Universidade Federal de Uberlândia

Faculdade de Engenharia Elétrica

8º trabalho de Inteligência Artificial

Fundamentos de algoritmos genéticos

Aluno
Roní G. Gonçalves
10921EEL026

Professor
Keiji Yamanaka

Uberlândia, 17 de julho de 2014

Resumo

Um programa implementando os princípios básicos de algoritmos genéticos foi feito para se entender melhor como atuam os mecanismos de seleção, cruzamento e mutação numa população de números que são candidatos para minimizarem uma dada função $f(x)$.

Palavras-chave: inteligência artificial, algoritmos genéticos, seleção, cruzamento, mutação.

Abstract

An application that implements the fundamentals of genetic algorithms was developed in order to better understand the selection, crossover and mutation mechanisms in a population in which contains the candidates that minimize a given function $f(x)$.

Keywords: artificial intelligence, genetic algorithms, selection, crossover, mutation.

Sumário

1 Introdução

Ao longo deste trabalho os mecanismos básicos presentes nos algoritmos genéticos são apresentados e exemplificados com código feito em Python.

Os mecanismos são unidos para resolverem um problema de minimização de uma função $f(x)$ proposto como exemplo no artigo de Marco Tomassini [2].

O primeiro processo é a criação de um primeiro conjunto de números binários gerados aleatoriamente; o segundo processo trata-se da seleção dos indivíduos mais aptos da geração anterior por meio da função de aptidão; em seguida, os indivíduos selecionados são aleatoriamente arranjados aos pares para que haja ou não o cruzamento entre eles de acordo com uma probabilidade de cruzamento p_c. Finalmente após todas essas etapas, de acordo com uma probabililidade p_m de mutação os indivíduos originários do cruzamento de seus pais podem sofrer ou não mutação em algum de seus bits para que, a partir do processo de seleção, tudo se repita até que alguma condição seja alcançada. No presente trabalho, tal condição foi o número de gerações criadas.

Como exemplo de aplicação, tal algoritmo foi usado para encontrar o valor mínimo ou, pelo menos, um valor próximo ao mínimo da função $f(x) = -\left|x \sin \sqrt{|x|}\right|$ dentro do intervalo[1] de 0 a 512, isto é, encontrar x^*:

$$f(x^*) = \min\{f(x) \mid \forall x \in [0; 512]\}$$

Tomando $g(x) = -f(x)$, podemos transformar o problema anterior num de maximização de $g(x)$. Isso é feito para simplificar o algoritmo de forma a termos sempre valores positivos para as probabilidades calculadas futuramente. No entanto, o valor de x^* que maximiza $g(x)$ é o mesmo que minimiza $f(x)$, pois os problemas são equivalentes:

$$g(x^*) = \max\{g(x) \mid \forall x \in [0; 512]\}$$

O gráfico das funções $f(x)$ e $g(x)$ podem ser visualizados nas figuras 1 e 2.

Pela análise do gráfico, percebe-se que o máximo ou o mínimo das funções é próximo de 400. Na verdade, o valor real é 421.

[1] Tal intervalo é escolhido somente para números pares devido à natureza simétrica da função $f(x)$.

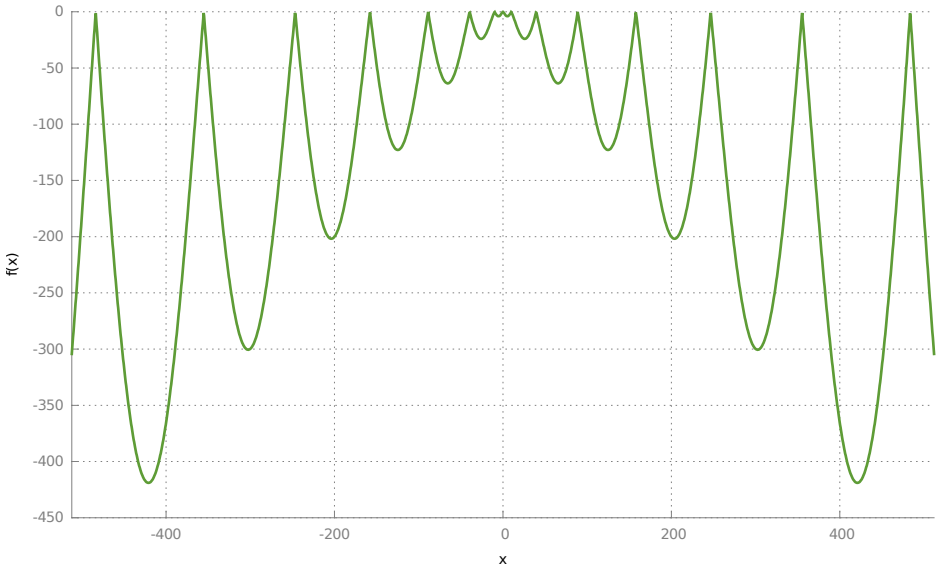

Figura 1: Gráfico da função $f(x)$ no intervalo de -512 a 512.

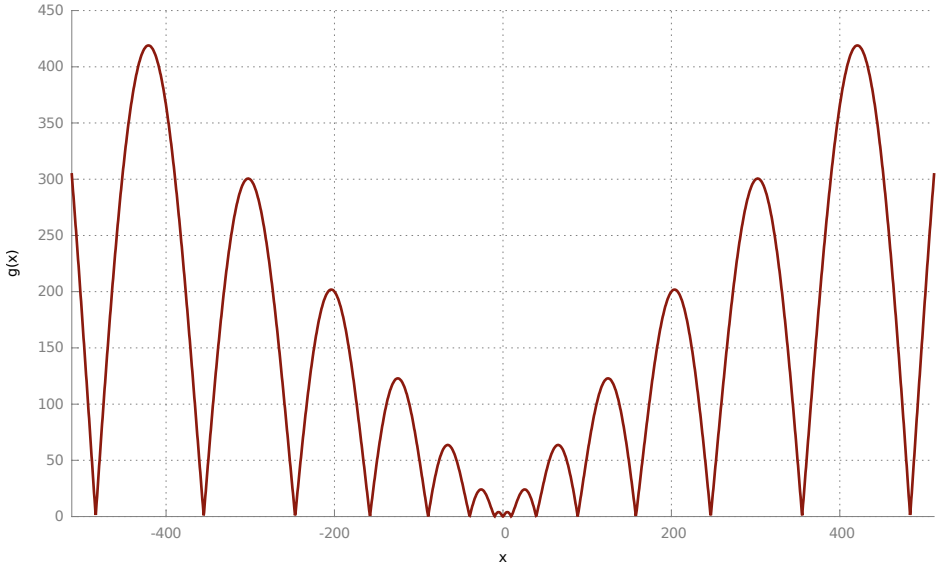

Figura 2: Gráfico da função $g(x)$ no intervalo de -512 a 512.

2 Gerando aleatoriamente a primeira geração G_0

Duas funções são usadas para criar a primeira geração constituída de *strings* contendo zeros e uns representando, em binário, os números candidatos a serem o valor que maximiza $g(x)$.

A primeira função, `gera_individuo_aleatorio()`, apenas seleciona nove[2] vezes de forma pseudo-aleatória um dos caracteres – 0 ou 1 – da *string_base*.

Enquanto que a segunda função, `gera_populacao()`, usa a primeira função para preencher uma lista com um número de indivíduos igual ao parâmetro *TamanhoDaPopulacao*.

```
def gera_individuo_aleatorio():

        return (''.join(random.choice(string_base) for
            j in range(TamanhoDaString)))

def gera_populacao(TamanhoDaPopulacao):

        a = []

        for i in range(TamanhoDaPopulacao):
                a.insert(i, gera_individuo_aleatorio()
                    )

        return a
```

3 Calculando a função de aptidão e probabilidades de cada indivíduo da população

Antes de se aplicar a função de aptidão, é necessário fazer algumas manipulações com os indivíduos, pois eles são representados por *strings* contendo apenas zeros e uns tais como: "000101010", "000011001", "000001101"...

Essas *strings* são convertidas em números decimais pela seguinte função:

```
def binario_em_decimal(Lista):
```

[2]Para que haja 512 combinações diferentes de zeros e uns, são necessários nove dígitos binários: $2^9 = 512$. Porém, dessa forma o número 512 não fica representado, pois com nove dígitos binários podemos representar 512 números de 0 a $511 = 2^9 - 1$.

```
        a = []

        for i in range(len(Lista)):
                a.insert(i, int(Lista[i], 2))

        return a
```

Após passar por esta função, os indivíduos são representados por números decimais[3] tais como: 42, 25, 13... e agora podem ser enviados à função que calcula suas aptidões.

3.1 Função de aptidão

Também conhecida em inglês por *fitness*, a função de aptidão é aquela que avalia o valor de x_i, candidato a máximo de $g(x)$, e nos dá a medida de quão bom o i-ésimo candidato é. Nesse caso, a função de aptidão é a própria função g e quanto maior o valor de $g(x_i)$, melhor é o candidato i. Temos, então, que:

$$f_{aptidao} = \left| x \sin \sqrt{|x|} \right|$$

O que, em Python, pode ser escrito como:

```
def calcula_aptidao(Cromossomo_Individuo):

        return (abs((Cromossomo_Individuo)*(sin(sqrt(
           Cromossomo_Individuo)))))
```

Vale lembrar que o valor de `Cromossomo_Individuo` é aquele calculado pela função `binario_em_decimal()`.

3.1.1 Dicionários ou *hashes*

Uma geração qualquer G_i é representada, em Python, por uma lista contendo seus indivíduos. A lista tem o tamanho igual ao número de indivíduos contidos na geração G_i. Cada um deles pode ser acessado por meio de seus respectivos índices que vão de 0 até (Número de indivíduos −1). Por exemplo: numa lista chamada pop, o primeiro elemento dela é acessado como `pop[0]`.

[3] Evidentemente que dentro do computador todos os números são representados de forma binária e não decimal.

No entanto, essa maneira de guardar a identidade do indivíduo e o seu valor é insuficiente para atrelá-lo ao valor de sua função de aptidão. Para fazer isso, usou-se uma estrutura de dados nativa do Python chamada dicionário ou, em inglês, *hash*.

Os dicionários são um conjunto de pares do tipo (chave, valor). Para este problema específico as chaves são os índices que representam cada indivíduo de uma certa geração da lista; os valores são os resultados da função de aptidão bem como as probabilidades calculadas subseqüentemente como se verá a seguir.

Por meio, então, dos dicionários é possível associar a um indivíduo qualquer a sua função de aptidão:

```
def associa_aptidao(Populacao):

        dicionario = {}

        for i in range(len(Populacao)):

                dicionario[i] = calcula_aptidao(
                   Populacao[i])

        return dicionario
```

3.2 Probabilidades dos indivíduos serem selecionados

A função de aptidão já mede o quão bons são os candidatos a máximo. É necessário ainda que, para cada candidato, seja atribuída uma probabilidade de ser selecionado proporcional ao valor de sua função de aptidão. Isso é feito em três passos:

1. Para cada geração, calcula-se a soma S dos valores da função de aptidão de todos os seus indivíduos;
2. Para cada indivíduo, calcula-se a probabilidade p_i como sendo a razão entre sua função de aptidão e S e a associa ao respectivo indivíduo;
3. Cria-se uma "roleta" contendo c_i, as somas parciais e sucessivas de p_i.

As equações referentes ao passo-a-passo anterior são:

$$S = \sum_{i=1}^{TamGeracao} g(x_i)$$

$$p_i = \frac{g(x_i)}{S}$$

$$c_i = \sum_{k=1}^{i} p_k, \quad i = 1, 2, 3, \ldots, TamanhoDaGeracao$$

O código da função que realiza o primeiro passo é como se segue:

```
def calcula_aptidao_total(Dicionario):

        aptidao_total = 0

        for i in range(len(Dicionario)):

                aptidao_total = aptidao_total +
                   Dicionario[i]

        return aptidao_total
```

A função que executa o segundo passo pode ser escrita como:

```
def associa_probabilidade(Dicionario, AptidaoTotal):

        for i in range(len(Dicionario)):

                Dicionario[i] = Dicionario[i]/
                   AptidaoTotal

        return Dicionario
```

O último passo é conseguido como mostra a próxima função associa_prob_cumulativa():

```
def associa_prob_cumulativa(Dicionario):

        prob_acumulada = 0

        for i in range(len(Dicionario)):

                prob_acumulada = prob_acumulada +
                   Dicionario[i]
```

```
8
                Dicionario[i] = prob_acumulada

11      return Dicionario
```

4 Selecionando os indivíduos da próxima geração

Tendo todas as probabilidades dos indivíduos calculadas e associadas a eles, resta apenas selecionar quais serão os escolhidos para cruzarem e gerarem os novos indivíduos da próxima geração.

```
1  def seleciona(Populacao, Dicionario):

       PopulacaoSelecionada = []
4
       DicionarioAuxiliar = {}

7      ListaAuxiliar = []

       ListaAuxiliar = Dicionario.values()
10
       indice = 0

13     for i in range(len(Dicionario)):

           DicionarioAuxiliar[Dicionario[i]] = i
16
       for i in range(len(Populacao)):

19         numero_aleatorio = random.uniform
               (0.0,1.0)

           ListaAuxiliar.append(numero_aleatorio)
22
           ListaAuxiliar.sort()

25         indice = ListaAuxiliar.index(
               numero_aleatorio)
```

```
                if(indice == (len(ListaAuxiliar) - 1))
                    :
28
                        indice = indice - 1

31              ListaAuxiliar.remove(numero_aleatorio)

                PopulacaoSelecionada.insert(i,
                    Populacao[DicionarioAuxiliar[
                    ListaAuxiliar[indice]]])
34
        return PopulacaoSelecionada
```

4.1 Princípio de funcionamento da roleta

Ao obter as probabilidades acumuladas c_1, c_2, c_3, forma-se um tipo de gráfico de setores. Usando-se a função `random.uniform()`, obtém-se um número r aleatório que é comparado aos c_i de tal forma que se $c_0 < r \leqslant c_1$, então o indivíduo selecionado será aquele correspondente ao índice 1.

5 Cruzando os indivíduos selecionados

Uma vez que os indivíduos de uma dada geração n são selecionados, eles são arranjados aleatoriamente aos pares e cruzados para formarem os indivíduos da próxima geração $n+1$. É importante notar que, nesse processo de seleção e cruzamento, os pais originais são perdidos para sempre.

O cruzamento, ou melhor, a troca de informações entre os pais se dá da seguinte forma: um número aleatório entre zero e o valor do índice do penúltimo elemento da *string* que representa um indivíduo é escolhido e a partir desse índice duas *substrings* são criadas contendo uma parte da informação original dos "pais"; após isso, elas são comutadas entre os pais criando os filhos.

Algumas conversões de tipos acontecem por conta da escolha da estrutura de dados feita para representar a informação genética que cada indivíduo carrega. Em Python, os elementos de uma *string* podem ser acessados individualmente, porém eles não podem ser alterados. Assim, as *strings* são convertidas em listas que permitem não só acessar seus membros pelos índices, como também alterá-los. Feitas todas as alterações na informação dos indivíduos eles são novamente convertidos em *strings* por meio

do método `join()`.

Além disso, o cruzamento não ocorre sempre. Na verdade, ele pode ocorrer se um número aleatório dentro de uma distribuição uniforme com valores entre 0 e 1 for maior que o valor definido para a probabilidade de cruzamento p_c que, nesse caso, vale 0,6.

```
def cruza(Populacao):

        pai = ""
        mae = ""

        indice_pai = random.choice(range(len(Populacao
            )))

        indice_mae = random.choice(range(len(Populacao
            )))

        pai = Populacao[indice_pai]
        mae = Populacao[indice_mae]

        indice_cruzamento = random.choice(range(0,(len
            (pai)-2)))

        pai = list(pai)
        mae = list(mae)

        for i in range(indice_cruzamento, len(pai)):

                aux = ''

                aux = pai[i]

                pai[i] = mae[i]

                mae[i] = aux

        pai = ''.join(pai)
        mae = ''.join(mae)

        Populacao[indice_pai] = pai
```

```
        Populacao[indice_mae] = mae

        return Populacao
```

6 Mutando os indivíduos de uma geração

A mutação é o último mecanismo que age sobre uma nova geração, logo após ela ser formada por meio do cruzamento dos indivíduos selecionados da geração anterior. Ela pode ou não acontecer; para cada indivíduo e para cada bit[4] que compõe o cromossomo do elemento da população, é sorteado um número aleatório entre zero e um de uma distribuição uniforme – usa-se a função `random.uniform(0.0,1.0)` – e caso o número aleatório sorteado seja maior que a probabilidade p_m de mutação, então o indivíduo terá seus genes mutados. A probabilidade de mutação vale 0,01.

```
def muta(Populacao):

        indice_populacao = random.randint(0, (len(
           Populacao)-1))

        auxilio = Populacao[indice_populacao]

        auxilio = list(auxilio)

        indice_individuo = random.randint(0, (len(
           auxilio)-1))

        if (auxilio[indice_individuo] == '1'):

                auxilio[indice_individuo] = '0'

        else:
                auxilio[indice_individuo] = '1'

        auxilio = ''.join(auxilio)

        Populacao[indice_populacao] = auxilio
```

[4]Nesse caso, na verdade, é para cada caractere que compõe a *string*.

```
        return Populacao
```

7 Resultados

Nesta seção, são apresentados alguns dos resultados obtidos variando-se alguns dos inúmeros parâmetros possíveis deste algoritmo. Um fato importante de ser notado é a natureza parcialmente aleatória dos resultados obtidos. Isso significa que após a execução do programa, não necessariamente o valor x^* que minimiza a função $f(x)$ será realmente encontrado: pode-se encontrar um valor razoavelmente próximo.

