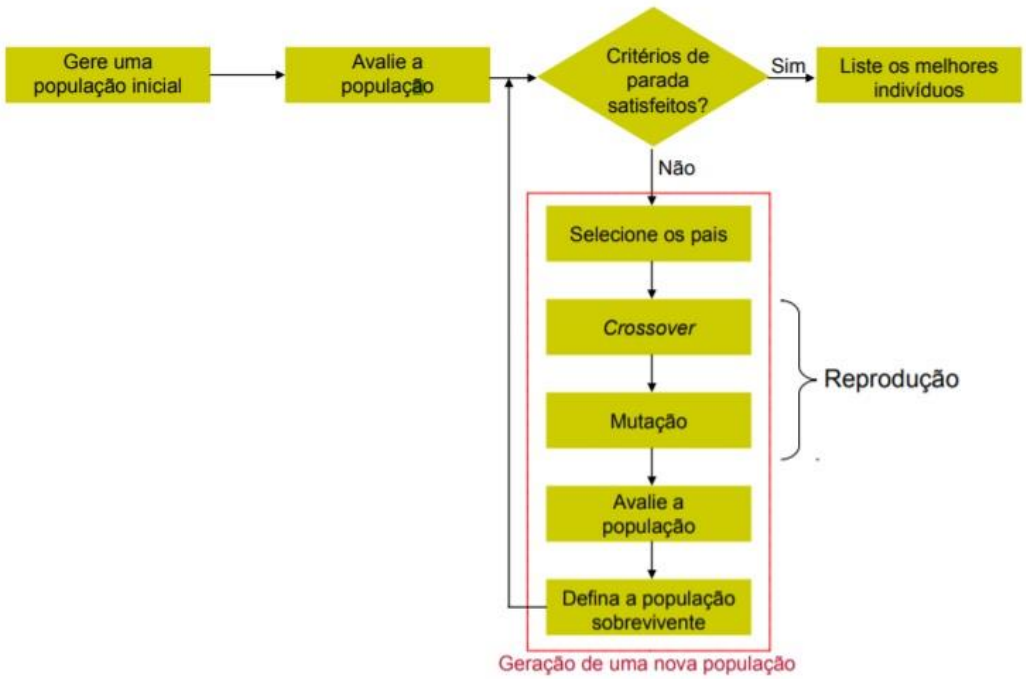
## 1. Introdução

Um **algoritmo genético** (**AG**) é uma técnica de busca utilizada na ciência da computação para achar soluções aproximadas em problemas de otimização e busca, fundamentado principalmente pelo americano John Henry Holland. Algoritmos genéticos são uma classe particular de algoritmos evolutivos que usam técnicas inspiradas pela biologia evolutiva como hereditariedade, mutação, seleção natural e recombinação (ou *crossover*).
O que é um AG:

Algoritmos genéticos diferem dos algoritmos tradicionais de otimização em basicamente quatro aspectos:

- Baseiam-se em uma codificação do conjunto das soluções possíveis, e não nos parâmetros da otimização em si;
- os resultados são apresentados como uma população de soluções e não como uma solução única;
- não necessitam de nenhum conhecimento derivado do problema, apenas de uma forma de avaliação do resultado;

## 2. Estrutura de um AG:




Problema da Mochila

O **problema da mochila** (em inglês, *Knapsack problem*) é um problema de optimização combinatória. O nome dá-se devido ao modelo de uma situação em que é necessário preencher uma mochila com objetos de diferentes pesos e valores. O objetivo é que se preencha a mochila com o maior valor possível, não ultrapassando o peso máximo.[1]

O problema da mochila é um dos 21 problemas NP-completos de Richard Karp, exposto em 1972. A formulação do problema é extremamente simples, porém sua solução é mais complexa. Este problema é a base do primeiro algoritmo de chave pública (chaves assimétricas).[2]

Normalmente este problema é resolvido com programação dinâmica, obtendo então a resolução exata do problema, mas também sendo possível usar PSE (procedimento de separação e evolução). Existem também outras técnicas, como usar algoritmo guloso[3], meta-heurística (algoritmos genéticos) para soluções aproximadas.

( **Problema da Mochila** ) Versão sequencial escrito em Phyton 3

Problema: Preencher uma mochila de capacidade limitada com objetos de diferentes pesos e valores cujo objetivo é preencher a mochila com o maior valor possível desde que não ultrapasse a capacidade máxima.

Exemplo:

| Item | 1 | 2 | 3 | 4 | 5 | 6 | 7 | N |
|---|---|---|---|---|---|---|---|---|
| Benefício | 5 | 8 | 3 | 2 | 7 | 9 | 4 | ... |
| Peso | 7 | 8 | 4 | 10 | 4 | 6 | 4 | ... |

Capacidade: 22

Adicione itens de modo a obter o máximo benefício

**0 - Solução utilizando algoritmos genéticos:**
Representação do indivíduo através de cromossomos de valores binários, considere 1 como iten pego e 0 como itens não pego:
Lista cromossomo

| Item | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| Cromossomo | 1 | 0 | 1 | 0 | 1 | 1 | 0 |
| Existe? | sim | não | sim | não | sim | sim | não |

Solução: 1010110 equivale a capacidade do individuo carregar os itens: 1, 3, 5, 6.

**3 Algoritmo:**

**Gerar uma população Inicial**

Entradas:

```
def gerar_pop(tam_pop, peso, cap_max):
```

a função retorna uma quantidade determinada de itens aleatórios de cromossomos onde cada individuo não ultrapassa a capacidade máxima da mochila.

**Avaliar a população**

Será reservado uma uma lista de análise para armazenar a quantidade total que cada individuo é capaz de armazenar. Exemplo considere uma população com 4 individuos, o resultado será respectivamente.

| Indice | Benefício | Peso |
|---|---|---|
| 1 | 35 | 20 |
| 2 | 15 | 21 |

| | | |
|---|---|---|
| 3 | 10 | 6 |
| 4 | 25 | 12 |

**Selecionar os individos aptos mais cruzamento:**

```
def crossover(populacao, peso, valor, tx_mutacao):
```

A função crossover, faz o cruzamento com os individuos aptos, neste caso a taxa de mutação está implementada junto com crossover, a taxa de mutação é uma porcentagem com relação a quantidade de itens da população. Os individuos que forem menor ou igual a capacidade máxima serão preservados para próxima geração, os demais individuos serão cruzados incluindo a mutação para aumentar a variedade genética. No final a nova população será ordenada por valor e peso respectivamente, sempre o primeiro individuo da lista será a melhor solução.

**Critério de parada:**

Após gerar a população inicial, será analizado e exibidos os melhores individuos da geração atual, se for definido uma quantidade determinada de gerações o algoritmo genético só irá ser encerrado quando for feito o cruzamento gerando novas populações até chegar ao número máximo de gerações definida pelo usuário. Para facilitar o entendimento considere o algoritmo abaixo:

```
max_geracao = 10

while (geracao_atual != max_geracao):
    if (geracao_atual != max_geracao):
        crossover(populacao, analise, tx_mutacao)
          geracao_atual +=1

print(populacao[0]) # Exibe 1 melhor : [valor, peso, cromossomo]
```

Um dos problemas do algoritmo genético, é quando a quantidade de entradas aumenta, o tempo de processamento aumenta exponencialmente, uma solução seria paralelizar o algoritmo para dividir as tarefas com o intuito de diminuir o tempo de processamento. No final deste texto será apresentado uma comparação de tempo com relação a versão sequencial e a versão paralela do problema da mochila solucionado em python.

**4 - Versão paralela do problema da mochila (biblioteca  multithread)**

Analizando o problema da mochila foi observado que a função que exige maior esforço computacional é o cruzamento, então uma solução para paralelizar o cruzamento será dividir a quantidade de itens da população pela quantidade de threads definido pelo usuário (de 1 a 6). Considere o cabeçalho da função paralelizada:

```
import threading
def crossover(populacao, tx_mutacao, peso, valor, ini, fim):
```

a função retorna uma nova lista ordenada resultado do cruzamento + mutação

dos melhores individuos, o conteúdo a ser calculado será a parte inicial da população até parte final da população definida pela divisão do tamanho da população pela quantidade de threads. Considere o exemplo para 2 threads:

```
itens = tam_pop/processos
i0 = 0
i1=int(i0+itens)
i2=int(i1+itens)

t1 = threading.Thread(target=crossover, args=(populacao, tx_mutacao, peso, valor, i0, i1))

t2 = threading.Thread(target=crossover, args=(populacao, tx_mutacao, peso, valor, i1, i2))

t1.start()
t2.start()

t1.join()
t2.join()
```

onde i0 e i1 é o intervalo a ser calculado pelo processo 1 dos individuos da população, o processo 2 será calculado no intervalo de i1 a i2 e assim respectivamente.

