Algoritmo Genético e o Problema Caixeiro Viajante

Uma Reprodução Metodológica

Yle Severino Carvalho
Departamento de Engenharia Elétrica
Universidade Federal do Triângulo Mineiro (UFTM)
Uberaba, Brasil
yle.severino@pm.me

Elder Vicente de Paulo Sobrinho
Departamento de Engenharia Elétrica
Universidade Federal do Triângulo Mineiro (UFTM)
Uberaba, Brasil
elder.sobrinho@uftm.edu.br

***Abstract*—This article seeks to solve the multiple traveling salesman problem (mTSP) using genetic algorithms (GA). The mTSP has many real-life applications, so finding ways to solve this problem is very useful, however it is an NP-hard problem, in other words, exact algorithms tend to be expensive to solve. This article seeks to solve the mTSP using GA, using the sequential constructive crossover (SCX) operator. Several adapted instances of the TSPLIB were used as tests and the results were compared with an article that developed the same approach.**

***Index Terms*—Genetic Algorithms, Travelling Salesman Problem, Multiple Travelling Salesman Problem, sequential constructive crossover**

I. Introdução

Travelling salesman problem (TSP), também conhecido como problema do caixeiro viajante, apresenta a seguinte incógnita: "dado vários pontos em um mapa com diferentes distâncias, qual é a rota de menor distância na qual sejam visitados todos os pontos, com retorno ao ponto de origem"? Este é um problema *NP-hard* (*non-deterministic polynomial-time hard*) [1], portanto ainda não existe um algoritmo de busca que encontre a melhor resposta em um tempo de execução polinomial. O problema do caixeiro viajante é aplicado em muitos campos, sendo: roteamento de veículos, fabricação de microchips, roteamento de pacotes, perfuração em placas de circuito impresso [2].

O problema que será desenvolvido nesse artigo é o problema de múltiplos caixeiros viajantes (*multi travel salesman problem - mTSP*), ele é uma variante do conhecido problema do caixeiro viajante (*traveling salesman problem TSP*). O mTSP é um problema de roteamento que atribui M caixeiros a N pontos (endereços), e cada pontos deve ser visitado por um caixeiro. Em especial, com intuito de obter o menor custo total. Este problema também possui muitas aplicações reais, como: distribuição de materiais de emergência [3], roteamento de veículos [4] e programação de sistemas de laminação a quente [5].

O objetivo do presente trabalho é replicar a metodologia implementada por Al-Furhud and Hussain [6] na solução do mTSP. Para replicar sua metodologia primeiramente foi necessário obter as instâncias de teste utilizadas em seu artigo. As situações de teste foram obtidas de um conjunto de exemplos da biblioteca TSPLIB [7].

No artigo do Al-Furhud and Hussain [6] é apresentado uma solução ao problema mTSP utilizando algoritmo genético em conjunto com vários operadores de cruzamento. Contudo, apesar do presente trabalho ter como foco principal a replicação da metodologia apresentada por Al-Furhud and Hussain [6], não será desenvolvido todos os operadores de cruzamento descritos no artigo [6]. Por não haver tempo necessário para desenvolver todos os operadores de cruzamento, foi decidido implementar somente o operador *Constructive Crossover Operator* (SCX), visto que o operado SCX obteve os melhores resultados em vários artigos relacionados [8, 6, 9]. Portanto, no presente trabalho será replicado apenas o operador de cruzamento (SCX) [6]. Outra diferença do presente trabalho em relação ao artigo [6] está na linguagem de programação. O trabalho original utilizou C++ e, neste, o algoritmo foi implementado na linguagem de programação denominada Julia [10]. Houve diversos motivos que levaram à utilização da linguagem Julia, dentre eles podemos citar o fato dela ter uma sintaxe parecida com Python, linguagem com a qual o autor possui maior familiaridade; ser gratuita e *open source*; além de ser indicada para ambientes que necessitam de alta performance [11]. Todavia, por ser uma linguagem diferente da utilizada por Al-Furhud and Hussain [6], espera-se encontrar resultados semelhantes, porém com variações, visto que os algoritmos genéticos são muito sensíveis à aleatoriedade do sistema [12]. Isso se deve em especial à diferença de implementação dos algoritmos de geração de números aleatórios: Julia utiliza a estratégia *Mersenne Twister (MT)* [13] e C++ o método *Linear congruential generator (LCG)* [14, 15].

Em alguns testes, os resultados obtidos foram como esperado. Em outras palavras, são próximos aos do artigo replicado, porém com desvio padrão diferente. Por outro lado, alguns experimentos apresentaram resultados diferentes daqueles vistos no artigo original [6]. Provavelmente isso se deve ao fato do autor ter omitido alguns detalhes de implementação/teste.

Este artigo está organizado na seguinte forma: na seção II são apresentadas as definições do problema de múltiplos caixeiros viajante e do algoritmo genético (GA). Já na seção III é exemplificado o passo a passo do funcionamento do algoritmo desenvolvido. Na seção IV é apresentado o ambiente de teste e os resultados obtidos. Na seção V detalha-se as considerações finais, apresentando as dificuldades enfrentadas

na replicação do trabalho [6] e os resultados obtidos. Por fim na seção VI, são apresentados os artigos relacionados ao tema.

