou não, caso já tenham sido visitados.

3.3.6 Heurística de Adição de Mercado (2003)

Esta heurística foi desenvolvida por Laporte [34], e consiste em um algoritmo exato que utiliza como base o algoritmo de *branch-and-cut*, podendo ser aplicado ao PCVc capacitado e ao PCVc não-capacitado.

A etapa inicial desta solução utiliza *heurística de adição de mercados* (MAH), que consiste em acrescentar, a cada iteração, um mercado que tenha disponibilidade de um produto que ainda não tenha sua demanda atendida até o ciclo atual. Em seguida, é criado um conjunto de mercados que disponibilizam determinado produto pelo seu menor custo. Então, é adicionado na solução o mercado que apresentar o maior preço entre os menores, como é apresentado em [29].

3.3.7 Pré-Processamento e Intensificação de Teeninga & Volgenant (2004)

Em [35], são apresentados procedimentos que melhoram a qualidade de algumas heurísticas previamente propostas: *Heurística de Economia Generalizada*, *Heurística de Adição de Mercadorias* e *Heurística de Redução de Caminho*. O procedimento de pré-processamento, como definido em [29], serve para "identificar mercados que podem ser ótimos. Essa informação é utilizada nas rotinas de inicialização dos outros algoritmos". Por sua vez, o processo de intensificação é realizado sobre a ideia de que, ao se retirar um mercado somente da solução, pode não ocorrer redução no custo total; porém, a retirada de k mercados consecutivos pode resultar em uma redução considerável no custo total. [29] ainda afirma que "a redução de k mercados e a sua re-introdução na solução também pode gerar uma redução no custo

total". No trabalho proposto por [35], o autor afirma que este procedimento, mesmo aumentando o tempo computacional utilizado, faz com que a qualidade da solução seja melhorada.

3.3.8 Busca Local (2005)

A aproximação heurística de Busca Local proposta em [36] pode ser aplicada para o PCVc capacitado e não-capacitado. O primeiro passo desta heurística consiste em criar uma solução que contenha todos os mercados, utilizando o algoritmo do vizinho-mais-próximo. Em seguida, a solução criada é melhorada, aplicando-se o algoritmo de Lin-Kernighan, proposto em [37]. Após estes dois passos, a solução passa por uma busca local aplicada a dois grupos de vizinhança e, por fim, um procedimento de diversificação chamado *shaking*, que serve para diversificar uma solução quando não há mais melhorias na etapa de busca local, é aplicado.

Em [29], Bagi define o procedimento de *l-ConsecutiveExchange* como um "esquema iterativo que troca l vértices consecutivos em um ciclo viável por um conjunto de vértices que não estão nesse ciclo". Quando um ótimo local é encontrado, o procedimento reduz o valor de l, o procedimento realiza trocas de conjuntos de l mercados da solução inicial por outros que estão fora do ciclo. Neste procedimento, há duas etapas. A primeira, chamada de *l-ConsecutiveDrop*, é realizada a fim de diminuir o tamanho do ciclo ao retirar l mercados consecutivos. A segunda etapa, definida como *RestoreFeasibility*, tem a função de recuperar a viabilidade da solução, caso ela seja perdida no estágio anterior. Já o procedimento de *Insertion* adiciona todos os possíveis vértices que resultem em uma diminuição no custo objetivo.

3.3.9 Metaheurística baseada em Colônia de Formigas (2006)

Em [38], é proposta uma metaheurística que possui como base a ideia da otimização por colônia de formigas, que é inspirado no comportamento de formigas, que se movem de forma aleatória, explorando o espaço ao redor do formigueiro a fim de buscar alimento. Quando encontram alimento, as formigas retornam ao formigueiro, deixando pelo caminho um rastro de feromônio. Quanto mais feromônio a formiga deposita no caminho, maior a quantidade de formigas que realizaram este caminho até o alimento. Ou seja, a chance de este ser o melhor (ou menor) caminho é intensificada. Dessa maneira, este caminho torna-se uma solução otimizada, levando em consideração o nível de feromônio presente no caminho. Como é apresentado em [39], muitos estudos concluíram que as formigas percorrem o caminho que possui maior nível de feromônio, e que este era, por sua vez, o caminho mais curto.

O algoritmo de colônia de formigas foi proposto por [40] em 1992. O modelo natural de caminho de feromônio de insetos deu início a um sistema artificial. As formigas artificiais são definidas como agentes. A análise do comportamento das formigas artificiais foi baseada no modelo do comportamento das formigas reais.

Para o PCVc, o algoritmo proposto por [38] é definido como *Dynamic Multi-Dimensional Anamorphic Traveling Ant* (DMD-ATA), que é uma evolução do algoritmo original de colônia de formigas adaptado para o PCVc não capacitado. Neste algoritmo, um agente, ou seja, uma formiga, move-se de um mercado a outro em um grafo do PCVc, como é definido em [29]. O autor ainda afirma que "após co-

letar todos os produtos, a formiga não retorna para o depósito imediatamente; ela tem a possibilidade de visitar mais mercados com uma probabilidade sendo diminuída". No fim do percurso, os agentes deixam nos caminhos percorridos um nível de feromônio que é proporcional à qualidade da solução ótima.

As melhorias encontradas pelos agentes também são armazenadas, para que a melhoria passe a fazer parte da solução. Um coeficiente de evaporação é aplicado ao caminho anterior para reduzir a quantidade de feromônio depositada no caminho.

Por fim, [29] afirma que "para conseguir melhor performance, são utilizados procedimentos de busca local". Ou seja, ao fim do processo de otimização baseado em colônia de formigas, é realizado um processo de intensificação utilizando busca local. Em [38], é definido que o algoritmo DMD-ATA somente é aplicado ao PCVc não capacitado, e então o autor compara seus resultados com o algoritmo de busca local de [36]. Os algoritmo DMD-ATA apresentam resultados melhores que o algoritmo de busca local simples. Porém, mesmo sendo realizado em uma máquina com capacidade computacional maior, ainda apresenta um tempo de processamento muito maior que o algoritmo de busca local simples.

3.4 Heurísticas MAH e Busca Local

Neste trabalho, serão estudadas e comparadas duas heurísticas presentes na literatura: a *Heurística de Adição de Mercado*, desenvolvida por Laporte em [34] e definida em seu idioma original como MAH - Marketing Adding Heuristic e a *Heurística de Busca Local* desenvolvida por [36]. A escolha destas heurísticas deu-se pelo fato de serem as duas heurísticas mais estudadas para essa variante, além da melhoria e da importância que elas carregam.

O algoritmo de Branch-and-Cut, utilizado dentro da heurística de MAH - Heurística de Adição de Mercados, é a heurística que apresentou mais melhorias comparadas às heurísticas anteriores. Já a heurística de Busca-Local, mesmo não sendo a mais recente, é a heurística mais consistente dentre os resultados obtidos. Uma prova disso é que os trabalhos mais recentes da área, que envolvem colônia de formigas e algoritmos transgenéticos, realizam uma etapa chamada de intensificação com busca-local para obter soluções mais otimizadas, e ainda assim os resultados apresentam maior tempo computacional até chegarem à solução ótima.

MAH - Heuristica de Adição de Mercados

Em [34], é descrita a *Heurística de adição de mercados* (MAH) para o PCVc restrito. Esta heurística gradualmente estende um ciclo, inserindo em cada etapa um novo fornecedor, que vende um produto cuja demanda ainda não está totalmente atendida.

Inicialmente, a heurística MAH determina em quais fornecedores cada produto está disponível a preços mais baixos, e entre estes mínimos, acrescenta ao passeio o fornecedor que apresente o preço mais elevado do produto, de acordo com uma regra de economia máxima definida (custo dos produtos ou custo da viagem).

Por esta heurística ser comumente utilizada, sua ênfase tem sido aplicada no ganho de velocidade. Porém, ainda em [34], são apresentados resultados computacionais em que a heurística demonstra um

desempenho empírico satisfatório quando os custos de roteamento são baseados em distâncias euclidianas, baseadas nas medidas de um sistema de coordenadas cartesianas. Os resultados do autor comprovam ainda que o método estende progressivamente um ciclo, inserindo em cada passo um novo mercado v_{i^*} vendendo um produto p_{k^*} cujas necessidades de compra ainda não estão totalmente satisfeitas. Denotando por $\overline{K}$ o conjunto de tais produtos e por $\overline{M}$ o conjunto de mercados não visitados, o próximo mercado v_{i^*} a ser inserido é o que produz
$b_{k^*i^*} = \max_{p_k \in \overline{K}}\{\min_{v_i \in \overline{M} \cap M_k} b_{ki}\}$.

A partir da equação acima, [34] enfatiza que a inclusão de v_{i^*} no ciclo atual é realizada de acordo com o critério padrão de máxima economia. Uma vez obtido um ciclo viável, a etapa seguinte consiste em um pós-otimização realizada iterativamente, agindo sobre o conjunto de fornecedores na solução e do custo da viagem.

Além disso, dado um conjunto $\overline{M}$ de mercados não visitados, o custo de roteamento pode ser melhorado através da solução de um PCV no subgrafo gerado por $\overline{M}$. Na solução proposta por [34], é utilizado, na implementação, um mecanismo de 2-*opt* simples, que é o método de busca mais simples para o problema do caixeiro viajante, sendo a vizinhança da solução S definida pelo conjunto de soluções que podem ser alcançadas através da permutação de duas arestas não-adjacentes em S, segundo [41].

Somado a isso, no procedimento de pós-otimização, o conjunto de mercados visitados é modificado dinamicamente, retirando um mercado de $\overline{M}$, introduzindo um novo mercado em $\backslash M$ ou trocando um mercado de $\overline{M}$ com um mercado de $M \backslash \overline{M}$, enquanto a viabilidade da solução é mantida. Por fim, o autor [34] afirma que a rentabilidade de uma determinada seleção de mercado é avaliada pela aplicação do algoritmo UFLP e do mecanismo de 2-*opt*.

Algoritmo de Branch-and-cut para solucionar o Problema do Caixeiro Comprador - Para a Heurística MAH

O algoritmo de *Branch-and-cut*, proposto em [34], possui o número de restrições geradas pelo algoritmo crescendo exponencialmente em relação ao número de mercados, n, e um número grande de variáveis, em parte devido à presença de variáveis z_{ki}, que representam a quantidade do produto p_k comprado no mercado v_i para todos os produtos $p_k \in K$ e todos os vértices $v_i \in M_k$.

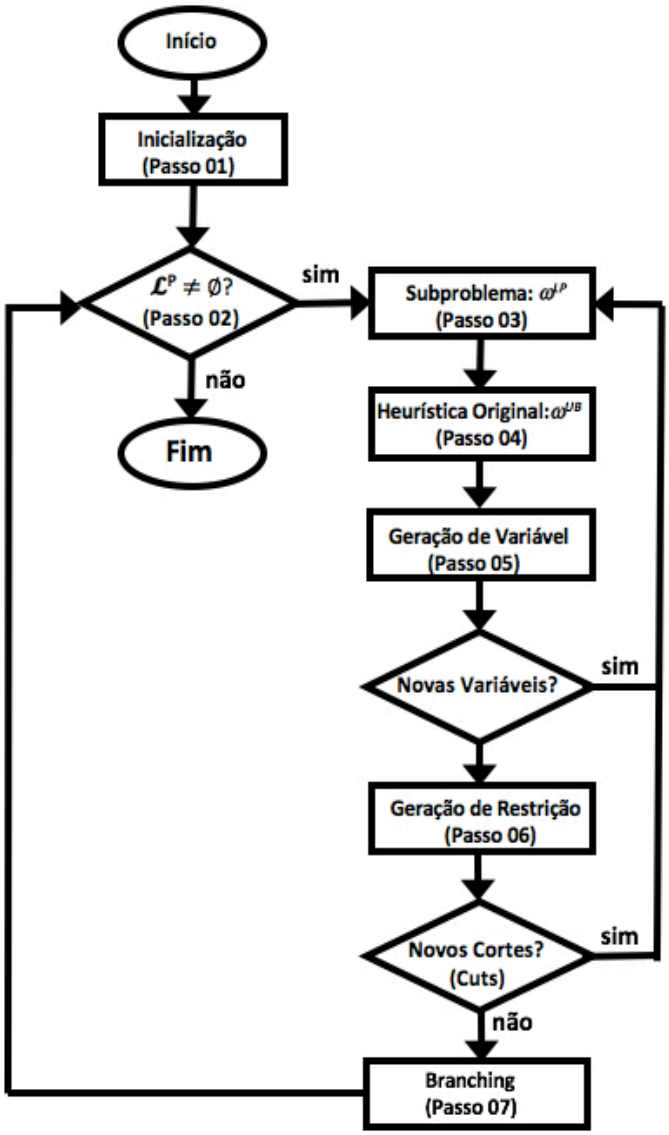

Figura 3.3: Fluxograma do Algoritmo de Branch-and-cut de Laporte.

