INSTITUTO LATINO AMERICANO DE CIÊNCIAS
DA VIDA E DA NATUREZA (ILACVN)
ENGENHARIA FÍSICA

Estudo do Problema do Caixeiro Viajante por meio de solvers

Abner De Almeida Costa

Foz do Iguaçu
2022

INSTITUTO LATINO AMERICANO DE CIÊNCIAS DA VIDA E DA NATUREZA (ILACVN)
ENGENHARIA FÍSICA

Estudo do Problema do Caixeiro Viajante por meio de solvers

Abner De Almeida Costa

Trabalho de Conclusão de Curso apresentado ao Instituto Latino Americano da Vida e Natureaza como requisito parcial à obtenção ao título de Bacharel em Engenharia Física

Orientador: Prof. Dr. Rodrigo Bloot

Foz do Iguaçu
2022

ABNER DE ALMEIDA COSTA

ESTUDO DO PROBLEMA DO CAIXEIRO VIAJANTE POR MEIO DE SOLVERS

Trabalho de Conclusão de Curso apresentado ao Instituto Latino Americano da Vida e Natureaza como requisito parcial à obtenção ao título de Bacharel em Engenharia Física

Orientador: Prof. Dr. Rodrigo Bloot

Aprovado pela banca examinadora em: ___/___/_____

Prof. Dr. Rodrigo Bloot - UNILA
Prof.º Orientador

Dr. Tiago Antonio Alves Coimbra - CEPETRO-UNICAMP
Membro titular

Prof. Dr. Emidio Santos Portilho Junior - UNIOESTE
Membro titular

Dedico esse trabalho aos meus pais Ediuson e Sônia.

AGRADECIMENTOS

Agradeço a Deus que por meio de pessoas especiais me deu condições para chegar até aqui. Agradeço aos meus pais Ediuson e Sônia pelo amor e motivação, minhas irmãs Tainah e Tâmara que me deram suporte quando precisei. Agradeço a Marta minha tia que nunca mediu custos e esforços para que concluísse essa etapa. Ao Rodrigo, Leumam, Regina Helena, Leonardo Bedani, Henry Piovesana, Luana, Valdecir Zatta, Arlene Zatta e Evandro Grigio por serem fundamentais para que esse objetivo fosse alcançado. Agradeço ao professor Rodrigo Bloot por me orientar neste trabalho. Por fim agradeço a IBM por disponibilizar seu solver de modo gratuito para a realização dos testes.

"Entrega o teu caminho ao Senhor, confia nele, e o mais Ele fará."
Davi (Salmo 37:5)

Costa, Abner de Almeida.**Estudo do Problema do Caixeiro Viajante por meio de solvers** .66 páginas. 2022.Trabalho de Conclusão de Curso (Graduação em Engenharia Física) – Universidade Federal da Integração Latino-Americana, Foz do Iguaçu, 2022.

Resumo

O problema do caixeiro viajante é um problema de alto impacto social além de ser relevante do ponto de vista da otimização. Este problema tem sido um desafio ao longo dos anos com relação aos requisitos de eficiência bem como eficácia da solução. Várias técnicas foram desenvolvidas ao longo dos anos para tratar este problema. No presente trabalho, por meio de comparações, alguns solvers disponíveis foram testados em relação ao tempo de execução e performance para obter o valor ótimo. Foram avaliadas algumas instâncias de porte reduzido e moderados. Os resultados obtidos mostram que, mesmo em instâncias onde a matriz de distâncias é pequena, alguns dos solvers testados apresentaram dificuldades na resolução do problema tanto no tempo de execução quanto na performance para obter o caminho ótimo.

Palavras chave: Otimização, Pyomo, Caixeiro Viajante.

Costa, Abner de Almeida.**Study of the Traveling Salesman Problem through solvers** .66 pages. 2022.Trabalho de Conclusão de Curso (Graduação em Engenharia Física) – Universidade Federal da Integração Latino-Americana, Foz do Iguaçu, 2022.

Abstract

The traveling salesman problem is a problem of high social impact and, it is relevant from an optimization point of view as well a challenge over the years with respect the efficiency and optimal performance of the solution. Several techniques have been developed over the years to deal with this problem. In the present work, through comparisons, some available solvers were tested in relation to execution time and performance to obtain the optimal value. Some small and moderate instances were evaluated. The results obtained show that, even in instances where the distance matrix is small, some of the tested solvers presented difficulties to reach the solution, presenting large execution time and/or poor performance to reach the optimal path.

Keywords Optimization, Pyomo, Traveling Salesman.

Costa, Abner de Almeida. **Estudio del Problema del Vendedor Viajero mediante solvers**.66 páginas. 2022. Trabajo de finalización de curso (Graduación en Ingeniería Física) - Universidad Federal para la Integración Latinoamericana, Foz do Iguaçu, 2022.

Resumen

El problema del viajante de comercio es un problema de alto impacto social y es relevante desde el punto de vista de la optimización y un desafío a lo largo de los años en cuanto a la eficiencia y eficacia de la solución. A lo largo de los años se han desarrollado varias técnicas para abordar este problema. En el presente trabajo, a través de comparaciones, se probaron algunos solucionadores disponibles en relación con el tiempo de ejecución y el rendimiento para obtener el valor óptimo. Se evaluaron algunas instancias de tamaño pequeño y moderado. Los resultados obtenidos muestran que, incluso en instancias donde la matriz de distancia es pequeña, algunos de los solucionadores probados presentaron dificultades tanto en el tiempo de ejecución como en el rendimiento con respecto a la ruta óptima.

Palavras chave: Optimización, Pyomo, Vendedor Viajero.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Problemas de otimização aparecem nos mais variados campos de aplicação e, em especial, são de grande importância os problemas de otimização linear inteira. Desde a introdução do método Simplex por George Datzing, muitas outras técnicas foram criadas com o propósito de resolver problemas de otimização lineares e não-lineares. Atualmente existe uma grande quantidade de solvers disponíveis para o uso em tarefas de minimização de problemas. No presente trabalho avaliamos a performance de alguns solvers quando aplicados ao Problema do Caixeiro Viajante.

1.1 CONTEXTUALIZAÇÃO DO PROBLEMA

O problema do caixeiro viajante, é um problema de grande importância segundo (GOLDBARG; LUNA, 2000) que ressalta três elementos que a justificam, a grande aplicação prática, uma enorme reação com outros modelos e uma grande dificuldade de resolução exata, ainda mais, segundo (GOLDBARG; LUNA, 2000) é um dos problemas mais conhecidos problemas matemáticos. Os problemas lidam em sua maior parte com tours ou passeios sobre pontos de demanda e oferta, e acrescenta, que dentre os tipos de passeios um dos mais importantes é denominado Hamiltoniano e cujo nome é devido a William Rowan Hamilton que em 1857 propôs um jogo que denominou Around the World cuja história pode ser encontrada em (COOK, 2012). William, de fato, não foi o primeiro a propor tal problema, mas seu jogo o popularizou.

Em (WOLSEY, 1998) o autor expõe o problema do caixeiro viajante como um problema de programação inteira, que está dentro do conjunto de problemas de programação linear. Segundo (WOLSEY, 1998) o problema do caixeiro viajante é um problema combinatório, no sentido que, é um subconjunto de um conjunto finito, e utilizando a técnica de enumeração é possível resolvê-lo. Porém ao contar o número de possibilidades (WOLSEY, 1998) mostra que o problema tem tamanho da ordem de $(n-1)!$ concluindo que usando a técnica de enumeração completa só é possível resolver problemas para valores muito pequenos de n, e não somente o problema do caixeiro viajante, mas também os problemas de programação inteira.

Segundo (GOLDBARG; LUNA, 2000) o problema do caixeiro viajante recebeu diversas formulações, dentre elas, algumas podem ser consideradas canônicas tanto por sua larga difusão na literatura especializada, como por desenvolverem modos peculiares de caracterização do problema. Dentre elas a formulação de Dantzing, Fulkerson e Jhonson (DFJ), (1954) formula o caixeiro viajante como um problema de programação binária sobre um grafo que é o trabalho referência para o problema. Outra formulação importante, é a de Miller, Tuckler e Zemlin (MTZ), (1960) denominada como Folha de Cravo, ambas formulações possuem um número de restrições de subtours de ordem O(2^n). Os autores (GOLDBARG; LUNA, 2000) citam ainda mais duas formulações a de Fox, Gavish e Graves (1980) denominada Time Dependent e a formulação de Claus de Padberg e Sung (1988), sendo as mais importantes segundo (GOLDBARG; LUNA, 2000).

Tendo em vista a dificuldade para enumerar os problemas de programação inteira, diversas técnicas foram desenvolvidas, uma das técnicas exatas mais famosas é o método Simplex que é empregado por solvers da atualidade como o Coin-Or Branch-and-Cut de código aberto, o IBM ILOG CPLEX da IBM, e GLPK, a técnica foi desenvolvida por G. Dantzing (1947). Os solvers CBC e GLPK utilizam ainda outra técnica exata chamada Branch-and-Bound, solver CBC aplica essa técnica em conjunto com a técnica Branch-and-cut, que segundo (GOLDBARG; LUNA, 2000) trata-se de uma técnica enumerativa, portanto exata. Segundo (MACULAN; FAMPA, 2004) em 1972 foi apresentado um problema teórico por Klee e Minty para ser resolvido pelo método Simplex, o problema deixou aberta uma questão quanto a existência de um método eficiente para resolver problemas de programação linear uma vez que o simplex executava $2^n - 1$ iterações para solucionar o problema.

Também segundo (MACULAN; FAMPA, 2004), em 1979 Khachian propôs um método para solução polinomial dos problemas de programação linear, o método ficou conhecido como método dos elipsoides, que apesar da grande importância teórica se mostrou ineficaz. Em 1984 Karmarkar revolucionou a área de programação linear com um algoritmo de complexidade polinomial e bom desempenho para problemas práticos, essa publicação deu origem ao método de Pontos Interiores. A referência (MACULAN; FAMPA, 2004) ainda acrescenta que comparando-se os métodos de pontos interiores com Simplex, os primeiros serão melhores se critérios teóricos forem considerados, porém, na prática ambos competem até hoje.

Os grafos, elementos da formulação de DFJ, se tornaram importantes no ramo da pesquisa operacional. Segundo (HART et al., 2013) nos últimos anos a teoria de grafos se estabeleceu como uma importante ferramenta em várias áreas dentre elas a pesquisa operacional, destacando portanto a aplicação da teoria não somente na formulação do caixeiro viajante, mas a toda a área de pesquisa operacional. Por fim chegamos aos métodos heurísticos, que não produzem respostas exatas, porém

garantem uma boa resposta para os grandes problemas de programação inteira, em um tempo relativamente bom.

1.2 OBJETIVOS

Na sequência, seguem os objetivos gerais e específicos que nortearam o trabalho:

- Objetivo principal - O presente trabalho teve como objetivo principal comparar solvers na resolução de instâncias do caixeiro. Os solvers selecionados foram o Coin-Or Branch-and-Cut de código aberto, o IBM ILOG CPLEX da IBM e GLPK aplicados em instâncias do Caixeiro Viajante. Para uma realização dinâmica das comparações foi utilizado a biblioteca Pyomo, uma ferramenta que traduz o problema para cada um dos solvers. Para efeitos de comparação e controle, também foi testado um solver implementado no MatLab, da MathWorks, de autoria de (NARAYANAN, 2022).
- Objetivo específico – O presente trabalho teve como objetivo específico compreender o funcionamento da biblioteca Pyomo e sua funcionalidade para o testes de vários solvers em uma base de dados.

1.3 ORGANIZAÇÃO DO TRABALHO

O trabalho esta organizado em seis capítulos sendo o capítulo 1 destinado a introdução. No capítulo 2, estão os fundamentos sobre os quais o trabalho foi construído, sendo eles uma abordagem seletiva e simplificada sobre teoria de grafos e aspectos elementares sobre programação inteira. No capítulo três é apresentada duas das principais formulações do problema do caixeiro viajante explicando o problema e, de maneira resumida, alguns dos métodos de solução. Os métodos comentados são utilizados pelos Solvers empregados na análise das instâncias do problema. No quarto capítulo, é apresentado um pequeno manual prático do uso da biblioteca Pyomo do Python, a qual é uma interface para a implementação e resolução de problemas de programação inteira, disponibilizando uma função para a chamada de um solver que esteja instalado no computador do usuário e permitindo o uso de vários programas através de uma única linguagem de programação. No capítulo cinco são apresentadas as instâncias do problema que foram escolhidas para análise, os solvers utilizados, especificação das condições sobre as quais o problema foi analisado e os resultados obtidos. No capítulo 6 é feita a análise dos resultados e conclusão do trabalho.

2 FUNDAMENTOS

No presente capítulo usaremos as referências (GOLDBARG; LUNA, 2000), (TRUDEAU, 1993), (WOLSEY, 1998) e (WILSON, 1998) para descrever aspectos da Programação Linear Inteira bem como alguns elementos da teoria dos Grafos. No entanto, não temos como objetivo apresentar um material completo sobre o tema e sim apresentar tópicos úteis para o desenvolvimento do trabalho. Estudar em detalhes as referências citadas pode ajudar os que tenham interesse em aprofundar o tema.

2.1 PROGRAMAÇÃO INTEIRA

O conceito conhecido como programação inteira é o nome dado a uma classe de problemas de otimização em que as variáveis não assumem valores no domínio contínuo. Uma vez que as soluções são procuradas em um domínio discreto, esses problemas compõe um conjunto maior de problemas chamado Programação Linear. Os problemas de programação inteira se dividem em problemas puros e mistos.

Problemas puros de programação inteira, têm variáveis de decisão discretas sem exceção, e por isso são chamados puros. Problemas mistos, são os problemas onde pelo menos uma das variáveis é inteira com as demais contínuas. Suponha então o seguinte problema de otimização dado por

$$\begin{gathered}\textbf{Minimizar} \quad z = \mathbf{c} \cdot \mathbf{x} \\ \text{s.a.} \\ A.\mathbf{x} \leq \mathbf{b} \\ \mathbf{0} \leq \mathbf{x} \quad \textbf{e} \quad \mathbf{x} \in \mathbb{Z}^n,\end{gathered}$$

com vetor $\mathbf{c}$ dado. Este problema trata-se de um exemplo de programação linear inteira puro. Para um problema misto, é necessário pelo menos duas variáveis, com uma delas sendo discreta e a outra contínua. No exemplo abaixo, $\mathbf{x} \in \mathbb{R}^n$ e $\mathbf{y} \in \mathbb{Z}^n$, temos

$$\begin{gathered}\textbf{Minimizar} \quad z = \mathbf{c} \cdot \mathbf{x} + \mathbf{d} \cdot \mathbf{y} \\ \text{s.a.}\end{gathered}$$

$$W\mathbf{x} \leq \mathbf{0}$$
$$\mathbf{0} \leq \mathbf{y},$$

com $\mathbf{c} \in \mathbb{R}^n$ e $\mathbf{d} \in \mathbb{R}^n$ valores dados do problema. Estes exemplos ilustram situações que podem acontecer em problemas de otimização. A introdução de variáveis discretas pode adicionar várias dificuldades para a obtenção de solução por meio de alguma técnica.

