Natural Methods (MetaHeuristics) Assignment

Artificial Intelligence – Spring 2021

02/18/2020

By Thomas Trankle

Summary

This report defines how to use two popular metaheuristic algorithms, namely Simulated Annealing (SA) and Genetic Algorithm (GA) to solve the commonly known bounded Knapsack problem. The introduction gives a brief introduction to the problem and its constraints, as well as my general layout for the description of each algorithm. The body of the report procedurally steps through possible implementations of each algorithm. Both algorithms perform well in unknown solution spaces but have many subjective parameters or internally defined functions which could affect the overall efficiency of both procedures.

Introduction

The body of this reports depicts a code-like walkthrough of how one could solve the bounded Knapsack problem using either Simulated Annealing Algorithm or a Genetic Algorithm. The problem is defined as follows:

There are *n* items we can pick from to fill our knapsack. However for each of these $\{ x_1 , x_2 , \ldots , x_n \}$ items, there is an associated weight, w_i, and value, v_i. Additionally, the knapsack is constrained by an upper limit on the total weight, where

$$W_{tot} = \sum_{i=1}^{n} w_i x_i \leq W_{max}$$

Furthermore, one can select multiple of each item, but no item can be selected more than *c* times, that is

$$x_i \in \{0,1, \ldots , c\}$$

The goal of the problem is to maximize the value of the knapsack based on the items we place inside of it, subject to the two constraints above.

$$\max \sum_{i=1}^{n} v_i x_i$$

For both algorithms, and any metaheuristic algorithm, one of the most important steps can be defining the initial solution, otherwise known as the starting point. For both algorithms, I assumed one would want to have a higher chance of picking a specific item that had a higher associated value. Therefore, to avoid a uniform distribution for the probability of picking any item, I partitioned the distribution into segments mimicking the value of an item. Now, when a random number is generated between 0 and 1, if the item has a higher value associated with it, there is a higher chance the random number will fall into that bin, thus favoring said valued items. This new distribution is named XDistList in both algorithms. However, the only difference between SA and GA is that we must run this initializing procedure multiple times to fill the population.

In general, the procedural layout to both algorithms is similar. Firstly, I created two global functions one can use to get the weight and value of the Knapsack at any given time. These two functions are global functions, meaning they can be used on both algorithms, because they are regular checks that need to be done throughout both procedures. Next, we generate an initial solution, or Knapsack, for the problem.

Then we automatically set the best solution to the initial solution or one of the initial solutions (for GA) since no other solutions have been generated but these initial solutions satisfy the constraints and have a higher value than zero. One important note, is that to keep the genetic algorithm as a binary-string, my implementation of the algorithm will increase the number of index item positions by a factor of c. Meaning that if c =5 for example and we have 3 items, then the length of the Knapsack solution will be 15, where the first 5 index locations correspond to the same item, and the next 5 correspond to the next item and so on and so forth. Then, the substance of each respective algorithms is run while utilizing the appropriate functions until the stopping criteria is met. For SA, the stopping criteria is defined by the temperature T, and whether it is below a specified value and for GA the stopping criteria is simply a set number of iterations. At the end, both functions will have stored the best solution is has seen that satisfies the constraints.

Global Functions:

Fucnction called getValue(list)

Pass In: a list containing the coefficients for the number of a particular item in the knapsack

This function will calculate the dot product between the input list and the global variable *V*

Pass out: The scalar value of the dot product between the input and global variable V

End function

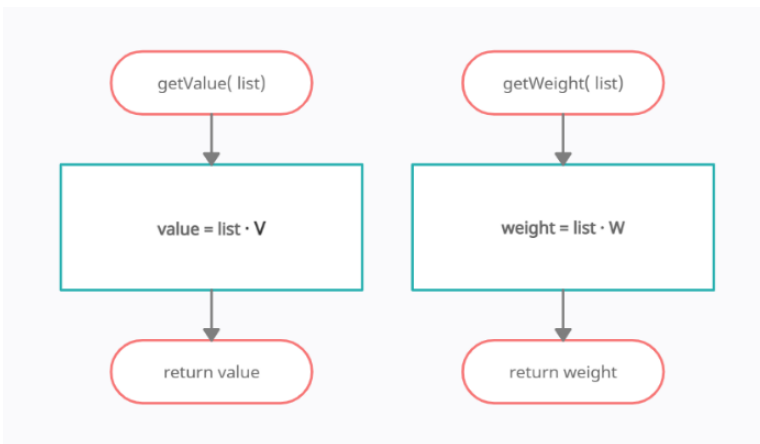
Function called getWeight(list)

Pass In: a list containing the coefficients for the number of a particular item in the knapsack

This function will calculate the dot product between the input list and the global variable *W*

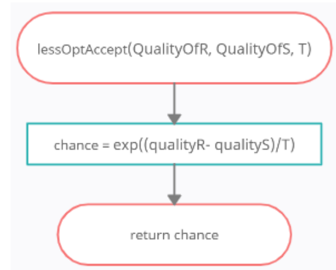
Pass out: The scalar value of the dot product between the input and global variable W

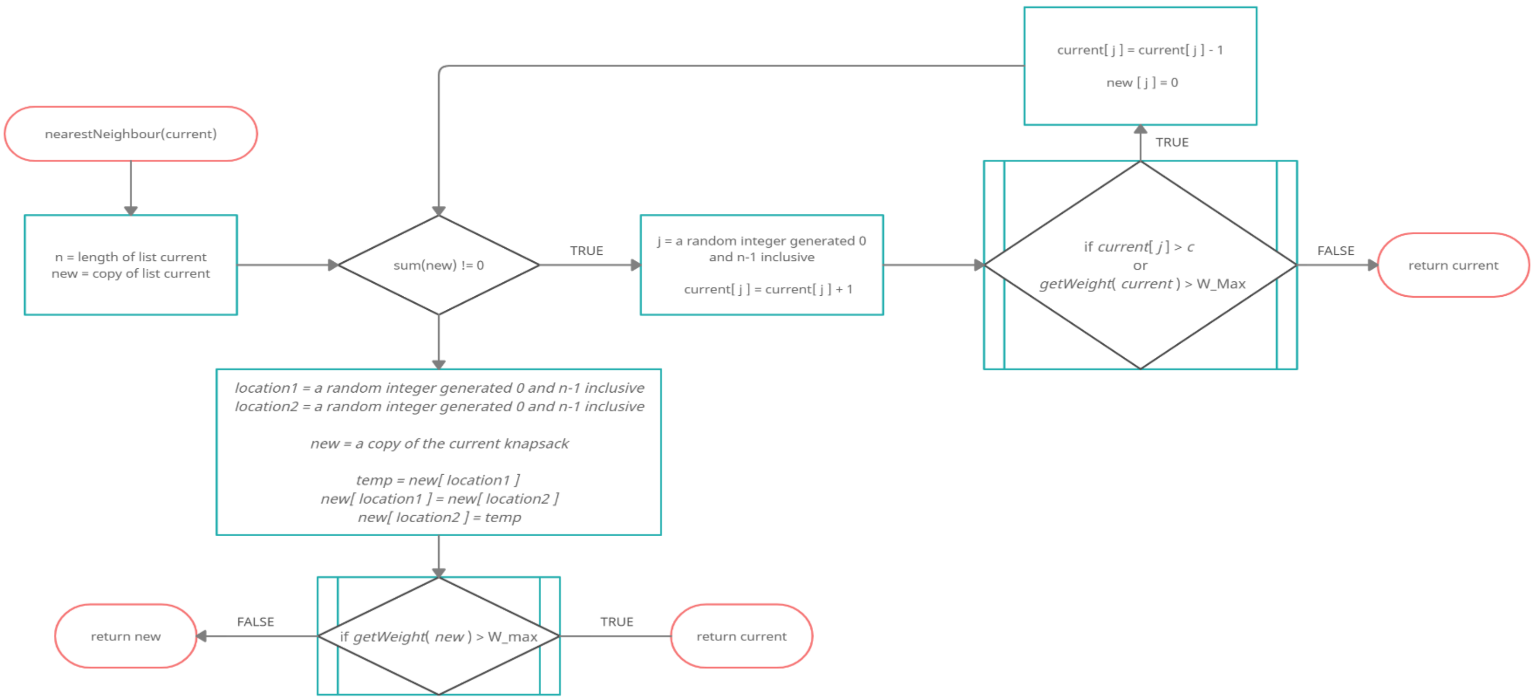
End function



Simulated Annealing

Function called lessOptAccept(qualityR, qualityS, T)
Pass in: Quality or value of one knapsack solution R
Quality or value of another knapsack solution S
Current temperature value T
Compute and Pass out exp((qualityR- qualityS)/T)
Endfunction



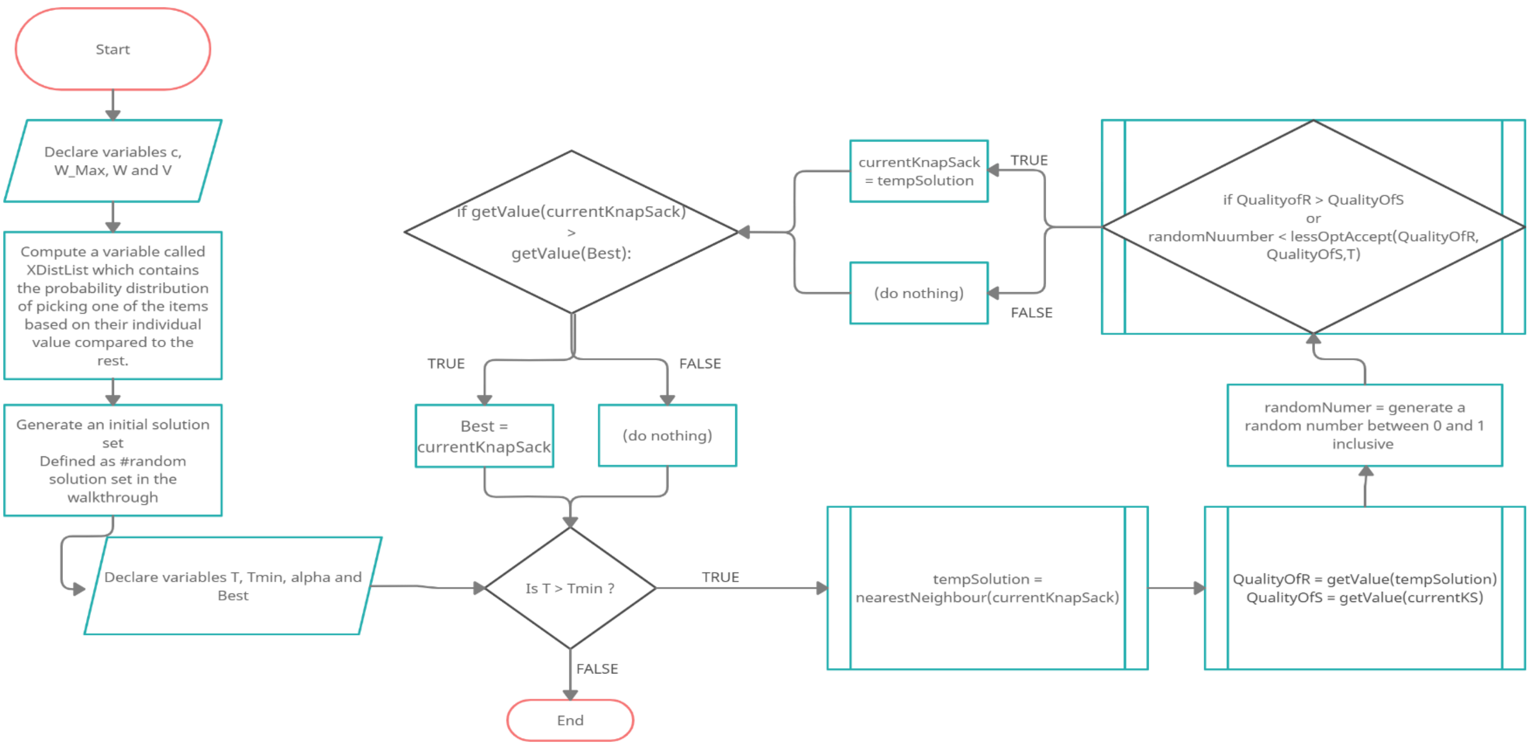


Function called nearestNeighbour(current)
Pass In: The *current* knapsack coefficient solution as a list

Declare integer *n* = length of the input list *current*
Declare list *new* = a copy of the *current* knapsack list solution to keep track of change attempts
while sum(values in list *new*) != (does not equal) 0:
Declare *j* = a random integer generated 0 and *n*-1 inclusive
current[*j*] = *current*[*j*] + 1
if *current*[*j*] > *c* or *getWeight*(*current*) > W_Max:
current[*j*] = *current*[*j*] - 1
new [*j*] = 0
restart the while loop from the beginning (continue command in python)
Endif
return/output the list *current* from the function
end while

if we can't add any item to the knapsack because of constraints, then we will try swap two item
coefficients (number of that particular item). Otherwise, return what was inputted and try again
next iteration

Declare integer *location1* = a random integer generated 0 and *n*-1 inclusive
Declare integer *location2* = a random integer generated 0 and *n*-1 inclusive
Redeclare list *new* = a copy of the *current* knapsack list solution to see if swap is possible
Declare integer *temp* = *new*[*location1*]
new[*location1*] = *new*[*location2*]
new[*location2*] = *temp*
if *getWeight*(*new*) > W_max:
    return/output the list *current* from the function
else:
    return/output the list *new* from the function
Endif
End function



MAIN
Declare integer *c* # c is the maximum non-negative integer value for each # respective item
Declare integer *W_max* # upper total weight limit of the knapsack
Declare a list *W* # W is a list containing the respective weights for each # item_i
Declare a list *V* # V is a list containing the respective value for each # item_i
Declare *currentKnapSack* = [0] * length of list *W* # empty place holder to initialize starting solution

Generate random starting solution that is within the feasible region.
Give preference to available items that have a higher associated value
Declare list XdistList = [V[i]/sum(V) for i in range(len(V))]
for i = 0 to len(XdistList) - 1:
    if i == 0:
        start the loop again with the next i value (continue)

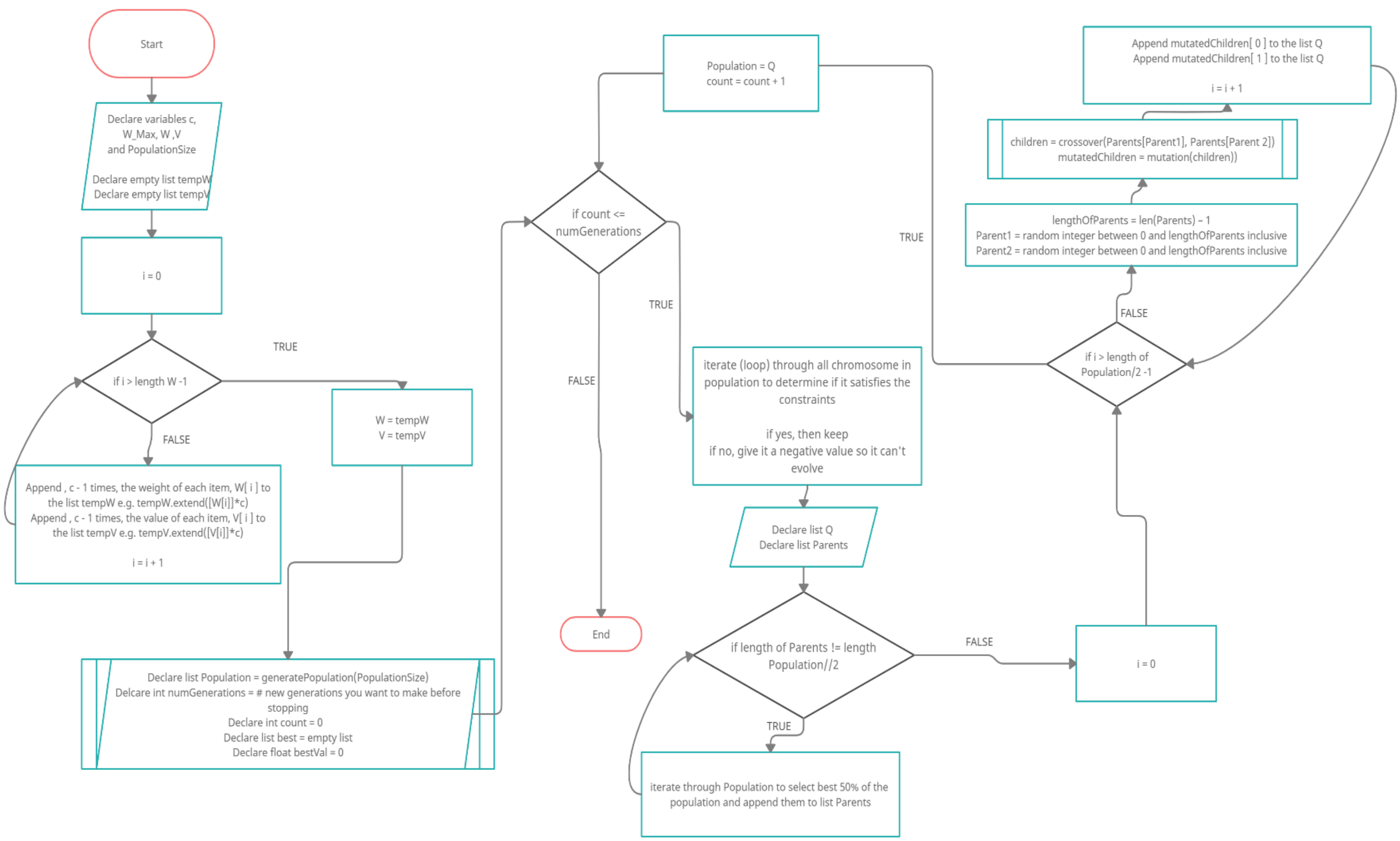
```
        else:
                XdistList[ i ] += XdistList[ i-1 ]
        Endif
End loop
#random solution set
for i = 0  to 100:
        random_num = random()
        for j = 0 to  len(XdistList) - 1:
                Declare float randomNum = generate a random number between 0 and 1 inclusive
                if randomNum <= XdistList[ j ]:
                        Declare list temp = [] + currentKnapSack
                        temp[ j ] + =  temp[ j ] + 1
                        if (temp[j]) > c or getWeight( temp )  > W_max:
                                jump out of current loop (break)
                        else:
                                currentKnapSack[ j ] = currentKnapSack[ j] + 1
                                jump out of current loop (break)
                        Endif
                else:
                        start the loop again with the next j value (continue)
                Endif
Declare float T = 1                     # starting “temperature” value
Declare float Tmin = 0.000001           # stooping “temperature” point
Declare float alpha = 0.9               # the change factor for T after every iteration
Declare list Best = [] + currentKnapSack  # a copy of the best KnapSack solution seen so far
        while T > Tmin:
        T = T * alpha
        Declare list tempSolution = nearestNeighbour(currentKnapSack)
        Declare integer QualityOfR = getValue(tempSolution)
        Declare integer QualityOfS = getValue(currentKS)
        Declare float randomNumber = generate a random number between 0 and 1 inclusive
        if QualityOfR > QualityOfS or randomNumber < lessOptAccept(QualityOfR, QualityOfS, T):
                currentKnapSack = tempSolution
        else:
                pass (do nothing)
        Endif
        if getValue(currentKnapSack) > getValue(Best):
                Best = [] + currentKnapSack
        else:
                Pass (do nothing)
        Endif
End while
print out the Best solution
```

Genetic Algorithm

```
Function called generatePopulation(PopulationSize):
        Pass In: An integer the defines the size of the Population

        Declare list Population = an empty list as a place holder to append values to
        Declare list XDistList = empty list that will hold the value associated distribution
        for i = 0 to length(V) -1:
                Append V [ i ] / TotalSum of V to the list XDistList
        End loop
        for i = 0 to the length of XDistList - 1:
                if i == 0:
                        start the loop again with the next i value (continue)
                else:
                        XDistList[i] =  XDistList[ i ] + XDistList[i-1]
                Endif
        End loop
        for each chromosome in PopulationSize:
                currentKnapSack = create a list of 0's with a length of W
                for i = 0 to 100:
                        randomNumber = generate a float between 0 and 1 inclusive
                        for j = 0 to length of XDistList - 1:
                                if randomNumber <= XDistList[ j ]:
                                        Declare list temp = copy of currentKnapSack list
                                        temp[ j ] = temp[ j ] + 1
                                        if (temp[ j ]) > c or getWeight(temp) > W_max:
                                                jump out of current loop (break)
                                        else:
                                                currentKnapSack[ j ] = currentKnapSack[ j ] + 1
                                                jump out of current loop (break)
                                        Endif
                                Endif
                        End loop
                Append currentKnapSack to the Population list
                End loop
   return Population

Function called crossover(PopulationOfA, PopulationOfB):
      '''This function implements the one point crossover algoritm'''
        Pass In: Two list of list or arrays containing Population A and a Population B
        Declare int n = length of PopulationOfA
        Declare int k = random integer chosen uniformly from 0 to the length of n - 1 inclusive
        if k != 0:
                temp = PopulationOfA [ k:n ]                            # k:n is a slice from index k to
                PopulationOfA [ k:n ] = PopulationOfA [ k:n ]           # index n - 1
                PopulationOfB[ k:n ] = temp
        Endif
        return list containing [PopulationOfA, PopulationOfB]

Function called mutation(children):
```

Pass In: a list of lists containing two Populations (A and B)

Declare float probability = 1 / length of the first list in children   i.e. len(children[0])
for i = to the length of children - 1:
    for j = 0 to the length of children[i] - 1:
        if probability >= random float between 0 and 1 inclusive:
            children[i][j] = swap the binary integer of children[i][j]
                e.g. int(not children[i][j])
        Endif
return children



MAIN

Declare integer *c*   # c is the maximum non-negative integer value for each
respective item

Declare integer *W_max*   # upper total weight limit of the knapsack
Declare a list *W*   # W is a list containing the respective weights for
each   # item_i
Declare a list *V*   # V is a list containing the respective value for each
item_i
Declare int PopulationSize = Define the size of your population

Declare empty list tempW = []
Declare empty list tempV = []

```
for i = 0 to length of W - 1:
        Append , c - 1 times, the weight of each item, W[ i ] to the list tempW e.g.  tempW.extend([W[i]]*c)
        Append , c - 1 times, the value of each item, V[ i ] to the list tempV e.g.  tempV.extend([V[i]]*c)
End loop
W = tempW
V = tempV

Declare list Population = generatePopulation(PopulationSize)
Delcare int numGenerations = the number of new generations you want to make before stopping
Declare int count = 0               # keep track of which generation we are currently on
Declare list best = empty list that will later hold the best chromosome list we've seen thus far
Declare float bestVal = 0           # the score/fitness of the best solution thus far
while count <= numGenerations:
        FitnessScores = []
        for each chromosome list in Population:
                if getWeight(W) <= W_max:
                        fitnessOfChromosome = getValue(chromosome)
                        Append fitnessOfChromosome to the list FitnessScores
                        if bestVal==0 or fitnessOfChromosome > getValue(best):
                                best = a copy of the current list chromosome
                                bestVal = getValue(best)
                        else:
                                Append a large negative number such as -999 to FitnessScores
                Endif
        End loop

        Declare list Q = empty list that we will fill with the next generation
        Declare list Parents = empty list we store the best parents of the current generation in
        while length of Parents != length Population//2:
                for i = 0 to length of Population - 1:
                        if FitnessScores[ i ] == max(FitnessScores):
                                Append Population[ i ] to the list Parent
                                FitnessScores[ i ] = 0
                                jump out of current loop (break)
                End loop

        for i = 0 to length of (Population)//2) – 1 :
                Declare int lengthOfParents = len(Parents) – 1
                Declare int Parent1 = random integer between 0 and lengthOfParents inclusive
                Declare int Parent2 = random integer between 0 and lengthOfParents inclusive
                children = crossover(Parents[Parent1], Parents[Parent 2])
                mutatedChildren = mutation(children)
                Append mutatedChildren[ 0 ] to the list Q
                Append mutatedChildren[ 1 ] to the list Q
        End loop
        Population = Q
        count = count + 1
print out the best solution
```

Conclusion

Both algorithms have the potential to solve the bounded Knapsack problem. However, because this is a metaheuristics problem and the optimal solution is not known and nonlinear. So, neither algorithm can guarantee to find the optimal solution unless the they have enough time. For Simulated Annealing we are guaranteed to find the optimal solution if given enough time, or if the decays function for how we decrease T is small. With Genetic Algorithms, the problem at hand requires enough evolutions/generation to find the optimal/ideal solution. Meaning we might not correctly define the correct number of generations to achieve this ideal solution or the number of evolutions to find the ideal solution is simply too long and would take too much time. Like Simulated Annealing, it will return a sub-optimal solution if the above-mentioned parameters are not correct. Notably, with both algorithms we could add a conditional statement in the main body of the solutions to only stop if the current best solution, which satisfies the constraints, surpasses a minimum value.

Recommendations and Comments

There are limitations or alternatives to both algorithms. The nearestNeighbour function makes use off add items to the knapsack and if that fails, then it will at least try to swap the coefficients of two randomly selected items. However, there are many other ways to generate a nearest neighbour, such as shuffling the entire knapsacks coefficients by moving all coefficients in a random direction. Additionally, we specified alpha and Tmin values, but these are completely subjective and user dependent.

Similarly, the Genetic Algorithm makes use of one-point crossover and random shuffle vector for mutation, but this is also not necessarily the best method of either crossover or mutation. Therefore, the true determination of the best parameters and internal methods for the algorithms remains ambiguous and are probably dependent on the specific knapsack conditions, that is the number, weight and values of items – as opposed to the generalized case for which the above-stated algorithms will sufficiently solve but might not achieve the optimal solution.