Local Search: Some Algorithms, Explanations and Sample Codes

Mir Imtiaz Mostafiz

1 Example Problem

In this tutorial, we will use 8 Queen problem as an example. Each solution will be a string which is a permutation of "12345678"- denoting the index of columns for each particular rows. For example, "34125768" means the 8 queens are in - (row 1,column 3), (row 2, column 4), (row 3, column 1) and so on.

2 helper.py

helper.py is a python script which contains the implementations of regular routines in a local search algorithm. This file is just like a C header or Java package file. You are asked to put all those 4 python files (Helper, HillClimbing, SteepestAscentHillClimbing and SimulatedAnnealing) in same folder.

Listing 1: helper.py

```
"""
Header files for random number generation, math operations and permutation generation
"""
import random
import math
from itertools import permutations

def generate_random_permutation(combination = '12345678'):
    """Generates a random permutation of given combination string.
    Keyword arguments:
    combination -- the input string to be permuted(default "12345678")
    returns a random permutation
    """

    # Get all permutations of string combination
    perms = permutations(combination)
    # Making the list of all permutations
    permList = list(perms)
    # generating a random index in [0,len(list)-1]
    idx = random.randint(0,len(permList))
    #concataneting all the characters of new permutation
    random_combination = ''.join(permList[idx])

    return random_combination


def printBoard(combination):
    """Prints the chessboard.

    Keyword arguments:
    combination -- a string denoting which row contains queens in which column

    """
```

```
    #board to be print, dimension 8X8
    board_array = []
    #Filling the board with *
    for i in range(0,8):
        board_array.append([])
        for j in range(0,8):
            board_array[i].append("*")
    #Placing queens in the board based on input combination
    for i in range(len(combination)):
        board_array[i][int(combination[i])-1] = "Q"
    for i in range(0,8):
        print(board_array[i])

def shuffle_function(combination_main):
    """Shuffle a combination string.

    Keyword arguments:
    combination_main -- input combination string

    """
    #Making temporary string
    combination = combination_main [:]
    #Generating random start and endpoints for shuffling
    i = random.randint(0,len(combination)-2)
    j = random.randint(i+1,len(combination)-1)

    list_combination = list(combination)
    #Shuffling in [i,j] interval
    shuffle_list = list_combination[i:j+1]
    random.shuffle(shuffle_list)
    list_combination[i:j+1] = shuffle_list
    combination = ''.join(list_combination)
    return combination

def swap_function(combination_main):
    """Shuffle a combination string.

    Keyword arguments:
    combination_main -- input combination string

    """

    combination = combination_main [:]
    #Generating random start and endpoints for shuffling
    i = random.randint(0,len(combination)-2)
    j = random.randint(i,len(combination)-1)
    #Swapping in [i,j] interval
    list_combination = list(combination)
    list_combination[i],list_combination[j] = list_combination[j],list_combination[i]
    combination = ''.join(list_combination)
    return combination

def generate_next_state(combination, tweak_function = swap_function):
    """generates next random state of a given state
    Keyword arguments:
    combination- input state
    tweak_function- the tweaking function you want to use- it can be
        swap_function(default) or shuffle_function
    returns a new generated state
    """
    return tweak_function(combination)
```

```
def fitness_function(combination):
    """calculates the fitness of a given state, i.e.- number of non-attacking queen pairs
    Keyword arguments:
    combination- input state

    returns the fitness/ number of non-attacking pairs in this case
    """
    fitness = 28 #non-attacking pairs

    for i in range(0,7):
        for j in range(i+1,8):

            if abs(i-j) == abs( int(combination[i])- int(combination[j])):
                fitness -= 1 #one attacking pair found
    return fitness
```

3 Hill Climbing

3.1 Reading Materials

Chapter 2: Algorithm 4 - Essentials of Metaheuristics.

3.2 Algorithm

Algorithm 1: Hill Climbing

```
S = some initial candidate solution
repeat
    R = tweak(Copy(S))
    if Quality(R)>Quality(S) then
        S= R

until (S is the ideal solution || we have run out of time)
return S
```

3.3 Implementation

Listing 2: HillClimbing.py

```
"""
import header.py and all its functions
"""
import helper
from helper import *

def do_hill_climbing(tweak_function = swap_function):
    """
    Runs hill climmbing algorithm
    Keyword argument:
    tweak_function- the tweaking function you want to use- it can be
        swap_function(default) or shuffle_function
    returns solution state and its fitnes
    """
    #Initialization step
    current_fitness = None
    current = generate_random_permutation()
```

```
    iteration = 200 #number of iterations, you can change it

    while(iteration>=0):
        iteration -=1
        current_fitness = fitness_function(current) #calculating fitness
        #print('current',current, current_fitness)
        if current_fitness == 28:
            break
        #Modification step
        #generates next step and calculates fitness
        neighbour = generate_next_state(current,tweak_function)
        neighbour_fitness = fitness_function(neighbour)
        #print('neighbour',neighbour, neighbour_fitness)
        if current_fitness < neighbour_fitness:
            #print("assigning")
            current = neighbour

    return current,current_fitness

if __name__ == "__main__":

    random.seed()
    print("Solving 8 queen problem")

    #You can use shuffle_function instead of swap_function
    solution, fitness = (do_hill_climbing(swap_function))
    print("Solution using Hill Climbing")
    printBoard(solution)
    print("Fitness is ",fitness)
```

4 Steepest Ascent Hill Climbing

4.1 Reading Materials

Chapter 2: Algorithm 5 - Essentials of Metaheuristics.

4.2 Algorithm

Algorithm 2: Steepest Ascent Hill Climbing

```
1  S = some initial candidate solution
2  n = number of tweaks desired to sample the gradient
3  repeat
4      R = tweak(Copy(S))
5      for n − 1 times do
6          W = tweak(Copy(S))
7          if Quality(W)>Quality(R) then
8              R= W
9
10     if Quality(R)>Quality(S) then
11         S= R
12
13 until (S is the ideal solution || we have run out of time)
14 return S
```

4.3 Implementation

Listing 3: SteepestAscentHillClimbing.py

```
"""
import header.py and all its functions
"""

import helper
from helper import *

def do_steepest_ascent_hill_climbing(tweak_function = swap_function):

    """
    Runs steepest ascent hill climmbing algorithm
    Keyword argument:
    tweak_function- the tweaking function you want to use- it can be
        swap_function(default) or shuffle_function
    returns solution state and its fitnes
    """
    #Initialization step
    current_fitness = None
    current = generate_random_permutation()
    iteration = 200 #number of iterations, you can change it
    number_of_tweaks = 10 #number of tweaks, you can change it

    while(iteration>=0):
        iteration -=1
        current_fitness = fitness_function(current) #calculating fitness
        #print('current',current, current_fitness)
        if current_fitness == 28:
            break
        #Modification step
        #generates next step and calculates fitness

        neighbour = generate_next_state(current,tweak_function)

        neighbour_fitness = fitness_function(neighbour)
        #print('neighbour',neighbour, neighbour_fitness)
        #Choosing new generation from candidates
        for i in range(1,number_of_tweaks):

            candidate_neighbour = generate_next_state(current,tweak_function)
            candidate_neighbour_fitness = fitness_function(neighbour)
            if neighbour_fitness < candidate_neighbour_fitness:
                #print("assigning")
                neighbour = candidate_neighbour


        if current_fitness < neighbour_fitness:
            #print("assigning")
            current = neighbour

    return current,current_fitness


if __name__ == "__main__":

    random.seed()
    print("Solving 8 queen problem")
    #You can use shuffle_function instead of swap_function
    solution, fitness = (do_steepest_ascent_hill_climbing(swap_function))
```

```
    print("Solution using Steepest Ascent Hill Climbing")
    printBoard(solution)
    print("Fitness is ",fitness)
```

5 Simulated Annealing

5.1 Reading Materials

Chapter 4.1.2- Artificial Intelligence: A modern approach by Russel and Norvig.

5.2 Algorithm

Algorithm 3: Simulated Annealing

Input: Schedule: A scheduling Function

S = some initial candidate solution
t = temparature/time
for $t = 1$ *to* ∞ **do**
    T = Schedule(t)
    if $T = 0$ **then**
        return S
    R = tweak(Copy(S))
    ΔH = Quality(R)−Quality(S)
    if $\Delta H > 0$ **then**
        S= R
    else
        threshold_probability = $e^{\frac{\Delta H}{T}}$
        random_probabiltiy = a random number uniformly chosen between 0.0 and1.0
        if *random_probabiltiy* $\leq$ *threshold_probability* **then**
            S= R
return S

5.3 Implementation

Listing 4: SimulatedAnnealing.py

```
"""
import header.py and all its functions
"""

import helper
from helper import *

"""
Scheduling functions
"""

def do_linear_schedule(t):

    """
    generates a temparature based on input time(linear)
    keywords:
    t - input time
    returns 100-0.5*t
```

```
    """
    T = 100-0.5*t
    return T

def do_exponential_schedule(t):
    """
    generates a temparature based on input time(exponential)
    keywords:
    t - input time
    returns 100*math.exp(-0.5*t)
    """


    T = 100*math.exp(-0.5*t)
    return T

def do_simulated_annealing(tweak_function=swap_function, schedule_function =
     do_linear_schedule):
    """
    Runs hill climmbing algorithm
    Keyword argument:
    tweak_function- the tweaking function you want to use- it can be
        swap_function(default) or shuffle_function
    schedule_function- the scheduling function you want to use- it can be
        do_linear_schedule(default) or do_exponential_schedule
    returns solution state and its fitnes
    """
    #Initialization step

    current = generate_random_permutation()


    iteration = 200

    for t in range(0,iteration):

        current_fitness = fitness_function(current)
        T = schedule_function(t)
        if T <= 0:
            return current,current_fitness
        #Modification step
        #generates next step and calculates fitness

        neighbour = generate_next_state(current,tweak_function)
        neighbour_fitness = fitness_function(neighbour)
        delta_E = neighbour_fitness - current_fitness
        if delta_E>0:
            current = neighbour

        else:
        #Choosing worse solution based on probability
            selection_probability = math.exp(delta_E/T)
            random_generated_value = random.random()
            if random_generated_value <= selection_probability:
                current = neighbour

    return current,current_fitness

if __name__ == "__main__":

    random.seed()
    print("Solving 8 queen problem")
```

```
#You can use shuffle_function instead of swap_function
solution, fitness = (do_simulated_annealing(swap_function,do_linear_schedule))
print("Solution using Simulated Annealing")
printBoard(solution)
print("Fitness is ",fitness)
```