II. Background

Segundo Bektas [9] o problema de múltiplos caixeiros viajantes *(mTSP)* é uma generalização do problema do caixeiro viajante *(TSP)* em que mais de um caixeiro é permitido. Ele pode ser definido como um conjunto de pontos, onde temos uma origem na qual os m caixeiros iniciam suas rotas e uma métrica de custo (geralmente, representa a distância ou tempo para percorrer os pontos). O objetivo do *mTSP* é determinar um conjunto de rotas para m caixeiros de forma a minimizar o custo total das rotas. Os requisitos necessários para solucionar o problema são: todos os caixeiros devem começar e terminar no ponto de origem, sendo que cada ponto deve ser visitado exatamente uma vez por apenas um caixeiro. Já que esse problema é classificado como *NP-hard* [9], é interessante utilizar algoritmos não exatos (ex. GA) para encontrar uma solução

Algoritmo genético (GA) é uma meta-heurística inspirada no processo de seleção natural que pertence à classe de algoritmos evolutivos [16]. GAs tendem a gerar soluções ótimas para problemas de otimização e busca [16], como TSP [17] e mTSP [18]. Então, o primeiro passo para implementar um algoritmo genético consiste em especificar os seguintes aspectos: o que representará os genes; quem serão os cromossomos; qual será o critério de seleção e como o cruzamento e a mutação irá ocorrer [19]. Deste modo, nas próximas subseções apresentamos cada um destes itens.

A. Genes

Em 1866, Mendel identificou que a natureza armazena informações dos indivíduos no material genético [20], sendo essas responsáveis por definir o fenótipo do indivíduo. Então, essas informações genéticas determinam aspectos como aparência e habilidades de um dado indivíduo. Partindo deste contexto, o conceito de genes usado em algoritmos genéticos apresenta semelhanças em relação às descobertas de Mendel, uma vez que suas combinações influenciam nas características do indivíduo.

Os genes para o problema de mTSP representam os pontos que constituem as rotas dos caixeiros. A combinação dos genes dos caixeiros gera vários resultados diferentes nos custos das rotas, essa combinação dá origem ao que é denominado em algoritmos genéticos como cromossomos.

B. Cromossomo e População

Em algoritmo genético, um conjunto de soluções candidatas é denominado população, já uma solução candidata isolada é comumente denominada de cromossomo [19]. Um cromossomo para mTSP pode ser representado de várias maneiras diferentes, a saber: um cromossomo *(one-chromosome)* [21], multi-cromossomos *(multi-chromosome)* [22], dois cromossomos *(two-chromosome)* [21] e cromossomo de duas partes *(two-part chromosome)* [23]. Geralmente o que define a escolha da representação cromossômica é o operador de cruzamento que será utilizado no GA. Todavia as representações têm impacto na variabilidade das soluções candidatas [22], dado que algumas representações geram cromossomos diferentes para a mesma solução candidata.

Para facilitar a visualização dos leitores sobre os cromossomos foi criado na Figura 1 um exemplo simples de mTSP com solução. É um exemplo que possui 2 caixeiros e 8 pontos para ser percorrido, nele é representando a rota do caixeiro 1 com uma linha contínua e do caixeiro 2 com linha pontilhada.

O método de representação de múltiplos cromossomos envolve representar cada caixeiro separadamente dentro do cromossomo, assim como está representado na Figura 2. Essa representação é a mais intuitiva para mTSP, pois cada caixeiro fica "separado" do outro. Esse método de representação garante o menor número de soluções candidatas duplicadas dentre as outras citadas anteriormente [24].

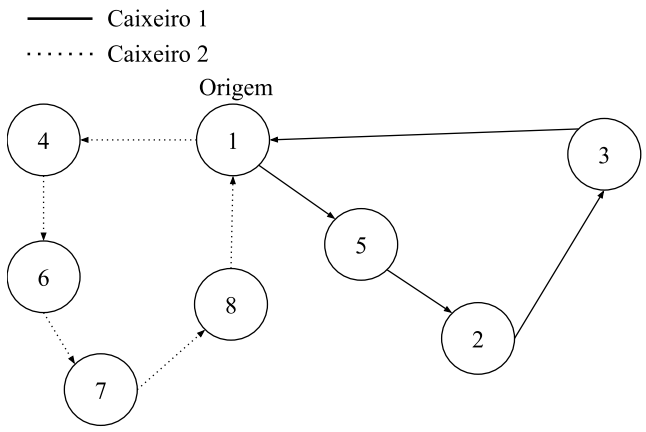

Figura 1. Caso de exemplo – problema mTSP

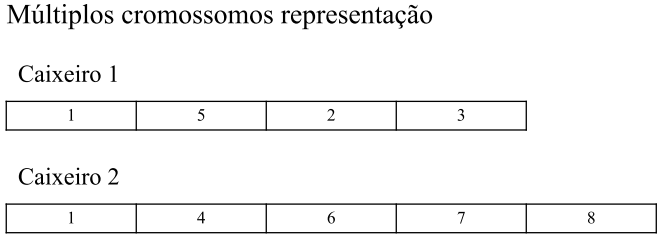

Figura 2. Representação cromossômica (baseado na Figura 1)

O método de representação com único cromossomo tradicional envolve separar os caixeiros dentro de um único vetor de dados, com tamanho $N + M - 1$ (onde N = numero de pontos e M = numero de caixeiros), no caso do exemplo $8+2-1 = 9$. Este método insere valores negativos decrescentes entre as rotas dos caixeiros para que assim seja facilmente identificado e não confundido com os pontos das rotas, fazendo com que as rotas de cada caixeiro sejam bem delimitadas no cromossomo. Este tipo de arranjo está representado na Figura 3.

