Universidade Federal do Rio Grande do Norte - UFRN
Departamento de Informática e Matemática Aplicada - DIMAp
Inteligência Artificial - DIM0613

Implementação e Avaliação de um Algoritmo Genético para o Problema das Oito Rainhas

Discente:

Hiago Mayk Gomes de Araújo Rocha

Docente:

Anne Magaly de Paula Canuto

Natal - RN
31 Março 2016

1 Introdução

O problema das 8 rainhas consiste em, dado um tabuleiro de xadrez de tamanho padrão (8 linhas e 8 colunas) e 8 peças do tipo rainha, tentar alocar essas rainhas no tabuleiro de forma que elas não se conflitem. Um conflito é gerando quando duas ou mais rainhas estão se atacando.

Este trabalho tem como objetivo apresentar a implmentação e avaliação da solução para o problema das oito rainhas usando um Algoritmo Genético.

No decorrer do trabalho serão descritos a modelagem do problema, a implementação seguindo a estrutura e especificação de um Algoritmo Genético e resultados obtidos para diferentes configurações de parâmetros algoritmo.

2 Visão geral do algoritmo

Os algoritmos genéticos baseiam-se no processo natual de evolução dos seres e são úteis para resolver problemas de busca e otimização.

"Quanto melhor um indivíduo se adaptar ao seu meio ambiente, maior será sua chance de sobreviver e gerar descendentes": este é o conceito básico da evolução genética biológica. Os Algoritmos Genéticos além de serem estratégias de gerar-e-testar muito elegante, por serem baseados na evolução biológica, são capazes de identificar e explorar fatores ambientais e convergir para soluções ótimas, ou aproximadamente ótimas em níveis globais.

A idéia básica do algoritmo é segunte: inicialmente, é gerada uma população formada por um conjunto aleatório de indivíduos que podem ser vistos como possíveis soluções do problema. Durante o processo evolutivo, esta população é avaliada. Para cada indivíduo é dada uma nota, ou índice, refletindo sua habilidade de adaptação a determinado ambiente. Uma porcentagem dos mais adaptados são mantidos, enquanto os outros são descartados (darwinismo). Os membros mantidos pela seleção podem sofrer modificações em suas características fundamentais através de mutações e cruzamento (*crossover*) ou recombinação genética gerando descendentes para a próxima geração. Este processo, chamado de reprodução, é repetido até que uma solução satisfatória seja encontrada.

Embora possam parecer simplistas do ponto de vista biológico, estes algoritmos são suficientemente complexos para fornecer mecanismos de busca adaptativo poderosos e robustos.

3 Detalhes de implementação

A implementação do algoritmo foi feita usando a linguagem de programação Python e foi baseada em um artigo publicado pela revista Omnia Exatas na edição de Janeiro/Junho de 2009. O artigo avalia e descreve a solução para o problema para N-Rainhas, por esse motivo foram feitas adaptações para o problema das 8rainhas, que é o objetivo deste trabalho.

Nas subseções seguintes apresentaremos os detalhes da implementação seguindo as etapas de um Algoritmo Genético.

3.1 Representação cromossomica

A representação cromossomica de uma configuração é dada por um vetor de inteiros (lista de inteiros em Python) com tamanho 8 em que seus índices representam as linhas onde a rainha está alocada e seus elementos, que são números gerados aleatóriamente no intervalo entre 0 e 7, representam as colunas.

Essa representação garante que não exista conflito entre as rainhas tanto nas linhas quanto nas colunas do tabuleiro. Em uma abordagem ingênua onde se pode alocar rainhas em qualquer lugar do tabuleiro independente de haver conflito de linhas ou de colunas, teriamos que examinar $64!/56! = 178,462,987,637,760$ possíveis maneiras de colocar 8 peças nas 64 casas do tabuleiro e após isso teriamos que checar se existem rainhas que se atacam. Com a representação cromossomica que modelamos consegue-se restringir o espaço de busca considerando apenas as permutações dos números entre 0 e 7 como uma configuração do problema. Note que agora precisamos gerar $8! = 40320$ configurações e testar, apenas nas diagonais do tabuleiro, se as rainhas de cada coluna colocadas na linhas dada pela permutação atacam umas as outras.

