Kanapsack Problem databases 01:

- 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 [71]:
```
!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 [72]:
```
import six
import sys
sys.modules['sklearn.externals.six'] = six
import mlrose
import numpy as np
```

0/1 knapsack problem using a hill climbing algorithm for P01.

- 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 [73]:
```
max_weight = 165
weights = [23, 31, 29, 44, 53, 38, 63, 85, 89, 82]
values = [92,57,49,68,60,43,67,84,87,72]
state = np.array([1, 1, 1, 1, 0, 1, 0, 0, 0, 0])
len(weights)
```

Out[73]:
```
10
```

In [74]:
```
items = []
for i in range(len(weights)):
    items.append(('Item' + str(i), weights[i], values[i]))

items
```

Out[74]:
```
[('Item0', 23, 92),
 ('Item1', 31, 57),
 ('Item2', 29, 49),
 ('Item3', 44, 68),
 ('Item4', 53, 60),
 ('Item5', 38, 43),
 ('Item6', 63, 67),
 ('Item7', 85, 84),
 ('Item8', 89, 87),
 ('Item9', 82, 72)]
```

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

In [75]:
```
# 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
```

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 [76]:
```
#sol otima
#test_state = [0, 0, 1, 1, 1, 0, 1, 1, 0, 0]
#state      = [1, 1, 1, 1, 0, 1, 0, 0, 0, 0]
#total_value = get_cost(test_state)
#print(f'{total_value}')
total_value = get_cost(state)
print(f'profit: {total_value} ')
```

```
profit: 309
```

In [77]:
```
# custom fitness function object
fitness_cust = mlrose.CustomFitness(get_cost)
```

In [78]:
```
#creating enviroment
problem = mlrose.DiscreteOpt(length = 10, fitness_fn = fitness_cust,
```

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 [79]:
```
# 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[79]:
```
(array([1, 1, 1, 1, 0, 1, 0, 0, 0, 0]), 309.0, array([], dtype=floa
t64))
```

In [80]:
```
# Evaluate the fitness of the given state
get_cost(best_state)
```

Out[80]:
```
309
```

In [81]:
```
print("SolucaoHC: ", best_state)
print("Fitness Value:", best_fitness)
```

```
SolucaoHC:  [1 1 1 1 0 1 0 0 0 0]
Fitness Value: 309.0
```

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 [82]:
```
best_state, best_fitness, curve = mlrose.random_hill_climb(problem, r
best_state, best_fitness, curve
```

Out[82]:
```
(array([1, 1, 1, 1, 0, 1, 0, 0, 0, 0]),
 309.0,
 array([  0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
0.,
        179., 179., 179., 179., 179., 179., 179., 179., 179., 179.,
179.,
        252., 252., 252., 252., 252., 252., 252., 309., 309., 309.,
309.,
        309., 309., 309., 309., 309., 309., 309.,   0.,   0.,   0.,
0.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
0.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
0.,
          0.,   0.,   0.,   0., 198., 266., 266., 266., 266., 266.,
309.,
        309., 309., 309., 309., 309., 309., 309., 309., 309., 309.,
0.,
          0.,   0.,   0.,   0.,   0.,   0., 217., 217., 217., 217.,
217.,
        217., 309., 309., 309., 309., 309., 309., 309., 309., 309.,
309.,
        309.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
0.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
0.,
        178., 178., 178., 178., 270., 270., 270., 270., 270., 270.,
270.,
        270., 270., 270., 270.]))
```

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 (179, 252, 266...), the algorithm starts at 0 and runs through a lot of neighbors, always keeping track (even the lowest ones) of the best neighboring states with the highest cost. 309 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 [83]:
```
# Define a schedule for simulated annealing (you can customize this s
schedule = mlrose.ExpDecay(init_temp=1000, exp_const=0.01, 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 [103]:
```
# Evaluate the temperature parameter at time t = 1 and exp_const=0.01
t1 = schedule.evaluate(1)
print(f"T(1) = {t1}")
```

