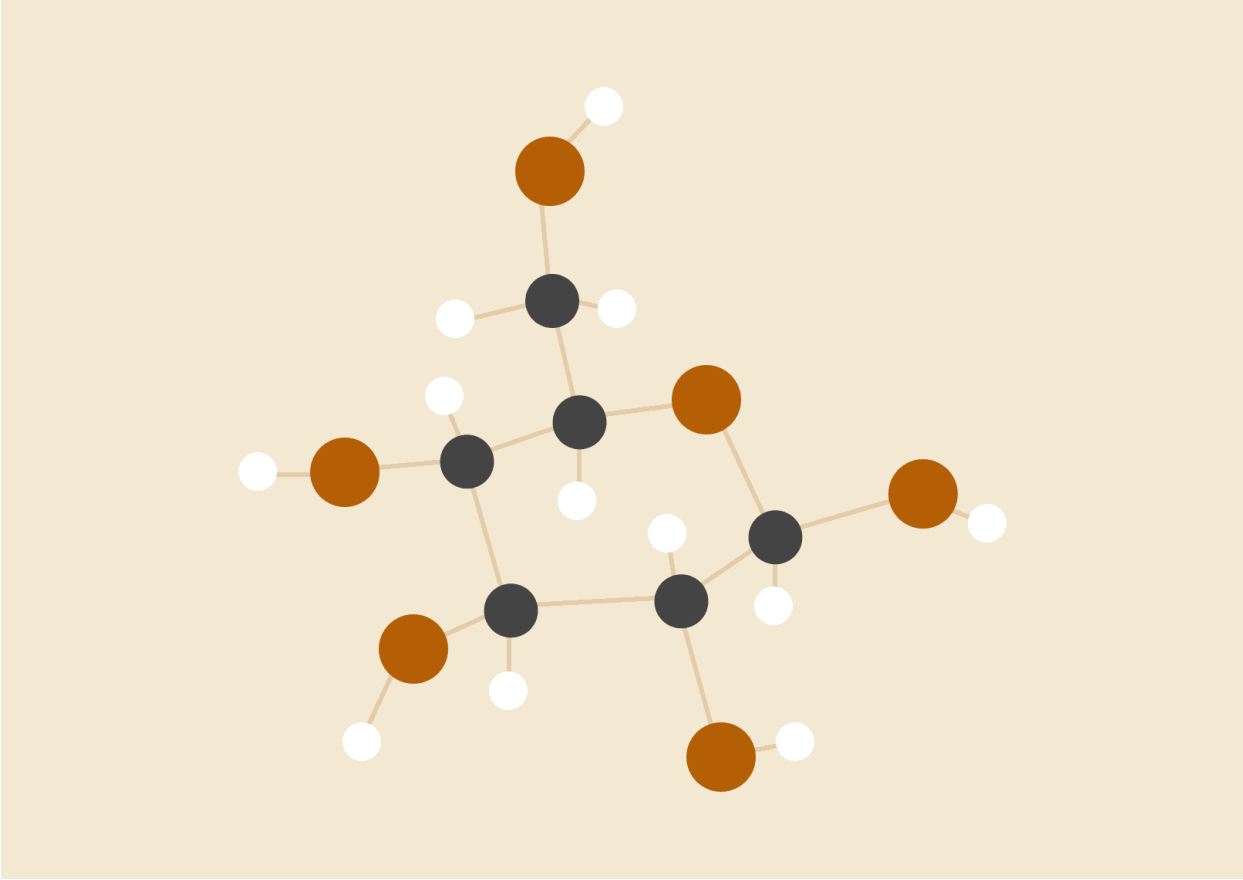
Trabalho Final PAA

Resolução do Problema MAX-3SAT com Metaheurística Genetic Algorithm



Davi Montesuma
Kaio Rodrigues
Vinicius A. Sampaio

CIÊNCIAS DA COMPUTAÇÃO UECE
Professor Dr. Leonardo S. Rocha
2022

INTRODUÇÃO

O problema MAX-3SAT é uma generalização do problema 3SAT. Diferente do 3SAT, que tenta determinar se existe uma atribuição de valores capaz de satisfazer todas as cláusulas, o MAX-3SAT tenta determinar uma atribuição que satisfaz o máximo número de cláusulas possíveis.