Neste artigo será utilizado uma variante do método de representação com único cromossomo, ela é denominada de

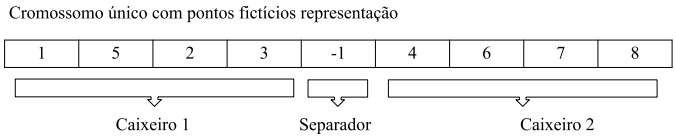

Figura 3. Abordagem com único cromossomo (baseado na Figura 1)

único cromossomo com pontos artificiais *(one-chromosome with artificial depots – OCWAD)*, a qual foi utilizada no trabalho original [6]. Essa técnica é bastante similar a representação cromossômica única tradicional, visto que a única diferença está na forma de separar os caixeiros, dado que ao invés de utilizar números negativos, é criado um ponto artificial de numeração igual a soma de todos os pontos mais um $(N+1)$ como mostrado na Figura 4. Esse ponto artificial adicionado ao problema possuirá todas as características do ponto de origem, portanto o custo do caixeiro sair desse ponto fictício e ir para qualquer outro ponto será exatamente igual ao custo se ele estivesse saindo da origem. Para ilustrar melhor será realizado um exemplo na próxima subseção.

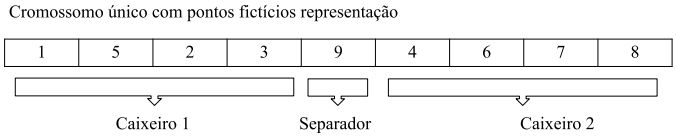

Figura 4. Abordagem OCWAD (baseado na Figura 1)

C. Seleção (Fitness)

Durante cada geração sucessiva, uma parte da população existente é selecionada para gerar uma nova geração. Neste processo, um pedaço da população original é selecionado de acordo com seu desempenho (*Fitness Function*). Em particular, no caso do mTSP, as soluções candidatas que possuírem os menores custos/distâncias são selecionadas, ou seja, os cromossomos de menor custo são usados na nova geração [19].

Para ilustrar como se realiza o cálculo de custo total de um cromossomo, vamos usar como referência o cromossomo da Figura 4, a saber: $\{1, 5, 2, 3, 9, 4, 6, 7, 8, 1\}$. Neste caso, utilizamos a matriz de adjacência da Tabela I para descrever os custos entre pares de cromossomos. Neste cromossomo, existem nove genes $(8 + 2 - 1)$ incluindo o ponto artificial 9 (representado na Figura 4 como *Separador)*. Então este representa a seguinte rota: $1 \rightarrow 5 \rightarrow 2 \rightarrow 3 \rightarrow 9 \rightarrow 4 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 1$ extraída da Figura 1. Isso significa que o primeiro caixeiro visitou os pontos 1, 5, 2 e 3 sequencialmente. Além disso, o segundo caixeiro visitou os pontos 1, 4, 6, 7 e 8 sequencialmente. Seguindo, para calcular o custo total do cromossomo basta buscar o custo na matriz de adjacência (Tabela I), usando a linha o ponto de origem e a coluna o ponto de destino. Por exemplo, para encontrar o custo de ir do ponto 1 para o ponto 5, basta olhar o valor da linha 1 coluna 5, que é 86. Portanto, repete-se este procedimento para todos os pontos no cromossomo e encontra-se o custo, a saber: $86 + 15 + 32 + 52 + 20 + 69 + 94 + 93 + 32 = 493$.

Um detalhe importante é a diagonal da matriz, nela está escrito os valores 999. A diagonal da matriz de adjacência para o problema mTSP pode ser desconsiderada, visto que não é útil para o problema os custos de pontos conectando com eles mesmos.

Outro detalhe importante sobre a matriz de adjacência está na coluna 9 e linha 9, pois é possível notar que os valores, tanto nesta linha quanto na coluna são iguais ao da linha 1 e coluna 1 respectivamente, isso se deve ao fato do ponto fictício 9 representar o ponto de origem do problema que comumente é o ponto 1.

Tabela I
MATRIZ DE ADJACÊNCIA COM 9 PONTOS

Ponto	1	2	3	4	5	6	7	8	9
1	999	12	54	20	86	88	93	25	999
2	31	999	32	29	47	91	88	71	31
3	52	31	999	63	65	55	80	61	52
4	61	56	62	999	4	69	30	88	61
5	89	15	59	84	999	8	13	17	89
6	20	23	95	83	29	999	94	9	20
7	14	20	44	32	79	40	999	93	14
8	32	32	31	11	51	19	69	999	32
9	999	12	54	20	86	88	93	25	999

D. Cruzamento construtivo sequencial[1]

O cruzamento é um operador em GA que busca selecionar os melhores genes de 2 cromossomos e gerar um novo cromossomo filho com os melhores genes. Existem vários operadores de cruzamento implementados para o mTSP, alguns deles são *ordered crossover* (OX) [25], *cycle crossover* (CX) [26], *sequential constructive crossover* (SCX) [6], sendo que cada um se diferencia pelo seu critério de seleção de genes que serão repassados aos filhos na nova geração. O operador de cruzamento que foi desenvolvido neste artigo é o *SCX*, o qual também foi usado em [6]. Segundo Al-Furhud and Hussain [6], a principal característica do operador SCX é examinar sequencialmente os cromossomos pais e comparar os custos de seus pontos em relação ao ponto de origem do cromossomo filho. Caso algum ponto dos cromossomos pais já tenha sido inseridos no filho, é inserido no filho o ponto de cromossomo pai que ainda não foi inserido, caso os pontos de ambos os pais já tenham sido inseridos no filho é gerado um novo ponto aleatoriamente (conforme demonstrado na Subseção A da metodologia).

E. Mutação

Para que os algoritmos genéticos funcionem adequadamente é necessário que exista uma boa variabilidade genética [19]. Por esse motivo, é necessário implementar a operação de mutação. O objetivo é aumentar a diversidade dos cromossomos na população, ou seja, aumentar o conjunto de soluções possíveis. A mutação implementada neste trabalho

[1]Sequential Constructive Crossover Operator - SCX

foi a mesma utilizada no artigo que está sendo replicado, ela é a troca aleatória dos pontos.

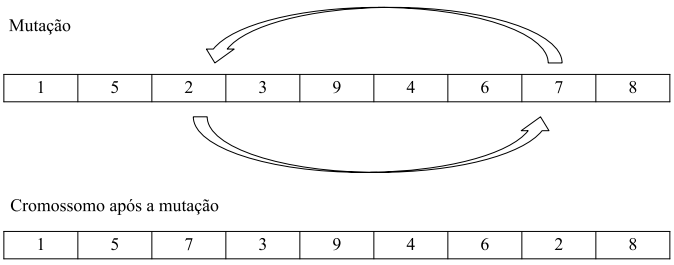

Figura 5. Representação de uma mutação

Na Figura 5 é exemplificado o funcionamento da mutação. Nele, troca-se de lugar o ponto 2 com o ponto 7 de modo aleatório apenas para exemplificar o funcionamento do método de cruzamento. O algoritmo de mutação implementado seleciona todos os indivíduos com uma probabilidade pré-determinada, geralmente definida em testes com o algoritmo. Neste artigo será utilizado a mesma probabilidade de mutação do artigo original [6], a saber 10%.

Como apresentado nas subseções anteriores e em resumo, algoritmos genéticos são utilizados para resolver problemas de otimização onde algoritmos exatos possuem um custo computacional muito alto como é descrito em [16]. Em muitos problemas, GAs tendem a convergir para ótimos locais ou mesmo pontos arbitrários, em vez do ótimo global do problema. A probabilidade de isso ocorrer depende do critério de seleção do seu problema e das condições iniciais como a probabilidade de mutação e cruzamento. Contudo este problema pode ser aliviado aumentando a taxa de mutação ou usando técnicas de seleção que mantêm uma população diversificada de soluções, ou até mesmo modificando o tipo de cromossomo implementado.

III. Metodologia

Na metodologia será exemplificado a operação de cruzamento com o operador SCX, e também será mostrado o algoritmo implementado na forma de pseudocódigo, além disso também será apresentado os parâmetros iniciais que foram definidos para a execução do algoritmo.

A. Exemplo de cruzamento SCX

No cruzamento dos cromossomos com operador SCX, são comparados ponto a ponto de cada cromossomo sequencialmente e os que resultarem em um custo menor são inseridos no cromossomo filho. Para exemplificar melhor será realizado o cruzamento entre os dois cromossomos representados na Figura 6. Considerando a matriz de adjacência de 9 pontos (Tabela I).

Primeiramente, são comparados os dois primeiros pontos logo após a origem (1), ou seja, nos pontos 5 e 3 da Tabela I, é possível notar que (1 $\rightarrow$ 5) = 86 e (1 $\rightarrow$ 3) = 54, logo (1 $\rightarrow$ 3) $<$ (1 $\rightarrow$ 5), portanto o ponto 3 é adicionado no filho, tornando-o assim (1, 3).

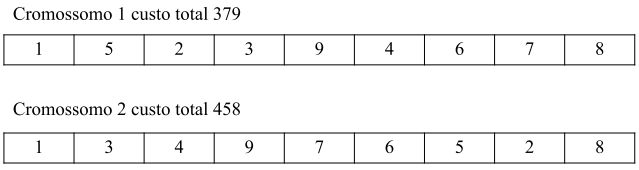

Figura 6. Cromossomos para exemplificar o cruzamento SCX

Depois é comparado o ponto 2 do cromossomo 1 com o ponto 4 do cromossomo 2, como (3 $\rightarrow$ 2) = 31 e (3 $\rightarrow$ 4) = 63, assim (3 $\rightarrow$ 2) $<$ (3 $\rightarrow$ 4), portanto é adicionado o ponto 2 no filho, tornando-o assim (1, 3, 2)

Depois é comparado o ponto 3 do cromossomo 1 com o ponto 9 do cromossomo 2, como (2 $\rightarrow$ 3) = 32 e (2 $\rightarrow$ 9) = 31, assim (2 $\rightarrow$ 9) $<$ (2 $\rightarrow$ 3), logo é adicionado o ponto 9 no filho, tornando-o assim (1, 3, 2, 9)

Depois é comparado o ponto 9 do cromossomo 1 com o ponto 7 do cromossomo 2, como o ponto 9 já foi inserido na solução filha, logo é inserido o ponto 7, tornando-o assim (1, 3, 2, 9, 7)

Depois é comparado o ponto 4 do cromossomo 1 com o ponto 6 do cromossomo 2, como (7 $\rightarrow$ 4) = 32 e (7 $\rightarrow$ 6) = 40, assim (7 $\rightarrow$ 4) $<$ (7 $\rightarrow$ 6), logo é adicionado o ponto 4 no filho, tornando-o assim (1, 3, 2, 9, 7, 4)

Depois é comparado o ponto 6 do cromossomo 1 com o ponto 5 do cromossomo 2, como (4 $\rightarrow$ 6) = 69 e (4 $\rightarrow$ 5) = 4, assim (4 $\rightarrow$ 5) $<$ (4 $\rightarrow$ 6), logo é adicionado o ponto 5 no filho, tornando-o assim (1, 3, 2, 9, 7, 4, 5)

Depois é comparado o ponto 7 do cromossomo 1 com o ponto 2 do cromossomo 2, como é possível notar esses 2 pontos já estão presentes no filho, portanto não porem ser inserido, logo é gerado um ponto aleatório dentro os restantes para ser inserido nesse local, neste caso será inserido o ponto 6 tornando-o assim (1, 3, 2, 9, 7, 4, 5, 6)

Depois é comparado o ponto 8 do cromossomo 1 com o ponto 8 do cromossomo 2, assim é inserido o ponto 8 no filho tornando-o assim (1, 3, 2, 9, 7, 4, 5, 6, 8)

Como todos os ponto foram visitados, para terminar a solução é necessário retornar à origem o ponto 1 e adicionado ao final, logo o resultado do filho é (1, 3, 2, 9, 7, 4, 5, 6, 8, 1) e seu custo total resulta em (54 + 31 + 31 + 93 + 32 + 8 +32 = 281).

B. Estrutura do GA usado no trabalho

O algoritmo genético replicado possui um funcionamento muito parecido com um GA normal, os seja, é um GA que possui os operadores comuns, a saber cruzamento, seleção e mutação. O funcionamento do algoritmo genético pode ser resumido em: primeiro é criado uma população aleatória de 50 cromossomos; logo após será feito o cruzamento com os 50 cromossomos, para realizar o cruzamento primeiro é selecionado um cromossomo e depois é aleatoriamente selecionado outro cromossomo, por exemplo, o cruzamento começa no cromossomo 1, depois é aleatoriamente selecionado o cromossomo 3, assim é realizado o cruzamento e adicionado

na população, depois é selecionado o cromossomo 2 e aleatoriamente selecionando o cromossomo 42 para o cruzamento, assim é repetido esse processo até que seja selecionado os 50 cromossomos para realizar o cruzamento; depois cada cromossomo é testado na probabilidade de sofrer mutação, a mutação não alterar o cromossomo e sim gera um novo cromossomo que é inserido na população e por último é realizado a seleção, nessa etapa é filtrado a população, ou seja, é retirado os cromossomo para que a população volte ao número de 50 cromossomos. Esse algoritmo foi descrito logo a seguir em pseudo código.

Uma vez que os operadores do GA já foram definidos, a próxima etapa antes de executar o algoritmo consiste em definir os parâmetros de inicialização da GA, esses parâmetros são: tamanho da população; probabilidade de cruzamento; probabilidade de mutação e número limite de geração. De acordo com [19] os melhores valores para esses parâmetros varia de acordo com o problema e a forma que foi implementado o algoritmo genético, logo a forma recomendada de definir esses parâmetros é com base em testes. Contudo como o objetivo desse artigo e de replicar a metodologia implementada por Al-Furhud and Hussain [6], os valores utilizados para esses parâmetros serão os mesmo do artigo replicado, e podem ser facilmente visualizados na Tabela II.

