Igor Neves Faustino
Eduardo Barbosa de Oliveira

Algoritmo Genético para o Problema do Caixeiro-Viajante

Relatório técnico de atividade prática solicitado pelo professor Juliano Henrique Foleiss na disciplina de Inteligencia Artificial do Bacharelado em Ciência da Computação da Universidade Tecnológica Federal do Paraná.

Universidade Tecnológica Federal do Paraná – UTFPR

Departamento Acadêmico de Computação – DACOM

Bacharelado em Ciência da Computação – BCC

Campo Mourão

Maio / 2018

Resumo

O projeto tem como objetivo desenvolver um algoritmo genético para resolver o problema do caixeiro-viajante, este algoritmo por sua vez possui certas exigências para maximizar seu resultado, como por exemplos, a presença ou não do elitismo entra as gerações. Serão utilizados, para teste, problemas conhecidos do caixeiro-viajante, encontrados no conjunto de dados da TSPLIB.

Palavras-chave: Caixeiro-Viajante. Algoritmo. Genético.

Sumário

1 Introdução

O algoritmo genético são algoritmos inspirados no principio da evolução das especies proposto por Charles Darwin (PACHECO et al., 1999). Este algoritmo consiste basicamente em gerar novos estados a partir de dois estados pais, simulando assim a passagem de gerações. Assim, assim como na teoria de Darwin, os melhores resultados serão propagados para as próximas gerações.

Este conceito será utilizado para o problema do caixeiro-viajante, um problema simples, porem que não é de fácil solução (HOFFMAN; PADBERG; RINALDI, 2013). Os problemas utilizados para teste podem ser encontrados na base de dados do TSPLIB (SKOROBOHATYJ,).

Ao final, será comparado diversas abordagens do algoritmo genético para assim verificar as possíveis implicações no resultado final encontrado.

2 Objetivos

- Criar um algoritmo genético para resolver o problema do caixeiro-viajante.
- Verificar quais tipos de alterações no algoritmo resultam em um melhor resultado.

3 Fundamentação

Nesta sessão são apresentadas as fundamentações necessárias para este trabalho.

3.1 Problema do caixeiro-viajante

Este é um tópico bastante abordado em computação, pois apesar de ser um problema muito simples de se compreender, é um problema muito complexo de ser resolvido.

O problema consiste em uma lista de cidades, onde, dado uma cidade inicial, precisa-se passar por todas as cidades fazendo o menor caminho possível e retorna à cidade inicial.

O problema do caixeiro-viajante (PCV) é um problema NP-Difícil em otimização combinatória, ou seja, sendo PCV um problema NP-Difícil, todo problema L em NP pode ser reduzido em tempo polinomial para PCV.(NP-HARDNESS, 2018)

Na teoria da complexidade computacional, a versão de decisão do PCV, que considera um grafo que decide o caminho à ser tomado, tendo como base a menor distância, o problema se torna NP-Completo,(TRAVELLING. . . , 2018) ou seja, um problema que é NP e NP-Difícil ao mesmo tempo.

3.2 Algoritmo Genético

Um algoritmo genético é uma busca heurística baseada no evolucionismo de Charles Darwin. Este algoritmo simula o processo seleção natural, ou seja, os indivíduos que melhor se adaptam são selecionados para reprodução com o intuito de ter-se uma nova geração melhor que a anterior. (MALLAWAARACHCHI, 2017)

O processo de seleção natural começa selecionando os indivíduos que tem o coeficiente de adaptação (fit) melhor. Eles vão produzir características que serão passadas para as gerações futuras. Se os pais tiverem um fit bom, seus filhos terão chances maiores de sobreviver. Esse processo é feito iterativamente e ao final, será encontrada a geração com os maiores fits.(MALLAWAARACHCHI, 2017)

Sendo assim, 5 fases são consideradas no algoritmo genético:

1. População inicial: é o início do algoritmo, é onde são escolhidos os membros da população. Normalmente, a escolha é feita de maneira aleatória.
2. Função de fit: nessa fase, é calculado o fit de cada membro da população.
3. Seleção: na seleção, escolheremos os pais que criaram a próxima geração. É levado em consideração o fit, ou seja, os membros da população que possuem os maiores fit, tem maiores chances de serem escolhidos.
4. Reprodução: é a criação da próxima geração. Escolhe-se um algoritmo para a reprodução.
5. Mutação: para evitar a estagnação, temos a mutação de um filho, ou seja, alteramos uma característica do filho gerado. Normalmente temos uma taxa de mutação para definir quais as probabilidades de acontecer a mesma.

