Desenvolvimento de um Executor Simbólico para o Problema do Caixeiro Viajante utilizando Python e a biblioteca Z3

Venícius Oliveira, Rosialdo Vicente

***Abstract*—This article presents the development of a Symbolic Executor for the Traveling Salesman Problem (TSP) using the Python programming language and the Z3 library. The VCP is a classic combinatorial optimization problem that seeks to find the shortest path that visits a set of cities and returns to the city of origin. The proposed symbolic approach aims to explore efficient solutions through the analysis of constraints and logical reasoning. By implementing symbolic execution algorithms, we demonstrate how Z3 can be used to verify properties and optimize solutions in the context of PCV.**

***Index Terms*—Executor Simbólico, Problema do Caixeiro Viajante, Python, Z3, Otimização Combinatória, Análise de Restrições, Algoritmos, Execução Simbólica.**

I. Introdução

O Problema do Caixeiro Viajante (PCV) é um dos problemas mais conhecidos da teoria da computação e da otimização combinatória. Ele consiste em encontrar o menor caminho que permite a visita a um conjunto de cidades, retornando à cidade de origem, passando por cada cidade uma única vez. Este problema é classificado como NP-difícil, o que significa que encontrar uma solução ótima em tempo polinomial para todas as instâncias é considerado inviável na prática [1].

Diversas abordagens heurísticas e exatas têm sido propostas ao longo dos anos para tratar o PCV. Entre as heurísticas, destacam-se os algoritmos de Lin e Kernighan e o de inserção de vértices, que buscam soluções aproximadas para o problema[2]. No entanto, o avanço de métodos simbólicos, como a execução simbólica, tem proporcionado novos caminhos para a resolução de problemas complexos, como o PCV[3]. A execução simbólica permite explorar múltiplas possibilidades de execução ao tratar variáveis de entrada como símbolos, oferecendo uma forma mais abrangente de testar e verificar programas[3].

Neste contexto, a biblioteca Z3, um solucionador de Satisfatibilidade Modulo Theories (SMT), desenvolvida pela Microsoft Research, destaca-se como uma ferramenta poderosa para a execução simbólica e a verificação de programa[5]. O Z3 permite a combinação de diferentes teorias de primeira ordem, como aritmética e vetores de bits, facilitando a resolução de problemas de otimização. Este trabalho visa desenvolver um Executor Simbólico para o PCV utilizando Python e a biblioteca Z3, com o objetivo de explorar a eficiência dessa abordagem na busca por soluções otimizadas.

Setembro 23, 2024

II. O Problema do Caixeiro Viajante

O Problema do Caixeiro Viajante (PCV), conforme descrito por [1], consiste em encontrar a rota mais curta possível para visitar um conjunto de cidades, partindo de uma cidade de origem, passando por todas as outras exatamente uma vez e retornando à cidade de partida. Apesar de sua formulação simples, o PCV é um problema amplamente reconhecido e estudado na área de otimização combinatória. Ele pertence à classe de problemas NP-difíceis, o que significa que não se conhece um algoritmo que sempre consiga resolver o problema de forma eficiente (em tempo polinomial) para todas as suas instâncias. Em problemas grandes, encontrar uma solução ótima pode levar um tempo exponencial, tornando-o um dos maiores desafios da teoria da computação.

III. Fundamentação Teórica

1) **Problema do Caixeiro Viajante (PCV)**: O Problema do Caixeiro Viajante (PCV) é um dos problemas mais desafiadores da otimização combinatória. Ele envolve encontrar o menor caminho possível que permita a visita a um conjunto de cidades, passando por cada uma exatamente uma vez e retornando ao ponto de partida. O PCV é classificado como NP-difícil, o que significa que não existe uma solução eficiente conhecida para todas as instâncias do problema[1]. Diversas abordagens têm sido desenvolvidas para enfrentar o PCV, desde algoritmos exatos até heurísticas. Entre as heurísticas mais notáveis está o algoritmo de Lin e Kernighan, que é amplamente utilizado por sua eficiência em produzir soluções próximas do ótimo. Este algoritmo utiliza trocas de segmentos de arestas para melhorar iterativamente uma solução inicial, o que permite resolver instâncias de até 100 cidades em tempos razoáveis, com complexidade de $O(n^2)$ [4]. Esse método representa uma das heurísticas mais eficazes para o PCV, sendo aplicado em diversas variantes do problema.
2) **Execução Simbólica**: A execução simbólica é uma técnica poderosa na verificação de software e otimização de problemas complexos. Introduzida por King em 1976, ela trata as variáveis de entrada de um programa como símbolos, em vez de valores concretos, permitindo a exploração de todas as possíveis execuções de um programa. Essa abordagem é particularmente útil em problemas como o PCV, onde o espaço de busca é exponencialmente grande[3]. Na execução simbólica, o programa

é executado de forma simbólica, e todas as condições de execução são mantidas como expressões lógicas que podem ser verificadas por solvers de restrições, como o Z3. Isso permite uma verificação mais eficiente de programas, testando múltiplos cenários de execução simultaneamente.

3) **Satisfiability Modulo Theories (SMT) e a Ferramenta Z3**: O Z3 é uma ferramenta de Satisfiability Modulo Theories (SMT), desenvolvida por De Moura e Bjørner, que se destaca por sua capacidade de resolver fórmulas lógicas complexas envolvendo diferentes teorias, como aritmética, arrays e vetores de bits. O Z3 combina técnicas de Satisfiability Testing (SAT) com a capacidade de lidar com múltiplas teorias lógicas, permitindo verificar propriedades de programas e resolver problemas de otimização que envolvem restrições complexas[5]. Ao integrar diversas teorias lógicas em um único ambiente de solução, o Z3 se tornou uma ferramenta fundamental para a verificação formal de programas, testes automáticos e execução simbólica. Ele é amplamente utilizado em aplicações que exigem análise precisa e otimização de problemas combinatórios, como o Problema do Caixeiro Viajante, além de oferecer suporte para diferentes linguagens de programação, incluindo Python, o que facilita sua integração com diversas plataformas de desenvolvimento[6].
4) **Aplicação da Execução Simbólica e SMT no Problema do Caixeiro Viajante**: A combinação da execução simbólica com o solver SMT Z3 oferece uma abordagem promissora para o PCV. Enquanto a execução simbólica permite explorar múltiplos caminhos no espaço de busca, o Z3 resolve as condições simbólicas geradas ao longo das execuções. Esta combinação é particularmente eficiente para resolver instâncias do PCV com maior precisão e otimização. O uso de SMT solvers possibilita a verificação de restrições complexas, como as associadas à otimização das rotas no PCV, de maneira eficiente e sistemática[5].

IV. DESCRIÇÃO E COMPLEXIDADE DOS PRINCIPAIS ALGORITMOS UTILIZADOS PELA FERRAMENTA Z3

A biblioteca Z3 implementa uma série de algoritmos otimizados para resolver problemas complexos de Satisfiability Modulo Theories (SMT). Os principais algoritmos envolvidos incluem:

1) **E-Matching**: O Z3 usa um motor de correspondência eficiente (E-matching) para lidar com quantificadores. Quantificadores, como $\forall$ (para todo) e $\exists$ (existe), são comuns em fórmulas SMT, e instanciá-los corretamente é um desafio significativo. O E-matching é uma técnica que trabalha sobre o grafo de equivalência (E-graph) para identificar padrões de variáveis quantificadas e encontrar instâncias onde esses padrões podem ser aplicados. Ao usar o E-matching, o Z3 pode instanciar variáveis de maneira eficiente, identificando quais valores de variáveis satisfazem a fórmula quantificada sem gerar instâncias desnecessárias. O Z3 implementa novos algoritmos de E-matching que identificam correspondências em E-graphs de forma incremental. Isso significa que ele não precisa refazer o processo de correspondência toda vez que uma nova informação é introduzida, melhorando significativamente o desempenho.[7]. [7] menciona que, embora o problema de E-matching seja teoricamente NP-difícil (NP-hard), o número de correspondências pode ser exponencial no tamanho do E-grafo. No entanto, o impacto prático do uso de E-matching para a instanciação de quantificadores resulta principalmente da busca e manutenção de conjuntos de padrões que podem recuperar novas correspondências de maneira eficiente conforme as operações no E-grafo as introduzem.
2) **SAT Solver baseado no DPLL**: O núcleo do Z3 é um resolvedor SAT moderno, baseado no algoritmo DPLL (Davis-Putnam-Logemann-Loveland). O DPLL é uma abordagem clássica para a resolução de problemas de satisfatibilidade lógica, onde o objetivo é determinar se uma fórmula booleana pode ser avaliada como verdadeira ou falsa. Este problema pertence à classe NP-commpleto, que significa que a verificação de uma solução é feita em tempo polinomial, porém, encontrar a solução em si pode ser exponencial. A complexidade do DPLL, no pior caso, é $O(2^n)$, onde n é o número de variáveis. Apesar de utilizar otimizações como a propagação de restrições booleanas, sua natureza exploratória faz com que a complexidade permaneça exponencial no pior caso, devido ao grande número de possíveis atribuições de variáveis a serem consideradas[11].

V. FERRAMENTAS UTILIZADAS

O desenvolvimento do código para resolver o Problema do Caixeiro Viajante (PCV) foi feito utilizando duas ferramentas principais: Python[8] e a biblioteca Z3[5].

O Python foi escolhido como base devido à sua simplicidade e versatilidade. A linguagem oferece uma ampla gama de bibliotecas e ferramentas que facilitam o desenvolvimento de soluções complexas. Além disso, a integração do Z3 com Python permite a utilização eficaz do solver para resolver problemas de otimização e lógica simbólica.

Paralelamente, utilizou-se o Z3, um solver SMT (Satisfiability Modulo Theories) desenvolvido pela Microsoft Research. O Z3 é projetado para resolver problemas que envolvem restrições lógicas e aritméticas e é amplamente utilizado em otimização simbólica, verificação de software e resolução de problemas combinatórios, como o PCV. No contexto do PCV, o Z3 foi essencial para:

1) **Definição de variáveis de decisão simbólicas**: As rotas entre as cidades foram modeladas como variáveis simbólicas que o Z3 pode manipular e resolver.
2) **Imposição de restrições**: O Z3 garantiu que cada cidade fosse visitada exatamente uma vez e evitou a formação de subciclos.
3) **Busca pela solução otimizada**: O Z3 calculou a rota que minimiza a distância total percorrida, considerando todas as restrições aplicadas.

' Em suma, a integração de Python com o Z3 resultou em uma solução eficaz e acessível para o Problema do Caixeiro Viajante, oferecendo uma abordagem robusta, flexível e eficiente para resolver o problema.

VI. Desenvolvimento da solução

O desenvolvimento do Executor Simbólico para o Problema do Caixeiro Viajante (PCV) começou com uma análise detalhada dos conceitos fundamentais da execução simbólica e de como ela poderia ser aplicada de forma eficiente na resolução deste problema específico. Para isso, partimos de uma base teórica sólida, utilizando o artigo de King (1976) como referência inicial. Com esse embasamento, direcionamos nossos esforços para a implementação do solver Z3, que desempenhou um papel crucial na formulação simbólica das variáveis e restrições que compõem o PCV.

A primeira etapa prática envolveu a definição de variáveis simbólicas que representassem as rotas entre as cidades. Cada uma dessas rotas foi modelada como uma variável inteira $x[i][j]$, onde i e j representam as cidades de origem e destino, respectivamente. Estas variáveis foram tratadas de maneira simbólica, o que nos permitiu explorar todas as possíveis combinações de rotas. Usamos o Z3 para manipular essas variáveis de decisão e garantir que cada caminho respeitasse as restrições impostas pelo problema.

O Z3 foi particularmente eficiente ao impor restrições cruciais para o PCV. Para garantir que o caixeiro viajante visitasse cada cidade exatamente uma vez, introduzimos uma série de restrições que limitavam as saídas e entradas de cada cidade. Isso foi feito através de somatórios que asseguravam que de cada cidade só haveria uma rota de saída e que cada cidade só seria visitada uma única vez. Adicionalmente, para evitar que o caixeiro formasse subciclos (circuitos internos menores que prejudicariam a solução global), utilizamos variáveis auxiliares $u[i]$, que funcionam como etiquetas de ordem nas cidades visitadas, assegurando que o solver mantivesse um progresso contínuo e evitasse rotas redundantes.

A função objetivo do nosso Executor Simbólico foi então definida de maneira clara: minimizar a soma das distâncias percorridas entre as cidades. Para isso, modelamos a matriz de distâncias como um conjunto de constantes que foi multiplicado pelas variáveis de decisão $x[i][j]$. O Z3 foi utilizado para calcular a soma total dessas distâncias e encontrar a rota que minimizava o valor objetivo, garantindo que a solução respeitasse todas as restrições simbólicas impostas.

A integração entre a execução simbólica e o solver Z3 foi um fator decisivo para o sucesso da implementação. O Z3, com suas capacidades de manipular teorias complexas e variáveis simbólicas, permitiu que explorássemos o espaço de soluções do PCV de maneira eficiente e otimizada. Ao desenvolver esta solução em Python, conseguimos aproveitar a simplicidade e flexibilidade da linguagem, permitindo que o código fosse tanto acessível quanto eficaz.

Essa abordagem, centrada no uso do Executor Simbólico, demonstrou-se robusta para lidar com a complexidade inerente do Problema do Caixeiro Viajante, ao mesmo tempo em que utilizava as ferramentas de otimização fornecidas pelo Z3 para garantir uma solução viável e otimizada.

VII. Complexidade da Solução

A complexidade do código para resolver o *Problema do Caixeiro Viajante* com o Z3 é influenciada pelos algoritmos internos da ferramenta, especialmente o *SAT Solver* baseado no *DPLL* (Davis-Putnam-Logemann-Loveland). O número de variáveis booleanas $x[i][j]$, que representam as rotas entre as cidades, cresce quadraticamente com o número de cidades n. No entanto, a complexidade do solver está associada principalmente ao número de decisões que ele precisa fazer, resultando em uma complexidade *exponencial*.

O algoritmo *DPLL*, utilizado para resolver problemas de satisfatibilidade, tem uma complexidade de $O(2^n)$ no pior caso, onde n é o número de cidades. Mesmo com otimizações, como a propagação de restrições e o aprendizado de cláusulas, a natureza exponencial do problema permanece. Portanto, ao aplicar o Z3 ao Problema do Caixeiro Viajante, a complexidade final do código é $O(2^n)$ [5], [11].

VIII. Resultados Obtidos

Após a conclusão do desenvolvimento do código, realizamos testes para verificar o comportamento da solução com diferentes entradas de dados, e se ela seria capaz de encontrar as soluções corretas para as matrizes fornecidas. A seguir, apresentamos o desempenho da nossa solução com as matrizes de exemplo:

- **Matriz 4x4**: Essa matriz representa um problema com 4 cidades, onde analisamos distâncias relativamente longas.

$$\begin{bmatrix} 0 & 100 & 150 & 200 \\ 100 & 0 & 120 & 80 \\ 150 & 120 & 0 & 90 \\ 200 & 80 & 90 & 0 \end{bmatrix}$$

- **Matriz 5x5**: Aqui, a matriz representa 5 cidades com distâncias muito próximas entre si.

$$\begin{bmatrix} 0 & 2 & 3 & 4 & 5 \\ 2 & 0 & 2 & 3 & 4 \\ 3 & 2 & 0 & 2 & 3 \\ 4 & 3 & 2 & 0 & 2 \\ 5 & 4 & 3 & 2 & 0 \end{bmatrix}$$

Nosso desenvolvimento se baseia em uma função chamada `tsp_solver` que é chamada com tendo como entrada a matriz que queremos responder, desta função estão todas as verificações e execuções do nosso codigo este codigo pode ser totalmente visualizado a partir do repositorio do GitHub[10]