```
T(1) = 990.0498337491681
```

In [104]:
```
# Call the simulated_annealing function to solve the problem
best_state, best_fitness, curve = mlrose.simulated_annealing(problem,
best_state, best_fitness, curve
```

Out[104]:
```
(array([1, 1, 1, 1, 0, 1, 0, 0, 0, 0]),
 309.0,
 array([  0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
0.,
         0.,   0.,   0.,   0.,   0., 181., 181.,   0.,   0.,   0.,
0.,
         0.,   0., 155., 212.,   0., 217.,   0.,   0.,   0., 236.,
0.,
         0.,   0., 236.,   0.,   0.,   0., 209., 258., 166., 106.,
149.,
        92., 176., 176.,   0., 159., 251.,   0.,   0.,   0.,   0.,
0.,
         0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
0.,
         0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0., 147.,
0.,
         0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
0.,
         0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
0.,
         0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
0.,
         0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
0.,
       244.,   0.,   0.,   0., 244., 244., 244., 244., 244., 244.,
244.,
       152., 152., 152., 220.,   0., 234., 234., 174.,   0.,   0.,
0.,
         0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
0.,
         0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
0.,
         0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
0.,
         0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
0.,
         0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
0.,
         0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
0.,
         0.,   0.,   0., 204., 136., 136.,   0.,   0.,   0.,   0.,
0.,
         0.,   0.,   0.,   0., 130., 130., 130.,  87., 147.,   0.,
0.,
         0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
0.,
         0.,   0.,   0.,   0.,   0., 178., 178., 106., 149., 216.,
216.,
       216., 149., 149., 241., 309., 309., 309., 241., 198., 198.,
198.,
       198., 141., 141., 201., 141., 225., 225., 176., 176., 176.,
176.,
       225., 225., 225., 225., 225., 225., 225., 225., 225., 225.,
```

```
225.,
       225., 225., 225., 225., 225., 225.,   0.,   0., 270., 270.,
270.,
       221., 221., 221., 221., 221., 221., 221., 221., 221., 221.,
221.,
       221., 149., 192., 241., 192., 241., 309., 309., 309., 309.,
309.,
       309., 309., 309., 309., 309., 309., 309., 309., 309., 309.,
309.,
       309.  309.  309.  309.  309. ]))
```

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**: [1, 1, 1, 1, 0, 1, 0, 0, 0, 0] This is an array representing the optimal solution found by the algorithm.
- **Best fitness**: 309 says that the combination of items in the best state yields a total value of 309.
- **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 309 towards the end suggest that the algorithm consistently found a solution with a fitness of 309, indicating it likely reached an optimal or near-optimal solution

Bonus: The Hill Climb algorithm w/o mlrose

0/1 knapsack problem using a hill climbing algorithm for P01.

- 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.
- Main idea: The algorithm iteratively improves the solution by exploring neighboring states and selecting the state with the highest value until no better solution can be found or a maximum number of iterations is reached. The log parameter is used to log the progress during the execution of the algorithm. In other words, this algorithm maintain a single node and searches by moving to a neighboring node

In [92]:
```
#take a current state, based on this, take the best neighbor or set c
import random


class KnapsackSolver:
    def __init__(self, weights, values, capacity):  # args we want to
        self.weights = weights
        self.values = values
        self.capacity = capacity
        self.n = len(weights)

        # Initialize state with a given binary representation
        self.state = [1, 1, 1, 1, 0, 1, 0, 0, 0, 0]  # Provided initi

    #this cost is used to see which neighbor is better/ This method k
    #so I want to maximize it
    def get_cost(self, state): # Calculate the total value of the sel
        total_value = sum(state[i] * self.values[i] for i in range(se
        # If the total weight exceeds the capacity, return a negative
        total_weight = sum(state[i] * self.weights[i] for i in range(
        if total_weight > self.capacity:
            return -total_value
        return total_value

    def get_neighbors(self, state):
        # Generate neighboring states by flipping the value (0 to 1 c
        neighbors = []
        for i in range(self.n):
            neighbor = state.copy()
            neighbor[i] = 1 - neighbor[i]  # Flip the value
            neighbors.append(neighbor)
        return neighbors

