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

Ven´ıcius 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 TSP 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**

1. O PROBLEMA DO CAIXEIRO VIAJANTE

O Problema do Caixeiro Viajante (PCV), conforme descrito por [1], consiste em encontrar a rota mais curta poss´ıvel para visitar um conjunto de cidades, partindo de uma cidade de origem, passando por todas as outras exatamente uma vez e retornando a` cidade de partida. Apesar de sua formulac¸a˜o simples, o PCV e´ um problema amplamente reconhecido e

**solutions in the context of TSP.**

***Index Terms*—Executor Simbo´lico, Problema do Caixeiro Viajante, Python, Z3, Otimizac¸a˜o Combinato´ria, Ana´lise de Restric¸o˜es, Algoritmos, Execuc¸a˜o Simbo´lica.**

* 1. INTRODUC¸ A˜ O

**O**

Problema do Caixeiro Viajante (PCV) e´ um dos prob- lemas mais conhecidos da teoria da computac¸a˜o e da otimizac¸a˜o combinato´ria. Ele consiste em encontrar o menor caminho que permite a visita a um conjunto de cidades, retornando a` cidade de origem, passando por cada cidade uma u´nica vez. Este problema e´ classificado como NP-dif´ıcil, o que significa que encontrar uma soluc¸a˜o o´tima em tempo polinomial para todas as instaˆncias e´ considerado invia´vel na

pra´tica [1].

Diversas abordagens heur´ısticas e exatas teˆm sido propostas ao longo dos anos para tratar o PCV. Entre as heur´ısticas, destacam-se os algoritmos de Lin e Kernighan e o de inserc¸a˜o de ve´rtices, que buscam soluc¸o˜es aproximadas para o problema[2]. No entanto, o avanc¸o de me´todos simbo´licos, como a execuc¸a˜o simbo´lica, tem proporcionado novos cam- inhos para a resoluc¸a˜o de problemas complexos, como o PCV[3]. A execuc¸a˜o simbo´lica permite explorar mu´ltiplas possibilidades de execuc¸a˜o ao tratar varia´veis de entrada como s´ımbolos, oferecendo uma forma mais abrangente de testar e verificar programas[3].

Neste contexto, a biblioteca Z3, um solucionador de Sat- isfatibilidade Modulo Theories (SMT), desenvolvida pela Mi- crosoft Research, destaca-se como uma ferramenta poderosa para a execuc¸a˜o simbo´lica e a verificac¸a˜o de programa[5]. O Z3 permite a combinac¸a˜o de diferentes teorias de primeira or- dem, como aritme´tica e vetores de bits, facilitando a resoluc¸a˜o de problemas de otimizac¸a˜o. Este trabalho visa desenvolver um Executor Simbo´lico para o PCV utilizando Python e a biblioteca Z3, com o objetivo de explorar a eficieˆncia dessa abordagem na busca por soluc¸o˜es otimizadas.

Setembro 23, 2024

estudado na a´rea de otimizac¸a˜o combinato´ria. Ele pertence

a` classe de problemas NP-dif´ıceis, o que significa que na˜o se conhece um algoritmo que sempre consiga resolver o problema de forma eficiente (em tempo polinomial) para todas as suas instaˆncias. Em problemas grandes, encontrar uma soluc¸a˜o o´tima pode levar um tempo exponencial, tornando-o um dos maiores desafios da teoria da computac¸a˜o.

1. FUNDAMENTAC¸ A˜ O TEO´ RICA
2. **Problema do Caixeiro Viajante (PCV)**: O Problema do Caixeiro Viajante (PCV) e´ um dos problemas mais desafiadores da otimizac¸a˜o combinato´ria. Ele envolve encontrar o menor caminho poss´ıvel que permita a visita a um conjunto de cidades, passando por cada uma exatamente uma vez e retornando ao ponto de partida. O PCV e´ classificado como NP-dif´ıcil, o que significa que na˜o existe uma soluc¸a˜o eficiente conhecida para todas as instaˆncias do problema[1]. Diversas abordagens teˆm sido desenvolvidas para enfrentar o PCV, desde algoritmos exatos ate´ heur´ısticas. Entre as heur´ısticas mais nota´veis esta´ o algoritmo de Lin e Kernighan, que e´ amplamente utilizado por sua eficieˆncia em produzir soluc¸o˜es pro´ximas do o´timo. Este algoritmo utiliza tro- cas de segmentos de arestas para melhorar iterativamente uma soluc¸a˜o inicial, o que permite resolver instaˆncias de ate´ 100 cidades em tempos razoa´veis, com complexidade de O(n²) [4]. Esse me´todo representa uma das heur´ısticas mais eficazes para o PCV, sendo aplicado em diversas variantes do problema.
3. **Execuc¸a˜o Simbo´lica**: A execuc¸a˜o simbo´lica e´ uma te´cnica poderosa na verificac¸a˜o de software e otimizac¸a˜o de problemas complexos. Introduzida por King em 1976, ela trata as varia´veis de entrada de um programa como s´ımbolos, em vez de valores concretos, permitindo a explorac¸a˜o de todas as poss´ıveis execuc¸o˜es de um pro- grama. Essa abordagem e´ particularmente u´til em prob- lemas como o PCV, onde o espac¸o de busca e´ exponen- cialmente grande[3]. Na execuc¸a˜o simbo´lica, o programa

e´ executado de forma simbo´lica, e todas as condic¸o˜es de execuc¸a˜o sa˜o mantidas como expresso˜es lo´gicas que podem ser verificadas por solvers de restric¸o˜es, como o Z3. Isso permite uma verificac¸a˜o mais eficiente de programas, testando mu´ltiplos cena´rios de execuc¸a˜o si- multaneamente.

1. **Satisfiability Modulo Theories (SMT) e a Ferra- menta Z3**: O Z3 e´ uma ferramenta de Satisfiability Modulo Theories (SMT), desenvolvida por De Moura e Bjørner, que se destaca por sua capacidade de resolver fo´rmulas lo´gicas complexas envolvendo diferentes teo- rias, como aritme´tica, arrays e vetores de bits. O Z3 combina te´cnicas de Satisfiability Testing (SAT) com a capacidade de lidar com mu´ltiplas teorias lo´gicas, permitindo verificar propriedades de programas e re- solver problemas de otimizac¸a˜o que envolvem restric¸o˜es complexas[5]. Ao integrar diversas teorias lo´gicas em

plementa novos algoritmos de E-matching que iden- tificam correspondeˆncias em E-graphs de forma in- cremental. Isso significa que ele na˜o precisa refazer o processo de correspondeˆncia toda vez que uma nova informac¸a˜o e´ introduzida, melhorando significati- vamente o desempenho.[7]. [7] menciona que, embora o problema de E-matching seja teoricamente NP-dif´ıcil (NP-hard), o nu´mero de correspondeˆncias pode ser ex- ponencial no tamanho do E-grafo. No entanto, o im- pacto pra´tico do uso de E-matching para a instanciac¸a˜o de quantificadores resulta principalmente da busca e manutenc¸a˜o de conjuntos de padro˜es que podem re- cuperar novas correspondeˆncias de maneira eficiente conforme as operac¸o˜es no E-grafo as introduzem.

2) **SAT Solver baseado no DPLL**: O nu´cleo do Z3 e´ um resolvedor SAT moderno, baseado no algoritmo DPLL (Davis-Putnam-Logemann-Loveland). O DPLL e´

um u´nico ambiente de soluc¸a˜o, o Z3 se tornou uma uma abordagem cla´ssica para a resoluc¸a˜o de problemas

ferramenta fundamental para a verificac¸a˜o formal de pro-

de satisfatibilidade lo´gica, onde o objetivo e´

determi-

gramas, testes automa´ticos e execuc¸a˜o simbo´lica. Ele e´ nar se uma fo´rmula booleana pode ser avaliada como

amplamente utilizado em aplicac¸o˜es que exigem ana´lise

verdadeira ou falsa. Este problema pertence a`

classe

precisa e otimizac¸a˜o de problemas combinato´rios, como o Problema do Caixeiro Viajante, ale´m de oferecer suporte para diferentes linguagens de programac¸a˜o, in- cluindo Python, o que facilita sua integrac¸a˜o com diver- sas plataformas de desenvolvimento[6].

4) **Aplicac¸a˜o da Execuc¸a˜o Simbo´lica e SMT no Prob- lema do Caixeiro Viajante**: A combinac¸a˜o da execuc¸a˜o simbo´lica com o solver SMT Z3 oferece uma abordagem promissora para o PCV. Enquanto a execuc¸a˜o simbo´lica permite explorar mu´ltiplos caminhos no espac¸o de busca, o Z3 resolve as condic¸o˜es simbo´licas geradas ao longo das execuc¸o˜es. Esta combinac¸a˜o e´ particularmente efi- ciente para resolver instaˆncias do PCV com maior pre- cisa˜o e otimizac¸a˜o. O uso de SMT solvers possibilita a verificac¸a˜o de restric¸o˜es complexas, como as associadas a` otimizac¸a˜o das rotas no PCV, de maneira eficiente e sistema´tica[5].

1. DESCRIC¸ A˜ O E COMPLEXIDADE DOS PRINCIPAIS ALGORITMOS UTILIZADOS PELA FERRAMENTA Z3

A biblioteca Z3 implementa uma se´rie de algoritmos otimizados para resolver problemas complexos de Satisfiability

NP-commpleto, que significa que a verificac¸a˜o de uma soluc¸a˜o e´ feita em tempo polinomial, pore´m, encontrar a soluc¸a˜o em si pode ser exponencial. A complexidade do DPLL, no pior caso, e´ *O*(2*n*), onde *n* e´ o nu´mero de varia´veis. Apesar de utilizar otimizac¸o˜es como a propagac¸a˜o de restric¸o˜es booleanas, sua natureza explo- rato´ria faz com que a complexidade permanec¸a exponen- cial no pior caso, devido ao grande nu´mero de poss´ıveis atribuic¸o˜es de varia´veis a serem consideradas[11].

1. FERRAMENTAS UTILIZADAS

O desenvolvimento do co´digo 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 a` sua simplicidade e versatilidade. A linguagem oferece uma ampla gama de bibliotecas e ferramentas que facilitam o desenvolvimento de soluc¸o˜es complexas. Ale´m disso, a integrac¸a˜o do Z3 com Python permite a utilizac¸a˜o eficaz do solver para resolver problemas de otimizac¸a˜o e lo´gica simbo´lica.

Paralelamente, utilizou-se o Z3, um solver SMT (Satisfiabil- ity Modulo Theories) desenvolvido pela Microsoft Research.

Modulo Theories (SMT). Os principais algoritmos envolvidos

O Z3 e´

projetado para resolver problemas que envolvem

incluem:

1. **E-Matching**: O Z3 usa um motor de correspondeˆncia eficiente (E-matching) para lidar com quantificadores. Quantificadores, como (para todo) e (existe), sa˜o comuns em fo´rmulas SMT, e instancia´-los correta- mente e´ um desafio significativo. O E-matching e´ uma te´cnica que trabalha sobre o grafo de equivaleˆncia (E- graph) para identificar padro˜es de varia´veis quantifi- cadas e encontrar instaˆncias onde esses padro˜es po- dem ser aplicados. Ao usar o E-matching, o Z3 pode instanciar varia´veis de maneira eficiente, identificando quais valores de varia´veis satisfazem a fo´rmula quan- tificada sem gerar instaˆncias desnecessa´rias. O Z3 im-