A Figura 1 dar uma representação gráfica de como é feita a representação cromossomica.

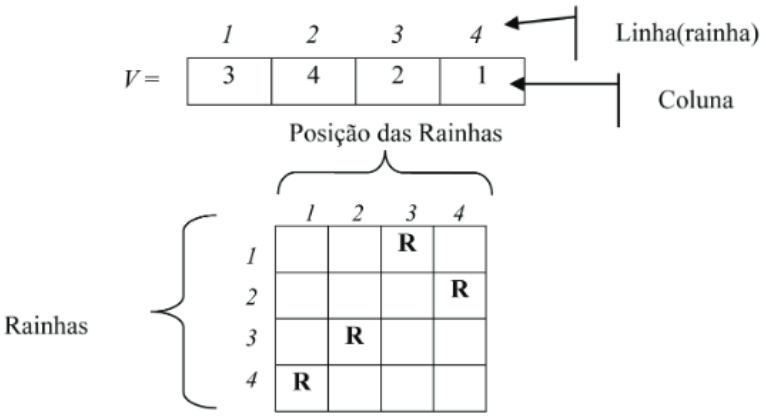

Figura 1: Representação cromossomicas das configurações do problema.

A representação da população é dada por uma matriz (lista de listas de inteiro em Python) cujas linhas representam cada elemento da população. A figura 2 apresenta uma representação gráfica da modelagem da população.

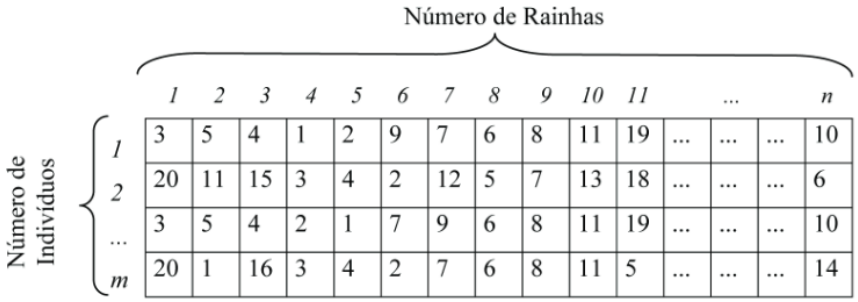

Figura 2: Representação da população é dada por uma matriz onde cada linha representa uma configuração. Essa representação é generica para o proplema das n-rainhas, porém aqui restringimos a apenas 8 rainhas.

3.2 Função objetivo

O cálculo da função objetivo é dado pela verificação e contagem da quantidade de conflitos existentes entre as rainhas nas diagonais do tabuleiro, já que como explicado anteriormente, os conflitos nas linhas e nas colunas são tratados pela própria representação cromossomica. Para a realização desse cálculo são criadas duas matrizes 8 por 8, uma denominada de matriz diagonal positiva e a outra denominada de matriz diagonal negativa. A matriz diagonal positiva é usada na verificação de conflitos nas diagonais positivas do tabuleiro e cada elemento dela é calculado pela subtração do índice referente a linha pelo índice referente a coluna. Analogamente, a matriz diagonal negativa é usada na verificação de conflitos nas diagonais negativas do tabuleiro e cada elemento dela é calculado pela soma do índice referente a linha pelo índice referente a coluna. Com isso, obten-se para cada diagonal, em ambas matrizes, uma constrante que a representa.

A verificação de conflitos em cada configuração é feita por inspeção dos seus elementos: pega-se um índice do vetor (lista em Python) de configuração e o elemento a quem ele se refere (linha e coluna de ambas matrizes respectivamente) e verifica nas matrizes diagonais se existem configurações que se referem as mesmas constantes, contante essa que representa uma diagonal.

O somatório da quantidade de elementos que se referem a uma mesma constante na matriz diagonal positiva indica os conflitos existentes entre as rainhas apenas nas diagnais positivas do tabuleiro. Da mesma forma, o somatório da quantidade de elementos que se referem a uma mesma constante na matriz diagonal negativa indica os conflitos existentes entre as rainhas apenas nas diagnais negativas do tabuleito. A soma desses dois somatórios indica o total da quantidade de conflitos de uma determinada configuração. As Figuras 3 e 4 apresentam graficamente as matrizes diagonais positiva e negativa respectivamente e como é feito o cálculo dos conflitos.

1	**0**	**-1**	**-2**	**-3**
2	**1**	**0**	**-1**	**-2**
3	**2**	**1**	**0**	**-1**
4	**3**	**2**	**1**	**0**

Figura 3: Representação gráfica da matriz diagonal positiva.

3.3 Seleção

A seleção é feita pelo método de torneio que consiste na escolha de dois elementos da população que possuam a melhor aptidão, que para o nosso caso, é o menor número de conflitos.

3.4 Recombinação ou *crossover*

Para a recombinação foi usado a o método *Partially Matched Crossover*, mais conhecido como PMX. No processo de recombinação, escolhe-se aleatóriamente dois pontos (índices do vetor) das configurações selecionadas na etapa de seleção, os mesmo pontos para ambas configurações. Esses

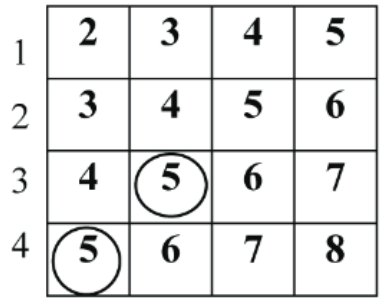

Figura 4: Representação gráfica da matriz diagonal negativa.

pontos representam o intervalo onde ocorrerá a troca de genes, logo, um receberá todos os genes do outro naquele mesmo intervalo. Essa troca é feita também em ambas configurações. Logo após é estabelecida uma relação simétrica e transitiva entre os números (genes dos cromossomos) que estão dentro da região onde foi realizada a troca com o objetivo de substituir os números repetidos que estão fora da região selecionada. Após realizada a substituição obtem-se dois descendentes, os quais terão a quantidade de colisões calculadas pela função objetivo e o que possuir a melhor quantidade de colisões é o que substituirá alguma configuração da população na etapa de substituição da população.

A Figura 5 apresenta os passos aqui descrito.

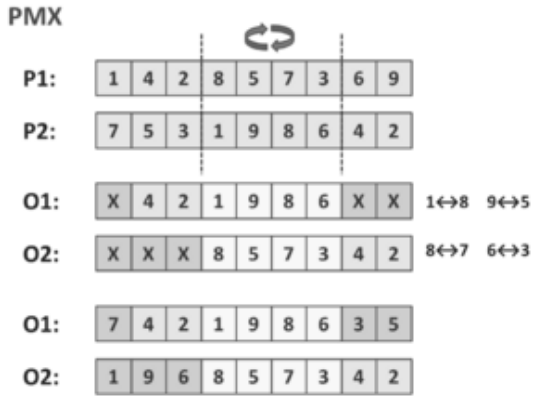

Figura 5: Passos seguidos no método PMX.

3.5 Mutação

A mutação é dada de maneira simples. Apenas se escolhe alealoriamente duas rainhas e troca suas posições referentes a colunas no tabuleiro. Em termos se implementação, isso se refere a escolher dois índices de um vetor que representa uma configuração e trocar os seus respectivos números. A Figura 6 e 7 ilustra como é feita essa troca.