A seguir, apresentamos as soluções obtidas para cada uma das matrizes fornecidas e o processo de verificação:

- **Matriz 4x4**:

  Número de cidades: 4

  Caminho sugerido: [0, 2, 3, 1, 0]

  Cálculo da soma das distâncias:
  $0 \rightarrow 2$: 150
  $2 \rightarrow 3$: 90
  $3 \rightarrow 1$: 80

$1 \rightarrow 0$: 100

Soma total: 150 + 90 + 80 + 100 = 420

Verificação de outros caminhos possíveis:
100 + 120 + 90 + 200 = 510
100 + 80 + 90 + 150 = 420
150 + 120 + 80 + 200 = 550
150 + 90 + 80 + 100 = 420
200 + 80 + 120 + 150 = 550
200 + 90 + 120 + 100 = 510

O caminho sugerido de custo 420 é de fato o menor. Solução correta.

- **Matriz 5x5**:

  Número de cidades: 5
  Solução encontrada:
  Caminho sugerido: [0, 1, 4, 3, 2, 0]

  Cálculo da soma das distâncias:
  $0 \rightarrow 1$: 2
  $1 \rightarrow 4$: 4
  $4 \rightarrow 3$: 2
  $3 \rightarrow 2$: 2
  $2 \rightarrow 0$: 3
  Soma total: 2 + 4 + 2 + 2 + 3 = 13

  Verificação de outros caminhos possíveis:
  2 + 2 + 2 + 2 + 5 = 13
  2 + 2 + 3 + 2 + 4 = 13
  2 + 3 + 2 + 3 + 5 = 15
  2 + 3 + 2 + 3 + 3 = 13
  2 + 4 + 3 + 2 + 4 = 15
  2 + 4 + 2 + 2 + 3 = 13
  3 + 2 + 3 + 2 + 5 = 15
  3 + 2 + 4 + 2 + 4 = 15
  3 + 2 + 3 + 4 + 5 = 17
  3 + 2 + 2 + 4 + 2 = 13
  3 + 3 + 4 + 3 + 4 = 17
  3 + 3 + 2 + 3 + 2 = 13
  4 + 3 + 2 + 3 + 5 = 17
  4 + 3 + 4 + 3 + 3 = 17
  4 + 2 + 2 + 4 + 5 = 17
  4 + 2 + 3 + 4 + 2 = 15
  4 + 2 + 4 + 2 + 3 = 15
  4 + 2 + 3 + 2 + 2 = 13
  5 + 4 + 2 + 2 + 4 = 17
  5 + 4 + 3 + 2 + 3 = 17
  5 + 3 + 2 + 3 + 4 = 17
  5 + 3 + 2 + 3 + 2 = 15
  5 + 2 + 3 + 2 + 3 = 15
  5 + 2 + 2 + 2 + 2 = 13

  O caminho sugerido de custo 13 é de fato o menor. Solução correta.

- **Matriz 20x20**: Essa matriz representa um problema com 20 cidades, onde foram analisadas distâncias variadas entre as cidades.
  Por motivos de formatação a cada duas linhas na matriz de exemplo é representada uma linha na matriz original.

$$\begin{bmatrix}
0 & 24 & 16 & 32 & 10 & 25 & 38 & 43 & 18 & 27 \\
14 & 41 & 35 & 22 & 39 & 47 & 15 & 30 & 42 & 19 \\
24 & 0 & 20 & 12 & 30 & 28 & 31 & 11 & 33 & 40 \\
29 & 38 & 21 & 27 & 23 & 10 & 39 & 44 & 17 & 36 \\
16 & 20 & 0 & 25 & 12 & 39 & 18 & 21 & 34 & 15 \\
36 & 45 & 19 & 23 & 14 & 33 & 42 & 29 & 48 & 22 \\
32 & 12 & 25 & 0 & 45 & 17 & 22 & 30 & 19 & 28 \\
24 & 41 & 15 & 14 & 40 & 27 & 26 & 13 & 44 & 32 \\
10 & 30 & 12 & 45 & 0 & 26 & 38 & 20 & 48 & 23 \\
31 & 39 & 22 & 14 & 20 & 17 & 21 & 10 & 34 & 46 \\
25 & 28 & 39 & 17 & 26 & 0 & 29 & 11 & 19 & 38 \\
10 & 16 & 35 & 27 & 23 & 45 & 20 & 30 & 24 & 34 \\
38 & 31 & 18 & 22 & 38 & 29 & 0 & 42 & 10 & 45 \\
36 & 24 & 22 & 30 & 11 & 40 & 20 & 48 & 39 & 23 \\
43 & 11 & 21 & 30 & 20 & 11 & 42 & 0 & 50 & 13 \\
36 & 25 & 12 & 32 & 17 & 19 & 29 & 35 & 22 & 30 \\
18 & 33 & 34 & 19 & 48 & 19 & 10 & 50 & 0 & 29 \\
27 & 43 & 36 & 23 & 41 & 33 & 15 & 40 & 22 & 11 \\
27 & 40 & 15 & 28 & 23 & 38 & 45 & 13 & 29 & 0 \\
20 & 14 & 35 & 30 & 19 & 11 & 16 & 32 & 28 & 43 \\
14 & 29 & 36 & 24 & 31 & 10 & 36 & 36 & 27 & 20 \\
0 & 12 & 28 & 19 & 21 & 30 & 39 & 22 & 18 & 25 \\
41 & 38 & 45 & 41 & 39 & 16 & 24 & 25 & 43 & 14 \\
12 & 0 & 35 & 22 & 31 & 19 & 20 & 37 & 21 & 24 \\
35 & 21 & 19 & 15 & 22 & 35 & 22 & 12 & 36 & 35 \\
28 & 35 & 0 & 30 & 48 & 42 & 25 & 38 & 16 & 30 \\
22 & 27 & 23 & 14 & 14 & 27 & 30 & 32 & 23 & 30 \\
19 & 22 & 30 & 0 & 45 & 40 & 33 & 10 & 50 & 31 \\
39 & 23 & 14 & 40 & 20 & 23 & 11 & 17 & 41 & 19 \\
21 & 31 & 48 & 45 & 0 & 10 & 15 & 34 & 27 & 19 \\
47 & 10 & 33 & 27 & 17 & 45 & 40 & 19 & 33 & 11 \\
30 & 19 & 42 & 40 & 10 & 0 & 20 & 29 & 44 & 38 \\
15 & 39 & 42 & 26 & 21 & 20 & 20 & 29 & 15 & 16 \\
39 & 20 & 25 & 33 & 15 & 20 & 0 & 45 & 36 & 50 \\
30 & 44 & 29 & 13 & 10 & 30 & 48 & 35 & 40 & 32 \\
22 & 37 & 38 & 10 & 34 & 29 & 45 & 0 & 19 & 15 \\
42 & 17 & 48 & 44 & 34 & 24 & 39 & 22 & 22 & 28 \\
18 & 21 & 16 & 50 & 27 & 44 & 36 & 19 & 0 & 12 \\
19 & 36 & 22 & 32 & 46 & 34 & 23 & 30 & 11 & 43 \\
25 & 24 & 30 & 31 & 19 & 38 & 50 & 15 & 12 & 0
\end{bmatrix}$$

Após mais de 6 horas de execução, o código não retornou uma solução viável dentro do tempo esperado. Esse resultado está alinhado com a natureza do problema do Caixeiro Viajante, que é NP-difícil, e cujo tempo de execução aumenta exponencialmente com o número de cidades. Para instâncias com 10 ou mais cidades, é esperado que o solver leve mais tempo para encontrar uma solução ou mesmo não consiga em um tempo razoável. Esse comportamento não indica um erro no código, mas sim uma possível limitação do desenvolvimento, visto que o aumento no número de cidades impacta negativamente

o tempo de execução. Logo, não foi possível avaliar com detalhes o resultado dessa matriz.

