

# Homework 4

## Hill Climbing Methods

Daniel Carpenter & Kyle (Chris) Ferguson

April 2022

### Contents

<b>Question 1: Strategies</b>	<b>2</b>
(a) Initial Solution . . . . .	2
(b) Neighborhood Structures . . . . .	2
(c) Infeasibility . . . . .	2
<b>Global Variables</b>	<b>3</b>
<b>Key Functions</b>	<b>3</b>
<b>Question 2: Local Search with Best Improvement</b>	<b>5</b>

## 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. *TODO*
3. *TODO*

### (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*

```
# 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

```
# =====
# 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