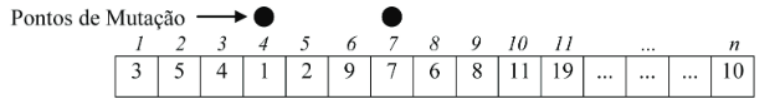

Figura 6: Escolha dos números a serem trocados no processo de mutação.

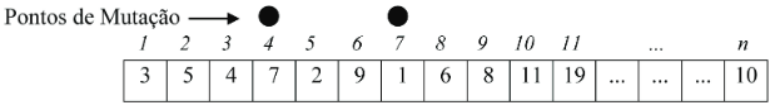

Figura 7: Após se realizado a troca dos números.

3.6 Substituição da população

Somenete é aproveitado as configurações geradas nas etapas de seleção, recombinação e mutação se ela possuir uma quantidade de conflitos menor do que a configuração da população que possui a maior quantidade de conflitos. Na solução não estamos restringindo a geração de soluções repetidas, logo, pode existir soluções ótimas para o problema com configurações repetidas, ou seja, podemos obter um número de soluções ótimas muito além da quanntidade máxima de soluções para o problema das 8 rainhas que é de 92 soluções distintas.

4 Resultados obtidos

Nessa seção apresentamos os testes realizados para diferentes valores do tamanho da população, quantidade de iterações, taxa de mutação e taxa de cruzamento. É importante observar que a variação desses parâmetros pode influênciar bastante no resultado.

O tamanho da população influencia no desempenho global e na eficiencia dos Algoritmos Genéticos. Com uma população pequena o desempenho pode cair, pois deste modo a população fornece uma pequena cobertura do espaço de busca do problema. Já com uma população grande obtem-se uma corbetura representativa do domínio do problema, além de prevenir convergências prematuras para soluções locais ao invés de globais. No entanto, para se trabalhar com grandes populações, são necessários maiores recursos computacionais, ou que o algoritmo trabalhe por um período de tempo muito maior. Pode-se observar que nos caso de teste para população acima de 1000 o algortimo demorou cerca de 10 minutos para gerar os resultados.

A quantidade de iterações influencia no tempo de execução do algoritmo e também, em conjunto com o tamanho da população, na convergência para uma solução ótima do problema. Com uma quatidade pequena de iterações pode-se chegar uma quantidade pequena de soluções ótima ou até mesmo não conseguir chegar a nem uma solução. Já com uma quantidade grande de iterações pode-se chegar a varias soluções ótimas porém com um maior tempo de execução.

Para a taxa de mutação ten-se que, uma baixa taxa de mutação previne que uma dada posição fique estagnada em um valor, além de possibilitar que se chegue em qualquer ponto do espaço de busca. Já com com uma taxa muito alta a busca se torna essencialmente aleatória.

Já Para a taxa de cruzamento, quanto maior for esta taxa, mais rapidamente novas estruturas serão introduzidas na população. Mas se esta for muito alta, estruturas com boas aptidões poderão ser retiradas mais rapido um valor alto, a maior parte da população será substituída, mas com valores muito altos pode ocorrer perda de estruturas de alta aptidão. Com um valor baixo, o algoritmo pode tornar-se muito lento.

A seguir apresentamos os resultados obtidos. Para cada configuração de parâmetros foram testados 5 vezes e calculados o tempo de execução em milisegundos e a quantidade de soluções ótimas encontradas.

Tamanho da população: 50
Quantidade de iterações: 50
Taxa de mutação: 50
Taxa de cruzamento: 50

Tempo de excução: 0.147876024246
Quantidade de soluções ótimas encontradas: 9

Tempo de excução: 0.146083831787
Quantidade de soluções ótimas encontradas: 11

Tempo de excução: 0.149431943893
Quantidade de soluções ótimas encontradas: 9

Tempo de excução: 0.13797211647
Quantidade de soluções ótimas encontradas: 2

Tempo de excução: 0.133819103241
Quantidade de soluções ótimas encontradas: 9

Tamanho da população: 50
Quantidade de iterações: 100
Taxa de mutação: 100
Taxa de cruzamento: 100