Apesar de parecer simples, a quantidade de soluções possíveis cresce de forma exponencial ao número de variáveis, fazendo que algoritmos que procuram a resposta exata do problema demorem semanas ou centenas de anos para encontrar a solução.

Para resolver esse problema são utilizados algoritmos baseados em heurísticas que tentam chegar o mais próximo da solução ótima utilizando técnicas iterativas que guiam a solução para o máximo global.

Neste trabalho utilizamos a metaheurística genética para chegar o mais próximo da solução ótima.

METODOLOGIA

Algoritmo Genético

A metaheurística genética é um algoritmo evolucionário que simula a seleção natural através de funções baseadas na reprodução, mutação e "*survival of the fittest*".

O conceito de seleção natural aplicado a áreas fora do reino da biologia vem desde a época que Charles Darwin publicou *"On the Origin of Species"*, com comparações sendo feitas em modelos econômicos.

Na computação a metaheurística genética vem como uma maneira de realizar processos de otimização e pesquisa, conseguindo, em geral, bons resultados. Entre as aplicações da metaheurística genética podemos citar sua utilização na busca e otimização de hiperparâmetros para

modelos de machine learning.

Uma aplicação interessante foi sua utilização na criação de uma antena para a aeronave espacial NASA ST5.

Apesar de bastante interessante, a metaheurística genética sofre com problemas onde a função de *fitness* é muito complexa. Por isso dificilmente vemos ela sendo utilizada para criação de mecanismos muito complexos, como motores ou aeronaves. Se limitando a atuar na geração de pequenas partes, como antenas ou hélices.

Parâmetros

Nossa implementação da metaheurística genética recebe os seguintes parâmetros:

- Tamanho da população;
- Gerações;
- Taxa de mutação;
- Quantidade de membros da população a serem considerados “melhores”;
- Quantidade da população a participar da reprodução da próxima geração;

A escolha dos valores dos parâmetros foi feita de forma empírica, onde testes com vários valores diferentes foram realizados.

O tamanho da população foi 1000 para os 3 casos teste, a quantidade de gerações foram 100 para SAT1 e SAT2 e 300 para SAT3, a taxa de mutação foi 0.05 para os 3 casos, a quantidade de membros escolhidos como melhores e para participar da reprodução foram 10 e 100 para os 3 casos.

Código

Código disponível em: https://github.com/vasampaio/genetic-algorithm-paa

Leitura do arquivo:

```
textfile = 'SAT2.txt'
param = np.loadtxt(textfile,max_rows=1,dtype=int)
constr = np.loadtxt(textfile,skiprows=1,dtype=int)
```

Definição de parâmetros:

```
population_size = 1000
generations = 100
```

```
mutation_rate = 0.05
N = 10
K = 100
MAX_REP = 20
n_var = param[0]
n_constr = param[1]
```

A função fitness foi feita da seguinte forma:

```
def score(x):
    score = 0
    for i in constr:
        x1,x2,x3 = x[abs(i[0])-1],x[abs(i[1])-1],x[abs(i[2])-1]
        if i[0] < 0:
            x1 = 1 - x1
        if i[1] < 0:
            x2 = 1 - x2
        if i[2] < 0:
            x3 = 1 - x3
        if x1 + x2 + x3 >= 1:
            score += 1
    return score
```

A função de reprodução (crossover) foi feita da seguinte forma:

```
def crossover(parent1,parent2):
    child = np.zeros(n_var)
    for i in range(n_var):
        if np.random.rand() < 0.5:
            child[i] = parent1[i]
        else:
            child[i] = parent2[i]
    return child


def reproduce(population):
    children = []
    top = select_top(population,topN(population,N))
```

```
    rest = select_top(population,topN(population,K))
    for i in range(population_size):
        parent1 =  top[np.random.randint(len(top))]
        parent2 =  rest[np.random.randint(len(rest))]
        child = crossover(parent1,parent2)
        child = mutate(child)
        children.append(child)
    return children
```

Podemos ver na função de crossover que utilizamos o crossover uniforme, ou seja cada cromossomo do filho resultante é escolhido de forma aleatória entre seus progenitores. Existem outras formas de crossover como o de 1 ponto ou o de k-pontos.

A função de mutação foi feita da seguinte forma:

```
def mutate(child):
    for i in range(n_var):
        if np.random.rand() < mutation_rate:
            child[i] = 1 - child[i]
    return child
```

Para encontrar os melhores membros da geração temos as seguintes funções:

```
def find_best(population):
    scores = []
    for i in population:
        scores.append(score(i))
    return np.argmax(scores)

def topN(population, n):
    scores = []
    for i in population:
        scores.append(score(i))
    return sorted(range(len(scores)), key = lambda sub: scores[sub])[-n:]

def select_top(population, top):
    x = []
    for i in top:
        x.append(population[i])
    return x
```

Inicialização da população:

```
population = np.random.randint(2,size=(population_size,n_var))
```

Execução principal:

```
results = []
for i in range(generations):
    population = reproduce(population)
    best = find_best(population)
    results.append(score(population[best]))
```

Complexidade

Tendo g = número de gerações, n = tamanho da população, v = número de variáveis e c = número de cláusulas.

Para a função score temos **O(c)**

Para a função crossover temos **O(v)**

Para a função reproduce temos **O(nv)**

Para a função mutate temos **O(v)**

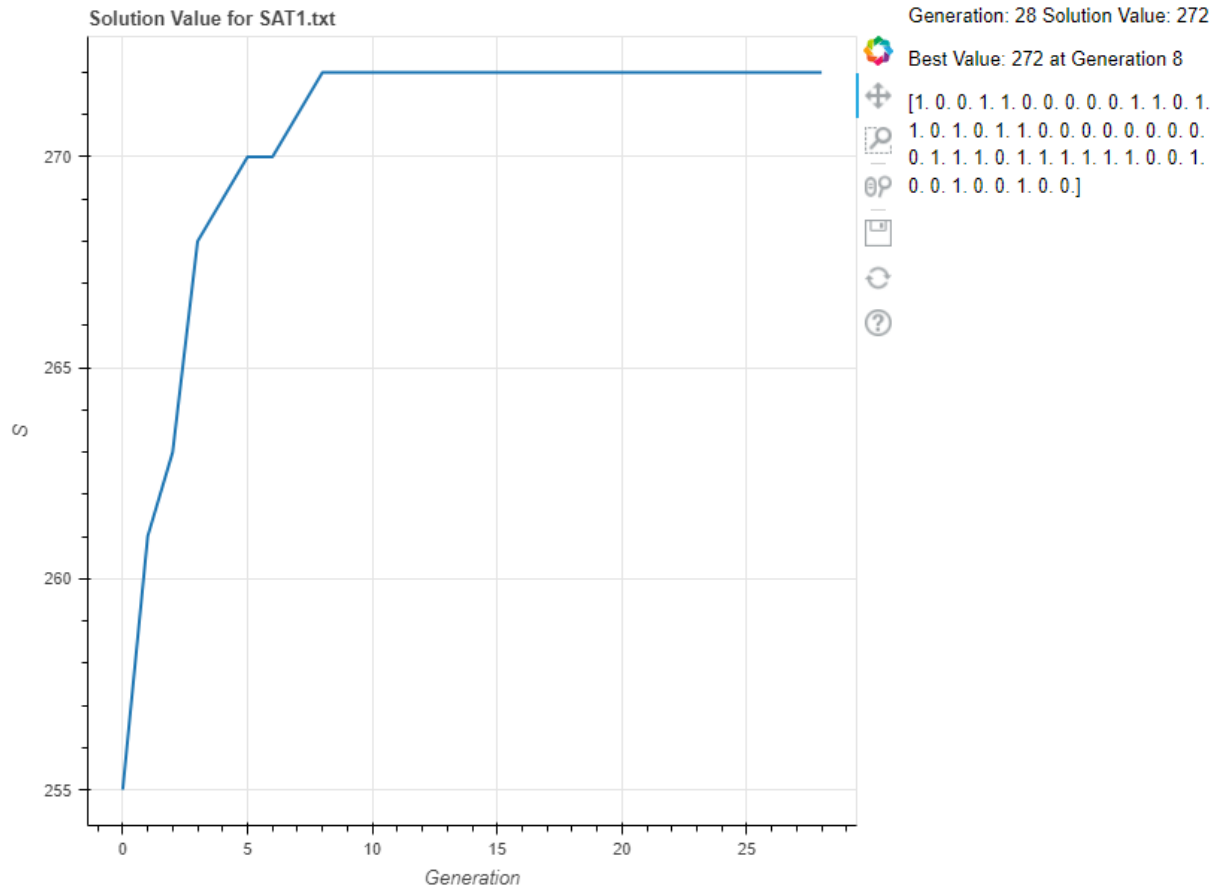
Para a função best temos **O(nc)**

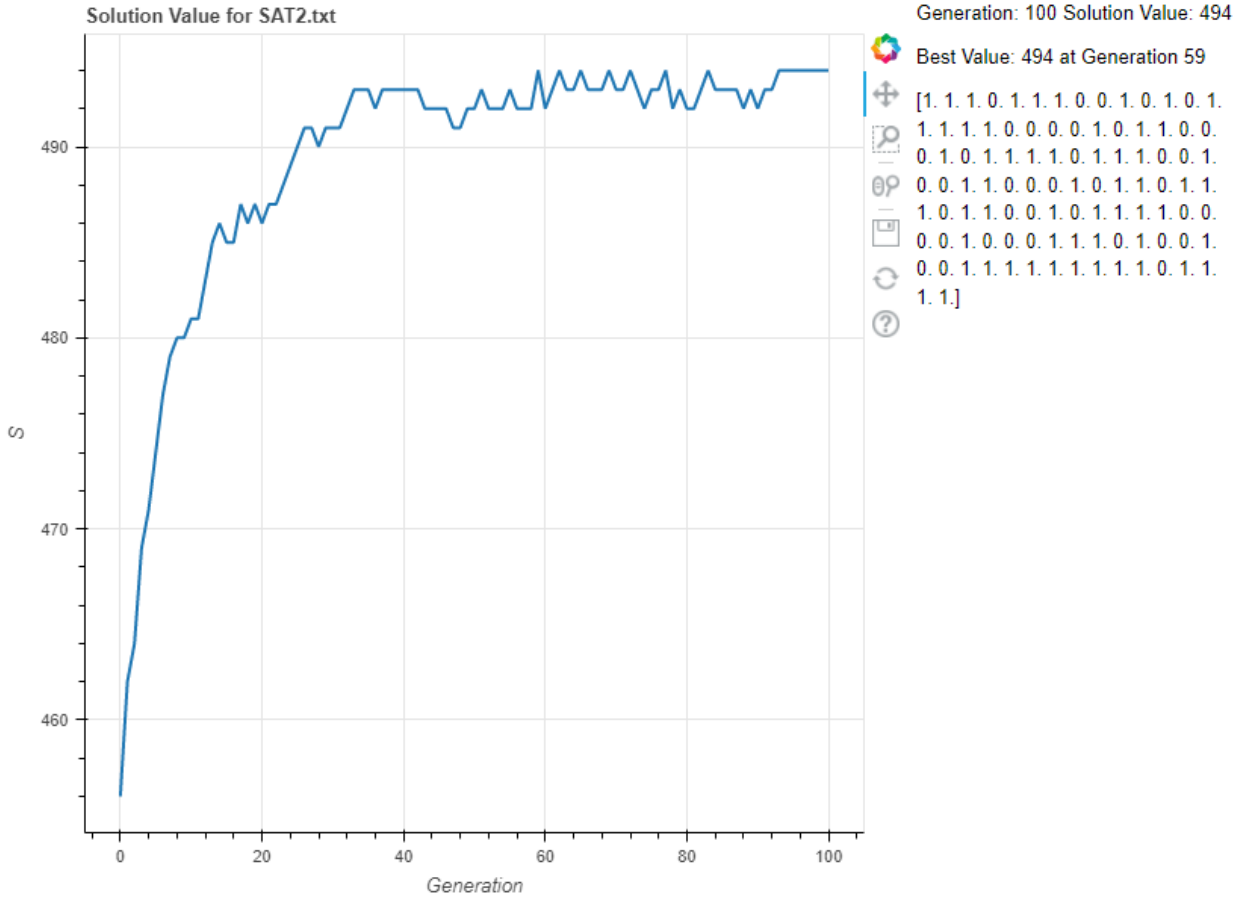
Execução principal:

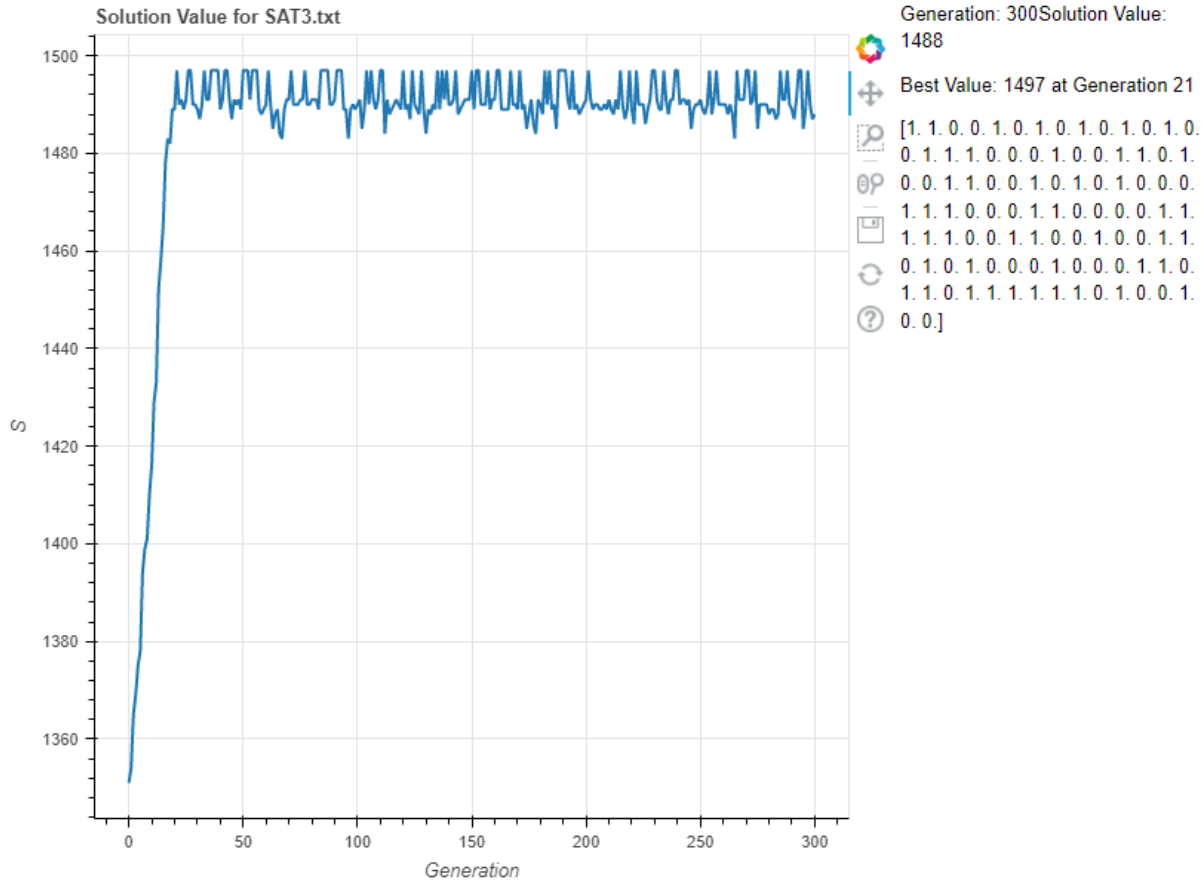
O(gn(v+c))

RESULTADOS

Instâncias Comparativas				Solução	Solução Obtida			Aproximação
Nº	Ref	n	m	Ótima (S*)	Inicial	tempo(s)	S	Gap (S/S*)
1	SAT1	50	275	275	259	46.54	272	0.98
2	SAT2	100	500	499	456	304.38	494	0.98
3	SAT3	100	1500	1497	1350	3475.9	1497	1
Média das aproximações com resultados conhecidos ->								0.986




Solution Value for SAT2.txt
S
Generation
490
480
470
460
0
20
40
60
80
100
Generation: 100 Solution Value: 494
Best Value: 494 at Generation 59
[1. 1. 1. 0. 1. 1. 1. 0. 0. 1. 0. 1. 0. 1.
1. 1. 1. 1. 0. 0. 0. 0. 1. 0. 1. 1. 0. 0.
0. 1. 0. 1. 1. 1. 1. 0. 1. 1. 1. 0. 0. 1.
0. 0. 1. 1. 0. 0. 0. 1. 0. 1. 1. 0. 1. 1.
1. 0. 1. 1. 0. 0. 1. 0. 1. 1. 1. 1. 0. 0.
0. 0. 1. 0. 0. 0. 1. 1. 1. 0. 1. 0. 0. 1.
0. 0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 1. 1.
1. 1.]



Como podemos ver na tabela e nos gráficos, a metaheurística genética consegue resultados bons chegando muito próximo ou acertando o valor ótimo. Porém seu custo em processamento é muito caro.

Os fatores que mais influenciam na demora da computação de uma geração são o tamanho da população e o cálculo da função fitness. A função fitness é dependente do problema, alterá-la significa alterar o problema. O tamanho da população afeta a quantidade de possíveis soluções exploradas em cada geração. Usar uma população muito pequena limita a pesquisa no espaço de respostas e possivelmente gera soluções não muito boas.

Em nossos experimentos percebemos que em uma população de tamanho menor que 1000 (testamos 500 e 100) os resultados ficavam muito distantes dos ótimos.

CONCLUSÃO

A metaheurística genética é um método muito interessante com diversas aplicações na

indústria e no futuro das tecnologias de machine learning. Percebemos em nossos experimentos no entanto uma dificuldade do método atingir o ótimo de forma consistente. Tal fato é provavelmente derivado da simplicidade de nossa implementação.

Algumas características como o elitismo (onde o melhor membro de uma população é repetido na geração seguinte) não foram introduzidas nesta versão do código. Essa escolha nos permitiu visualizar os efeitos da variação da taxa de mutação nos resultados de forma mais clara.

Para trabalhos futuros, pensamos em adicionar o elitismo ao código, e alguma forma de paralelismo para melhorar o tempo de execução. Também ficamos interessados nas metaheurísticas genéticas adaptativas, onde há uma introdução de novas metaheurísticas para adaptação dos parâmetros.

REFERÊNCIAS

1. Hornby, G., Globus, A., Linden, D. and Lohn, J., 2006. *Automated antenna design with evolutionary algorithms.* In Space 2006 (p. 7242).
2. M. Srinivas and L. M. Patnaik, *"Adaptive probabilities of crossover and mutation in genetic algorithms,"* in IEEE Transactions on Systems, Man, and Cybernetics, vol. 24, no. 4, pp. 656-667, April 1994, doi: 10.1109/21.286385.
3. Da Ronco C., Benini E. (2014) *A Simplex-Crossover-Based Multi-Objective Evolutionary Algorithm.* In: Kim H., Ao SI., Amouzegar M., Rieger B. (eds) IAENG Transactions on Engineering Technologies. Lecture Notes in Electrical Engineering, vol 247. Springer, Dordrecht. https://doi.org/10.1007/978-94-007-6818-5_41
4. Riazi, A. *Genetic algorithm and a double-chromosome implementation to the traveling salesman problem.* SN Appl. Sci. 1, 1397 (2019). https://doi.org/10.1007/s42452-019-1469-1
5. CONTRIBUTORS TO WIKIMEDIA PROJECTS. *Genetic algorithm*. Disponível em: <https://en.wikipedia.org/wiki/Genetic_algorithm>. Acesso em: 5 jan. 2022.
6. CONTRIBUTORS TO WIKIMEDIA PROJECTS. *Crossover (genetic algorithm)*. Disponível em: <https://en.wikipedia.org/wiki/Crossover_(genetic_algorithm)>. Acesso em: 5 jan. 2022a.