Um esquema simplificado do algortimo usado é mostrado nos passos abaixo:

1. Cria n indivíduos aleatórios para compor a geração inicial G_0;
2. Enquanto o número de gerações for menor que um valor mínimo estipulado, faça:
   (a) Calcula a função de aptidão de cada indivíduo;
   (b) Calcula a probabilidade de cada indivíduo;
   (c) De acordo com a probabilidade calculada, seleciona indivíduos para comporem a próxima geração;
   (d) Aleatoriamente cruza os indivíduos selecionados;
   (e) A partir do resultado dos cruzamentos, cria a nova geração;
   (f) Aplica a mutação sobre a nova geração.

7.1 Variação do número de gerações

Ao variar a quantidade de gerações dois aspectos aparecem: (1) o aumento do número de gerações não garante que o mínimo ou máximo da função seja encontrado. Isto significa, por exemplo, que o valor mínimo da função $f(x)$, 421, pode ser encontrado na execução do algoritmo com $n = 50$ no passo 1 e, às vezes, esse mesmo mínimo não será encontrado caso o programa execute novamente com $n = 1000$; (2) apesar de a presença do melhor indivíduo não ser garantida pelo aumento do número de gerações criadas, tal aumento tende a melhora a aptidão média da população.

7.2 Variação do número de indivíduos de uma população

Nas tabelas que se seguem, os indivíduos da última geração G_{69} são mostrados. Os parâmetros probabilidade de cruzamento p_c, probabilidade de mutação p_m e quantidade de gerações criadas foram mantidos constantes e iguais a 0,6, 0,01 e 70 respectivamente. Nesses testes, o único parâmetro a ser alterado foi o número de indivíduos (5, 10, 15 e 20) para cada tabela.

teste	indivíduos
1	[367, 367, 367, 367, 367]
2	[198, 198, 198, 198, 198]
3	[291, 291, 291, 291, 291]
4	[280, 280, 280, 280, 280]
5	[407, 407, 407, 407, 407]

Tabela 1: 5 execuções do mesmo programa com uma população composta por 5 membros.

teste	indivíduos
1	[294, 294, 294, 294, 294, 294, 294, 294, 294, 294]
2	[320, 320, 320, 320, 320, 320, 320, 320, 320, 320]
3	[407, 407, 407, 407, 407, 407, 407, 407, 407, 407]
4	[421, 421, 421, 421, 421, 421, 421, 421, 421, 421]
5	[459, 459, 459, 459, 459, 459, 459, 459, 459, 459]

Tabela 2: 5 execuções do mesmo programa com uma população composta por 10 membros.

7.3 Variação das probabilidades de cruzamento e de mutação

7.3.1 Probabilidade total de cruzamento

No caso em que todos os indivíduos selecionados de uma geração n se cruzem para originar a geração $n + 1$, não há um comprometimento tão grande dos resultados obtidos. Isso mostra que, para este problema, uma condição "generosa"de ocorrência de cruzamentos não é algo tão crítico assim. Como exemplo, mostra-se a geração final após a execução do programa com $p_m = 0{,}01$ e número de geração limite igual a 70: 422, 422, 422, 422, 418, 422, 420, 421, 422, 422, 418, 422, 422, 418, 422, 422, 422, 430,

teste	indivíduos
1	[426, 426, 426, 426, 426, 426, 426, 426, 426, 426, 426, 426, 426, 426, 426]
2	[426, 426, 426, 426, 426, 426, 426, 426, 426, 426, 426, 426, 426, 426, 426]
3	[432, 432, 432, 432, 432, 432, 432, 432, 432, 432, 432, 432, 432, 432, 432]
4	[441, 441, 441, 441, 441, 441, 441, 441, 441, 441, 441, 441, 441, 441, 441]
5	[430, 430, 430, 430, 430, 430, 430, 430, 430, 430, 430, 430, 430, 430, 430]

Tabela 3: 5 execuções do mesmo programa com uma população composta por 15 membros.

430, 418, 422, 418, 421, 422, 422, 438, 418, 422, 422, 422, 418, 420, 422, 418, 418, 434, 438, 420, 426, 422, 422, 422, 428, 438, 418, 422, 422, 421, 422, 422, 422, 422, 422, 422, 422, 422, 430, 422, 422, 422, 418, 422, 426, 422, 422, 430, 418, 420, 422, 428, 418, 418, 422, 422, 418, 422, 430, 422, 422, 422, 434, 422, 422, 422, 418, 418, 418, 422, 422, 422, 422, 430, 422, 422, 422, 422, 422, 422, 418, 422.

7.3.2 Diminuição da probabilidade de cruzamento

7.3.3 Probabilidade total de mutação

Com uma probabilidade de 100% de ocorrer mutação, o "ambiente"em que os indivíduos vivem é muito extremo e faz com que um resultado razoável não seja alcançado. Numa execução com $p_c = 0{,}6$ e o limite de gerações igual a 70, a última geração obtida com 100 indivíduos foi: 209, 195, 195, 211, 214, 195, 195, 195, 193, 195, 195, 211, 211, 211, 211, 198, 214, 195, 193, 193, 211, 197, 211, 195, 198, 198, 195, 195, 211, 195, 195, 211, 195, 209, 195, 198, 195, 198, 195, 195, 195, 197, 214, 195, 195, 193, 209, 195, 197, 193, 195, 195, 198, 211, 211, 195, 195, 195, 195, 195, 211, 211, 195, 195, 209, 195, 195, 195, 195, 197, 211, 195, 195, 198, 195, 209, 195, 198, 211, 193, 198, 195, 197, 195, 195, 195, 211, 195, 195, 193, 211, 198, 195, 197, 195, 195, 214, 211, 195, 195.

Como se vê, os resultados são muito ruins sabendo que o valor exato é 421.

7.3.4 Aumento suave na probabilidade de mutação

7.4 Supressão da mutação

Sem a mutação, os indivíduos da última população são muito similares e as diferenças são devidas somente ao cruzamento entre os pais.

teste	indivíduos
1	[418, 423, 423, 418, 418, 418, 423, 423, 423, 418, 418, 418, 418, 418, 418, 418, 418, 418, 418, 418]
2	[410, 414, 410, 410, 410, 410, 410, 410, 410, 410, 410, 410, 410, 410, 410, 410, 410, 410, 410, 410]
3	[427, 427, 427, 427, 427, 427, 427, 427, 427, 427, 427, 427, 427, 427, 427, 427, 427, 427, 427, 427]
4	[432, 434, 434, 432, 432, 432, 434, 432, 432, 432, 434, 434, 434, 432, 432, 434, 434, 434, 434, 432]
5	[424, 424, 424, 424, 424, 424, 424, 424, 424, 424, 424, 424, 424, 424, 424, 424, 424, 424, 424, 424]

Tabela 4: 5 execuções do mesmo programa com uma população composta por 20 membros.

7.5 Supressão do cruzamento

Sem o cruzamento, os indivíduos se igualam muito rapidamente. O resultado converge de forma bem rápida e o único mecanismo que provê alguma variedade nas populações é a mutação, mas que ocorre com muito menos freqüência.

8 Conclusões

Os algoritmos genéticos apresentam aplicações interessantes e importantes também em áreas onde não há ainda soluções matemáticas analíticas ou numéricas para certos problemas. Por exemplo, num caso de otimização de funções, a única forma de se encontrar realmente um mínimo ou máximo global seria esgotando todas as possibilidades, o que num problema com domínio nos números reais seria uma tarefa impossível dada a limitação computacional existente hoje. Os algoritmos genéticos permitem encontrar aproximações para esses problemas percorrendo de uma forma metódica boa parte do espaço de busca onde a resposta correta se encontra. A esse metodismo é acrescentada certa aleatoriedade por meio dos mecanismos de cruzamento e mutação que não deixam o algoritmo procurar por respostas somente num conjunto restrito de supostas soluções. Como se pôde ver na seção 7, ainda que não haja garantias de se obter a resposta correta, um resultado próximo do ideal é encontrado.

9 Referências

[1] Keiji Yamanaka, *Notas de aula da disciplina Inteligência Artificial*. Faculdade de Engenharia Elétria, Universidade Federal de Uberlândia, 2014.

[2] Marco Tomassini, *A survey on genetic algorithms*. A ser publicado em Annual Reviews of Computational Physics, World Scientific.