Homework 4

Hill Climbing Methods

Daniel Carpenter & Kyle (Chris) Ferguson

April 2022

Contents

Please see the final page for the summary output.

Question 1: Strategies

(a) Initial Solution

Define and defend a strategy for determining an initial solution to this knapsack problem for a neighborhood-based heuristic.

- Our algorithm for the `initial_solution()` function randomly generates a list of binary $\in (0, 1)$ values for the knapsack problem, `1` if an item is included in the knapsack and `0` if the item is excluded
- Since the solution could randomly generate an infeasible solution (i.e., the `totalWeight` $\geq$ `maxWeight`), the `initial_solution()` function handles it by randomly removing items from the knapsack until it is under the `maxWeight`.
- After the generation of the initial solution, the evaluate function searches for better solutions.
- We considered beginning with nothing in the knapsack (list of `0`'s from item `0` to `n`), but we researched and found that a common approach is to begin with a randomly generated solution.

(b) Neighborhood Structures

Describe 3 neighborhood structure definitions that you think would work well for this problem. Compute the size of each neighborhood.

1. Using variable neighborhood search, the algorithm attempts to find a "global optimum", where it explores "distant" neighborhoods relative to the incumbent solution. Similar to other approaches, it will repeat until it finds a local optima. This approach may provide an enhancement since it will compare the incumbent solution to other solutions "far" from it, providing a better opportunity to finding the global maximum. *Metaheuristics—the metaphor exposed* by Kenneth Sorensen provides an overview of this concept as well.
2. Simulated annealing may also work well since it will analyze multiple items to be placed in the knapsack; however, some items may be chosen over others which could cause a local minimum to occur. See *Metaheuristics—the metaphor exposed* by Kenneth Sorensen page 7.
3. Lastly, the bee colony swarm algorithm could allow for a good approach when overcoming local optima. Since the method keeps track of solutions and allows for exploration of other solutions, it may allow a similar approach to how k random restarts works.
4. Without any adjustment to the neighborhoods: For each neighborhood, there are 150 neighbors. Since the knapsack problem uses a n-dimensional binary vector, the total solution space is 2^n, which is 2^{150}

(c) Infeasibility

During evaluation of a candidate solution, it may be discovered to be infeasible. In this case, provide 2 strategies for handling infeasible solutions:

Note both approaches are similar:

1. *Chosen Method in model:* If the solution is infeasible (i.e., the `totalWeight` $\geq$ `maxWeight`), then we will *randomly* remove values from the knapsack until the bag's weight is less than the max allowable weight. The function then recursively reevaluates using the `evaluation()` function.

2. If the solution is infeasible, then we will *iteratively* (from last item in list to beginning) remove values from the knapsack until the bag's weight is less than the max allowable weight.

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
from random import Random  # need this for the random number generation -- do not change
import numpy as np

# Set the seed
seed = 51132021
myPRNG = Random(seed)

n = 150 # number of elements in a solution

# create an "instance" for the knapsack problem
value = []
for i in range(0, n):
    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
```

Key Global Functions

Functions to provide *initial solution*, create a *neighborhood* and *evaluate* better solutions
Please assume these are referenced by following code chunks

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

# monitor the number of solutions evaluated
solutionsChecked = 0

# function to evaluate a solution x
def evaluate(x, r):

    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 ---------------------------------------------------
```

```python
    # 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]


# ===============================================================================
# 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


# ===============================================================================
# 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
```

Question 2: Local Search with Best Improvement

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

# variable to record the number of solutions evaluated
solutionsChecked = 0

x_curr = initial_solution()  # x_curr will hold the current solution
x_best = x_curr[:]  # x_best will hold the best solution

r = randIdx = myPRNG.randint(0,n-1) # a random index for evaluation

# f_curr will hold the evaluation of the current soluton
f_curr = evaluate(x_curr, r)
f_best = f_curr[:]


## BEGIN LOCAL SEARCH LOGIC ---------------------------------------------------
done = 0

while done == 0:

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

    for s in Neighborhood:  # evaluate every member in the neighborhood of x_curr
        solutionsChecked = solutionsChecked + 1
        if evaluate(s, r)[0] > f_best[0]:

            # find the best member and keep track of that solution
            x_best = s[:]
            f_best = evaluate(s, r)[:]  # and store its evaluation

    # Checks for platueau and feasibility
    if f_best == f_curr and (f_curr[1] < maxWeight):  # if there were no improving solutions in the nei
        done = 1

    else:
        x_curr = x_best[:]  # else: move to the neighbor solution and continue
        f_curr = f_best[:]  # evalute the current solution