4 Materiais

Para a realização deste projeto serão utilizados os seguintes materiais:

- Notebook Intel® Core™ i7-2670QM CPU @ 2.20GHz x 8, com 8 GB de memoria RAM
- Sistema operacional Manjaro Linux 17.1.8.
- Python 2.7.15

5 Decisões de projeto

Nesta sessão será apresentado todas as partes do algoritmo genético implementado de forma detalhada.

5.1 Reprodução

O algoritmo de reprodução, ou *crossover* é uma das partes principais para este algoritmo, com ele, é possível criar um novo estado juntando dois estados pais, simulando assim uma reprodução. Foi implementado duas versões deste algoritmo.

5.1.1 *Crossover* ordenado

O *crossover* ordenado garante que a solução gerada continuará sendo válida, uma vez que todo nó N aparece no caminho apenas uma vez, sendo assim, dada duas soluções p1 e p2, para K $>$ 0. Foi seguido o seguinte pseudo-código demonstrado no Listing 1:

```
1 r = p1
2 p = lista da posi  o de k elementos aleat rios em ordem
3 s = subconjunto com os k elementos de p1 correspondentes as
    posi  es na lista p
4 p_ord =  ndices  dos elementos de s ordenados em rela  o a suas
    posi  es em p2
5 para cada  ndice  i dos elementos de s:
6 r[p[i]] = s[p_ord[i]]
```

Listing 1: Pseudo-código do *crossover* ordenado (FOLEISS, 2018)

Tendo esse pseudo-código como base, implementamos a função de *crossover*, demonstrada no Listing 2

```
1  def crossover(p1, p2):
2      # cria um filho
3      filho = copy.copy(p1)
4
5      # inicia p com um uma quantidade aleat ria de n meros
          aleat rios e ordena
6      p = random.sample(range(0, len(p1)), random.randint(1, len(p1
          )-1))
7      p.sort()
8
9      s = []
10
11     # para cada elemento de p.. inserir em s o elemento
          correspondente em p1
12     for i in p:
```

```
13          s.append(p1[i])
14
15      p_ord = []
16
17      # para cada elemento de p2.. se este aparece em s.. ent o
           seu  ndex  em s eh inserido ao final de p_ord
18      for i in p2:
19          if i in s:
20              p_ord.append(s.index(i))
21
22      for i in range(len(s)):
23          filho[p[i]] = s[p_ord[i]]
24
25      return filho
```

Listing 2: Código do *Crossover* Ordenado

Observando o código acima pode ser notado que o pseudo-código foi seguido, porém, sofrendo leves alterações.

Na linha 6, foi gerado o vetor p com a função sample da biblioteca random, que tem como objetivo gerar valores aleatórios dentro de um intervalo sem repeti-los. Sendo assim, o primeiro parâmetro da função é o intervalo que vão ser gerados os números aleatórios, e o segundo parâmetro é a quantidade de números que vão ser gerados. Logo após isso,na linha 7, foi usado a função sort para ordenar o vetor, como é requisitado no pseudo-código.

O restante do código foi implementado assim como define o pseudo-código.

5.1.2 *Crossover* alternativo

Para implementar esta versão do *crossover*, foi utilizado como base no Listing 3

```
1 def crossover(p1, p2):
2     r = copy.copy(p1)
3     corte = random.randrange(1, len(p1))
4     for i in range(corte, len(p1)):
5         if (p1[i] not in p2[0:i]) and (p2[i] not in p1[0:i]):
6             r[i] = p2[i]
7     return r
```

Listing 3: Código base para o *crossover* alternativo (FOLEISS, 2018)

Porem, este código não é completo, falhando para alguns casos de testes, como por exemplo para o caso representado na Figura 1, Neste caso, ao passar o valor 6 que esta na posição 3, este valor fica duplicado na posição 5. O problema está no fato de não

ser possível trocar os valores da posição 5, o que mantém o valor 6 repetido, quebrando assim a propriedade do caixeiro-viajante.

