Universidade Estadual de Maringá
Centro de Tecnologia
Departamento de Informática

Ciência da Computação
Modelagem e Otimização Algorítmica

Avaliação 3

Professor: Ademir Aparecido Constantino

129037	Murilo Luis Calvo Neves
129268	Leandro Eugênio Farias Berton

Data: 31/01/2025

Conteúdo

1 Introdução

Este trabalho consiste na implementação e análise de um algoritmo genético para o problema do Caixeiro Viajante (PCV, ou TSP em inglês). Para se implementar o algoritmo, foram realizadas as etapas de leitura e armazenamento dos dados de entrada, construção da população inicial e operações de cruzamento e seleção até atingir o critério de parada.

Assim como visto na primeira parte desta matéria, o PCV consiste em determinar o menor caminho que percorre um conjunto de vértices visitando cada um exatamente uma vez e retornando ao vértice inicial. O PCV possui complexidade NP-Difícil, i.e, não se conhece ainda um algoritmo que encontre uma solução exata em tempo polinomial, por isso, heurísticas são frequentemente utilizadas para encontrar soluções próximas da ótima em tempo computacionalmente aceitável.

Após a construção do algoritmo, foram realizados os testes para as instâncias u574, pcb1173, pr1002, brd14051, fnl4461, d15112, pla33810 e pla85900, todas pertencentes ao TSPLib. Aqui foram utilizadas as seguintes estratégias em cada etapa:

- **Leitura e armazenamento dos dados de entrada:** Foi seguida a mesma estratégia do primeiro trabalho.
- **Estrutura para representação da população:** Foi utilizada uma *struct* em C para se representar a solução juntamente com demais dados auxiliares.
- **Construção da população inicial:** Foi utilizada uma variação das heurísticas construtivas do primeiro trabalho juntamento com a adição de um não-determinismo às mesmas. Em especial, foi utilizada heurística do Vizinho Mais Próximo (VMP).
- **Seleção:** Optou-se pela seleção utilizando o método de roleta.
- **Operadores de cruzamento:** Aqui foram utilizados os operadores de *Zoning* [citar] e EXX [citar], que foram escolhidos após uma pesquisa bibliográfica sobre o tema.
- **Função de mutação:** Foi-se utilizada uma mutação baseada em *pair-swap* não-determinístico.
- **Critério de parada:** Foram utilizados três critérios de parada, o primeiro é a quantidade de gerações sem melhoria, o segundo é um limite de gerações (como observado em [citar]) e também o *timeout* do primeiro trabalho.

Para efeitos de comparação entre os operadores, serão analisados para cada instância o GAP%, que consiste em:

$$GAP\% = \frac{Alg - MS}{MS} \times 100 \tag{1}$$

Onde Alg é o custo gerado pelo algoritmo sendo analisado e MS é a melhor solução conhecida na literatura.

1.1 Detalhes de implementação

1.1.1 Considerações gerais

Assim como no primeiro trabalho, por questão de performance, os algoritmos serão implementados em linguagem C. As instâncias estão codificadas utilizando o formato padrão .tsp, onde são definidas as coordenadas de cada vértice (nó) como em um plano x,y (considera-se um grafo completo).

Foram utilizados um computador de mesa e um laptop para a implementação e execução dos algoritmos heurísticos. O primeiro possui um processador Apple M2 com 16GB de RAM (arquitetura ARM), o outro conta com um processador Intel i5-11300H com 8GB de RAM (arquitetura X86). Durante as execuções, notou-se uma grande diferença na velocidade de processamento entre as duas

máquinas. Desse modo, assumir-se-á ambos o tempo de execução e a quantidade de iterações como métricas de execução, a fim de uma comparação mais justa e precisa entre as diversas instâncias e parâmetros.

1.1.2 Leitura dos dados de entrada

A leitura dos dados de entrada foi reutilizada integralmente do primeiro trabalho, sendo feita por uma função de *parse* que se encontra em um arquivo fora da *main*, onde há uma função que recebe o arquivo de entrada e monta as estruturas de dados correspondente, bem como faz a análise, limpeza e leitura das informações contidas nele.

1.1.3 Armazenamento dos dados

Um conceito muito importante no contexto dos algoritmos genéticos é o de população. Dessa forma, é imprescindível implementá-lo de uma maneira eficiente e intuitiva, pois essa será a base de todo o desenvolvimento. Pensando nisso, decidiu-se criar uma *struct* para a representação de uma população, a qual encapsula o tamanho da população, seus cromossomos (um vetor de vetores com os índices das cidades que formam uma rota) e também a avaliação (ou aptidão) de cada indivíduo.

```
typedef struct populacao
{
    int tamanho;
    int **cromossomo;
    float *avaliacao;
}populacao;
```

O uso dessa estrutura de dados personalizada facilitou a passagem de parâmetros entre as várias funções que compõe o algoritmo final, além de melhorar a legibilidade do código. A *struct* populacao é alocada dinamicamente a cada execução.

Os dados de cada vértice estão armazenados em uma *struct* do tipo:

```
typedef struct {
    float x;
    float y;
} coordenada;
```

E estes estão agrupados em um vetor de *DIMENSION* elementos. Note que, devido às restrições de *overflow* da linguagem C, esse vetor (e demais vetores também, como o de solução) tiveram que ser declarados por uma chamada explícita de alocação de memória para o programa, uma vez que instâncias muito grandes atingem e ultrapassam os limites estabelecidos pela linguagem.

Por sua vez, a saída foi representada por um vetor de inteiros, onde:

$$V_k = j \longrightarrow \text{O k-ésimo vértice da solução é o vértice j} \tag{2}$$

Note também que o vetor de saída possui *DIMENSÃO + 1* elementos, pois há uma volta para o vértice inicial ao fim do ciclo, além de que na representação interna do programa, o vértice i no arquivo TSP é representado pelo valor $i - 1$, pois vetores na linguagem C iniciam-se em 0.

1.1.4 Arquivos de saída

Para facilitar as análises das execuções do algoritmo, decidiu-se utilizar um formato de arquivo textual que aqui foi nomeado de *timestamp*. Esse arquivo é aberto ao início da execução do programa e a cada iteração tem os seguntes dados escritos em suas linhas:

- Iteração atual
- Tempo de execução

- Melhor custo obtido
- Custo médio da população atual
- Pior custo da população atual
- Melhor custo da população atual

Um exemplo de linha do *timestamp* tem a seguinte aparência:

```
186 1.151000 21383.734375 22268.556641 22891.314453 21383.734375
```

1.1.5 Arquivos de teste

Para facilitar a automatizar execuções, foi utilizado um arquivo de testes escrito na linguagem Python que realiza a combinação entre todas as possibilidades de parâmetros que se deseja testar e realiza uma chamada para o *shell* do sistema operacional (SO) executando o programa em C compilado.

Como o comando varia para diferentes SOs, para se executar o arquivo de testes deve-se:

1. Definir os parâmetros que se deseja testar
2. Definir o nome do arquivo binário compilado
3. Definir o comando *shell* a ser executado

Com isso, esse arquivo de teste executa todos os testes em sequência, e, conforme eles vão sendo concluídos, renomeia cada arquivo de teste com os seus parâmetros como nome e os move para uma pasta separada chamada *timestamps*, que deve ser criada antes da execução.

1.1.6 Outras automatizações

Além da automatização por meio dos arquivo de testes, foram realizadas outras automatizações para a análise dos arquivos *timestamp* gerados após as execuções desejadas. Esses arquivos estão disponíveis no repositório e servem para auxiliar a geração de gráficos e a extração de valores dos arquivos sem a necessidade de abrir manualmente cada um.

1.1.7 Valores aleatórios

Para o contexto deste trabalho, considerou-se suficiente o grau de aleatoriedade oferecido pela função $rand()$ do C. A função $rand()$ é limitada por uma constante definida como $RAND\_MAX$, que em sistemas UNIX é definida como:

$$RAND\_MAX = \begin{cases} 2^{32} - 1 = 4294967295, \text{para máquinas de 32 bits (não são o caso deste trabalho)} \\ 2^{64} - 1 = 18446744073709551615, \text{para máquinas de 64 bits} \end{cases} \tag{3}$$

No entanto, para sistemas Windows, $RAND\_MAX$ é definida como apenas 32767, o que para este trabalho é insuficiente. Com isso, foi necessário a implementação de uma nova função de valores aleatórios, nomeada de $randMelhorado()$, que busca contornar essa limitação de $RAND\_MAX$ e fornecer uma valor entre 0 e INT_MAX, que é o limite de inteiros da máquina.

Essa função foi feita ao se gerar dois valores aleatórios (*unsigned*) utilizando-se $rand()$, em seguida, foram utilizadas manipulações de bits para se concatenar os bits dos dois valores em um inteiro completo.

1.1.8 Paralelização das execuções

Para se acelerar o processo de execução dos casos de teste, os parâmetros de teste foram divididos entre os dois integrantes da equipe, a fim de paralelizar a execução e evitar o peso excessivo para um sistema.

Além disso, para cada máquina utilizada na execução dos testes, os casos de testes definidos foram divididos entre 3/4 processos de SO distintos por meio dos terminais, e isso maximizou a utilização dos processadores para as execuções devido a melhor utilização dos *cores* dos processadores.

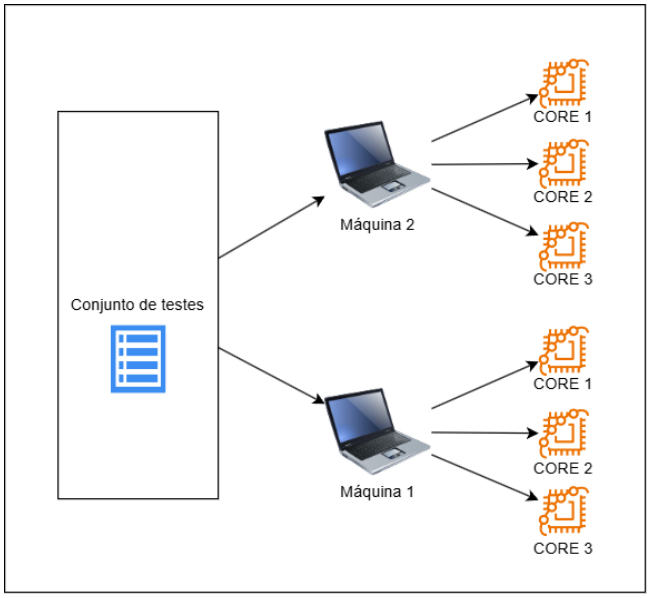

Figura 1: Diagrama da paralelização das execuções

Além disso, foi realizada uma análise sobre o uso de *threads* de *kernel* do SO para se paralelizar as execuções, porém essa ideia acabou não sendo implementada devido a alta diferença entre as interfaces de *threads* entre os sistemas Windows e Linux.

1.2 Disponibilização dos resultados

Todos os arquivos *timestamp* com o resultado das execuções, bem como os artigos consultados para a fundamentação teórica desse trabalho, estão disponíveis em uma pasta compartilhada na plataforma Google Drive. O link de acesso é o seguinte:

https://drive.google.com/drive/folders/10u-MtP5iBZ6CVk1t9krUM0reg93HSUgo?usp=sharing

E ele pode ser acessado por meio deste código QR:

Figura 2: Pasta do drive com artefatos do trabalho

2 Módulos do algoritmo genético

Nesta seção serão apresentadas as decisões de implementação de cada módulo do algoritmo genético, bem como suas respectivas fundamentações. Além do material visto em sala de aula, também foram consultados os artigos [1] e [2] para um melhor aprofundamento nessa classe de algoritmos. Houve, também, uma investigação dos melhores parâmetros para as instâncias analisadas nesse trabalho que será discutida posteriormente.

2.1 Critérios de parada

Neste trabalho foram implementados dois critérios de parada:

1. Quantidade de iterações sem melhoria
2. Tempo limite bruto

O primeiro critério, quantidade de iterações sem melhoria, foi deixado como um parâmetro para a execução, a fim de se avaliar o seu impacto na eficácia do algoritmo genético em obter soluções para as instâncias. Por sua vez, o tempo limite bruto implementado teve como função limitar execuções demasiadamente longas e foi variado conforme a necessidade dos testes. Durante as execuções dos testes esse valor limite foi alterado conforme a necessidade para valores como 6h, 12h, 18h, 24h e 36h.

2.2 Avaliação

Devido a natureza do problema do caixeiro viajante (PCV/TSP), o parâmetro definido como avaliação de cada indivíduo da população foi o seu custo total de ciclo.

O custo total de cada ciclo é definido como:

$$\sum_{i=0}^{dim} dist(V_i, V_{i+1}) \tag{4}$$

Observação: V possui $dim + 1$ elementos.

Onde $dist(A, B)$ é a distância entre dois vértices do problema. Nota-se aqui que, como o TSP se trata de um problema de minimização, o indivíduo com o **menor** custo é considerado o melhor nesse contexto.

2.3 Seleção

A seleção dos indivíduos foi realizada pela realização de uma roleta. Os passos para se selecionar indivíduos aqui foram:

1. Calcular para cada indivíduo a probabilidade como sendo o inverso de seu custo
2. Normalizar as probabilidades pela soma de todos os custos
3. Escolher por acúmulo o número desejado de indivíduos

Note que aqui não foi permitida a repetição de indivíduos, i.e, cada indivíduo que for selecionado para ser gerador da próxima geração pode apenas fazer isso uma vez a cada iteração. Outro detalhe é que, neste passo, devido à implementação do ZX gerar apenas um indivíduo novo, a implementação do OX foi alterada para apenas gerar um também. Com isso, a quantidade de filhos gerados a cada iteração é a metade da quantidade de pais selecionados para cruzamento. Essa quantidade de pais foi definida como 15% do tamanho da população (com o mínimo de 2 indivíduos), consequentemente a quantidade de filhos foi de 7.5%.

2.4 Cruzamento

Foi feita uma revisão bibliográfica usando o motor de busca Google Schoolar e foram encontrados dois principais artigos [3] [4] que faziam uma listagem de operadores de cruzamento e seus respectivos resultados em benchmarks. Tendo em vista esse material, foram escolhidos os operadores utilizados nesse trabalho.

2.4.1 Zoning Crossover (ZX)

O ZX é um operador de cruzamento proposto por Kuroda em 2010 [5] que apresenta bons resultados em benchmarks. Embora a ideia de funcionamento seja clara e intuitiva, em sua publicação original não há um pseudocódigo da implementação, o que se configurou como uma dificuldade a mais para a utilização desse operador no trabalho. Assim, a versão implementada aqui foi desenvolvida inteiramente pelos membros da equipe, seguindo a proposta apresentada no artigo e adaptando para a linguagem de programação C.

O princípio de funcionamento do ZX se resume em trocar as arestas de dois genitores dentro de uma zona delimitada (daí vem o nome característico) e as reconectar gerando uma nova rota. Segue-se, primordialmente, três passos:

Passo 1: Randomicamente, uma cidade é selecionada e é a partir dela que será centrada e delimitada a zona de troca. O largura L_x e a altura L_y dessa janela podem ser calculadas por:

$$L_x = X_{largura} \times \alpha \tag{5}$$

$$L_y = Y_{altura} \times \alpha \tag{6}$$

Tais que $X_{largura}$ e Y_{altura} são a diferença entre o maior e o menor valor de x e y, respectivamente, entre todas as cidades. Além disso, valor de alpha dita o tamanho da janela de corte e, assim como recomendado no artigo, foi escolhido um valor entre 0,1 e 0,3.

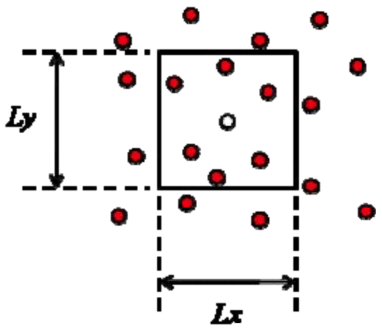

Figura 3: Exemplo da delimitação de uma zona de troca.

Passo 2: Nesse estágio, as arestas de ambos os genitores que cruzam as bordas da zona delimitada são deletadas. Em seguida, as arestas dentro da janela são trocadas entre os genitores já na estrutura dos filhos.

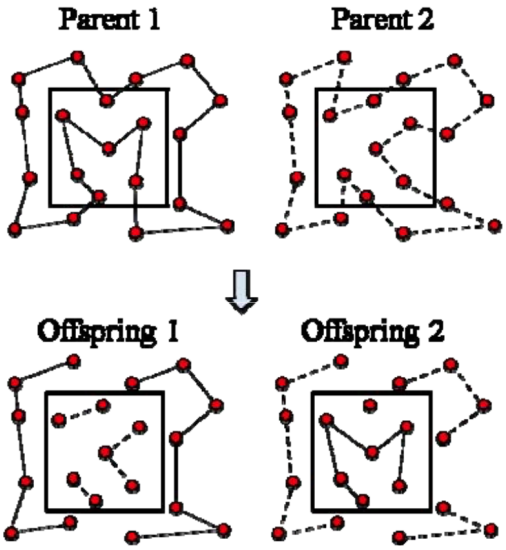

Figura 4: Exemplo de troca das arestas entre dois genitores.

Passo 3: O último passo consiste em fazer as ligações necessárias entre os vértices (cidades) que não estão devidamente conectados com o restante da rota, seguindo uma estratégia gulosa: sempre conecta-se com aresta ao vértice mais próximo.

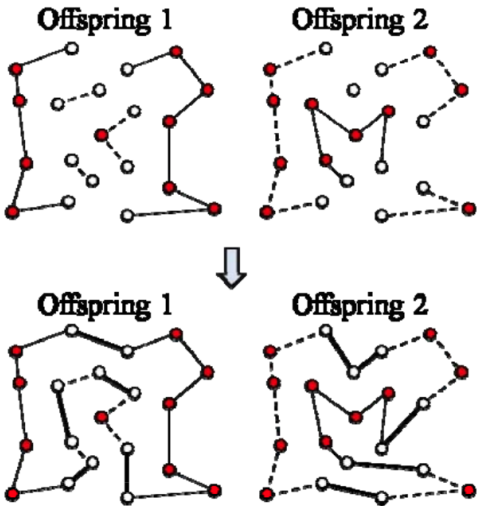

Figura 5: Exemplo de dois filhos gerados a partir da conexão dos vértices trocados.

A estratégia de implementação utilizada foi criar um vetor binário auxiliar para cada genitor que indica componentes conexos. Com isso, é possível detectar as "bordas onde em um determinado vértice falta conexões. O pseudocódigo desse algoritmo encontra-se no apêndice 7.1, embora recomenda-se olhar o código implementado em C para uma maior precisão de entendimento.

2.4.2 Order Crossover (OX)

O algoritmo de Order Crossover (OX) é um operador de cruzamento proposta por Davis em 1985 [6], e consiste no seguinte processo:

1. Do genitor 1, escolher uma subseção aleatória e copiar para o indivíduo filho
2. Copiar os demais vértices do genitor 2 para o filho, em ordem

Uma ilustração que representa bem essa operação é apresentada em [7]:

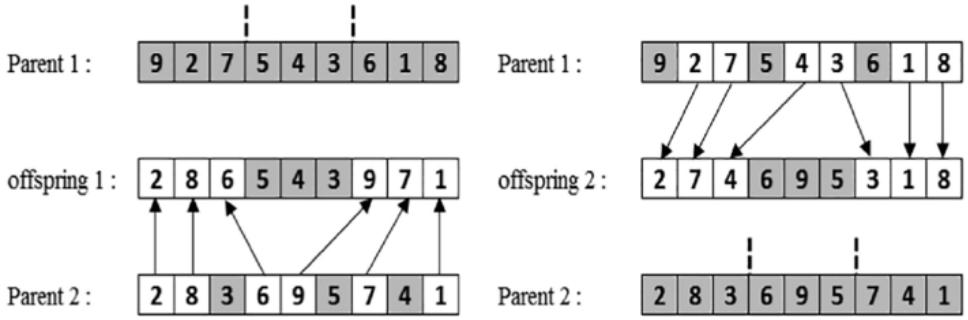

Figura 6: Exemplo do operador OX

Neste trabalho, os pontos de corte inicial e final foram obtidos por meio de uma escolha aleatória de dois pontos do vetor, com validações extras para casos de borda. Note que esse processo tem complexidade $O(n^2)$, uma vez que para inserir os vértices do genitor 2 para o filho deve-se primeiro verificar se este não está na seção advinda do genitor 1, que pode ter tamanho aleatório entre 0 e n.

2.5 Mutação

A função de mutação utilizou-se do método de *pair swap*, implementado na primeira avaliação da disciplina. A primeira versão implementada funcionava da seguinte maneira:

1. Para cada indivíduo, gere um valor aleatório e compare com a taxa de mutação, se menor, prossiga
2. Escolha dois vértices aleatórios
3. Troque os vértices

No entanto, por mais que essa função de mutação inicial tenha trazido bons resultados para instâncias pequenas (como a kroA100 e a u574) em testes preliminares, quando o tamanho da instância passa a aumentar, o efeito observado da mutação tende a ser quase nulo, assim não sendo eficaz para aumentar a variabilidade da população.

Pensando nisso, chamando-se a chance de mutação de α, foi definida uma outra variável chamada de γ que representa a proporção do gene que será mutada. Um exemplo seria, com $\alpha = 1\%$ e $\gamma = 25\%$, uma população de 100 indivíduos de dimensão de 2000 vértices teria, em média, 1 indivíduo mutado e este indivíduo mutado teria 500 trocas *pair swap*.

Com essa nova variável inserida, o algoritmo se torna:

1. Para cada indivíduo, gere um valor aleatório e compare com a taxa de mutação (α), se menor, prossiga
2. Calcule o número de mutações com $\gamma * Dim$
3. Realize $\gamma * Dim$ mutações entre vértices aleatórios

Para este trabalho, utilizou-se como critério que o valor de γ é igual ao valor de α, i.e, a taxa de mutação que é escolhida para o teste também é utilizada como proporção de mutação.

2.6 Busca local

Para a busca local, utilizou-se a implementação do 2-opt já feita no trabalho anterior. No entanto, devido ao custo computacional observado, um dos laços do 2-opt (laço externo) foi modificado para, ao invés de varrer o vetor todo, varrer apenas uma seção de 10% dele que é definida aleatoriamente. O laço interno se manteve inalterado.

2.7 Atualização

A estratégia de atualização de população empregada nesse trabalho baseia-se em uma mescla da abordagem *steady stated* com o elitismo, na qual, a cada geração, são selecionados aleatoriamente 15% da população para serem os genitores e são gerados 7% de novos indivíduos. Esses só entrarão na população se forem melhores que os piores já presentes. Para isso, foi implementada uma função para a atualização da população que recebe a população atual e os filhos gerados na iteração, assumindo que ambas estejam ordenadas pela aptidão.

3 Investigação dos parâmetros

Após a implementação dos módulos do algoritmo genético na linguagem C, foram realizados diversos testes para se averiguar quais parâmetros de entrada produziriam os melhores resultados. É evidente que, em um estudo a ser publicado, seria necessário a repetição desses testes várias vezes a fim de, por meio de testes estatísticos, confirmar as conclusões através deles obtidas, mas esse não se configura como o foco desse trabalho. Com isso, e também com o auxílio do arquivo de testes, foram realizadas as investigações a seguir.

3.1 Alpha

Como já apresentado na seção XX, operador de cruzamento *ZX* utiliza um parâmetro α que delimita o tamanho da região da janela de corte que será trocada no cruzamento entre os dois genitores. Na literatura [5] recomenda-se que tal valor fique entre 0,1 e 0,3. Desse modo, nesse trabalho, foram executados testes a fim de avaliar qual seria o melhor número a ser adotado. Para tanto, os valores de alfa testados foram $\alpha \in \{0.05, 0.1, 0.2, 0.3, 0.4\}$ para as instâncias *brd14051*, *fnl4461* e *u574*. Além disso, foi adotada a taxa de mutação como sendo 1%, 50 indivíduos na população e um número máximo 100 gerações sem melhoras.

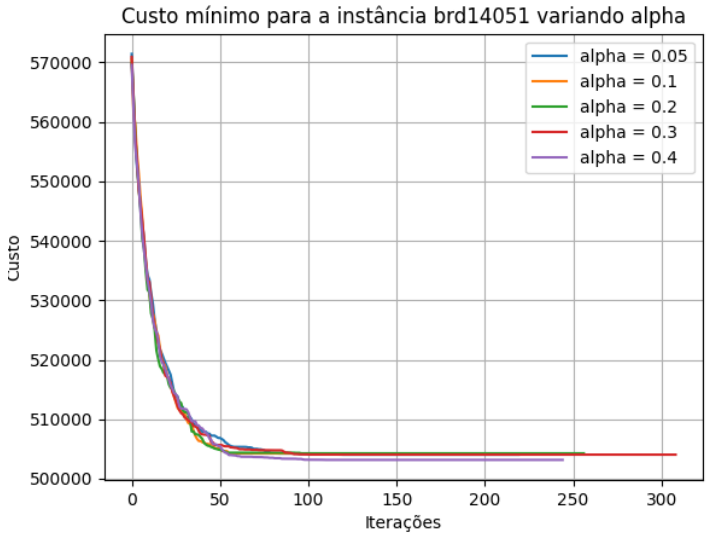

Figura 7: Custo do melhor indivíduo da população ao decorrer das iterações para a instância brd14051 com a variação do parâmetro alpha.

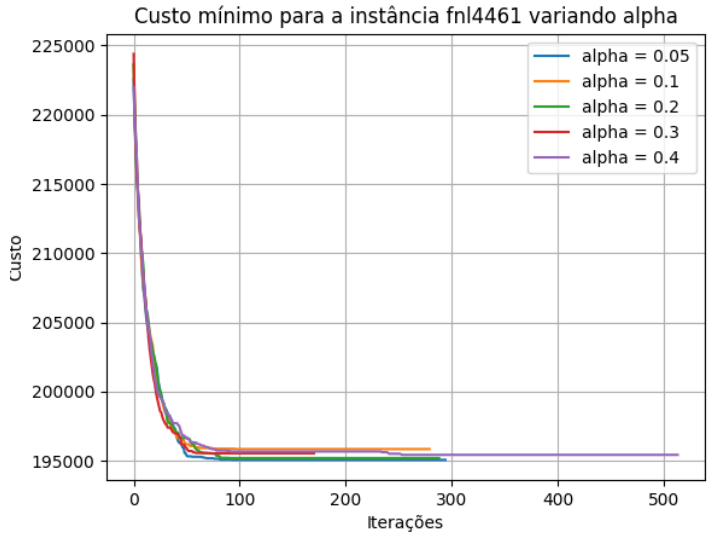

Figura 8: Custo do melhor indivíduo da população ao decorrer das iterações para a instância fnl4461 com a variação do parâmetro alpha.

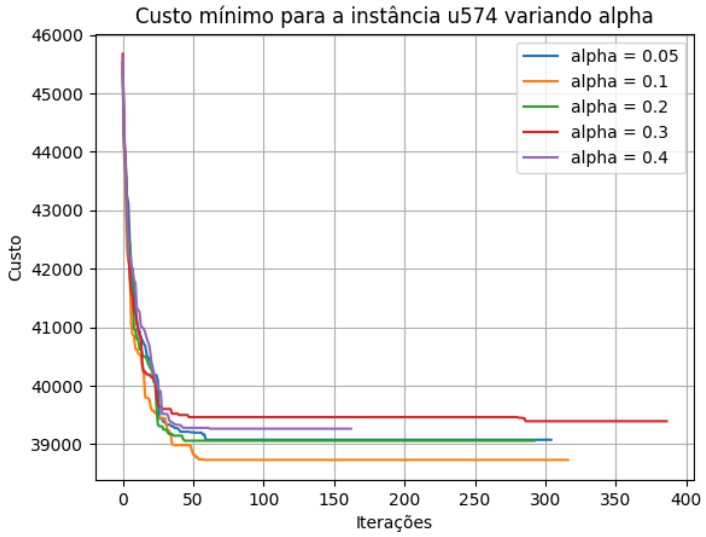

Figura 9: Custo do melhor indivíduo da população ao decorrer das iterações para a instância u574 com a variação do parâmetro alpha.

Valor alpha	**Melhor indivíduo**	**M.S.C.**	**GAP %**
0.05	504031,5	469385	7,38%
0.1	504235,7813	469385	7,42%
0.2	504263,4063	469385	7,43%
0.3	504033,3438	469385	7,38%
0.4	503152,6875	469385	7,19%

Tabela 1: Resultados da variação de alpha para a instância brd14051.

Nota-se que o valor de alpha tem uma relação com a convergência da população, em que um número maior pode levar a misturas mais significativas dos dois genitores e, portanto, o filho acaba sendo mais diferente, demorando mais para convergir a um resultado "estável". Por outro lado, o parâmetro

alpha não demonstrou muita influência na melhor solução encontrada (vide o GAP%), uma vez que a variação observada nas diferentes execuções é tolerável devido a natureza não determinística da geração da população inicial e, por consequência, de todo o algoritmo. Assim, foi escolhido $\alpha = 0.2$ por ser um valor que apresentou resultados bons e consistentes em todos os cenários de teste.

3.2 Impacto da mutação

Uma análise que se mostrou interessante foi o impacto que a mutação ocasiona no comportamento das populações. Nesses testes, foram variados os valores de mutação e se manteve os demais parâmetros de entrada fixos. Os valores fixados foram:

1. Instância: kroA100
2. Tamanho da pop.: 100
3. Critério de parada: 100

O principal impacto dessa mudança se observa nos seguintes gráficos:

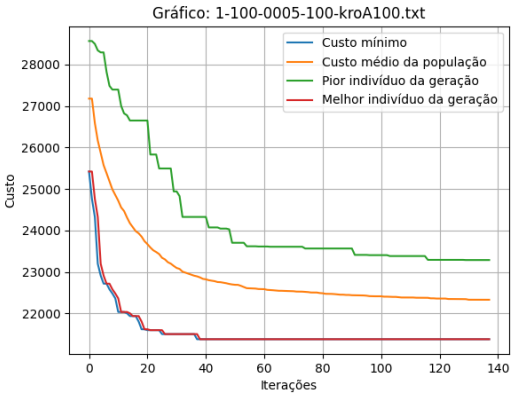

Figura 10: Gráfico do comportamento da população com a taxa de mutação em 0.5%

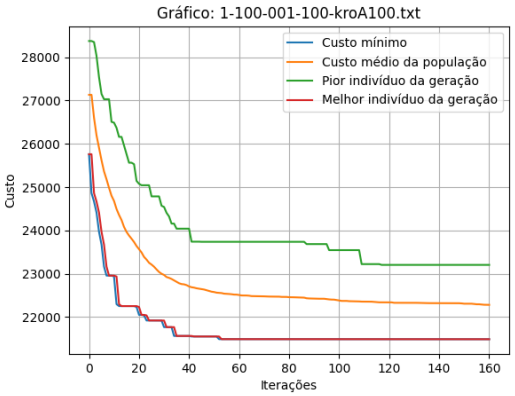

Figura 11: Gráfico do comportamento da população com a taxa de mutação em 1%

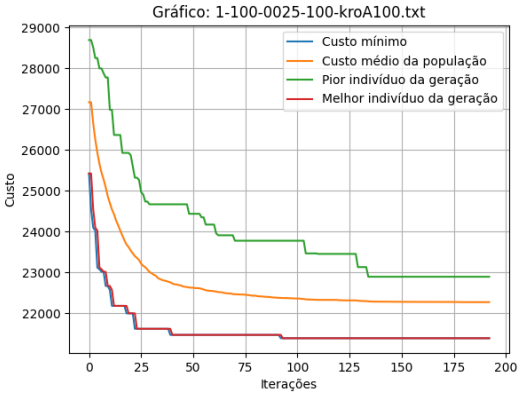

Figura 12: Gráfico do comportamento da população com a taxa de mutação em 2.5%

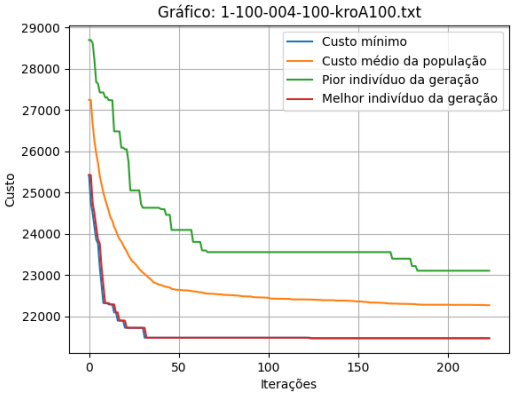

Figura 13: Gráfico do comportamento da população com a taxa de mutação em 4%

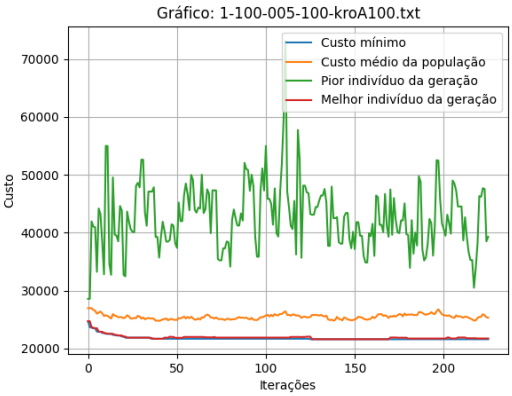

Figura 14: Gráfico do comportamento da população com a taxa de mutação em 5%

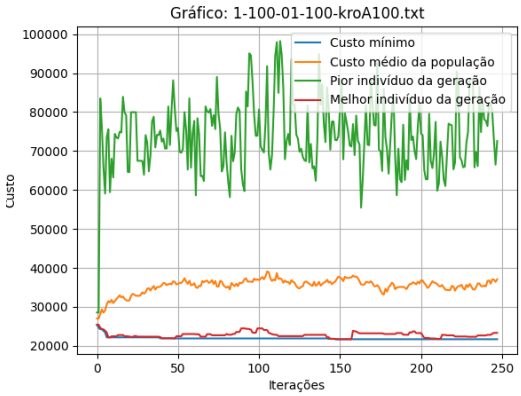

Figura 15: Gráfico do comportamento da população com a taxa de mutação em 10%

Nota-se nestes testes que, após um certo *threshold* de mutação entre 4% e 5% para essa instância, o comportamento da população tende a se tornar erradico. Os resultados numéricos obtidos foram:

Instância	Alg. cr.	N. Pop.	Mut.%	Crit. par.	Resultado	MS	GAP%	t(s)
kroA100	OX	100	10.00	100	21708.82	21282	2.01	1.53s
kroA100	OX	100	5.00	100	21581.42	21282	1.41	1.38s
kroA100	OX	100	1.00	100	21487.44	21282	0.97	0.98s
kroA100	OX	100	4.00	100	21465.75	21282	0.86	1.38s
kroA100	OX	100	2.50	100	21383.73	21282	0.48	1.19s
kroA100	OX	100	0.50	100	21377.52	21282	0.45	0.84s

Tabela 2: Comparação entre as melhores soluções conhecidas da literatura e as obtidas pelo algoritmo com base nos parâmetros de execução

Além disso, após a primeira rodada execução, ao analisar o gráfico gerado para cada instância, notou-se um comportamento peculiar no custo do pior indivíduo da população: a oscilação era consideravelmente grande. Assim, uma segunda rodada de execução foi posta em prática, agora considerando a taxa de mutação como sendo 0,5% ao invés de 1%. Os resultados não divergiram muito, o pior indivíduo da população ainda continuou oscilando com uma menor amplitude. Tal acontecimento pode ser observado logo abaixo, em uma execução para a instância pla33810:

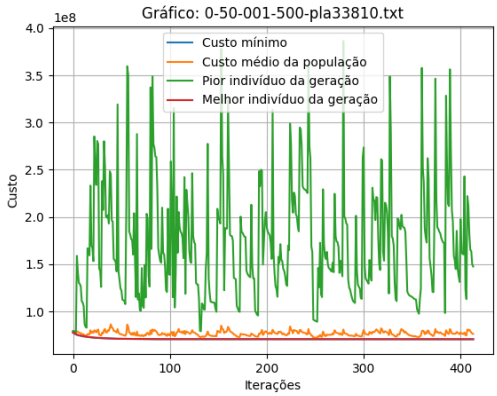

Figura 16

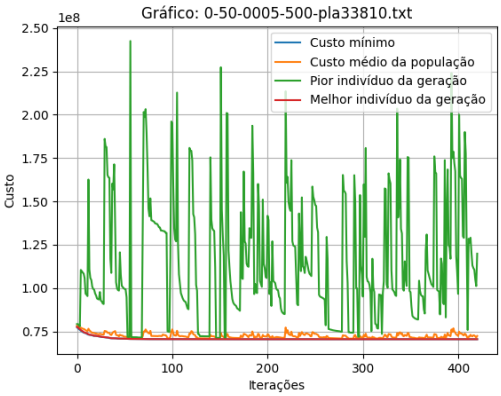

Figura 17

Após isso, foram realizados outros testes com mutações ainda menores para algumas instâncias, utilizando valores como 0.1% e 0.01% e os resultados foram que, além de o compontamento gráfico ter sido o mesmo, os valores GAP% não divergiram (na realidade, foram iguais ou ligeiramente piores do que as execuções com 1%).

3.3 Impacto do número de indivíduos

Um fator importante para o algoritmo genético, também citado em sala, é o tamanho da população. Dessa forma, foram executados testes que visavam explorar a influência do tamanho da população no resultado final. Para esses testes, fixaram-se o operador de cruzamento ZX, a taxa de mutação em 1% e o critério de parada de 500 gerações sem melhora, além de um limite de execução de 12h. As variações da população foram de 50, 500, 1000, e 10000 indivíduos. Esse conjunto de testes foi executado na melhor máquina disponível devido a grande exigência de poder computacional. Os resultados são apresentados logo a seguir:

Instância	Alg. cr.	N. Pop.	Mut.%	Crit. par.	Resultado	MS	GAP%	t(s)
fnl4461	ZX	50	1.00	500	195053.39	182566	6.84	1130.49s
fnl4461	ZX	500	1.00	500	194741.22	182566	6.67	11903.70s
fnl4461	ZX	1000	1.00	500	194469.28	182566	6.52	12494.46s
fnl4461	ZX	10000	1.00	500	194435.36	182566	6.50	39726.92s
Total	–	–	–	–	–	–	–	18h07m33s

Tabela 3: Comparação entre as melhores soluções conhecidas da literatura e as obtidas pelo algoritmo variando o tamanho da população.

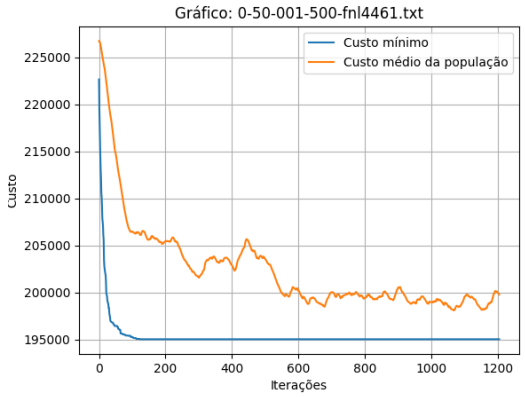

Figura 18: Gráfico do comportamento da população com 50 indivíduos.

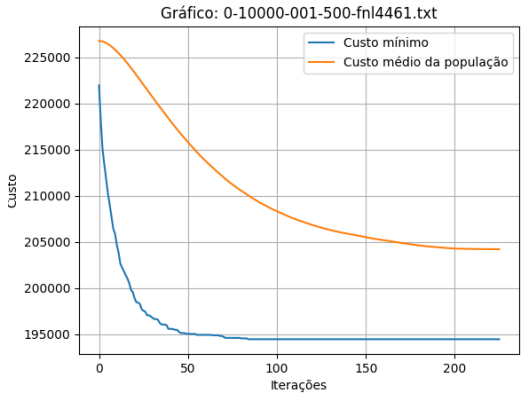

Figura 19: Gráfico do comportamento da população com 10000 indivíduos.

Com base nos valores obtidos, observa-se que o tamanho da população teve uma leve influência no valor da solução final, quebrando um pouco das expectativas iniciais desse experimento. Dessa forma, não é possível afirmar que o custo-benefício de manter uma população relativamente grande é bom, uma vez que isso vem com um custo computacional elevado em troca de uma tênue melhora no resultado final.

3.4 Limite de tempo de execução

A fim de explorar melhor a capacidade do algoritmo genético aqui estudado, definiu-se um teste variando o limite de execução entre 6 horas e 24 horas. Foram escolhidas as duas maiores instâncias para execução, utilizando uma população de 50 indivíduos, taxa de mutação de 1% e um máximo de 500 gerações sem melhora. Logo abaixo são apresentados os resultados do teste:

Instância	Alg. cr.	N. Pop.	Mut.%	Crit. par.	Resultado	MS	GAP%	t(s)
pla33810-24h	ZX	50	1.00	500	70637960.00	66048945	6.95	36652.59s
pla33810-6h	ZX	50	1.00	500	70502928.00	66048945	6.74	21446.05s
pla85900-6h	ZX	50	1.00	500	150969232.00	142382641	6.03	20680.61s
pla85900-24h	ZX	50	1.00	500	150709104.00	142382641	5.85	85611.23s
Total	–	–	–	–	–	–	–	45h39m49s

Tabela 4: Comparação entre execuções variando o tempo máximo de processamento.

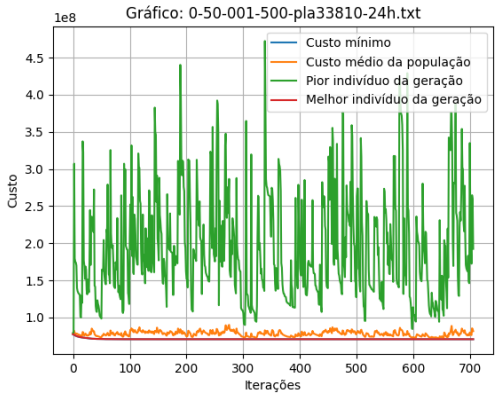

Figura 20: Gráfico do comportamento da população da instância pla33810 com timeout de 24h.

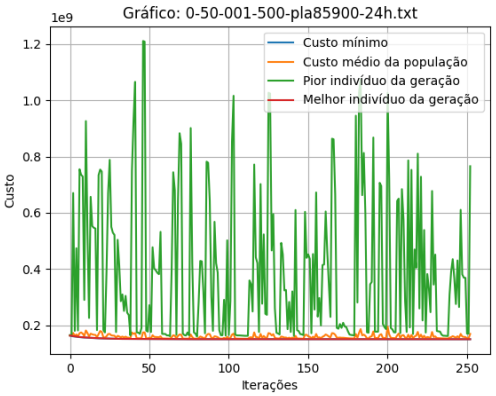

Figura 21: Gráfico do comportamento da população da instância pla85900 com timeout de 24h

Nota-se, observando os resultados obtidos, que não necessariamente tem-se uma melhora no valor final quando aumentado o tempo de execução, sendo essa uma medida relativa ao analisar a diferença entre os GAPs. Como exposto na tabela, o tempo máximo de execução de seis horas apresentou-se o suficiente para todas as instâncias.

3.5 Valores de entrada

Para se definir os parâmetros de entrada para as execuções de todas as instâncias, primeiramente foram realizados alguns testes para se buscar obter o melhor conjunto deles de maneira a diminuir o GAP% ao mesmo tempo em que se mantém um tempo razoável de execução.

Com isso, foram variados os seguintes parâmetros:

1. Tamanho da população: 50 ou 100 indivíduos
2. Algoritmo de *crossover*: ZX ou OX
3. Taxa de mutação: 0.5%, 1% e 10%
4. Critério de parada: 100, 500 e 1000

Nesta seção, ao todo, foram realizados 36 testes, totalizando 53 horas, 42 minutos e 18 segundos de execução.

3.5.1 ZX

Os resultados dos testes de parâmetros ao se utilizar o algoritmo de cruzamento ZX são apresentados a seguir:

Instância	Alg. cr.	N. Pop.	Mut.%	Crit. par.	Resultado	MS	GAP%	t(s)
fnl4461	ZX	50	10.00	1000	198973.91	182566	8.99	975.13s
fnl4461	ZX	50	10.00	500	197993.66	182566	8.45	528.62s
fnl4461	ZX	100	10.00	100	197586.92	182566	8.23	269.26s
fnl4461	ZX	50	10.00	100	196383.23	182566	7.57	157.41s
fnl4461	ZX	100	10.00	500	196170.95	182566	7.45	1076.26s
fnl4461	ZX	50	0.50	100	195794.06	182566	7.25	256.99s
fnl4461	ZX	50	0.50	1000	195782.38	182566	7.24	1234.74s
fnl4461	ZX	50	0.50	500	195770.42	182566	7.23	736.81s
fnl4461	ZX	100	1.00	500	195513.78	182566	7.09	1475.06s
fnl4461	ZX	100	1.00	100	195342.00	182566	7.00	757.32s
fnl4461	ZX	100	1.00	1000	195298.11	182566	6.97	4240.88s
fnl4461	ZX	100	0.50	500	195215.30	182566	6.93	2512.67s
fnl4461	ZX	100	0.50	1000	195217.66	182566	6.93	2440.47s
fnl4461	ZX	50	1.00	500	195064.61	182566	6.85	763.65s
fnl4461	ZX	100	0.50	100	195046.25	182566	6.84	884.18s
fnl4461	ZX	100	10.00	1000	195008.06	182566	6.82	2051.35s
fnl4461	ZX	50	1.00	1000	194956.31	182566	6.79	3975.28s
fnl4461	ZX	50	1.00	100	194885.27	182566	6.75	268.57s
Total	–	–	–	–	–	–	–	06h49m56s

Tabela 5: Comparação dos resultados obtidos com a variação de diversos parâmetros experimentais para o operador ZX na instância fnl4461

3.5.2 OX

Os resultados dos testes de parâmetros ao se utilizar o algoritmo de cruzamento OX foram:

Instância	Alg. cr.	N. Pop.	Mut.%	Crit. par.	Resultado	MS	GAP%	t(s)
fnl4461	OX	100	10.00	500	199585.34	182566	9.32	11838.55s
fnl4461	OX	50	10.00	500	199303.38	182566	9.17	4992.54s
fnl4461	OX	50	10.00	1000	198800.91	182566	8.89	4472.61s
fnl4461	OX	100	10.00	100	197734.94	182566	8.31	2606.42s
fnl4461	OX	100	10.00	1000	197684.58	182566	8.28	19799.45s
fnl4461	OX	50	10.00	100	197458.64	182566	8.16	1276.78s
fnl4461	OX	50	1.00	100	195714.47	182566	7.20	1715.11s
fnl4461	OX	50	0.50	100	195596.31	182566	7.14	1902.60s
fnl4461	OX	100	1.00	100	195477.91	182566	7.07	10393.84s
fnl4461	OX	50	0.50	500	195294.23	182566	6.97	8871.30s
fnl4461	OX	50	0.50	1000	195208.25	182566	6.92	12922.98s
fnl4461	OX	100	0.50	1000	195104.94	182566	6.87	20051.56s
fnl4461	OX	100	1.00	1000	195080.17	182566	6.85	21558.54s
fnl4461	OX	100	0.50	100	194981.66	182566	6.80	3834.50s
fnl4461	OX	100	0.50	500	194911.20	182566	6.76	21552.34s
fnl4461	OX	100	1.00	500	194880.48	182566	6.75	11345.31s
fnl4461	OX	50	1.00	1000	194864.38	182566	6.74	3270.93s
fnl4461	OX	50	1.00	500	194829.34	182566	6.72	6343.23s
Total	–	–	–	–	–	–	–	46h52m22s

Tabela 6: Comparação dos resultados obtidos com a variação de diversos parâmetros experimentais para o operador OX na instância fnl4461

Um ponto importante de salientar é a vasta diferença no tempo de execução entre os dois operadores. Isso é ocasionado pelo fato de os testes com o operador ZX terem sido executados no computador de mesa, que é muito mais poderoso em termos de processamento. Os testes com o operador OX foram realizados em um *laptop* mais simples e menos poderoso, levando muito mais tempo para executar os mesmos testes.

Outro ponto importante é que esse tempo de execução foi dividido em três processos paralelos, efetivamente reduzindo para 1/3 o custo de tempo real de espera.

4 Resultados

4.1 Resultados gerais

4.1.1 Operador ZX

Instância	Alg. cr.	N. Pop.	Mut.%	Crit. par.	Resultado	MS	GAP%	t(s)
u574	ZX	50	1.00	500	38967.55	36905	5.59	18.66s
pcb1173	ZX	50	1.00	500	61699.73	56892	8.45	144.41s
pr1002	ZX	50	1.00	500	272047.50	259045	5.02	53.61s
brd14051	ZX	50	1.00	500	503090.03	469385	7.18	18658.01s
fnl4461	ZX	50	1.00	500	195053.39	182566	6.84	1130.49s
d15112	ZX	50	1.00	500	1688365.62	1573084	7.33	8148.97s
pla33810	ZX	50	1.00	500	70502928.00	66048945	6.74	21446.05s
pla85900	ZX	50	1.00	500	150969232.00	142382641	6.03	20680.61s
Total	–	–	–	–	–	–	–	19h31m17s

Tabela 7: Comparação entre as melhores soluções conhecidas da literatura e as obtidas pelo algoritmo com base nos parâmetros de execução definidos.

Instância	Alg. cr.	N. Pop.	Mut.%	Crit. par.	Resultado	MS	GAP%	t(s)
u574	ZX	50	0.50	500	39236.69	36905	6.32	10.84s
pcb1173	ZX	50	0.50	500	61839.48	56892	8.70	49.83s
pr1002	ZX	50	0.50	500	272490.25	259045	5.19	32.16s
brd14051	ZX	50	0.50	500	502263.28	469385	7.00	8358.53s
fnl4461	ZX	50	0.50	500	196014.38	182566	7.37	965.89s
d15112	ZX	50	0.50	500	1686080.62	1573084	7.18	9276.51s
pla33810	ZX	50	0.50	500	70499360.00	66048945	6.74	21489.68s
pla85900	ZX	50	0.50	500	150865792.00	142382641	5.96	20796.78s
Total	–	–	–	–	–	–	–	16h56m15s

Tabela 8: Comparação entre as melhores soluções conhecidas da literatura e as obtidas pelo algoritmo com base nos parâmetros de execução definidos.

4.1.2 Operador OX

Instância	Alg. cr.	N. Pop.	Mut.%	Crit. par.	Resultado	MS	GAP%	t(s)
u574	OX	50	1.00	500	38942.84	36905	5.52	93.74s
pcb1173	OX	50	1.00	500	61230.77	56892	7.63	557.06s
pr1002	OX	50	1.00	500	273271.66	259045	5.49	303.20s
brd14051	OX	50	1.00	500	501852.94	469385	6.92	12058.10s
fnl4461	OX	50	1.00	500	195432.05	182566	7.05	9316.60s
d15112	OX	50	1.00	500	1688253.25	1573084	7.32	7476.86s
pla33810	OX	50	1.00	500	70557600.00	66048945	6.83	21455.45s
pla85900	OX	50	1.00	500	150896352.00	142382641	5.98	20845.64s
Total	–	–	–	–	–	–	–	20h01m43s

Tabela 9: Comparação entre as melhores soluções conhecidas da literatura e as obtidas pelo algoritmo com base nos parâmetros de execução definidos.

4.2 Gráficos dos resultados para cada instância

4.2.1 u574

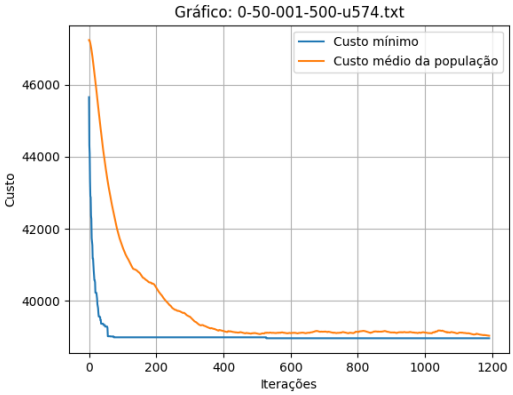

Figura 22: Visualização gráfica do resultado do operador ZX na instância u574

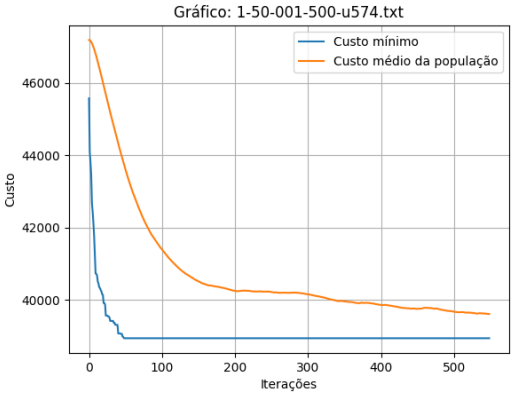

Figura 23: Visualização gráfica do resultado do operador OX na instância u574

4.2.2 pcb1173

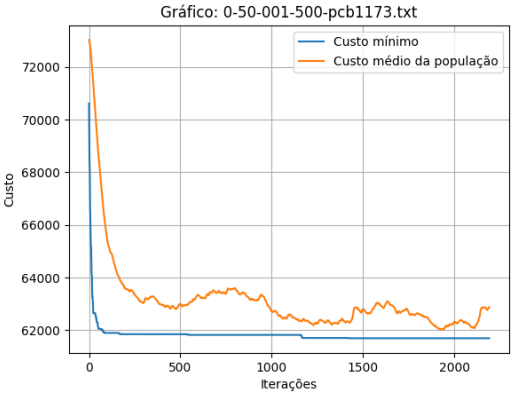

Figura 24: Visualização gráfica do resultado do operador ZX na instância pcb1173

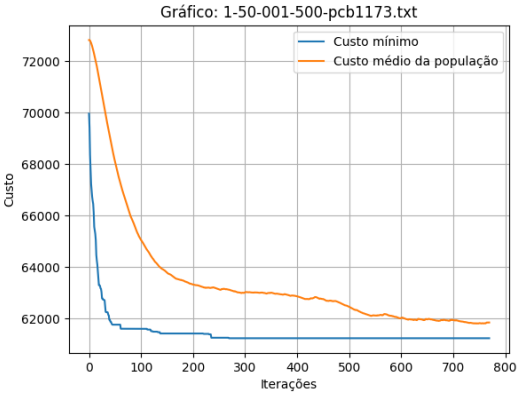

Figura 25: Visualização gráfica do resultado do operador OX na instância pcb1173

4.2.3 pr1002

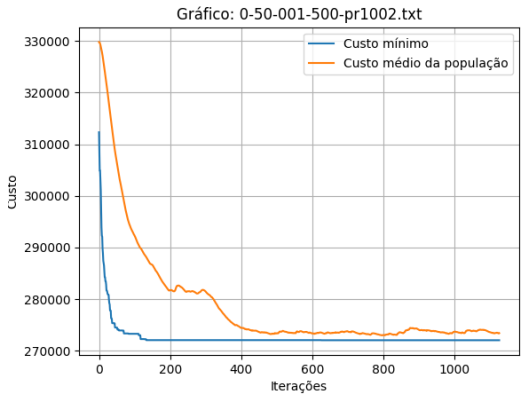

Figura 26: Visualização gráfica do resultado do operador ZX na instância pr1002

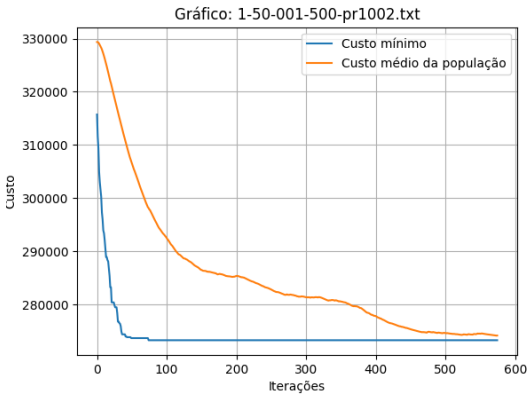

Figura 27: Visualização gráfica do resultado do operador OX na instância pr1002

4.2.4 brd14051

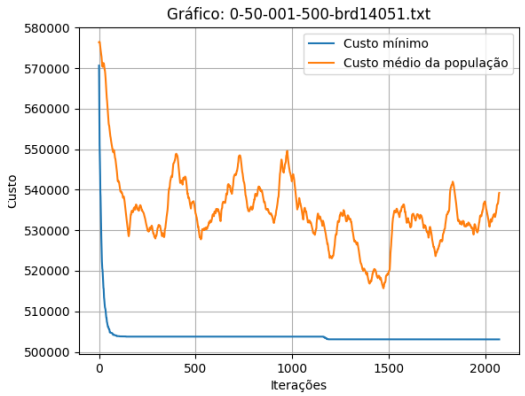

Figura 28: Visualização gráfica do resultado do operador ZX na instância brd14051

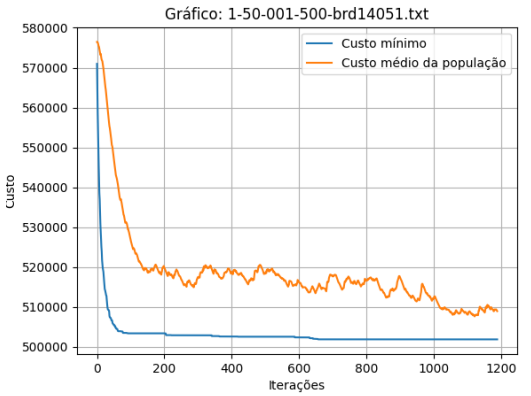

Figura 29: Visualização gráfica do resultado do operador OX na instância brd14051

4.2.5 fnl4461

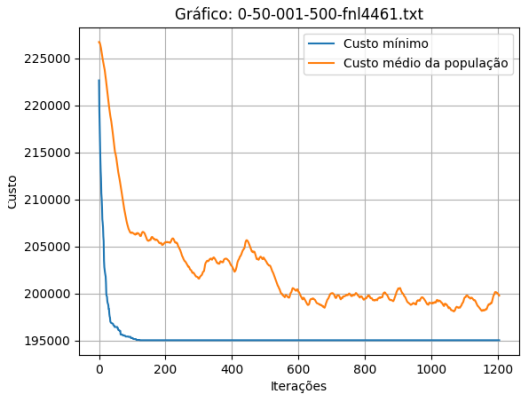

Figura 30: Visualização gráfica do resultado do operador ZX na instância fnl4461

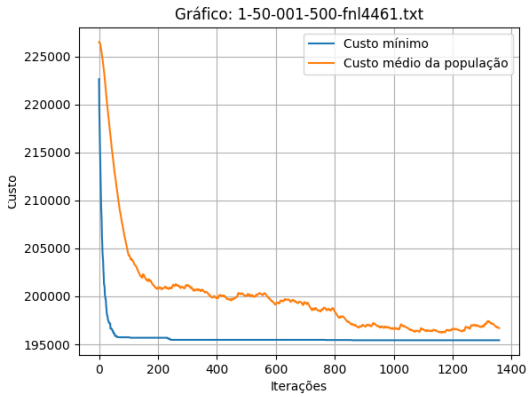

Figura 31: Visualização gráfica do resultado do operador OX na instância fnl4461

4.2.6 d15112

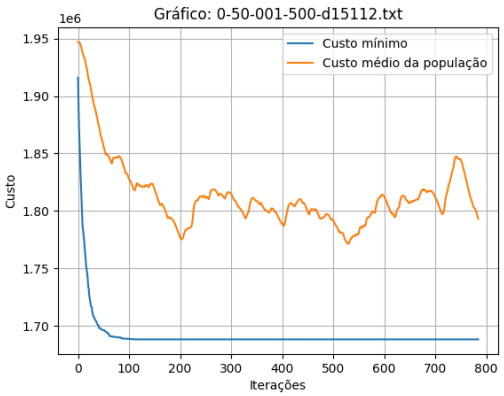

Figura 32: Visualização gráfica do resultado do operador ZX na instância d15112

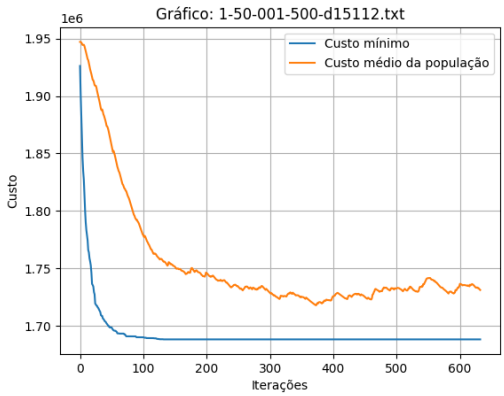

Figura 33: Visualização gráfica do resultado do operador OX na instância d15112

4.2.7 pla33810

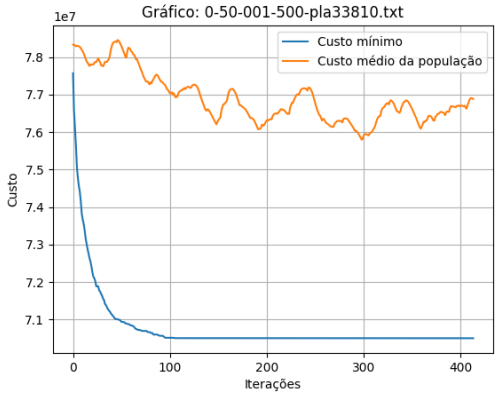

Figura 34: Visualização gráfica do resultado do operador ZX na instância pla33810

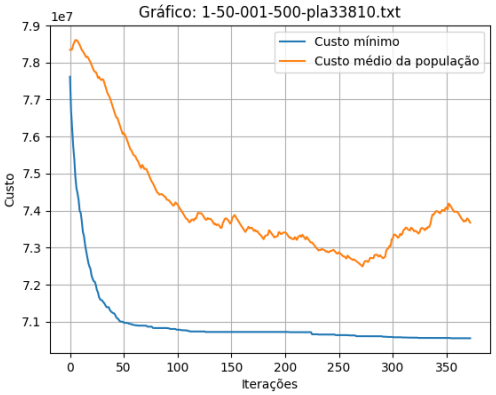

Figura 35: Visualização gráfica do resultado do operador OX na instância pla33810

4.2.8 pla85900

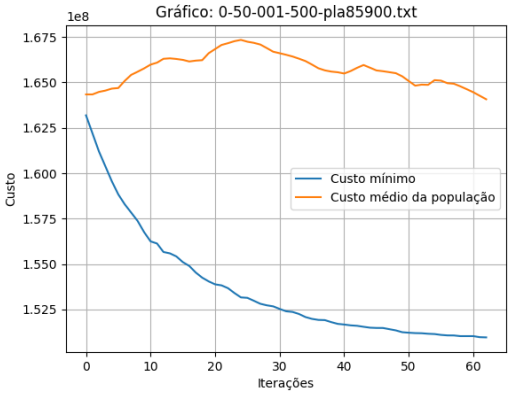

Figura 36: Visualização gráfica do resultado do operador ZX na instância pla85900

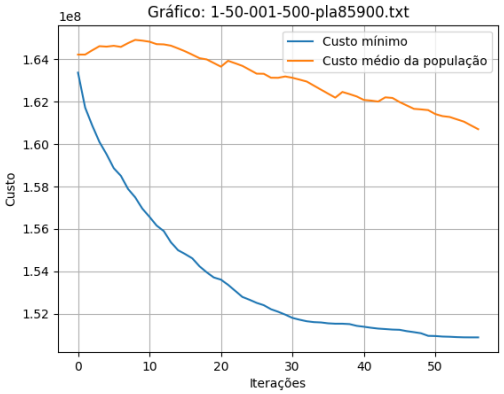

Figura 37: Visualização gráfica do resultado do operador OX na instância pla85900

4.3 Gráficos comparativos

Comparando os eixos em um mesmo gráfico para cada instância, obtemos os seguintes gráficos:

u574

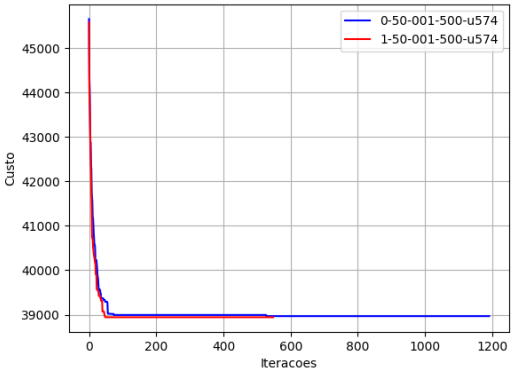

Figura 38: Comparação entre os operadores ZX (azul) e OX (vermelho) para a instância u574

pcb1173

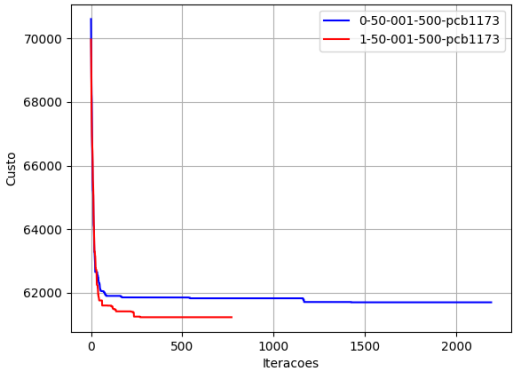

Figura 39: Comparação entre os operadores ZX (azul) e OX (vermelho) para a instância pcb1173

pr1002

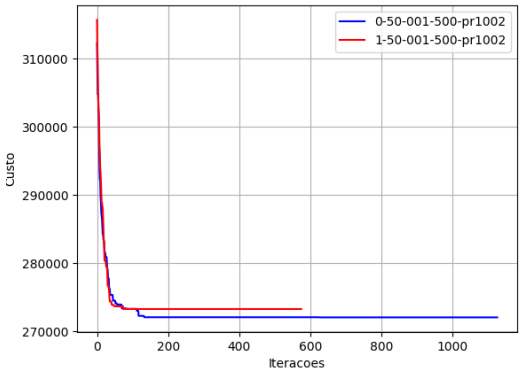

Figura 40: Comparação entre os operadores ZX (azul) e OX (vermelho) para a instância pr1002

brd14051

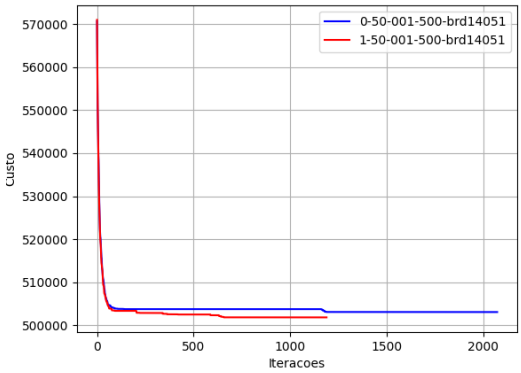

Figura 41: Comparação entre os operadores ZX (azul) e OX (vermelho) para a instância brd14051

fnl4461

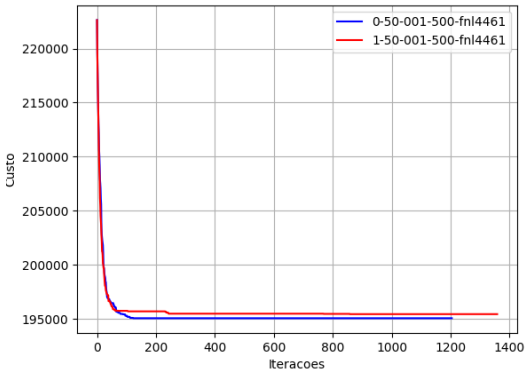

Figura 42: Comparação entre os operadores ZX (azul) e OX (vermelho) para a instância fnl4461

d15112

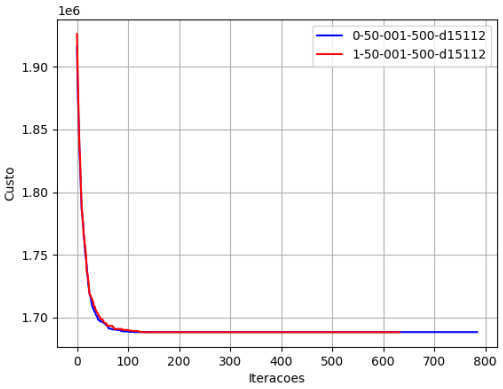

Figura 43: Comparação entre os operadores ZX (azul) e OX (vermelho) para a instância d15112

pla33810

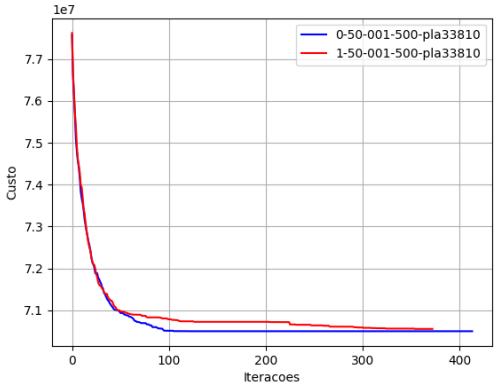

Figura 44: Comparação entre os operadores ZX (azul) e OX (vermelho) para a instância pla33810

pla85900

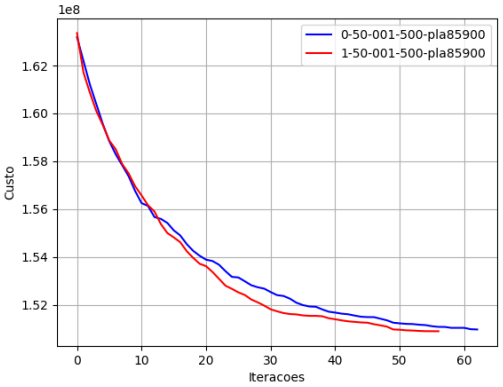

Figura 45: Comparação entre os operadores ZX (azul) e OX (vermelho) para a instância pla85900

5 Análise

Observando os resultados obtidos, nota-se que não se observa uma grande diferença dos valores GAP% obtidos por meio da utilização do operador ZX quando comparado com o OX, o que pode ser observado na tabela abaixo:

Instância	ZX	OX	ZX - OX
u574	5.59	5.52	0.07
pcb1173	8.45	7.63	0.82
pr1002	5.02	5.49	-0.47
brd14051	7.18	6.92	0.26
fnl4461	6.84	7.05	-0.21
d15112	7.33	7.32	0.01
pla33810	6.74	6.83	-0.09
pla85900	6.03	5.98	0.05

Tabela 10: Diferença entre os valores GAP% obtidos pelos dois algoritmos de cruzamento utilizados

Além disso, pode-se notar que, para todas as instâncias testadas, o comportamento da melhor solução obtida é similar nas duas abordagens, porém, ao se observar o comportamento do custo médio da população, nota-se uma diferença entre elas. Para o ZX, o comportamento tende a ser um pouco mais erratico do que para o OX, o que pode ser observado nas instâncias pcb1173, brd14051, fnl4461, d15112 e pla33810.

No entanto, essa instabilidade não indicou afetar a capacidade do ZX de obter resultados similares ao OX, ficando então mais no campo de curiosidade para possíveis experimentações posteriores.

6 Conclusão

Neste trabalho, foram implementados diversos módulos que compõem um algoritmo genético para se analisar a eficiência desta meta-heurística quando comparada a outros métodos aproximativos. O código implementado foi escrito na linguagem C e herdou módulos da avaliação anterior da disciplina, e a execução dos testes foi dividida entre os dois integrantes da equipe de modo a paralelizar o tempo de execução.

Os módulos foram implementados e alguns parâmetros de teste foram definidos para realizar a experimentação. Notou-se que, para o ZX, o valor de alpha não produziu grande influência no resultado obtido. Por sua vez, a mutação foi testada e notou-se que, para os testes realizados, valores superiores a 5% geraram resultados piores, enquanto que valores entre 0.01% e 4% não tiveram diferenças significativas entre si.

Para o número de indivíduos, notou-se que, por mais que houve uma pequena redução no GAP% obtido ao se aumentar a população, o elevado custo computacional se torna proibitivo à medida que os ganhos são diminutivos, caindo de 6.84% com 50 indivíduos para apenas 6.5% com 10000, e levando 35.15x o tempo de processamento. E o limite do tempo de execução bruto não apresentou grandes melhoras quando foram testados os limites de 6h de execução se comparados ao limite de 24h de execução.

Com isso, foram realizados testes variando-se os parâmetros de entrada de tamanho de população, algoritmo de *crossover*, taxa de mutação e critério de parada, e após a realização de 36 testes foi escolhido o melhor conjunto de parâmetros para se executar todas as demais instâncias.

A execução das demais instâncias demonstrou que não houve diferença significativa no desempenho entre os algoritmos do ZX e OX, produzindo resultados similares que não diferiram por valores maiores que 0.82%.

O tempo total de execução foi:

- Testes preliminares: 2 dias, 15 horas, 47 minutos e 22 segundos
- Testes de parâmetros: 2 dias, 5 horas, 42 minutos e 18 segundos
- Execuções finais: 2 dias, 8 horas, 29 minutos e 15 segundos

Tempo total executando (aprox.): 7 dias e 6 horas

Quando comparado ao algoritmos testados na primeira avaliação (heurísticos), nota-se que, em média, foram conseguidos resultados tão bons quanto os melhores obtidos com as estratégias lá implementadas, porém a um custo computacional bem mais elevado. De maneira geral, os objetivos deste trabalho, que eram a implementação, análise e teste de um algoritmo genético, foram alcançados com sucesso.

7 Apêndice

7.1 Pseudocódigo operador ZX

Algorithm 1: Operador de Crossover ZX para o TSP

Input: Dois genitores $genitor1$ e $genitor2$
Output: Filho resultante do crossover

Inicializar vetores binários $bin\_genitor1$ e $bin\_genitor2$ com zeros;
Inicializar vetor $filho$ com -1;
Selecionar aleatoriamente uma cidade inicial:;
$ind\_cidade\_inicial \leftarrow rand()$ mod $dimensao$;
$cidade\_inicial \leftarrow listaDeVertices[ind\_cidade\_inicial]$;
Chamar $gerar\_conjunto\_pertencente\_regiao(\alpha, genitor1, bin\_genitor1, cidade\_inicial)$;
Chamar $gerar\_conjunto\_pertencente\_regiao(\alpha, genitor2, bin\_genitor2, cidade\_inicial)$;
$indice\_filho \leftarrow 0$, $indice\_genitor1 \leftarrow 0$, $indice\_genitor2 \leftarrow 0$;
while $bin\_genitor2[indice\_genitor2] = 0$ **do**
    $indice\_genitor2 \leftarrow indice\_genitor2 + 1$;
while $bin\_genitor2[indice\_genitor2] = 1$ **do**
    $indice\_genitor2 \leftarrow indice\_genitor2 + 1$;
while $bin\_genitor2[indice\_genitor2] = 0$ **do**
    $filho[indice\_filho] \leftarrow genitor2[indice\_genitor2]$;
    $bin\_genitor2[indice\_genitor2] \leftarrow 1$;
    $indice\_filho \leftarrow indice\_filho + 1$;
    $indice\_genitor2 \leftarrow indice\_genitor2 + 1$;
while $indice\_filho < dimensao$ **do**
    $distancia\_genitor1 \leftarrow$
     $menor\_distancia(bin\_genitor1, filho[indice\_filho - 1], indice\_genitor1, 1)$;
    $distancia\_genitor2 \leftarrow$
     $menor\_distancia(bin\_genitor2, filho[indice\_filho - 1], indice\_genitor2, 0)$;
    if $distancia\_genitor1 = \infty$ ***and*** $distancia\_genitor2 = \infty$ **then**
        break;
    if $distancia\_genitor1 \leq distancia\_genitor2$ **then**
        while $bin\_genitor1[indice\_genitor1] = 1$ ***and*** $indice\_filho < dimensao$ **do**
            $filho[indice\_filho] \leftarrow genitor1[indice\_genitor1]$;
            $bin\_genitor1[indice\_genitor1] \leftarrow 0$;
            $indice\_filho \leftarrow indice\_filho + 1$;
            $indice\_genitor1 \leftarrow indice\_genitor1 + 1$;
    else
        while $bin\_genitor2[indice\_genitor2] = 0$ ***and*** $indice\_filho < dimensao$ **do**
            $filho[indice\_filho] \leftarrow genitor2[indice\_genitor2]$;
            $bin\_genitor2[indice\_genitor2] \leftarrow 1$;
            $indice\_filho \leftarrow indice\_filho + 1$;
            $indice\_genitor2 \leftarrow indice\_genitor2 + 1$;
$filho[dimensao] \leftarrow filho[0]$ // Fecha o ciclo;
return filho;

Referências

[1] BEASLEY, D.; BULL, D. R.; MARTIN, R. R. An overview of genetic algorithms: Part 1, fundamentals. *University computing*, v. 15, n. 2, p. 56–69, 1993.

[2] KUMAR, M.; HUSAIN, D. M.; UPRETI, N.; GUPTA, D. Genetic algorithm: Review and application. *Available at SSRN 3529843*, 2010.

[3] LARRANAGA, P.; KUIJPERS, C. M. H.; MURGA, R. H.; INZA, I.; DIZDAREVIC, S. Genetic algorithms for the travelling salesman problem: A review of representations and operators. *Artificial intelligence review*, v. 13, p. 129–170, 1999.

[4] DOU, X.-A.; YANG, Q.; GAO, X.-D.; LU, Z.-Y.; ZHANG, J. A comparative study on crossover operators of genetic algorithm for traveling salesman problem. In: . c2023. p. 1–8.

[5] KURODA, M.; YAMAMORI, K.; MUNETOMO, M.; YASUNAGA, M.; YOSHIHARA, I. A proposal for zoning crossover of hybrid genetic algorithms for large-scale traveling salesman problems. In: . c2010. p. 1–6.

[6] DAVIS, L. Applying adaptive algorithms to epistatic domains. In: . IJCAI'85. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., c1985. p. 162–164.

[7] LAHJOUJI EL IDRISSI, A.; TAJANI, C.; KRKRI, I.; FAKHOURI, H. *Immune based genetic algorithm to solve a combinatorial optimization problem: Application to traveling salesman problem: Volume 5: Advanced intelligent systems for computing sciences*. 01 2019. p. 906–915.