Tempo de excução: 0.432214975357
Quantidade de soluções ótimas encontradas: 16

Tempo de excução: 0.434607028961
Quantidade de soluções ótimas encontradas: 2

Tempo de excução: 0.452836036682
Quantidade de soluções ótimas encontradas: 20

Tempo de excução: 0.436857223511
Quantidade de soluções ótimas encontradas: 17

Tempo de excução: 0.44628405571
Quantidade de soluções ótimas encontradas: 21

Tamanho da população: 100
Quantidade de iterações: 50
Taxa de mutação: 50
Taxa de cruzamento: 50

Tempo de excução: 0.270770072937
Quantidade de soluções ótimas encontradas: 15

Tempo de excução: 0.271322965622
Quantidade de soluções ótimas encontradas: 18

Tempo de excução: 0.26748585701
Quantidade de soluções ótimas encontradas: 15

Tempo de excução: 0.260529994965
Quantidade de soluções ótimas encontradas: 18

Tempo de excução: 0.272617101669
Quantidade de soluções ótimas encontradas: 9

Tamanho da população: 100
Quantidade de iterações: 100
Taxa de mutação: 100
Taxa de cruzamento: 100

Tempo de excução: 0.860871076584
Quantidade de soluções ótimas encontradas: 28

Tempo de excução: 0.851602077484
Quantidade de soluções ótimas encontradas: 26

Tempo de excução: 0.856492042542
Quantidade de soluções ótimas encontradas: 31

Tempo de excução: 0.856260061264
Quantidade de soluções ótimas encontradas: 39

Tempo de excução: 0.850182056427
Quantidade de soluções ótimas encontradas: 25

Tamanho da população: 300
Quantidade de iterações: 50
Taxa de mutação: 50
Taxa de cruzamento: 50

Tempo de excução: 0.769104003906
Quantidade de soluções ótimas encontradas: 32

Tempo de excução: 0.876961946487
Quantidade de soluções ótimas encontradas: 40

Tempo de excução: 0.936657190323
Quantidade de soluções ótimas encontradas: 44

Tempo de excução: 0.799993038177
Quantidade de soluções ótimas encontradas: 34

Tempo de excução: 0.812148094177
Quantidade de soluções ótimas encontradas: 36

Tamanho da população: 300
Quantidade de iterações: 100
Taxa de mutação: 100
Taxa de cruzamento: 100

Tempo de excução: 2.58958506584
Quantidade de soluções ótimas encontradas: 67

Tempo de excução: 2.62282395363
Quantidade de soluções ótimas encontradas: 73

Tempo de excução: 2.61409783363
Quantidade de soluções ótimas encontradas: 79

Tempo de excução: 2.65116286278
Quantidade de soluções ótimas encontradas: 71

Tempo de excução: 2.63370299339
Quantidade de soluções ótimas encontradas: 88

Tamanho da população: 500
Quantidade de iterações: 50
Taxa de mutação: 50
Taxa de cruzamento: 50

Tempo de excução: 1.439868927
Quantidade de soluções ótimas encontradas: 54

Tempo de excução: 1.51716899872
Quantidade de soluções ótimas encontradas: 61

Tempo de excução: 1.45010995865
Quantidade de soluções ótimas encontradas: 54

Tempo de excução: 1.54794001579
Quantidade de soluções ótimas encontradas: 62

Tempo de excução: 1.50517106056
Quantidade de soluções ótimas encontradas: 62

Tamanho da população: 500
Quantidade de iterações: 100
Taxa de mutação: 100

Taxa de cruzamento: 100

Tempo de excução: 4.51298713684
Quantidade de soluções ótimas encontradas: 113

Tempo de excução: 4.54015994072
Quantidade de soluções ótimas encontradas: 118

Tempo de excução: 4.55170989037
Quantidade de soluções ótimas encontradas: 115

Tempo de excução: 4.5067949295
Quantidade de soluções ótimas encontradas: 109

Tempo de excução: 4.55071282387
Quantidade de soluções ótimas encontradas: 133

Tamanho da população: 1000
Quantidade de iterações: 500
Taxa de mutação: 75
Taxa de cruzamento: 75

Tempo de excução: 40.0411128998
Quantidade de soluções ótimas encontradas: 180

Tempo de excução: 45.9370200634
Quantidade de soluções ótimas encontradas: 180

Tempo de excução: 41.3267250061
Quantidade de soluções ótimas encontradas: 185

Tempo de excução: 40.9194839001
Quantidade de soluções ótimas encontradas: 183

Tempo de excução: 41.5393671989
Quantidade de soluções ótimas encontradas: 188

Tamanho da população: 5000
Quantidade de iterações: 1000
Taxa de mutação: 80
Taxa de cruzamento: 80

Tempo de excução: 975.736050844
Quantidade de soluções ótimas encontradas: 342

Tempo de excução: 810.701250076
Quantidade de soluções ótimas encontradas: 322

Tempo de excução: 1003.64282393

Quantidade de soluções ótimas encontradas: 329

Tempo de excução: 859.744053125
Quantidade de soluções ótimas encontradas: 336

empo de excução: 851.692075014
Quantidade de soluções ótimas encontradas: 329

Tamanho da população: 5000
Quantidade de iterações: 500
Taxa de mutação: 45
Taxa de cruzamento: 90

Tempo de excução: 245.710528135
Quantidade de soluções ótimas encontradas: 344

Tempo de excução: 283.025739908
Quantidade de soluções ótimas encontradas: 319

Tempo de excução: 321.218425989
Quantidade de soluções ótimas encontradas: 318

Tempo de excução: 272.793746948
Quantidade de soluções ótimas encontradas: 286

Tempo de excução: 270.759430885
Quantidade de soluções ótimas encontradas: 324

5 Código da implementação

O código abaixo é responsável receber os parâmetros do usuários e executar a classe que implementa o Algoritmo Genético.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

#main.py

from agenetico import AGenetico
import time

print "Algoritmo Genetico: Oito Rainhas"
populacao = input("Digite o tamanho da populacao:")
iteracoes = input("Digite a quantidade de iteracoes:")
taxaMutacao = input("Digite a taxa de mutacao:")
taxaCruzamento = input("Digite a taxa de cruzamento:")

# Passa o tamanho do tabuleiro
aGenetico = AGenetico(8)

# Inicia a contagem do tempo
inicio = time.time()

# Passa o tamanho da populao
aGenetico.criaPopulacao(populacao)
aGenetico.calculaDiagonais()

for i in range(iteracoes):
    descendente = aGenetico.recombincaoPMX(taxaCruzamento)
    aGenetico.melhoraPopulacao(taxaMutacao)
    aGenetico.substituirPopulacao(descendente)

# Finaliza a contagem do tempo
fim = time.time()
print
print "Tempo de excucao: ", fim - inicio
print "Quantidade de solucoes otimas encontradas: " +
    str(aGenetico.qtdSolucoesOtimas())

#print " Solues  otimas:"
#aGenetico.printSolucoesOtimas()
```

O código a seguir implementa a classe referente ao Algoritmo Genético.

```
    #!/usr/bin/env python
# -*- coding: utf-8 -*-

#tabuleiro.py

from random import randint

class AGenetico(object):

    def __init__(self, tamanho):
        self.tamanho = tamanho
        self.populacao = []
        self.diagonalPositiva = []
        self.diagonalNegativa = []
        self.resultFObjetivo = []


    #Cria a populacao inicial
    def criaPopulacao(self, quantidade):
        for i in range(quantidade):
            pop = []
            gerados = []

            for j in range(self.tamanho):
                r = randint(0, self.tamanho-1)

                while(r in gerados):
                    r = randint(0, self.tamanho-1)

                gerados.append(r)
                pop.append(r)

            self.populacao.append(pop)


    def calculaColisoesDPositiva(self, pop):
        colisoes = 0
        verificados = []
        for i in range(len(pop)):
            if(self.diagonalPositiva[i][pop[i]] in verificados):
                 colisoes = colisoes + 1
            else:
                verificados.append(self.diagonalPositiva[i][pop[i]])

        return colisoes


    def calculaColisoesDNegativa(self, pop):
        colisoes = 0
        verificados = []
```

```
51          for i in range(len(pop)):
52              if(self.diagonalNegativa[i][pop[i]] in verificados):
53                  colisoes = colisoes + 1
54              else:
55                  verificados.append(self.diagonalNegativa[i][pop[i]])
56
57          return colisoes
58
59
60      def funcaoObjetivo(self, pop):
61          return (self.calculaColisoesDPositiva(pop) + self.calculaColisoesDNegativa(pop))
62
63
64      # Selecao por torneio
65      def seleciona(self):
66          i = randint(0, self.tamanho-1)
67          j = randint(0, self.tamanho-1)
68          escolhido1 = 0
69          escolhido2 = 0
70
71          # Evita escolher um cara que ja seja solucao
72          while(i == j or self.funcaoObjetivo(self.populacao[j]) == 0):
73              j = randint(0, self.tamanho-1)
74
75          if self.funcaoObjetivo(self.populacao[i]) <
                self.funcaoObjetivo(self.populacao[j]):
76              escolhido1 = i
77          else:
78              escolhido1 = j
79
80          i = randint(0, self.tamanho-1)
81
82          while(escolhido1 == i):
83              i = randint(0, self.tamanho-1)
84
85          j = randint(0, self.tamanho-1)
86
87          # Evita escolher um cara que ja seja soluo otima
88          while(i == j or escolhido1 == j or self.funcaoObjetivo(self.populacao[j]) == 0):
89              j = randint(0, self.tamanho-1)
90
91          if self.funcaoObjetivo(self.populacao[i]) <
                self.funcaoObjetivo(self.populacao[j]):
92              escolhido2 = i
93          else:
94              escolhido2 = j
95
96          return ((escolhido1, self.populacao[escolhido1]), (escolhido2,
                self.populacao[escolhido2]))
97
98
99      # Partialy-mapped crossover (PMX)
100     # Extensao da  combinao  apresentada no artigo
```

```
101     def recombincaoPMX(self, taxa):
102         rand = faixa = randint(1, 101)
103 
104         if rand <= taxa:
105             faixa = randint(1, self.tamanho-1)
106             inicio = randint(0, (self.tamanho-1)-faixa)
107 
108             # Selecao por torneio
109             escolhido1, escolhido2 = self.seleciona()
110 
111             # Observe a troca nos cromossomos
112             descendente1 = self.populacao[escolhido2[0]][inicio:(inicio+faixa)]
113             descendente2 = self.populacao[escolhido1[0]][inicio:(inicio+faixa)]
114 
115             relacao = []
116             for i in range(len(descendente1)):
117                 relacao.append([descendente1[i], descendente2[i]])
118 
119             flag = True
120             while(flag):
121                 flag = False
122                 for rel in relacao:
123                     for relAux in relacao:
124                         if rel[1] == relAux[0]:
125                             rel[1] = relAux[1]
126                             relacao.remove(relAux)
127                             flag = True
128 
129             p1 = self.populacao[escolhido1[0]][:inicio]
130             p2 = self.populacao[escolhido2[0]][:inicio]
131             for rel in relacao:
132                 if rel[0] in p1:
133                     for i in range(len(p1)):
134                         if p1[i] == rel[0]:
135                             p1[i]  = rel[1]
136 
137             for rel in relacao:
138                 if rel[1] in p2:
139                     for i in range(len(p2)):
140                         if p2[i] == rel[1]:
141                             p2[i]  = rel[0]
142 
143             f1 = self.populacao[escolhido1[0]][inicio+faixa:]
144             f2 = self.populacao[escolhido2[0]][inicio+faixa:]
145             for rel in relacao:
146                 if rel[0] in f1:
147                     for i in range(len(f1)):
148                         if f1[i] == rel[0]:
149                             f1[i]  = rel[1]
150 
151             for rel in relacao:
152                 if rel[1] in f2:
153                     for i in range(len(f2)):
```