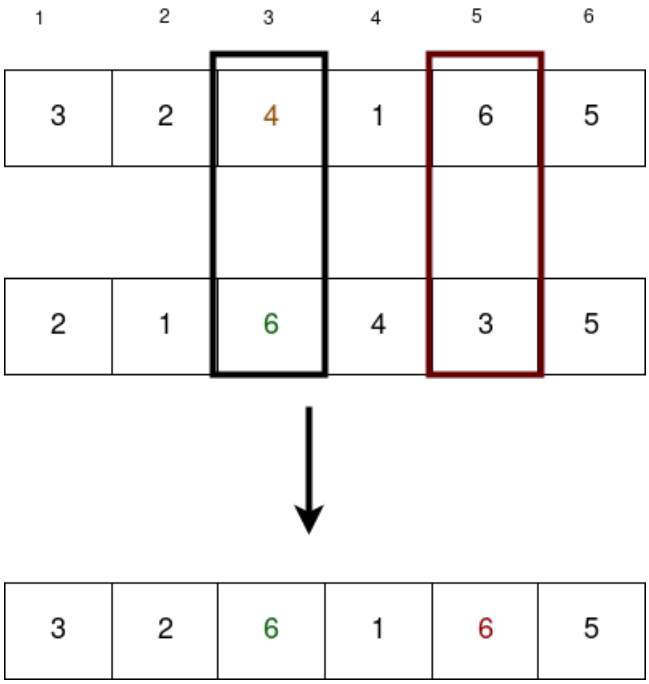

Figura 1: Contra exemplo da função do *crossover* alternativo.

Logo, para evitar este caso, foi utilizado uma adaptação, a qual evita a repetição de valores no vetor resultante. O Listing 4 demonstra a versão utilizada deste *crossover*.

```
1  def crossover_alternativo(p1, p2):
2      # cria um filho
3      filho = copy.copy(p1)
4
5      # define um corte de maneira aleatoria
6      corte = random.randrange(1, len(p1))
7
8      for i in range(corte, len(p1)):
9          # verifica se P2[i] esta entre as posicoes 0 e corte do
               vetor filho
10         if (p2[i] not in filho[0:i]):
11             # caso nao esteja
12
13             # o valor de filho[i] substitui a posicao que tem o
                   valor igual
14             # a p2[i] no vetor filho
15             filho[filho.index(p2[i])] = filho[i]
16
17             # agora filho[i] recebe p2[i]
```

```
18                 filho[i] = p2[i]
19         return filho
```

Listing 4: Código base para o *crossover* alternativo

Neste novo código, ao substituir um valor por um de p2, o valor a ser duplicado é inserido no lugar do valor igual ao que vai ser inserido por p2, ou seja, um valor nunca é duplicado pois, o valor sairia do vetor substitui o valor repetido, como visto na Figura 2.

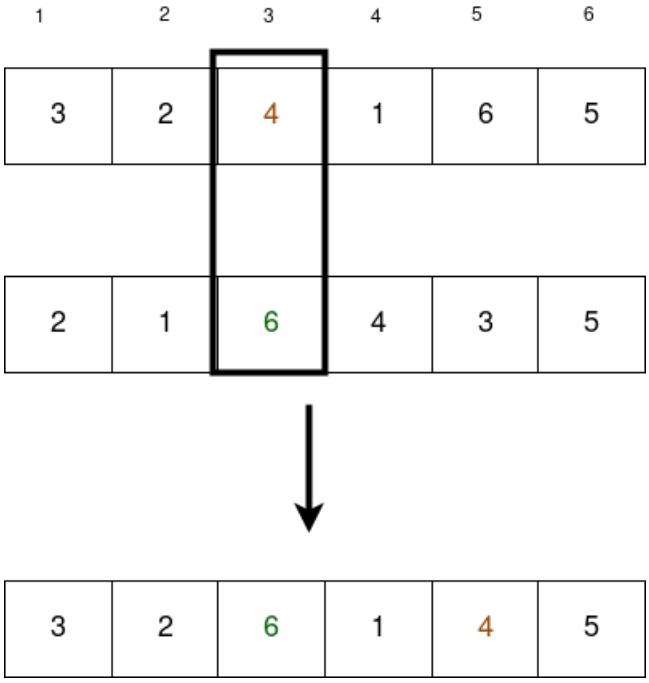

Figura 2: Exemplo do comportamento da função de *crossover* alternativa.

A figura demonstra apenas uma iteração do algoritmo, nota-se que ao invés de apenas substituir o valor 3 na posição 3 pelo valor 6 na mesma posição no segundo vetor, o valor 3 é replicado para a posição 5, evitando assim que exista um numero repetido.

5.2 Mutação

O processo de mutação modifica um individuo alterando o seu estado, com isto, é possível ter uma maior variabilidade de indivíduos. Para este projeto, dois tipos de mutação foram implementados.

A escolha de qual mutação vai ser escolhida ficou definida por um parâmetro, que pode ser definido como 1, que é o caso onde utilizará a mutação 1. Ou pode ser definido como 2, que é o caso onde utilizará a mutação 2. Ainda, pode ser definido como qualquer outro valor (inclusive não definir nenhum valor), onde ele irá sortear qual das duas mutações utilizará.