        # print("\nTotal number of solutions checked: ", solutionsChecked)
        # print("Best value found so far: ", f_best)

print("\nFinal number of solutions checked: ", solutionsChecked, '\n',
      "Best value found: ", f_best[0], '\n',
      "Weight is: ", f_best[1], '\n',
      "Total number of items selected: ", np.sum(x_best), '\n\n',
      "Best solution: ", x_best)

# for the summary output
```

```
##
## Final number of solutions checked:  11250
##  Best value found:  24760.699999999997
##  Weight is:  2427.6
##  Total number of items selected:  20
##
##  Best solution:  [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
```

```
q2 = [solutionsChecked, np.sum(x_best), f_best[1], f_best[0]]
```

Question 3: Local Search with First Improvement

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

# variable to record the number of solutions evaluated
solutionsChecked = 0

x_curr = initial_solution()  # x_curr will hold the current solution
x_best = x_curr[:]  # x_best will hold the best solution

r = randIdx = myPRNG.randint(0,n-1) # a random index

# f_curr will hold the evaluation of the current soluton
f_curr = evaluate(x_curr, r)
f_best = f_curr[:]


## BEGIN LOCAL SEARCH LOGIC ---------------------------------------------------
done = 0

while done == 0:

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

    for s in Neighborhood:  # evaluate every member in the neighborhood of x_curr
        solutionsChecked = solutionsChecked + 1
        if evaluate(s, r)[0] > f_best[0]:

            # find the best member and keep track of that solution
            x_best = s[:]
            f_best = evaluate(s, r)[:]  # and store its evaluation

            break # >> Exit loop << (first accept change from best acceptance)

    # Checks for platueau and feasibility
    if f_best == f_curr and (f_curr[1] < maxWeight):  # if there were no improving solutions in the nei
        done = 1

    else:
        x_curr = x_best[:]  # else: move to the neighbor solution and continue
        f_curr = f_best[:]  # evalute the current solution

        print("\nTotal number of solutions checked: ", solutionsChecked)
        print("Best value found so far: ", f_best)
```

```
##
## Total number of solutions checked:  1
## Best value found so far:  [16370.1, 2484.5]
##
## Total number of solutions checked:  4
## Best value found so far:  [14764.6, 2378.2000000000003]
```

```
##
## Total number of solutions checked:  5
## Best value found so far:  [15264.9, 2432.8]
##
## Total number of solutions checked:  9
## Best value found so far:  [14243.6, 2460.2]
##
## Total number of solutions checked:  14
## Best value found so far:  [12617.800000000001, 2276.7]
```

```
print("\nFinal number of solutions checked: ", solutionsChecked, '\n',
      "Best value found: ", f_best[0], '\n',
      "Weight is: ", f_best[1], '\n',
      "Total number of items selected: ", np.sum(x_best), '\n\n',
      "Best solution: ", x_best)

# for the summary output
```

```
##
## Final number of solutions checked:  17
##  Best value found:  12617.800000000001
##  Weight is:  2276.7
##  Total number of items selected:  15
##
##  Best solution:  [1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

```
q3 = [solutionsChecked, np.sum(x_best), f_best[1], f_best[0]]
```

Question 4: Local Search with Random Restarts

4.1 Hill Climbing First Accept Function `hillClimbFirstAccept()`

Function that includes all of the hill climbing with *first acceptance* logic
Returns a list of best solution found (see below for details of list)

```
# Returns a list of the best solution found:
#   [0] totalValue:       Total value of the value bag
#   [1] totalWeight:      Associated weight of the bag
#   [2] solutionsChecked: Number of solutions checked
#   [3] numberOfItems:    Total number of items packed
#   [4] itemsPacked:      A list of the items packed

# The indices of the solutions returned from `hillClimbFirstAccept()` function
VALUE_IDX      = 0 # The value index of the output to the hill climb function
WEIGHT_IDX     = 1 # Weight of the solution
SOL_CHCKED_IDX = 2 # The numner of solutions checked
NUM_ITEMS_IDX  = 3 # The number of items in the solutions knapsack
ITEMS_PCKD_IDX = 4 # List of the items packed

def hillClimbFirstAccept():

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

    # variable to record the number of solutions evaluated
    solutionsChecked = 0

    x_curr = initial_solution()  # x_curr will hold the current solution
    x_best = x_curr[:]  # x_best will hold the best solution

    r = randIdx = myPRNG.randint(0,n-1) # a random index

    # f_curr will hold the evaluation of the current soluton
    f_curr = evaluate(x_curr, r)
    f_best = f_curr[:]


    ## BEGIN LOCAL SEARCH LOGIC --------------------------------------------------
    done = 0

    while done == 0:

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

        for s in Neighborhood:  # evaluate every member in the neighborhood of x_curr
            solutionsChecked = solutionsChecked + 1
            if evaluate(s, r)[0] > f_best[0]:

