Problema do Caixeiro-viajante Aplicado ao Jogo Snake

Arthur Severo, Victor Le Roy e Vinicius Nascimento

Otimização 1 / CEFET-MG

July 5, 2023

Resumo: Este artigo descreve como foi solucionado o problema do jogo Snake de coletar todas as comidas realizando o menor caminho a partir de um algoritmo genético de minimização e do solver CPLEX. Neste, será descrito o problema, bem como a descrição do modelo utilizado, a explicação do algoritmo em detalhes e comparação entre os resultados obtidos pelo CPLEX e pelo algoritmo genético.

Abstract: This article describes how the problem of collecting all the food in the Snake game by taking the shortest path was solved using a genetic algorithm for minimization and the CPLEX solver. It will present the problem description, the description of the model used, a detailed explanation of the algorithm, and a comparison of the results obtained by CPLEX and the genetic algorithm.

Contents

1 Introdução

Neste artigo será apresentado o processo utilizado para encontrar um caminho ótimo no jogo Snake, de forma que sejam visitados todos os pontos em que há comida, assim como a comparação dos resultados obtidos por diferentes soluções. Para esta comparação foi utilizado o software da IBM, o CPLEX [CPL], para executar o algoritmo simplex com base na modelagem do problema do caixeiro-viajante e obter uma solução ótima do problema. Esta solução será comparada aos resultados obtidos por um algoritmo genético.

2 Descrição do problema e das técnicas de solução

O problema do caixeiro-viajante é um desafio clássico de otimização combinatória, que tem como objetivo encontrar o caminho mais curto que um caixeiro-viajante deve percorrer para visitar um conjunto de cidades. O problema é conhecido por sua alta complexidade, uma vez que a quantidade de possíveis rotas cresce exponencialmente com o número de cidades.

No contexto deste trabalho, adaptamos esse problema ao jogo Snake, introduzindo algumas restrições específicas. Em vez de cidades, temos células de comida em um tabuleiro representando a área de jogo, e a cobra precisa percorrer o tabuleiro de forma a visitar cada uma dessas células ao menos uma vez. O objetivo é encontrar o caminho mais eficiente que permita que a cobra conclua o objetivo. No decorrer do trabalho, foram utilizados diversos algoritmos de busca para encontrar uma solução satisfatória para o problema, desconsiderando quaisquer colisões da cobra consigo mesma.

Inicialmente, utilizamos o algoritmo A* (A estrela), que é um algoritmo de busca heurística amplamente utilizado em problemas de caminho mais curto. No entanto, devido à natureza complexa e dinâmica do jogo Snake, o A* sozinho não foi suficiente para encontrar soluções próximas da ótima.

Diante dessa limitação recorremos aos algoritmos genéticos, que são algoritmos não determinísticos inspirados no processo de evolução natural. A seleção natural, o cruzamento e a mutação são aplicados para gerar uma nova geração de indivíduos com caminhos possivelmente mais curtos.

Foram testadas duas variações do algoritmo genético: a primeira tem uma abordagem mais simples, onde cada iteração considerava o caminho mais curto para cada uma das comidas partindo da posição atual da cabeça da cobra. Neste primeiro algoritmo, o caminho mais curto foi definido pelo menor número de passos que a cobra precisou dar para pegar as comidas disponíveis. Na segunda variação foi introduzida a ideia de um caminho total como indivíduo, em que o ótimo seria escolhido pelo caminho em que a soma da distância entre todas as comidas seria a menor. Este último será abordado com mais profundidade no decorrer deste artigo.

Essas variações do algoritmo genético mostraram-se mais adequadas para

lidar com a complexidade do jogo Snake. Ao longo de várias iterações, o algoritmo genético pôde explorar diferentes combinações de movimentos e evoluir gradualmente caminhos que minimizassem a quantidade de células percorridas pela cobra ao ingerir todas as comidas do tabuleiro de forma a tentar se aproximar da solução ótima com um tempo de execução razoável.

3 Coleta de dados

Para a coleta de dados foi utilizado o software CPLEX [CPL] bem como a saída resultante do algoritmo genético.

4 Modelagem

4.1 Grafo

A princípio, para facilitar o entendimento da modelagem, é importante ilustrar como foram definidos os grafos e as restrições específicas do jogo que se diferenciam do modelo clássico do caixeiro-viajante, que serão utilizados para executar os algoritmos:

1. As cidades, definidas como as comidas no jogo Snake, estão todas conectadas entre si;
2. A distância entre as cidades é definida pela distância Manhattan;
3. O ponto de origem do grafo é a cabeça da cobra;
4. A cobra deve passar por todas as comidas e voltar para o ponto de origem;

4.2 Caixeiro-viajante

O modelo para o caixeiro-viajante utilizado [Woc20] foi proposto por Miller-Tucker-Zemlin (MTZ). Neste modelo cada cidade $i = 1, ..., n$ possui arestas para as outras cidades vizinhas $j = 1, ..., n$. Se i e j não são vizinhas entende-se que não há rota direta entre as duas cidades. Portanto, pode-se afirmar que uma aresta ij é a aresta de saída da cidade i e a de entrada na cidade j. Sendo assim, a variável de decisão x_{ij}, que expressa existência de arestas diretas, é dada por:

- $x_{ij} = 1$, se existe aresta direta entre as cidades i e j;
- $x_{ij} = 0$, caso contrário.

Mas, já que trabalhamos com grafos completos, a variável x_{ij} funcionará da seguinte forma:

- $x_{ij} = 1$, se a aresta entre as cidades i e j fizer parte do caminho escolhido;
- $x_{ij} = 0$, caso contrário.

Também é definida uma variável auxiliar c_{ij} que expressa o custo de sair da cidade i e chegar na cidade j. Dessa forma, a função objetivo por ser expressa por:

- min $f(x) = \sum_{i=1}^{n} \sum_{j=1,\ j\neq i}^{n} c_{ij} \times x_{ij}$

Para definir as restrições é necessário criar a variável u_i que funciona com uma auxiliar atrelada a cada cidade existente.

Sujeito às restrições:

1. $\sum_{i=1, i\neq j}^{n} x_{ij} = 1$, para todo $j = 1, 2, 3, ..., n$
   (Aresta que saem da cidade i até a cidade j);
2. $\sum_{j=1, j\neq i}^{n} x_{ij} = 1$, para todo $i = 1, 2, 3, ..., n$
   (Arestas que entram na cidade j a partir da cidade i);
3. $u_i - u_j + nx_{ij} \leq n - 14$
   (Restrições de eliminação de subrotas);
4. $2 \leq i \neq j \leq n$;
5. $x_{ij} \in \{0, 1\}$, para todo $i, j = 1, 2, 3, ..., n$, sendo $i \neq j$;
6. $u_i \in \mathbb{R}^+$ tal que $i = 1, 2, 3, ..., n$.

4.3 Distância de Manhattan

A distância de Manhattan, também conhecida como distância L1 ou geometria do táxi, é uma métrica usada para calcular a distância entre dois pontos em um espaço euclidiano com um sistema de coordenadas fixo. A sua representação é dada pela soma das diferenças absolutas entre as coordenadas dos pontos.

Por exemplo, no plano, a distância de Manhattan entre o ponto P1 com coordenadas $(x1, y1)$ e o ponto P2 em $(x2, y2)$ é $|x1 - x2| + |y1 - y2|$. A distância de Manhattan é comumente usada em desenvolvimento de jogos como uma heurística para o Algoritmo A*. É importante notar que a distância de Manhattan depende da rotação do sistema de coordenadas, mas não depende de sua reflexão em torno de um eixo ou suas translações. [Wik]

5 Algoritmo Genético

Para a implementação do algoritmo genético utilizado, foi necessário estabelecer um passo a passo para facilitar o entendimento do fluxo de obtenção do resultado desejado:

1. O back-end recebe os dados do front-end (tabuleiro do jogo Snake). Estes dados contém as posições de todas as comidas presentes no jogo, bem

como a posição inicial da cobra;

2. Traduz as informações recebidas de forma a obter-se uma matriz de mesmo tamanho que o tabuleiro contendo as posições das comidas e a posição inicial da cobra;
3. Execução do algoritmo (função objetivo, seleção, crossover e mutação) e tradução do resultado obtido para movimentar a cobra corretamente.

Com o passo a passo bem definido, o objetivo seguinte foi a implementação do algoritmo genético, explicado a seguir:

5.1 Entrada e tradução dos dados

A entrada de dados é uma matriz de tamanho predefinido e as posições em que as comidas e a cobra estarão no instante inicial de jogo. A tradução destes dados foi dada da seguinte maneira:

1. O primeiro passo é traduzir a matriz recebida de forma a obtermos as posições (x, y) das comidas e da posição inicial da cobra;
2. O segundo passo foi realizar o cálculo da distância de Manhattan entre todos os pontos de interesse, sendo eles as comidas e a posição inicial da cobra;
3. Com a distância de Manhattan calculada para todos os pontos, são geradas estruturas de dados constituídas de uma tupla de formato [posição (x,y), distância de Manhattan] de forma a possibilitar a criação de um grafo imaginário onde todos os vértices são os pontos de interesse conectados entre si e o custo das arestas é definido pelos valores obtidos pelo cálculo da distância de Manhattan.

5.2 O algoritmo

O algoritmo genético se divide em *função de fitness*, *seleção*, *crossover* e *mutação*, além do indivíduo que compõe a população de cada geração criada durante a execução do algoritmo.

5.2.1 Indivíduos

Os indivíduos possuem como gene o caminho que a cobra deverá percorrer, isto é, uma lista ordenada contendo todas as comidas cujas quais a cobra deve passar.

Um ponto importante a ser ressaltado é que a ordem das comidas é gerada aleatoriamente para cada indivíduo da primeira população. Isto é feito para aumentar a diversidade da população.

5.2.2 Função fitness

A função fitness, ou função objetivo, é definida pelo custo necessário para seguir o caminho presente em seu gene. Este cálculo é feito utilizando da distância de Manhattan obtida no passo de tradução dos dados e deve ser realizado para todos os indivíduos da população. Um ponto importante é que, como o problema estudado é de minimização, quanto menor o custo, melhor será o indivíduo comparado aos outros de sua mesma geração.

5.2.3 Seleção

Após o cálculo da função objetivo, a população é reordenada e remapeada de modo que os indivíduos com menor custo, ou menor distância percorrida, sejam os mais relevantes na geração, isto é, estes serão os "primeiros". Em seguida, o processo de seleção se inicia. Este processo é definido como um sorteio realizado por uma roleta em que, caso um indivíduo seja selecionado, este passará para a próxima geração, caso contrário o mais relevante é copiado para a próxima população.

5.2.4 Crossover

Após o processo de seleção, é executado o algoritmo de *crossover*, que nada mais é do que a seleção de 2 indivíduos, que serão um pai e uma mãe, e a criação de uma combinação de seus genes de forma a criar filhos que farão parte da próxima geração. Essa combinação foi definida como um corte dentro dos genes dos pais e que o resultado deste corte será invertido nos filhos, ou seja, o corte do pai vai para o gene da mãe e vice-versa.

Este corte pode gerar um problema que é a possibilidade de existir a mesma comida repetida em um mesmo gene, em outras palavras, a cobra teria que passar em um local que já havia sido "comido" e deixará um outro alimento para trás. Para solucionar este problema, foi utilizado um algoritmo chamado PMX. De forma simples, este algoritmo serve para cruzamento e utiliza de permutações para evitar com que os pontos de interesse se repitam, neste caso, as comidas.

Um fator importante a ser citado é que foi utilizado um valor arbitrário e predefinido para representar a taxa de *crossover*, isto é, foi definido uma taxa para evitar que todos os indivíduos realizem o cruzamento, porém não é uma garantia, já que é gerado um número aleatório para ser comparado com esta taxa. Caso o número sorteado seja menor do que o valor da taxa, o *crossover* é executado, caso contrário, os pais continuam para a geração seguinte.

5.2.5 Mutação

Por fim, a mutação é exercitada. De modo semelhante ao *crossover*, na mutação também existe uma taxa que evita que todos os indivíduos passem por este processo. O processo de escolha funciona da mesma forma que o processo

citado no cruzamento, porém, o intuito aqui é gerar novos indivíduos ao mesmo tempo que tenta-se manter os melhores já existentes para a geração seguinte.

Caso o indivíduo sendo estudado caia no processo de mutação, ocorre o seguinte procedimento:

1. O primeiro passo é o sorteio de dois índices que representam posições das comidas presentes no gene;
2. Com as posições sorteadas, trocamos as comidas referentes aos índices sorteados. Por exemplo, caso os números sejam 1 e 5, as comidas 1 e 5 trocaram de lugar entre si;
3. Por fim, este novo indivíduo mutante segue para a próxima geração.

Após toda a população ser submetida a todas as fases apresentadas, uma nova geração é iniciada e o algoritmo executa até o número máximo de gerações definidas.

6 Algoritmo Simplex

Antes de recorrer a biblioteca CPLEX, surgiu a ideia de utilizar uma solução própria para encontrar o melhor caminho. O foco da implementação seria utilizar do conceito de resolução alto nível do Simplex e implementar em Python 3.

6.1 Matrizes e Vetores

Com o intuito de facilitar manuseio de matrizes e vetores foram definidas classes de nome respectivo bem como operações pertinentes para recuperar linhas e colunas de forma rápida. Além disso, necessária a implementação de produto escalar entre lista e vetor. Isso tudo para ser capaz de lidar com as informações que viriam do vetor C de coeficientes da função objetivo, matriz A de coeficientes das restrições, bem como matrizes B e N que representam os coeficientes da base e da não básica separadamente. Além dessas a matriz b contendo os termos independentes das restrições.

6.2 Método de Gaus

Toda essa estrutura foi definida para que fosse mais fácil implementar o algoritmo numérico para solução de equações. Ver exemplo da Figura 1.

Mesmo que não seja o mais otimizado, o Método de Gaus permite encontrar uma matriz diagonal de coeficientes equivalente a matriz original de forma a tornar computacionalmente factível a solução de um sistema linear de equações. A implementação adotada se aproxima do primeiro modelo apresentado por Frederico Campos [Fre01].

```
Gx = h

[1;-3;2][x1]   [11]
[-2;8;1][x2] = [-15]
[4;-6;5][x3]   [29]

L   multi_factor              a              b    operation
1                              1;-3; 2      11
2   m21 = -(-2)/1 = 2         -2; 8;-1     -15
3   m31 = -(4)/1 = -4          4;-6; 5      29
4                              0; 2; 3       7    m21*L1 + L2
5   m22 = -(6)/2 = -3          0; 6;-3     -15    m31*L1 + L3
6                              0; 0;-12    -36    m22*L4 + L5

G = [ 1;-3; 2 ]
    [ 0; 2; 3 ]
    [ 0; 0;-12]

h = [ 11 ]
    [  7 ]
    [-36 ]

x3 = -36/-12 = 3
x2 = (7 - 3*3)/2 = -1
x1 = (11 - (-3)*(-1) - (2)*(3))/1 = 2
```

Figure 1: Exemplo Método de Gaus

Em linhas gerais, tem-se uma matriz G qualquer e os termos independentes h. A solução consiste em definir pivôs e somar as linhas de forma a obter uma matriz diagonal superior em relação à diagonal principal. Dessa forma, a primeira variável a ser recuperada será a última e todas as outras são recuperadas a partir demais.

O grande problema desse algoritmo é que ele depende da matriz ser diagonalizável. Segundo Boldrini [Jos80], uma matriz é diagonalizável se e somente se ela possuir um conjunto completo de autovetores linearmente independentes. Isso será um problema e será discutido a frente.

6.3 Núcleo

A implementação obedece ao modelo analítico do simplex a risca. Antes de começar mudanças de base é feita avaliação inicial da função objetivo. Para começar as iterações, são necessários 5 parâmetros iniciais: vetor CB e CN que possuem os coeficientes da função objetivo relacionados as variáveis básicas e não básicas respectivamente. As matrizes B e N de coeficientes das equações de restrições relacionadas com as variáveis básicas e não básicas respectivamente. Por fim, o vetor b com os termos independentes das equações de restrição.

Com as matrizes B transposta, CB e Método de Gaus é possível recuperar

vetor lambda transposto. Em sequência, com as matrizes B e b é possível recuperar o vetor Xb. No próximo passo são encontrados os custos relativos Cxn para cada variável não básica usando CN, lambda transposto e colunas de N.

Caso exista candidato a entrar na base e exista um épsilon viável as colunas de B e N, CB e CN trocam posições, definindo nova base. Portanto, é feito cálculo da função objetivo no instante atual. O algoritmo segue em loop até que as condições de parada sejam atingidas ou o método de Gaus não seja capaz de resolver o problema.

6.4 Modelo e Inconsistências

No caso do Snake, tem-se em primeiro momento informações sobre as distâncias de Manhattan. Entretanto, é necessário construir as matrizes necessárias com base na quantidade de comidas no tabuleiro.

O que acabou por desqualificar essa implementação nas comparações de desempenho é que mesmo com esses tratamentos que foram abordados, em muitos casos ocorriam dependência linear entre as equações de restrições a medida que as colunas trocavam de lugar. Provavelmente, bibliotecas mais robustas como a CPLEX estão preparadas para lidar com esses casos de borda. Coisa que não é possível identificar antes da execução do modelo.

7 Resultados e Discussões

Para todas as execuções realizadas utilizamos as seguintes quantidades de pontos de interesse: $[20, 35, 50, 65, 80]$. Estes valores foram escolhidos pois durante a execução do CPLEX, a partir de 80 comidas, o tempo de execução ultrapassava facilmente 2 horas de duração, além de estourar o limite de memória e, por conta disto, será demonstrado apenas as instâncias em que a execução no software CPLEX não estoure esses limites.

7.1 CPLEX

Table 1: Resultados obtidos para o algoritmo simplex obtido pelo CPLEX

Algoritmo CPLEX					
Numero de comidas	Resultado	Tempo de Execução (s)	Iterações	Nós do branch-and-bound	Gap
20	266	0.33	215	0	0%
35	318	0.97	50741	3613	0.67%
50	384	10.68	497149	31464	0.53%
65	400	10.62	610709	19949	0.98%
80	444	3302.14	12816355	779295	0.26%

Como explicado anteriormente, apenas os resultados que estavam dentro

estipulado foram demonstrados e, mesmo que os demonstrados estejam dentro do que foi considerado viável, percebe-se uma tendência para o aumento do tempo de execução, das iterações e nos valores de *branch-and-bound* conforme os pontos de interesse aumentam. Os valores de gap não necessariamente irão aumentar, pois o ideal é sempre se aproximar do valor ótimo e, mesmo que não obtivemos os valores exatos das soluções ótimas, isto é, o valor de gap sendo 0, conseguimos valores bem próximos e que satisfazem o intuito do projeto. A partir dos dados obtidos e demonstrados na tabela, pode-se concluir sobre os resultados obtidos pelo CPLEX:

- Como esperado, o algoritmo do caixeiro-viajante aumenta sua complexidade dos cálculos conforme o aumento de comidas a serem coletadas. Isto pode ser visto a partir do aumento significativo de tempo que ocorre a cada intervalo do aumento de 30 pontos de interesse. Não só isso, mas o número de iterações também aumenta significativamente a cada número de comidas.
- Com relação à exatidão do resultado obtidos, basta olhar a coluna representada pelo título de "Gap". Este resultado gera a conclusão de que todos pontos ótimos encontrados pelo software foram bem próximos ao valor que poderíamos chamar de solução ótima, isto ocorre, porque quanto mais próximo de 0, mais provado será que este resultado é um ótimo. Obtivemos o 0 em apenas uma linha, isto é, uma única linha obteve a otimalidade, e isto provavelmente se deve ao fato das diversas soluções encontradas pelo software, os diversos cálculos realizados por ele e o fato de que o modelo utilizado neste projeto não é o mais otimizado, mas como todos os resultados foram bem próximos de 0, consideramos estes valores satisfatórios para o intuito desejado.
- Por fim, considerando os nós do algoritmo *branch-and-bound* percebe-se um aumento considerável conforme o aumento dos pontos de interesse. Isso é um resultado esperado, uma vez que este algoritmo realiza cortes de forma a tentar eliminar subproblemas que não contêm solução ótima.

7.2 Algoritmo Genético

Para os resultados obtidos pelo algoritmo genético, foi realizado sua execução quatro vezes para cada intervalo de comidas. Desta forma, obtivemos o seguinte resultado:

Table 2: Resultados obtidos para o algoritmo genético

Algoritmo Genético		
Numero de comidas	Resultados	Tempo de execução (s)
20	[352, 290, 280, 362]	[4,27, 4,46, 4,47, 4,44]
35	[452, 454, 436, 444]	[6,7, 6,5, 6,4, 6,5]
50	[520, 528, 568, 526]	[8,6, 8,5, 8,5, 8,6]
65	[626, 646, 596, 596]	[10,44, 10,44, 10,5, 10,5]
80	[792, 718, 672, 718]	[12,359, 12, 12,8, 12,3]

Com os resultados obtidos no Algoritmo Genético, pode-se concluir:

- Os resultados obtidos não serão sempre o mesmo, porém sempre estão buscando por um valor mínimo. Isto é, não é possível prever um valor por conta do algoritmo ser estocástico (ou não determinístico), ou seja, o algoritmo pode encontrar um valor mínimo local e, por conta disto, retornar valores bem diferentes considerando várias execuções.
- Mesmo que o algoritmo seja não determinístico, o tempo de execução é sempre bem próximo, o que mostra que o algoritmo é consistente com relação a esta variável.

Um ponto importante a ser ressaltado é que estes resultados poderiam ser melhores caso fosse realizado um ajuste fino no algoritmo, isto é, fosse analisado, individualmente, cada problema estudado e alterar os dados, sendo eles o número de geração, a quantidade de indivíduos e as taxas de crossover e mutação. Isto não foi feito neste projeto para mantermos uma base para a comparação dos resultados obtidos.

7.3 Simplex

Mesmo que o algoritmo apresente alguns cálculos inconsistentes, foram obtidos os seguintes resultados com o Simplex:

Table 3: Resultados obtidos para o algoritmo simplex obtido pelo Simplex

Algoritmo Simplex					
Numero de comidas	Resultado	Tempo de Execução (s)	Iterações	Nós do branch-and-bound	Gap
20	220	2.34	34	x	x%
35	287	27.37	82	x	x%
50	3266	67.47	94	x	x%
65	4265	163.74	115	x	x%
80	6301	379.00	148	x	x%

Com os resultados obtidos no Simplex, foi possível notar que:

- O tempo de execução aumenta consideravelmente com a solução por meio do método de Gaus que já era esperado pois é uma solução $O(n^3)$ sendo

chamada várias vezes.

- O tempo de execução só não foi maior porque casos de borda de dependência linear foram encontrados no meio do processamento e, portanto, a execução foi interrompida. Isso justifica valores muito altos encontrados para função objetivo.
- Essa solução analítica só é funcional em casos com quantidade pequena de variáveis e mesmo assim precisa de melhorias para dar conta desses casos onde não é possível usar o Método de Gaus.
- Cabe ressaltar que o arquivo model.json usado para o teste com 80 comidas tem incríveis 6MB. Para um arquivo JSON, isso é muita coisa e mostra como essa solução escalou mal.

7.4 Comparação entre execuções

Pela inconsistência dos resultados, o Simplex, será ignorado nesse tópico. Para realizar a comparação entre os resultados obtidos entre o CPLEX e o algoritmo genético, foi necessário realizar o cálculo da média e do desvio padrão nos dados do genético para realizar uma análise de maneira mais precisa.

Table 4: Média e desvio padrão realizados nos resultados do algoritmo genético

Algoritmo Genético				
Número de comidas	Resultados	Desvio padrão	Tempo de execução (s)	Desvio padrão
20	321	41.968	4	0.094
35	448	8.226	6.5	0.126
50	548	35.534	8.55	0.058
65	611	24.495	10.47	0.035
80	718	49.652	12.330	0.330

Com base na tabela de média e desvio padrão para os resultados do Algoritmo Genético é possível dizer que:

1. O algoritmo genético gerou resultados distantes do ótimo para cada número de comidas em, respectivamente, 55, 130, 164, 211 e 274 unidades, mostrando que quanto maior a quantidade de comidas, mais distante do ótimo é o resultado.
2. O tempo de execução do algoritmo genético se mantém menor do que a execução do simples a partir de, aproximadamente, 50 comidas no tabuleiro.
3. O desvio padrão de ambos os resultados e os tempos de execução do algoritmo genético mostram que, enquanto os resultados podem apresentar valores diversos, o tempo de execução parece apresentar boa confiabilidade.

8 Conclusão

Com base na análise dos resultados apresentados, pode-se assumir que o algoritmo genético é uma alternativa eficiente para problemas de grande escala. Isto acontece, pois, mesmo que o algoritmo entregue respostas em que o ótimo não é um valor próximo, o tempo de execução é significativamente inferior ao se comparar ao CPLEX.

No entanto, ao considerar casos em que é necessário obter o valor ótimo, é recomendável utilizar o CPLEX. É importante ressaltar que o desempenho do CPLEX é superior em soluções com um número menor de pontos de interesse, quando comparado ao algoritmo genético implementado. Além de apresentar um tempo de execução mais rápido, o CPLEX fornece resultados próximos ao ótimo, o que significa que o caminho percorrido é menor.

9 Informações adicionais

1. Para visualizar a execução do algoritmo genético (o que possuiu menor caminho) lado a lado com a execução do CPLEX, basta visualizar este vídeo: `https://youtu.be/EEhUep0MpmE`;
2. Para visualizar o código fonte do projeto: `https://github.com/Sevzera/otm1-cefet`.

References

[Jos80] BOLDRINI José Luiz et al. *Álgebra linear*. 1980.

[Fre01] CAMPOS Frederico Ferreira. *Algoritmos Numericos*. 2001.

[Woc20] Claudemir Woche. *Modelagem e Resolução do Problema do Caixeiro Viajante com Python e Pyomo*. [Online; accessed 2021-05-07]. 2020. URL: http://www.opl.ufc.br/pt/post/tsp/.

[CPL] IBM CPLEX. *Cplex optimizer*. URL: https://www.ibm.com/br-pt/analytics/cplex-optimizer.

[Wik] a enciclopédia livre Wikipédia. *Geometria do táxi*. URL: https://pt.wikipedia.org/wiki/Geometria_do_t%C3%A1xi.