University of Stirling
Computing Science and Mathematics

CSCU9YE - Artificial Intelligence

Lab 3: Local Search Algorithms - Hill-climbing

This lab builds on Lab 2. The idea is to implement local search algorithms to solve the Knapsack problem.

Given a knapsack of capacity *W*, and a number n of items, each with a weight and value. The objective is to maximise the total value of the items in the knapsack.

If for a given solution, the sum of the items' weights is greater than the capacity *W*, we say that the solution is invalid, as it does not satisfy the constraint. This happens when the total weight of the items exceeds the capacity. In your implementation of the algorithms for this lab, you should handle invalid solutions by discarding them, that is, when generating solutions, you will keep only those that satisfy the constraint.

We will use the same instances and format used in Lab 2. Here the instance data: knap20.txt, kanp200.txt, with 20 and 200 items, respectively.

Programming activities

Your task in this lab is to solve the Knapsack problem, using different strategies:

A. A modified version of the random search heuristic implemented in Lab 2, which will consider only valid solutions. That is, solutions with total weight smaller than the capacity of the sack.
B. A random mutation hill-climber using the 1-bitflip move operator
C. A multi-start version of the hill-climber algorithm B

To visualise your work you should produce two plots

I. A trace of the improving solutions obtained on a single run of algorithm B
II. A comparison using a box plot of the performance of algorithms A and C

Detailed description of your tasks:

- To facilitate your task, we give below the Python code for generating a random valid solution, where `cap` is the maximum capacity of the Knapsack, where `random_sol(n)` is the function you implemented in Lab 2, producing a random binary string of length *n*.

```
def random_sol_valid(n):
    binvalid = True
    while binvalid:
        s = random_sol(n)
        v, w = evaluate(s)
        binvalid = (w > cap)
    return s, v, w
```

- Implement a function called `def random_search_valid(tries)` which receives an integer number indicating the number of repetitions, and returns the best solution found, and the list with the values obtained in each try. Specifically, the function should return 4 values: `values, best_sol, best_val, best_wei`. Where `values` is a list containing the values of all the generated solutions. This is required for producing Plot II, that is comparing the performance of this algorithm against algorithm C. Your implementation will have a loop, which generates `tries` number of random valid solutions using `random_sol_valid(n).` You should keep the best solution found, to be able to return it at the end of the function. After calling your function, you should print the best solution found and its value and weight.

- Implement a random mutation best improvement hill-climbing method using a modular approach. Follow the guidelines below and implement the supporting functions:

    1. A function called `def local_optima(sol)`, which receives as input a solution (binary vector) and returns a Boolean value of `True` if the solution is a local optima with respect to the 1-bitflip neighborhood, and `False` if it is not. Remember that a solution is a local optima if it has better value than all its neighbours.

    2. A function called `def random_valid_neig(sol)`, which receives as input a solution `sol`, and returns a valid random neighbour of it, using the bit-flip move operators. This is the operator that changes a single bit from the solution. So you should first select a random position from the solution and flip it (change it to 0 if it is 1, and vice versa). The function should return 3 values the valid neighbour, its value and its weight.

    3. A function called `def hill_climbing(),` which implements a random mutation hill-climbing heuristics using the 1-bitflip neighborhood, using your functions 1 and 2. The stopping condition should be when a local optima has been reached. Function 1 provides you with this test. Your function should store all the improving solutions obtained in the path to the local optima. This will be used to produce Plot I, from a single run of the algorithm. Below you can find the pseudo-code of this function:

```
procedure hill-climbing
begin
  s = random initial solution solution
  repeat
     evaluate solution (s)
     s' = random neighbour of s
     if evaluation(s') is better than evaluation(s)
     s = s'
  until stopping-criteria satisfied
  return s
end
```

- Implement a function called `def multi_hc(tries)` which receives an integer number indicating the number of repetitions and returns the best solution found after `tries` number of calls to the `hill_climbing()` function. The function should also returns the list with the values obtained in each try. Specifically, the function should return 4 values: `values, best_sol, best_val, best_wei`. Where `values` is a list containing the values of all the obtained local optima. This is required for producing plot II, that is, comparing the performance of this algorithm against algorithm A. Your implementation will have a loop, which generates `tries` number of local optima using `hill_climbing.` You should keep the best local optimum found, to be able to return it at the end of

the function. After calling your function, you should print the best local optimum found and its value and weight.

Guidelines for producing the plots

I. **A trace of the improving solutions obtained on a single run of algorithm B**. Once the algorithm has terminated, plot the list of values using Matplotlib. Specifically use a `plt.plot(y)` call where *y* keeps the values. The plot function generates an *x* coordinate by default to match the size of your y vector. You should see line increasing, indicating the progress of the algorithm towards larger values, that is, better solutions.

II. **A comparison using a box plot of the performance of algorithms A and C.** You can produce a boxplot comparing two series of values using Matplotlib. To do so, combine in a list the two list of values. For example using the following Python lines:

```
results = [values_rnd, values_mhc]
plt.figure()
plt.boxplot(results,labels = ['random','hill-climbing'])
plt.show()
```

Checkpoint:

Source code and best solution found for:

1. Random search method
2. Random mutation hill-climbing method
3. Visualisation of Plot I
4. Visualisation of Polt II
5. Which algorithm produced the best results?