```
                    if f2[i] == rel[1]:
                        f2[i]  = rel[0]

        descendente1 = p1+ descendente1 + f1
        descendente2 = p2 + descendente2 + f2
        if self.funcaoObjetivo(descendente1) < self.funcaoObjetivo(descendente2):
            return descendente1
        else:
            return descendente2

    else:
        #  Selecao  por torneio
        escolhido1, escolhido2 = self.seleciona()

        # Observe a troca nos cromossomos
        descendente1 = self.populacao[escolhido1[0]]
        descendente2 = self.populacao[escolhido2[0]]

        if self.funcaoObjetivo(descendente1) < self.funcaoObjetivo(descendente2):
            return descendente1
        else:
            return descendente2


def melhoraPopulacao(self, taxa):
    rand = randint(1, 101)

    if rand <= taxa:
        melhorada = []
        for pop in self.populacao:
            i = randint(0, self.tamanho-1)
            j = randint(0, self.tamanho-1)

            newPop = pop[:]
            newPop[i], newPop[j] = newPop[j], newPop[i]

            if self.funcaoObjetivo(newPop) < self.funcaoObjetivo(pop) and newPop not
                 in self.populacao:
                melhorada.append(newPop)
            else:
                melhorada.append(pop)

            self.populacao = melhorada[:]


def obterMaior(self):
    maior = (0, self.funcaoObjetivo(self.populacao[0]))

    for i in range(len(self.populacao)):
        if maior[1] < self.funcaoObjetivo(self.populacao[i]):
                maior = (i, self.funcaoObjetivo(self.populacao[i]))

    return maior
```

```
206
207
208     def substituirPopulacao(self, escolha):
209         maior = self.obterMaior()
210         if escolha not in self.populacao:
211             if self.funcaoObjetivo(escolha) <= maior[1]:
212                 self.populacao[maior[0]] = escolha[:]
213
214
215     def calculaDiagonais(self):
216          for i in range(self.tamanho):
217             positiva = []
218             negativa = []
219             for j in range(self.tamanho):
220                 positiva.append(i - j)
221                 negativa.append(i + j)
222
223             self.diagonalPositiva.append(positiva)
224             self.diagonalNegativa.append(negativa)
225
226
227     def printPopulacao(self):
228         for pop in self.populacao:
229             print self.funcaoObjetivo(pop), pop
230
231
232     def printDiagonalPositiva(self):
233         for dp in self.diagonalPositiva:
234             print dp
235
236
237     def printDiagonalNegativa(self):
238         for dn in self.diagonalNegativa:
239             print dn
240
241
242     def qtdSolucoesOtimas(self):
243         cont = 0
244         for pop in self.populacao:
245             func = self.funcaoObjetivo(pop)
246             if func == 0:
247                 cont = cont + 1
248
249         return cont
250
251
252     def printSolucoesOtimas(self):
253         for pop in self.populacao:
254             func = self.funcaoObjetivo(pop)
255             if func == 0:
256                 print func, pop
```

6 Referências

Uma solução do problema das n rainhas através de algoritmos genéticos - Eliane Vendramini de Oliveira

Link: https://sistemas.riopomba.ifsudestemg.edu.br/dcc/materiais/1638062552_rainhas.PDF

Problema das 8 rainhas - Marathoncode

Link: http://marathoncode.blogspot.com.br/2012/06/importancia-de-algoritmos-eficientes.html