Programming languages (TC-2006)

Project 02

Read me first

- This project is supposed to be solved in teams of three people. Then, you will have to balance the work in order to finish it on time. If you do not have a team or you feel that you have no other option than solving this exam by yourself, notify the instructor as soon as possible by email to register this situation.
- A template with some codes is distributed along with this document. Please use the assigned spaces in the template to write your codes. Be aware that the grade will only be assigned to the students whose full names and IDs are visible in the code (see template). In other words: if your name and student ID are not in the file with the code, you will not receive any points.
- You are allowed to look for help. You can even discuss your solutions with your classmates. Hwever, **if any of the codes provided are not the result of your own work, the submission will be cancelled for all the team members**.
- Only one file must be submitted to Canvas as part of your submission: the template distributed along with this document but modified in such a way that it contains your implementations of the requested functions. If your submission contains more than one file, it will be cancelled.
- Only one of the team members must submit the solution to the project (that is the reason why we need the full names and student IDs in the file with your codes).
- Include only working code in your submission. If your code does not compile, that will automatically cancel your submission.

The knapsack problem

The knapsack problem (KP) is a fundamental and extensively studied problem in combinatorial optimization, not only for its theoretical interest but also because of its many practical applications. This problem is classified as NP-hard, and several exact and approximate algorithms have been developed to solve it.

The KP is formally defined as a set of n items, where each of these items has a profit p_j and a weight w_j, and a container with a capacity C. Solving a KP requires selecting a subset of items in such a way that their combined profit is maximized while their total weight remains under the container capacity, C. The KP can also be represented as a linear integer programming formulation, using equations (1)–(3).

$$\text{maximize} \quad \sum_{j=1}^{n} p_j x_j \tag{1}$$

$$\text{subject to} \quad \sum_{j=1}^{n} w_j x_j \leq C \tag{2}$$

$$x_j \in \{0,1\} \;\; j = 1, \ldots, n \tag{3}$$

There are many interesting variants of the KP but, for the sake of brevity, I have omitted a detailed discussion of them in this document.

In this project you will solve the knapsack problem by using a hill-climbing strategy, both sequential and concurrent, and see the benefits of concurrency when dealing with time-consuming processes.

The structure of the problem instances

In order to test your solving methods, I provide you with nine knapsack problem instances (from now on just referred to as instances). These instances are distributed along with the code template and can be accessed by calling the function `getInstance/1` in the code template. Each instance is defined by a tuple `{Capacity, Items}`, where `Capacity` is an integer with the knapsack capacity and `Items` is a list of items. In this list of items, each item is represented as a tuple `{Weight, Profit}`, where `Weight` indicates the weight of the item and `Profit`, its profit.
For example, the code:

```
{12, [{5, 10}, {4, 7}, {8, 1}, {3, 5}, {7, 10}]}
```

Represents an instance where the capacity (the maximum weight) of the knapsack is 12 units, and five items are available to be packed in such a knapsack. For example, the first of these items weights five units and has a profit of 10 units.

Evaluation of the solutions (15%)

Write a function in Erlang that, given an instance and a solution, calculates the weight packed inside the knapsack, as well as its corresponding profit. Complete the code for function `evaluate/2`, distributed along with the code template.
The first thing to understand is how to represent a solution for an instance. The simplest way is to define a solution as a list of booleans to mask the items. In other words, the positions in the solution that contain `true` indicate that its corresponding item is packed within the knapsack. If the value in the solution contains `false`, the corresponding item

in the instance is not packed within the knapsack.

For example, the code:

```
[true, false, false, false, true]
```

Indicates that only the first and fifth items will be packed into the knapsack (or, at least, we will try to pack). Then, if this solution applies to the previous knapsack problem example, the result will be `{12, 20}`. This means that, the weight of the items in the knapsack is 12 units, and its profit is 20 units.

IMPORTANT: Please note that even when the solution states that an item is to be packed within the knapsack, if it does not fit (its weight is larger than the current capacity of the knapsack) the item will not be packed and the next element in the solution will be revised. In other words, no item that exceeds the current capacity of the knapsack can be packed. In all the cases, the solution is evaluated from left to right.

Mutation of the solutions (15%)

So far we have a way to generate random solutions (see function `rndSolution/1` in the code template), but we also need a way to create new solutions based on existing ones. In other words, we need a way to 'mutate' existing solutions. Write a function `mutate/2` that takes a solution and a probability to mutate an element of the solution. The function must go through all the elements in the solution (recall that they are boolean values) and, with the probability provided as argument, change its value. Have in mind that the changes to the elements in the solution are independent.

Sequentially solving an instance (35%)

Implement a solver for the knapsack problem that works as follows (n stands for the number of solutions to evaluate):

```
procedure SOLVE(instance, n)
    solution ← RANDOMSOLUTION(instance)
    solution.eval ← EVALUATE(solution, instance)
    for i = 0 to n do
        candidate ← MUTATE(solution)
        candidate.eval ← EVALUATE(candidate, instance)
        if candidate.eval > solution.eval then
            solution ← candidate
            solution.eval ← candidate.eval
        end if
    end for
```

return solution
end procedure

If you take a closer look at the previous algorithm you should notice that this algorithm is just a slightly modified version of the hill climbing method. It is important to mention that the output of function `solve/2` must be a tuple with three elements: `{Solution, Weight, Profit}`, where `Solution` contains the actual solution found, and `Weight` and `Profit` indicate the total weight and profit packed within the knapsack, respectively.

Concurrently solving an instance (35%)

Implement another solver for the knapsack problem. This time, the solver will create and handle various processes in such a way that, instead of generating and evaluating the n solutions sequentially, this solver will divide the work into different processes. This new solver works as follows:

procedure SolveConcurrent(instance, n, m)
  for i = 0 **to** m **do**
    process[i] ← createProcess
    run Solve(instance, $\lfloor n/m \rfloor$) in process[i]
  end for
  solution ← best solution found from all the processes.
  return solution
end procedure

Please note that the concurrent solver relies on the sequential one. In other words, the concurrent solver will split the evaluation of the n solutions into m independent processes that are run in a concurrent way. Once the processes finish evaluating their solutions, the concurrent solver will collect the best solutions found by the independent processes (m solutions resulting from m processes) and return the best one of them. As in the previous case, the output of function `solveConcurrent/3` must be a tuple with three elements: `{Solution, Weight, Profit}`, where `Solution` contains the actual solution found, and `Weight` and `Profit` indicate the total weight and profit packed within the knapsack, respectively.

Is concurrency worth the effort?

Now that you have two different solvers for this problem (one sequential and another one concurrent), it is time to check which one is faster. In other words, is concurrency worth the effort? Distributed along the code template there is a function called `solve/3` that runs the solvers and returns both the running time of the process (in seconds) and the profit of the best solution found by each solver. Try solving some of the instances provided and

analyze your results. Although you are not requested to submit any evidence of this specific experiment, it is a good opportunity to see what happens when challenging problems must be solved, and how concurrency can help.

Deliverables

Prepare an ERL file that contains the functions requested (in its corresponding module) and submit it to Canvas. **Please, do not submit other formats but ERL**. To prepare your ERL file, use the code template distributed along with this document. The template contains some test cases for each function to help you verify that your codes work as requested.

I promise to apply my knowledge, strive for its development, and not use unauthorized or illegal means to complete this activity, following the Tecnológico de Monterrey Student Code of Honor.