    #HERE IS THE MAIN IDEA WHERE I WANT TO MAXIMIZE THE GET_COST FUNC
    def hill_climb(self, maximum=None, log=False):
        count = 0
        """function hill-climb(problem):
              current = initial state of problem
              repeat:
                  neighbor = highest valued neighbor of current
                  if neighbor not better than current:
                      return current
                  current = neighbor #if it's better
        """


        while maximum is None or count < maximum: #could specify maxi
            count += 1 ##number of iterations
            best_neighbors = []
            best_neighbor_cost = None
            #consider all neighbors for that state and the cost to it
            for neighbor in self.get_neighbors(self.state):
                cost = self.get_cost(neighbor)
                #checks if neighbor is best so far
                if best_neighbor_cost is None or cost > best_neighbor
                    best_neighbor_cost = cost #update and
                    best_neighbors = [neighbor] #keep track of best n
                elif best_neighbor_cost == cost:
                    best_neighbors.append(neighbor)
```

```
            if best_neighbor_cost <= self.get_cost(self.state):
                #best neighbor is worse than current state/ no better
                return self.state

            # Move to a highest-valued neighbor because above
            self.state = random.choice(best_neighbors)

            if log:
                print(f"Iteration {count}: Cost {best_neighbor_cost}"

        return self.state
```

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 [93]:
```
# Example usage with p01 parameters
max_weight = 165
weights = [23, 31, 29, 44, 53, 38, 63, 85, 89, 82]
values = [92,57,49,68,60,43,67,84,87,72]
initial_state = [1, 1, 1, 1, 0, 1, 0, 0, 0, 0]
```

In [94]:
```
solver = KnapsackSolver(weights, values, max_weight)
best_solution = solver.hill_climb(log=True)
```

In [95]:
```
print("Best Solution:", best_solution)
print("Best Value:", solver.get_cost(best_solution))
```

```
Best Solution: [1, 1, 1, 1, 0, 1, 0, 0, 0, 0]
Best Value: 309
```

0/1 knapsack problem using a hill climbing algorithm with random restart for P05

In [96]:
```
#rather than just hill climb one time, we can hill climb multiple tim
#So here I've implemented a function for random restart that restarts
#and repeat, on that number of times, this hill-climb method 'self.hi
import random


class KnapsackSolver:
    def __init__(self, weights, values, capacity):
        self.weights = weights
        self.values = values
        self.capacity = capacity
        self.n = len(weights)

        # Initialize state with a random binary representation (0 or
        self.state = [random.randint(0, 1) for _ in range(self.n)]

    def get_cost(self, state):
        # Calculate the total value of the selected items
        total_value = sum(state[i] * self.values[i] for i in range(se
        total_weight = sum(state[i] * self.weights[i] for i in range(
        # If the total weight exceeds the capacity, return a negative
        if total_weight > self.capacity:
            return -total_value
        return total_value

    def get_neighbors(self, state):
        # Generate neighboring states by flipping the value (0 to 1 o
        neighbors = []
        for i in range(self.n):
            neighbor = state.copy()
            neighbor[i] = 1 - neighbor[i]  # Flip the value
            neighbors.append(neighbor)
        return neighbors

    def hill_climb(self, maximum=None, log=False):
        count = 0
        best_state = self.state

        while maximum is None or count < maximum:
            count += 1
            best_neighbors = []
            best_neighbor_cost = None

            for neighbor in self.get_neighbors(self.state):
                cost = self.get_cost(neighbor)
                if best_neighbor_cost is None or cost > best_neighbor
                    best_neighbor_cost = cost
                    best_neighbors = [neighbor]
                elif best_neighbor_cost == cost:
                    best_neighbors.append(neighbor)

            if best_neighbor_cost <= self.get_cost(self.state):
                # If no better neighbor is found return the same
                return best_state