O Listing 5 demonstra como isso foi implementado:

```
def mutacao(x, id = 3):
    if id == 1:
        id = 0
    elif id == 2:
        id = 1
    else:
        id = random.random()
```

Listing 5: Seleção do tipo de mutação

Observando o código, pode ser observado que na definição da função, o valor padrão para id, que se refere a qual mutação utilizará, é diferente de 1 ou 2, ou seja, por padrão, a função sorteará qual mutação irá utilizar.

Note ainda, que após iniciarmos a função, o id é convertido pra uma distribuição probabilística, ou seja, é possível utilizar a mesma variável mesmo se for o caso de ser sorteado aleatoriamente ou se for definido por parâmetro.

5.2.1 Mutação 1

A mutação 1 é bem simples. Ela consiste em sortear dois índices aleatórios e trocá-los de posição.O Listing 6 demonstra como foi implementado essa mutação:

```
if id <0.5:
        # mutacao 1
        # troca duas posicoes de lugar

        index = random.sample(range(0, len(x)), 2)
        aux = x[index[0]]
        x[index[0]] = x[index[1]]
        x[index[1]] = aux
```

Listing 6: Código Mutação 1

Nota-se que no inicio é realizado uma verificação de qual o id, que será verdadeiro tanto quando o valor de id vir por parâmetro e posteriormente for convertido, ou quando for sorteado esse valor (chance de 50%).

Após isso, dois índices são sorteado, novamente utilizando o sample para não repetir os valores. Logo a seguir, os valores correspondentes aos índices sorteados são trocados de posição.

5.2.2 Mutação 2

A mutação 2 é um pouco mais complexa que a mutação 1, porém com uma ideia parecida. Ela consiste em sortear um índice, após isso, verifica se o valor à qual o índice aponta é par ou ímpar.

Se for par, ele irá procurar no vetor o próximo valor par. Após ter encontrado, irá armazenar seu índice numa variável auxiliar. Feito isso, irá trocar o valor de onde os índices (sorteado e encontrado) apontam.

Se for ímpar, fará igual ao se fosse par, porém, irá buscar o próximo ímpar.

O Listing 7 demonstra como foi implementado essa mutação:

```
1  else:
2          # mutacao 2
3
4          index = random.randrange(0, len(x))      #sorteia uma
              posicao aleatoria do vetor
5          aux = index+1                            #armazena a
              proxima posicao da sorteada
6          if x[index] % 2 == 0:                    #se o valor da
              posicao sorteada for par
7              while x[aux % len(x)] % 2 != 0:      #enquanto a
                  proxima posicao for impar (logica de lista circular
                  )
8                  aux += 1
9              aux = aux%len(x)                     #volta o aux para
                   posicao de vetor
10             aux2 = x[aux]                        #
11             x[aux] = x[index]                    #Troca indice
                  sorteado com o proximo valor par encontrado
12             x[index] = aux2                      #
13         else:                                    #se o valor da
              posicao sorteada for impar
14             while x[aux % len(x)] % 2 == 0:      #enquanto a
                  proxima posicao for par (logica de lista circular)
15                 aux += 1
16             aux = aux%len(x)                     #Volta o aux para
                   posicao de vetor
17             aux2 = x[aux]                        #
18             x[aux] = x[index]                    #Troca indice
                  sorteado com o proximo valor impar encontrado
19             x[index] = aux2                      #
20     return x
```

Listing 7: Código mutação 2

Logo no início do código pode ser visto o '*else*', seguindo ainda a distribuição probabilística, essa mutação tem 50% de chance de ser escolhida (100% - 50% da mutação 1 = 50%) por sorteamento, ou ainda, pode ser escolhida por parâmetro.

Sendo escolhida, a mutação 2 irá sortear um índice do vetor, como feito na linha 4. Após isso, é pego o índice imediatamente após o mesmo, ou seja, é somado o valor 1 ao índice. Depois, é verificado se o valor à qual o índice aponta é par ou ímpar (linhas 6 e 13). Após isso, as duas execuções, tanto para ímpar, quanto para par, será feita de maneira parecidas. Irá buscar o próximo valor par/ímpar no vetor, armazenará o índice dele e fará a troca.

Nota-se que na implementação foi utilizada a lógica de uma lista circular, pois o sorteio de um valor poderia cair na última posição e quando fosse buscar, não teria o restante do vetor para buscar. Para evitar isso, quando ele chega ao final do vetor, ele volta pra buscar no início do mesmo, ou seja, implementando a lógica de lista circular. Para isso, foi utilizado o resto da divisão pelo tamanho do vetor, que sempre resulta e, um índice válido.