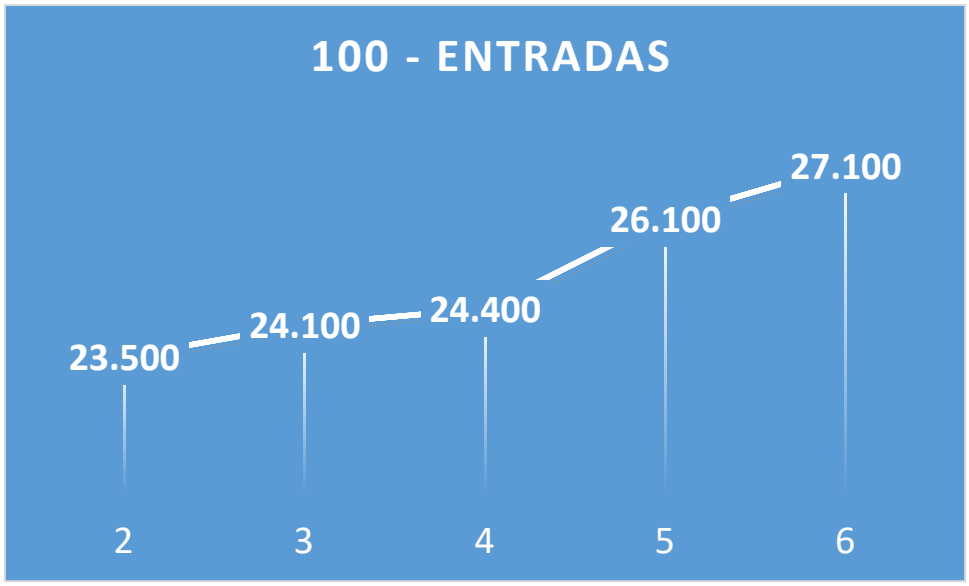
**4 - Teste de desempenho (Metricas)**

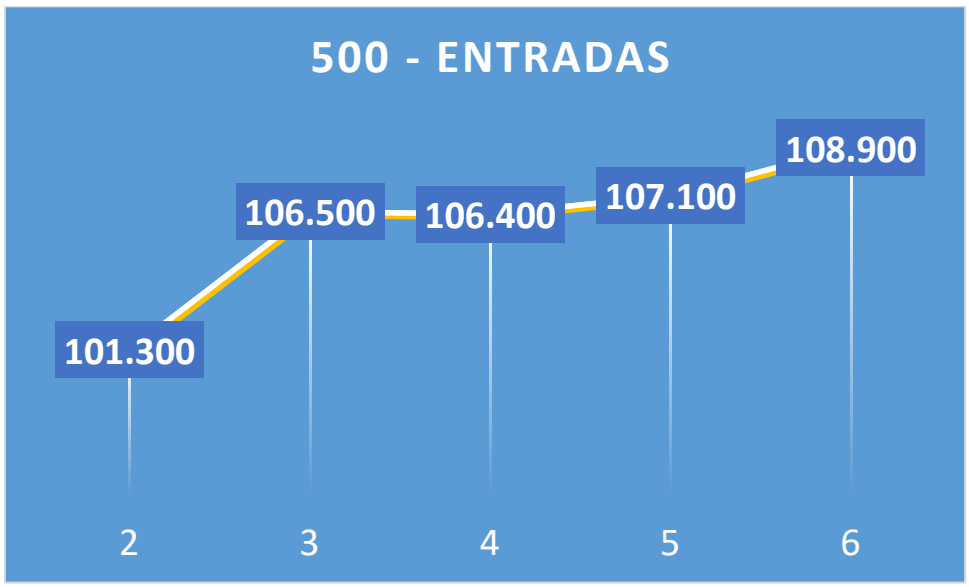
- **Tempo em Minutos**

| Entradas | SEQ | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 100 | 0.405 | 0.392 | 0.402 | 0.407 | 0.435 | 0.452 |
| 500 | 1.975 | 1.688 | 1.775 | 1.773 | 1.785 | 1.815 |
| 1000 | 4.185 | 3.802 | 3.785 | 3.750 | 3.775 | 3.810 |
| 2000 | 8.458 | 7.372 | 7.337 | 7.975 | 7.458 | 7.993 |
| 5000 | 18.605 | 16.723 | 16.635 | 16.692 | 17.008 | 18.303 |
| 10000 | 40.633 | 40.103 | 39.303 | 38.585 | 40.240 | 40.597 |

Valores para versão paralela e squencial:
População: 2000
Gerações: 500
Tx Mutação 5

**Análise gráfica (Tempo x Threads):**

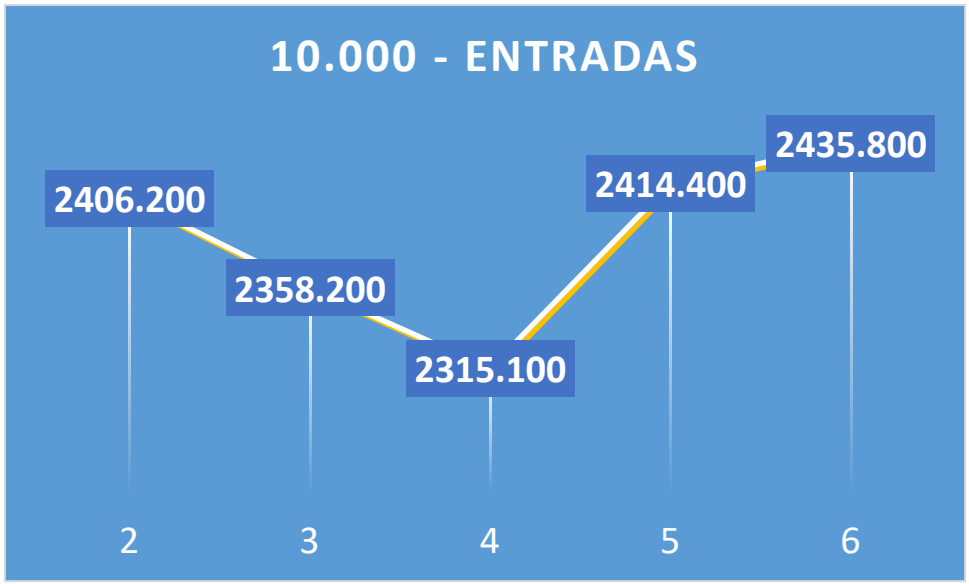

100 - ENTRADAS
23.500
24.100
24.400
26.100
27.100
2
3
4
5
6


500 - ENTRADAS
101.300
106.500
106.400
107.100
108.900
2
3
4
5
6


1000 - ENTRADAS
228.100
227.100
225.000
226.500
228.600
2
3
4
5
6


2.000 - ENTRADAS
228.100
227.100
225.000
226.500
228.600
2
3
4
5
6


5.000 - ENTRADAS
228.100
227.100
225.000
226.500
228.600
2
3
4
5
6


10.000 - ENTRADAS
2406.200
2358.200
2315.100
2414.400
2435.800
2
3
4
5
6

## 5 - Eficiência e Speedup

$$s(p) = \frac{T(1)}{T(p)}$$

$$E(p) = \frac{S(p)}{p}$$

T(1) : Tempo de execução 1 Processador
T(o): Tempo de execução p processadores

| Eficiência | | | | | | |
|---|---|---|---|---|---|---|
| Entradas | | 2 | 3 | 4 | 5 | 6 |
| 100 | | 0.517 | 0.336 | 0.249 | 0.186 | 0.149 |
| 500 | | 0.585 | 0.371 | 0.278 | 0.221 | 0.181 |
| 1000 | | 0.550 | 0.369 | 0.279 | 0.222 | 0.183 |
| 2000 | | 0.574 | 0.384 | 0.265 | 0.227 | 0.176 |
| 5000 | | 0.556 | 0.373 | 0.279 | 0.219 | 0.169 |
| 10000 | | 0.507 | 0.345 | 0.263 | 0.202 | 0.167 |

| Speedup | | | | | | |
|---|---|---|---|---|---|---|
| Entradas | | 2 | 3 | 4 | 5 | 6 |
| 100 | | 1.034 | 1.008 | 0.996 | 0.931 | 0.897 |
| 500 | | 1.170 | 1.113 | 1.114 | 1.106 | 1.088 |
| 1000 | | 1.101 | 1.106 | 1.116 | 1.109 | 1.098 |
| 2000 | | 1.147 | 1.153 | 1.061 | 1.134 | 1.058 |
| 5000 | | 1.113 | 1.118 | 1.115 | 1.094 | 1.016 |
| 10000 | | 1.013 | 1.034 | 1.053 | 1.010 | 1.001 |

## 6 - Conclusões e trabalhos futuros:

Um dos problemas de programar com threads em python é que o interpretador usa o GIL (Global Interpreter Lock) garantindo que apenas que uma thread tenho acesso a um recurso específico, a justificativa desta limitação é evitar problemas de gerenciamento de memória, por tanto com threads em python não dá pra usar efetivamente 100% dos recursos do processador, foi por este motivo que os resultados de desempenho não foram tão satisfatórios, uma solução para acessar efetivamente o recurso dos processadores seria usar a biblioteca multiprocessing, que ao invés de trabalhar threads, são utilizado processos que trabalham independentemente, a desvantagem é que não compartilham dados globais, é necessário sincronizar os dados por meio de filas por isso dependendo de como o codigo for escrito por melhorar ou não o desempenho.

Referências:

https://mathworld.wolfram.com/KnapsackProblem.html

https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/mochila-bool.html

Slides Prof. Bruno UERN:  <PP-AG-Mochila, PP-Aula-08>