Table of Contents

Using Local Search Algorithms to Solve the Knapsack Problem

Introduction

Local search algorithms are optimization algorithms which we use to find an optimal solution to a problem starting from a certain state. These algorithms come in handy when we don't care about the path that lead to a solution but only about the exact solution.

In this lab we have included the three well-known local search algorithms named:

1. Hill-Climbing algorithm
2. Simulated Annealing algorithm
3. Genetic algorithm

In this lab report we demonstrate the beauty of local search algorithms in order to solve a popular problem called the knapsack problem. This problem is a problem in combinatorial optimization.

Problem Statement

We are given items with their weights and values attached. The problem asks given a set of items, each with a weight and a value, determine the number of each item to be included in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. Each element is allowed to occur at most three times.

In order to solve this problem we have used the above three algorithms. In all the algorithms we will be given the same type of data, and in all of them we will start from an initial state and optimize it until we reach the most optimal state. The way to optimize the state is different in all the algorithms.

Procedure

There are actually three steps used in all the algorithms. But each algorithm use different methods to find optimal neighbor to the current state. First the input file contains 10 elements, the second one contains 15, and the third one contains 20 elements in all these algorithms the following three steps are performed.

1. Generate the initial state
2. Find a way to find a more optimal solution to the current state
3. Benchmark the result using graphs

1. Hill-Climbing algorithm to solve the knapsack problem

This algorithm holds the current state and on each iteration move to the next neighboring state with a higher value. If there is no element with a higher value the algorithm stops execution. Hence it may be caught by a local maximum value or plateaus.

Procedure

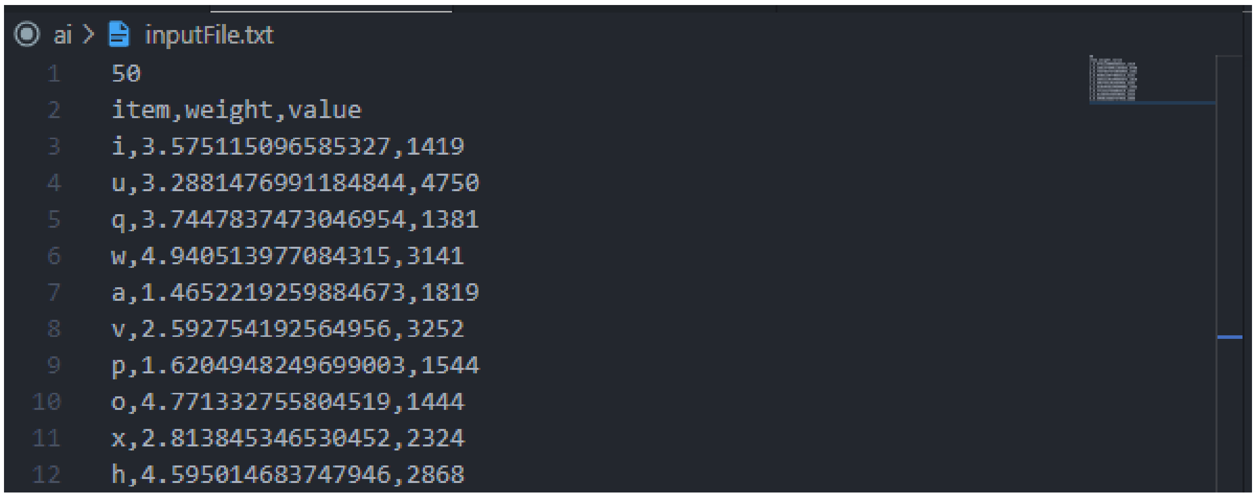
In order to solve the knapsack problem we have followed the following steps.

1. **Initial State:** the initial state is generated in such a way that there is exactly one element in the array. This can be taken as a solution with only one element have a value and all other elements have exactly zero values.
2. **Neighbor State:** the neighbors are generated in such a way that all the items with repetitions less than the maximum repetitions (one) are added to the current state. The next neighbor is the one with the maximum value.
3. **Update the current state:** Finally if the selected neighbor state has a value greater than the current state, it is definitely selected, otherwise the function is terminated since we have no state greater than the current state.

These three steps are repeated on each iteration until there is no state with a value greater than the current value.

Results

The results show that the value increase as long as we have not reached the limit weight. These are the generated file.



ai > inputFile.txt

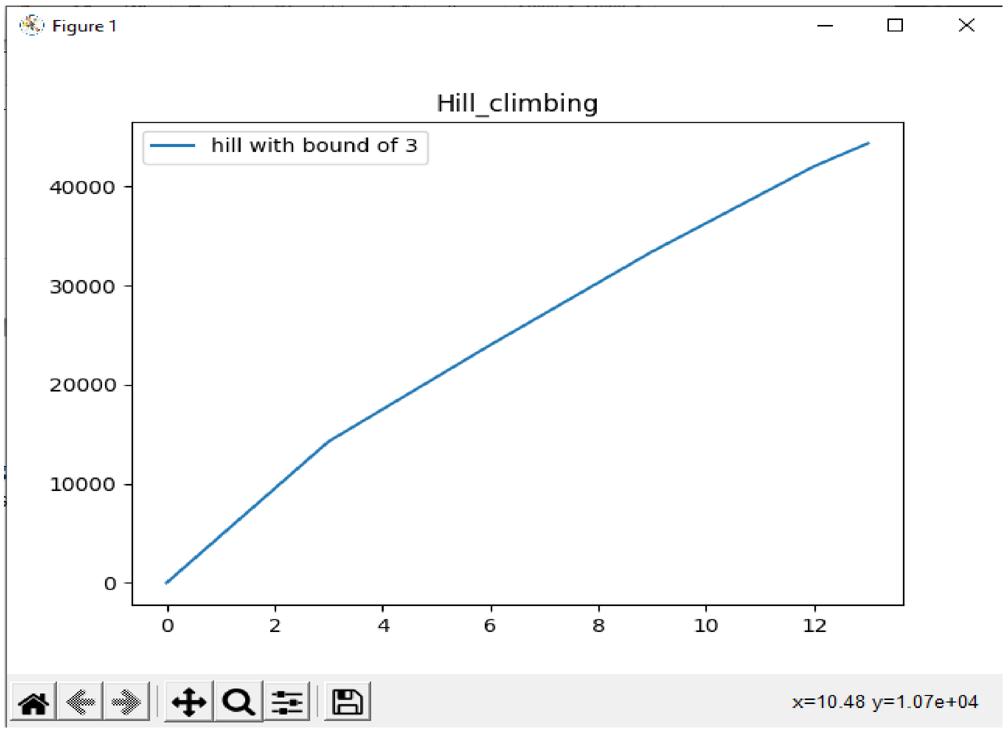
```
50
item,weight,value
i,3.575115096585327,1419
u,3.2881476991184844,4750
q,3.7447837473046954,1381
w,4.940513977084315,3141
a,1.4652219259884673,1819
v,2.592754192564956,3252
p,1.6204948249699003,1544
o,4.771332755804519,1444
x,2.813845346530452,2324
h,4.595014683747946,2868
```

The following table shows the occurrence of each element in the generated result.

Item	**occurrence**
i	0
u	3
q	0
w	3
a	0
v	3
p	0
o	0
x	1
h	3

As we can see from above the occurrence of each elements is limited by the maximum occurrence given (in this case three). The total weight generated from the above solution is 49 which is less than the limit 50. And the total value generated is 44357.

Here is the graph that shows the trace of the generated weights and values. The graph is value versus number of iterations graph.



As we can see from above the graph ascends until it reaches a state where there are no neighbors greater than it. In our case the tip of the graph is a local maximum that is not the most optimal solution but the one that is closest to the optimal one.

2. Simulated-Annealing algorithm to solve the knapsack problem

Procedure

In order to solve the knapsack with simulated annealing we have followed the following steps.

1. **Initial State:** Our initial state is a randomly generated solution or any combination of the items that have a total weight less than the limit weight.
2. **Neighbor State:** In order to generate a neighbor, we first randomly select an index from the current state and changed its occurrence to any randomly generated value between zero and maximum occurrence.
3. **Update the current state:** If the value of the generated neighbor is greater than the current state, it is accepted, otherwise the Boltzmann probability is calculated and compared to the probability of current (random number between

0 and 1), if it has higher probability it is accepted, otherwise the state stays the same.

These three steps are repeated on each iteration until the temperature becomes zero.

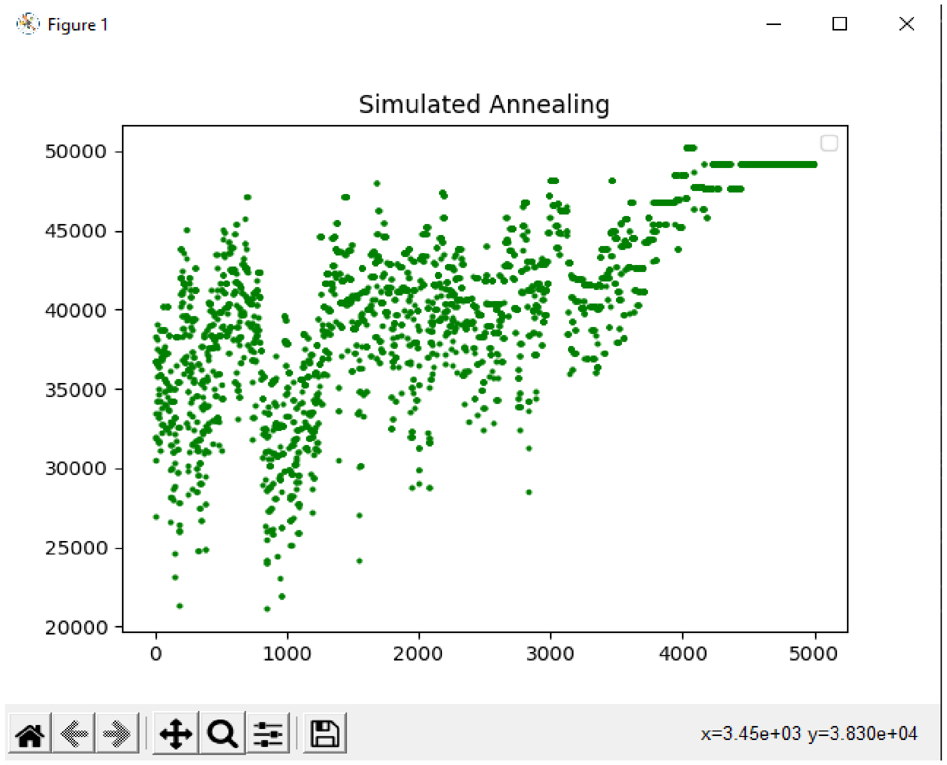
Results

The following table shows the occurrence of each element in the generated result.

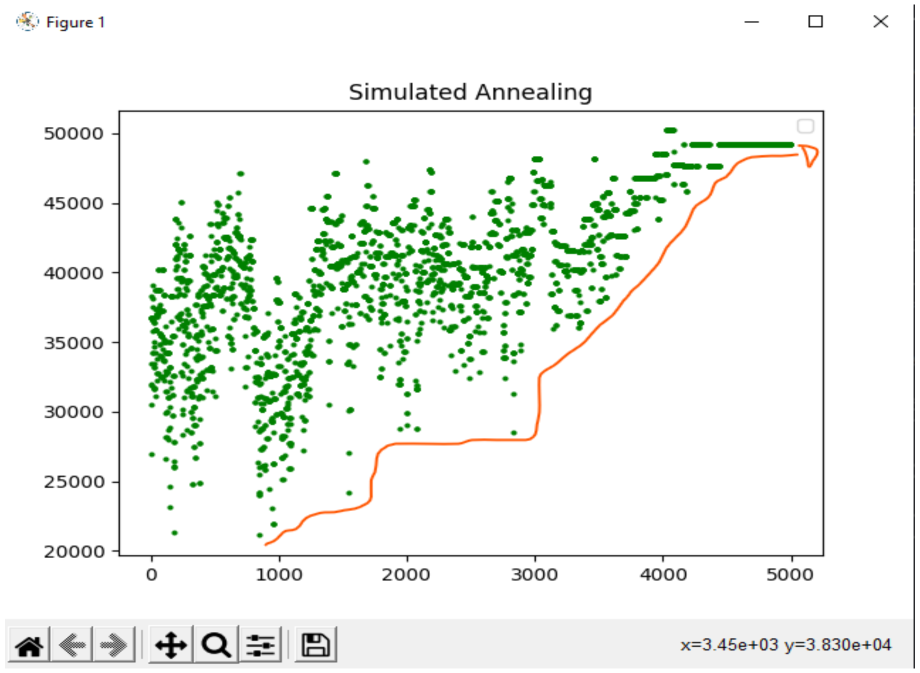
Item	occurrence
i	0
u	3
q	0
w	2
a	3
v	3
p	2
o	0
x	2
h	2

As we can see from above the occurrence elements is randomly generated and limited to three. The total weight generated from the above solution is 49.91 which is less than the limit 50. And the total value generated is 50217.

Here is the graph that shows the trace of the generated weights and values. The graph is value versus number of iterations graph.



As we can see from above the graph, the generated values are scattered in every place. But if we take a closer look, as the iteration increases forward (temperature decrease in our case) the values generated are more and closer to the higher and more optimal values. In the graph it looks like this:



3. Genetic Algorithms to solve the knapsack problem

Procedure

In order to solve the knapsack with genetic algorithms we have followed the following steps.

1. **Initial Population:** Our initial population is a collection of randomly generated single solutions that doesn’t outweigh the limit.
2. **Neighbor State:** on each iteration, two parents are randomly selected from the population, and then the parents will reproduce to create an offspring. Then the offspring is mutated to create a better version of itself. Then the child is added to the new population. The above step is repeated until the length of the population equal the current population.
3. **Update the current state:** after the new population is generated, the current population is replaced by the new population. The we have a population of off springs.

Result

The following table shows the occurrence of each element in the generated result.

Item	**occurrence**
i	0
u	3
q	0
w	2
a	3
v	3
p	2
o	0
x	2
h	2

As we can see from above the occurrence elements is randomly generated and limited to three. The total weight generated from the above solution is 49.91 which is less than the limit 50. And the total value generated is 46624.

Here is the graph that shows the trace of the generated weights and values. The graph is value versus number of iterations graph.

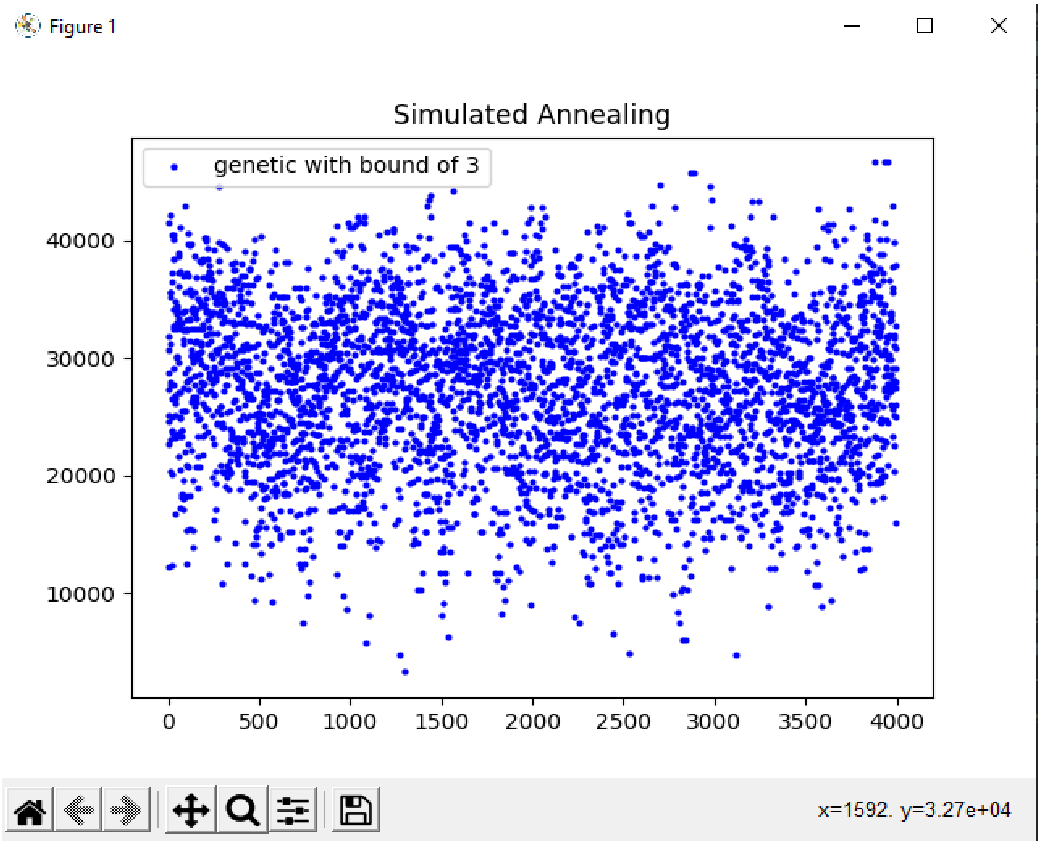


Figure 1
Simulated Annealing
genetic with bound of 3
x=1592. y=3.27e+04