

Homework 4

Hill Climbing Methods

Daniel Carpenter & Kyle (Chris) Ferguson

April 2022

Contents

Question 1: Strategies	2
(a) Initial Solution	2
(b) Neighborhood Structures	2
(c) Infeasibility	2
Global Variables	3
Key Functions	3
Question 2: Local Search with Best Improvement	6
Question 3: Local Search with First Improvement	12
Question 4: Local Search with Random Restarts	14
4.1 Hill Climbing First Accept Function <code>hillClimbFirstAccept()</code>	14
4.2 Random Restarts Function <code>kRestartsHillClimbFirstAccept()</code>	15
4.3 Call the function <code>kRestartsHillClimbFirstAccept()</code>	16
Question 5: Local Search with Random Walk	18
Function for HC w. Random Walk, using <i>Best Acceptance</i>	18
Call the Function <code>hillClimbRandWalk()</code>	19

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. 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}
2. 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.
3. 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.

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

    # 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]

# =====
# 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 solution
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 plateau 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[:] # evaluate the current solution

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

##
## Total number of solutions checked: 150
## Best value found so far: [20960.6, 2492.0]
##
## Total number of solutions checked: 300
## Best value found so far: [19036.8, 2384.2]
##
## Total number of solutions checked: 450
```

```

## Best value found so far: [20309.5, 2381.1000000000004]
##
## Total number of solutions checked: 600
## Best value found so far: [22066.3, 2473.0]
##
## Total number of solutions checked: 750
## Best value found so far: [22450.2, 2413.1]
##
## Total number of solutions checked: 900
## Best value found so far: [23532.4, 2450.9]
##
## Total number of solutions checked: 1050
## Best value found so far: [24362.699999999997, 2487.3]
##
## Total number of solutions checked: 1200
## Best value found so far: [23897.0, 2311.0]
##
## Total number of solutions checked: 1350
## Best value found so far: [25699.6, 2432.3]
##
## Total number of solutions checked: 1500
## Best value found so far: [26737.3, 2490.0]
##
## Total number of solutions checked: 1650
## Best value found so far: [23055.9, 2434.4]
##
## Total number of solutions checked: 1800
## Best value found so far: [23526.0, 2415.7000000000003]
##
## Total number of solutions checked: 1950
## Best value found so far: [24111.0, 2473.8]
##
## Total number of solutions checked: 2100
## Best value found so far: [23645.300000000003, 2297.5]
##
## Total number of solutions checked: 2250
## Best value found so far: [25447.9, 2418.8]
##
## Total number of solutions checked: 2400
## Best value found so far: [24266.2, 2462.6000000000004]
##
## Total number of solutions checked: 2550
## Best value found so far: [24061.899999999998, 2323.8]
##
## Total number of solutions checked: 2700
## Best value found so far: [25099.6, 2381.5]
##
## Total number of solutions checked: 2850
## Best value found so far: [24980.300000000003, 2499.7000000000003]
##
## Total number of solutions checked: 3000
## Best value found so far: [24718.9, 2462.2000000000003]
##
## Total number of solutions checked: 3150

```

```

## Best value found so far: [24980.300000000003, 2499.7000000000003]
##
## Total number of solutions checked: 3300
## Best value found so far: [24718.9, 2462.2000000000003]
##
## Total number of solutions checked: 3450
## Best value found so far: [24980.300000000003, 2499.7000000000003]
##
## Total number of solutions checked: 3600
## Best value found so far: [24718.9, 2462.2000000000003]
##
## Total number of solutions checked: 3750
## Best value found so far: [25835.0, 2482.1000000000004]
##
## Total number of solutions checked: 3900
## Best value found so far: [25369.3, 2305.8]
##
## Total number of solutions checked: 4050
## Best value found so far: [27046.199999999997, 2395.2]
##
## Total number of solutions checked: 4200
## Best value found so far: [28080.8, 2482.2]
##
## Total number of solutions checked: 4350
## Best value found so far: [26984.300000000003, 2489.8]
##
## Total number of solutions checked: 4500
## Best value found so far: [26518.6, 2313.5]
##
## Total number of solutions checked: 4650
## Best value found so far: [27103.6, 2371.6]
##
## Total number of solutions checked: 4800
## Best value found so far: [25812.5, 2456.0]
##
## Total number of solutions checked: 4950
## Best value found so far: [26073.9, 2493.5]
##
## Total number of solutions checked: 5100
## Best value found so far: [25812.5, 2456.0]
##
## Total number of solutions checked: 5250
## Best value found so far: [26073.9, 2493.5]
##
## Total number of solutions checked: 5400
## Best value found so far: [25812.5, 2456.0]
##
## Total number of solutions checked: 5550
## Best value found so far: [26073.9, 2493.5]
##
## Total number of solutions checked: 5700
## Best value found so far: [25812.5, 2456.0]
##
## Total number of solutions checked: 5850

```



```

## Best value found so far: [26073.9, 2493.5]
##
## Total number of solutions checked: 6000
## Best value found so far: [25812.5, 2456.0]
##
## Total number of solutions checked: 6150
## Best value found so far: [26073.9, 2493.5]
##
## Total number of solutions checked: 6300
## Best value found so far: [25812.5, 2456.0]
##
## Total number of solutions checked: 6450
## Best value found so far: [26073.9, 2493.5]
##
## Total number of solutions checked: 6600
## Best value found so far: [25900.4, 2477.8]
##
## Total number of solutions checked: 6750
## Best value found so far: [23680.3, 2366.8]
##
## Total number of solutions checked: 6900
## Best value found so far: [25437.1, 2458.7000000000003]
##
## Total number of solutions checked: 7050
## Best value found so far: [24394.199999999997, 2415.3]
##
## Total number of solutions checked: 7200
## Best value found so far: [25532.1, 2488.5]
##
## Total number of solutions checked: 7350
## Best value found so far: [25066.4, 2312.2]
##
## Total number of solutions checked: 7500
## Best value found so far: [26446.5, 2489.0]
##
## Total number of solutions checked: 7650
## Best value found so far: [26101.1, 2487.8]
##
## Total number of solutions checked: 7800
## Best value found so far: [25635.4, 2311.5]
##
## Total number of solutions checked: 7950
## Best value found so far: [26717.6, 2349.3]
##
## Total number of solutions checked: 8100
## Best value found so far: [27547.9, 2385.7]
##
## Total number of solutions checked: 8250
## Best value found so far: [26336.699999999997, 2472.6]
##
## Total number of solutions checked: 8400
## Best value found so far: [26682.1, 2473.8]
##
## Total number of solutions checked: 8550

```

```

## Best value found so far: [25788.1, 2388.5]
##
## Total number of solutions checked: 8700
## Best value found so far: [27465.0, 2477.8999999999996]
##
## Total number of solutions checked: 8850
## Best value found so far: [26289.199999999997, 2439.9]
##
## Total number of solutions checked: 9000
## Best value found so far: [25823.5, 2263.6]
##
## Total number of solutions checked: 9150
## Best value found so far: [27465.0, 2477.8999999999996]
##
## Total number of solutions checked: 9300
## Best value found so far: [25066.1, 2499.0]
##
## Total number of solutions checked: 9450
## Best value found so far: [24600.4, 2322.7]
##
## Total number of solutions checked: 9600
## Best value found so far: [25584.9, 2429.8]
##
## Total number of solutions checked: 9750
## Best value found so far: [26415.2, 2466.2]
##
## Total number of solutions checked: 9900
## Best value found so far: [24643.5, 2484.8999999999996]
##
## Total number of solutions checked: 10050
## Best value found so far: [24177.8, 2308.6]
##
## Total number of solutions checked: 10200
## Best value found so far: [25934.6, 2400.5]
##
## Total number of solutions checked: 10350
## Best value found so far: [24504.899999999998, 2442.6]
##
## Total number of solutions checked: 10500
## Best value found so far: [25365.699999999997, 2489.7999999999997]
##
## Total number of solutions checked: 10650
## Best value found so far: [24900.0, 2313.5]
##
## Total number of solutions checked: 10800
## Best value found so far: [25934.6, 2400.5]
##
## Total number of solutions checked: 10950
## Best value found so far: [24677.399999999998, 2445.5]
##
## Total number of solutions checked: 11100
## Best value found so far: [24760.699999999997, 2427.6]

```

```
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)
```

[illegible]

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 solution
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 nei
        done = 1

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


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_CHKCD_IDX  = 2 # The number 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[:] # evaluate the current solution

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

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

```

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 through 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 restarts

        # Check to see if the current solution is better than the incumbent.
        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):

```



```
##  
## 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]  
##  
## [19460.300000000003, 19, [[13006.8, 2449.8999999999996, 208, 19, [1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1
```

Question 5: Local Search with Random Walk

Function for HC w. Random Walk, using *Best 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
Binomial experiment chosen since it models a probability of success or failure,
in this case a success is considered continuing to climb the hill and search
for better solution. Failure is going on a random walk.

```
def hillClimbRandWalk(probOfClimb=0.50): # default probability 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 solution
    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 plateau and feasibility
        if f_best == f_curr and (f_curr[1] < maxWeight): # if there were no improving solutions in the

            # 1 if take hill climb path, 0 if take random walk path
            # Binomial chosen since gives result given probability of success
            takeHillClimbPath = np.random.binomial(1, probOfClimb, 1)

            # If the binomial experiment chooses to take the hill climb path,
            # then we know we are done. Else, choose a random neighbor
            if takeHillClimbPath == 1:
```

```
done = 1

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

    # Commented out just to keep the output cleaner
    # 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)
```

Call the Function `hillClimbRandWalk()`

Call the function, given the probability to hill climb (rather than random walk)

```
hillClimbRandWalk(probOfClimb=0.33)
```

[illegible]