∀ ∃

restric¸o˜es lo´gicas e aritme´ticas e e´ amplamente utilizado em otimizac¸a˜o simbo´lica, verificac¸a˜o de software e resoluc¸a˜o de problemas combinato´rios, como o PCV. No contexto do PCV, o Z3 foi essencial para:

1. **Definic¸a˜o de varia´veis de decisa˜o simbo´licas**: As rotas entre as cidades foram modeladas como varia´veis simbo´licas que o Z3 pode manipular e resolver.
2. **Imposic¸a˜o de restric¸o˜es**: O Z3 garantiu que cada cidade fosse visitada exatamente uma vez e evitou a formac¸a˜o de subciclos.
3. **Busca pela soluc¸a˜o otimizada**: O Z3 calculou a rota que minimiza a distaˆncia total percorrida, considerando todas as restric¸o˜es aplicadas.

’ Em suma, a integrac¸a˜o de Python com o Z3 resultou em uma soluc¸a˜o eficaz e acess´ıvel para o Problema do Caixeiro Via- jante, oferecendo uma abordagem robusta, flex´ıvel e eficiente para resolver o problema.

1. DESENVOLVIMENTO DA SOLUC¸ A˜ O

O desenvolvimento do Executor Simbo´lico para o Prob- lema do Caixeiro Viajante (PCV) comec¸ou com uma ana´lise detalhada dos conceitos fundamentais da execuc¸a˜o simbo´lica e de como ela poderia ser aplicada de forma eficiente na resoluc¸a˜o deste problema espec´ıfico. Para isso, partimos de uma base teo´rica so´lida, utilizando o artigo de King (1976) como refereˆncia inicial. Com esse embasamento, direcionamos nossos esforc¸os para a implementac¸a˜o do solver Z3, que desempenhou um papel crucial na formulac¸a˜o simbo´lica das varia´veis e restric¸o˜es que compo˜em o PCV.

A primeira etapa pra´tica envolveu a definic¸a˜o de varia´veis simbo´licas que representassem as rotas entre as cidades. Cada uma dessas rotas foi modelada como uma varia´vel inteira *x*[*i*][*j*], onde *i* e *j* representam as cidades de origem e destino, respectivamente. Estas varia´veis foram tratadas de maneira simbo´lica, o que nos permitiu explorar todas as poss´ıveis combinac¸o˜es de rotas. Usamos o Z3 para manipular essas varia´veis de decisa˜o e garantir que cada caminho respeitasse as restric¸o˜es impostas pelo problema.

O Z3 foi particularmente eficiente ao impor restric¸o˜es cruci- ais para o PCV. Para garantir que o caixeiro viajante visitasse cada cidade exatamente uma vez, introduzimos uma se´rie de restric¸o˜es que limitavam as sa´ıdas e entradas de cada cidade. Isso foi feito atrave´s de somato´rios que asseguravam que de cada cidade so´ haveria uma rota de sa´ıda e que cada cidade so´ seria visitada uma u´nica vez. Adicionalmente, para evitar que o caixeiro formasse subciclos (circuitos internos menores que prejudicariam a soluc¸a˜o global), utilizamos varia´veis auxiliares *u*[*i*], que funcionam como etiquetas de ordem nas cidades visitadas, assegurando que o solver mantivesse um progresso cont´ınuo e evitasse rotas redundantes.

A func¸a˜o objetivo do nosso Executor Simbo´lico foi enta˜o definida de maneira clara: minimizar a soma das distaˆncias percorridas entre as cidades. Para isso, modelamos a matriz de distaˆncias como um conjunto de constantes que foi multi- plicado pelas varia´veis de decisa˜o *x*[*i*][*j*]. O Z3 foi utilizado para calcular a soma total dessas distaˆncias e encontrar a rota que minimizava o valor objetivo, garantindo que a soluc¸a˜o respeitasse todas as restric¸o˜es simbo´licas impostas.