Em um problema de otimização onde as variáveis de decisão são discretas, é possível que por intuição pensar que o ponto pertencente ao domínio inteiro que está mais próximo da solução ótima contínua, seja o ponto ótimo do caso discreto, porém isso não é sempre verdadeiro. Considere portanto o seguinte problema de (GOLDBARG; LUNA, 2000):

$$\textbf{Maximizar} \quad z = x_1 + 19x_2$$
$$\text{s.a}$$
$$x_1 + 20x_2 \leq 50$$
$$x_1 + x_2 \leq 20$$
$$x_1, x_2 \text{ variáveis inteiras},$$

cuja solução ótima, quando considerado o domínio contínuo, é: $x_1 = 18,42$, $x_2 = 1,57$, com ótimo $z = 48,42$. Tal solução, entretanto, não pode ser encontrada no domínio das restrições. Ao tentar aproximar o resultado, para pontos inteiros mais próximos obtemos a seguinte tabela:

x_1	x_2	z
19	2	inviável
19	1	38
18	2	inviável
18	1	37

Tabela 1: Tabela comparando candidatos em comparação ao conjunto factível.

A Tabela 1 mostra que os valores de x_1 e x_2 mais próximos ao ponto de ótimo, ou não estão no domínio restrito (variáveis contínuas) ou quado estão no domínio restrito apresentam valores bem menores que o ótimo que se obtém no domínio contínuo. A solução para as variáveis no domínio inteiro é $x_1 = 10$, $x_2 = 2$ onde $z = 48$. portanto, a aproximação do valor ótimo de um problema no domínio inteiro, através dos inteiros mais próximos da solução no domínio contínuo não é necessariamente um máximo global no domínio discreto, e isso torna necessário testar todos os pontos no domínio do problema para conhecer a solução exata.

Em várias situações do cotidiano, os problemas que surgem, exigem alguma otimização, onde, as variáveis de decisão, ou pelo menos uma delas assume valores

discretos. Quando isso ocorre, a solução do problema de otimização pode, mas não precisa coincidir com a solução do mesmo problema para variáveis contínuas, e ainda mais, pode nem mesmo estar na direção do valor ótimo, pois, tal variável discretizada pode não estar definida próximo à solução contínua. Em geral, o que ocorre é que precisamos aproximar a solução contínua, para os valores discretos próximos, assumindo o que gera a melhor solução. Além disso, problemas de otimização inteira puros, muitas vezes podem ser usados para resolver problemas de otimização combinatória.

Em um problema de otimização combinatória, por exemplo, é dado um conjunto $N = [1, 2, ...n]$, pesos c_j, e um subconjunto S contido em N. Como um exemplo considere tentar resolver o seguinte problema, formulado por (WOLSEY, 1998), e dado por

$$\min_{S \subseteq N} \quad \left[\sum_{j \in S} c_j : S \subseteq N \right]. \tag{1}$$

Existem diversas combinações de subconjuntos de N e os pesos c_j são discretos, de modo que torna-se necessário testar cada um dos subconjuntos a fim de determinar a solução exata uma vez que a solução é um subconjunto de N. Os problemas de otimização linear inteira ao contrário dos problemas de otimização linear contínuos, não possuem em geral algoritmos eficientes para obter a solução exata, isso se deve ao grande número de possíveis soluções ou subconjuntos que precisam ser testados. Portanto, quanto maior o tamanho do problema, maior é o número de possibilidades que precisam ser testadas e isso implica em um fenômeno denominado complexidade computacional quando atacados do ponto de vista da "força bruta". A complexidade computacional é medida proporcionalmente ao número de operações que um computador precisa realizar para resolver o problema. As possibilidades de possíveis soluções de alguns problemas variam na ordem de 2^n, outros na ordem de $n!$, ou $(n-1)!$ que é o caso do problema do caixeiro viajante que discutiremos adiante neste capítulo. Na Tabela 2, abaixo, extraída de (WOLSEY, 1998) observa-se a explosão do número de operações do problema com o aumento de n.

n	$log(n)$	$n^{0.5}$	n^2	2^n	$n!$
10	3.32	3.16	10^2	1.02×10^3	3.6×10^6
100	6.64	10	10^4	1.27×10^{30}	9.33×10^{157}
1000	9.97	31.62	10^6	1.07×10^{301}	4.02×10^{2567}

Tabela 2: Tabela baseada em (WOLSEY, 1998) demonstrando como a quantidade de elementos afetam as operações de acordo com as técnicas de resolução do problema.

Pelo motivo citado acima, problemas de otimização linear inteira não possuem algoritmos eficientes para o cálculo da solução exata do problema. Para isso desenvolveu-se outras técnicas que não encontram uma solução ótima exata mas

encontram uma solução viável, são os chamados algoritmos Heurísticos ou aproximativos. Segundo a definição de (GOLDBARG; LUNA, 2000) uma heurística é uma técnica que busca alcançar uma boa solução utilizando um esforço computacional considerado razoável, sendo capaz de garantir a viabilidade ou a otimalidade da solução encontrada, ou ainda, em muitos casos, ambas, especialmente nas ocasiões em que essa busca partir de uma solução viável próxima ao ótimo.

Uma Heurística evita calcular todos as possíveis soluções, no caso, o que se faz é escolher um grupo de possíveis soluções e tomar a melhor solução dentre as possibilidades consideradas. Existem duas técnicas básicas que classificam as heurísticas pelo método de busca (GOLDBARG; LUNA, 2000).

- Buscar examinar um número crescente de combinações entre as variáveis selecionadas em um determinado estágio da decisão e as variáveis dos estágios seguintes (intensificação ou melhoria na qualidade da solução).
- Buscar considerar um número cada vez maior de variáveis em cada nível (diversificação ou aumento do alcance [diâmetro] da busca).

Nosso interesse, no presente trabalho, é estudar o Problema do Caixeiro Viajante (PCV) e para isso alguns comentários sobre Grafos são apresentados na próxima seção.

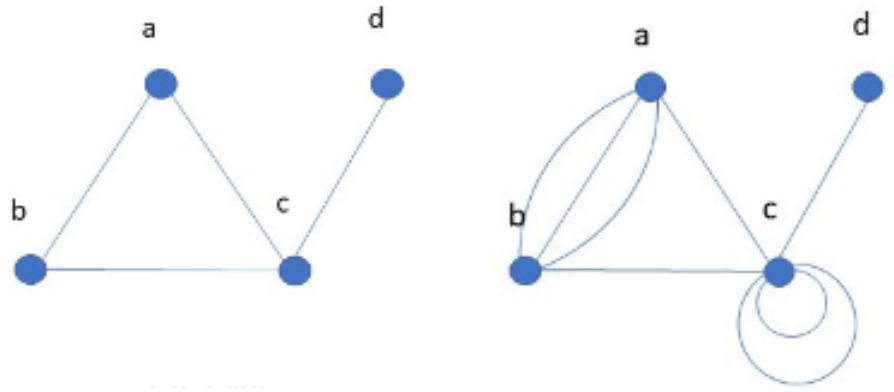
2.2 COMENTÁRIOS SOBRE GRAFOS

De acordo com (SIPSER, 2007), um grafo consiste de um conjunto de pontos com linhas conectando alguns deles. Aos pontos denomina-se nós (ou vértices) e as linhas denominamos arestas. Neste contexto, um grafo simples G é formado por um conjunto não vazio de vértices $V(G)$ e um conjunto finito de arestas $A(G)$, onde cada aresta une dois vértices quaisquer. Uma aresta que une dois vértices a e b é denominada ab. Para o caso ilustrado na Figura 1 (à esquerda), temos $V(G) = \{a, b, c, , d\}$ e $A(G) = \{ab, ac, bc, cd\}$ conjunto dos vértices e arestas.

Os grafos podem ter dois vértices unidos por mais de uma aresta, é possível ainda permitir que uma aresta ligue um vértice a ele mesmo que são chamados loops. Os grafos com esse tipo de arestas são chamados grafos gerais como ilustrado na Figura 1 (à direta). Para um grafo G_1 com um par de vértices ligados por mais de uma aresta e com um vértice com loops, temos a seguinte representação,$V(G_1) = \{a, b, c, d\}$ e $A(G_1) = \{ab, ab, ab, ac, bc, cc, cc, cd\}$.

Isomorfismo é o nome dado a uma característica comum a dois grafos. Sejam dois grafos G_1 e G_2 onde em cada grafo nenhum par de vértices está ligado por mais de uma aresta, de modo que o número de arestas em G_1 e G_2 são os mesmos.

Figura 1: Representação de um Grafo Simples (à esquerda) e representação geral (à direita)



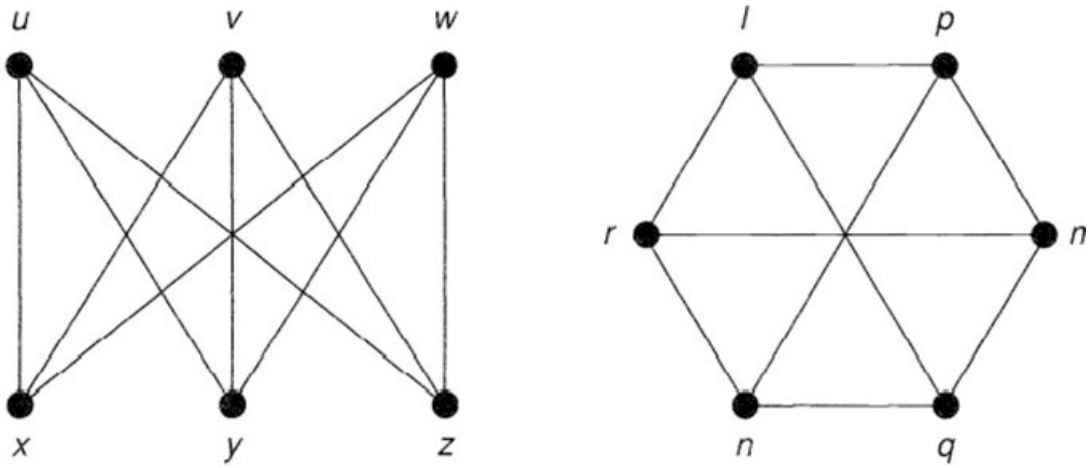
Fonte: O autor

Nos grafos representados na Figura 2 podemos notar que ambos possuem nove arestas e não possuem mais de uma aresta ligando cada par de vértices, portanto pela definição são isomórficos. Observe que se os vértices não forem diferenciados, isto é, nomeados, o número de isomorfismos diminui, por exemplo o caso de grafos com 3 vértices, se os vértices forem nomeados, teremos isomorfismos para o caso de nenhuma aresta, uma aresta, duas arestas e 3 arestas como ilustrado na Figura 3. Porém se os vértices não forem nomeados, não teremos tais isomorfismos, uma vez que, os vértices não possuem diferenças entre si. A nomeação de vértices ocorre sempre quando é necessário diferenciar um vértice do outro, isto é, os objetos abstraídos para a figura do vértice são diferentes e essa diferença é relevante no resultado que se quer obter utilizando o grafo.

Nós dizemos que dois vértices de um grafo são adjacentes se existir uma aresta que ligue um vértice ao outro. Similarmente, duas arestas são adjacentes se ambas tiverem um mesmo vértice em comum. Além disso, o grau de um vértice corresponde ao número de arestas ligadas a este vértice. Para vértices com loops cada aresta que forma um loop, é contabilizada duas vezes. Dessa forma temos que um vértice de grau um, é um vértice final, vértices isolados tem grau zero. Na Figura 1 (à direita), podemos observar que d é um vértice final, possuindo grau 1, e na Figura 1 (à esquerda), os vértices a e b tem grau 2.

É importante notar que a soma de todos os graus de todos os vértices é sempre um número par, esse resultado foi obtido primeiramente por Leonard Euler em 1736, e é conhecido como lema de Handshaking. Esse lema pode ser deduzido interpretando cada aresta como uma pessoa e cada vértice ligado por uma aresta, como uma das mãos da pessoa, sendo assim um grafo com n arestas (pessoas) possui $2n$ ligações (graus ou mãos). Um corolário imediato é que num grafo qualquer o número de

Figura 2: A figura abaixo ilustra o Isomorfismo entre dois Grafos



Fonte: Graph Theory (WILSON, 1998)

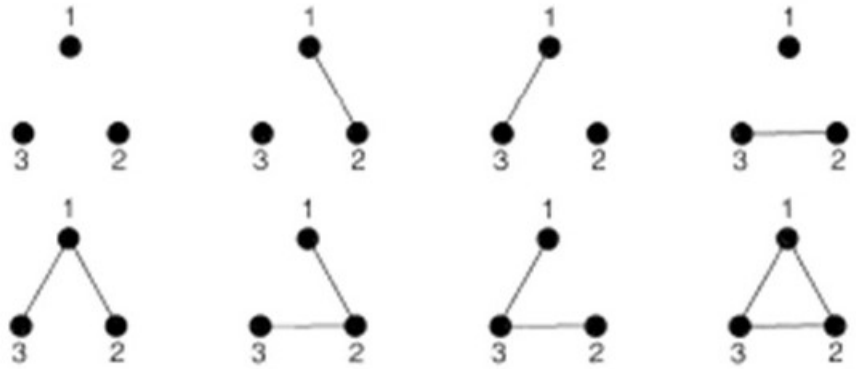
vértices com grau ímpar é par. Representaremos o grau de um vértice pela seguinte função:

$$g(v) = deg(v). \tag{2}$$

Um subgrafo de G é um grafo cujos vértices pertencem à $V(G)$ e cujas arestas pertencem à $A(G)$, representa-se um subgrafo como a subtração de vértices e arestas de um grafo. Seja um grafo G, um conjunto de arestas $B(G)$, subconjunto de $A(G)$ e um conjunto de vértices $S(G)$ subconjunto de $V(G)$, onde $B(G)$ são arestas ligadas $S(G)$, um subgrafo de G é representado por $G - S$ que excluí todos os vértices de G contidos em S e suas respectivas arestas incidentes $B(G)$. Quando um vértice é retirado de um grafo e mantêm-se as ligações indiretas realizadas por intermédio do vértice retirado dizemos que o grafo foi contraído. Denota-se por $G\backslash e$ os grafos contraídos. A Figura 4 ilustra esse caso, onde o vértice v é contraído.

Embora seja didático representar grafos graficamente, essa representação não é viável para manipulação. Existem duas matrizes principais que podem definir grafos, a matriz de adjacência e a matriz de incidência. Na matriz de adjacência $A(nXn)$ cada elemento $a(i, j)$ recebe o valor correspondente ao número de adjacências entre os vértices i e j, onde o número total de vértices do grafo é n.

Figura 3: Isomorfismo para vértices nomeados



Fonte: Graph Theory (WILSON, 1998)

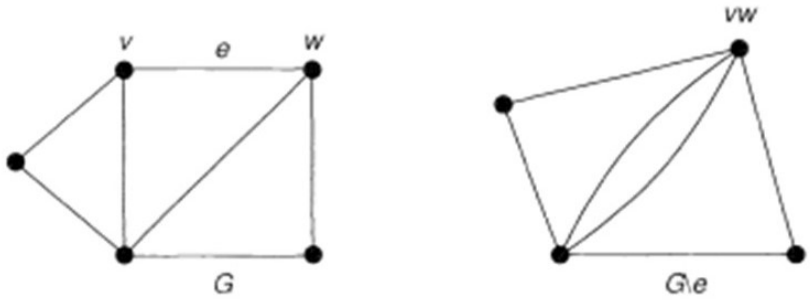
$$A = \begin{array}{c} \\ i_1 \\ i_2 \\ i_3 \\ i_4 \end{array} \begin{array}{c} \begin{array}{cccc} j_1 & j_2 & j_3 & j_4 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 2 \\ 0 & 1 & 0 & 1 \\ 1 & 2 & 1 & 0 \end{bmatrix} \end{array} \tag{3}$$

A matriz acima, é a matriz de adjacências de um grafo com 4 vértices como o representado no lado direito da Figura 4 . Quando dois vértices (i, j) estão ligados o elemento da matriz $A(i, j)$ recebe um número inteiro correspondente ao úmero de adjacências entre os vértices i e j. A matriz de incidências $I(nXm)$ cujos elementos $p(n, m)$ são 1 se o vértice $i = [1, 2, ..., n]$ é ligado pela aresta $j = [1, 2, ..., m]$ e 0 caso contrário, nessa representação n é número de vértices e m o número de arestas. Para o grafo de 5 vértices da Figura 4 onde o vértice v é o segundo vértice, w é o quarto vértice e os vértices abaixo destes respectivamente são o terceiro e quarto vértice.

$$I = \begin{array}{c} \\ i_1 \\ i_2 \\ i_3 \\ i_4 \\ i_5 \end{array} \begin{array}{c} \begin{array}{ccccccc} j_1 & j_2 & j_3 & j_4 & j_5 & j_6 & j_7 \end{array} \\ \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \end{array} \tag{4}$$

Definição de Passeio: Um passeio em um grafo G é uma sequências finita de arestas que pode ser representado como: $v_0v_1, v_1v_2...v_{m-1}v_m$ ou ainda $v_0 \to v_2... \to v_m$

Figura 4: Contração de um grafo



Fonte: Graph Theory (WILSON, 1998)

no qual duas arestas consecutivas podem ser adjacentes ou idênticas. O número de arestas de um passeio é chamado comprimento do passeio. Passeios possuem uma definição muito geral, por isso, define-se algumas restrições, para identificar casos particulares.

Definição de Trilha: Se um passeio possui todas as restas distintas, este passeio recebe o nome de Trilha. Se esse passeio além de ter todas as arestas distintas tem todos os vértices distintos (exceto o caso de $v_0 = v_m$) então esse passeio é chamado caminho.

Um caminho ou uma trilha são fechados se, $V_0 = v_m$, e um caminho fechado contendo pelo menos uma aresta é chamado ciclo. Temos portanto que loops são ciclos. Note que um grafo é conexo, se e somente se, existe um caminho entre cada par de vértices, isto é se existirem dois vértices de um grafo que não podem ser ligados por nenhum caminho, esse grafo é desconectado.

Definição de Grafos Hamiltonianos: Um grafo é dito hamiltoniano se existe uma trilha fechada que passa por cada vértice apenas uma vez, tal trilha deve ser um ciclo. O ciclo é chamado Ciclo Hamiltoniano e o grafo, Grafo Hamiltoniano.

Teorema (Ore, 1960): Se G é um grafo simples e com $3 \leq n$ vértices e se:

$$n \leq deg(v) + deg(w) \tag{5}$$

para cada par de vértices não adjacentes v e w, então G é Hamiltoniano.

Corolário (Dirac, 1952): Se G é um grafo simples com $3 \leq n$ vértices e se $deg(v) \leq 2$ para cada vértice v, então G é Hamiltoniano.

O problema de interesse do presente trabalho é o do Caixeiro Viajante, por isso no próximo capítulo serão discutidos alguns aspectos do mesmo. Por definição, no problema do Caixeiro Viajante, estamos interessados em uma rota ótima que passe

uma única vez por todas as cidades. Em outras palavras, o grafo que descreve o problema deve ser Hamiltoniano.

3 PROBLEMA DO CAIXEIRO VIAJANTE

No presente capítulo são discutidos os aspectos teóricos do problema do caixeiro viajante (PCV). Matematicamente o PCV é um problema de otimização combinatória, com complexidade computacional do tipo NP- completo. Discutiremos sem aprofundar o conceito de complexidade mas para um entendimento mais detalhado sobre complexidade computacional recomendamos a leitura de (SIPSER, 2007). Na prática, como descrito em (COOK, 2012), o PCV aparece em muitas situações cotidianas, como entrega de faturas, entregas de correio, delivery de alimentos, transporte escolar e até em situação que envolvem problemas físicos.

3.1 DESCRIÇÃO DO PROBLEMA

Considere um caixeiro viajante sediado em uma cidade chamada Orange. Ele precisa deslocar-se pelas cidades Floripa, Frísia, Antuérpia e retornar para Orange. O trajeto de miníma distância entre as cidades garante que ele pode ter uma boa economia de combustível. Colocando estas rotas em uma tabela temos

	Orange	Floripa	Frísia	Antuérpia
Orange	0	54	17	79
Floripa	54	0	49	109
Frísia	17	49	0	91
Antuérpia	79	109	91	0

Tabela 3: Tabela contendo as cidades e suas distâncias, em metros, a serem percorridas pelo viajante.

O problema do caixeiro viajante consiste em encontrar a rota que passe por n cidades, onde, cada cidade deve ser visitada apenas uma vez, retornando à cidade de origem após visitar a última cidade, cuja distância total percorrida seja mínima. A distância entre cada uma das cidades é representada pela matriz D, onde cada elemento $d(i, j)$ representa a distância entre as cidades i e j. Pela simetria do problema, $d(i, j) = d(j, i)$. E, em relação ao viajante de nosso exemplo, a matriz de distância é

$$D = \begin{bmatrix} 0 & 54 & 17 & 79 \\ 4 & 0 & 49 & 109 \\ 17 & 49 & 0 & 91 \\ 79 & 109 & 91 & 0 \end{bmatrix}. \tag{6}$$

Note ainda que a representação é semelhante a da matriz de um grafo, cada vértice representa uma cidade e cada aresta tem um valor associado, que corresponde a distância entre os dois vértices adjacentes aos quais ela incide. De forma sucinta, o principal objetivo é percorrer estas cidades pelo menor caminho. No caso deste exemplo, pela simplicidade, podemos traçar todas as rotas possíveis na seguinte tabela:

Rota	Distância total
O-Fl-Fr-A-O	54+49+91+79=273
O-Fl-A-Fr-O	54+109+91+17=271
O-Fr-Fl-A-O	17+49+109+79=254
O-A-Fr-Fl-O	79+91+49+54=273
O-A-Fl-Fr-O	79+109+49+17=254
O-Fr-A-Fl-O	17+91+109+54=271

Tabela 4: Tabela contendo as rotas possíveis para o problema.

No exemplo descrito acima por meio da análise das rotas é fácil encontrar as soluções do problema. Porém, quando o número de cidades aumenta o número de possibilidades aumenta ainda mais, de maneira exponencial, sendo um problema de alta complexidade computacional para ser analisado por meio de "força bruta". Mais especificamente, o objetivo é encontrar o Caminho Hamiltoniano cuja soma dos valores associados a cada aresta seja mínimo. Temos então $(n-1)!$ rotas possíveis. Sendo este, de acordo com (SIPSER, 2007), um problema de otimização NP-hard. É de nosso interesse obter uma resposta aproximada para o problema já que, até o momento, não foi reportado se existe algum algoritmo em tempo polinomial que pode resolver este problema.

O problema do caixeiro viajante foi abordado de diversas maneiras, as quais estão registradas na literatura com o objetivo de viabilizar a solução do problema. Algumas dessas maneiras de abordagem ficaram famosas por características peculiares sendo consideradas canônicas. Uma dessas formulações foi proposta em (DANTZIG; FULKERSON; JOHNSON, 1954) e é conhecida como DFJ. Nesta formulação, o PCV foi formulado com uma função objetivo, sujeito a restrições binárias como segue e cuja representação é dada por um grafo G. A formulação matemática do problema é dada

por

$$\textbf{Minimizar} \quad z = \sum_{i=1}^{n} \sum_{j=1}^{n} c_{i,j} x_{i,j} \tag{7}$$

sujeito a:

$$\sum_{i=1}^{n} x_{i,j} = 1, \forall j \in N \tag{8}$$

$$\sum_{j=1}^{n} x_{i,j} = 1, \forall i \in N \tag{9}$$

$$\sum_{i,j \in S} x_{i,j} \leq |S| - 1, \forall S \subset N \tag{10}$$

$$x_{i,j} \in \{0, 1\}, \forall i, j \in N. \tag{11}$$

Nesta formulação proposta, quando a aresta (i, j) do grafo é escolhida como parte da solução, a variável $x_{i,j}$ binária assume valor igual a um e permanece zero caso contrário. Na restrição (10), S é um subgrafo de G e $|S|$ representa o número de vértices de S. As viagens de uma cidade para ela mesma são proibidas, isto é, não há loops e assim as variáveis $x_{i,i}$ não existem, isso implica que teremos uma redução de $N \times N$ variáveis para $N \times (N - 1)$ variáveis inteiras. As restrições (8) e (9) garantem que ao sair de um vértice qualquer do grafo, somente um novo vértice seja escolhido como destino, ou ainda, que um vértice seja alcançado por um e somente um outro vértice, isso garante que não haja passeios paralelos entre duas cidades dentro do passeio geral. Por fim a restrição (10) vai eliminar passeios pré-hamiltonianos, isto é, passeios hamiltonianos que não incluem todas as cidades, tornando-os ilegais.

Pela definição de circuitos hamiltonianos o número de arestas é menor ou igual ao de vértices e, para um subgrafo de cinco vértices, teremos que os possíveis passeios devem conter até quatro arestas. De fato para um grafo com quatro arestas e cinco vértices é possível existir um caminho hamiltoniano, porém estes pré hamiltonianos de ordem menor já foram eliminados quando S era um subgrafo com quatro vértices. Consequentemente, os pré-hamiltonianos são eliminados dos menores para os maiores.

Comentamos sobre a formulação DFJ por ser pioneira, outras formulações foram apresentadas em anos posteriores e uma formulação diferente foi proposta por (MILLER; TUCKER; ZEMLIN, 1960) uma generalização do problema conhecida como formulação MTZ. Os autores formularam o problema da seguinte forma: Um caixeiro viajante precisa passar por $n - 1$ cidades indexadas por $2, ..., n$ somente uma única vez. O depósito, que é o ponto de partida, e ponto de retorno corresponde ao índice 1. Neste caminho, o caixeiro pode retornar ao depósito t vezes (incluindo o último retorno) não devendo ele visitar mais do que p cidades diferentes em uma única viajem

e, neste caso, significa que o viajante não vai retornar para o depósito. Para $t = 1$ e $n \geq p$ temos a formulação usual do Caixeiro Viajante.

Do ponto de vista de um problema de otimização temos de (GOLDBARG; LUNA, 2000) e (MILLER; TUCKER; ZEMLIN, 1960) que a precisa formulação fica dada por

$$\textbf{Minimizar} \quad z = \sum_{i=1}^{n} \sum_{j=1}^{n} c_{i,j} x_{i,j} \tag{12}$$

sujeito a:

$$\sum_{i=2}^{n} x_{i,1} = t \tag{13}$$

$$\sum_{i=1}^{n} x_{j,i} = 1, (j = 2, ..., n) \tag{14}$$

$$\sum_{j=1}^{n} x_{j,i} = 1, (i = 2, ..., n) \tag{15}$$

$$u_i - u_j + p x_{i,j} \leq p - 1, (2 \leq i \neq j \leq n) \tag{16}$$

$$u_i \geq 0, (2 \leq i \leq n) \tag{17}$$

$$x_{i,j} \in \{0, 1\}, \forall i, j \in N. \tag{18}$$

Os valores u_i são números reais arbitrários e as restrições sobre ele tem o objetivo de eliminar rotas do caixeiro que não passam pelo depósito. Na próxima seção uma visão geral dos métodos de solução é apresentada.

3.2 SOBRE ALGUNS MÉTODOS DE SOLUÇÃO

Neste capítulo abordaremos alguns dos métodos de resolução de Problemas de Programação Linear com melhores desempenhos, explicando sucintamente o funcionamento de de cada método.

3.2.1 Método Simplex

O método Simplex surgiu do Teorema Fundamental da Programação Linear como descrito em (DANTZIG, 1951), trata-se de um algoritmo que busca pela solução ótima a partir de uma solução básica viável. Uma solução viável está contida no conjunto restrito das variáveis determinado pela restrição do problema e chamado conjunto factível. O primeiro passo do algoritmo Simplex é que o problema esteja escrito na forma padrão. Colocar o problema na forma padrão basicamente é reescrever as restrições de desigualdades como igualdades, adicionando-se variáveis chamadas variáveis de folga, as quais recebem este nome exatamente por representarem um valor que ajusta

a relação entre as variáveis do problema para que estejam no domínio restrito. Dessa forma obtém-se um sistema $A\mathbf{x} = \mathbf{b}$ onde A é uma matriz $(m \times n)$ e n corresponde ao número de variáveis de folga adicionadas.

Segundo a definição 3.1 de (GOLDBARG; LUNA, 2000) Uma base B de uma matriz A $m \times n$ é uma matriz quadrada de m vetores coluna linearmente independentes em $\mathbb{R}^m$. As variáveis associadas a essas colunas denomina-se variáveis básicas. Desta forma é possível decompor o vetor de variáveis do problema em dois conjuntos, o conjunto de variáveis básicas e o conjunto de variáveis não-básicas. O vetor composto das variáveis básicas dadas por $\mathbf{x}_b = B^{-1}\mathbf{b}$, com variáveis não básicas nulas é chamado de solução básica. Quando uma solução básica tem componentes não negativos esta é chamada solução básica viável. O conjunto de vetores $\mathbf{x}$ dados por $A\mathbf{x} = \mathbf{b}$ é chamado conjunto de soluções viáveis $\mathcal{C} = \{\mathbf{x} \quad | \quad A\mathbf{x} = \mathbf{b}\}$. Não apresenta dificuldade mostrar que o conjunto $\mathcal{C}$ é na realidade um conjunto convexo.