A Figura 3.3 demonstra graficamente o passo-a-passo do algoritmo desenvolvido em [34]. Algumas definições serão apresentadas a seguir, seguindo o trabalho proposto por [34], para facilitar o entendimento dos passos do algoritmo. São elas:

$$M^* := \{v_0\} \cup \left\{ \; v_i \in M : \text{existe} \quad p_k \in K \text{ tal que} \quad \textstyle\sum_{v_j \in M_k \setminus \{v_i\}} q_{kj} < d_k. \; \right\}$$

Ou seja, M^* representa o conjunto de vértices, ou mercados, que devem fazer parte de qualquer solução do PCVc.

$$K^* := \left\{ \; p_k \in K : \textstyle\sum_{v_i \in M_k} q_{kj} = d_k \; \right\}$$

K^* representa o conjunto de produtos sem as opções de decisão de mercado. Para essas definições, o autor usa três tipos de variáveis de decisão:

$$x_e := \begin{cases} 1, & \text{se a aresta e pertence à solução} \\ 0, & \text{em caso contrário} \end{cases} \quad \forall \text{e} \in E$$

sendo E o conjunto de arestas,

$$y_i := \begin{cases} 1, & \text{se o vértice } v_i \text{ pertence à solução} \\ 0, & \text{em caso contrário} \end{cases} \quad \forall v_i \in V$$

sendo V o conjunto de vértices e por fim,

z_{ki} := a quantidade de produtos p_k comprados no mercado v_i para todo $p_k \in K$ e todo $v_i \in M_k$.

A partir do fluxograma e das definições apresentadas acima, apresenta-se o passo-a-passo do algoritmo:

1. **Passo 1 (Inicialização) :**

   I - Calcular o conjunto de mercados M^* e o conjunto de produtos K^*, definir $y_i = 1$ para todo vértice $v_i \in M^*$, e $z_{ki} = q_k$ para todo $p_k \in K^*$ e $v_i \in M_k$.

   II - Calcular um limite superior w^{UB} no valor da solução ótima do PCVc w^{OPT} por meio da heurística MAH.

   III - Definir L^V, um subconjunto de vértices, incluindo todas as variáveis x_e associadas às 10 arestas de menor custo incidentes a cada vértice, e variáveis z_{ki} correspondentes aos 10 mercados mais baratos para cada produto.

   IV - Definir L_C, incluindo todas as restrições com $S := M_k$ para todo p_k, bem como os limites inferior e superior das variáveis.

   V - Definir um primeiro PL (programa linear), incluindo todas as variáveis y_i, todas as variáveis x_e com arestas pertencentes à solução heurística e as 5 arestas de menor custo incidentes a cada vértice, e todas as variáveis z_{ki} correspondentes às atribuições feitas na solução heurística e aos 5 mercados mais baratos para cada um dos produtos. Por fim, inicialize uma lista L^P de subproblemas com este PL.

2. **Passo 2 (Teste de Término) :**

   I - Se L^P estiver vazio, parar com a solução atual.

   II - Caso contrário, selecionar um subproblema de L^P, de acordo com a melhor estratégia de limite inferior, ou seja, o menor limite inferior.

   III - Definir o contador de iteração = 0.

3. **Passo 3 (Solução do Subproblema) :**

   I - Definir $\theta = \theta + 1$. O autor [34] neste passo utiliza o CPLEX 6.0 para solucionar o PL e armazenar o valor da solução em w^{PL}.

II - Remover permanentemente do PL e de todos os descendentes do subproblema atual as variáveis não-básicas para as quais o produto de seu limite superior e seu custo reduzido ultrapasse $w^{UB} - w^{PL}$.

III - Remover temporariamente todas as variáveis não-básicas cujos custos reduzidos ultrapassem um limite igual a 0.001, e todas as restrições que estiverem inativas durante 5 iterações ou cuja variável não-utilizada ultrapasse um limite igual a 0.01.

IV - Se a solução não for um múltiplo de 5, ir para o Passo 6.

4. **Passo 4 (Heurística Original) :**

I - Tentar aprimorar a solução atual, estabelecida, aplicando o procedimento heurístico do Passo 1, utilizando a solução PL atual como ponto de partida.

5. **Passo 5 (Geração de Variáveis) :**

I - Pesquisar L^V para variáveis com custo reduzido negativo. Se nenhum deles puder ser identificado, realizar uma busca similar sobre todas as variáveis. Se alguma variável for identificada, adicioná-las a L^V. Se nenhuma variável de custo reduzido negativo for identificada, ir para o Passo 6.

II - Caso contrário, adicionar todas as variáveis identificadas ao PL e ir para o Passo 4.

6. **Passo 6 (Geração de Restrição) :**

I - Pesquisar L^C por restrições violadas. Se nenhuma puder ser identificada, executar um procedimento de separação; se forem detectadas quaisquer restrições violadas, adicioná-las a L^C. Se nenhuma restrição violada tiver sido identificada e o Passo 6 tiver sido iniciado logo após o término do Passo 3, então ir para o Passo 5.

II - Caso contrário, se forem identificadas restrições violadas, adicionar as 50 restrições mais violadas ao PL e ir para o Passo 3.

7. **Passo 7 (Branching - Ramificações) :**

I - Se $w^{PL} \geq w^{UB}$, ir para o Passo 2.

II - Caso contrário, se a solução for um conjunto inteiro $w^{UB} := w^{PL}$, atualizar a melhor solução conhecida e ir para o Passo 3.

III - Criar dois subproblemas, ramificando uma variável fracionária, inserindo-os na lista L^P e ir para o Passo 3.

Resultados do Algoritmo de Branch-and-Cut de Laporte

Em [7], os autores utilizaram 4 classes de instâncias de teste:

Classe 1 Contém 33 instâncias simétricas de mercados, definidas com a mesma entrada de dados, definidos por [42] no algoritmo de branch-and-bound, descrito na Subseção 3.3.5.

Classe 2 As instâncias são geradas aleatoriamente, usando o processo descrito em [30] na heurística de adição de mercadorias. Os autores utilizaram o custo de roteamento simétrico ao invés de custos assimétricos. As instâncias foram definidas com $|V| = 50, 100, 150$ e 200, e $|K| = 50, 100, 150$ e 200.

Classe 3 As instâncias foram definidas gerando a princípio $|V|$ vértices de coordenadas inteiras no quadrado $[0.1000] \times [0.1000]$, de acordo com uma distribuição uniforme e definindo custos de roteamento por distâncias euclidianas. Cada produto p_k foi associado com $|M_k|$ mercados selecionados aleatoriamente, onde $|M_k|$ foi gerado aleatoriamente em $[1, |V| - 1]$. As características restantes destas instâncias são definidas da mesma forma que na Classe 1.

Foram utilizadas cincos instâncias para cada valor de $|V|$ e $|K|$ para analisar a performance do algoritmo. Nas tabelas abaixo, são apresentados os resultados médios das cincos instâncias, para cada valor de $|K|$, encontrados em [34]. Para cada grupo de instâncias, os autores fornecem as estatísticas Visitados, Total sec, Nós e o valor PC. As informações contidas na tabela são definidas da seguinte maneira:

|V|: número de vértices (incluindo o mercado inicial);

|K|: número de produtos;

***Visitados*:** número de mercados visitados na solução ótima;

***Total sec*:** tempo computacional total gasto pelo código de branch-and-cut;

***Nós*:** número de nós gerados.

***PC*:** Percentual do custo dos produtos sobre o custo total de uma solução ótima.

|K|	Visitados	Total sec	Nós	PC	Visitados	Total	Nós	PC	Visitados	Total	Nós	PC
50	3.6	0.0	1.0	33.1	8.4	1.6	3.0	48.8	16.8	1.8	4.2	60.2
100	4.4	4.8	1.4	41.1	10.4	8.6	8.2	52.2	22.0	1.2	2.6	67.8
150	5.8	11.4	2.6	40.7	13.2	17.8	10.6	49.8	24.4	0.0	1.0	72.4
200	6.4	17.4	1.8	43.3	13.8	49.2	25.4	54.5	26.6	0.0	1.0	74.2

Tabela 3.3: Classe 01: |V| = 33.

|K|	Visitados	Total sec	Nós	PC	Visitados	Total	Nós	PC	Visitados	Total	Nós	PC
50	13.6	56.4	12.6	18.5	34.4	17.2	7.8	44.4	45.6	3.6	11.4	60.8
100	24.2	168.0	43.8	21.1	52.0	7.0	3.4	59.2	66.8	1.8	5.4	76.2
150	35.6	196.2	59.0	29.5	71.0	6.8	3.4	71.5	79.2	3.6	6.2	85.4
200	42.4	141.0	42.6	32.0	76.4	12.4	5.0	80.0	83.6	9.0	12.6	90.5

Tabela 3.4: Classe 02: |V| = 100.

|K|	Visitados	Total sec	Nós	PC	Visitados	Total	Nós	PC	Visitados	Total	Nós	PC
50	9.2	104.4	19.8	9.4	10.4	122.4	1.4	43.1	35.0	81.0	8.6	74.5
100	13.0	179.8	14.2	10.7	14.2	309.4	2.2	50.0	48.8	34.8	3.0	81.5
150	17.8	1,552.8	193.0	12.6	18.0	423.0	3.0	55.3	60.6	69.0	11.8	85.4
200	19.0	256.4	11.8	15.0	19.8	344.4	6.6	56.9	77.2	154.2	42.2	88.2

Tabela 3.5: Classe 03: |V| = 100.

Na coluna dos Nós, quando há o valor 1, significa que o problema não requer *branching* (ramificação).

Como pode ser observado a partir das Tabelas 3.3, 3.4 e 3.5 e afirmado por [34], a dificuldade de resolver uma instância não é diretamente afetada pela magnitude dos custos dos produtos em comparação com os custos de roteamento. Os autores também concluem que uma instância das Classes 1 e 2 tende a ser mais fácil quando os custos dos produtos são maiores do que os custos de roteamento, mesmo que uma decisão ótima envolva a seleção de mais mercados. Portanto, a dificuldade do problema apresenta maior relação com $|M|$ e $|K|$.

Heurística de Local-search - Busca Local

Em [36], os autores propõem uma abordagem heurística alternativa para a solução do PCVc na versão geral, isto é, com demandas de produtos e ofertas em mercados. A heurística proposta pelos autores é baseada em um esquema de busca local, utilizando uma definição de vizinhança especial. A vizinhança e a abordagem heurística são especialmente geradas para casos em que os custos de roteamento são baseados em distâncias euclidianas, porém [36] afirma que também poderia ser aplicado para resolver outras instâncias simétricas.

Os autores afirmam que a ideia principal da heurística proposta de busca local é baseada em duas famílias de vizinhanças, onde um procedimento específico para atingir um mínimo local é desenvolvido para cada uma delas. O primeiro procedimento apresentado por [36] executa um esquema iterativo, trocando l vértices consecutivos em um dado ciclo que é viável com um conjunto de vértices que não pertencem a esse ciclo. O valor l é reduzido assim que uma solução ótima local é alcançada. Este procedimento é definido pelos autores como $l - ConsecutiveExchange$. Já o segundo procedimento insere tantos vértices quanto forem possíveis, sempre que uma inserção resulte em uma redução no valor objetivo; os autores definem o segundo procedimento como $Insertion$.

É utilizada uma estrutura de dados para acelerar a avaliação de um inserção ou remoção de um mercado em uma solução parcial. É constatado ainda pelos autores que, toda vez que uma solução é avaliada e ocorre uma inserção ou remoção, a avaliação de uma solução modificada pode ser eficientemente recalculada utilizando uma *ad hoc* [1]. Uma solução parcial ρ tem uma representação interna consistindo de uma sequência de mercados em V^ρ e um vetor dinâmico para cada produto $p_k \in K$. Em [36], é demonstrado que cada componente desses vetores corresponde a um mercado $v_i \in V^\rho \hat{M}_k$ e contém a

[1] Problemas ad hoc são aqueles cuja solução não envolve estruturas de dados, algoritmos ou técnicas genéricas.

oferta q_{ki}, além do preço unitário b_{ki}. Estes itens são pré-classificados de acordo com o custo de compra b_{ki}. Ainda de acordo com esta estrutura de dados, a inserção de um novo mercado $v_i \in MV^\rho$ em uma solução parcial ρ levaria a uma complexidade de tempo de $O(log|V^\rho \cap M_k|)$ para cada produto $p_k \in K$.

Além disso, os autores afirmam que a avaliação do custo total de aquisição levaria a uma complexidade de tempo de $O(|V^\rho||K|)$. No caso particular do problema do caixeiro comprador irrestrito, esta complexidade seria reduzida para $O(1)$ para cada produto $p_k \in K$.

Algoritmo Geral para solucionar o Problema do Caixeiro Comprador - Heurística de Busca Local

Uma solução inicial é apresentada por [36], a qual contém todos os vértices e é construída pela clássica, e muito utilizada, heurística do vizinho-mais-próximo. Em seguida é melhorada pelo procedimento *Lin-Kernighan*, descrito por [43] como um algoritmo baseado na troca de arestas. Tais trocas possuem o objetivo de transformar um determinado percurso em um percurso de menor custo total, da mesma maneira que acontece no 2-opt. Contudo, o *Lin-Kernighan* utiliza uma estrutura de vizinhança k-opt, que consiste em uma generalização do algoritmo 2-opt, isto é, o número de arcos trocados a cada passo é variável para diferentes valores de k.

O autor ainda aponta o fato de que uma das dificuldades da abordagem k-opt apontadas na literatura é que o número de operações necessárias para realizar o teste de todas as k trocas aumenta rapidamente conforme é aumentado o número de cidades. Como é difícil saber qual valor de k deve-se usar para obter um melhor custo benefício entre o tempo de processamento e a qualidade da solução, o algoritmo *Lin-Kernighan* introduz um mecanismo exemplificado em [43] que, a cada iteração, busca o melhor valor para k, verificando as trocas que podem ser realizadas a fim de resultar num percurso de menor custo.

O processo do algoritmo de *Lin-Kernighan* pode ser repetido diversas vezes, tendo como percurso inicial um percurso T criado aleatoriamente ou por algum outro método conhecido. A Figura 3.4 apresenta um exemplo de trocas de arestas que podem ser realizadas pelo algoritmo *Lin-Kernighan*, onde o autor [43] representa o percurso por um círculo.