                # find the best member and keep track of that solution
                x_best = s[:]
                f_best = evaluate(s, r)[:]  # and store its evaluation
```

```
            break # >> Exit loop << (first accept change from best acceptance)

    # Checks for plateau and feasibility
    if f_best == f_curr and (f_curr[1] < maxWeight):  # if there were no improving solutions in the
        done = 1

    else:
        x_curr = x_best[:]  # else: move to the neighbor solution and continue
        f_curr = f_best[:]  # evalute the current solution

        # print("\nTotal number of solutions checked: ", solutionsChecked)
        # print("Best value found so far: ", f_best)


return [              # Return a list of important values:
    f_best[0],        # totalValue
    f_best[1],        # totalWeight
    solutionsChecked, # solutionsChecked
    np.sum(x_best),   # numberOfItems
    x_best            # itemsPacked
    ]
```

4.2 Random Restarts Function `kRestartsHillClimbFirstAccept()`

Function that calls the first acceptance function and repeats `k` number of times
Returns the best solution, best solution's index, and the list of restarted solutions

```
def kRestartsHillClimbFirstAccept(k_restarts, numSolutionsToShow):

    # List of the optimal solutions, including the returned output from the
    # `hillClimbFirstAccept()` function
    optimalSolutions = []
    bestIdx          = 0  # Stores the index of the best value

    # Iterate throug k restarts of hill climbing with first accept
    for theCurrentRestart in range(0, k_restarts):
        optimalSolutions.append(hillClimbFirstAccept())

        # See the optimal value of the restart
        # print('Sol. Idx: [%g]' % theCurrentRestart, '\tVal: %g' %
        #       optimalSolutions[theCurrentRestart][VALUE_IDX]) # Comment to hide best value from resta

        # Check to see if the current solution is better than the incumbant.
        if (theCurrentRestart != 0) and (  optimalSolutions[theCurrentRestart][VALUE_IDX]
                                         > optimalSolutions[bestIdx][VALUE_IDX]):

            # If this solution is better, then store it as the best index
            bestIdx = theCurrentRestart


    # Simple function to print a solution (from list idx) of restarted solutions
    def printSolution(solutionIdx):
```

```
        # Print the output
        print('Solution Index: ', solutionIdx, '\n',
              'Solution value:',  optimalSolutions[solutionIdx][VALUE_IDX], '\n',
              'Solution weight:', optimalSolutions[solutionIdx][WEIGHT_IDX], '\n',
              'Number of solutions checked:', optimalSolutions[solutionIdx][SOL_CHCKED_IDX], '\n',
              'Number of items in bag:',   optimalSolutions[solutionIdx][NUM_ITEMS_IDX], '\n',
              'List of items packed:',   optimalSolutions[solutionIdx][ITEMS_PCKD_IDX], '\n'
              )


    # RETRIEVE AND PRINT SOLUTIONS  -------------------------------------------

    # Print best solution
    print('\n-------- THE *BEST* SOLUTION --------'), printSolution(bestIdx)

    print('\n-------- %g Other Solutions --------\n' % numSolutionsToShow)

    # Print solutions (number to show defined in the function)
    for solutionNum in range(0, numSolutionsToShow):
        printSolution(solutionNum) # print another example

    # for the summary output
    q4 = [optimalSolutions[bestIdx][SOL_CHCKED_IDX],
          optimalSolutions[bestIdx][NUM_ITEMS_IDX],
          optimalSolutions[bestIdx][WEIGHT_IDX],
          optimalSolutions[bestIdx][VALUE_IDX]
          ]

    # Return the best solution, best idx, and the list of restarted solutions
    return (q4)
```

4.3 Call the function `kRestartsHillClimbFirstAccept()`

Call the function and show the first 2 solutions Show output with two values of k in inputs

```
numSolutionsToShow = 2  # Number of solutions to show. Could be the optimal FYI

# Call function - Random restarts with *first* acceptance hill climbing

## Restart 10 times
q4_1 = kRestartsHillClimbFirstAccept(k_restarts=10, numSolutionsToShow=2)

## Restart 50 times
```

```
##
## -------- THE *BEST* SOLUTION --------
## Solution Index:  2
##  Solution value: 16106.6
##  Solution weight: 2490.7000000000003
##  Number of solutions checked: 20
##  Number of items in bag: 21
##  List of items packed: [1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0
```