O Teorema 3.1 de (GOLDBARG; LUNA, 2000) diz que o conjunto de soluções viáveis de um problema de programação linear é um conjunto convexo e o teorema 3.2 de (GOLDBARG; LUNA, 2000) acrescenta que toda solução básica viável do sistema $A\mathbf{x} = \mathbf{b}$ é um ponto extremo do conjunto $\mathcal{C}$ de soluções viáveis. Além disso, o teorema 3.3 de (GOLDBARG; LUNA, 2000) comprova a existência de uma correspondência biunívoca entre os pontos extremos do conjunto de soluções viáveis e as soluções básicas. Os resultados teóricos deixam claro que o número de pontos extremos do conjunto de soluções viáveis é finito. A existência de uma solução viável implica necessariamente na existência de uma solução básica viável.

Com essas premissas bem definidas, chega-se então ao teorema 3.4 de (GOLDBARG; LUNA, 2000) que afirma que se uma função objetivo possui um máximo ou um mínimo finito, então pelo menos uma solução ótima é um ponto extremo do conjunto $\mathcal{C}$, e ainda, se a função objetivo assume o máximo ou mínimo em mais de um ponto do conjunto convexo $\mathcal{C}$ então ela toma o mesmo valor pra qualquer combinação convexa desses pontos. O algoritmo Simplex vai portanto testar uma sequência de soluções básicas viáveis em busca da solução ótima.

O algoritmo Simplex executa basicamente as seguintes etapas: Dado o problema de otimização na forma padrão, é preciso de uma solução básica viável inicial, essa solução como visto anteriormente é um dos pontos extremos do conjunto convexo dado pelas restrições. De posse da solução verifica-se se essa solução é ótima, isso é feito comparando com outra solução básica viável (extremo), se outro extremo possuir valor melhor para a função objetivo, o método caminha em direção ao melhor vértice. O processo termina, quando o ponto extremo corrente tem o melhor valor quando comparado com os demais. Para maiores detalhes sobre o método recomendamos a leitura de (DANTZIG, 1951) onde uma rica discussão é realizada.

3.2.2 Método de Ponto Interior

O método de Pontos Interiores baseia-se na ideia de procurar um valor ótimo da função objetivo partindo de um ponto inicial dentro do espaço convexo de possíveis soluções, isto é, um ponto que não é um ponto extremo, ao contrário do método Simplex, e indo em direção a um extremo considerando uma vizinhança por onde o caminho pode variar, que diminui conforme o ponto corrente se aproxima de um ponto extremo. Escolhe-se portanto uma direção de busca e um passo, que são atualizados a cada iteração e uma função barreira logarítmica, com peso atualizado a cada iteração.

Segundo (MACULAN; FAMPA, 2004), para a determinação da direção de descida é muito utilizado o algorítimo Afim-escala, este algorítimo procura uma direção de máximo aclive dentro de uma região de confiança dada por um elipsoide. Dada a direção de busca, um problema surge, pois o elipsoide de busca se torna cada vez menor quando se aproxima da fronteira do poliedro de soluções viáveis. Com isso é introduzida uma função logarítmica de barreira que mantém a busca no centro do poliedro, a inserção dessa função gerou o método conhecido como método de barreiras, que como afirma (MACULAN; FAMPA, 2004) é um caso particular dos métodos de penalidades. Dessa forma a função objetivo, para $x_j > 0$ e μ parâmetro positivo, é dada por

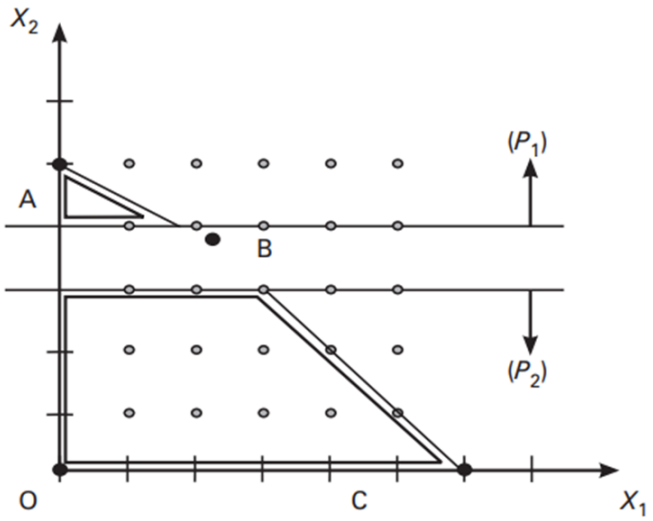
$$f(x, \mu) := c^T x + \mu \sum_{j=1}^{n} log(x_j) \tag{19}$$

Como também explicado em (MACULAN; FAMPA, 2004) aplicando as condições de Karush-Kunh-Tucker (KKT) chega-se as condições de otimalidade para o problema com barreira na forma primal-dual. A direção por onde se caminha nesse método é baseada na aplicação do método de Newton ao sistema de equações do problema primal-dual. Portanto este método se baseia em técnicas onde a função obedece os critérios de otimalidade descritas em (RIBEIRO; KARAS, 2013).

3.2.3 Método Branch and Bound

O método Branch and Bound,desenvolvido por (LAND; DOIG, 1960), é um método enumerativo relacionado com programação dinâmica, método este bem explicado em (SIPSER, 2007), que enumera as possíveis soluções em um grafo na forma de árvore, onde o primeiro nó representa o problema inicial. A medida que a árvore é formada, surgem subproblemas a cada nó. O algoritmo avalia se uma sub-árvore deve ser mantida e dividida para melhor análise ou se deve ser descartada com base no melhor resultado. O que de fato ocorre é o particionamento do espaço de soluções, de forma que o problema original é repetidamente decomposto em subproblemas sujeitos a novas restrições.

Figura 5: Contração de um grafo



Fonte: Otimização Combinatória e Programação linear (GOLDBARG; LUNA, 2000)

Como explicado em (MACULAN; FAMPA, 2004) e (WILHELM; KLEINA, 2022), o conjunto de soluções viáveis é dividido com base na comparação do problema restrito com o problema relaxado (variáveis podem assumir valores contínuos). O valor ótimo da solução contínua é admitido como limite superior, ou inferior dependendo do sentido da função objetivo. No ponto onde ocorre esse máximo é feito um corte no conjunto de soluções viáveis tomando uma das variáveis como discreta e dividindo o conjunto em duas partes. No exemplo abaixo retirado de (GOLDBARG; LUNA, 2000) temos que

$$\textbf{Maximizar} \quad z = 5x_1 + 8x2$$
$$\text{sujeito a:}$$
$$x_1 + x_2 \leq 6$$
$$5x_1 + 9x_2 \leq 45.$$

A solução ótima relaxada é encontrada em $x_1 = 9/4$ e $x_2 = 15/4$. Fazer um corte em x_2 significa separar o conjunto contínuo em duas partes, excluindo valores de x_2 contínuos pelo arredondamento para cima e para baixo, de modo que não há valores possíveis de x_2 entre 3 e 4, criando dois subconjuntos de soluções viáveis como ilustrado na Figura 5. Portanto, o problema inicial fica então dividido em dois subproblemas, que possuem limites inferiores e superiores. Cada subproblema criado é representado por um nó na árvore ao qual são associados dois outros nós correspon-

dentes aos limites superiores e inferiores. Assim forma-se uma árvore de busca pela solução ótima.

Um caminho que sai do nó raiz e vai para um nó folha é um elemento do espaço de soluções e os nós contidos nesse elemento são chamados de nós solução, os nós que satisfazem as restrições são chamados de nós resposta. Como mencionado anteriormente a cada nó adicionado como possível solução são aplicadas operações específicas do algoritmo para determinar se a busca deve ou não avançar em torno daquele nó. Como bem descrito em (WILHELM; KLEINA, 2022) algoritmo consiste de três componentes principais. Na primeira, o algoritmo calcula os limites inferiores e/ou superiores para o valor da função objetivo do subproblema dado. Na segunda, é estabelecida uma estratégia para selecionar o subespaço a ser investigado. Por fim, na terceira é composta de uma regra de ramificação de um subespaço no caso de, após a análise, o subespaço não possa ser descartado.

Para cada nó é computado um limite superior ou inferior que identifica a melhor solução que o nó pode gerar, assim num problema de minimização por exemplo, se o menor valor associado a um nó leva a uma solução maior que a solução corrente esse nó é descartado. Como bem detalhado em (LAND; DOIG, 1960), (MACULAN; FAMPA, 2004) e (WILHELM; KLEINA, 2022) as etapas do algoritmo podem ser resumidas da forma abaixo:

- Encontre a solução ótima para o modelo de programação linear com restrição de variáveis inteiras relaxadas;
- No nó raiz a solução relaxada é o limite superior e a solução inteira o limite inferior;
- selecione a variável com maior ramificação (parte fracionária), crie duas novas restrições para essa variável refletindo os valores se uma $\leq$ e outra $\geq$;
- crie dois nós para cada restrição nova;
- resolva o problema inicial com as novas restrições adicionadas aos nós.
- a solução relaxada é o limite superior de cada um desses nós, e a inteira máxima é o limite inferior
- se o processo produzir uma solução viável com menor valor inteiro limite inferior dentre qualquer nó a solução ótima foi atingida. Caso contrário volte ao terceiro item.

4 EXPLORANDO A INTERFACE PYOMO

Neste capitulo vamos descrever a ferramenta Pyomo seguindo de perto as importantes referências (HART et al., 2013), (CARVALHO; NETO, 2020) e o próprio manual do Pyomo dado em (PYOMO, 2022). Nestes materiais, um rico compêndio sobre o tema pode ser encontrado. O Pyomo consiste de uma bliblioteca existente na linguagem de programação Python e pode ser utilizado no contexto da resolução de problemas de Programação Linear. Esta biblioteca fornece sintaxe útil para descrever os modelos matemáticos, permite a formulação de modelos de grande porte de forma compacta e separa a declaração de modelo e dados. Não trata-se de um solver e sim uma interface que traduz os modelos de otimização, a serem resolvidos, para poderem ser usados em algum solver de interesse.

Os modelos escritos em linguagem Pyomo são traduzidos para a forma esperada pelo solver, dessa forma, é possível resolver o mesmo problema com diversos solvers, com pouca ou nenhuma modificação no problema. Importa-se o Pyomo da seguinte maneira:

```
import matplotlib.pyplot as plt
```

Trata-se de uma ferramenta de grande utilidade para Cientistas, Engenheiros e outros profissionais que necessitam resolver problemas de otimização no seu dia a dia.

4.1 MODELOS

Matematicamente, uma equação pode definir uma classe de problemas, isto é, para um grupo de problemas pode existir algumas características comuns que os definem, um exemplo desse tipo de problema é dado na Equação (20). Tais problemas são chamados abstratos pois não são específicos.

Um problema específico é frequentemente usado como exemplo para explicar uma classe de problemas, porém o exemplo não é um problema abstrato, e sim um problema concreto. Modelos matemáticos sem especificação, são modelos matemáticos abstratos, e modelos matemáticos especificados, são, modelos matemáticos concre-

tos. Por exemplo, as equações a seguir são um problema abstrato:

$$\textbf{Maximizar} \quad z = \mathbf{c} \cdot \mathbf{x} \tag{20}$$

$$s.a$$
$$A\mathbf{x} \leq \mathbf{b}$$
$$\mathbf{x} \leq \mathbf{0}.$$

Quando as variáveis são especificadas o modelo é então concreto. A Equação (21) mostra um exemplo de modelo concreto:

$$\textbf{Maximizar} \quad z = 2x_1 + 5x_2 \tag{21}$$

$$s.a$$
$$3x_1 < 0$$
$$x_2 > 0.$$

Em Pyomo é possível trabalhar com os dois tipos de formulação, de forma que podemos ter dois tipos de modelos, abstratos e concretos. Os modelos na biblioteca Pyomo são essencialmente objetos e estes objetos possuem componentes. Os componentes são usados para a definição de como deve ser o modelo e, portanto, ao declarar um modelo estamos criando um objeto que possui componentes. Tais componentes definem como o problema de otimização será formulado. Em outras palavras, o modelo fornece um esqueleto sobre o qual o problema de otimização pode ser estruturado.

Para definir um problema de otimização geral, precisamos definir variáveis, função objetivo, parâmetros e restrições. Para a modelagem com Pyomo temos ainda os conjuntos que servem para indexação. Um modelo abstrato é declarado como

```
modelo = pyEnv.AbstractModel(),
```

e um modelo concreto é declarado como:

```
modelo = pyEnv.ConcreteModel().
```

Se usarmos a função $pprint()$ obtemos respostas a respeito do modelo criado nos casos acima, obtém-se o seguinte resultado:

```
modelo.pprint()
>>> 0 Declarations:
```

O Pyomo define o modelos matemáticos como uma classe e, desta forma, quando declara-se um modelo estamos criando um objeto de tal classe o qual possui outros objetos que o definem. Tais objetos são definidos como Conjuntos, Variáveis, Objetivos, Restrições e Parâmetros. Tais objetos são caracterizados em concordância com a própria linguagem matemática.

4.2 PECULIARIDADES

A programação com Pyomo possuiu algumas peculiaridades na forma de indexar e de determinar valores para variáveis bem como para os componentes dos conjuntos. Nesta seção uma discussão é feita sobre tais peculiaridades.

Os objetos dentro de um modelo Pyomo podem ser declarados em duas formas: a primeira com valor inicial e a segunda de forma indexada isto é, num modelo contendo uma variável que pode assumir valores discretos. Tais valores discretos podem vir a depender de outros fatores e é possível declarar um valor de cada vez ou indexá-los e declará-los todos de uma única vez. Por exemplo, ao declarar variáveis é possível determinar o valor inicial dessa variável com o argumento $initialize$, porém a declaração de um grupo de valores necessita de uma forma mais complexa para determinar os valores iniciais de cada uma. Vamos exemplificar isto logo abaixo.

```
import pyomo.environ as pyEnv

modelo = pyEnv.ConcreteModel()

modelo.A = pyEnv.Set(initialize=[1,2,3])
modelo.B = pyEnv.Set(initialize=['Branco', 'Preto'])
modelo.x = pyEnv.Var()
modelo.y = pyEnv.Var(modelo.A, modelo.B)
modelo.o = pyEnv.Objective(expr=modelo.x)
modelo.c = pyEnv.Constraint(expr = modelo.x >= 0)

def d_rule(modelo, a):
    return a*modelo.x <= 0
modelo.d = pyEnv.Constraint(modelo.A, rule=d_rule)
```

A função Set é responsável pela declaração dos conjuntos, o argumento $initialize$ é usado para especificar qual é o valor inicial do conjunto. Os conjuntos A e B são portanto declarados dessa forma. A função Var é responsável pela declaração de variáveis, note que x não recebe nenhum valor, pois como indicado em (HART et al., 2013) e (CARVALHO; NETO, 2020), as funções sem argumentos apenas especificam como o objeto vai ser declarado.