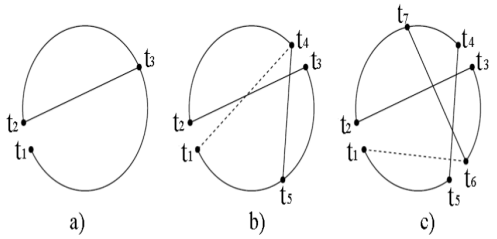

Figura 3.4: Exemplos de iterações do algoritmo *Lin-Kernighan*.

O autor ainda observa que o algoritmo *Lin-Kernighan* efetua trocas contínuas de arestas até que não seja mais possível encontrar trocas que permitam ou propiciem a obtenção de $k-trocas$ na iteração. Tendo este objetivo de melhoria da performance de seu algoritmo, oito regras foram definidas para o

algoritmo. as Regras 1 a 6 foram criadas a fim de limitar a busca do algoritmo, restringindo a escolha das arestas; as Regras 7 e 8 tem a função de guiar o processo de busca:

1 Somente trocas sequenciais são permitidas.

2 O ganho parcial G_i deve ser positivo.

3 O percurso deve resultar num ciclo.

4 Uma aresta previamente excluída não pode ser adicionada, e uma aresta previamente adicionada não pode ser excluída.

5 A busca é encerrada se o percurso encontrado for igual ao encontrado anteriormente.

6 A busca por uma aresta a ser adicionada $y_i = (t_{2i}, t_{2i+1})$ é restrita aos cinco vizinhos mais próximos do ponto atual (x_i).

7 Na escolha da aresta y_i (com $i > 2$), a cada aresta, é dada a prioridade $c(x_{i+1}) - c(y_i)$.

8 Se existem duas alternativas para x_i, é escolhida a que tiver o maior $c(x_i)$.

Após a aplicação do algoritmo de *Lin-Kernighan*, é realizado o seguinte esquema iterativo. Um loop interno calcula o valor l e obtém soluções melhores iterativamente aplicando *l-ConsecutiveExchange* e o *Insertion*, onde o *l-ConsecutiveExchange* consiste em trocar um conjunto de l vértices consecutivos, pertencentes a um ciclo viável ρ, com outros vértices fora do ciclo, em um procedimento de dois estágios:

1. O primeiro estágio é denominado como *l-ConsecutiveDrop*, e tenta reduzir o comprimento do ciclo, removendo vértices consecutivos.

2. O segundo estágio é denominado *RestoreFeasibility*, e tenta restaurar a viabilidade, se esta for perdida na etapa anterior.

Ou seja, exatamente um único vértice é removido de um ciclo viável, e um número de inserções consecutivas são realizadas enquanto uma melhoria na função objetivo é alcançada. Já o *Insertion* adiciona um novo vértice ao atual ciclo viável ρ, caso tal inserção implique em uma redução no custo total de ρ. O algoritmo desse procedimento é apresentado na Figura 3.5.

Em [36], é explicado que ambos os procedimentos são aplicados até que nenhuma melhoria adicional seja alcançada. Uma vez que o laço interno conclui, um procedimento de perturbação é realizado para aumentar o ciclo atual com novos vértices. Dessa forma, em [36], um esquema de perturbação, chamado *Shaking*, controla tanto o número de vértices adicionados como o número de iterações do loop externo. Em particular, como é realizado no algoritmo do PCVc apresentado na Figura 3.6 , para uma dada solução ρ, cada vértice não pertencente a V^ρ incrementa ρ se o custo de deslocamento da solução incrementada não aumentar em mais de ρ por cento. A porcentagem ρ é iterativamente reduzida e o procedimento para quando nenhum vértice é inserido.

Algoritmo: Procedimento *l-ConsecutiveExchange*

Entrada: Um ciclo viável σ e $1 \leq l < |V^{\sigma}|$
Saída: Um ciclo viável σ

```
1.  enquanto l ≥ 1 faça
2.      σ' := l-ConsecutiveDrop (σ, l)
3.      se σ' não é viável então
4.          σ':= RestoreFeasibility (σ')
5.      fim do se
6.      se f(σ') ≥ f(σ) ou σ' não é viável então
7.          l := l – 1
8.      senão
9.          σ := σ'
10.     fim do se
11. fim do enquanto
12. retornar σ
```

Figura 3.5: Algoritmo para o procedimento de *ConsecutiveDrop* o PCVc.

Algoritmo: Algoritmo Geral para o PCVc

Entrada: Uma instância do PCVc
Saída: Um ciclo viável σ

```
1.  σ' := Vizinho-mais-próximoPCV(V)
2.  σ := σ'
3.  repetir
4.      repetir
5.          σ':= l-ConsecutiveExchange(σ')
6.          σ':= Insertion(σ')
7.      até não apresentar melhoria
8.      se f(σ') < f(σ) então
9.          σ := σ'
10.     fim do se
11. até Shaking(σ')
12. retornar σ
```

Figura 3.6: Algoritmo geral para o PCVc.

Resultados do Algoritmo de Local Search

A proposta apresentada por Ledesma em [36] foi testada sobre a série de problemas gerados aleatoriamente, descritos por [25], contendo instâncias para as versões restritas e irrestritas do TPP. A primeira localização corresponde ao domicílio. Cada produto p_k foi associado a $|M_k|$ mercados selecionados aleatoriamente. O autor [36] ainda ressalta que, quanto maior o valor de λ, menor é o comprimento de seu ciclo ótimo. Por exemplo, com $\lambda = 0$, a instância torna-se um PCV, enquanto que, com $\lambda = 1$, teremos o PCVc irrestrito. Cinco instâncias foram geradas para cada valor de n, m e k.

As Tabelas 3.6 e 3.7 comparam os resultados do trabalho proposto por [36] com os resultados obtidos por [25] para o PCVc restrito e irrestrito.

		%Visitados	#	CAH		Busca Local	
				%gap	Segundos	%gap	Segundos
m	50	26	20	3.05	1	0.07	3
	100	16	20	1.89	9	0.14	10
	150	11	20	2.12	20	0.03	14
	200	9	18	2.59	34	0.32	19
n	50	8	25	0.35	16	0.07	5
	100	13	22	0.59	71	0.24	13
	150	16	22	0.45	225	0.10	20
	200	17	20	0.50	331	0.08	21

Tabela 3.6: Resultados computacionais médios, instâncias irrestritas (ótimas).

		%Visitados	#	CAH		Busca Local	
				%gap	Segundos	%gap	Segundos
m	50	73	140	1.42	5	0.15	5
	100	71	135	2.00	33	0.50	26
	150	73	119	2.55	113	0.50	26
	200	69	46	4.61	156	1.15	100
n	50	63	131	3.32	34	0.72	43
	100	71	117	1.83	65	0.56	38
	150	77	94	1.51	58	0.42	32
	200	80	98	1.96	83	0.29	32
λ	0.1	100	70	0.16	81	0.00	8
	0.5	99	68	0.50	96	0.20	25
	0.7	88	66	1.65	71	0.10	29
	0.8	85	68	3.02	72	0.57	62
	0.9	59	64	5.24	49	1.43	86
	0.95	39	41	3.09	11	0.68	26
	0.99	19	63	2.51	v5	0.31	19

Tabela 3.7: Resultados computacionais médios, instâncias restritas (ótimas).

A coluna CAH representa a abordagem proposta por [25] da heurística de adição de mercadoria, e a coluna Busca Local corresponde ao algoritmo de busca local descrito por [36], explicado na Subseção 3.4 deste trabalho. Cada coluna apresentada na Tabela 3.7 de [36] mostra a qualidade da solução heurística sobre a solução ótima obtida utilizando o método exato descrito em [34]. Cada linha contém

os resultados médios relacionados ao subconjunto de instâncias resolvidas para otimização pelo método exato e agrupadas de acordo com o valor m, n e k. A coluna indicada por # dá o número de instâncias envolvidas em cada linha, ou seja, o número de instâncias com uma solução ótima conhecida do método exato, desenvolvido por [34]. A coluna $\%Visitados$ mostra o número médio de mercados envolvidos em uma solução ótima calculada pelo algoritmo exato descrito em [34].

As Tabelas 3.6 e 3.7 mostram claramente que a abordagem proposta por [36] oferece soluções muito próximas das ótimas. Mesmo que o problema de cobertura do conjunto seja considerado um problema difícil, o curto tempo consumido no experimentos em [36] é explicado pelo pequeno tamanho das instâncias do subproblema que o autor soluciona. Tanto a qualidade como o tempo consumido nestas instâncias pequenas e médias não são tão dependentes de λ e n quanto de m. A qualidade é ligeiramente melhor quando se aproxima de 1, porque $\lambda = 0$ produz instâncias do PCV, enquanto $\lambda = 0.9$ produz instâncias envolvendo tanto o roteamento ótimo quanto a seleção de mercados.

As colunas $\%gap$ mostram a porcentagem máxima e média da diferença entre essa heurística e a escolha exata. Mais especificamente, o procedimento RestoreFeasibility, que seleciona um conjunto de vértices com base em estimativas da redução da função objetivo, também foi resolvido de forma otimizada usando o código branch-and-cut descrito em [34]. Observa-se a partir desta tabela que a *RestoreFeasibility* seleciona e insere novos mercados com um gap de $2,5\%$ de erro. Apesar desta lacuna, o procedimento provou ser eficaz nos experimentos de [36].

Para escolher o valor inicial do parâmetro l, bem como para provar que o *l-Consecutive-Exchange* tem um melhor desempenho que a abordagem clássica em que exatamente uma retirada, *drop*, e vários movimentos de adição são realizados, as seguintes experiências foram realizadas pelo autor:

1 Para cada instância de PCVc restrito de referência com $\lambda \in \{0.8, 0.9, 0.95, 0.99\}$, o procedimento *l-Consecutive-Exchange* foi executado com diferentes valores iniciais de l.

2 Este procedimento parte de uma solução do PCV, removendo iterativamente sequências de vértices consecutivos e inserindo outros para restaurar a viabilidade.

3 Também foi experimentada a abordagem descrita em PCVc restrito e não restrito envolvendo até $m = 350$ e $n = 200$, obtendo desempenhos similares ao comparar a qualidade da solução heurística com o *LP-relaxation* do modelo em [34]. A diferença média foi de cerca de $0,5\%$, enquanto o tempo de computação nunca foi superior a um minuto.

Quanto à qualidade das soluções e ao esforço computacional, a proposta de busca local de [36] melhora as abordagens propostas em [25]. A qualidade da heurística de [36] para selecionar os vértices a serem adicionados no procedimento *RestoreFeasibility* é medida na Tabela 3.7.

Capítulo 4

O problema do Caixeiro Viajante Alugador

Neste capítulo, é apresentada uma visão geral sobre o Problema do Caixeiro Viajante Alugador (PCVa), destacando-se a NP-Dificuldade do problema, heurísticas e algoritmos aproximativos para o PCVa, além de apresentar resultados obtidos por diferentes abordagens.

4.1 Introdução

O Problema do Caixeiro Viajante Alugador é uma generalização do PCV, onde um indivíduo visita um dado conjunto de cidades, iniciando e finalizando a rota na mesma cidade, utilizando veículos disponíveis para o aluguel a fim de realizar o transporte em questão. Diversos tipos de veículos são disponibilizados para o aluguel pela empresa locatária e cada veículo, por sua vez, possui características e custos de operação próprios. Fazem parte destes custos o consumo de combustível, taxas de pedágio e a quantia paga pelo aluguel. Há também uma taxa adicional a ser paga pelo retorno do veículo para a cidade na qual foi alugada, caso este seja entregue em uma cidade diferente da inicial. O objetivo, assim como o PCV original, é sempre minimizar o custo total da viagem; porém, nesta variante, ainda é somado o custo de retorno dos veículos para a cidade inicial.

O Problema do Caixeiro Viajante Alugador (PCVa) é definido, segundo [44], por um grafo completo $G = (V, E, W)$, onde $V = 1, 2, \ldots, n$ é o conjunto de vértices do grafo, $E = 1, \ldots, m$ é o conjunto de arestas do grafo e $W = 1, \ldots, w$ é o conjunto das distâncias entre os vértices do grafo, (ou o comprimento das arestas do conjunto E. Além disso, é possível contabilizar o total dos custos relacionando cada carro a um valor associado ao peso ou comprimento das arestas (i, j) do grafo inicial, já o custo total de operação de um determinado carro k ao decorrer de uma aresta (i, j) pode ser definido pela variável $c^k_{i,j}$. Dessa forma, a matriz de custo $C^k = [c^k_{ij}]$ indica o custo total do aluguel por distância viajada, combustível utilizado e possíveis taxas de pedágio entre quaisquer duas cidades i e j com um veículo k. Uma matriz $D^k = [d^k_{ij}]$ indica o custo de retorno do veículo k alugado no vértice i e devolvido no vértice j, onde $i \neq j, d^k_{ij} = 0$ se $i = j$.