IX. Análise dos Resultados

Os testes realizados com diferentes matrizes de distâncias evidenciam a eficácia e robustez do algoritmo. Para cada matriz fornecida, o algoritmo foi capaz de encontrar o caminho com o menor custo total, respeitando as restrições impostas pelo Problema do Caixeiro Viajante (TSP). A seguir, detalhamos os principais aspectos observados durante a execução dos testes:

1) **Comportamento em Matriz 4x4 (Distâncias Longas)**: Na matriz com 4 cidades e distâncias consideravelmente longas, o algoritmo foi capaz de encontrar a solução ótima com um custo total de 420. Durante a verificação de outros caminhos possíveis, o algoritmo garantiu que não havia rota mais curta. Esse resultado demonstra a eficiência em cenários onde há uma grande disparidade entre as distâncias, como foi o caso dos valores que variavam entre 80 e 200. Mesmo com essa variação, a solução sugerida foi correta, e o tempo de execução permaneceu dentro dos limites aceitáveis para esse tamanho de matriz.
   A análise dos caminhos alternativos confirmou que a rota sugerida era de fato a mais eficiente. Isso demonstra que o algoritmo foi capaz de realizar uma boa exploração do espaço de busca, minimizando o custo total sem comprometer a correção da solução.
2) **Comportamento em Matriz 5x5 (Cidades Próximas)**: Na matriz 5x5, onde as distâncias entre as cidades são menores e mais próximas, o algoritmo também foi eficiente em encontrar a solução ótima com custo total de 13. Este cenário é interessante pois as distâncias entre as cidades são muito semelhantes, o que poderia aumentar a dificuldade em diferenciar entre caminhos alternativos. Contudo, conseguiu identificar o melhor caminho, demonstrando sua capacidade de lidar com pequenos valores e pequenas diferenças entre as distâncias. Durante a verificação de outros caminhos possíveis, foi observado que vários caminhos tinham custos similares, mas nenhum deles foi inferior ao caminho sugerido pelo algoritmo, validando mais uma vez a precisão da solução obtida. O desempenho do algoritmo se manteve estável, mesmo com um número maior de cidades e distâncias relativamente pequenas, reforçando sua capacidade de resolver problemas de TSP em diferentes contextos.
3) **Comportamento em Matriz 20x20 (Número elevado de cidades)**: No caso da matriz 20x20, o algoritmo não conseguiu encontrar uma solução satisfatória mesmo após 6 horas de execução contínua. É importante notar que, mesmo com o código correto, o problema do Caixeiro Viajante é NP-difícil, e o tempo de execução aumenta exponencialmente com o número de cidades. Para instâncias com 10 ou mais cidades, é esperado que o solver leve mais tempo para encontrar uma solução ou mesmo não consiga em tempo razoável. Isso não indica um erro no código, mas sim a natureza computacionalmente intensa do problema. Os testes até 10 cidades foram realizadas de forma satisfatoria o que indica que pode ser uma limitação da implementação para conseguir achar um resultado com uma entrada maior.
4) **Validação da Precisão dos Resultados**:A validação dos resultados obtidos foi realizada por meio da comparação com todos os caminhos possíveis gerados pelas permutações das cidades. Essa etapa foi importante para assegurar que o caminho sugerido fosse de fato o mais curto. A análise mostrou que, em ambos os casos, o algoritmo foi capaz de encontrar a rota com o menor custo total. A exibição dos outros caminhos possíveis e seus respectivos custos confirma a precisão da solução proposta.
   Além disso, o fato de que o algoritmo foi testado com diferentes configurações de matrizes, tanto com distâncias longas quanto curtas, demonstra sua versatilidade em resolver problemas de TSP em diversos contextos, sem comprometer a precisão ou eficiência.

