# Homework 5

Heuristic Search Methods

Daniel Carpenter

April 2022

## Contents

***Please see the final page for the summary output.***

## Global Variables

Input variables like the *random seed*, *values and weights* data for knapsack, and the *maximum allowable weight*
Please assume these are referenced by following code chunks

```python
# Import python libraries
import copy
import math
from random import Random
import numpy as np

# to setup a random number generator, we will specify a "seed" value
seed = 5113
myPRNG = Random(seed)

# to setup a random number generator, we will specify a "seed" value
# need this for the random number generation -- do not change
seed = 51132021
myPRNG = Random(seed)

# number of elements in a solution
n = 150

# create an "instance" for the knapsack problem
value = []
for i in range(0, n):
    # value.append(round(myPRNG.expovariate(1/500)+1,1))
    value.append(round(myPRNG.triangular(150, 2000, 500), 1))

weights = []
for i in range(0, n):
    weights.append(round(myPRNG.triangular(8, 300, 95), 1))

# define max weight for the knapsack
maxWeight = 2500
```

## 1 *Question 1:* Simulated Annealing

- In order to determine the initial temperature, the function user is able to change the initial solution to whatever they would like. Unlike the mentioned methods to calculate the initial temperature, this algorithm differs.

### 1.1 Evalutation Function

```
# ===============================================================================
# EVALUATE FUNCTION - evaluate a solution x
# ===============================================================================

# monitor the number of solutions evaluated
solutionsChecked = 0

# function to evaluate a solution x
def evaluate(x, r=myPRNG.randint(0,n-1)):

    itemInclusionList = np.array(x)
    valueOfItems      = np.array(value)
    weightOfItems     = np.array(weights)

    totalValue  = np.dot(itemInclusionList, valueOfItems)   # compute the value of the knapsack selecti
    totalWeight = np.dot(itemInclusionList, weightOfItems)  # compute the weight value of the knapsack 

    # Handling infeasibility --------------------------------------------------

    # If the total weight exceeds the max allowable weight, then
    if totalWeight > maxWeight:

        # Randomly remove ann item. If not feasible, then try evaluating again until feasible
        randIdx = myPRNG.randint(0,n-1) # generate random item index to remove
        x[r] = 0                        # Don't include the index r from the knapsack
        evaluate(x, r=randIdx)          # Try again on the next to last element

    else:
        # Finish the process if the total weight is satisfied
        # (returns a list of both total value and total weight)
        return [totalValue, totalWeight]

    # returns a list of both total value and total weight
    return [totalValue, totalWeight]

# Indices of the list returned from the evalution function
VALUE_IDX  = 0 # Value of the bag
WEIGHT_IDX = 1 # Weight of the bag
```

### 1.2 Neighborhood Function

```
# =============================================================================
# NEIGHBORHOOD FUNCTION - simple function to create a neighborhood
# =============================================================================

# 1-flip neighborhood of solution x
def neighborhood(x):

    nbrhood = []

    # Set up n number of neighbors with list of lists
    for i in range(0, n):
        nbrhood.append(x[:])

        # Flip the neighbor from 0 to 1 or 1 to 0
        if nbrhood[i][i] == 1:
            nbrhood[i][i] = 0
        else:
            nbrhood[i][i] = 1

    return nbrhood
```

## 1.3 Initial Solution Function

```
# =============================================================================
# INITIAL SOLUTION FUNCTION - create the initial solution
# =============================================================================

# create a feasible initial solution
def initial_solution():

    x = [] # empty list for x to hold binary values indicating if item i is in knapsack

    # Create a initial solution for knapsack (Could be infeasible), by
    # randomly create a list of binary values from 0 to n. 1 if item is in the knapsack
    for item in range(0, n):
        x.append(myPRNG.randint(0,1))

    totalWeight = np.dot(np.array(x), np.array(weights)) # Sumproduct of weights and is included


    # While the bag is infeasible, randomly remove items from the bag.
    # Stop once a feasible solution is found.
    knapsackSatisfiesWeight = totalWeight <= maxWeight # True if the knapsack is a feasible solution, e

    while not knapsackSatisfiesWeight:

        randIdx = myPRNG.randint(0,n-1) # Generate random index of item in knapsack and remove item
        x[randIdx] = 0

        # If the knapsack is feasible, then stop the loop and go with the solution
        totalWeight = np.dot(np.array(x), np.array(weights)) # Recalc. Sumproduct of weights and is inc
        if (totalWeight <= maxWeight):
            knapsackSatisfiesWeight = True

    return x
```

### 1.4 Stopping Criterion Function

- Stopping criterion method used is to stop at a given number of total iterations.

```
# Stopping criterion
def stopTheProcedure(k, TOTAL_ITERS):
    return k == TOTAL_ITERS # stop if the iteration are greater than the max iters to perform
```

### 1.5 Function that Holds the Caunchy Cooling Schedule

- This function takes in the initial temperature and a value of k to implement the Caunchy cooling schedule, which formally is defined as: $t_k = \frac{T_0}{1+k}$

```
# The cooling Schedule for the Caunchy method
def caunchyCoolSchedule(INITIAL_TEMP , k):
    # Calculuate and return the schedule output (t_k)
    t_k = INITIAL_TEMP / (1 + k)
    return(t_k)
```

### 1.6 Function that Holds the Boltzmann Cooling Schedule

- This function takes in the initial temperature and a value of k to implement the Boltzmann cooling schedule, which formally is defined as: $t_k = \frac{T_0}{\log(1+k)}$

```
# The cooling Schedule for the Boltzmann method
def boltzmannCoolSchedule(INITIAL_TEMP , k):
    # Calculuate and return the schedule output (t_k)
    t_k = INITIAL_TEMP / np.log(1 + k)
    return(t_k)
```

### 1.7 Function that Retreives a Cooling Schedule

- The `simAnnealKnapsack()` is passed a cooling schedule (either Caunchy or Boltzemann) and then this function uses it for the simulated annealing cooling schedule.

```
# General function that takes in a cooling schedule and calculates t[k]
# meant for ease of changing the cooling schedule
def coolingSchedule(scheduleFunction, INITIAL_TEMP , k):
    return(scheduleFunction(INITIAL_TEMP , k))
```