4.2 NP-dificuldade do problema

Como descrito em [45], esta variante do problema consiste em determinar um ciclo hamiltoniano em um grafo G, por meio da junção de caminhos criados sobre todos os vértices de G. Definimos $T = 1, \ldots, t$ um conjunto contendo todos os t subgrafos H_r de G, $r \in T$, e $V(H_r)$ os vértices de H_r. Os subgrafos H_r deste problema têm as seguintes propriedades:

$$\bigcup_{r=1}^{t} V(H_r) = V \tag{4.1}$$

$$|V(H_s) \cap V(H_r)| \leq 1 \quad \forall\, r, s \tag{4.2}$$

A equação 4.1 consiste em uma restrição responsável por determinar que a união de todos os caminhos existentes percorram todos os vértices de G. Por sua vez, a equação 4.2 é uma restrição que garante que dois subgrafos diferentes não podem ter mais de um vértice em comum; esta condição existe para impedir que subciclos sejam formados. Porém, o autor [45] afirma que as restrições relativas às equações 4.1 e 4.2 não são suficientes para garantir o ciclo do PCVa. Dessa forma, também é necessário que os subgrupos t, quando considerados em grupos de mesma quantidade, não tenham mais de um vértice em comum. Pela definição de grafos hamiltonianos, sabemos que, por este problema tratar de ciclos hamiltonianos, cada caminho feito por um carro em um subgrafo H_r visita todos os vértices de H_r. Partindo desta premissa, o autor [45] reforça as seguintes proposições:

1. O caminho do subgrafo H_r tem que ser atribuído a um carro diferente dos carros atribuídos aos caminhos vizinhos durante a construção de um ciclo hamiltoniano em G.

2. Os custos das arestas de cada subgrafo correspondem aos custos de operação do carro que atravessa H_r.

3. Quando $t \geq 2$, o custo total considera o custo de retorno de cada carro alugado na cidade i e devolvido na cidade j, $i \neq j$.

Além disso, é definido por [14] que um ciclo hamiltoniano e os problemas do caminho hamiltoniano são problemas NP-completos bem conhecidos.

Dessa forma, devido ao que foi exposto acima, a dificuldade de resolver o PCVa é pelo menos a mesma dificuldade de resolução do PCV. No entanto, embora algumas soluções do PCV sejam também soluções de PCVa, este tem um número de soluções viáveis maior do que o anterior e incorpora todas as exigências do PCV, tal como outras várias classes de problemas de encaminhamento de veículos que são conhecidos por serem tão ou mais difíceis do que o PCV.

Pode-se dizer ainda que o PCV é um caso particular de PCVa, onde existe apenas um veículo disponível para ser alugado. O autor [46] aponta ainda que o espaço de soluções do PCVa, que é um conjunto formado por todas as suas soluções, é exponencialmente maior do que o espaço de soluções do PCV. Considerando $G = (V, E)$ um grafo completo e que PCVa é total, sem restrições e sem repetição,

qualquer permutação realizada nos vértices de G torna-se uma solução viável para o problema, considerando apenas um dos k carros possíveis. Uma vez que há k carros disponíveis para alugar, há $O(k.n!)$ combinações diferentes e viáveis que atendem à condição de uso de apenas um carro diferente no PCVa.

Ao analisar $k \geq 2$, [46] demonstra que todo ciclo hamiltoniano do PCVa pode ser particionado em até $k - subcaminhos$ contínuos, onde, para cada uma das possibilidades que compõem essas partições, pode haver um valor característico, ou específico, para a rota. Esse valor pode ser associado à necessidade de se considerar os custos relativos ao retorno dos carros alugados, e também à distribuição dos custos da rota poder variar em função da composição de cada carro.

4.3 Resultados obtidos por diferentes abordagens

Esta seção apresenta os principais algoritmos e heurísticas disponíveis para o PCVa. A partir destes estudos, novas soluções e heurísticas vêm sendo desenvolvidas para o problema. O PCVa é um problema recente, descrito por [46] como uma nova variante do PCV. Por esse motivo, há na literatura trabalhos relacionados entre si, buscando a cada publicação melhorar as soluções até então obtidas.

4.3.1 Algoritmo Memético (1989)

O *Algoritmo Memético* (Memetic Algorithm - MA) faz parte de uma classe de algoritmos chamados *algoritmos evolucionários*Segundo [46], os algoritmos meméticos retornam excelentes resultados, quando aplicados a soluções de problemas de roteamento. Um fator relevante é a presença de vizinhanças e os algoritmos de *Busca Local* satisfatoriamente eficientes e bem consolidados para esses problemas, além de apresentarem resultados que direcionam as soluções para soluções de alta qualidade e com maior eficiência quando comparadas aos resultados dos demais algoritmos e soluções reais. O algoritmo inicia uma geração de uma população inicial, e os indivíduos que fazem parte dela passam por um procedimento de *busca local* (Variable Neighborhood Descent - VND). Após essa etapa, ocorre uma evolução da solução onde, de acordo com [46], "a cada geração, um percentual de indivíduos da população [...] não irá participar da experiência evolucionária". A etapa seguinte consiste em selecionar dois indivíduos, que serão os *pais*, e submetê-los a uma recombinação, gerando dois indivíduos novos, chamados de *filhos*. Por sua vez, os *filhos* sofrem uma mutação e assim sucessivamente. A cada iteração, [46] afirma que "parte da população corrente é substituída por novos indivíduos".

MA1 x MA2

No primeiro algoritmo memético MA1, utiliza-se uma versão da heurística do vizinho-mais-próximo, adaptado para o PCVa para inicializar a população. Para adequar os cromossomos, segundo [46], é realizado o cálculo do inverso do valor da função objetivo. O segundo algoritmo, MA2, possui muita semelhança com o MA1; as diferenças existentes entre eles encontram-se na maneira como são geradas as populações iniciais e como são aprimorados os cromossomos. Neste algoritmo, a população inicial também é gerada utilizando-se uma versão da heurística do vizinho-mais-próximo, porém, a quantidade de carros disponíveis é selecionado dentro do intervalo [2,$nCar$], onde $nCar$ é o número de carros disponíveis em

determinada instância. Além disso, através do *TSP Solver*, desenvolvido por [47], é gerada uma solução ótima para cada carro do intervalo, adicionando-as à população inicial. Experimentos apresentados em [46] comprovam que o algoritmo MA2 possui melhores resultados quando comparado ao MA1 (quando é observada a qualidade da solução). Mesmo o MA2 sendo mais lento, as soluções obtidas através dele são mais consistentes.

4.3.2 Colônia de Formigas (1992)

O Algoritmo Colônia de Formigas, apresentado na Subseção 3.3.9, também pode ser aplicado à variante PCVa. Em [46], é definido que este algoritmo "é baseado em população e inspira-se no comportamento forrageiro de formigas". Além disso, também há o foco na coordenação individual de cada formiga e também na cooperação como um todo.

Os passos do algoritmo são:

1. Feita uma distribuição de k formigas em arestas aleatórias do grafo.

2- Cada formiga constrói uma solução individual ao visitar diversas arestas do grafo.

3- Tendo como posição inicial o vértice i, a formiga escolhe a próxima aresta j dentre os vizinhos possíveis. A cada visita realizada, a formiga deposita uma quantidade de feromônio τ_{ij} no caminho percorrido.

4- Após todas as formigas já terem construído as suas soluções, é necessário realizar a atualização do feromônio τ_{ij}. Esta atualização pode ser realizada no depósito do feromônio ao percorrer um caminho e na evaporação do feromônio quando é necessário eliminar um caminho da solução.

5- A solução final pode ser obtida através de um número máximo de iterações ou ainda pela estagnação das iterações, ou seja, quando todas as formigas optam pelo mesmo caminho, onde há maior concentração de feromônio.

Em [46], os resultados da comparação entre o algoritmo MA2 e o Algoritmo Colônia de Formigas mostram que a taxa de sucesso do algoritmo MA2 atinge 100% enquanto a taxa do Algoritmo Colônia de Formigas é de apenas 56%. O tempo de execução de ambos os algoritmos é similar. O autor afirma ainda que "o Algoritmo Memético (MA2) proposto supera em desempenho o algoritmo Colônia de Formigas nas instâncias não-euclidianas. Os testes aplicados às instâncias euclidianas não apresentaram diferença estatisticamente significativa entre os dois algoritmos."

4.3.3 GRASP (1995)

A metaheurística *Greedy Randomized Adaptive Search Procedures* (GRASP), apresentado em [48], é constituída por um método iterativo, onde cada uma das iterações é composta pela fase construtiva e pela fase de busca local. Para gerar a solução inicial, usa-se um método iterativo, ou seja, um elemento adicionado por vez até que a solução se complete. Em seguida, é construída uma lista de candidatos, e então é formada uma *Lista Restrita de Candidatos* (LRC) formada pelos melhores elementos da lista de

candidatos inicial. O método GRASP tem como característica a escolha aleatória dos elementos da lista LRC, sem se preocupar com o melhor elemento. O autor [46] define a fase de construção como "iterativa, adaptativa, aleatória e possivelmente gulosa". É iterativa pois a solução inicial viável é gerada elemento a elemento; adaptativa, já que as características de cada elemento são carregadas de uma iteração para outra; e gulosa pois toda adição de elementos é feita partindo-se da lista LRC, formada pelos k melhores candidatos. Para definirmos o tamanho da lista LRC, temos duas opções: uma com o parâmetro $\alpha = 1$, que traz ao algoritmo um comportamento aleatório, e outra com $\alpha = 0$, que transforma o algoritmo em um comportamento guloso. Por fim, a etapa de busca local é uma etapa de melhorias, com o objetivo de encontrar uma solução ótima local na vizinhança da solução obtida, como visto em [46]. Este algoritmo possui uma etapa de busca local porque a solução obtida na etapa construtiva não tem a garantia de ser um ótimo local. Em [48] é dito que uma das características que tornam o algoritmo GRASP interessante e atraente é a facilidade de implementação.

4.3.4 VND (1997)

A metaheurística *Variable Neighborhood Descent* (VND) foi proposta por [49], sendo um método que abrange o espaço de busca sempre variando sistematicamente as estruturas de vizinhança. Dentro do procedimento do VND, a solução ótima é composta por três estruturas de vizinhanças, onde cada estrutura é relacionada a um tipo de busca local diferente. Em [46], o autor define que o VND "procura explorar de forma gradativa as vizinhanças da solução corrente, percorrendo as mais próximas e dirigindo-se para as mais distantes". Dessa maneira, são exploradas as vizinhanças com menor custo computacional, ou seja, mais próximas. Quando não houver mais possíveis melhorias, parte-se para a exploração de uma outra vizinhança. Esta busca por uma nova vizinhança é realizada através de busca local até que todas as vizinhanças sejam exploradas. Quando uma melhoria é encontrada, a busca realizada pelo algoritmo é reiniciada, partindo da primeira vizinhança.

De acordo com o [49], existem três princípios básicos para o algoritmo VND:

1- Quando encontramos um ótimo local em uma determinada vizinhança, isso não significa dizer que este é um ótimo local de outras vizinhanças.

2- Quando obtemos um ótimo global, ou seja, a melhor solução dentre todas as vizinhanças, este é um ótimo local para todas as vizinhanças.

3- Em grande parte dos problemas, os ótimos locais de uma vizinhança são relativamente próximos.

Além destes princípios, em [50] e [46] são apresentadas oito características para o algoritmo VND; são elas:

1 Simples: É um princípio simples e claro que pode ser aplicado em diversos outros problemas.

2 Preciso: O algoritmo é formulado utilizando termos matemáticos que trazem precisão, sem ter relação com suposições.

3 Coerente: Pode ser aplicado a problemas particulares que seguem todos os princípios e passos da metaheurística.

4 Eficiente: Consegue alcançar o ótimo ou próximo da solução para a maior parte dos problemas.

5 Efetivo: Consome um tempo computacional moderado até atingir a solução ótima ou uma solução próxima da ótima.

6 Robusto: Apresenta um desempenho consistente sobre uma variedade de instâncias.

7 Amigável: Algoritmo fácil de entender, usar e pode ser claramente expressado.

8 Inovador: Esta metaheurística proporciona a criação de novos tipos de aplicações.

Os resultados computacionais apresentados em [46] comprovam que a comparação dos algoritmos VND e GRASP sempre apresenta um desempenho melhor para o algoritmo VND, mesmo quando o tempo estabelecido para a execução do VND seja o tempo do algoritmo GRASP.

4.3.5 Algoritmo híbrido GRASP/VND

Esta metaheurística foi desenvolvida para o PCVa juntando os conceitos do algoritmo GRASP e do VND. O autor [46] a define como uma maneira natural de criar um algoritmo híbrido, utilizando os algoritmos GRASP e VND, visando aumentar o potencial de intensificação que utiliza a técnica de VND, para aumentar a eficiência da etapa de busca local do algoritmo GRASP.

A versão do algoritmo GRASP/VND proposto para o PCVa, apresentado por [46], tem como passo inicial uma fase de pré-processamento, onde soluções do PCV ótimas são geradas, uma para cada carro disponível, com o *Concorde TSP Solver*, desenvolvido por [47]. O autor ainda afirma que o algoritmo híbrido GRASP/VND obteve melhores resultados no experimento computacional comparativo com a melhor das heurísticas, anteriormente apresentada na Subseção 4.3.4, o algoritmo VND. Os resultados foram comparados fixando-se os tempos de processamento de cada um deles. Esses tempos de execução são apresentados por [46] como os requisitos que cada algoritmo possui para obter seu melhor desempenho. O algoritmo híbrido é mais eficiente do que o VND tanto em instâncias Euclidianas e não-Euclidianas.

4.4 Algoritmo Memético (MA) e Algoritmo híbrido (EA+ALSP)

Nesta seção, são apresentados o Algoritmo Memético (MA) e Algoritmo híbrido (EA+ALSP), suas etapas, pseudocódigos e resultados computacionais.

