Homework 5

Heuristic Search Methods

Daniel Carpenter

April 2022

Contents

Please see the final page for the summary output.

1 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

```
# 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
```

2 *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.

2.1 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
```

2.2 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
```

2.3 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
```

2.4 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
```

2.5 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)
```

2.6 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)
```

2.7 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))
```

2.8 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 == caunchyCoolSchedule:
        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)
```

2.9 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
##
## -----------------------------------
```

3 *Question 2:* Genetic Algorithm

3.1 Function for Genetic Algorithm

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

3.2 Call Function for Genetic Algorithm

4 Summary Output of each Model

```
# Get the list of questions to send to R
# output = [q2, q3, q4_1, q4_2, q5_1, q5_2]
```

```
# library(reticulate) # Package to convert Python to R / R
# to Python # >> Convert Python list to R list object <<
# pyList <- py$output # Create empty data frame to append
# results to df <- data.frame() for (i in 1:length(pyList))
# { newRow <- as.data.frame(pyList[i][1]) colnames(newRow)
# <- c('Iterations', '# Items Selected', 'Weight',
# 'Objective') # Append the new row df <- rbind(df, newRow)
# } rownames(df) <- c('Local Search with Best Improvement',
# 'Local Search with First Improvement', 'Local Search with
# Random Restarts (k=10)', 'Local Search with Random
# Restarts (k=50)', 'Local Search with Random Walk, using
# First Acceptence (p=0.75)', 'Local Search with Random
# Walk, using First Acceptence (p=0.01)' )
```

```
# knitr::kable(df)
```