            # Move to a highest-valued neighbor because above
            self.state = random.choice(best_neighbors)
            best_state = self.state
```

```
            if log:
                print(f"Iteration {count}: Cost {best_neighbor_cost}"

        return best_state

    def random_restart(self, maximum, log=False): #randomly restart r
        best_state = self.hill_climb()
        best_cost = self.get_cost(best_state)

        for i in range(maximum):
            self.state = [random.randint(0, 1) for _ in range(self.n)
            state = self.hill_climb() #ATTEMPTING HILL CLIMBING FROM
            cost = self.get_cost(state)

            if cost > best_cost:
                best_state = state
                best_cost = cost

            if log:
                print(f"Random Restart {i}: Cost {cost}")

        return best_state, best_cost

#after running the hill-climbing algorithm on some particular, random
#This algorithm never make a move that makes our situation worse. It
```

In [100]:
```
# Example usage with provided parameters
weights = [23, 31, 29, 44, 53, 38, 63, 85, 89, 82]
values = [92,57,49,68,60,43,67,84,87,72]
capacity = 165
```

In [101]:
```
solver = KnapsackSolver(weights, values, capacity)
best_solution, best_value = solver.random_restart(maximum=50, log=Tru
```

```
Random Restart 0: Cost 216
Random Restart 1: Cost 309
Random Restart 2: Cost 284
Random Restart 3: Cost 234
Random Restart 4: Cost 170
Random Restart 5: Cost 189
Random Restart 6: Cost 204
Random Restart 7: Cost 195
Random Restart 8: Cost 284
Random Restart 9: Cost 309
Random Restart 10: Cost 154
Random Restart 11: Cost 220
Random Restart 12: Cost 224
Random Restart 13: Cost 139
Random Restart 14: Cost 239
Random Restart 15: Cost 234
Random Restart 16: Cost 209
Random Restart 17: Cost 309
Random Restart 18: Cost 195
Random Restart 19: Cost 184
Random Restart 20: Cost 216
Random Restart 21: Cost 309
Random Restart 22: Cost 232
Random Restart 23: Cost 181
Random Restart 24: Cost 224
Random Restart 25: Cost 176
Random Restart 26: Cost 195
Random Restart 27: Cost 181
Random Restart 28: Cost 184
Random Restart 29: Cost 236
Random Restart 30: Cost 276
Random Restart 31: Cost 195
Random Restart 32: Cost 276
Random Restart 33: Cost 151
Random Restart 34: Cost 259
Random Restart 35: Cost 139
Random Restart 36: Cost 209
Random Restart 37: Cost 309
Random Restart 38: Cost 216
Random Restart 39: Cost 309
Random Restart 40: Cost 284
Random Restart 41: Cost 224
Random Restart 42: Cost 309
Random Restart 43: Cost 183
Random Restart 44: Cost 216
Random Restart 45: Cost 276
Random Restart 46: Cost 284
Random Restart 47: Cost 181
Random Restart 48: Cost 197
Random Restart 49: Cost 244
```

In [102]:
```
print("Best Solution:", best_solution)
print("Best Value:", best_value)
```

```
Best Solution: [1, 1, 1, 1, 0, 1, 0, 0, 0, 0]
Best Value: 309
```

Conclusions:

1. The first code is using mlrose library which we evaluate the fitness of a state vector.
2. The second part always starts with the same fixed initial state, which might lead to a local max/min depending on the problem. With its fixed initial state, may be more suitable for situations where you have a good initial estimate of the solution, and you want to fine-tune it using hill climbing without introducing randomness in the process.
3. In contrast, the third code, introduces randomness in the initial state, which can help explore different parts of the solution space. With its random_restart method, has the potential to find better solutions by attempting hill climbing from various starting points. This can be advantageous when dealing with a highly non-convex problem like the knapsack problem. As we found out after calculating a better state here.

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 [ ]: