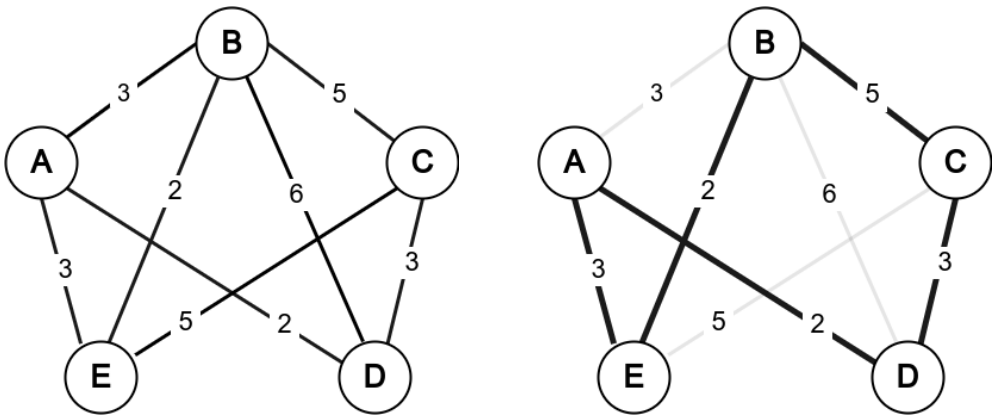
Algoritmo Genético para solução do Caixeiro Viajante

João Victor Melo[1], **Vitor Melo Lopes**[1], **Wesley Vitor**[1], **Rafael Nogueira Leal**[1]

[1]Departamento de Computação - Universidade Federal do Piauí (UFPI)
Piauí - PI

1. Problema

O caixeiro viajante trata-se de um problema combinatorial de otimização no qual, dado um grafo ponderado $G(V, E)$, deve-se encontrar um percurso hamiltoniano cujo peso das arestas é mínimo. Esse problema é dito NP-difícil, ou seja, não foi descoberto algoritmo em tempo polinomial capaz de resolvê-lo. De fato, podem ser geradas $|V|!$ permutações sobre o conjunto de vértices, que representam o conjunto completo das possíveis soluções, e mesmo com técnicas de projeto de algoritmos e heurísticas, ainda há problemas de escalabilidade para problemas maiores. Uma instância do problema resolvida:



(a) Problema do caixeiro viajante. (b) Solução ótima.

Figure 1.

2. Implementação

A implementação do algoritmo genético está organizada em 2 classes: a classe TravellingSalesperson define o problema, lendo algumas combinações a partir da especificação TSPlib [1] de problemas combinatoriais do tipo, a forma de codificação, decodificação/função objetivo e validação de uma solução; a classe GeneticAlgorithm efetivamente roda o algoritmo e salva seu estado a cada iteração, sendo construído a partir de um problema e funções que definem seus operadores genéticos, com a possibilidade de alterar alguns parâmetros genéticos, a saber, se o crossover substitui os indivíduos escolhidos, se é aplicada alguma correção ou detecção de solução fora inviável, o tamanho da população, a probabilidade de crossover, a probabilidade de mutação, a proporção de elitismo, o máximo de iterações e o máximo de iterações sem melhoria.

2.1. Cromossomos

A codificação dos cromossomos é uma permutação do percurso da possível solução. Por ser uma permutação do conjunto de vértices, não deve haver elementos repetidos, propriedade refletida pelos operadores escolhidos.

2.2. Função Objetivo

A função objetivo é o somatório do peso das arestas encontradas na decodificação da permutação do percurso descrito no cromossomo, de modo que quanto menor, melhor.

2.3. Seleção

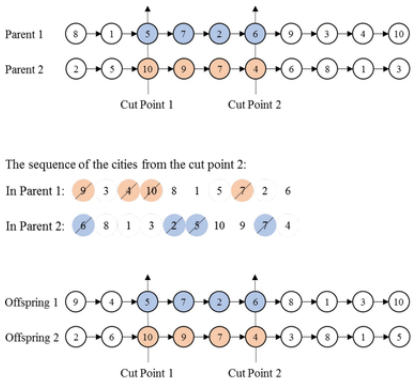
Para a seleção dos indivíduos, foi escolhido o **operador de seleção proporcional de aptidão**, ou roleta, de forma que a cada indivíduo foi atribuída uma probabilidade proporcional à sua aptidão, de modo que os indivíduos mais aptos têm maior chance de gerar filhos.

2.4. Cruzamento

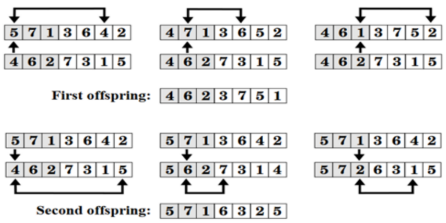
Para a operação de cruzamento, utilizou-se o **operador OrderedX**, de crossover ordenado. O operador OX (figura 2(a)) garante a unicidade dos genes após a operação, sendo ideal para a codificação em permutação.

Primeiro, selecionam-se dois loci na forma de índices de substrings dos pais selecionados pelo operador de seleção, então cada pai repassará sua substring para a região correspondente nos dois cromossomos filhos. Logo depois, para cada pai, os genes pertencentes ao outro pai, e que não estão contidos na substring transferida são inseridos na ordem que aparecem a partir do segundo ponto de inserção.

Também foi implementado o **operador Partially MappedX** (figura 2(b)) de cruzamento com um ponto de corte, que gera os filhos trocando substrings até o ponto de corte entre os pais e mapeando os genes alterados para outras posições no cromossomo. Esse operador também tem a característica de gerar permutações sintaticamente válidas do problema. Ambos operadores se mostraram eficientes para o problema.



(a) Operador OX [2].



(b) Operador PMX [3].

Figure 2.

2.5. Mutação

Na mutação, usou-se o **operador de inversão de centro**, que consiste em escolher um ponto aleatório no cromossomo e inverter as substrings antes e depois dele.

Também foram implementados os **operadores de deslocamento**, que desloca uma substring na solução final em alguns loci, e **de permuta**, que, para cada bit, tem uma chance de inverter os genes à esquerda e à direita.

É interessante notar que para os operadores de inversão de centro e deslocamento foi necessário aumentar a mutação para 15% - 30% para que houvesse uma melhor exploração, enquanto o operador de permuta, que é bit a bit e não preserva substrings, já exibia boa exploração em 1% a 5% de mutação.

2.6. Parâmetros genéticos

Alguns parâmetros do algoritmo podem ser alterados:

1. **Crossover com substituição**: Refere-se à remoção dos pais quando geram filhos, de modo a aumentar a variabilidade genética. Ligado, por padrão.
2. **Método de correção de solução fora dos limites**: Pode-se gerar outra solução ('regenerate'), ignorar as soluções inviáveis ('ignore'), randomizar novas soluções ('randomize') ou penalizar as soluções inviáveis ('penalize'). Ignorar por padrão.
3. **Tamanho da população**: inteiro que representa a quantidade de indivíduos a cada iteração. 100 por padrão.
4. Probabilidade de crossover: implementada como a proporção de indivíduos da nova geração que são gerados por crossover. 0,7 por padrão.
5. **Probabilidade de mutação**: implementada como a probabilidade da mutação ocorrer para todo filho gerado por crossover, e caso ocorra, uma vez ou bit a bit, a depender da mutação. 0.1 por padrão.
6. **Taxa de elitismo**: proporção mais forte da população anterior que será transplantada da antiga população para a mais nova, substituindo os mais fracos. 0.15 por padrão
7. **Máximo de iterações**: define o máximo de iterações a serem feitas antes de parar. 200 por padrão.
8. **Máximo de iterações sem melhora**: define o máximo de iterações sem melhora no melhor fitness da população. 50 por padrão.

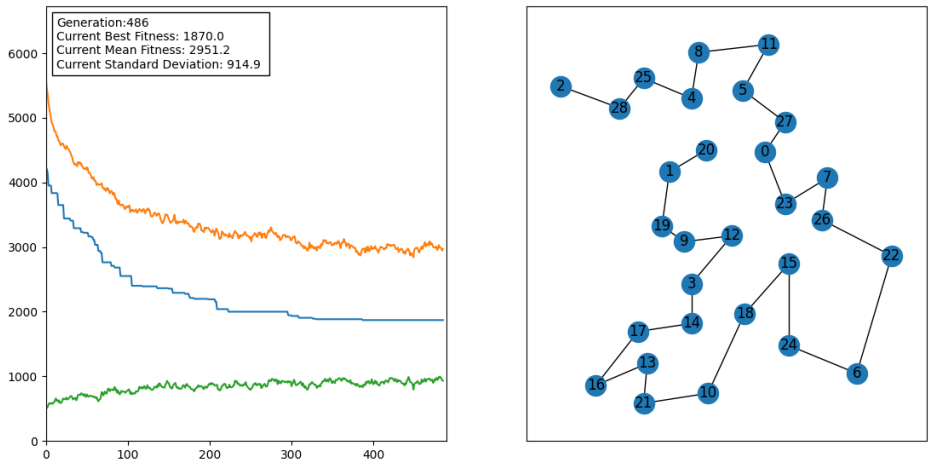
2.7. Condição de Parada

Como critério de parada, optou-se por utilizar tanto o número de iterações máximas, quanto estagnação, sendo esta última opcional. Para limitar a quantidade de iterações.

3. Resultados

A fim de evidenciar a eficácia, o algoritmo foi submetido a dois problemas, um deles se trata da instância informada nos requisitos do trabalho, enquanto o outro trata-se de uma instância de grafo completo, possuindo 29 vértices, ambos dentro do diretório "instances".

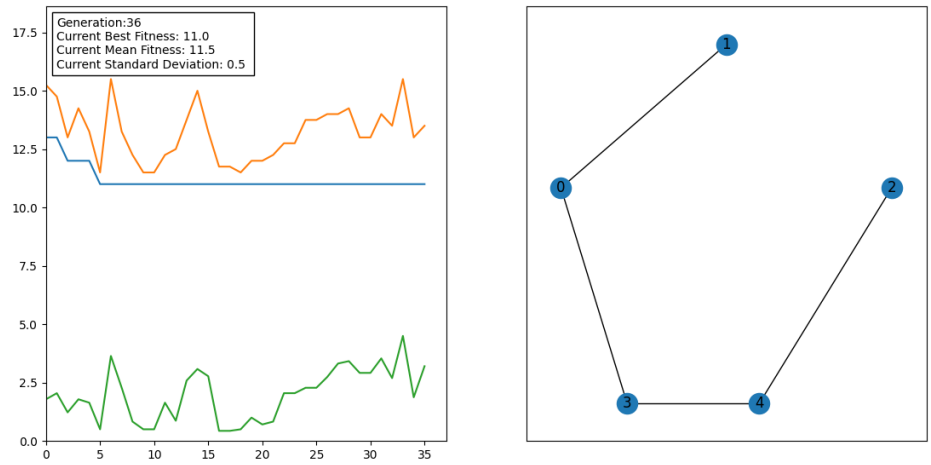
Para o problema de 29 vértices, o algoritmo foi testado utilizando o operador OX para o crossover, juntamente a mutação de inversão de centro, o método de correção de solução fora dos limites *ignore*, o tamanho da população igual a 300, a probabilidade de crossover igual a 0.7, a probabilidade de mutação igual a 0.1, a taxa de elitismo igual a 0.15, o máximo de iterações igual a 800, o máximo de iterações sem melhora igual a 100. Foi gerado o seguinte resultado:



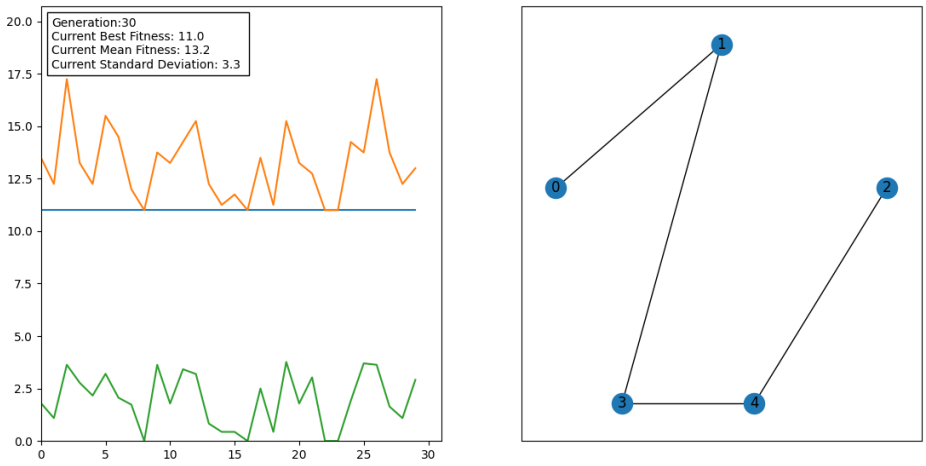
(a) Resultado do Algoritmo.

Figure 3.

Já para a instância informada nos requisitos do trabalho, utilizou-se os mesmos operadores genéticos acima, diferindo-se apenas em alguns parâmetros, tais quais, o tamanho da população que foi alterado para 5, a taxa de elitismo para 0.2, o máximo de iterações para 200, o máximo de iterações sem melhoria para 30, por fim o método de correção de solução fora do limite foi alterado para *regenerate*. obtendo-se os seguintes resultados:



(a)



(b)

References

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

[2] I. Ilhan and G. Gökmen, "A list-based simulated annealing algorithm with crossover operator for the traveling salesman problem," *Neural Computing and Applications*, vol. 34, May 2022. DOI: 10.1007/s00521-021-06883-x.

[3] G. Üçoluk, "Genetic algorithm solution of the tsp avoiding special crossover and mutation," *Intelligent Automation and Soft Computing*, vol. 3, Jan. 2002. DOI: 10.1080/10798587.2000.10642829.