Algoritmo 1: Algoritmo genético

```
Inicializa a população de tamanho 50 aleatoriamente;
Geração = 0;
repita
    Geração += 1;
    Executar o cruzamento com a probabilidade 100%;
    Realizar mutação com probabilidade 10%;
    Selecionar 50 cromossomos para próxima geração;
até Geração = 2000;
```

C. Experimento

O algoritmo descrito foi escrito na linguagem de programação denominada Julia, o código fonte desse artigo pode ser encontrado no Github[2]. Para execução experimental, utilizou-se um computador com: processador $i5$ (modelo 3570k), 8GB de RAM, sistema operacional Fedora (versão 34) e instalação da linguagem Julia (versão 1.6). O algoritmo genético foi executado 30 vezes para cada instância de teste com o número m de caixeiros e coletado o valor da solução obtida, depois de coletados os valores foi gerado a Tabela III.

Um problema interessante enfrentado pelos pesquisadores para mTSP pontuado por [24] é que não existe um conjunto de testes específicos para o problema mTSP, o que é realizado na literatura é a conversão de testes criados de TSP para mTSP [6, 24, 8, 27]. Por esse motivo, as instâncias de teste são descritas como adaptadas da TSPLIB. As instâncias selecionadas para serem adaptadas da TSPLIB foram eil51, kroA100 e ch150, que estão representadas na Tabela III como MTSP-51, MTSP-100 e MTSP-150. Os números escritos nas descrições dos testes representam os números de pontos de cada instância.

[2]https://github.com/yleseverino/MTPSP-GA

Tabela II
PARÂMETROS UTILIZADOS NOS TESTES

Parâmetros	Valores
População	50
Probabilidade de cruzamento	100%
Probabilidade de mutação	10%
Número de gerações	2000
Número execuções de cada instância	30

IV. RESULTADOS

Tabela III
RESULTADOS OBTIDOS

		Resultados obtidos			**Artigo replicado [6]**		
Instância	**m**	**Best**	**Med.**	**Std**	**Best**	**Med.**	**Std.**
MTSP-51	3	454	523	35	456	480	9
	5	527	572	31	488	496	3
	10	718	752	19	621	631	6
MTSP-100	3	33452	39360	2691	25107	25746	477
	5	35225	42384	3506	26317	27250	545
	10	46047	51062	3472	31149	32422	565
	20	69095	70626	1358	44543	45989	814
MTSP-150	3	10343	12523	838	33404	34974	930
	5	11761	13694	969	34531	35844	680
	10	15329	16299	595	36514	38344	643
	20	14969	16474	572	48354	49450	616
	30	14505	1616	767	63624	65812	1389

Os resultados obtidos podem ser encontrados na Tabela III. A tabela está separada entre os resultados obtidos com esse estudo e os resultados obtidos pelo artigo replicado [6]. Na coluna denominada ”Best” são apresentados os melhores resultados obtidos em determinada instância de teste com o número de caixeiros. Já as colunas ”Med.” e ”Std.” apresentam os valores de média e seu correspondente desvio padrão, com os 30 resultados durante o experimento. Todavia é possível notar que quase todos os resultados obtidos do algoritmo implementado em Julia tiveram um desvio padrão maior. Isto pode ser explicado devido as linguagens C++ e Julia possuirem um algoritmo de geração de números aleatórios diferentes, sendo eles o *Linear congruential generator (LCG)* [15] e *Mersenne Twister (MT)* [13] respectivamente. Os algoritmos de LCGs, diferente do MT, tendem a não distribuir uniformemente as respostas, principalmente em intervalos curtos como escolher algum ponto em um vetor de dados [28], que é uma operação muito utilizado em GAs. Por isso, o desvio padrão das soluções obtidas com a linguagem C++ tende a ser menor, consequentemente isso gera menor variabilidade genética. Esse pode ser o motivo da variação dos resultados na instancia de teste MTSP-51.

Entretanto as instâncias de testes MTSP-100 e MTSP-150, tiveram resultados muito distintos do artigo original. Uma

possível explicação para essa discrepância se deve ao fato do autor não especificar certos detalhes, como qual instância de teste da TSPLIB for convertida e usada nos experimentos. Destacamos que, na TSPLIB, há diversas instâncias de 100 e 150 pontos. Caso o autor tivesse utilizado a mesma notação do artigo [24] seria mais fácil identificar exatamente o teste que foi rodado, pois neste trabalho é utilizado a mesma notação da TSPLIB para representar a instância de teste.

V. Considerações Finais

Ao replicar o algoritmo proposto por Al-Furhud and Hussain [6] enfrentou-se duas grandes dificuldades. A principal delas foi de não ter acesso ao código fonte do artigo, o que gerou algumas dificuldades pontuais na implementação do algoritmo, visto que não foi possível confirmar determinados aspectos. Por exemplo, se o cruzamento dos cromossomos era para ser feito somente na população gerada, ou se os cromossomos filhos gerados durante o processo poderiam também ser inseridos na operação. A segunda foi não saber exatamente qual instância de teste de 100 e 150 pontos foi utilizada, algo que poderia ser resolvido também com o fornecimento do nome exato da instância adaptada. Todavia foi possível comprovar o funcionamento do algoritmo, que além disso se mostrou muito rápido implementado em Julia.

VI. Trabalhos Relacionados

Nesta seção será discutido artigos que possuem uma proposta semelhante a este artigo, ou seja, pesquisas que possuem o objetivo de solucionar o problema mTSP através do uso de algoritmos genéticos.

No artigo do Al-Furhud and Hussain [6], ele modela o seu GA com o cromossomo único com pontos artificiais, dado que com esse tipo de cromossomo é possível utilizar operadores de cruzamento de TSP em mTSP, o que é realizado em seu trabalho. Al-Furhud and Hussain [6] comparam no seu artigo os operadores *sequential constructive crossover operator (SCX), adaptive sequential constructive crossover operator (ASCX), reverse greedy sequential constructive crossover operator (GSCX)* e *comprehesive sequential constructive crossover operator (CSCX)*, segundo esse estudo, o operador de cruzamento que desempenhou o melhor desempenho foi *CSCX*, pois para na maioria dos testes realizados ele conseguiu os melhores resultados.

Yuan et al. [27] modela o cromossomo em duas partes, o que garante menos duplicidades de soluções candidatas. Neste artigo foi realizado um comparativo de 4 operadores de cruzamento distintos, a saber: *OX + A, PMX + A e CX + A e TCX*. Foi realizado o experimento em 3 instâncias de testes convertidas da TSPLIB com até 5 números diferentes de caixeiros, com o algoritmo de mutação de troca de genes aleatórios com probabilidade definida de 1% e com a probabilidade de cruzamento de 85%. Conforme relatado, boa parte dos operadores de cruzamento existentes para codificação de cromossomos de duas partes limitam a variabilidade das GAs, entretanto isso não ocorreu com o operador TCX, pois ele conseguiu gerar cromossomo mais diferentes entre si e portanto chegou em soluções melhores.

Shuai et al. [24] também utiliza cromossomo em duas partes para reduzir a redundância das soluções candidatas. Neste artigo, o autor ressalta que não existe instância de exemplo para realizar os teste de mTSP e que a maior parte dos trabalhos científicos utiliza instâncias adaptadas da TSPLIB. Em seu trabalho, o autor desenvolve um GA e compara seu desempenho com vários outros. A forma como esse artigo compara os resultados é muito interessante pois ao invés de comprimir os resultados em valores de média e desvio padrões, todos os resultados são disponibilizados em visualizações gráficas, ou seja, nenhuma informação é perdida desse modo. O algoritmo desenvolvido por Shuai et al. [24], mostrou resultados mais efetivos e eficientes em suas comparações com os outros algoritmos.

Al-Omeer and Ahmed [8] utiliza cromossomo único e compara seu desempenho com operadores: *Alternating Position Crossover (AEX), Edge Recombination Crossover (ERX), Order Crossover (OX), Partial Matched Crossover (PMX)* e *(SCX)*. Al-Omeer and Ahmed [8] conclui que o *SCX* é o melhor método de cruzamento, uma vez que ele encontra a melhor solução com desvio padrão pequeno do resultado. Contudo quando o tempo gasto pelos operadores também é comparado o operado *PMX* tornasse interessante, uma vez que apesar dele não ter conseguido os melhores resultados teve resultados bons com um tempo de execução, que foram de 3 até 50 vezes mais rápido que os outros operadores.

No artigo de Bektas [9] é feita uma revisão bibliográfica sobre o mTSP. Nele são discutidas as variações do mTSP, bem como onde essa modelagem de problema tem sido aplicada e quais têm sido os resultados obtidos. Além disso, Bektas [9] também faz um revisão sobre várias metodologias de solução mTSP, desde abordagens com algoritmos exatos, como *Branch and Bound, Cutting plane*, utilizando heurísticas, como *genetic algorithms* e *Neural networks* até soluções que buscam transformar o problema em TSP. Ele conclui que o problema mTSP é um problema muito importante tanto para o campo teórico como também em aplicações práticas e também diz que métodos de soluções adaptados do problema TSP não são muito eficientes para solucioná-los.

Agradecimentos

O autor agradece ao orientador do artigo Elder Vicente de Paulo Sobrinho e a Júlia Jordana Zuanazzi ao apoio no projeto.

Referências

[1] C. H. Papadimitriou, "The Euclidean travelling salesman problem is NP-complete," *Theoretical Computer Science*, vol. 4, no. 3, pp. 237–244, jun 1977. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/0304397577900123

[2] G. Reinelt, *The Traveling Salesman*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, vol. 840. [Online]. Available: http://link.springer.com/10.1007/3-540-48661-5

[3] M. Liu and P. Zhang, "New hybrid genetic algorithm for solving the multiple traveling saleman problem: an example of distribution of emergence materials," *J Syst Manag*, vol. 23, no. 02, pp. 247–254, 2014.

[4] R. D. Angel, W. L. Caudle, R. Noonan, and A. Whinston, "Computer-assisted school bus scheduling," *Management Science*, vol. 18, no. 6, pp. B–279–B–288, 1972. [Online]. Available: https://doi.org/10.1287/mnsc.18.6.B279

[5] L. Tang, J. Liu, A. Rong, and Z. Yang, "A multiple traveling salesman problem model for hot rolling scheduling in shanghai baoshan iron & steel complex," *European Journal of Operational Research*, vol. 124, no. 2, pp. 267–282, 2000. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S037722179900380X

[6] M. A. Al-Furhud and Z. Hussain, "Genetic Algorithms for the Multiple Travelling Salesman Problem," *International Journal of Advanced Computer Science and Applications*, vol. 11, no. 7, 2020. [Online]. Available: http://thesai.org/Publications/ViewPaper?Volume=11&Issue=7&Code=IJACSA&SerialNo=68

[7] G. Reinelt, "TSPLIB–a traveling salesman problem library," *ORSA Journal on Computing*, vol. 3, no. 4, pp. 376–384, 1991.