5.3 Função Fitness

A função *fitness* é a função que será o maximizada pelo algoritmo genético, neste caso é a soma total da distancia entre todas as cidades. A função descrita no Listing 8 recebe um vetor contendo a ordem de visita de todas os pontos e uma matriz contendo todas as distancias. A função retorna o custo de tomar esse determinado caminho, levando em conta o custo de retornar do ultimo ponto para o inicial, fechando assim o ciclo.

```
1 def fitness(vet, matriz_distancias):
2     soma = 0
3     for i in range(0, len(vet)-1):
4         soma += matriz_distancias[vet[i]][vet[i+1]]
5     return soma + matriz_distancias[vet[0]][vet[-1]]
```

Listing 8: Implementação da função fitness.

5.4 Algoritmo Genético

O algoritmo genético é a principal parte da implementação, ele consiste basicamente na simulação de um ambiente natural, onde os indivíduos se reproduzem e geram uma nova população que substitui a geração antiga. Para este algoritmo, cada individuo é uma possível solução do problema do caixeiro-viajante, sendo que as melhores soluções possuem maior chance de gerar novos filhos. O Listing 9 demonstra a principal parte desta implementação.

```
1     # sera processado geracoes ate ocorrer um caso onde nao
          exista
2     # mais melhorias por x geracoes
3     while n_maximo_sem_mudancas < estagnacao:
```

```
4           p_nova = []
5           print(n_maximo_sem_mudancas)
6
7           # para cada individuo na populacao
8           for i in range(len(pop_inicial)):
9               # escolhe dois pais
10              x = random_select(pop, f, matriz)
11              y = random_select(pop, f, matriz)
12
13              # realiza a reproducao
14              if use_crossover_alternativo:
15                  novo = crossover_alternativo(x, y)
16              else:
17                  novo = crossover(x, y)
18
19              # realiza a mutacao
20              r = random.randrange(0, 100)
21              if r < tx_mutacao:
22                  mutacao(novo, id_mutacao)
23
24              # insere filho na nova pop
25              p_nova.append(novo)
```

Listing 9: Trecho da função principal do algoritmo genetico

O trecho descrito demonstra a criação de uma nova população. A qual consiste em um novo estado criado com base nos estados de seus pais, apos a criação de cada filho, este ainda possui uma pequena chance de sofrer um processo de mutação, o que transforma seu estado atual, causando assim uma maior variabilidade ao resultado.

5.5 Condição de Parada

A condição de parada implementada foi o numero de gerações sem ocorrer uma melhora no resultado obtido, ou seja, apos 'n' iterações sem melhoras, o algoritmo será terminado.

5.6 Elitismo

O elitismo consiste em manter indivíduos de uma população para outra, ou seja, aproveitar os melhores indivíduos da população original para assim montar uma nova população apenas com os indivíduos mais qualificados entre os filhos e seus pais. A implementação do elitismo pode ser visto no Listing 10

```
    ...
    if elitismo:
        # ordena uma copia do fit da populacao
        # desta forma o vertor ficara com ordenado do
        # pior para o melhor valor
        pop_sort = copy.copy(fit)
        pop_sort.sort(reverse=True)  # maior para o menor

        # cria um vetor com o fitness dos filhos
        filhos = []
        for i in p_nova:
            filhos.append(f(i, matriz))

        # para cara fit da pop, verificar se existe um
        # fit melhor na nova pop
        for pai in pop_sort:
            menor = float('inf')
            for filho in filhos:
                if pai > filho:
                    if (p_nova[filhos.index(filho)] not in pop):
                        if filho < menor:
                            # Menor e o melhor filho que consegue
                            # substituir o pai
                            # e que ainda nao tenha sido inserido
                                no vetor
                            menor = filho
            if menor != float('inf'):
                # Por fim, o pai e substituido pelo melor filho
                pop[fit.index(pai)] = p_nova[filhos.index(menor)]
            # caso o pai seja melhor que os filhos, o mesmo e
               mantido
            # para a proxima geracao
    ...
```

Listing 10: Implementação do elitismo.

O comportamento deste algoritmo é bem simples, para cada elemento dá população pai, é selecionado um elemento melhor na população filha, caso nenhum filho seja melhor que um determinado individuo pai, este individuo será mantido na população final. Caso não exista o elitismo, a população pai é totalmente substituída por seus filhos.

6 Avaliação experimental

O algoritmo implementado foi utilizado para os seguintes problemas, encontrados na biblioteca TSPLIB.

- a280
- berlin52
- kroA100
- kroC100
- kroD100
- pr76

É importante notar que quanto maior for o tempo de execução, ou seja, uma condição de parada maior, melhor será o resultado obtido. Para este exemplo, todos os teste foram rodados com os seguintes parâmetros: estagnação igual a 15, tamanho da população igual a 200 e taxa de mutação igual a 5. Para cada teste, foi extraído a media de 10 execuções, as quais podem ser conferidas na Tabela 1

Tabela 1: Resultado dos testes contendo media e porcentagem de erro

Conjunto de Teste	Media dos Resultados	Numero medio de Gerações	Dimensão do Problema	Elitismo	Tipo Crossover	Mutação	Porcentagem de Erro (%)
a280	22260.98415713	425,4	280	Sim	ordenado	1	763,16340442
a280	26697,82403129	125,2	280	sim	ordenado	2	935,200621609
a280	11297,53518195	1416,25	280	sim	alternativo	1	338,058750754
a280	23167,2295083429	330,28	280	sim	alternativo	2	798,302811491
a280	30721,52674751	25,3	280	não	ordenado	1	1091,2185633
a280	30459,2193781625	26,875	280	não	ordenado	2	1081,047668793
a280	30516,3462817714	30	280	não	alternativo	1	1083,262748421
a280	30666,3312308571	27	280	não	alternativo	2	1089,078372658
berlin52	10274,880400467	281,3	52	Sim	ordenado	1	36,235486615
berlin52	12155,619177997	194,5	52	sim	ordenado	2	61,172357173
berlin52	11842,69168642	271,3	52	sim	alternativo	1	57,023225755
berlin52	14903,11845882	132,1	52	sim	alternativo	2	97,601676728
berlin52	23384,22190902	24,1	52	não	ordenado	1	210,053326823
berlin52	23419,72800794	29,6	52	não	ordenado	2	210,524105117
berlin52	23373,98442872	28,1	52	não	alternativo	1	209,917587228
berlin52	23221,02569393	31,2	52	não	alternativo	2	207,889494748
kroA100	66527,0533268444	484,5	100	Sim	ordenado	1	212,597750807
kroA100	97450,61304765	157,3	100	sim	ordenado	2	357,901574324
kroA100	58508,33089111	517,4	100	sim	alternativo	1	174,919325679
kroA100	84882,74800871	204	100	sim	alternativo	2	298,847608348
kroA100	139856,068042167	25	100	não	ordenado	1	557,156602021
kroA100	140315,406329714	32,43	100	não	ordenado	2	559,314943754
kroA100	139091,397637286	30,14	100	não	alternativo	1	553,56356375
kroA100	139214,6843552	29,2	100	não	alternativo	2	554,142864182
kroC100	84873,21498038	341,8	100	Sim	ordenado	1	309,047255195
kroC100	89703,22934047	265,9	100	sim	ordenado	2	332,32555468
kroC100	52933,69682341	532,7	100	sim	alternativo	1	155,114448038
kroC100	83896,4687593222	228,33	100	sim	alternativo	2	304,339817626
kroC100	137776,6700671	27,4	100	não	ordenado	1	564,015952899
kroC100	137752,9809995	32,8	100	não	ordenado	2	563,901783216
kroC100	138063,0855092	29,6	100	não	alternativo	1	565,396334807
kroC100	137953,9437126	28,8	100	não	alternativo	2	564,870324896
kroD100	85934,29659586	251,9	100	Sim	ordenado	1	303,561081036
kroD100	79098,97131399	207,2	100	sim	ordenado	2	271,461309824
kroD100	50613,41958964	598,7	100	sim	alternativo	1	137,688642762
kroD100	80690,7433157429	200,14	100	sim	alternativo	2	278,936523508
kroD100	135522,5850628	24,6	100	não	ordenado	1	536,435545519
kroD100	133795,4374245	29,5	100	não	ordenado	2	528,324586383
kroD100	132525,0932008	32,9	100	não	alternativo	1	522,358848506
kroD100	132683,7009859	41	100	não	alternativo	2	523,103695811
pr76	227130,3593046	385,5	76	Sim	ordenado	1	109,996726398
pr76	276041,9761237	203,4	76	sim	ordenado	2	155,218683719
pr76	204348,849239429	511,3	76	sim	alternativo	1	88,933744986
pr76	289910,366508429	170,14	76	sim	alternativo	2	168,040908762
pr76	471253,867226286	21,6	76	não	ordenado	1	335,704719188
pr76	479247,601092143	36	76	não	ordenado	2	343,09544383
pr76	473222,402051143	36,3	76	não	alternativo	1	337,524757118
pr76	468463,636154143	30,43	76	não	alternativo	2	333,124969863

Durante a execução dos teste, foi feito o *plot* de um gráfico para visualizar os resultados obtidos ao longo de todas as gerações. Com isto, dois comportamentos distintos foram observados: com e sem elitismo. A figura 3 exibe o gráfico obtido quando o é utilizado o elitismo para selecionar a nova população,

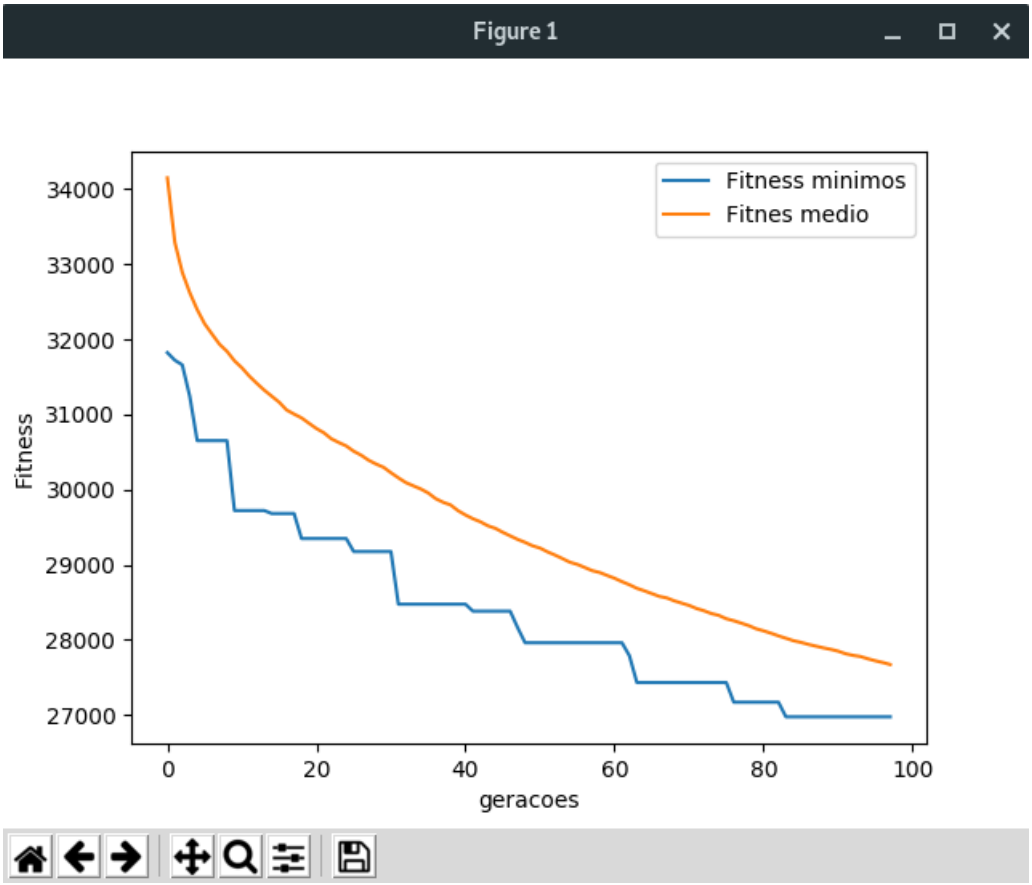

Figura 3: Gerações utilizando o elitismo.

É possível notar que o gráfico, ao passar das gerações, tende a descer, melhorando sempre o resultado obtido, isto ocorre por haver a seleção dos melhores indivíduos para a próxima geração.

A Figura 4 demonstra uma situação onde não ocorreu o elitismo, é possível reparar que, como não ocorre uma seleção dos indivíduos, não há uma melhora constante ao passar das populações. Logo, graças a condição de parada estabelecido, a execução do algoritmo tende a ser menor, sem passar por muitas gerações.

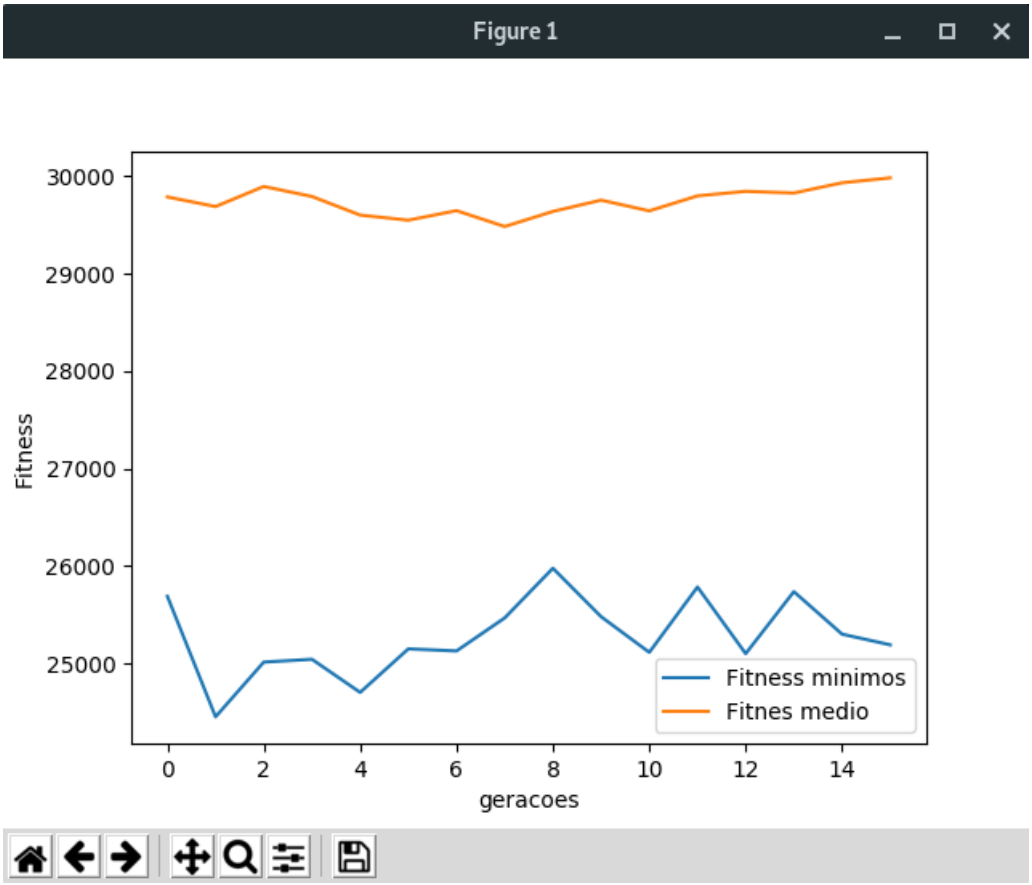

Figura 4: Comportamento das gerações sem utilizar elitismo.

Como pode ser observado, em sua grande maioria, os resultados obtidos não foram satisfatórios, isto se deve ao fato do baixo valor de estagnação, podendo ser resolvido aumentando o valor utilizado como condição de para, porem, aumentando assim o numero de gerações e consequentemente o tempo de execução de cada teste.

Pode também ser notado, que o *crossover* alternativo e a mutação 1 foram mais efetivo em conjunto de dados maiores, resultando em um taxa de erro significantemente menor para o conjunto a280. Para conjuntos menores, a maior diferença a ser analisada é o uso do elitismo, que em todos os casos testados melhorou significantemente o resultado final obtido.

7 Conclusões

Com o desenvolvimento deste presente trabalho, foi possível observar o funcionamento de um algoritmo genético para solucionar um problema real. Também foi possível observar em como pequenas mudanças podem afetar o resultado obtido. Com isso, foi concluído como um algoritmo genético depende da quantidade de gerações para aprimorar seu resultado e em como a presença do elitismo pode facilitar a convergência do algoritmo para um solução mais aceitável.

8 Referências

FOLEISS, J. H. *Trabalho 1: Problema do Caixeiro Viajante com Algoritmo Genético*. [S.l.], 2018. Especificação do Trabalho 1. Citado 2 vezes nas páginas 6 e 7.

HOFFMAN, K. L.; PADBERG, M.; RINALDI, G. Traveling salesman problem. In: *Encyclopedia of operations research and management science*. [S.l.]: Springer, 2013. p. 1573–1578. Citado na página 4.

MALLAWAARACHCHI, V. *Introduction to Genetic Algorithms - Including Example Code*. Towards Data Science, 2017. Disponível em: <https://towardsdatascience.com-/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3>. Citado na página 5.

NP-HARDNESS. Wikimedia Foundation, 2018. Disponível em: <https://en.wikipedia-.org/wiki/NP-hardness>. Citado na página 4.

PACHECO, M. A. C. et al. Algoritmos genéticos: princípios e aplicações. *ICA: Laboratório de Inteligência Computacional Aplicada. Departamento de Engenharia Elétrica. Pontifícia Universidade Católica do Rio de Janeiro. Fonte desconhecida*, p. 28, 1999. Citado na página 4.

SKOROBOHATYJ, G. *MP-TESTDATA - The TSPLIB Symmetric Traveling Salesman Problem Instances*. Disponível em: <http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp-/>. Citado na página 4.

TRAVELLING salesman problem. Wikimedia Foundation, 2018. Disponível em: <https://en.wikipedia.org/wiki/Travelling_salesman_problem>. Citado na página 4.