A integrac¸a˜o entre a execuc¸a˜o simbo´lica e o solver Z3 foi um fator decisivo para o sucesso da implementac¸a˜o. O Z3, com suas capacidades de manipular teorias complexas e varia´veis simbo´licas, permitiu que explora´ssemos o espac¸o de soluc¸o˜es do PCV de maneira eficiente e otimizada. Ao desenvolver esta soluc¸a˜o em Python, conseguimos aproveitar a simplicidade e flexibilidade da linguagem, permitindo que o co´digo fosse tanto acess´ıvel quanto eficaz.

Essa abordagem, centrada no uso do Executor Simbo´lico, demonstrou-se robusta para lidar com a complexidade inerente do Problema do Caixeiro Viajante, ao mesmo tempo em que utilizava as ferramentas de otimizac¸a˜o fornecidas pelo Z3 para garantir uma soluc¸a˜o via´vel e otimizada.

1. COMPLEXIDADE DA SOLUC¸ A˜ O

A complexidade do co´digo para resolver o *Problema do Caixeiro Viajante* com o Z3 e´ influenciada pelos algoritmos internos da ferramenta, especialmente o *SAT Solver* baseado no *DPLL* (Davis-Putnam-Logemann-Loveland). O nu´mero de varia´veis booleanas *x*[*i*][*j*], que representam as rotas entre as cidades, cresce quadraticamente com o nu´mero de cidades

*n*. No entanto, a complexidade do solver esta´ associada principalmente ao nu´mero de deciso˜es 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* e´ o nu´mero de cidades. Mesmo com otimizac¸o˜es, como a propagac¸a˜o de restric¸o˜es e o aprendizado de cla´usulas, a natureza exponencial do problema permanece. Portanto, ao aplicar o Z3 ao Problema do Caixeiro Viajante, a complexi- dade final do co´digo e´ *O*(2*n*) [5], [11].

1. RESULTADOS OBTIDOS

Apo´s a conclusa˜o do desenvolvimento do co´digo, real- izamos testes para verificar o comportamento da soluc¸a˜o com diferentes entradas de dados, e se ela seria capaz de encontrar as soluc¸o˜es corretas para as matrizes fornecidas. A seguir, apresentamos o desempenho da nossa soluc¸a˜o com as matrizes de exemplo:

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

 

0 100 150 200

100 0 120 80

 150 120 0 90

200 80 90 0

* **Matriz 5x5**: Aqui, a matriz representa 5 cidades com distaˆncias muito pro´ximas entre si.

 

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

Nosso desenvolvimento se baseia em uma func¸a˜o chamada tsp\_solver que e´ chamada com tendo como entrada a matriz que queremos responder, desta func¸a˜o esta˜o todas as verificac¸o˜es e execuc¸o˜es do nosso codigo este codigo pode ser totalmente visualizado a partir do repositorio do GitHub[10]

A seguir, apresentamos as soluc¸o˜es obtidas para cada uma das matrizes fornecidas e o processo de verificac¸a˜o:

# Matriz 4x4:

Nu´mero de cidades: 4

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

Ca´lculo da soma das distaˆncias: 0 → 2: 150

2 → 3: 90

3 → 1: 80

1 → 0: 100

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

Verificac¸a˜o de outros caminhos poss´ıveis: 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

* + **Matriz 20x20**: Essa matriz representa um problema com

20 cidades, onde foram analisadas distaˆncias variadas entre as cidades.

*Por motivos de formatac¸a˜o a cada duas linhas na matriz de exemplo e´ representada uma linha na matriz original.*

0 24 16 32 10 25 38 43 18 27

 

14 41 35 22 39



14 35 22 39 0 24 16 32 10 25

38 43 18 27 14





24 0 20 12 30 28 31 11 33 40

29 38 21 27 23 

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 38 | 21 | 27 | 24 | 0 20 12 |  |
| 11 | 33 | 40 | 29 |  |  |

O caminho sugerido de custo 420 e´

Soluc¸a˜o correta.

de fato o menor.

41

31





30 28



# Matriz 5x5:

Nu´mero de cidades: 5 Soluc¸a˜o encontrada:

Caminho sugerido: [0, 1, 4, 3, 2, 0]

Ca´lculo da soma das distaˆncias: 0 → 1: 2

1 → 4: 4

4 → 3: 2

3 → 2: 2

2 → 0: 3

Soma total: 2 + 4 + 2 + 2 + 3 = 13

Verificac¸a˜o de outros caminhos poss´ıveis:

16 20 0 25 12 39 18 21 34 15

36 45 19 23 14



38 45 19 23 16 20 0 25 12 39

 



18 21 34 15 36

32 12 25 0 45 17 22 30 19 28

 

24 41 15 14 40



41 15 14 40 32 12 25 0 45 17

22 30 19 28 24





10 30 12 45 0 26 38 20 48 23

 

31 39 22 14 20

39 26





|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 22 | 14 | 20 | 10 30 12 45 0 | |  |
| 20 | 48 | 23 | 31 |  |  |

38

 

25 28 39 17 26 0 29 11 19 38

10 16 35 27 23





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

38 10 45

36







16 35 27 23 25 28 39 17 26 0

29 11 19 38 10 

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 | 18 | 22 | 38 29 0 42 | |  |
| 24 | 22 | 30 | 11 |  |  |

36



|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 24 | 22 | 30 | 38 | 31 | 18 | 22 | 38 | 29 |
| 42 | 10 | 45 | 36 |  |  |  |  |  |
| 11 | 21 | 30 | 20 | 11 | 42 | 0 | 50 |  |
| 25 | 12 | 32 | 17 |  |  |  |  |  |
| 25 | 12 | 32 | 43 | 11 | 21 | 30 | 20 |  |
| 0 | 50 | 13 | 36 |  |  |  |  |  |
| 33 | 34 | 19 | 48 | 19 | 10 | 50 | 0 | 29 |
| 43 | 36 | 23 | 41 |  |  |  |  |  |
| 43 | 36 | 23 | 18 | 33 | 34 | 19 | 48 |  |
| 50 | 0 | 29 | 27 |  |  |  |  |  |
| 40 | 15 | 28 | 23 | 38 | 45 | 13 | 29 |  |
| 14 | 35 | 30 | 19 |  |  |  |  |  |

0

43 13



36



36

11

42



18



27

27 19



10



27

0

20



20

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 14 | 35 | 30 | 27 40 15 | 28 23 38 |
| 13 | 29 | 0 | 20 |  |

45

Apo´s mais de 6 horas de execuc¸a˜o, o co´digo na˜o retornou uma soluc¸a˜o via´vel dentro do tempo esperado. Esse resultado esta´ alinhado com a natureza do problema do

5 + 2 + 3 + 2 + 3 = 15

Caixeiro Viajante, que e´

NP-dif´ıcil, e cujo tempo de

5 + 2 + 2 + 2 + 2 = 13

O caminho sugerido de custo 13 e´ Soluc¸a˜o correta.

de fato o menor.

execuc¸a˜o aumenta exponencialmente com o nu´mero de cidades. Para instaˆncias com 10 ou mais cidades, e´ es- perado que o solver leve mais tempo para encontrar uma soluc¸a˜o ou mesmo na˜o consiga em um tempo razoa´vel. Esse comportamento na˜o indica um erro no co´digo, mas

sim uma poss´ıvel limitac¸a˜o do desenvolvimento, visto que o aumento no nu´mero de cidades impacta negativamente

o tempo de execuc¸a˜o. Logo, na˜o foi poss´ıvel avaliar com detalhes o resultado dessa matriz.

1. ANA´ LISE DOS RESULTADOS

Os testes realizados com diferentes matrizes de distaˆncias evidenciam a efica´cia e robustez do algoritmo. Para cada matriz fornecida, o algoritmo foi capaz de encontrar o caminho com o menor custo total, respeitando as restric¸o˜es impostas pelo Problema do Caixeiro Viajante (TSP). A seguir, detal- hamos os principais aspectos observados durante a execuc¸a˜o dos testes:

1. **Comportamento em Matriz 4x4 (Distaˆncias Longas)**: Na matriz com 4 cidades e distaˆncias consideravelmente longas, o algoritmo foi capaz de encontrar a soluc¸a˜o o´tima com um custo total de 420. Durante a verificac¸a˜o de outros caminhos poss´ıveis, o algoritmo garantiu que na˜o havia rota mais curta. Esse resultado demonstra a eficieˆncia em cena´rios onde ha´ uma grande disparidade entre as distaˆncias, como foi o caso dos valores que variavam entre 80 e 200. Mesmo com essa variac¸a˜o, a soluc¸a˜o sugerida foi correta, e o tempo de execuc¸a˜o per- maneceu dentro dos limites aceita´veis para esse tamanho de matriz.

A ana´lise 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 explorac¸a˜o do espac¸o de busca, minimizando o custo total sem comprometer a correc¸a˜o da soluc¸a˜o.

1. **Comportamento em Matriz 5x5 (Cidades Pro´ximas)**: Na matriz 5x5, onde as distaˆncias entre as cidades sa˜o menores e mais pro´ximas, o algoritmo tambe´m foi eficiente em encontrar a soluc¸a˜o o´tima com custo total de 13. Este cena´rio e´ interessante pois as distaˆncias entre as cidades sa˜o 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 pe- quenos valores e pequenas diferenc¸as entre as distaˆncias. Durante a verificac¸a˜o de outros caminhos poss´ıveis, foi observado que va´rios caminhos tinham custos similares, mas nenhum deles foi inferior ao caminho sugerido pelo algoritmo, validando mais uma vez a precisa˜o da soluc¸a˜o obtida. O desempenho do algoritmo se manteve esta´vel, mesmo com um nu´mero maior de cidades e distaˆncias relativamente pequenas, reforc¸ando sua capacidade de resolver problemas de TSP em diferentes contextos.
2. **Comportamento em Matriz 20x20 (Nu´mero elevado de cidades)**: Para a matriz de 20x20, o algoritmo na˜o conseguiu obter uma soluc¸a˜o adequada mesmo apo´s 6 horas de processamento cont´ınuo. Vale ressaltar que o

demande mais tempo ou, em alguns casos, na˜o consiga encontrar uma soluc¸a˜o dentro de um per´ıodo aceita´vel. Isso reflete a complexidade intr´ınseca do problema e na˜o representa uma falha no co´digo. Os testes realizados para casos com ate´ 10 cidades tiveram sucesso, sugerindo que a dificuldade enfrentada com instaˆncias maiores pode estar relacionada a limitac¸o˜es na implementac¸a˜o ao lidar com entradas de maior escala.

1. **Validac¸a˜o da Precisa˜o dos Resultados**:A validac¸a˜o dos resultados obtidos foi realizada por meio da comparac¸a˜o com todos os caminhos poss´ıveis gerados pelas permutac¸o˜es das cidades. Essa etapa foi importante para assegurar que o caminho sugerido fosse de fato o mais curto. A ana´lise mostrou que, em ambos os casos, o algoritmo foi capaz de encontrar a rota com o menor custo total. A exibic¸a˜o dos outros caminhos poss´ıveis e seus respectivos custos confirma a precisa˜o da soluc¸a˜o proposta.

Ale´m disso, o fato de que o algoritmo foi testado com diferentes configurac¸o˜es de matrizes, tanto com distaˆncias longas quanto curtas, demonstra sua versa- tilidade em resolver problemas de TSP em diversos contextos, sem comprometer a precisa˜o ou eficieˆncia.