X. Justificativa para o Uso da Execução Simbólica

A execução simbólica se destaca como uma alternativa eficiente aos métodos tradicionais de teste de software, especialmente em contextos onde há uma vasta gama de possíveis entradas e ramificações condicionais. Em vez de depender exclusivamente de um conjunto limitado de casos de teste concretos, a execução simbólica utiliza variáveis simbólicas para representar as possíveis entradas, permitindo uma análise abrangente de todas as execuções potenciais do programa. Ao simular todas as possíveis ramificações e condições com base em fórmulas simbólicas, esse método oferece uma cobertura muito maior, garantindo que uma maior parte do espaço de busca seja explorada[3]. Além disso, a execução simbólica permite testar programas complexos de forma mais eficiente, ao resolver automaticamente condições lógicas e matemáticas associadas ao comportamento do programa. Isso é feito através do uso de solvers SMT, como o Z3, que verifica formalmente se as condições geradas durante a execução simbólica são satisfeitas. Essa abordagem permite não apenas identificar possíveis erros no código, mas também garantir a correção formal do programa de forma automática, sem a necessidade de intervenção manual, como destacam os estudos sobre o Z3[5]. Portanto, ao oferecer uma cobertura mais ampla e integrar métodos formais de verificação, a execução simbólica supera as limitações dos métodos tradicionais de teste, sendo especialmente eficaz na exploração de múltiplos caminhos de execução e na verificação automática de propriedades complexas.

XI. Conclusão

Este trabalho apresentou o desenvolvimento de um executor simbólico para o Problema do Caixeiro Viajante (PCV) utilizando a linguagem Python e a biblioteca Z3. A execução simbólica, combinada com o uso do solucionador SMT, demonstrou ser uma abordagem eficaz para explorar múltiplas soluções possíveis e verificar restrições lógicas associadas ao problema. O Z3 provou ser uma ferramenta eficiente para lidar

com as restrições impostas pelo PCV, garantindo que todas as cidades fossem visitadas uma única vez e evitando a formação de subciclos, enquanto buscava a rota de menor custo total.

Os testes realizados em matrizes de distâncias com diferentes configurações confirmaram a precisão e robustez da solução desenvolvida. Tanto em cenários com distâncias longas quanto em cenários com distâncias curtas, o algoritmo foi capaz de encontrar a solução ótima, reforçando a sua versatilidade e eficiência.

Por fim, a integração entre a execução simbólica e o solver Z3, aliado à simplicidade do Python, mostrou-se uma abordagem poderosa para resolver problemas de otimização combinatória complexos como o PCV. Este trabalho abre caminho para novas explorações no uso de ferramentas SMT e técnicas de execução simbólica na solução de outros problemas de otimização.

REFERENCES

[1] L. R. Lopes and F. H. V. Martinez, "O Problema do Caixeiro Viajante," 2023.

[2] M. Goldbarg and E. Goldbarg, *Grafos: conceitos, algoritmos e aplicações*. Elsevier, 2012.

[3] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385-394, 1976.

[4] S. Lin and B. W. Kernighan, "An effective heuristic algorithm for the traveling-salesman problem," *Operations Research*, vol. 21, no. 2, pp. 498-516, 1973.

[5] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337-340.

[6] L. De Moura and N. Bjørner, "Satisfiability modulo theories: introduction and applications," *Communications of the ACM*, vol. 54, no. 9, pp. 69-77, 2011.

[7] L. De Moura and N. Bjørner, "Efficient E-matching for SMT solvers," in *Automated Deduction–CADE-21: 21st International Conference on Automated Deduction*, Bremen, Germany, July 17-20, 2007, Proceedings 21, Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 183-198.

[8] Python Software Foundation, *Python Documentation*, Disponível em: https://docs.python.org/, Acesso em: outubro de 2023.

[9] L. de Moura and N. Bjørner, *Z3: An Efficient SMT Solver*, em *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, Springer-Verlag, 2008.

[10] R. Vicente and V. Oliveira, *RosialdoVenicius_FinalProject_AA_RR_2024*, Disponível em: https://github.com/Rosialdo/RosialdoVenicius_FinalProject_AA_RR_2024, Acesso em: setembro de 2024.

[11] Cas Craven, Bhargavi Narayanasetty, Dan Zhang, *Solving Satisfiability with a Novel CPU/GPU Hybrid Solution*, University of Texas, Department of Electrical and Computer Engineering, 2008.