Universidade de São Paulo
Instituto de Física de São Carlos

Problema da Mochila

Otimização combinatória

Professor:
Luciano da Fontoura Costa

Aluno:
Vinícius Sousa Dutra (13686257)

7 de dezembro de 2023

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Códigos

1 Definição

Formalmente podemos considerar o problema como sendo

$$maximize \sum v_i x_i \tag{1}$$

$$subject\,to \sum w_i x_i \leq W \tag{2}$$

Considerando que v_i seja um conjunto de valores, w_i seja um conjunto de pesos, W seja o peso total e x_i seja o número de itens escolhidos, temos um valor máximo c_i associado a cada item.

Dado que todas as demais variáveis são fixas, a variável real no contexto do problema é x_i. O conjunto de soluções cresce de maneira exponencial, quando consideramos o caso mais simples no qual apenas um exemplar de cada item é permitido, o número de soluções viáveis é dado por 2^{n-1}, onde n é o numero total de itens.

Isso ocorre devido ao fato de que o número total de soluções possíveis é 2^n, e para cada solução não viável X, a sua solução complementar $\overline{X}$ é uma solução viável.

No caso que iremos estudar com $N = 1000$ e com o número de exemplares variados, o limite inferior de soluções é de 10^{301}. Um limite superior para o conjunto específico de dados usados foi calculado realizando a seguinte operação

$$\log(L_{superior}) = \sum \log(c_i + 1) \tag{3}$$

$$L_{superior} \approx 10^{734} \tag{4}$$

2 Geração de Dados

Os dados são gerados de maneira aleatória usando esse pequeno script Python apresentado abaixo

Código 1: Geração do csv

```python
import pandas as pd
from numpy.random import randint
N=1000
values=randint(0,100,size=N)
sizes=randint(1,10,size=N)
max_items=randint(1,10,size=N)
data = {'values': values, 'sizes': sizes, 'items': max_items}
df = pd.DataFrame(data)
df.to_csv('data.csv', index=False)
```

3 Busca aleatória

A busca aleatória consiste em criar uma sequência de N zeros. Em seguida, são escolhidos índices de forma aleatória, e para cada índice, o espaço vazio é preenchido com um número aleatório de itens selecionados. O processo é pausado até que o peso máximo seja atingido. Em cada iteração, a melhor solução encontrada é guardada.

Código 2: Busca aleatória

```
value = 0
track_value = np.zeros(MAX_ITERATIONS, dtype=int)
for i in range(MAX_ITERATIONS):
    indexes = np.random.choice(range(0, len(df)), size=len(df), replace=False)
    seq = np.zeros(len(df), dtype=int)
    temp_weight = 0
    for index in indexes:
        rand_quantity = np.random.randint(0, df['items'][index])
        extra_weight = df['sizes'][index] * rand_quantity
        temp_weight += extra_weight
        if temp_weight < CAPACITY:
            seq[index] = rand_quantity
        else:
            break
    new_value = total_value(seq)
    if value < new_value:
        value = new_value
    track_value[i] = value
assert (
    np.all(seq <= df['items']) and total_size(seq) < CAPACITY
), 'A Solução não é válida'
```

4 Simulated Annealing

O Simulated Annealing utilizado é semelhante ao utilizado no Problema do Caixeiro Viajante (TSP - Traveling Salesman Problem). As soluções vizinhas são encontradas realizando três operações básicas.

- Adição aleatória de item
- Remoção aleatória de item
- Swap de índices

Para preservar a viabilidade da solução, é sempre verificado se a operação é permitida. No final de ambos os códigos, é verificado se a melhor solução é viável.

Código 3: Simulated Annealing

```
value = np.zeros(MAX_ITERATIONS, dtype=int)
value_seq = total_size(seq)
for index in range(MAX_ITERATIONS):
    temperature = update_temperature(index)
    random_sub, random_add = np.random.randint(len(df), size=2)
    new_seq = np.copy(seq)
    # adding and subtract
    if new_seq[random_sub] > 0:
        new_seq[random_sub] -= 1
    if new_seq[random_add] < df['items'][random_add]:
        new_seq[random_add] += 1
    # random swap
    index_a, index_b = np.random.randint(len(df), size=2)
    if new_seq[index_a] < df['items'][index_b]:
        if new_seq[index_b] < df['items'][index_a]:
            new_seq[index_a], new_seq[index_b] = (
                new_seq[index_b],
                new_seq[index_a],
            )
    fitness_newseq = total_value(new_seq)
    if total_size(new_seq) < CAPACITY:
        if fitness_newseq > value_seq:
            seq = new_seq
            value_seq = fitness_newseq
        else:
            deltaE = fitness_newseq - value_seq
            if np.exp(deltaE / temperature) > np.random.random():
                seq = new_seq
                value_seq = fitness_newseq
    value[index] = value_seq
assert (
    np.all(seq <= df['items']) and total_size(seq) <= CAPACITY
), 'A Solução não é válida'
```

5 Conclusão

Ambos os algoritmos foram testados no mesmo conjunto de dados com o mesmo número de iterações. Observou-se que, para poucas iterações, os algoritmos são equivalentes. Nas figuras abaixo, observou-se que o método aleatório teve um desempenho um pouco melhor do que o Simulated Annealing (SA).

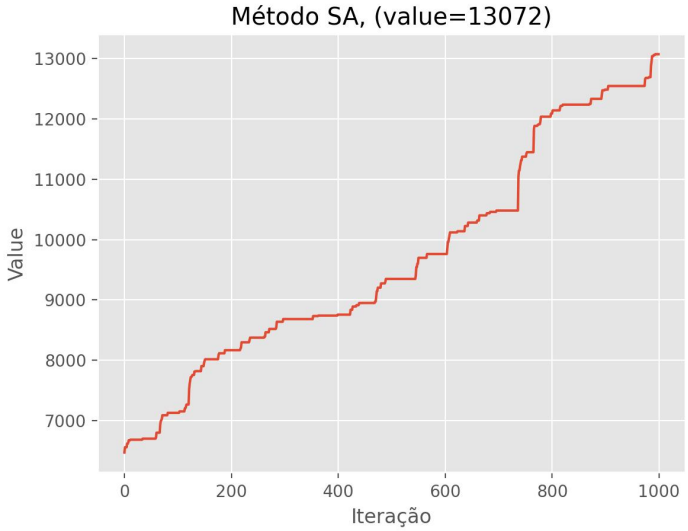

Figura 1: Simulated Annealing, 10^3 iterações

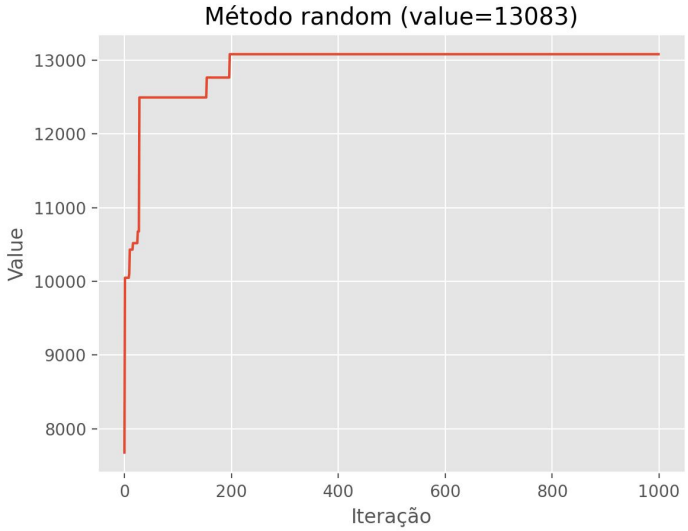

Figura 2: Busca aleatória, 10^3 iterações

No entanto, para 10^5 iterações, é visível uma estagnação da busca aleatória. Isso ocorre porque, como discutido, o conjunto de soluções é muito grande para encontrar boas soluções aleatoriamente. Nesse caso, o Simulated Annealing (SA) encontrou soluções cerca de 3 vezes melhores.

Também foi observada uma diferença significativa no tempo de execução, em que o SA teve um tempo de execução de $8s$ e a busca aleatória de $120s$. Como o SA altera soluções já existentes em vez de criar novas em cada iteração, ele acaba executando mais rapidamente.

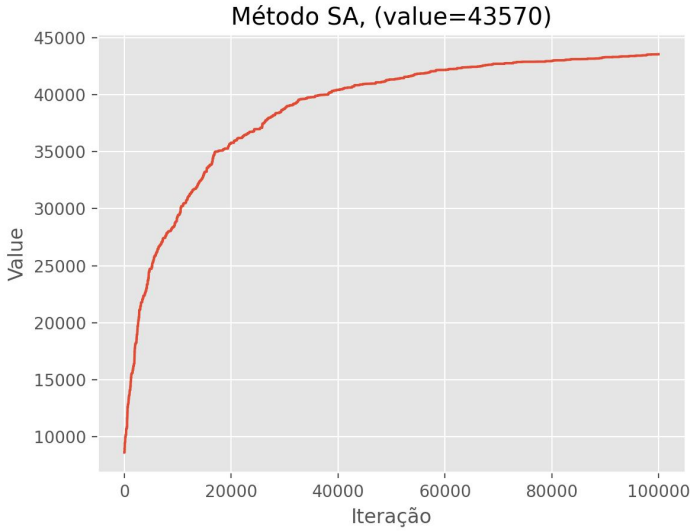

Figura 3: Simulated Annealing, 10^5 iterações

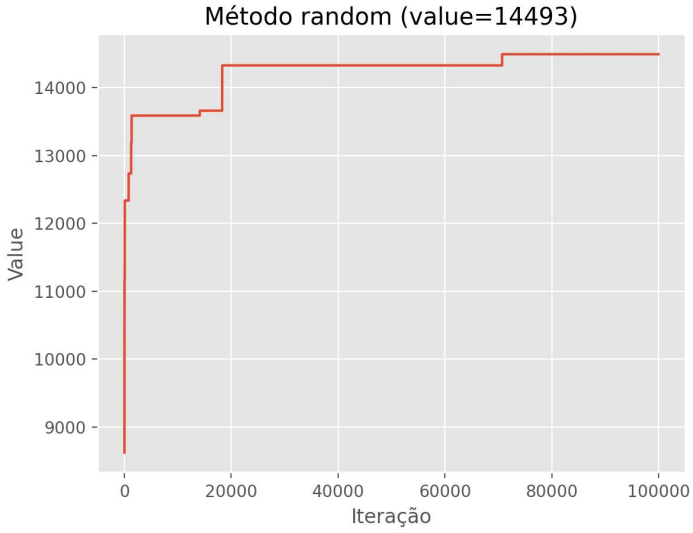

Figura 4: Busca aleatória, 10^5 iterações