4.4.1 Algoritmos Meméticos (MA1 e MA2)

No trabalho proposto por [27], é apresentado o pseudocódigo do *Algoritmo Memético* (Memetic Algorithm - MA) desenvolvido para o PCVa. São apresentados os seguintes parâmetros de entrada:

- Número de gerações (*nOffspring*)

- Tamanho da população (*sizePop*)

- Taxa de recombinação, ou seja, número de indivíduos que se reproduzem em cada geração (*txCros*)

- Taxa de mutação (*txMuta*)

- Taxa de renovação da população (*TxRenw*)

A aptidão de cada cromossomo é dada pelo inverso da função objetivo, o que significa que, quanto menor o valor da função objetivo, mais apto é o cromossomo. A Figura 4.1 abaixo apresenta o algoritmo MA.

Algoritmo: Algoritmo Memético para o PCVa

```
1.  main ( nameInstance, sizePop, nOffspring, txCros, txMuta, txRenw )
2.      instanceRead (nameInstance)
3.      Pop [] ← generateInitPop (sizePop)
4.      para i ← 1 até sizePop faça
5.          Pop[i] ← LocalSearchVND (Pop[i])
6.      fim do para
7.      para i ← 1 até nOffspring faça
8.        para j ← 1 to sizePop*txCros faça
9.            dad, mom ← parentsSelection ()
10.           son1, son2 ← Crossover (dad, mom)
11.           son1, son2 ← carsMutation (son1, son2, txMuta)
12.           LocalSearchVNC (son1, son2)
13.           se (son1, son2 < Pop[dad], Pop[mom])
14.                 Pop[dad] ← son1, Pop[mom] ← son2
15.           fim do se
16.       fim do para
17.       generateNewIndividuals (sizePop*txRenw)
18.   fim do para
19. retornar (Pop [0])
```

Figura 4.1: Algoritmo Memético para o PCVa - MA.

No algoritmo apresentado, a população inicial é gerada utilizando uma versão da heurística do vizinho-mais-próximo (*Nearest Neighbor*) no procedimento *generateInitPop()*, que recebe o tamanho da população como parâmetro de entrada.

Inicialmente, o algoritmo seleciona aleatoriamente um carro c e uma cidade de destino j, $j \neq 0$. Então, um caminho entre as cidades 0 e j é construído. A partir da cidade 0, o procedimento busca a cidade mais próxima, de acordo com os pesos das arestas correspondentes ao carro c. O procedimento acrescenta cidades de forma iterativa ao caminho de c até chegar à cidade j. Neste ponto, se houver cidades ainda não visitadas, a cidade j é definida como a nova origem. Um carro novo e uma nova cidade de destino são selecionados aleatoriamente. O procedimento continua até que todas as cidades sejam adicionadas ao caminho ou até que haja apenas um carro disponível. Neste último caso, o caminho tem de incluir todas as cidades restantes e é utilizada a mesma versão da heurística do vizinho-mais-próximo citada no início do procedimento. Finalmente, a cidade 0 é adicionada ao caminho do último carro. Nos passos em que as cidades e os carros são selecionados para serem percorridos, cada indivíduo da população inicial é submetido ao procedimento VND, descrito na Figura 4.2.

Algoritmo: Busca Local VND

```
1.  i ←; Sol* ←Sol
2.    repetir
3.      se (i=1)
4.          Sol ← InvertSol (Sol)
5.          Sol ← Insert&Saving (Sol)
6.      fim do se
7.      se (f(Sol) < f(Sol*))
8.          Sol* ← Sol; f(Sol*) ← f(Sol); i ←1
9.      fim do se
10.     senão
11.         i ← i+1
12.     fim do se_senão
13.   até (i ≥ 3)
14. retornar (Sol*)
```

Figura 4.2: Busca Local VND.

Os pais selecionados para a recombinação são escolhidos com um método similar a uma roda de roleta no procedimento *parentsSelection()* (ou seja, aleatoriamente), com probabilidades proporcionais, de acordo com a aptidão dos pais. Uma operação de recombinação multi-ponto adaptada para PCVa é usada para gerar dois filhos. O número de pontos no *crossover* é gerado aleatoriamente e a restauração pode ser necessária devido à inviabilidade de rotas ou atribuição de carros. Por exemplo, uma rota de um cromossomo C, representado na Figura 4.3, parte da cidade 0 e percorre o caminho: $[0 \rightarrow 3 \rightarrow 1 \rightarrow 8 \rightarrow 10 \rightarrow 1 \rightarrow 9 \rightarrow 4 \rightarrow 5 \rightarrow 10 \rightarrow 6]$. Esta solução não é viável, pois as cidades 1 e 10 são visitadas duas vezes, enquanto as cidades 2 e 7 não fazem parte do caminho, e este não é finalizado na cidade de início.

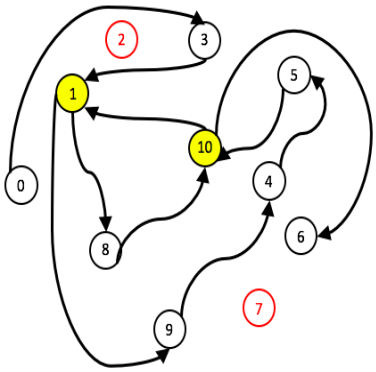

Figura 4.3: Rota de 10 cidades - Procedimento de Restauração.

Figura 4.4: Ordem de aluguel dos carros.

Dessa maneira, o procedimento de restauração altera a rota do cromossomo C para $[0 \rightarrow 3 \rightarrow 1 \rightarrow 8 \rightarrow 10 \rightarrow * \rightarrow 9 \rightarrow 4 \rightarrow 5 \rightarrow * \rightarrow 6]$, onde asteriscos são adicionados substituindo-se as cidades repetidas, neste caso, 10 e 1. O passo seguinte consiste em substituir aleatoriamente cada asterisco por uma cidade não visitada; neste caso, pelas cidades 2 e 7. Porém, a inviabilidade também é observada para o cromossomo C em relação ao aluguel de carros. A ordem de aluguel de carros para o cromossomo C onde $[1 \rightarrow 1 \rightarrow 1 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 3 \rightarrow 2 \rightarrow 2 \rightarrow 2 \rightarrow 2 \rightarrow 3]$, representado na Figura 4.4, não é possível para o problema considerado no trabalho apresentado por [27], uma vez que, neste trabalho, exige-se que cada carro seja alugado apenas uma vez. Novamente, o autor descreve que o procedimento de restauração substitui as repetições de carros por asteriscos. Em seguida, cada asterisco é substituído pelo carro que aparece na primeira posição anterior. Por fim, o aluguel de carros do cromossomo C é substituído por $[1 \rightarrow 1 \rightarrow 1 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 3 \rightarrow * \rightarrow * \rightarrow * \rightarrow * \rightarrow *$ e cada asterisco é substituído pelo carro que se encontra na última posição, ou seja, pelo carro 3. As soluções resultantes da recombinação são submetidas à mutação no procedimento *carsMutation()*. O operador de mutação verifica quais veículos não estão na solução representada para o cromossomo em questão. O autor [27] considera que os carros não atribuídos ao cromossomo são armazenados em uma lista específica para tal carro.

É definido que uma *taxa de mutação* determina o número de posições, ou seja, o número de cidades que são atribuídas aos carros de entrada. Após recombinação e mutação, os filhos são submetidos a uma busca local. A primeira versão do algoritmo memético, $MA1$, usa o mesmo procedimento de pesquisa local implementado no VND, apresentado na Figura 4.2. As soluções resultantes são comparadas com os seus pais, e os dois melhores dentre os quatro indivíduos sobrevivem.

Na sequência, é definido ainda que parte da população atual é substituída por novas soluções no procedimento *generateNewIndividuals()*. Os novos cromossomos são gerados com o método construtivo utilizado para criar a população inicial. A variável *txRenw* define a porcentagem da população que é substituída pelos novos indivíduos. Dessa forma, *sizePop*txRenw* novos indivíduos são criados para substituir os *sizePop*txRenw* piores indivíduos da população atual. Este processo de renovação promove a diversificação e evita a convergência prematura.

As Figuras 4.5 e 4.6 apresentam a rota percorrida pelo cromossomo C e a ordem de aluguel dos carros após os procedimentos que corrigem as restrições do problema em questão. Na Figura 4.5, as cidades 2 e 7 passam a fazer parte do caminho percorrido, além de não existirem cidades visitadas mais de uma vez. Na Figura 4.6, os carros obedecem à restrição de serem alugados uma única vez durante todo o caminho.

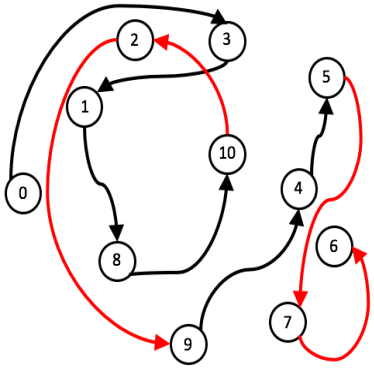

Figura 4.5: Rota Correta de 10 cidades - Procedimento de Restauração.

Figura 4.6: Ordem correta de aluguel dos carros.

Em [27], é apresentado também um segundo algoritmo memético, que é bastante semelhante ao algoritmo $MA1$ anteriormente apresentado. As diferenças estão na geração da população inicial e no procedimento de busca local utilizado para melhorar os cromossomos. Como em $MA1$, a população inicial é gerada com a versão da heurística de vizinho-mais-próximo, porém o número de carros disponíveis para cada indivíduo inicial é selecionado aleatoriamente no intervalo $[1, nCar]$. Além disso, as $nCar$ soluções ótimas para o PCV são obtidas com o *Concorde solver*, um software para resolver o PCV e outros problemas de otimização combinatória desenvolvido por [51] livremente disponível para uso acadêmico. Este software gera $nCar$ soluções, uma para cada carro, e são incluídas na população inicial. Após recombinação e mutação, os filhos são submetidos à busca local. O mesmo procedimento VND, apresentado na Figura 4.2, é implementado em $MA2$ como apresentado na Subseção 4.3.1.

O PCVa é um problema novo na literatura quando comparado a outras variantes. Justamente por esse motivo, em [27] é apresentada uma biblioteca de instâncias, denominada $CaRSLib$, com o objetivo de testar os algoritmos propostos. Todas as instâncias disponíveis são:

- **Total** : Todos os carros podem ser alugados em todas as cidades.
- **Sem restrições** : Todos os carros podem ser entregues em todas as cidades.
- **Sem repetição** : Cada carro pode ser alugado no máximo uma vez.
- ***Free*** : As taxas de retorno não estão correlacionadas com a distância entre as cidades.
- **Simétricos** : Os custos para ir da cidade i para a cidade j, e vice-versa, são iguais.
- **Completos** : Grafo completo, ou seja, há uma aresta entre cada par de cidades.

Além disso, em [27] o conjunto é composto de instâncias euclidianas e não euclidianas. Para cada conjunto, foram criados três grupos de instâncias; estes grupos dizem respeito a pesos primários associados às arestas. Os pesos de arestas primárias dos gráficos do primeiro grupo de instâncias foram adquiridos a partir de mapas reais. Os pesos das arestas primárias dos gráficos que modelam o segundo grupo de instâncias foram gerados aleatoriamente, de acordo com uma distribuição de probabilidade uniforme, no intervalo $[10, 500]$. O terceiro grupo de instâncias é baseado em instâncias $TSPLIB$ [55] [1].

O autor [27] afirma que se uma instância é definida como tendo $nCars$ carros, então o peso associado a cada aresta correspondente a cada carro t, onde $1 \leq t \leq nCars$, é escolhido aleatoriamente, com probabilidade uniforme, no intervalo $[1.1w, 2.0w]$, onde w representa o peso da aresta primária. As taxas de retorno de cada instância foram geradas aleatoriamente de acordo com uma distribuição de probabilidade uniforme na faixa $[0, 05\mu, 0, 10\mu]$ onde μ significa a média dos pesos das arestas do grafo que corresponde à instância dada. Os experimentos foram realizados em 20 casos euclidianos e 20 não-euclidianos.

Cada grupo é formado com dez instâncias de mapas reais, cinco instâncias aleatórias e cinco instâncias de tipo $TSPLIB$. O número de cidades varia entre 14 e 300 e o número de veículos varia entre 2 e 5. Para avaliar o desempenho dos algoritmos propostos, o autor utilizou a metodologia a seguir. Os critérios de parada para as versões de algoritmo MA foram $nOffspring = 20$ ou $maxGen = 0.30 \times nOffspring$ gerações, sem melhorias na melhor solução atual. A Tabela 4.1 apresenta os resultados obtidos por [27] em instâncias não-euclidianas, quando os tempos de processamento são fixados por $MA2$, onde:

- Nome: Nome da instância.

- C: Número de cidades.

- Carros: Número de carros disponíveis da instância.

- #: O valo da melhor solução.

- Média: Valor da média da solução.

- Freq: O número de vezes em que a melhor solução # foi encontrada por cada algoritmo.

- T(s): O tempo máximo de processamento.

- p: O valor que indica qual o algoritmo apresentou melhor resultado.

[1] Biblioteca que disponibiliza instâncias para o PCV.