A variável y é declarada com dois argumentos não nomeados, isto é, que não recebem um precedente que tipifique o argumento. Essa é a forma como a biblioteca Pyomo interpreta que a variável está sendo indexada. Com respeito a indexação, esta consiste de um ou mais argumentos não nomeados que aparecem antes de todos os demais argumentos nomeados. No caso acima temos um valor de y para cada par de indexadores $[i, j]$ pertencentes a A e B respectivamente, em outras palavras, temos uma matriz Y cujas linhas são indexadas por elementos do conjunto A e cujas colunas são indexadas por elementos do conjunto B, cujos elementos são valores de y para cada par $[i, j]$

O símbolo o que representa um objetivo e, que trata apenas de uma variável, e a restrição c também são definidos. A restrição d é uma restrição indexada, e para cada elemento de A, $a[i]$ temos uma restrição distinta. O argumento $initialize$ é usado para definir o valor da restrição, tal argumento chama a função d_rule a qual para cada elemento de A gera uma restrição diferente. Desta forma, o conjunto A faz o papel de indexar e também de definir as restrições. Se solicitarmos que seja impressa a restrição d temos a seguinte resposta:

```
d : Size=3, Index=A, Active=True
    Key : Lower : Body : Upper : Active
      1 :  -Inf :    x :   0.0 :   True
      2 :  -Inf : 2*x :   0.0 :   True
      3 :  -Inf : 3*x :   0.0 :   True
```

Conforme dito anteriormente a variável y embora declarada, não possui valores. Portanto, para determinar os valores de y temos três opções:

- Se um único escalar for dado como argumento, todos os valores da variável serão iguais a este escalar;
- Passar um dicionário Python, onde as chaves do dicionário são iguais as chaves do do índice definido para o componente;
- Chamar uma função que pode ser chamada para fornecer o valor pra cada componente da indexação;

Veja o exemplo abaixo:

```
import pyomo.environ as pyEnv

modelo = pyEnv.ConcreteModel()

l={1:1.5, 2:4.5, 3:5.5}
```

```
modelo.A = pyEnv.Set(initialize=[1,2,3])
modelo.x = pyEnv.Var(modelo.A, initialize=3.14)
modelo.y = pyEnv.Var(modelo.A, initialize=1)

def regra(m, i):
    return float(i) + 0.5

modelo.z = pyEnv.Var(modelo.A, initialize= regra)
```

Solicitando que as variáveis sejam impressas obtemos o seguinte resultado:

```
x : Size=3, Index=A
    Key : Lower : Value : Upper : Fixed : Stale : Domain
      1 :  None :  3.14 :  None : False : False :  Reals
      2 :  None :  3.14 :  None : False : False :  Reals
      3 :  None :  3.14 :  None : False : False :  Reals
y : Size=3, Index=A
    Key : Lower : Value : Upper : Fixed : Stale : Domain
      1 :  None :   1.5 :  None : False : False :  Reals
      2 :  None :   4.5 :  None : False : False :  Reals
      3 :  None :   5.5 :  None : False : False :  Reals
z : Size=3, Index=A
    Key : Lower : Value : Upper : Fixed : Stale : Domain
      1 :  None :   1.5 :  None : False : False :  Reals
      2 :  None :   2.5 :  None : False : False :  Reals
      3 :  None :   3.5 :  None : False : False :  Reals
```

4.3 CONJUNTOS

Conjuntos são componentes que servem principalmente para indexação de variáveis, parâmetros ou restrições e existem duas formas de criar conjuntos em Pyomo. Primeira forma consiste da função Set e a segunda é dada pela função RangeSet. A função RangeSet cria sequências simples conforme os argumentos são inseridos. No caso em que for inserido somente um argumento, este representará o valor final da sequência que inicia em 1 e tem passo 1.

```
import pyomo.environ as pyEnv
#CRIA MODELO PYOMO CONCRETO
modelo = pyEnv.ConcreteModel()
```

```
# CRIA CONJUNTO C
modelo.C= pyEnv.RangeSet(10)
modelo.C.pprint()
```

A função $pprint()$ é utilizada para imprimir o conjunto C onde obtemos a seguinte resposta:

```
>>> C : Dimen=1, Size=10, Bounds=(1, 10)
    Key  : Finite : Members
    None :   True :  [1:10]
```

O resposta do python nos dá informações sobre o conjunto onde $Bounds$ são os limites do conjunto, se ele é finito ou não, se há uma chave associado ao conjunto, e quais os membros desse conjunto. Note ainda que é usado um modelo concreto, por questão de praticidade, pois para mostrar o funcionamento das funções com modelos abstratos é necessário declarar os conjuntos, o que pode ser feito diretamente em modelos concretos.

Se forem inseridos dois parâmetros na funçao $RangeSet$ eles representarão os limites superior e inferior do intervalo com passo padrão igual a 1.

```
import pyomo.environ as pyEnv

#CRIA MODELO PYOMO CONCRETO
modelo = pyEnv.ConcreteModel()
# CRIA CONJUNTOS
modelo.C= pyEnv.RangeSet(1, 15)
modelo.C.pprint()
```

```
>>> C : Dimen=1, Size=15, Bounds=(1, 15)
        Key  : Finite : Members
        None :   True :  [1:15]
```

Se três argumentos forem inseridos o terceiro argumento será o passo.

```
import pyomo.environ as pyEnv

#CRIA MODELO PYOMO CONCRETO
modelo = pyEnv.ConcreteModel()
# CRIA CONJUNTOS
modelo.C= pyEnv.RangeSet(1, 15, 0.5)
modelo.C.pprint()
```

```
>>> C : Dimen=1, Size=29, Bounds=(1, 15)
            Key  : Finite : Members
            None :   True : [1], [1.5], [2.0], [2.5], [3.0],
            [3.5], [4.0], [4.5], [5.0], [5.5], [6.0], [6.5],
            [7.0], [7.5], [8.0], [8.5], [9.0], [9.5], [10.0],
            [10.5], [11.0], [11.5], [12.0], [12.5], [13.0],
            [13.5], [14.0], [14.5], [15.0]
```

Note que a função $RangeSet$ é útil para descrever conjuntos como sequências de números com limites e um passo bem determinado. Observe que tal característica não está presente nas especificações da função Set. Por outro lado, a função Set tem vários parâmetros dentre eles $within$ o qual especifica domínios que podem ser usados, por exemplo:

- Any - Qualquer valor
- Reals - reais
- PositiveIntegers - ReaisPositivos
- NonPositiveIntegers - Reais não positivos

```
import pyomo.environ as pyEnv

#CRIA MODELO PYOMO CONCRETO
modelo = pyEnv.ConcreteModel()
# CRIA CONJUNTOS
modelo.C= pyEnv.Set(within=pyEnv.NonNegativeReals)
modelo.C.pprint()
```

```
>>> C : Size=1, Index=None, Ordered=Insertion
    Key  : Dimen : Domain           : Size : Members
    None :     1 : NonNegativeReals :    0 :      {}
```

Outra opção importante é o argumento $initialize$, tal argumento recebe iteráveis e é usado para definir valores iniciais do conjunto. Este comando pode ser usado juntamente com uma função e ainda ser indexado por outros conjuntos. Quando tal procedimento ocorre temos que uma matriz de conjuntos é gerada.

```
import pyomo.environ as pyEnv
```

```
#CRIA MODELO PYOMO CONCRETO
modelo = pyEnv.ConcreteModel()
# CRIA CONJUNTOS
modelo.C= pyEnv.Set(initialize= [1, 2, 3])
modelo.C.pprint()

modelo.D = pyEnv.Set(initialize= ['amarelo', 'verde'])
modelo.D.pprint()

def indice(modelo):
    return (2*i+1 for i in modelo.C)
modelo.X = pyEnv.Set(initialize=indice)
modelo.X.pprint()

# CRIA MATRIZ DE CONJUNTOS
modelo.A = pyEnv.Set(modelo.C, dimen=3, initialize=indice)
modelo.A[1].pprint

    >>> C : Size=1, Index=None, Ordered=Insertion
            Key  : Dimen : Domain : Size : Members
            None :     1 :    Any :    3 : {1, 2, 3}
        D : Size=1, Index=None, Ordered=Insertion
            Key  : Dimen : Domain : Size : Members
            None :     1 :    Any :    2 : {'amarelo', 'verde'}
        X : Size=1, Index=None, Ordered=Insertion
            Key  : Dimen : Domain : Size : Members
            None :     1 :    Any :    3 : {3, 5, 7}
        A : Size=3, Index=C, Ordered=Insertion
            Key : Dimen : Domain : Size : Members
              1 :     3 :    Any :    1 : {(3, 5, 7),}
              2 :     3 :    Any :    1 : {(3, 5, 7),}
              3 :     3 :    Any :    1 : {(3, 5, 7),}
        {Member of A} : Size=3, Index=C, Ordered=Insertion
            Key : Dimen : Domain : Size : Members
              1 :     3 :    Any :    1 : {(3, 5, 7),}
```

No código acima, mostra-se que um conjunto pode conter valores numéricos e strings e o argumento $initialize$ pode ser usado para chamar uma função que define elementos do conjunto. Para tal operação, usamos outro conjunto previamente definido. Além disso, podemos ainda fazer operações matemáticas com conjuntos.

Observe que no exemplo a seguir é realizada a união, intersecção, diferença e "ou exclusivo".

```
import pyomo.environ as pyEnv

modelo = pyEnv.ConcreteModel()

modelo.A = pyEnv.Set(initialize=[1,2,3,4,5,6])
modelo.B = pyEnv.Set(initialize=[1,3,5,7,9])
modelo.G = modelo.A | modelo.B # set union
modelo.H = modelo.B & modelo.A # set intersection
modelo.I = modelo.A - modelo.B # set difference
modelo.J = modelo.A^modelo.B # set exclusive-or
modelo.K = modelo.A*modelo.B

modelo.G.pprint()
modelo.H.pprint()
modelo.I.pprint()
modelo.J.pprint()
modelo.K.pprint()
```

```
>>> G : Size=1, Index=None, Ordered=True
        Key  : Dimen : Domain : Size : Members
        None :     1 :  A | B :    8 : {1, 2, 3, 4, 5, 6, 7, 9}
    H : Size=1, Index=None, Ordered=True
        Key  : Dimen : Domain : Size : Members
        None :     1 :  B & A :    3 : {1, 3, 5}
    I : Size=1, Index=None, Ordered=True
        Key  : Dimen : Domain : Size : Members
        None :     1 :  A - B :    3 : {2, 4, 6}
    J : Size=1, Index=None, Ordered=True
        Key  : Dimen : Domain : Size : Members
        None :     1 :  A ^ B :    5 : {2, 4, 6, 7, 9}
    K : Size=1, Index=None, Ordered=True
        Key  : Dimen : Domain : Size : Members
        None :     2 :    A*B :   30 :
        {(1, 1), (1, 3), (1, 5), (1, 7), (1, 9), (2, 1), (2, 3),
        (2, 5), (2, 7), (2, 9), (3, 1), (3, 3), (3, 5), (3, 7),
        (3, 9), (4, 1), (4, 3), (4, 5), (4, 7), (4, 9), (5, 1),
        (5, 3), (5, 5), (5, 7), (5, 9), (6, 1), (6, 3),
```

```
            (6, 5), (6, 7), (6, 9)}
```

4.4 PARÂMETROS

Parâmetro são informações fornecidas para que seja encontrado um valor ótimo para uma dada variável de decisão. São basicamente condições do problema e podemos citar como exemplo constantes a_1, a_2...a_n de um dado polinômio. Parâmetros são declarados na biblioteca Pyomo por meio da função $Param$, que, sem argumentos apenas declara o objeto parâmetro, bem como os outros objetos do modelo. Veja o exemplo a seguir

```
import pyomo.environ as pyEnv

modelo = pyEnv.ConcreteModel()

modelo.a = pyEnv.Param()
```

A função $param$ recebe alguns argumentos como por exemplo $within$ que nesse caso é o conjunto delimitado de valores que podem ser recebido pelo parâmetro e $initialize$. Parâmetros também podem ser indexados por conjuntos ou vetores, e podem ser compostos por qualquer tipo de variável, string, floating, point, etc. O domínio padrão dos parâmetros é Any, isto é, o parâmetro pode ser de qualquer tipo.

- **within**: retorna o conjunto ao qual os parâmetros pertencem;
- **initialize**: retorna os dados para inicializar os valores dos parâmetros;

No código abaixo temos um exemplo em que a indexação de um grupo de parâmetros é feita utilizando um vetor N, onde, para cada índice de N temos um parâmetro.

```
import pyomo.environ as pyEnv

modelo = pyEnv.ConcreteModel()

N = [1,2,3]
def regra1(modelo,j):
    return (3*j)

modelo.a = pyEnv.Param(N, rule= regra1)
modelo.a.pprint()

    >>> a : Size=3, Index=a_index, Domain=Any, Default=None,
```

```
    Mutable=False
            Key : Value
              1 :     3
              2 :     6
              3 :     9
```

Podemos ainda utilizar tuplas e variáveis para criação de parâmetros. Suponha, por exemplo, que para cada par Fornecedor-Produto deseja-se criar um parâmetro cujo valor depende de duas variáveis. A primeira variável relacionada ao produto e outra relacionada ao fornecedor. Dadas informações iniciais sobre o fornecedor e o seu produto, é possível criar os parâmetros como no exemplo a seguir e em acordo com (CARVALHO; NETO, 2020; HART et al., 2013; PYOMO, 2022).

```
import pyomo.environ as pyEnv


modelo = pyEnv.ConcreteModel()

Fornecedores= {'Fornecedor1':10, 'Fornecedor2':12, 'Fornecedor3':15}
Produtos= {'maçã':3, 'limão':2, 'uva':7, 'morango':9}

def regra1(modelo,i):
    return Fornecedores[i]/2
modelo.A = pyEnv.Var(Fornecedores, initialize= regra1)

def regra2(modelo,j):
    return Produtos[j]
modelo.B = pyEnv.Var(Produtos, initialize= regra2)

def regra3(modelo,a,b):
    return modelo.A[a]*modelo.B[b]

modelo.C= pyEnv.Param(Fornecedores, Produtos, initialize= regra3)

modelo.C.pprint()
```

```
    >>> C : Size=12, Index=C_index, Domain=Any, Default=None, Mutable=False
              Key                : Value
      ('Fornecedor1', 'limão') :  10.0
       ('Fornecedor1', 'maçã') :  15.0
```

```
('Fornecedor1', 'morango') :  45.0
    ('Fornecedor1', 'uva') :  35.0
  ('Fornecedor2', 'limão') :  12.0
   ('Fornecedor2', 'maçã') :  18.0
('Fornecedor2', 'morango') :  54.0
    ('Fornecedor2', 'uva') :  42.0
  ('Fornecedor3', 'limão') :  15.0
   ('Fornecedor3', 'maçã') :  22.5
('Fornecedor3', 'morango') :  67.5
    ('Fornecedor3', 'uva') :  52.5
```

Consequentemente, o argumento $initialize$ também aceita a introdução de funções.