1. JUSTIFICATIVA PARA O USO DA EXECUC¸ A˜ O SIMBO´ LICA

A execuc¸a˜o simbo´lica se destaca como uma alternativa eficiente aos me´todos tradicionais de teste de software, espe- cialmente em contextos onde ha´ uma vasta gama de poss´ıveis entradas e ramificac¸o˜es condicionais. Em vez de depender exclusivamente de um conjunto limitado de casos de teste concretos, a execuc¸a˜o simbo´lica utiliza varia´veis simbo´licas para representar as poss´ıveis entradas, permitindo uma ana´lise abrangente de todas as execuc¸o˜es potenciais do programa. Ao simular todas as poss´ıveis ramificac¸o˜es e condic¸o˜es com base em fo´rmulas simbo´licas, esse me´todo oferece uma cobertura muito maior, garantindo que uma maior parte do espac¸o de busca seja explorada[3]. Ale´m disso, a execuc¸a˜o simbo´lica permite testar programas complexos de forma mais eficiente, ao resolver automaticamente condic¸o˜es lo´gicas e matema´ticas associadas ao comportamento do programa. Isso e´ feito atrave´s do uso de solvers SMT, como o Z3, que verifica formalmente se as condic¸o˜es geradas durante a execuc¸a˜o simbo´lica sa˜o satisfeitas. Essa abordagem permite na˜o apenas identificar poss´ıveis erros no co´digo, mas tambe´m garantir a correc¸a˜o formal do programa de forma automa´tica, sem a necessidade de intervenc¸a˜o manual, como destacam os estudos sobre o Z3[5]. Portanto, ao oferecer uma cobertura mais ampla e integrar me´todos formais de verificac¸a˜o, a execuc¸a˜o simbo´lica supera as limitac¸o˜es dos me´todos tradicionais de teste, sendo especialmente eficaz na explorac¸a˜o de mu´ltiplos caminhos de execuc¸a˜o e na verificac¸a˜o automa´tica de propriedades complexas.

Problema do Caixeiro Viajante pertence a` classe NP-

1. CONCLUSA˜ O

dif´ıcil, o que significa que o tempo necessa´rio para

resolver o problema cresce exponencialmente a`

me-

Este trabalho apresentou o desenvolvimento de um execu-

dida que o nu´mero de cidades aumenta. Assim, para instaˆncias com 10 cidades ou mais, e´ natural que o solver

tor simbo´lico para o Problema do Caixeiro Viajante (PCV) utilizando a linguagem Python e a biblioteca Z3. A execuc¸a˜o

simbo´lica, combinada com o uso do solucionador SMT, demonstrou ser uma abordagem eficaz para explorar mu´ltiplas soluc¸o˜es poss´ıveis e verificar restric¸o˜es lo´gicas associadas ao problema. O Z3 provou ser uma ferramenta eficiente para lidar com as restric¸o˜es impostas pelo PCV, garantindo que todas as cidades fossem visitadas uma u´nica vez e evitando a formac¸a˜o de subciclos, enquanto buscava a rota de menor custo total.

Os testes realizados em matrizes de distaˆncias com difer- entes configurac¸o˜es confirmaram a precisa˜o e robustez da soluc¸a˜o desenvolvida. Tanto em cena´rios com distaˆncias lon- gas quanto em cena´rios com distaˆncias curtas, o algoritmo

foi capaz de encontrar a soluc¸a˜o versatilidade e eficieˆncia.

o´tima, reforc¸ando a sua

Por fim, a integrac¸a˜o entre a execuc¸a˜o simbo´lica e o solver Z3, aliado a` simplicidade do Python, mostrou-se uma abordagem poderosa para resolver problemas de otimizac¸a˜o combinato´ria complexos como o PCV. Este trabalho abre caminho para novas explorac¸o˜es no uso de ferramentas SMT e te´cnicas de execuc¸a˜o simbo´lica na soluc¸a˜o de outros problemas de otimizac¸a˜o.

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 aplicac¸o˜es*. 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 *In- ternational 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*, Dispon´ıvel 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, *RosialdoVeni- cius FinalProject AA RR 2024*, Dispon´ıvel 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.