Instâncias				MA1			MA2			T(s)	p
Nome	C	Carros	#	Média	#	Freq	Média	#	Freq		
RJ14n	14	2	167	168	167	1	167	167	6	0	0.17
RN16n	16	2	188	195	192	0	191	188	4	1	0
PR25n	25	3	229	260	241	0	241	229	1	5	0
AM26n	26	3	204	210	206	0	213	207	0	6	1
MG30n	30	4	279	333	304	0	299	286	0	9	0
SP32n	32	4	265	294	280	0	284	275	0	12	0
RS32n	32	4	280	338	302	0	311	280	1	13	0
CO40n	40	5	625	747	663	0	660	627	0	24	0
NO45n	45	5	613	769	723	0	667	613	1	39	0
NE50n	50	5	682	867	771	0	736	687	0	48	0
LN100n	100	3	1,278	1,522	1,450	0	1,338	1,278	1	492	0
OS100n	100	4	1,117	1,419	1,268	0	1,259	1,126	0	385	0
AR200n	200	3	2,281	2,735	2,544	0	2,446	2,331	0	2,491	0
TS200n	200	5	1,971	2,526	2,272	0	2,241	1,971	1	2,514	0
CB300n	300	5	3,334	3,983	3,499	0	3,726	3,334	1	6,599	0
B52nA	52	3	1,385	1,590	1,489	0	1,447	1,385	1	86	0
ch130n	130	5	2,424	2,998	2,502	0	2,630	2,424	1	743	0
d198n	198	4	4,297	5,309	4,809	0	4,665	4,297	1	2,756	0
kroB150n	150	3	3,402	4,140	3,859	0	3,675	3,491	0	1,527	0
rd100nB	100	4	1,681	2,228	1,927	0	1,832	1,681	1	464	0

Tabela 4.1: Comparação de Algoritmos Meméticos em instâncias não-euclidianas com tempos de processamento fixados por MA2.

Os resultados que compõem a Tabela 4.1 confirmam o melhor desempenho do algoritmo $MA2$, independentemente do tempo de processamento máximo adotado nos experimentos. O valor da coluna p mostra que o algoritmo $MA2$ exibe um desempenho significativamente melhor em 18 instâncias não-euclidianas, enquanto o algoritmo $MA1$ apresenta um desempenho melhor em apenas 1 instância não-euclidiana. Os melhores resultados nas colunas *Média* e # também são obtidos pelo $MA2$ em 19 e 18 instâncias, respectivamente. O autor afirma ainda que o nível de confiança 99.999% é apontado pelo teste de comparação de proporções sobre o número de sucessos relativos aos valores médios.

4.4.2 Algoritmo Híbrido (Evolucionário+ALSP)

Em [53], é apresentado um algoritmo híbrido para o PCVa, formado por um *algoritmo evolucionário* e um método híbrido chamado *Procedimento de busca local adaptável* (ALSP). No *algoritmo evolucionário*, definido pelo autor como EA, a representação de seus indivíduos é realizada por meio de dois vetores de números reais. Um vetor é responsável pela definição da prioridade de escolha das cidades

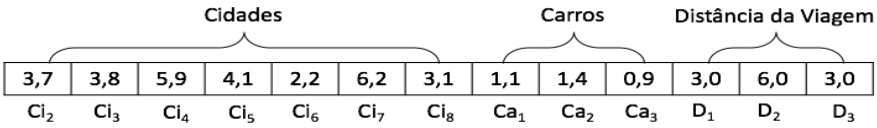

Figura 4.7: Vetor representando o PCVa.
Modificado de [53].

e o outro define a prioridade de escolha dos carros disponíveis para o aluguel. Por exemplo, se existirem as cidades a, com prioridade $1,45$, e b, com prioridade $0,20$, então a cidade a deve ser visitada antes da cidade b. Da mesma maneira, há prioridades para indicar quais carros devem ser utilizados primeiro.

Além disso, para que exista uma garantia de que um indivíduo saiba por qual distância, ou quanto tempo, um carro deve ser utilizado, os vetores devem possuir valores que serão utilizados para representar o número de cidades visitadas em cada viagem de um específico carro. "Os valores são normalizados antes do algoritmo de roteamento" [53], ou seja, se o valor de um carro c é 0.2, isso quer dizer que o carro c deve visitar 20% das cidades quando for alugado.

Para exemplificar o processo realizado pelo algoritmo, [53] apresenta o exemplo representado na Figura 4.7, onde:

1. Supõe-se que a instância do problema seja formada por 8 cidades e 3 carros.
2. Como a viagem possui seu início e fim na cidade 1, sua prioridade não precisa ser explicitada no vetor.
3. O problema abordado no trabalho é $Total$, ou seja, qualquer carro pode ser alugado em qualquer cidade. Dessa forma, qualquer carro poderia ser alugado na primeira cidade.
4. O carro que possui maior prioridade é o carro Ca_2, que será o primeiro a ser usado na viagem.
5. O autor [53] apresenta os valores normalizados como sendo $0,25$, $0,5$ e $0,25$.
6. Dessa forma, o carro Ca_2 será alugado na cidade Ci_1 e percorrerá as cidades na seguinte ordem: Ci_7, Ci_4, Ci_5 e Ci_3, obedecendo as prioridades das cidades.
7. Ao finalizar este percurso, ou seja, quando chegar na cidade Ci_3, o carro Ca_2 é devolvido para a cidade de início Ci_1, e paga-se uma taxa de retorno.
8. Na lista de prioridades dos carros, o próximo a ser alugado será o carro Ca_1. Este carro possui uma distância de viagem de tamanho 2, e visitará as cidades Ci_2 e Ci_8. Ao chegar na cidade Ci_8, o carro é devolvido à cidade Ci_3.
9. Por fim, o carro Ca_3 é alugado e visita a única cidade restante, a cidade Ci_6. Como é dito na definição do problema, o carro Ca_3 deve retornar à cidade Ci_1 a fim de completar a viagem do problema.

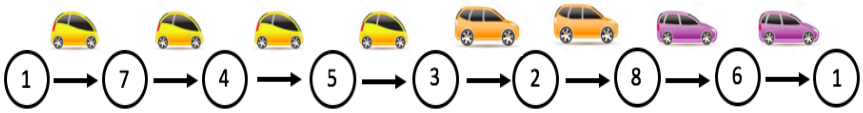

Figura 4.8: Representação gráfica do vetor do PCVa.
Modificado de [53].

Dessa maneira, uma solução derivada da representação do indivíduo pode ser representada pela Figura 4.8, onde, segundo [53], a solução pode ser representada por $i \rightarrow^c j$, onde o carro c é alugado para realizar a viagem de i para j. Em [53], é apresentado também o pseudocódigo do algoritmo evolucionário, representado na Figura 4.9, onde todos os parâmetros são definidos de acordo com os testes preliminares. Estes seguem as seguintes etapas:

1 Cálculo do custo médio das viagens: O algoritmo calcula o custo médio de todas as possíveis viagens, para todo par de cidades {i,j}, utilizando todos os carros disponíveis para o aluguel.

2 Configuração inicial da solução: Prioridades de carros e duração das viagens são escolhidas aleatoriamente para gerar a configuração inicial da solução.

Uma dificuldade encontrada em [53] foi justamente a escolha aleatória das prioridades das cidades. Os resultados obtidos não foram satisfatórios, acabaram sendo muito pobres, enquanto que a aplicação de uma abordagem gulosa, que leva em consideração os custos médios, resultou na geração de indivíduos muito semelhantes, trazendo uma má diversificação populacional. Por essa razão, foi aplicada uma estratégia que combina um procedimento aleatório para r cidades e uma segunda aplicação de um procedimento que é tanto guloso quanto aleatório para as cidades restantes.

Para realizar tal procedimento, as cidades restantes são ordenadas em uma lista L, conforme o custo médio de viagem entre as cidades e a última visitada. As cidades posicionadas no ínicio da lista possuem custos médios mais baixos, sendo assim, tornam-se as melhores opções locais. Após esta etapa, é realizado um processo de priorização nas cidades, que é representado em [53] por:

$$index = [x_1 * x_2 * n]$$

onde :

- $index$: posição da cidade escolhida na lista;
- x_1 e x_2: variáveis independentes no intervalo $(0, 1]$;
- n: tamanho da lista.

A etapa seguinte consiste em definir um número real que possui a função de representar a posição da cidade ao longo da sequência. A primeira cidade visitada recebe uma prioridade P_0. Para cada cidade seguinte, sua prioridade é calculada por $(1 - i/(n-1)) * P_0$. Como mencionado anteriormente, em [53],

não há a necessidade de uma prioridade para a cidade inicial, por isso P_0 faz parte do cálculo para as demais prioridades.

A Tabela 4.2 apresenta os valores e as descrições de todos os parâmetros utilizados no EA. O autor afirma que "estes valores foram definidos após muitos testes preliminares com diversos valores candidatos, buscando um bom equilíbrio entre performance e tempo computacional gasto" [53].

Parâmetros	Valor	Descrição
r	20%	Porcentagem das cidades escolhidas aleatoriamente para gerar um indivíduo.
totalIndiv	2000	Número total de indivíduos criados na população inicial.
popSize	150	Número de indivíduos que compõem cada geração do EA.
probMut	5%	Probabilidade de um indivíduo sofrer mutação.
sameLambda	50	Número de gerações consecutivas com o mesmo λ.
incLambda	1.04	Acréscimo em λ após *sameLambda* gerações.
intGen	100	Número de gerações entre dois mecanismos de intensificação.
upperIndiv	3	Número dos indivíduos da melhor geração que realizam intensificação.

Tabela 4.2: Parâmetros e Descrições do Algoritmo EA.
Modificado de [53].

Por fim, um valor de *totalIndiv* indivíduos são gerados, onde apenas os melhores *popSize* indivíduos fazem parte da *initialPopulation()* do algoritmo EA, exibido na Figura 4.9. Para cada geração, a população é dividida em três classes:

1. Classe A: Formada pelos 10% melhores indivíduos.
2. Classe C: Formada pelos piores 60% dos indivíduos.
3. Classe B: Formada pelos 30% restantes dos indivíduos.

A etapa seguinte, realizada pelo algoritmo EA em [53], consiste em um operador de recombinação que seleciona dois indivíduos aleatoriamente das classes A e B. A função da recombinação consiste em criar dois filhos, calculando a média de cada gene dos pais. Para que sejam produzidos filhos diferentes, uma pequena parcela da média é adicionada ao filho 1 e subtraída do filho 2. A parcela retirada da média consiste em um número real aleatório entre 0 e λ onde λ é o parâmetro que possibilita as diferenças existentes entre os filhos gerados e seus pais. Quanto mais o número tender a zero, menores as chances de os filhos serem muito diferentes dos pais. Porém, se o número for alto, os filhos gerados serão muito diferentes dos pais.

Algoritmo: Pseudocódigo para o Algoritmo Evolucionário

```
1. pop ← initialPopulation()
2. generation ← 1
3. bestSol ← bestIndividual(pop)
4. enquanto tempo total não excedido faça
5.         children ← recombination(pop)
6.         pop ← naturalSelection(pop+children)
7.         bestSol ← bestIndividual(pop)
8.         generation ← generation + 1
9.         se generation mod sameLambda = 0 então
10.            lambda = lambda * incLambda
11.            se generation mod intGen = 0 então
12.                intensificate(pop)
13.            fim do se
14.        fim do se
15.fim do enquanto
16.retornar bestSol
```

Figura 4.9: Pseudocódigo do Algoritmo EA.
Modificado de [53].

Outro procedimento apresentado em [53] é o $k - swap$ - um procedimento de mutação que troca as prioridades das cidades, onde k representa o número de cidades que serão utilizadas. Se um indivíduo melhor é encontrado, a mutação realizada é mantida, caso contrário, é descartada. O $k - swap$ será utilizado toda vez que um novo indivíduo for gerado, tanto pelo procedimento de recombinação quanto pela população inicial. Após a geração e mutação dos $popSize$ filhos, todos os indivíduos (sejam eles pais ou filhos) são misturados e utilizados para a geração única dos melhores $popSize$ indivíduos. Esse procedimento é realizado pelo $naturalSelection(pop + children)$ no algoritmo EA da Figura 4.9. Após alguns procedimentos de geração, o valor de $incLambda$ causa um acréscimo no valor de $Lambda$, visando possibilitar a geração de indivíduos mais diversificados. O passo seguinte do algoritmo consiste em um mecanismo de intensificação, que é aplicado a todas as $intGen$ gerações, onde os melhores $upperIndiv$ indivíduos e alguns dos piores realizam uma mutação $3 - swap$.

O algoritmo híbrido, proposto em [53], faz uso de uma nova formulação matemática para o PCVa. O primeiro modelo foi proposto por [52]. Após testes e comparações com [53], ficou evidenciado que algumas restrições não estavam presentes na primeira versão. Por isso, uma nova formulação matemática foi proposta para o PCVa; segundo (apud [53]), "com base na famosa formulação Miller-Tucker-Zemlin Travelman Salesman Problem (MTZ-TSP) de Miller, Tucker e Zemlin de 1960". A formulação matemática proposta por [53] é utilizada na forma de um procedimento de busca local, com poucas variáveis, poucas restrições e tempo computacional pequeno.

Para a formulação matemática, [53] define:

- A função objetivo é o somatório dos custos de viajar de uma cidade i para j, utilizando determinado carro c, e os custos de devolução dos carros.

- É criada uma sequência com as cidades visitadas, sempre visando eliminar subcaminhos.
- Cada cidade é visitada apenas por um carro, e a saída desta cidade é realizada também por um único carro.
- Cada carro pode ser alugado uma única vez.
- Um carro apenas deve ser entregue na cidade inicial em que foi alugado.
- Uma taxa de devolução é definida para todo carro c que é alugado em uma cidade i e devolvido em uma cidade j.

Complementando a abordagem proposta em [53], é realizado o procedimento de busca local, ALSP, apresentado na Figura 4.10, onde F é uma formulação matemática e S uma solução para F. Toda variável x_i possui um valor em S, definido por $x_i(S)$. É definido $C(S)$ como as restrições adicionais $x_i = x_i(S)\ \forall \mathrm{x}_i \in S$ adicionado à formulação matemática F. É apresentada também em [53] uma maneira de fixar todas as variáveis, a partir de uma formulação estendida $F \cup C(S)$. Se o valor de $F \cup C(S)$ for repassado a um otimizador, este não encontrará correções possíveis para realizar, resultando assim na própria solução inicial (S). Porém, existindo um subconjunto $S' \subset S$ e uma formulação estendida $F \cup C(S - S')$, o otimizador poderá obter a melhor solução existente aplicando correções nas variáveis $S - S'$, através de um procedimento de busca local em torno de $S - S'$, obtendo uma solução melhor que a solução inicial S ou, no pior dos casos, resultando na própria solução S.

A cada formulação estendida $F \cup C(S)$, é gerado um subproblema no algoritmo ALSP. O algoritmo ALSP inicia a solução resolvendo os subproblemas fáceis primeiro, melhorando gradualmente as soluções encontradas. Neste algoritmo, as variáveis são classificadas em quatro grupos:

Grupo 1: O primeiro grupo consiste em uma cópia de cada variável da solução S. É definido, em [53], que este primeiro grupo é constante e serve apenas para referência. Além disso, todas as outras variáveis também farão parte dos outros grupos.

Grupo 2: O segundo, é o grupo não-fixo (S') é formado justamente por variáveis que não possuem um valor fixo.

Grupo 3: O terceiro grupo também é fixo - ($S - S'$); é o grupo de variáveis x_i que ainda possuem um valor original fixado por $x_i(S)$.

Grupo 4: O quarto grupo é temporário; chamado grupo de teste, definido por T. As variáveis que fazem parte deste grupo são selecionadas do grupo fixo, porém são tratadas como não-fixas na etapa do ciclo principal ALSP corrente.

Durante a execução desses procedimentos, se uma solução melhor ou igual à atual for encontrada, as variáveis são transferidas do grupo fixo para o não-fixo. A Tabela 4.4 apresenta os quatro parâmetros do algoritmo ALSP, que são ajustados no decorrer de sua execução.

Parâmetros	Descrição
maxLRgap	Porcentagem que representa o subproblema a ser aceito no momento da execução.
size	Número de variáveis que serão testadas em cada etapa da execução.
timeOpt	Tempo atribuído ao otimizador para resolver $F \cup C(S)$.
k	Um multiplicador para o limite *timeOpt*.

Tabela 4.3: Parâmetros e Descrições do Algoritmo ALSP.

Modificado de [53].

Algoritmo: Pseudocódigo para o Algoritmo ALSP

```
1. Initiate(maxLRgap, size, timeOpt, k)
2. incumbent ← S
3. bestSol ← value(incumbent)
4. S' ← ∅
5. enquanto tempo total não excedido e |S – S'| > 0 faça
6.     T ← variáveis de S – S', com |T| = size
7.     S' ← S'∪ T
8.     LR ← linearRelaxation(F ∪ C(S – S'))
9.     se LR*maxLRgap < bestSol então
10.        update(maxLRgap)
11.        S' ← S' – T
12.    senão
13.        copy ← incumbent
14.        Incumbent ← optimize(F ∪ C(S – S'), incumbent, timeOpt)
15.        enquanto value(incumbent) < bestSol e  incumbent não é ótimo e tempo total não
   excedido faça
16.            bestSol ← value(incumbent)
17.            incumbent ← optimize(F ∪ C(S – S'), incumbent, k * timeOpt)
18.            se incumbent é ótimo então
19.                bestSol ← value(incumbent)
20.                reset(size, maxLRgap)
21.            senão
22.                incumbent ← copy
23.                S'← S' – T
24.                update(timeOpt, size)
25.            fim do se
26.        fim do while
27.    fim do se
28.fim do while
29.retornar incumbent
```

Figura 4.10: Pseudocódigo do Algoritmo ALSP.

Modificado de [53].

O objetivo do uso do algoritmo ALSP, descrito em [53], é de "melhorar uma solução inicial, como qualquer algoritmo de busca local". Caso seja definido um limite alto de tempo para a execução, é

provável que se obtenha a solução ótima, pois chegará um momento em que nenhuma variável precisará de correção. As etapas do algoritmo ALSP, apresentado na Figura 4.10, são descritas por [53] como segue:

1. Configuração Inicial:
   A solução *incumbent* é gerada por um algoritmo externo.
   A *bestSol* permanece com o $value(incumbent)$, ou seja, o valor atual da solução.
   S' é carregado com $\emptyset$, pois ainda não existem variáveis não-fixas.

2. Critérios de Parada:
   Limite de tempo excedido.
   Pelo menos um variável fica mantida.

3. Transferência de Variáveis:
   Um conjunto de variáveis fixas é adicionado ao grupo de teste.
   As variáveis são escolhidas aleatoriamente.
   S' passa a englobar as variáveis de teste.

4. Relaxamento Linear:
   É realizado o cálculo do relaxamento linear LR para o subproblema $(F \cup C(S - S'))$.
   O algoritmo segue executando a formulação, caso LR não possua um valor muito distante do valor da solução atual *bestSol*.
   Caso o valor seja maior que o $maxLRgap$, o algoritmo simplesmente não trabalha com este problema, por ser considerado difícil.
   As variáveis de teste são removidas do grupo não-fixo.
   O valor de $maxLRgap$ é atualizado, incrementado em $0,1\%$ para permitir futuras formulações, caso seja necessário.
   Se o intervalo for aceitável, é chamado o otimizador com as variáveis $F \cup C(S - S')$, *incumbent* e *timeOpt*.
   O otimizador retornará a própria solução que foi carregada ou uma melhor, caso exista.
   Caso o otimizador encontre uma melhor solução, mas devido ao tempo limite não consiga provar que ela é viável, um acréscimo no tempo ocorre com $k * timeOpt$ segundos.

5. Verificar Otimização:
   Se a otimização do subproblema pôde se concluída dentro do tempo determinado, o algoritmo ALSP considera que as variáveis não influenciaram na dificuldade do subproblema em questão e permanecem sendo variáveis não-fixas.
   Os valores de $size$ e $maxLRgap$ voltam a ser os iniciais.
   Esse retorno no valor acontece pois o algoritmo ALSP deve continuar sempre tentando otimizar os subproblemas mais fáceis possíveis.
   Caso a otimização não seja realizada por completo, o algoritmo ALSP identifica o subproblema como difícil e com tempo computacional alto, mantendo a solução atual para a iteração seguinte.

6. Tempo de execução:

   Quando uma solução precisa de mais tempo de execução para ser obtida, significa que $|T|$ ou $size$ são muito grandes e criam um espaço de solução muito grande para a otimização.

   Caso isso ocorra, o limite de tempo deve ser elevado e o valor de $size$ diminuído (nunca chegando ao valor 0 ou um negativo) até que o algoritmo atinja o novo tempo limite de execução e retornará a solução obtida.

Resultados Computacionais

Para a execução dos testes [53], foram utilizadas instâncias existentes na biblioteca CaRS [2] desenvolvida por [54]. Em [53], é afirmado que "em exemplos euclidianos, algoritmos meméticos e transgenéticos propostos por [52] apresentaram bons resultados em pouco tempo. Para este tipo de casos, a literatura já possui boas heurísticas". O algoritmo híbrido EA+ALSP obteve resultados próximos aos apresentados por [52], porém, depois de muitas horas de execução.

O algoritmo proposto em [53] foi submetido a um teste composto por dez etapas. O número de etapas foi definido após a execução de um teste com um limite de tempo alto não obter bons resultados, afinal, quando os subproblemas aplicados ao ALSP são difíceis, a obtenção da solução torna-se tão difícil quando o problema original. O experimento realizado fez uma comparação entre o Algoritmo Transgenético (TA) proposto em [52], o Algoritmo Evolucionário (EA) e a versão híbrida EA+ALSP proposta em [53]. Para tornar as comparações justas, o limite de tempo da execução dos algoritmos propostos em [53] foi reduzido a 66% do limite de tempo definido por [52]. A Tabela 4.5, adaptada de [53], apresenta os resultados médios e as melhores soluções obtidas em um conjunto de 30 execuções dos algoritmos EA e EA+ALSP; a Tabela 4.4 apresenta as descrições das colunas da Tabela 4.5.

Coluna	Descrição
n	Número de cidades de cada instância.
$\|C\|$	Número de carros de cada instância.
TA	Soluções do algoritmo TA em 30 execuções encontradas na literatura.
EA	Soluções obtidas nas 30 execuções do algoritmo EA com limite de tempo total.
$EA+ALSP$	Soluções onde o EA utilizou 20% do limite de tempo total e o melhor indivíduo gerado, repassado ao ALSP para realizar a estratégia de 10 etapas. Cada etapa recebeu 8% do limite de tempo total.

Tabela 4.4: Descrição da Tabela 4.5.

Fonte: O autor.

A partir da Tabela 4.5, é possível concluir que o algoritmo TA obteve resultados um pouco mais robustos em pequenas instâncias, enquanto o algoritmo proposto em [53] obteve melhor desempenho em todas as outras instâncias testadas. Quando comparadas apenas às melhores soluções, "o algoritmo $EA+ALSP$ é claramente melhor que o TA" [53]. Os valores em negrito apresentados na Tabela 4.5 representam

[2] http://www.dimap.ufrn.br/lae/en/projects/CaRS.php

as melhores soluções obtidas. O algoritmo TA apresentou um tempo computacional muito elevado, enquanto os algoritmos EA e EA+ALSP conseguiram obter resultados iguais ou melhores utilizando menor tempo computacional. Em [53], é afirmado que aproximadamente "85% das soluções finais foram obtidas até o 6º ou 7º estágio, o que significa que um tempo computacional menor não afetaria muito o desempenho EA+ALSP". Em [53], ainda são apresentados resultados que garantem que as melhores soluções do algoritmo híbrido EA+ALSP sempre são iguais ou melhores que as soluções do algoritmo TA.

Instância				TA		EA		EA+ALSP	
Nome	n	|C|	Tempo	média	melhor	média	melhor	média	melhor
BrasilRJ14n	14	2	0.6	167	**167**	167.6	**167**	167.7	**167**
BrasilRN16n	16	2	0.6	189	**188**	190.0	**188**	190.2	**188**
BrasilPR25n	25	3	10.6	227	**226**	228.8	**226**	227.6	**266**
BrasilAM26n	26	3	10.6	202	**202**	203.2	**202**	202.9	**202**
BrasilMG30n	30	4	21.3	277	**271**	280.1	**271**	277.9	**271**
Canoas30n	30	4	24.6	382	**376**	386.4	377	385.3	**376**
BrasilSP32n	32	4	26.0	259	**254**	264.6	**254**	261.0	**254**
BrasilRS32n	32	4	26.6	271	**269**	277.2	270	273.1	**269**
BrasilCO40n	40	5	56.0	583	**576**	590.8	577	585.0	**576**
BrasilNO45n	45	5	69.3	561	551	564.8	551	559.2	**548**
att48nA	48	3	89.3	1001	993	1002.6	992	995.3	**988**
BrasilNE50n	50	5	98.0	629	618	629.1	**611**	625.7	**611**
Santos50n	50	5	102.0	402	392	394.1	384	387.8	**382**
Berlin52nA	52	3	120.6	1350	1326	1324.8	1311	1315.0	**1303**
st70nB	70	4	276.6	934	910	905.9	890	888.7	**879**
Macapa80n	80	5	416.0	636	616	616.0	605	605.4	**599**
rat99nB	99	5	873.3	1432	1399	1400.7	1385	1367.2	**1349**
Londrina100n	100	3	738.0	1201	1186	1179.2	1166	1149.7	**1146**
w100nB	100	4	777.3	1703	1670	1667.5	1650	1624.3	**1615**
rd100nB	100	4	834.0	1442	1412	1410.4	1399	1368.0	**1357**
Osasco100n	100	4	666.0	1005	993	988.8	978	967.9	**964**
pr107n	107	5	1054.3	1740	1698	1701.1	1676	1645.1	**1631**
ch130n	130	5	1200.0	1737	1696	1687.0	1673	1641.0	**1632**
Cuiaba140n	140	4	1200.0	1364	1339	1329.2	1315	1300.0	**1293**
krob150n	150	3	1200.0	3018	2966	2947.9	2923	2856.3	**2845**
PortoVelho160n	160	3	1200.0	1446	1426	1415.7	1405	1388.2	**1382**
d198n	198	4	2400.0	3246	3188	3152.2	3130	3043.9	**3036**
Aracaju200n	200	3	2400.0	1984	1942	1897.3	1885	1844.5	**1839**
Teresina200n	200	5	2400.0	1467	1410	1384.3	1373	1353.4	**1343**
Curitiba300n	300	5	3600.0	2272	2222	2162.6	2145	2111.8	**2100**

Tabela 4.5: Resultados do Algoritmo EA+ALSP.

Modificado de [53].

Capítulo 5

Conclusão

Neste capítulo, são apresentadas as considerações finais sobre a pesquisa realizada e possíveis trabalhos futuros.