```
##
##
## -------- 2 Other Solutions --------
##
## Solution Index:  0
##  Solution value: 13006.8
##  Solution weight: 2449.8999999999996
##  Number of solutions checked: 208
##  Number of items in bag: 19
##  List of items packed: [1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
##
## Solution Index:  1
##  Solution value: 13627.5
##  Solution weight: 2381.0
##  Number of solutions checked: 7
##  Number of items in bag: 16
##  List of items packed: [1, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

```
q4_2 = kRestartsHillClimbFirstAccept(k_restarts=50, numSolutionsToShow=2)
```

```
##
## -------- THE *BEST* SOLUTION --------
## Solution Index:  9
##  Solution value: 19460.300000000003
##  Solution weight: 2438.7
##  Number of solutions checked: 14
##  Number of items in bag: 21
##  List of items packed: [1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0
##
##
## -------- 2 Other Solutions --------
##
## Solution Index:  0
##  Solution value: 17700.2
##  Solution weight: 2499.2999999999997
##  Number of solutions checked: 1
##  Number of items in bag: 19
##  List of items packed: [0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0
##
## Solution Index:  1
##  Solution value: 18250.7
##  Solution weight: 2386.9
##  Number of solutions checked: 6
##  Number of items in bag: 20
##  List of items packed: [1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0
```

Question 5: Local Search with Random Walk

5.1 Function for HC w. Random Walk, using *First Acceptance*

Function that performs hill climb with random walk, using best acceptance method. There is a probability of p to either hill climb or go on a random walk

```
def hillClimbRandWalkFirstAccept(prob=0.50):

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

    # variable to record the number of solutions evaluated
    solutionsChecked = 0

    x_curr = initial_solution()  # x_curr will hold the current solution
    x_best = x_curr[:]  # x_best will hold the best solution

    r = randIdx = myPRNG.randint(0,n-1) # a random index

    # f_curr will hold the evaluation of the current soluton
    f_curr = evaluate(x_curr, r)
    f_best = f_curr[:]


    ## BEGIN LOCAL SEARCH LOGIC ---------------------------------------------------
    done = 0

    while done == 0:

        # create a list of all neighbors in the neighborhood of x_curr
        Neighborhood = neighborhood(x_curr)
        test_prob = myPRNG.random() #chooses a random number between 0,1

        if test_prob >= prob: #if the random number is greater than the probability desired
            x_curr = Neighborhood[myPRNG.randint(0,n-1)][:]
            f_curr = evaluate(x_curr,r)


        else:
            for s in Neighborhood:  # evaluate every member in the neighborhood of x_curr
                solutionsChecked = solutionsChecked + 1
                if evaluate(s, r)[0] > f_best[0]:

                    # find the best member and keep track of that solution
                    x_best = s[:]
                    f_best = evaluate(s, r)[:]  # and store its evaluation

                    break # >> Exit loop << (first accept change from best acceptance)

        # Checks for platueau and feasibility
        if f_best == f_curr and (f_curr[1] < maxWeight):  # if there were no improving solutions in the
            done = 1
```

```
        else:
            x_curr = x_best[:]  # else: move to the neighbor solution and continue
            f_curr = f_best[:]  # evalute the current solution

            # print("\nTotal number of solutions checked: ", solutionsChecked)
            # print("Best value found so far: ", f_best)

    print("\nFinal number of solutions checked: ", solutionsChecked, '\n',
          "Best value found: ", f_best[0], '\n',
          "Weight is: ", f_best[1], '\n',
          "Total number of items selected: ", np.sum(x_best), '\n\n',
          "Best solution: ", x_best)

    # For printing final results
    q5 = [solutionsChecked, np.sum(x_best), f_best[1], f_best[0]]


    return (q5)
```

5.2 Call the function `hillClimbRandWalkFirstAccept()`

Show output with two values of p in inputs

```
# Probability of 75%
q5_1 = hillClimbRandWalkFirstAccept(prob=0.75)

# Probability of 1%
```

```
##
## Final number of solutions checked:  578
##  Best value found:  12149.5
##  Weight is:  2351.5999999999995
##  Total number of items selected:  18
##
##  Best solution:  [1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

```
q5_2 = hillClimbRandWalkFirstAccept(prob=0.01)
```

```
##
## Final number of solutions checked:  33
##  Best value found:  14784.599999999999
##  Weight is:  2462.2999999999997
##  Total number of items selected:  19
##
##  Best solution:  [1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

Summary Output of each Model

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

	Iterations	# Items Selected	Weight	Objective
Local Search with Best Improvement	11250	20	2427.6	24760.7
Local Search with First Improvement	17	15	2276.7	12617.8
Local Search with Random Restarts (k=10)	20	21	2490.7	16106.6
Local Search with Random Restarts (k=50)	14	21	2438.7	19460.3
Local Search with Random Walk, using First Acceptence (p=0.75)	578	18	2351.6	12149.5
Local Search with Random Walk, using First Acceptence (p=0.01)	33	19	2462.3	14784.6