Functionnal Programming

Prog1, Scala, L3

2015

1 The Knapsack Problem

Input Description:

- A set of items $S = \{1, ..., n\}$, where item i has size s_i and value v_i.
- A knapsack capacity C.

Problem: Find the subset $T \subset S$ which maximizes the value of $\sum_{i \in T} v_i$ given that $\sum_{i \in T} s_i \leq C$

At first, we suppose $\forall i, v_i = p_i$. Imagine that they are gold coins of differing sizes.

The goal of this exercise is to propose and compare several solutions to this classical problem. Please download the provided code. Inspect the `Instance.scala` file, which contains what you need to build a instances and solutions of this problem.

1.1 First approach: Recursive Branch and Bound

Please complete the file `BranchBoundSolver.scala`, with the following approach:

- Simply sweeps over all solutions with a recursive function (that you should write).
- For each possible solution, it it is better than the previously best solution known, it becomes the new best solution known.
- Partial solutions that are not possible (the knapsack is already full) are not even considered.

Once down test your code with some instances from `TestInstances.scala`.

1.2 Going Functionnal

The problem with the previous version is that it uses a variable to store the currently known best solution. Please complete the file `ExhaustiveSearch.scala`, that first generates all existing knapsacks in a `val`, filter that list to remove the invalid solutions (where the capacity is exceeded), and picks the maximal solution from the valid ones.

This version is preferable because it does not use any mutable data, but unfortunately, it is very inefficient as generating every knapsacks (even invalid ones) takes a lot of time. For large instances, the memory to store all instances will also become an issue.

1.3 Efficient Functionnal

Instead of generating all knapsacks, we will first cut the exploration for invalid partial solutions to save time. In addition, each recursive layer will directly return its best solution instead of returning a list of all valid solutions. This will greatly reduce the memory requirement.

Implement this approach in `SelectiveSearch.scala`. Since you don't know whether the exploration of a sub-tree will lead to a solution, the return type of your recursive function should be `Option[Solution]` (but the public interface of your `apply()` function should not be changed). Once you're done, test this code and compare the performance of your solutions.

1.4 Extension 1: Tests

Propose some particularly nasty problem instances that breaks more code. Instances without any solutions or with no perfect solution, or with a solution found on the first (last) branch or with several solutions are good candidates. Extra points if you can justify the quality of your test instances.

1.5 Extension 2: Not everything is a gold coin

Change your code to give both a size and a value to items. Don't fear the Shotgun Surgery ;)

1.6 Extension 3: Solve this problem using another Programming Style

Extra points if your style is original (clearly recognizable, but not listed in the lecture).

2 The N-queens Problem

Problem: Put n queens on a $n \times n$ board so than none of them can capture any other. Thus, no two queens can be placed on the same line, on the same row, or on the same diagonal.

Modeling is the most cruxial question when abording such a problem. You should first come up with a nice representation (a nice type) of the problem instances and of the partial solutions. This time, no code is provided. You are on your own.

2.1 Branch and Bound, or Functionnal ?

This problem is very similar to the Knapsack Problem. It can also be solved with either by:

- traversing all solutions, keeping track of the valid solutions found
- generating all solutions, and then filtering the valid ones
- searching all solutions and returning the valid ones directly

You should probably go for the third approach directly, unless you want to implement the other ones as an exercise.

2.2 Extension: Exercice de Style

As previously, I am interested in inovative solutions using other approaches and styles.

3 Higher Order Selective Search

There is a common pattern between both code that you wrote, and you should strive to factorize your code. The pieces of the puzzle could be the following:

- A function that given a partial solution, can say whether it is a valid solution or not.
- A function that given two partial solutions, can select the best one
- A function that given a state, can produce a set of partial solutions to explore next. It may be easier to conduct a Bread-first exploration instead of a Depth-first one. The set can be empty if we reached a leaf.
- A function that takes three functions as described above, and returns the best valid solution to the problem.