5.1 Considerações finais

O presente trabalho realizou uma pesquisa na literatura sobre o Problema do Caixeiro Viajante (PCV), identificando a sua importância teórica com diversas abordagens e técnicas publicadas. Além disso, foi demonstrada a grande aplicabilidade em tarefas cotidianas de pequeno e grande porte, como, por exemplo, o planejamento de compra de uma residência ou a implantação de instalações de determinada empresa. Foram levantadas informações sobre diversas variantes do PCV, o que possibilitou a identificação da variante mais estudada, o Problema do Caixeiro Viajante Comprador (PCVc), e a variante mais recente, Problema do Caixeiro Viajante Alugador (PCVa). A partir desta etapa, uma busca mais aprofundada foi realizada para as duas variantes, com o objetivo de levantar a formulação original de cada variante, definindo sua configuração e etapas, apresentar as principais abordagens existentes para cada variante, e então detalhar a heurística, algoritmos e resultados computacionais da melhor solução obtida para a variante do PCV em questão, até o período de conclusão deste trabalho.

Para a variante do PCVc, foram identificadas as principais abordagens com resultados computacionalmente viáveis, que são baseadas em um algoritmo de *branch-and-cut*, aplicado dentro de uma Heurística de Adição de Mercados (MAH), e um algoritmo com procedimento de busca-local, aplicado dentro da própria Heurística de Busca-Local. A heurística MAH apresenta uma evolução, e até mesmo, uma consolidação das abordagens previamente desenvolvidas. Por esse motivo, é tão amplamente estudada e utilizada nos trabalhos recentes, ainda que tenha sido desenvolvida há mais de 10 anos.

A Heurística de Busca-Local apresenta resultados com tempo computacional muito reduzidos quando comparados a outras abordagens; é um procedimento que possibilita a obtenção de uma solução ótima, em tempo computacional viável, mesmo sendo aplicada a uma instância grande. Esta heurística também foi escolhida para ser abordada neste trabalho por ser utilizada como um processo de intensificação em abordagens mais recentes, como algoritmos baseados em colônia de formigas e algoritmos

trangenéticos. Como estas abordagens são muito recentes, sem testes de comparação realmente consistentes, a Heurística de Busca-local permanece sendo o diferencial destes novos estudos.

A variante do PCVa, como descrito anteriormente, é um objeto de pesquisa recente, o que torna o número de trabalhos publicados significativamente menor que do PCVc. Por esse motivo, a seleção das abordagens a serem explicitadas no presente trabalho foi realizada pela avaliação de quais heurísticas foram realmente desenvolvidas para o PCVa, ao invés de serem adaptadas de outras variantes. A primeira abordagem apresentada foi o algoritmo memético, que utiliza as heurísticas de Busca-Local e do Vizinho-mais-Próximo. Os resultados apresentados na literatura comparam suas duas versões existentes, e apontam claramente que o algoritmo memético MA2 apresenta resultados melhores que a versão MA1, o que significa que este algoritmo apresenta melhores resultados se for comparado a qualquer abordagem anterior a ele.

A segunda abordagem selecionada para estudo trata-se de um algoritmo híbrido EA+ALSP, que é composto de um algoritmo evolucionário e um procedimento de busca local adaptável, desenvolvido recentemente como uma evolução do algoritmo TA - uma solução presente na literatura, porém com inconsistências em sua formulação. Este algoritmo híbrido foi comparado ao algoritmo TA, e apresentou resultados satisfatoriamente melhores. Seus resultados com instâncias pequenas foram muito próximos, porém, quando instâncias maiores foram adicionadas aos testes, o tempo computacional do algoritmo TA o tornou inviável, enquanto o tempo de execução do algoritmo híbrido poderia ter sido reduzido na configuração inicial, pois obteve melhores soluções antes de finalizar as 10 etapas preestabelecidas.

5.2 Trabalhos futuros

Como proposta para trabalhos futuros, esta pesquisa aponta o desenvolvimento e estudo de novas abordagens para o Problema do Caixeiro Viajante Alugador. Na literatura, hoje, não há muitos estudos abordando esta variável, o que não condiz com a aplicabilidade da mesma, que pode ser utilizada nos mais diversos setores, visando a melhoria de seus processos produtivos e a economia de suas soluções.

Referências Bibliográficas

[1] Herrera, B. A. *Combinação de exame de partículas com inspiração quântica e método Lin-Kernighan-Helsgaun aplicada ao Problema do Caixeiro Viajante*, Curitiba, 2007. 97p. Dissertação - Pontifícia Universidade Católica do Paraná. Programa de Pós-Graduação em Engenharia de Produção e Sistemas.

[2] Silveira, J. F. Porto D. *O Problema do Caixeiro Viajante.* 2000, Disponível em: <http://www.mat.ufrgs.br/ portosil/caixeiro.html>. Acesso em: 6 jun. 2016.

[3] Szwarcfiter, J. L. *Grafos e algoritmos computacionais. Vol 2.*, Campus, 1988.

[4] Davendra, D. *Traveling Salesman Problem, Theory and Applications, 2010.*, InTech 9,2010,336.

[5] Karp, R. M. *Reducibility Among Combinatorial Problems*, University of California at Berkeley, 1972.

[6] Ausiello, Giorgio et al. *Complexity and approximation: Combinatorial optimization problems and their approximability properties.*, Springer Science & Business Media, 2012.

[7] Laporte, G. *The Traveling Salesman Problem: An overview of exact and approximate algorithms*, European Journal Of Operational Research, 1992.

[8] Papadimitriou, C. H.; Vazirani, U. V.; Dasgupta, S. Algorithms, Boston: McGraw-Hill Higher Education, 2008.

[9] Goldbarg, M., Goldbarg, E. *Grafos: Conceitos, algoritmos e aplicações.*, Elsevier Brasil, 2012.

[10] Benoit, A. Robert, Y. Vivien, F. *A Guide to Algorithm Design: Paradigms, Methods, and Complexity Analysis. Chapman and HallCRC Applied Algorithms and Data Structures series*, CRC Press, 2013,380

[11] Déharbe, D. *Elementos de análise de Programas.*, 2003, DIMAp - Universidade Federal do Rio Grande do Norte.

[12] Rothlauf, F. *Design of modern heuristics: principles and application.*, Springer Science Business Media, 2011

[13] Dasagupta, S. Papadimitriou, C. Vazirani, U. *AMGH Editora, 2009*

[14] Garey, M. R., Johnson, D. S. *Computers and intractability: a guide to the theory of NP-completeness.*, 1979, San Francisco: W. H. Freeman.

[15] Ed Pegg Jr. *The Icosian Game, Revisited. Some extensions of Hamiltonian tours are explored.*, The Mathematica Journal 11:3, 2009

[16] Hamilton and the Icosian Game - Graph Theory History. *https://sites.google.com/site/math360graphtheory/hamilton-and-the-icosian-game*,

[17] Lu, Jingui, and Min Xie *Immune-Genetic Algorithm for Traveling Salesman Problem.*, TRAVELING SALESMAN PROBLEM, THEORY AND APPLICATIONS, Edited by Donald Davendra, Published by InTech (2010): 81-96

[18] Johnson, D.S. *Approximation algorithms for combinatorial problems.*, 1974 - J. Comput. System Sci., 9, 256-278.

[19] Hochbaum, D. S. *Approximation algorithms for NP-hard problems*, 1997, PWS Publishing Company.

[20] Carvalho, M. H. et al. *Uma introdução sucinta a Algoritmos de Aproximação*, 2001 - IME - Universidade de São Paulo - São Paulo.

[21] Bijit, L. S. *Algoritmos Heurísticos - Programación en Pascal*, 2003, Universidad Tecnica Federico Santa María - Departamento de Electronica - Chile.

[22] Pigatti, A. A. *Modelos e Algoritmos para o Problema de Alocação Generalizada (PAG) e Aplicações*, 2003 - PUC-Rio, Departamento de Informática - Dissertação de Mestrado - Rio de Janeiro.

[23] Bueno, F. *Métodos Heurísticos - Teorias e Implementações*, 2009, Instituto Federal de Santa Catarina - Araranguá SC.

[24] Manerbaa, D., Mansinia, R., Riera-Ledesma, J. *The Traveling Purchaser Problem and its variants*, 2016, European Journal of Operational Research.

[25] Boctor, F.F. Laporte, G., Renaud, J. *Heuristics for the traveling purchaser problem.*, Computers Operations Research 30 (2003) 491-504.

[26] Ramesh T. *Traveling purchaser problem.*, 1981 - Opsearch

[27] Goldbarg, M. C., Asconavieta, P. H., Goldbarg, E. F. G. *Memetic algorithm for the Traveling Car Renter Problem: an experimental investigation*, Memetic Comp. (2012), 4:89-108

[28] Corrêa, F. A., Lorena, L. A. N, Senne E., L. F. *Método de Geração de Colunas Aplicado ao Problema de Localização de Facilidades Não-Capacitado*, Instituto Nacional de Pesquisas Espaciais São José dos Campos, SP ? Brasil

[29] Bagi, L. B. *Algoritmo Transgenético na Solução do Problema do Caixeiro Comprador*, 2007, Dissertação de mestrado - Universidade Federal do Rio Grande do Norte - UFRN

[30] Pearn, W.L. Chien, R.C. *Improved solutions for the traveling purchaser problem.*, 1998, Computers Operations Research, 25(11), 879-885.

[31] Golden, B. L., Levy L., Dahl R. *Two generalizations of the traveling Salesman Problem.*, OMEGA 1981;9:439 45.

[32] Ong, H. L. *Approximate algorithms for the traveling purchaser problem.*, 1982, Operations Research Letters, 1, 201-205

[33] Voβ, *S.Dynamic tabu search strategies for the traveling purchaser problem.*, 1996 – *AnnalsofOperationsResearch*, 63, 253?275*Braunschweig* – *Germany*

[34] Laporte, G., Riera-Ledesma, J., Salazar-González, J. *A branch-and-cut Algorithm For The Undirected Traveling Purchaser Problem.*, 2003, Operations Research 51(6):940-951.

[35] Teeninga, A., Volgenant, A. *Improved heuristics for the traveling purchase problem.*, 2004, Computers Operations Research, 31, 139-150.

[36] Riera-Ledesma, J., Salazar-González, J. *A heuristic approach for the Travelling Purchaser Problem.*, 2003, European Journal of Operational Research 162 (2005) 142-152.

[37] Lin, S., Kernighan, B. W. *An Effective Heuristic Algorithm for the Traveling-Salesman Problem*, 1973, Journal Operations Research - Volume 21 Issue 2, Pages 498-516 INFORMS Institute for Operations Research and the Management Sciences (INFORMS), Linthicum, Maryland, USA

[38] Bontoux, B.; Feillet, D. *Ant colony optimization for the traveling purchaser problem.*, (2006), Computers and Operations Research.

[39] Koide, R. M. *Algoritmo de Colônia de Formigas Aplicado à Otimização de Materiais Compostos Laminados*, Universidade Tecnológica Federal do Paraná - Curitiba - Brasil

[40] Dorigo, M. *Optimization, Learning and Natural Algorithms.*, 1992, Departamento de Eletrônica, Politécnico di Milano - Itália

[41] Zuben, F. J. V. *Buscas locais para caixeiro viajante e introdução à aiNet*, 2013, Unicamp - SP.

[42] Singh, K. N., Van Oudheusden, D. L. *A branch and bound algorithm for the traveling purchaser problem.*, European Journal of Operational Research 1997;97:571-9.

[43] Prestes, A. N. *Uma Análise Experimental de Abordagens Heurísticas Aplicadas ao Problema do Caixeiro Viajante*, 2006, UFRN - Departamento de Informática e Matemática Aplicada . Programa de Pós-Graduação em Sistemas e Computação.

[44] Lopes, H. S., Takajashi, R. H. C. *Computação evolucionária em problemas de engenharia/organização.*, 2011, Omnipax - Curitiba PR.

[45] Goldbarg, M. C., Asconavieta, P. H., Goldbarg, E. F. G. *Algoritmos Evolucionários na Solução do Problema do Caixeiro Alugador.*, 2011, Computação Evolucionária em Problemas de Engenharia.

[46] Silva, P. H. A. *O problema do Caixeiro Alugador: Um estudo Algorítmico.*, 2011, Tese - Universidade Federal do Rio Grande do Norte. Centro de Ciências Exatas e da Terra. Departamento de Informática e Matemática Aplicada. Programa de Pós-Graduação em Sistemas e Computação. Natal - RN.

[47] Applegate, D. L., Bixby, R. E.; Chvatal, V.; Cook, W. J. The Traveling Salesman Problem: A Computational Study., 2006, Princeton University Press.

[48] Feo, T. A., Resende, M. G. C. *Greedy randomized adaptive search procedures.*, 1995, Journal of Global Optimization, 6, 109-133.

[49] Hansen, P., Mladenovic, N. *An introduction to variable neighborhood search.*, 1999, In Voss, S. Martello, I. H. Osman, and C. Roucairol, editors, MetaHeuristics: Advances and Trends in Local Search Paradigms for Optimization, pages 433? 458. Kluwer Academic Publishers, Boston, MA.

[50] Hansen, P., Mladenovic, N. *Variable Neighborhood Search.*, 2003, In Handbook of Metaheuristics, pages 145-184, Springer.

[51] Applegate, D. L., Bixby, R., Chvátal, V., Cook, W. *The traveling salesman problem: a computational study.*, 2006, Princeton University Press, Princeton.

[52] Goldbarg, M. C. *et al. A transgenetic algorithm applied to the traveling car renter problem.*, Expert Systs. with Appl., 40:6298-6310, 2013.

[53] Silva, A. R. V., Ochi, L. S. *An efficient hybrid algorithm for the Traveling Car Renter Problem* 2016, Expert Systems With Application, 64: 132-140.

[54] da Silva, P. *CaRS library. http://www.dimap.ufrn.br/lae/en/projects/CaRS.php.*, 2011, Online; acessado em 15-Dezembro-2016.

[55] TSPLIB *http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/*, Online; acessado em 16-Dezembro-2016.