## 1.8 Simulated Annealing Alg. Function

- Given a probability p, the algorithm can accept a non-improving move.
- The probability fomula is the following: $p = e^{\frac{-(f(s_1) - f(s_2))}{T}}$
  - Where $f(s_1)$ is the random solution, and $f(s_2)$ is the current solution since maximization.
  - The function uses a passed colling schedule *(either boltzmann or caunchy)*

```
# ============================================================================
# SIMULATED ANNEALING ALGORITHM
# ============================================================================

def simAnnealKnapsack(TOTAL_ITERS, INITIAL_TEMP, ACCEPTANCE_THRESHOLD, COOLING_METHOD):


    ## GET INITIAL SOLUTION ------------------------------------------------------

    solutionsChecked = 0 # Keep track of the number of solutions checked

    x_curr = initial_solution()  # x_curr will hold the current solution
    f_curr = evaluate(x_curr) # f_curr holds the evaluation of the current soluton

    ## BEGIN LOCAL SEARCH LOGIC -------------------------------------------------

    k_iter = 0 # Track the total iterations

    # Do not stop the procedure until the stoppping criterion is met
    while not stopTheProcedure(k_iter, TOTAL_ITERS):

        # Create a list of all neighbors in the neighborhood of x_curr
        Neighborhood = neighborhood(x_curr)

        m_iter = 0 # Track the iterations at each temperature
        numSolutionsAccepted = 0 # keep track of number of solutions accepted

        while numSolutionsAccepted < ACCEPTANCE_THRESHOLD: # must search m times at each temp
            solutionsChecked += 1 # Notate another solution checked

            # Randomly select solution from neighbor of current solution
            x_randSolution = Neighborhood[myPRNG.randint(0,n-1)]
            f_randSolution = evaluate(x_randSolution) # Evalute the solution


            # CHECK TO SEE IF RANDOM SOLUTION IS BETTER THAN CURRENT --------------

            # If the random solution knapsack value is better and is feasible...
            if (f_randSolution[VALUE_IDX] > f_curr[VALUE_IDX] and f_randSolution[WEIGHT_IDX] <= maxWeig
                x_curr = x_randSolution[:] # Store it as the current solution
                f_curr = f_randSolution[:]
                numSolutionsAccepted += 1  # Notate that we accepted a solution at this temp

            # EVEN THOUGH RANDOM SOLUTION WAS WORSE, ACCEPT IT WITH RANDOM PROB ---
            else:
                # difference between the random and current solution
                solutionsDelta = f_randSolution[VALUE_IDX] - f_curr[VALUE_IDX]

                # Using the given cooling schedule (boltzmann), update the temperature given iteration
                t_k = coolingSchedule(COOLING_METHOD, INITIAL_TEMP , k_iter)
                runifProb = myPRNG.uniform(0,1) # With a uniform probability (from 0 to 1)...

                # Decide whether to choose the worse solution with random uniform prob
                if (runifProb <= np.exp(1)**(-solutionsDelta / t_k)
                    and f_randSolution[WEIGHT_IDX] <= maxWeight):  # and is feasible
                    x_curr = x_randSolution[:] # Store it as the current solution
                    f_curr = f_randSolution[:]

            m_iter += 1 # Increment the iterations at a given temperature
        k_iter += 1 # Increment the total iterations


    # Output of the solution ------------------------------------------------------
    valueOfBestBag     = f_curr[VALUE_IDX]
    weightOfBestBag    = f_curr[WEIGHT_IDX]
    numItemsSelected   = np.sum(x_curr)
    selectedItemsInBag = x_curr[:]
    if COOLING_METHOD == cauchyCoolSchedule:
        METHOD_CHOSEN = 'Caunchy'
    else: METHOD_CHOSEN = 'Boltzmann'

    # Output a list for the summary output
    solution = [INITIAL_TEMP, METHOD_CHOSEN, ACCEPTANCE_THRESHOLD, k_iter,
                solutionsChecked, numItemsSelected, weightOfBestBag, valueOfBestBag]

    print('\n\n--------- SOLUTION OVERVIEW ---------\n\n',
          'Initial Temp t[0]:',          solution[0], '\n',
          'Method t[k]:',                solution[1], '\n',
          'Acceptance Threshold M[k]: ', solution[2], '\n',
          '# Temps. Checked:',           solution[3], '\n',
          '# Iters:',                    solution[4], '\n',
          '# Items:',                    solution[5], '\n',
          'Weight of Bag:',              solution[6], '\n',
          'Value of Bag:',               solution[7], '\n',
          '\n-------------------------------------'
          )

    return (solution)
```

### 1.9 Call Function for Simulated Annealing

```
# Call the algorithm given inputs (for Caunchy Method)
simA1 = simAnnealKnapsack(TOTAL_ITERS          = 100,
                          INITIAL_TEMP         = 1000,
                          ACCEPTANCE_THRESHOLD = 10,
                          COOLING_METHOD       = caunchyCoolSchedule
                          )

# Call the algorithm given inputs (for Caunchy Method)
```

```
##
##
## --------- SOLUTION OVERVIEW ---------
##
##  Initial Temp t[0]: 1000
##  Method t[k]: Caunchy
##  Acceptance Threshold M[k]:  10
##  # Temps. Checked: 100
##  # Iters: 4764
##  # Items: 24
##  Weight of Bag: 2461.9
##  Value of Bag: 26618.399999999998
##
## ------------------------------------
```

```
simA2 = simAnnealKnapsack(TOTAL_ITERS          = 200,
                          INITIAL_TEMP         = 500,
                          ACCEPTANCE_THRESHOLD = 5,
                          COOLING_METHOD       = caunchyCoolSchedule
                          )

# Call the algorithm given inputs (for boltzmann Method)
```

```
##
##
## --------- SOLUTION OVERVIEW ---------
##
##  Initial Temp t[0]: 500
##  Method t[k]: Caunchy
##  Acceptance Threshold M[k]:  5
##  # Temps. Checked: 200
##  # Iters: 5918
##  # Items: 24
##  Weight of Bag: 2415.5
##  Value of Bag: 25292.8
##
## ------------------------------------
##
## <string>:50: RuntimeWarning: overflow encountered in double_scalars
```

```
simA3 = simAnnealKnapsack(TOTAL_ITERS          = 100,
                          INITIAL_TEMP         = 1000,
                          ACCEPTANCE_THRESHOLD = 10,
                          COOLING_METHOD       = boltzmannCoolSchedule
                          )

# Call the algorithm given inputs (for boltzmann Method)
```

```
##
##
## --------- SOLUTION OVERVIEW ---------
##
##  Initial Temp t[0]: 1000
##  Method t[k]: Boltzmann
##  Acceptance Threshold M[k]:  10
##  # Temps. Checked: 100
##  # Iters: 4877
##  # Items: 21
##  Weight of Bag: 2482.1
##  Value of Bag: 24807.0
##
## ------------------------------------
##
## <string>:3: RuntimeWarning: divide by zero encountered in double_scalars
```

```
simA4 = simAnnealKnapsack(TOTAL_ITERS          = 150,
                          INITIAL_TEMP         = 750,
                          ACCEPTANCE_THRESHOLD = 20,
                          COOLING_METHOD       = boltzmannCoolSchedule
                          )
```

```
##
##
## --------- SOLUTION OVERVIEW ---------
##
##  Initial Temp t[0]: 750
##  Method t[k]: Boltzmann
##  Acceptance Threshold M[k]:  20
##  # Temps. Checked: 150
##  # Iters: 13115
##  # Items: 27
##  Weight of Bag: 2479.3999999999996
##  Value of Bag: 25363.5
##
## ------------------------------------
```

## 2 *Question 2:* Genetic Algorithm

### 2.1 Function for Genetic Algorithm

- Since function wraps multiple sub-functions, ***please see commentary and explanation of logic for written functions within code***

```
# Genetic Algorithm Fun    | Function defaults:
# Genetic Algorithm Fun    | Function defaults:
def geneticAlgorithmKnapsack(populationSize = n,    # size of GA population
                             Generations    = 1000, # number of GA generations
                             crossOverRate  = 0.9,  # 90% chance that two parents bred at some crossove
                             mutationRate   = 0.10, # 10% chance an offspring is mutated between 1 and
                             eliteSolutions = 10    # Number prior pop to keep from gen to gen
                             ):
    # monitor the number of solutions evaluated
    solutionsChecked = 0

    # ============================================================================
    # CREATE CHROMOSOME: Continuous Valued
    # Inputs: d is dimensions of chromosome
    # Logic Overview:
    #   similar to initialSolution() logic for Hill-Climbing
    #   Given a certain probability, put an item into the knapsack.
    #   If probability not met, then do not append anything
    # ============================================================================
    def createChromosome(d, prob=0.08):
        # DEfault 8% Prob of appending item.
        # Note 8% produces around 5 to 15 infeasible bags
        # this code as-is expects chromosomes to be stored as a list, e.g., x = []
        # write code to generate chromosomes, most likely want this to be randomly generated

        x = [] # empty list for x to hold binary values indicating if item i is in knapsack
        ITEM    = 1 # Variable to indicate that an item is  added
        NO_ITEM = 0 # ""                                    not added

        # Create a initial solution for knapsack (Could be infeasible), by
        # randomly create a list of binary values from 0 to d. 1 if item is in the knapsack
        for item in range(0, d):
            if myPRNG.random() < prob:
                x.append(ITEM)
            else:
                x.append(NO_ITEM)
        return x

    # ============================================================================
    # CREATE INITIAL POPULATION:

    # Call "createChromosome" function many times, and
    # Add each to a list of chromosomes (a.k.a., the "population")
    # Note Inputs: Technically none, but pulls `n` from the preset inputs
    # Returns:
    # the return object is a reversed sorted list of tuples:
```

```
# [1] the first element of the tuple is the chromosome;
# [2] the second element is the fitness value
# for example:  popVals[0] is represents the best individual in the population
# popVals[0] for a 2D problem might be  ([-70.2, 426.1], 483.3)  -- chromosome is the list [-70.2,
# ==============================================================================
def initializePopulation():  # n is size of population; d is dimensions of chromosome

    population = []
    populationFitness = []

    for i in range(populationSize): # (from inputs popSize)
        population.append(createChromosome(n))
        populationFitness.append(evaluate(population[i]))

    tempZip = zip(population, populationFitness)
    popVals = sorted(tempZip, key=lambda tempZip: tempZip[1], reverse=True)

    return popVals

# Indces of the above created chromosome
ITEMS_IDX         = 0
FITNESS_VALUE_IDX = 1

# ==============================================================================
# CROSSOVER
#
# Logic Overview:
# with some probability (i.e., crossoverRate) perform breeding via crossover,
# i.e. two parents produce two offspring:
# --- the first part of offspring1 comes from p1,
# --- and the second part of offspring1 comes from p2
#
# --- the first part of offspring2 comes from p2,
# --- and the second part of offspring2 comes from p1
# ==============================================================================
def crossover(parent1, parent2, prob=crossOverRate):

    # Breeding occurs since probability met
    if myPRNG.random() < prob:

        # Get the crossover point
        crossOverPoint = myPRNG.randint(0, n-1) # Random point in array

        def splitParentAtCrossOverPoint(parent):

            # Split parent at crossover point and return the pieces
            return (
                parent[:crossOverPoint ], # First piece
                parent[ crossOverPoint:]  # Second piece
            )

        # Get two pieces of each parent at cross over points
        par1_piece1, par1_piece2 = splitParentAtCrossOverPoint(parent1) # Parent 1
```

```
            par2_piece1, par2_piece2 = splitParentAtCrossOverPoint(parent2) # Parent 2

            # Swap pieces from the parents and put into the offspring
            offspring1 = par1_piece1 + par2_piece2 # First piece of parent 1, second piece p2
            offspring2 = par2_piece1 + par1_piece2 # First piece of parent 2, second piece p1

        # if no breeding occurs, then offspring1 and offspring2 are copies of parent1 and parent2, resp
        else:
            offspring1 = parent1[:]
            offspring2 = parent2[:]

        return offspring1, offspring2  # two offspring are returned


    # ===============================================================================
    # COMPUTE WEIGHT OF CHROMOSOME x
    # ===============================================================================
    def calcWeight(x):

        a = np.array(x)
        c = np.array(weights)

        # compute the weight value of the knapsack selection
        totalWeight = np.dot(a, c)

        return totalWeight  # returns total weight


    # ===============================================================================
    # CALC. ITEMS SELECTED
    # function to determine how many items have been selected in a particular chromosome x
    # ===============================================================================
    def itemsSelected(x):

        a = np.array(x)
        return np.sum(a)  # returns total number of items selected

    # ===============================================================================
    # EVALUATE: function to evaluate a solution x
    #
    # Logic Overview:
    # Sums up the value normally if weight is feasible,
    # If not feasible weight, then there is a strong penalty put on the item
    # Notice that the item will be put to the bottom of the bad and likely will be
    # removed.
    # ===============================================================================
    def evaluate(x):

        a = np.array(x)
        b = np.array(value)

        totalValue = np.dot(a, b)  # compute the value of the knapsack selection
        totalWeight = calcWeight(x)
        # Penalize knapsacks that exceed the total weight (from HW 4 solution)
        if totalWeight > maxWeight:
            totalValue = 0.85*totalValue - 10*(totalWeight - maxWeight)**2
        fitness = totalValue

        return fitness  # returns the chromosome fitness


    # ==============================================================================
    # TOURNAMENT SELECTION
    # k chromosomes are selected (with repeats allowed) and the best advances to the mating pool
    # function returns the mating pool with size equal to the initial population
    # ==============================================================================
    def tournamentSelection(pop, k=2):

        # randomly select k chromosomes; the best joins the mating pool
        matingPool = []

        while len(matingPool) < populationSize:

            ids = [myPRNG.randint(0, populationSize-1) for i in range(k)]
            competingIndividuals = [pop[i][FITNESS_VALUE_IDX] for i in ids]
            bestID = ids[competingIndividuals.index(max(competingIndividuals))]
            matingPool.append(pop[bestID][ITEMS_IDX])

        return matingPool


    # ==============================================================================
    # ROULETTE WHEEL
    #
    # Logic Overview:
    # 1. Spin wheel: generate random number between 0 and the sum of all pop.'s fitness
    # 2. Wheel spinning: From top fitness level to lowest, add up the fitness
    #    levels until we reach the random number generated.
    # 3. Spinning stops: Once we get to the random number, we select the chromosome.
    # 4. Document the winning chromosome by adding to the mating pool
    # 5. Repeat until the mating pool is of equal size to the initial population
    # ==============================================================================
    def rouletteWheel(pop):

        matingPool = []  # list of randomly selected parents to mate

        # Spin roulette wheel k times and add k number of random parents to pool
        while len(matingPool) < populationSize:

            # Implentation from: https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_p

            ## Calculate sum of a all fitnesses in pop.
            sumOfFitnessInPop = 0
            for chromosome in range(len(pop)):
                sumOfFitnessInPop += pop[chromosome][FITNESS_VALUE_IDX]
            ## Spin the wheel ---------------------------------------------------------

            ### Generate a random number between 0 and sumOfFitnessInPop
            ### Landed random number will contain the chosen chromosome
            endedSpinValue = myPRNG.uniform(0, sumOfFitnessInPop)


            partialSumOfFitnessVal = 0 # to hold the current running sum of fitness for chromosomes
            chromosome = -1 # chromosome index (will start at 0)
            bestID = 0 # to hold the random selected chromosome idx

            # Get running sum of chromosome fitness values until reached the randomly chosen chromosome
            while partialSumOfFitnessVal < endedSpinValue:
                chromosome += 1 # Increment chromosome
                partialSumOfFitnessVal += pop[chromosome][FITNESS_VALUE_IDX]

                # Spin finished - Get the best chromosome ID
                if partialSumOfFitnessVal > endedSpinValue:
                    bestID = chromosome

            ## Now add the chosen persone to the mating pool
            matingPool.append(pop[bestID][ITEMS_IDX])

        return matingPool # return list of k parents chosen to be in mating pool


    # ===============================================================================
    # MUTATE SOLUTIONS
    #
    # Logic Overview:
    # Given a certain probability (which is the mutationRate),
    # Remove or add a random item(s) from the original knapsack solution.
    # Could have more than 1 item removed/added (mutated), depending on the maxChangesAllowed
    # Return the mutated solution.
    # ===============================================================================
    def mutate(x, prob=mutationRate, maxChangesAllowed=2): # The probability is the mutation rate

        # If probability met then mutate
        if myPRNG.random() < prob:

            # Can change 1-max (random spread)
            numIdxsToChange = myPRNG.randint(1, maxChangesAllowed)

            # Change k number of elements based on numIdxsToChange
            for change in range(numIdxsToChange):
                mutatedElementIdx = myPRNG.randint(0, n-1)

                # Mutate the index by flipping from 0 to 1 or 1 to 0
                if x[mutatedElementIdx] == 1:
                    x[mutatedElementIdx] = 0
                else:
                    x[mutatedElementIdx] = 1
        return x


    # ==============================================================================
    # BREEDING
    # uses the "mating pool" and calls "crossover" function
    # ==============================================================================
    def breeding(matingPool):
        # the parents will be the first two individuals, then next two, then next two and so on

        children = []
        childrenFitness = []
        for i in range(0, populationSize-1, 2):
            child1, child2 = crossover(matingPool[i], matingPool[i+1])

            child1 = mutate(child1)
            child2 = mutate(child2)

            children.append(child1)
            children.append(child2)

            childrenFitness.append(evaluate(child1))
            childrenFitness.append(evaluate(child2))

        tempZip = zip(children, childrenFitness)
        popVals = sorted(tempZip, key=lambda tempZip: tempZip[1], reverse=True)

        # the return object is a sorted list of tuples:
        # the first element of the tuple is the chromosome; the second element is the fitness value
        # for example:  popVals[0] is represents the best individual in the population
        # popVals[0] for a 2D problem might be  ([-70.2, 426.1], 483.3)  -- chromosome is the list [-70

        return popVals


    # ==============================================================================
    # INSERTION
    #
    # Logic Overview:
    # Create a new population, then add the k highest knapsack solutions from the
    # past population.
    # Then, there will be n minus k remaining solutions that can be added to the
    # new population. Those slots will be the n-k highest children solutions.
    # Returns the new population of size n
    # ==============================================================================
    def insert(pop, kids, k=eliteSolutions): # k is the number of past pop to keep

        newPop = [] # The new population list

        # Calculate the parents and children to insert, based on k
        # will insert k prior population members and the n minus k top ranking kids
        parentsToInsert  = k
        childrenToInsert = len(pop)-k
        # Insert the top ranking k parents
        for chromosome in range(parentsToInsert):
            newPop.append(pop[chromosome])

        # Insert the top ranking n minus k kids
        for chromosome in range(childrenToInsert):
            newPop.append(kids[chromosome])

        return newPop


    # ==============================================================================
    # GET SUMMARY OF POPULATION
    # perform a simple summary on the population:
    #   1. returns the best chromosome fitness,
    #   2. the average population fitness, and
    #   3. the variance of the population fitness
    # ==============================================================================
    def summaryFitness(pop):
        a = np.array(list(zip(*pop))[1])
        return np.max(a), np.mean(a), np.min(a), np.std(a)


    # ==============================================================================
    # Simple head function - print top n chromosomes for a population
    # ==============================================================================
    def head(population, n=6): # note sorted by value desc
        print('Top %g Chromosomes of given Population:\n' % n,
              'Chrom.\t Value\t\t Weight\t\t\t Num. Items')
        for chromosome in range(1, n+1):
            print(' [%g]\t' % chromosome,
                  '%.1f'  %            population[chromosome-1][FITNESS_VALUE_IDX], '\t',  # Value (not
                  '%.1f'  % calcWeight(population[chromosome-1][ITEMS_IDX]), '\t', # Weight
                  '%g' % itemsSelected(population[chromosome-1][ITEMS_IDX])       # Num. Items Selecte


    # ==============================================================================
    # GET BEST SOLUTIONS
    # ==============================================================================
    def bestSolutionInPopulation(pop):
        print("Best solution: ", pop[0][0]) # Best solution should be the FIRST element! (sorting)
        print("Items selected: ", itemsSelected(pop[0][0]))
        print("Value: ", pop[0][1])
        print("Weight: ", calcWeight(pop[0][0]))


    # ==============================================================================
    # MAIN
    # ==============================================================================
    # GA main code
    Population = initializePopulation()

    # optional: you can output results to a file -- i've commented out all of the file out put for now
```

```
    # f = open('out.txt', 'w')  #---uncomment this line to create a file for saving output

    for j in range(Generations):

        # <--need to replace this with roulette wheel selection, e.g.:  mates=rouletteWheel(Population)
        mates=rouletteWheel(Population)
        Offspring = breeding(mates)
        Population = insert(Population, Offspring)

        # end of GA main code

        maxVal, meanVal, minVal, stdVal = summaryFitness(
            Population)  # check out the population at each generation
        # print to screen; turn this off for faster results
        # print("Iteration: ", j, summaryFitness(Population))

        # f.write(str(minVal) + " " + str(meanVal) + " " + str(varVal) + "\n")  #---uncomment this line

    # f.close()   #---uncomment this line to close the file for saving output

    print(summaryFitness(Population))
    bestSolutionInPopulation(Population)

    # Output a list for the summary output
    solutionOutput = [
        Generations, populationSize, crossOverRate, mutationRate, eliteSolutions,
        itemsSelected(Population[0][ITEMS_IDX]), calcWeight(Population[0][ITEMS_IDX]),
        Population[0][FITNESS_VALUE_IDX]
    ]

    return solutionOutput
```

### 2.2 Call Function for Genetic Algorithm

```
# ============================================================================
# CALL FUNCTION
# ============================================================================

GA_1 = geneticAlgorithmKnapsack(populationSize = n,    # size of GA population
                                Generations    = 250,  # number of GA generations
                                crossOverRate  = 0.9,  # 90% chance that two parents bred at some cross
                                mutationRate   = 0.10, # 10% chance an offspring is mutated between 1 a
                                eliteSolutions = 10    # Number prior pop to keep from gen to gen
                                )
```

```
## (32958.299999999996, 25218.475, 15313.7, 3673.7107857303718)
## Best solution:  [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0,
## Items selected:  18
## Value:  17957.6
## Weight:  2247.3
```

```
GA_2 = geneticAlgorithmKnapsack(populationSize = n,    # size of GA population
                                Generations    = 500,  # number of GA generations
                                crossOverRate  = 0.9,  # 90% chance that two parents bred at some cross
                                mutationRate   = 0.05, # 5% chance an offspring is mutated between 1 an
                                eliteSolutions = 20    # Number prior pop to keep from gen to gen
                                )
```

```
## (35212.100000000006, 30202.38046666667, 13064.1, 6366.004179538721)
## Best solution:  [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
## Items selected:  17
## Value:  17354.3
## Weight:  2485.7
```

```
GA_3 = geneticAlgorithmKnapsack(populationSize = n,    # size of GA population
                                Generations    = 1000, # number of GA generations
                                crossOverRate  = 0.9,  # 90% chance that two parents bred at some cross
                                mutationRate   = 0.025,# 2.5% chance an offspring is mutated between 1
                                eliteSolutions = 50    # Number prior pop to keep from gen to gen
                                )
```

```
## (35102.799999999996, 27953.876533333325, 11542.0, 10151.244358657312)
## Best solution:  [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0,
## Items selected:  16
## Value:  18519.2
## Weight:  2036.6999999999998
```

# 3 Summary Output of each Model

## 3.1 Simulated Annealing Output

| t[0] | Cooling,t[k] | M[k] | # Temps | Iters. | No. Items | Weight | Value |
|---|---|---|---|---|---|---|---|
| 1000 | Caunchy | 10 | 100 | 4764 | 24 | 2461.9 | 26618.4 |
| 500 | Caunchy | 5 | 200 | 5918 | 24 | 2415.5 | 25292.8 |
| 1000 | Boltzmann | 10 | 100 | 4877 | 21 | 2482.1 | 24807.0 |
| 750 | Boltzmann | 20 | 150 | 13115 | 27 | 2479.4 | 25363.5 |

## 3.2 Genetic Algorithm Output

| Generations | Pop Size | Crossover | Mutation | Elitism | No. Items | Weight | Value |
|---|---|---|---|---|---|---|---|
| 250 | 150 | 0.9 | 0.100 | 10 | 18 | 2247.3 | 17957.6 |
| 500 | 150 | 0.9 | 0.050 | 20 | 17 | 2485.7 | 17354.3 |
| 1000 | 150 | 0.9 | 0.025 | 50 | 16 | 2036.7 | 18519.2 |