[8] M. A. Al-Omeer and Z. H. Ahmed, "Comparative study of crossover operators for the mtsp," in *2019 International Conference on Computer and Information Sciences (ICCIS)*, 2019, pp. 1–6.

[9] T. Bektas, "The multiple traveling salesman problem: an overview of formulations and solution procedures," *Omega*, vol. 34, no. 3, pp. 209–219, jun 2006. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0305048304001550

[10] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM review*, vol. 59, no. 1, pp. 65–98, 2017. [Online]. Available: https://doi.org/10.1137/141000671

[11] J. Weibezahn and M. Kendziorski, "Illustrating the benefits of openness: A large-scale spatial economic dispatch model using the julia language," *Energies*, vol. 12, no. 6, 2019. [Online]. Available: https://www.mdpi.com/1996-1073/12/6/1153

[12] E. Cantú-Paz, "On random numbers and the performance of genetic algorithms," in *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO'02. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002, p. 311–318.

[13] J. Lang, "Random numbers — julia docs," 2021, [Online; accessed 18-Junho-2021]. [Online]. Available: https://docs.julialang.org/en/v1/stdlib/Random/

[14] Wikipédia, "Linear congruential generator — wikipédia, a enciclopédia livre," 2021, [Online; accessed 18-Junho-2021]. [Online]. Available: https://en.wikipedia.org/wiki/Linear_congruential_generator

[15] ISO, *ISO/IEC 9899:2011 Information technology — Programming languages — C*. Geneva, Switzerland: International Organization for Standardization, December 2011. [Online]. Available: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853

[16] M. Mitchell, *An introduction to genetic algorithms*. Cambridge, Mass: MIT Press, 1996.

[17] S. Chatterjee, C. Carrera, and L. A. Lynch, "Genetic algorithms and traveling salesman problems," *European Journal of Operational Research*, vol. 93, no. 3, pp. 490–510, Sep. 1996. [Online]. Available: https://doi.org/10.1016/0377-2217(95)00077-1

[18] A. Király and J. Abonyi, *Optimization of Multiple Traveling Salesmen Problem by a Novel Representation Based Genetic Algorithm*. Springer Berlin Heidelberg, 07 2011, vol. 366, pp. 241–269.

[19] D. Whitley, "A genetic algorithm tutorial," *Statistics and Computing*, vol. 4, no. 2, jun 1994. [Online]. Available: http://link.springer.com/10.1007/BF00175354

[20] G. Mendel, *Experiments in plant Hybrydization*. Brünn Natural History Society, 1865.

[21] E. C. BROWN, C. T. RAGSDALE, and A. E. CARTER, "A grouping genetic algorithm for the multiple traveling salesperson problem," *International Journal of Information Technology & Decision Making*, vol. 06, no. 02, pp. 333–347, 2007. [Online]. Available: https://doi.org/10.1142/S0219622007002447

[22] D. Singh, M. Singh, T. Singh, and R. Prasad, "Genetic algorithm for solving multiple traveling salesmen problem using a new crossover and population generation," *Computación y Sistemas*, vol. 22, 07 2018.

[23] M. Albayrak and N. Allahverdi, "Development a new mutation operator to solve the traveling salesman problem by aid of genetic algorithms," *Expert Systems with Applications*, vol. 38, no. 3, pp. 1313–1320, 2011. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417410006238

[24] Y. Shuai, S. Yunfeng, and Z. Kai, "An effective method for solving multiple travelling salesman problem based on nsga-ii," *Systems Science & Control Engineering*, vol. 7, no. 2, pp. 108–116, 2019. [Online]. Available: https://doi.org/10.1080/21642583.2019.1674220

[25] S. Noor, M. I. Lali, and M. S. Nawaz, "Solving job shop scheduling problem with genetic algorithm," *Science International*, vol. 27, pp. 3367–3371, 08 2015.

[26] I. M. Oliver, D. J. Smith, and J. R. C. Holland, "A study of permutation crossover operators on the traveling salesman problem," in *Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithms and Their Application*. USA: L. Erlbaum Associates Inc., 1987, p. 224–230.

[27] S. Yuan, B. Skinner, S. Huang, and D. Liu, "A new crossover approach for solving the multiple travelling salesmen problem using genetic algorithms," *European Journal of Operational Research*, vol. 228, no. 1, pp. 72–82, 2013. [Online]. Available: https://www.sciencedirect.

com/science/article/pii/S0377221713000908
[28] G. Marsaglia, “Random numbers fall mainly in the planes,” *Proceedings of the National Academy of Sciences*, vol. 61, no. 1, pp. 25–28, 1968. [Online]. Available: https://www.pnas.org/content/61/1/25