4.5 VARIÁVEIS

As variáveis tem o objetivo de receber valores, variáveis podem tem somente um valor ou um conjunto de valores conforme comentado anteriormente. Podem ser definidas manualmente, por dicionários, funções e até mesmo por um solucionador. São declaradas com a função Var que possui alguns parâmetros, por exemplo:

- **bounds** - limite superior e inferior do intervalo de valores que a variável pode assumir
- **initialize** - uma função que dá um valor inicial para uma variável
- **domain** ou **within** - conjunto de valores que a variável pode assumir

Uma variável é declarada dentro de um modelo, conforme demonstrado a seguir, podendo conter ou não argumentos

```
import pyomo.environ as pyEnv

modelo = pyEnv.ConcreteModel()

modelo.A = pyEnv.Var(initialize=1.5)
modelo.x = pyEnv.Var(within= pyEnv.Reals)
modelo.y = pyEnv.Var(domain= pyEnv.NonNegativeIntegers)
```

No exemplo acima a variável A recebe o valor 1.5, a variável x está dentro do conjunto dos números reais e y está no domínio dos números inteiros não negativos. Os argumentos $within$ e $domain$ também podem ser usados na definição de um conjunto

onde uma variável é nomeada através de uma função e neste caso temos que indexar a variável. Veja o exemplo a seguir

```
import pyomo.environ as pyEnv

modelo = pyEnv.ConcreteModel()

modelo.A = pyEnv.Set(initialize=[1,2,3])

def s_domain(model, i):
    return pyEnv.IntegerInterval(bounds=(i,i+1))
modelo.s = pyEnv.Var(modelo.A, domain=s_domain)
modelo.s.pprint()
```

```
    >>> s : Size=3, Index=A
    Key : Lower : Value : Upper : Fixed : Stale : Domain
    1 :     1 :  None :     2 : False :  True : 'IntegerInterval(1, 2)'
    2 :     2 :  None :     3 : False :  True : 'IntegerInterval(2, 3)'
    3 :     3 :  None :     4 : False :  True : 'IntegerInterval(3, 4)'
```

É importante observar que, embora o Pyomo aceite uma representação geral para determinação do domínio das variáveis, nem todo solver aceita essa representação, portanto essa representação somente será possível em modelos onde o solver selecionado suporte tal tipo de representação. Outra possibilidade de indexação ocorre para determinação de limites com o argumento $bounds$

```
import pyomo.environ as pyEnv

modelo = pyEnv.ConcreteModel()

modelo.A = pyEnv.Set(initialize=[1,2,3])
modelo.a = pyEnv.Var(bounds=(0.0,None))
inferior = {1:3.5, 2:5.5, 3:7.5}
superior = {1:1.5, 2:3.5, 3:6.5}

def limites(model, i):
    return (inferior[i], superior[i])
modelo.b = pyEnv.Var(modelo.A, bounds=limites)
modelo.b.pprint()
```

```
>>> b : Size=3, Index=A
        Key : Lower : Value : Upper : Fixed : Stale : Domain
          1 :   3.5 :  None :   1.5 : False :  True :  Reals
          2 :   5.5 :  None :   3.5 : False :  True :  Reals
          3 :   7.5 :  None :   6.5 : False :  True :  Reals
```

Em relação ao trabalho usando variáveis, é comum a solicitação de informações a respeito de uma variável com o objetivo de determinar valores para a realização de testes rápidos. Para imprimir somente o segundo valor da variável, por exemplo b do exemplo acima, podemos usar a seguinte expressão

```
modelo.b[2].pprint()
```

```
>>> b : Size=3, Index=A
        Key : Lower : Value : Upper : Fixed : Stale : Domain
          2 :   5.5 :  None :   3.5 : False :  True :  Reals
```

Por fim, o valor atual de uma variável pode ser obtido por meio da função $value$.

4.6 FUNÇÃO OBJETIVO

Do ponto de vista de um problema de otimização, a função objetivo é o objeto principal de interesse, e trata-se da função que deve ser maximizada ou minimizada. Em alguns casos existem mais de uma função objetivo. Um objetivo na biblioteca Pyomo é declarado como segue:

```
modelo.Obj= pyEnv.Objective()
```

A função objetivo também possui alguns argumentos que são caracterizados como:

- **expr** - fornece a expressão que define a função objetivo;
- **rule** - fornece uma regra dada por uma função que será chamada;
- **sense** - determina se o objetivo deve ser minimizado ou maximizado

Com objetivo de ilustrar, observe o programa abaixo, o qual segue diretrizes estabelecidas em (HART et al., 2013; CARVALHO; NETO, 2020; PYOMO, 2022)

```
import pyomo.environ as pyEnv

modelo = pyEnv.ConcreteModel()
```

```
A = ['r', 's', 't']

modelo.x = pyEnv.Var(A)
modelo.y = pyEnv.Var(within= pyEnv.NonNegativeReals)
modelo.Obj= pyEnv.Objective(expr= 2*modelo.y+1)
modelo.Obj.pprint()

def regra(modelo, i):
    return modelo.x[i]**2
modelo.d = pyEnv.Objective(A, rule= regra)
modelo.d.pprint()

    >>> Obj : Size=1, Index=None, Active=True
            Key  : Active : Sense    : Expression
            None :   True : minimize : 2*y + 1
        d : Size=3, Index=d_index, Active=True
            Key : Active : Sense    : Expression
               r :   True : minimize :    x[r]**2
               s :   True : minimize :    x[s]**2
               t :   True : minimize :    x[t]**2
```

Quando um objetivo é declarado como um objeto indexado, o Pyomo utiliza a função atribuída ao argumento $rule$ para gerar cada um dos objetivos, fornecendo a essa função os elementos do conjunto de índice. Se mais de um conjunto de índices for fornecido a biblioteca Pyomo itera sobre o produto vetorial dos conjuntos, isto é, para cada combinação possível de índices. Em alguns casos pode ser necessário ignorar algum índice, para isso é utilizada a função $Objective.Skip$. Podemos ilustrar isso por meio do exemplo abaixo.

```
import pyomo.environ as pyEnv


modelo = pyEnv.ConcreteModel()

A= ['R','S', 'T']
modelo.x = pyEnv.Var(A, initialize=1.0)

def regra(model, i):
    if i == 'R':
            return pyEnv.Objective.Skip
```

```
    return (modelo.x[i]**(2))
modelo.e = pyEnv.Objective(A, rule= regra)
modelo.e.pprint()
```

```
    >>> e : Size=2, Index=e_index, Active=True
            Key : Active : Sense    : Expression
              S :   True : minimize :    x[S]**2
              T :   True : minimize :    x[T]**2
```

4.7 RESTRIÇÕES

As restrições são geralmente dadas por funções, relativas a modelagem do problema de estudo, dentro das quais existem regras usando igualdades ou desigualdades. Existem diversas formas de expressar as restrições utilizando a função $bound$, ou delimitando um intervalo a partir da expressão: $lb \leq expr \leq ub$ onde lb é o limite inferior ub o limite superior e $expr$ são os valores que podem ser assumidos. É muito frequente a utilização de indexação para restrições em contrário dos objetivos que raramente são indexados. Para declarar uma restrição em geral é necessário existir uma variável ou algum conjunto declarado anteriormente. Entretanto, é possível tanto para variáveis quanto conjuntos somente declarar a restrição a posteriori fazer sua atribuição. Veja o exemplo

```
import pyomo.environ as pyEnv

modelo = pyEnv.ConcreteModel()

modelo.x = pyEnv.Var([1,2], initialize=1.0)
modelo.vazio= pyEnv.Constraint()
modelo.res = pyEnv.Constraint(expr=modelo.x[2]-modelo.x[1] <= 10)
modelo.vazio.pprint()
modelo.res.pprint()
```

```
    >>> vazio : Size=0, Index=None, Active=True
            Key : Lower : Body : Upper : Active
        res : Size=1, Index=None, Active=True
            Key  : Lower : Body        : Upper : Active
            None :  -Inf : x[2] - x[1] :  10.0 :   True
```

As restrições também podem ser indexadas, fazendo referência a variáveis e parâmetros, os quais podem ser utilizados dentro das respectivas restrições. Uma ilustração disto é apresentada no seguinte exemplo:

```
import pyomo.environ as pyEnv

modelo = pyEnv.ConcreteModel()

N = [1,2,3]
def regra1(modelo,j):
    return (3*j)

modelo.a = pyEnv.Param(N, rule= regra1)

b = {1:1, 2:7, 3:10}
modelo.y = pyEnv.Var(N, within= pyEnv.NonNegativeReals, initialize=0.0)

def regra3(modelo, i):
    return modelo.a[i] * modelo.y[i] >= b[i]
modelo.restricao = pyEnv.Constraint(N, rule= regra3)
modelo.restricao.pprint()

    >>> restricao : Size=3, Index=restricao_index, Active=True
            Key : Lower : Body    : Upper : Active
              1 :   1.0 : 3*y[1] :  +Inf :   True
              2 :   7.0 : 6*y[2] :  +Inf :   True
              3 :  10.0 : 9*y[3] :  +Inf :   True
```

As restrições indexadas são atribuídas da mesma forma que os objetivos, se quisermos que uma restrição associada a um determinado índice seja nula podemos utilizar a função $Constraint.Skip$. O próximo exemplo ilustra isto.

```
import pyomo.environ as pyEnv


modelo = pyEnv.ConcreteModel()

N = [1,2,3]
def regra1(modelo,j):
    return (3*j)

modelo.a = pyEnv.Param(N, rule= regra1)

modelo.a.pprint()
```

```
b = {1:1, 2:7, 3:10}
modelo.y = pyEnv.Var(N, within= pyEnv.NonNegativeReals, initialize=0.0)

def regra3(modelo, i):
    if i==2:
        return pyEnv.Constraint.Skip
    return modelo.a[i] * modelo.y[i] >= b[i]
modelo.restricao = pyEnv.Constraint(N, rule= regra3)
modelo.restricao.pprint()

    >>> restricao : Size=2, Index=restricao_index, Active=True
            Key : Lower : Body    : Upper : Active
              1 :   1.0 : 3*y[1] :  +Inf :   True
              3 :  10.0 : 9*y[3] :  +Inf :   True
```

Existem três tipos de restrições que a biblioteca Pyomo aceita. Tais formas de restrições podem ser usadas diretamente no argumento $expr$ ou ainda com uma determinada função para restrições indexadas.

- **Igualdade** - $expr_1 == expr_2$;
- **desigualdade** - $expr_1 \leq expr_2$ ou $expr_1 \geq expr_2$;

Na forma de intervalo, as expressões $expr_1$ e $expr_2$ não são constantes enquanto a e b o são. Depois de declaradas as restrições a biblioteca Pyomo busca pelo intervalo onde tal restrição possui validade numa tupla com atributos (lower, body, upper). Neste caso, as expressões não constantes são associadas ao atributo $body$, caso tenhamos uma restrição de igualdade o atributo $equality$ é considerado $True$ e os atributos $lower$ e $upper$ permanecem inalterados. Finalmente, é possível usar as funções $lslack$ e $uslack$ para encontrar as folgas inferior e superior respectivamente conforme o exemplo a seguir.

```
import pyomo.environ as pyEnv


modelo = pyEnv.ConcreteModel()

modelo.x = pyEnv.Var(initialize=1.0)
modelo.y = pyEnv.Var(initialize=1.0)
modelo.c1 = pyEnv.Constraint(expr= modelo.y - modelo.x <= 7.5)
```

```
modelo.c2 = pyEnv.Constraint(expr=-2.5 <= modelo.y - modelo.x)

print(pyEnv.value(modelo.c1.body)) # 0.0
print(modelo.c1.lslack())
print(modelo.c1.uslack())
print(modelo.c2.lslack())
print(modelo.c2.uslack())
```

```
>>> 0.0
    inf
    7.5
    2.5
    inf
```

4.8 RESOLVENDO MODELOS PYOMO E SOLVERS

Para e resolução de um modelo concreto na biblioteca Pyomo, é preciso selecionar um determinado solver compatível e que esteja no caminho do Python. Um modelo concreto pode ser resolvido com as seguintes linhas de código

```
import pyomo.environ as pyEnv

modelo = pyEnv.ConcreteModel()

modelo.opt = pyEnv.SolverFactory('glpk')
modelo.opt.solve(modelo)
```

Observe que, após selecionado qual solver será utilizado para resolver o modelo é necessário um comando para sua resolução. O Argumento da função $SolverFactory$ é o solver que será utilizado para resolver o modelo. No caso de um modelo abstrato, primeiro é necessário criar uma instância concreta do modelo e depois chamar a função para resolução do problema. Tal procedimento é descrito no exemplo abaixo.

```
import pyomo.environ as pyEnv

modelo= pyEnv.AbstractModel()

instance = modelo.create_instance()
modelo.opt = pyEnv.SolverFactory('glpk')
resultado = modelo.opt.solve(instance)
print(resultado)
```

No exemplo apresentado, a solução está atribuída ao comando $resultado$ e é impressa com o comando $print$ do Python. O Pyomo suporta uma ampla variedade de solvers como BARON, CBC, IBM CPLEX, GLPK e Gurobi. Alguns comerciais e outros livres. Para estes solvers ele possui interfaces especializadas, mas em geral suporta qualquer solver que leia arquivos AMPL ".nl", escreva ".sol"e tenha a capacidade de gerar modelos no formato GAMS que recuperem os resultados. Neste Trabalho serão usados os solvers IBM ILOG CPLEX de capacidade limitada e os dois livres CBC e GLPK.

5 APLICAÇÃO E COMPARAÇÕES

Neste capítulo serão apresentados resultados da aplicação dos métodos de solução ao problema do Caixeiro Viajante. As soluções por meio do Pyomo foram obtidas usando três solvers, o CBC, o CEPLEX Optimization Studio V22.1.0 da IBM em sua versão livre gratuita disponibilizada no site da IBM e também o solver GLPK MP/MIP versão 4.65. Afim de poder comparar a performance do Pyomo, os testes também foram rodados em uma rotina na linguagem de programação MatLab da MathWorks que pode ser encontrada no link: https://www.mathworks.com/matlabcentral/fileexchange/64654-travelling-salesman-problem o qual foi disponibilizado por (NARAYANAN, 2022) e utiliza várias ferramentas próprias do MatLab.

No caso do Pyomo, os algoritmos foram utilizados no mesmo computador e nas mesmas condições de trabalho para que não houvesse alguma influência que prejudicasse qualquer dos métodos. O computador utilizado para rodar os solvers via Pyomo possui processador i7 de 5ªgeração com turbo Boost com 3.0 GHz de processamento e 8GB de memória RAM. No caso do MatLab, por tratar-se de uma versão de 2017 do software, para garantir uma melhor performance na resolução das instâncias foi usado um processador i7 de 8ªgeração com 4.10 GHz de processamento e também 8GB de memória RAM.

5.1 TESTE DOS SOLVERS USANDO INSTÂNCIAS BEM COMPORTADAS NO PYOMO

Foram utilizadas oito instâncias do caixeiro viajante, com os três solvers mencionados acima, também foram comparados com um solver implementado em Matlab e disponível na Mathworks. Os parâmetros de comparação utilizados são a eficiência e eficácia dos métodos, isto é, a eficácia de produzir um resultado mais próximo da solução exata e a eficiência de fazê-lo no menor tempo possível. A Tabela 5 mostra as instâncias do problema do caixeiro viajante que foram testadas. Tais instâncias são distintas em relação ao tamanho do problema e número de condição da matriz de

distâncias.

Para tais instâncias apresentadas na Tabela 5 , foram então testados os 3 Solvers e montadas tabelas com os resultados da instância resolvida por cada um deles. Todos os solvers encontraram o mesmo tour mínimo, porém o tempo de execução foi diferente. Para problemas menores a diferença se torna mínima, porém para alguns problemas maiores há uma diferença de segundos como pode ser observada. Vamos dar uma breve descrição de cada solver.

Tabela 5: Instâncias do PCV bem comportadas

Instância	Número de Cidades
TSP1	4
TSP2	21
TSP3	8
TSP4	13
TSP5	4
TSP6	8
TSP7	6
TSP8	5

Solver CBC: O COIN-OR Branch-and-Cut ou como é conhecido CBC é um programa de código inteiro misto (MIP) e também trabalha com programação linear, é um programa de código aberto escrito em C++. O objetivo é ser uma biblioteca que pode ser chamada para criar solucionadores branch-and-cut personaizados. O CBC possibilita a escolha entre o algoritmo primal ou dual Simplex pela programação linear, o solucionador em MIP pode usar ambos algoritmos. Na Tabela 6 vemos os resultados deste solver para as instâncias de interesse dadas na Tabela 5.

Tabela 6: Resultados do solver CBC para as instâncias do PCV dadas na Tabela 5.

Instância	SOLVER	Tour mínimo	Tempo (s)
TSP1	CBC	64	0,03
TSP2	CBC	198	3,74
TSP3	CBC	55	0,12
TSP4	CBC	7293	9,25
TSP5	CBC	80	0,01
TSP6	CBC	88	0,14
TSP7	CBC	76	0,05
TSP8	CBC	31	0,04

Solver IBM Cplex: O IBM ILOG CPLEX é um solver voltado para programação inteira (otimização linear) isto é, resolve modelos de programação inteira. O solver atualmente opera os métodos Simplex e Ponto Interior, fornece ainda bibliotecas em C, C++, Java e Python e .NET para a resolução de problemas de programação inteira. Mais especificamente ele resolve problemas de otimização restritos onde a função

objetivo é expressa por uma função linear ou uma função quadrática convexa, e as variáveis podem ser contínuas ou discretas. O programa resolve problemas interativamente ou a partir de dados pré-definidos e entrega a solução interativamente ou em formato txt. A Concert Tecnology é um conjunto de bibliotecas que oferece uma API que permite a incorporação dos recursos nas lingugens C, C++, Python, Java e .NET. A CPLEX Callable Library é uma biblioteca C que permite ao programador incorporar o CPLEX em aplicativos como, Visual Basic, Fortran ou qualquer outra linguagem que pode chamar funções C. Na Tabela 7 vemos os resultados deste solver para as instâncias de interesse.

Tabela 7: Resultados do solver CPLEX para as instâncias do PCV dadas na Tabela 5.

Instância	SOLVER	Tour mínimo	Tempo (s)
TSP1	Cplex	64	0,09
TSP2	Cplex	198	0,20
TSP3	Cplex	55	0,13
TSP4	Cplex	7293	1,44
TSP5	Cplex	80	0,10
TSP6	Cplex	88	0,11
TSP7	Cplex	76	0,10
TSP8	Cplex	31	0,14

Solver GLPK: O GNU Linear Programing Kit ou como é mais conhecido GLPK é um solver de código aberto criado para resolver problemas de programação linear, inteira, mitos e outros relacionados o solver contém os seguintes componentes:1) Métodos primal e dual Simplex; 2) Método de ponto interior primal e dual; 3) Método de ramificação e corte; 4) Tradutor para GNU MathProg; 5) Interface API; 6)Solucionador de LP/MIP autônomo. Na Tabela 8 vemos os resultados deste solver para as instâncias de interesse.

Tabela 8: Resultados do solver GLPK para as instâncias do PCV dadas na Tabela 5.

Instância	SOLVER	Tour mínimo	Tempo (s)
TSP1	GLPK	64	0,14
TSP2	GLPK	198	1,05
TSP3	GLPK	55	0,10
TSP4	GLPK	7293	2,56
TSP5	GLPK	80	0,07
TSP6	GLPK	88	0,09
TSP7	GLPK	76	0,10
TSP8	GLPK	31	0,09

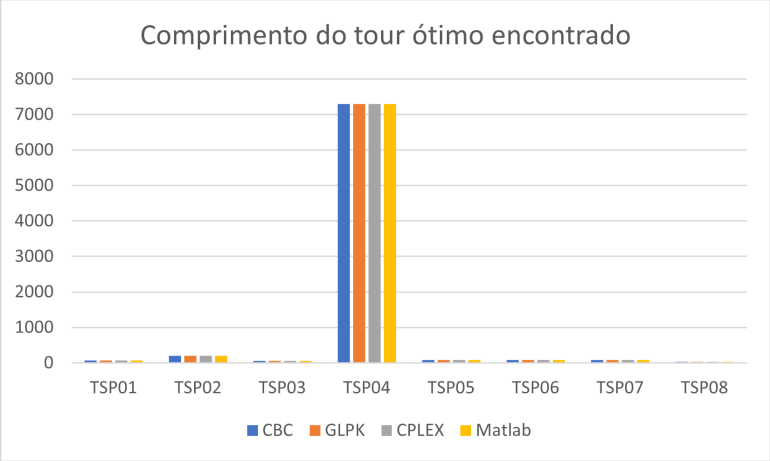
As instâncias bem comportadas também foram testadas em um solver do MatLab da Mathworks para fins de comparação. O solver é baseado em Programação Linear,

mais especificamente a função **intlinprog** que é própria do Software. Os resultados obtidos para cada caso seguem na tabela 9. Nesta primeira etapa os resultados puderam ser calibrados e comparados entre os solvers, na próxima etapa instâncias não tão bem comportadas serão analisadas. Aqui o termo não tão comportadas refere-se ao fato dos solvers terem dificuldades de atingir a rota com o tour mínimo exato.

Tabela 9: Resultados do solver do MatLab para as instâncias do PCV dadas.

Instância	Tour mínimo	Tempo (s)
TSP1	64	0.83
TSP2	198	0.95
TSP3	55	0.67
TSP4	7293	1.17
TSP5	80	0.08
TSP6	88	0.71
TSP7	76	0.40
TSP8	31	0.03

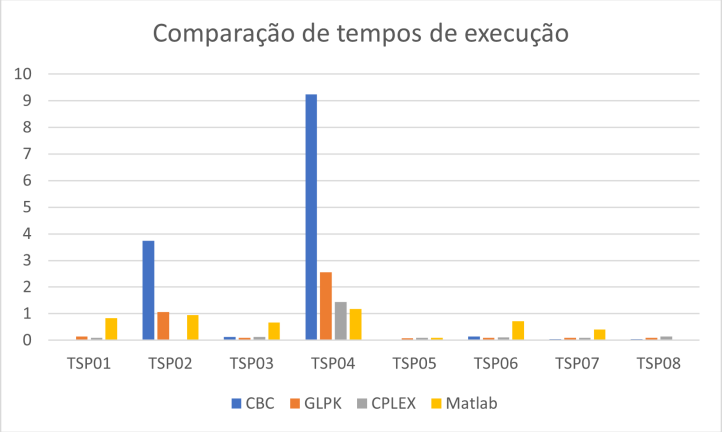
Figura 6: A figura mostra a comparação entre os métodos nas instâncias bem comportadas. Com relação ao caminho ótimo todos obtiveram os mesmos resultados.



Fonte: O autor

Plotando as informações de distância em um gráfico, é possível perceber que não houve diferença na qualidade das soluções geradas. Na Figura 6 é ilustrada as comparações das rotas ótimas obtidas para os exemplos da Tabela 5. Porém, o mesmo não pode ser dito quanto aos tempos de execução, observe na Figura 8 que, para as menores instâncias, o tempo de execução entre os solvers é comparável, porém para instâncias maiores as diferenças entre os solvers é maior. Interessante ressaltar ainda que os solvers operam com técnicas diferentes como citado anteriormente.

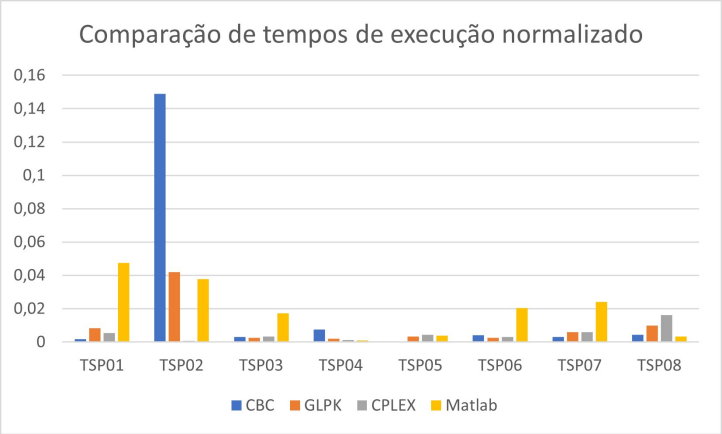
Figura 7: Comparação do tempo de execução em segundos de cada um dos solver usados. Fica evidente a baixa performance do CBC para as duas instâncias com mais cidades em relação aos demais.



Fonte: O autor

Fazendo uma normalização nos tempos de execução, com relação a distância média de cada instância, é possível perceber que embora o tempo de execução da instância TSP04 seja maior, quando a distância média da matriz de distâncias do problema é levada em conta, percebe-se que a instância TSP02 foi a instância que demandou mais tempo para execução. Essa análise sugere uma media de tempo com relação a complexidade do problema, que não depende somente do número de cidades a serem visitadas, mas ainda de outros fatores que influenciam no desempenho computacional, sendo um deles o número de condição da matriz de distâncias e a distância média da matriz de distâncias.

Figura 8: Comparação do tempo de execução em segundos de cada um dos solver usados, normalizado pela distância média em cada instância.



Fonte: O autor

5.2 TESTE DOS SOLVERS USANDO INSTÂNCIAS NÃO TÃO BEM COMPORTADAS NO PYOMO

Foram usadas também algumas instâncias encontradas no site http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95 disponibilizada por (HEIDELBERG, 2013) da universidade de Heidelberg. As instâncias estão descritas na Tabela 10 e no site encontra-se também as melhores respostas para estas instâncias representadas na Tabela 10 como tour ótimo. Para estas instâncias estabelecemos um tempo máximo de execução de uma hora e apesar de existir algumas de porte respeitável como ch130 e a280, alguns dos solvers apresentaram dificuldades mesmo em casos reduzidos como a base ulysses16. No site onde a base de dados para as instâncias não tão comportadas foi retirada não específica por quais técnicas as rotas ali apresentadas como ótimas foram obtidas.

Tabela 10: Instâncias do PCV não tão bem comportadas

Instância	Número de Cidades	tour ótimo
bayg29	29	1610
att48	48	10628
berlin52	52	7542
ch130	130	6110
kroA100	100	21282
a280	280	2579
ulysses16	16	6859

No solver CBC os dados testados apresentaram os resultados da Tabela 11 e, apesar de não conseguir resolver a maioria das instâncias, o solver obteve na instância berlin52 um resultado muito próximo ao resultado de referência.

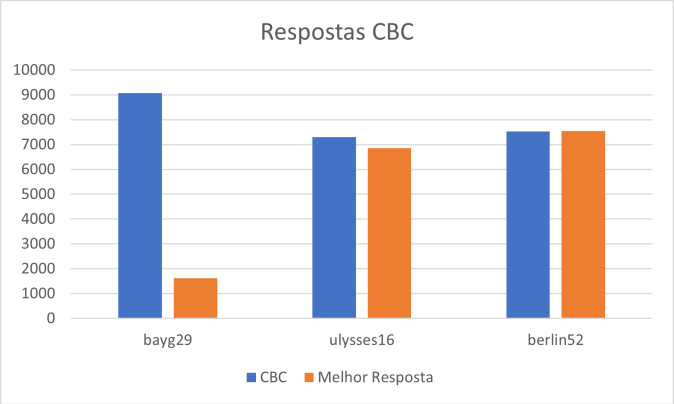
Tabela 11: Resultados de instâncias não tão bem comportadas no CBC

Instância	tempo (s)	tour ótimo
bayg29	112,9	9074,14
att48	t >1 hora	não convergiu
berlin52	392,11	7544
ch130	t >1 hora	não convergiu
kroA100	t >1 hora	não convergiu
a280	t >1 hora	não convergiu
ulysses16	208,52	7300

Na Figura 9, são comparados os resultados do solver CBC com os dados de referência fornecidos pelo site. Percebe-se que para as três instâncias que o solver resolveu foi encontrado um tour com comprimento maior que o melhor tour disponibilizado pelo site.

No solver CPLEX os dados testados apresentaram os resultados da Tabela 12. É preciso ressaltar que a versão utilizada do CPLEX é a versão gratuita. Portanto, esta

Figura 9: Comparação CBC com a melhor resposta



Fonte: O autor

versão do solver não permite resolver problemas de dimensão maior que 30 clientes e isto impossibilitou obter os demais resultados. Porém para a instância ulysses16 o CPLEX não encontrou a resposta que o site disponibiliza como resposta ótima.

Tabela 12: Resultados de instâncias não tão bem comportadas no CPLEX. Os valores não preenchidos correspondem a instâncias cujas dimensões não são cobertas pelo pacote gratuito da IBM.

Instância	tempo (s)	tour ótimo
bayg29	1,95	9074,14
att48	-	-
berlin52	-	-
ch130	-	-
kroA100	-	-
a280	-	-
ulysses16	4,15	7398

Os dados da instância ulysses16 são coordenadas do plano cartesiano dadas em quilômetros, essa informação não consta no conjunto de dados, mas, é deduzida, pelo fato de a melhor resposta estar escrita em milhares no site, o que sugere que os dados estejam na escala de quilômetros. Essa instância é interessante, pois, tem dimensão menor que a instância bem comportada TSP02 e nenhum solver foi capaz de encontrar o tour ótimo disponibilizado no site. A Tabela 13 mostra as coordenadas das cidades no plano cartesiano, com estas coordenadas é criada a matriz de distâncias que é introduzida nos solvers.

O solver CPLEX em comparação com a referência dada no site também apresentou tours ótimos maiores, essa comparação está expressa no gráfico a seguir a partir dos dados da Tabela 10 e da Tabela 12. No solver GLPK os dados testados apresentaram os resultados da Tabela 14. Observa-se que para instâncias de grandes

Tabela 13: coordenaas cartesianas da instância ulysses16

X	Y
38.24	20.42
39.57	26.15
40.56	25.32
36.26	23.12
33.48	10.54
37.56	12.19
38.42	13.11
37.52	20.44
41.23	9.10
41.17	13.05
36.08	-5.21
38.47	15.13
38.15	15.35
37.51	15.17
35.49	14.32
39.36	19.56

dimensões o solver não convergiu para uma resposta em menos de uma hora o qual era o limite de execução pré-estabelecido. Observa-se também, em comparação com o melhor tour encontrado disponibilizado no site, que o solver se aproximou porém não encontrou o tour ótimo disponibilizado.

Tabela 14: Resultados de instâncias não tão bem comportadas no GLPK

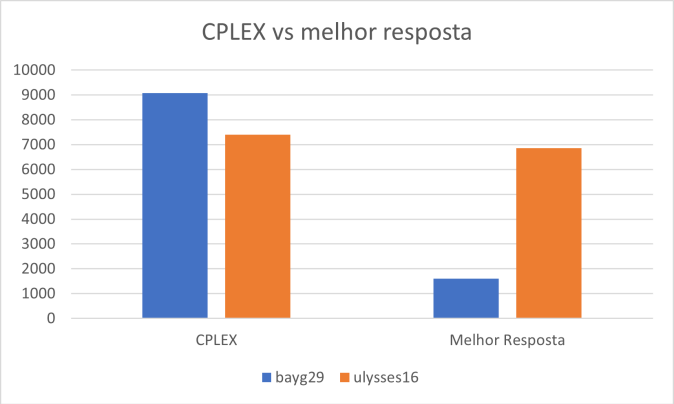
Instância	tempo (s)	tour ótimo
bayg29	150,66	9074,14
att48	t >1 hora	não convergiu
berlin52	t >1 hora	não convergiu
ch130	t >1 hora	não convergiu
kroA100	t >1 hora	não convergiu
a280	t >1 hora	não convergiu
ulysses16	520,96	7300

A comparação pode ser vista na Figuras 10 e 11, onde percebe-se a proximidade das respostas, com as respostas ótimas do site. Note ainda que para a instância berlin52 o solver não convergiu em menos de uma hora de processamento, enquanto o CBC convergiu e encontrou uma resposta melhor do que a resposta apresentada pelo site, como mostrado anteriormente.

Ainda foram testados os dados com o Matlab o qual conseguiu apresentar respostas em períodos curtos de execução das instâncias, gerando resultados interessantes, na Tabela 15 estão expostos esses resultados.

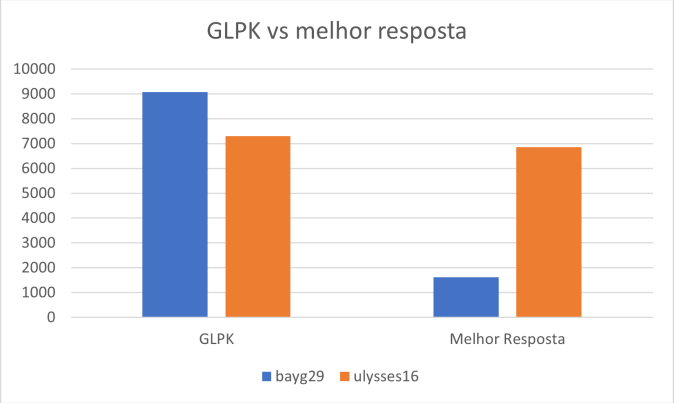
De todas as ferramentas usadas na resolução das instâncias o Matlab foi o único

Figura 10: Comparação Tour ótimo CPLEX vs Melhor resposta



Fonte: O autor

Figura 11: Comparação Tour ótimo GLPK vs Melhor resposta



Fonte: O autor

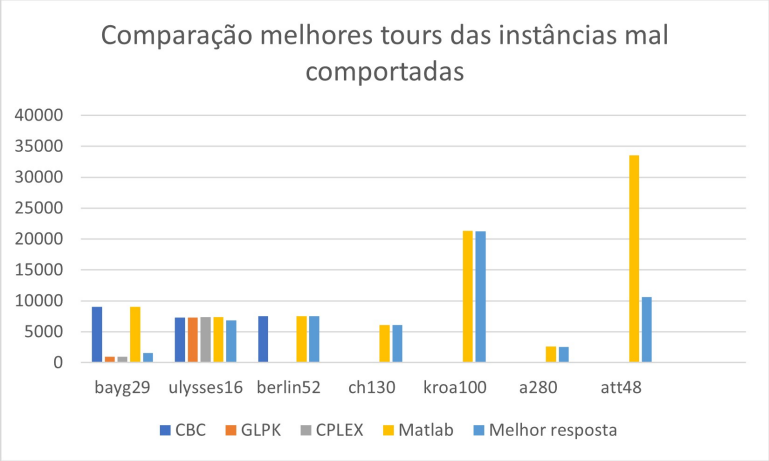
Tabela 15: Resultados de instâncias não tão bem comportadas no MatLab

Instância	tempo (s)	tour ótimo
bayg29	1.46	9074.14
att48	1.59	33551
berlin52	1.29	7544.36
ch130	62.92	6111.22
kroA100	68.6	21337.75
a280	2726.7	2618
ulysses16	0.264	7398.76

que conseguiu resolver todas as instâncias em menos de uma hora de processamento. A Figura 12 ilustra uma comparação geral dos tours ótimos encontrados entre si e entre a resposta do site, utilizada como referência.

Com relação ao tempo para resolução das instâncias na comparação gráfica representada na Figura 13 percebe-se que para a instância berlin52, o CBC conseguiu resolver e o GLPK não conseguiu ultrapassando 1 hora de processamento, como foi

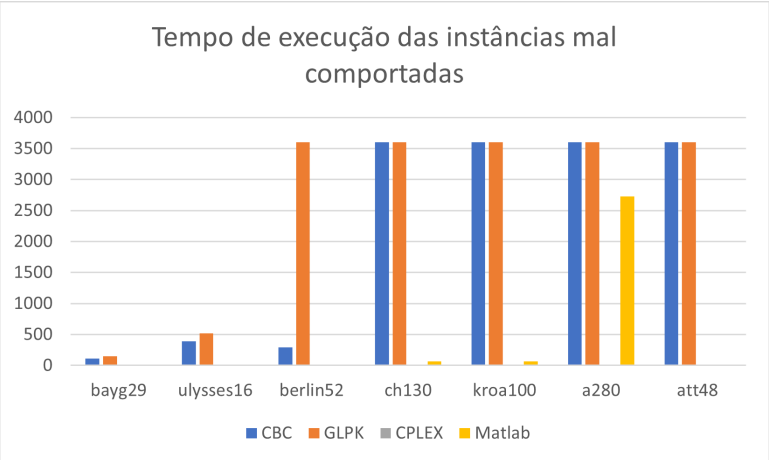
Figura 12: Comparação das resposta ótimas dos solvers com os melhores tours registrados no site onde a base de dados foi coletada.



Fonte: O autor

dito anteriormente. O CPLEX devido a limitação da versão gratuita, não aparece para todas as instâncias, e nas duas instâncias que resolveu não foi superior ao Matlab.

Figura 13: Comparação dos tempos de execução de todos os solvers utilizados. Para os dados byg29, ulysses16 e att48 o tempo de execução do Matlab foi satisfatório e bem abaixo dos demais não aparecendo na escala, mas podendo ser consultados na Tabela 15.

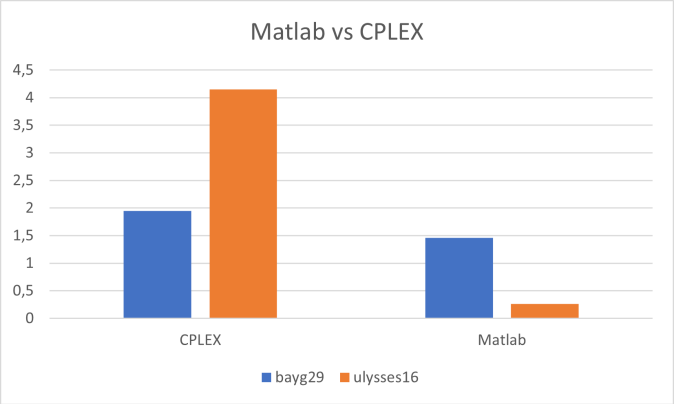


Fonte: O autor

Comparando o CPLEX com o Matlab em tempo de execução, ilustrado na Figura 14, é possível ter uma noção melhor dos tempos de execução para as instâncias que o CPLEX pode resolver, e percebe-se que o Matlab resolveu as instâncias em um tempo menor de execução.

Nos experimentos realizados chama a atenção o desempenho satisfatório do solver

Figura 14: Tempo de Execução CPLEX vc Matlab



Fonte: O autor

do MatLab o qual é também baseado em programação linear mista, em comparação com aos solvers implementados no do python. No próximo capítulo uma análise e discussão dos resultados aqui apresentados é realizada.

6 DISCUSSÃO E CONCLUSÕES

Para as respostas dos solvers em instâncias bem comportadas, nota-se que todas as instâncias foram eficazes em produzir boas respostas, os gráficos mostram que os tours ótimos encontrados foram os mesmos para todos os solvers, o que atesta a confiabilidade dos resultados, a diferença entre os solvers é melhor observada na comparação do tempo de execução. Para evitar flutuações no tempo de execução, os experimentos foram feitos para cada solver separadamente. Com o propósito de estabilizar as flutuações numéricas a memória foi constantemente limpa com o intuito de evitar erros numéricos e, desta forma, tornando o tempo medido confiável. Observa-se portanto que instâncias de maior dimensão são mais complexas fazendo com que o solver demore mais para encontrar a resposta, o que fica claro no gráfico da Figura 8.

Outro fator importante é que a maioria dos solvers levou mais tempo para concluir a instância TSP04 que é menor que a instância TSP02. Analisando as matrizes de distâncias, verifica-se algumas diferenças. Para a instância TSP02 as distâncias entre as cidades não ultrapassam dois dígitos, isto é, estão todas em ordem de dezenas, enquanto a instância TSP04, possui distâncias todas da ordem de centenas e milhar. Os demais testes tem dimensão menor, e possuem distâncias na ordem de dezenas, e nenhuma delas possui alguma distância maior na ordem de dezenas ou maior.

Para as instâncias mal comportadas, a menor é a ulysses16 nenhum solver conseguiu chegar ao valor de tour ótimo apresentado pelo site, de fato, tomando o tour ótimo apresentado pelo site e somando as distâncias a cada vértice, verificou-se que o tour ótimo apresentado como solução possui comprimento igual ao comprimento de tour ótimo que os solvers encontraram o que é um fato curioso sobre a base de dados.

Para as demais instâncias a maioria delas não foi resolvida em menos de 1 hora, pelo CBC e GLPK. Mas é interessante notar que para a instância bayg29 todos os solvers encontraram uma resposta divergente da resposta apresentada pelo site, porém o que mais chama a atenção é o fato de o CBC conseguir resolver uma instância de dimensão maior e apresentar um resultado muito próximo ao apresentado pelo site, resultado corroborado pelo resultado do Matlab, que foi o solver com melhor desem-

penho geral, encontrando respostas muito próximas e até melhores comparado ao site. Note que em semelhança ao que ocorre nas instâncias bem comportadas, uma instância de maior dimensão é resolvida com melhor resultado, e, ainda mais, quando comparada a instância att48, outra instância menor, o CBC falha em encontrar uma resposta em menos de uma hora para tal instância.

Um fato adicional que chama a atenção é que os solvers comerciais MatLab e Cplex conseguiram realizar o tempo execução dentro do limite imposto para instâncias em que foram testados. O uso da biblioteca Pyomo para as instâncias mais mal comportadas foi realizado corretamente uma vez que os testes nas instâncias mais simples foram bem sucedidos. A ferramenta é bastante adequada em um quadro de trabalho onde o uso de vários solvers é requerido para testar soluções em um determinado problema. No entanto, o desempenho em tempo de execução insatisfatório dos solvers livres CBC e GLPK em problemas cujas matrizes de distância possuem dimensões modestas indica que é necessário cuidado com o uso destas duas caixas pretas. Por fim noutou-se ainda o desempenho bastante saisfatório do solver do MatLab mostrando que trata-se de uma ferramenta ainda de grande relevância para o uso em provas de conceito e realização de backtests em experimentos de porte moderado.

REFERÊNCIAS

CARVALHO, C. W. V.; NETO, A. R. P. **Manual de uso da biblioteca Pyomo para Programação Matemática**. Fortaleza: UFC, 2020.

COOK, W. J. **In Pursuit of the Traveling Salesman**: Matematics at the Limits of Computations. Boston: Princenton, 2012.

DANTZIG, G.; FULKERSON, R.; JOHNSON, S. Solution of a Large-Scale Traveling-Salesman Problem. **Journal of the Operations Research Society of America**, USA, v.2, n.4, p.393–410, 1954.

DANTZIG, G.B. "Linear Programming,"in Problems for the Numerical Analysis of the Future, Proceedings of Symposium on Modern Calculating Machinery and Numerical Methods. **National Bureau of Standards**, USA, v.1, n.15, p.18–21, 1951.

GOLDBARG, M. C.; LUNA, H.P. L. **Otimização Combinatória e Programação Linear**. Rio de Janeiro: Editora Campus, 2000.

HART, William E. et al. **Pyomo — Optimization Modeling in Python**. Gewerbestrasse: Springer, 2013.

HEIDELBERG, University of. **Travelling Salesman Problem Data Base**. http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95.

LAND, A. H.; DOIG, A. G. An Automatic method of solving discrete programming problems. **Econometrica**, USA, v.28, n.1, p.497–520, 1960.

MACULAN, N.; FAMPA, M. H. C. **Otimização Linear**. Rio de Janeiro: UFRJ, 2004.

MILLER, C.E.; TUCKER, A.W.; ZEMLIN, R.A. Integer programming formulations and traveling salesman problems. **Journal of the Operations Research Society of America**, USA, v.7, n.4, p.326–329, 1960.

NARAYANAN, Santhanakrishnan. **Travelling Salesman Problem.** https://www.mathworks.com/matlabcentral/fileexchange/64654-travelling-salesman-problem.

PYOMO. **Pyomo Documentation**: Release 6.4.2.dev0. python: USA, 2022.

RIBEIRO, A. A.; KARAS, E. W. **Otimização Contínua**: Aspectos teóricos e computacionais. São Paulo: CENGAGE learning, 2013.

SIPSER, M. **Introdução à teoria da computação**. Boston: Cengage Learning, 2007.

TRUDEAU, R. J. **Introduction Graph Theory**. London: Dover, 1993.

WILHELM, V. E.; KLEINA, M. **Algoritmo Branch and Bound-Notas de aula**. Curitiba: UFPR, 2022.

WILSON, R. J. **Introduction Graph Theory**. London: Prentice Hall, 1998.

WOLSEY, L. A. **Integer Programing**. New York: John Wiley & Sons, 1998.