Kanapsack Problem databases 02:

- https://people.sc.fsu.edu/~jburkardt/datasets/knapsack_01/knapsack_01.html (https://people.sc.fsu.edu/~jburkardt/datasets/knapsack_01/knapsack_01.html)

Atividade com nota:

- Avaliar o algoritmo Hill Climbing para as bases P01 a P07;
- Utilizar a função de aptidão knapsack do Mlrose;
- Apresentar a melhor solução encontrada e comparar com a melhor solução global disponível para a base de dados

In [1]:
```
!pip install mlrose
```

```
Requirement already satisfied: mlrose in /home/a10/anaconda3/lib/py
thon3.11/site-packages (1.3.0)
Requirement already satisfied: numpy in /home/a10/anaconda3/lib/pyt
hon3.11/site-packages (from mlrose) (1.24.3)
Requirement already satisfied: scipy in /home/a10/anaconda3/lib/pyt
hon3.11/site-packages (from mlrose) (1.11.3)
Requirement already satisfied: sklearn in /home/a10/anaconda3/lib/p
ython3.11/site-packages (from mlrose) (0.0.post10)
```

In [2]:
```
import six
import sys
sys.modules['sklearn.externals.six'] = six
import mlrose
import numpy as np
```

Problem representation:

0/1 knapsack problem using a hill climbing algorithm for P02.

- The Problem: Given a set of items, each with a weight and a value, select a subset of the items to maximize the total value while keeping the total weight within a given capacity.

So the total_weight of any system configuration (any state) is constraint with a max_weight, I will call this max_weight W.

In [3]:
```
max_weight = 26
weights = [12, 7, 11, 8, 9]
values = [24, 13, 23, 15, 26]
state = np.array([0, 1, 1, 1, 0])
len(weights)
```

Out[3]:
```
5
```

In [4]:
```
items = []
for i in range(len(weights)):
    items.append(('Item' + str(i), weights[i], values[i]))

items
```

Out[4]:
```
[('Item0', 12, 24),
 ('Item1', 7, 13),
 ('Item2', 11, 23),
 ('Item3', 8, 15),
 ('Item4', 9, 26)]
```

This get_cost function is what I want to maximize. It's used to see which neighbor is better.

In [5]:
```
# Define the fitness function
def get_cost(state):
    total_value = sum(state[i] * values[i] for i in range(len(weights
    total_weight = sum(state[i] * weights[i] for i in range(len(weigh
    if total_weight > max_weight:
        return 0
    return total_value
```

In [6]:
```
#sol otima
#test_state = [0, 0, 1, 1, 0]
#state = [0, 1, 1, 1, 0]
#total_value = get_cost(test_state)
#print(f'{total_value} - weight: {total_weight} ')
total_value = get_cost(state)
print(f'profit: {total_value} ')
```

```
profit: 51
```

Which is nothing but this in an equation form:

$F(x_i) = \sum_{i=0}^{n-1} x_i v_i$, if $\sum_{i=0}^{n-1} x_i w_i \leq W$ this is the constraint

Where x_i represents a state vector $x = [x_0, x_1, \ldots, x_{n-1}]$. It denotes a number of copies of item i included in the knapsack.

In [7]:
```
# custom fitness function object
fitness_cust = mlrose.CustomFitness(get_cost)
```

In [8]:
```
#create enviroment
problem = mlrose.DiscreteOpt(length = 5, fitness_fn = fitness_cust, m
```

Hill Climb Method

- Main Idea: The algorithm iteratively improves the solution by exploring neighboring states and selecting the state with the highest value (accordingly with get_cost) until no better solution can be found or a maximum number of iterations is reached (100 below). In other words, this algorithm maintain a single node and searches by moving to a neighboring node.

In [9]:
```
# Call the hill_climb function to solve the problem
best_state, best_fitness, curve = mlrose.hill_climb(problem, max_ite
best_state, best_fitness, curve
```

Out[9]:
```
(array([0, 1, 1, 1, 0]), 51.0, array([], dtype=float64))
```

In [10]:
```
get_cost(best_state)
```

Out[10]:
```
51
```

In [11]:
```
print("SolucaoHC:  ", best_state)
print("Fitness Value:", best_fitness)
```

```
SolucaoHC:   [0 1 1 1 0]
Fitness Value: 51.0
```

Introducing Random Restart

Randomness should improve the optimization process value. Because for each iteration, it explores neighboring states by flipping the value (0 to 1 or 1 to 0) of a randomly selected item, and evaluates their costs and it keeps track of the best neighboring states with the highest cost.The current state is then updated to one of the best neighboring states. The process continues until no better neighbor is found or the maximum number of iterations is reached.

In [12]:
```
best_state, best_fitness, curve = mlrose.random_hill_climb(problem, r
best_state, best_fitness, curve
```

Out[12]:
```
(array([0, 1, 0, 1, 1]),
 54.0,
 array([39., 39., 39., 39., 39., 39., 39., 39., 39., 39., 41.,
41.,
        41., 41., 41., 54., 54., 54., 54., 54., 54., 54., 54.,
54.,
        54., 28., 28., 54., 54., 54., 54., 54., 54., 54., 54.,
54.,
        54., 50., 50., 50., 50., 50., 50., 50., 50., 50., 50.,
13.,
        13., 39., 39., 39., 39., 54., 54., 54., 54., 54., 54.,
54.,
        54., 54., 54., 41., 54., 54., 54., 54., 54., 54., 54.,
54.,
        54., 54., 39., 39., 39., 39., 39., 39., 39., 39., 39.,
54.,
        54., 54., 54., 54., 54., 54., 54., 54., 54., 54., 51., 51.,
51.,
        51., 51., 51., 51., 51., 51., 51., 51.,  0.,  0.,  0.,  0.,
47.,
        47., 47., 47., 47., 47., 47., 47., 47., 47., 28., 28.,
51.,
        51., 51., 51., 51., 51., 51., 51., 51., 51., 37., 37.,
37.,
        37., 37., 37., 37., 37., 37., 37., 37.]))
```

Conclusion

Rather than just hill climb one time, we can hill climb multiple times and figure out what is the best one that I've been able to find. So we've implemented a function for random restart that restarts some maximum number of times, and repeat, on that number of times, this hill-climb method 'random_hill_climb' and figure out what the cost is (for that state after randomly restart maximum times). As we can see there is a lot of local maximums (39, 41, 51...), the algorithm starts at 39 and runs through a lot of neighbors, always keeping track (even the lowest ones) of the best neighboring states with the highest cost. 54 is the global maximum for this dataset.

Simulated Annealing

In order to find a global maximum or global minimum we may need to make a move that makes our situation worse. Because sometimes if we get stuck in a local max/min, I want to dislodge from that in order to find the global max/min.

- Simulated Annealing is a great technique for that. Suppose some state-space, if I am in a current state and I'm looking for a global maximum and I'm trying to maximize the value of the state, Hill-Climbing would just take the state and look at the two neighbor ones, and always pick the one that is going to increase the value of the state. As said before "in order to find a global maximum/minimum we may need to make a move that makes our situation worse." such that later on we can find that global maximum. Once found this global max state we probably don't wan't to be moving to states worse than current state. This is why this metaphor for annealing -> start making more random moves and, over time, start to make fewer of those random moves based on a particular temperature schedule.

In [13]:
```
# Define a schedule for simulated annealing (you can customize this s
schedule = mlrose.ExpDecay(init_temp=1000, exp_const=0.04, min_temp=1
```

the idea is that this temperature is going to be higher early on and lower later on. It is exponentially decaying according to the formula: $T(t) = T_i e^{-rt}$ where:

- T(t) is the temperature at time t.
- $T(i)$ is the initial temperature at time t=0.
- r is the rate of exponential decay.
- t is the time variable.
- The exponential decay is represented by e^{-rt}.

For ex: You start off T_i=100, after 1s the temperature will be:

In [16]:
```
# Evaluate the temperature parameter at time t = 1 and exp_const=0.04
t1 = schedule.evaluate(1)
print(f"T(1) = {t1}")
```

```
T(1) = 960.7894391523232
```

In [15]:
```
# Call the simulated_annealing function to solve the problem
best_state, best_fitness, curve = mlrose.simulated_annealing(problem,
best_state, best_fitness, curve
```

Out[15]:
```
(array([0, 1, 0, 1, 1]),
 54.0,
 array([ 0.,  0., 39.,  0.,  0.,  0.,  0.,  0.,  0.,  0., 47.,  0.,
0.,
         0.,  0.,  0.,  0.,  0.,  0.,  0., 36., 13.,  0., 23., 36.,
0.,
         0.,  0.,  0.,  0., 37.,  0., 39., 13., 36., 13., 36.,  0.,
39.,
        39.,  0., 36., 13., 13.,  0., 23., 49., 49.,  0., 41., 41.,
54.,
        41.,  0.,  0.,  0., 47., 24.,  0., 23., 49., 23., 47.,  0.,
0.,
         0.,  0.,  0.,  0., 51.,  0.,  0.,  0.,  0.,  0., 54., 54.,
54.,
        54., 54., 54., 28., 54., 54., 41.,  0.,  0.,  0.,  0.,  0.,
0.,
         0., 28., 28., 13., 28., 15., 38., 38., 51., 38., 15.,  0.,
26.,
        41., 41., 41., 41., 41., 41., 41., 41., 41., 41., 41., 41.,
41.,
        41., 54., 54., 54., 54., 54., 54., 54., 54., 54., 54., 54.,
54.,
        54., 54., 54., 54., 54., 54., 54., 54., 54., 54., 54., 54.,
54.,
        54., 54., 54., 54., 54., 54., 54., 54., 54., 54., 54., 54.,
54.,
        54., 54., 54., 54., 54., 54., 54., 54., 54., 54., 54., 54.,
54.,
        54., 54., 54., 54., 54., 54., 54., 54., 54., 54., 54., 54.,
54.,
        54., 54., 54., 54., 54., 54., 54., 54., 54., 54., 54., 54.,
54.,
        54., 54., 54., 54., 54., 54., 54., 54., 54., 54., 54., 54.,
54.,
        54., 54., 54., 54., 54., 54., 54., 54., 54., 54., 54.]))
```

Conclusion

So the goal of this whole process is as we begin our search to find the local/global max/min, we can dislodge ourselves if we get stuck at a local max/min in order to eventually make our way to exploring the part of the state space that is going to be the best. And then as the temperature decreases, start to make fewer of those random moves, that is, not moving aroung too much to get into another part of the state space.

The simulated annealing algorithm ran through numerous iterations to find an optimal combination of items to include in a knapsack. The curve data indicates how the solution evolved over iterations, with the algorithm converging on the optimal solution towards the end.

Important: In order for finding the global maximum, I had to **adjust the parameters** in the ExpDecay schedule and in the simulated_annealing function.

- **Best State**: [0, 1, 0, 1, 1] This is an array representing the optimal solution found by the algorithm.
- **Best fitness**: 54.0 says that the combination of items in the best state yields a total value of 54.
- **Curve**: This is an array showing the fitness score at each iteration of the algorithm. It helps in understanding how the solution improved over time. The values fluctuate, indicating the exploration of different solutions. The repeated values of 54 towards the end suggest that the algorithm consistently found a solution with a fitness of 54,

In [ ]: