Academic year 2020-2021

Johannes Kepler Universität Linz

Programming in Julia

Block 2 — Project : part 1
Greedy heuristics for the set packing problem

Prof. Dr. Xavier Gandibleux

1 Objectives

This first part of the project aims to implement in Julia a heuristic solver for set packing problems (SPP). For a given numerical instance provided representing a SPP, you have to

1. implement the construction heuristic studied. This heuristic returns x_0 a feasible solution;
2. implement the improvement heuristic studied based on the k-p exchange from x_0;
3. conduct a numerical experiment of your algorithms on the numerical instances provided;
4. summarize your work and results in a short report.

2 Material provided

The material is available online on github (`https://github.com/xgandibleux/Linz2021`).

2.1 Instances

Two didactic instances and 5 test instances from `https://www.emse.fr/~delorme/SetPacking.html` are provided. The instances are weighted set packing problems. They are named as follow:

- `didactic.dat`
- `didacticLinz.dat`
- `pb_100rnd0100.dat`
- `pb_200rnd0100.dat`
- `pb_500rnd0100.dat`
- `pb_1000rnd0100.dat`
- `pb_2000rnd0100.dat`

2.2 Parser

A parser compliant with the format of the given instances is provided.

```julia
# ------------------------------------------------------
# Load an instance of SPP (format: OR-library)

function loadSPP(fname)
    f=open(fname)
    # Read the number of constraints (m) and variables (n)
    m, n = parse.(Int, split(readline(f)) )
    # Read the n coefficients of the objective function and create the vector C
    C = parse.(Int, split(readline(f)) )
    # Read the m constraints et rebuild the binary matrix A
    A=zeros(Int, m, n)
    for i=1:m
        # Read the number of non-zero values on the constraint i
        readline(f)
        # Read the indexes of non-zero values on the constraint i
        for valeur in split(readline(f))
          j = parse(Int, valeur)
          A[i,j]=1
        end
    end
    close(f)
    return C, A
end
```

2.3 Specifications

The formal definition of the two functions to code will comply these specifications:

```julia
# ------------------------------------------------------
# the construction heuristic

function GreedyConstruction(C, A)

   # To implement...

    return xfeas, zfeas
end

# ------------------------------------------------------
# the improvement heuristic

function GreedyImprovement(C, A, x, z)

   # To implement...

    return xbest, zbest
end
```

2.4 Run

The lines hereafter report a minimal main program on how to run your code :

```
using Printf
fname = "instances/didacticLinz.dat"

C, A = loadSPP(fname)

@time x, z = GreedyConstruction(C, A)
@printf("z(xInit) = %d \n\n",z)

@time xbest, zbest = GreedyImprovement(C, A, x, z)
@printf("z(xBest) = %d \n\n",zbest)
```

2.5 Results

Table 1 reports the trace of activity for the construction algorithm on a didactic instance.

```
Instance : instances/didacticLinz.dat

Construction
ivar   : [1, 2, 3, 4, 5, 6]
C      : [7, 2, 4, 6, 3, 1]
A      : [1 1 1 0 0 0; 0 1 0 0 1 1; 1 1 0 1 0 0; 0 1 1 0 1 0; 0 0 0 1 0 1; 0 0 1 1 1 0; 1 0 0 0 1 0]
U      : [2.33, 0.5, 1.33, 2.0, 0.75, 0.5]
jselec : 1
--------
ivar   : [6]
C      : [1]
A      : [1; 1]
U      : [0.5]
jselec : 1
--------
z(xInit) = 8
```

Table 1: Activity of the construction algorithm recorded on the instance `didacticLinz.dat`

With a naive implementation (which follows strictly the principles of the algorithm in reducing the structures A and C), Table 2 reports CPUt measured[1] at the end of the construction and the improvement phases. The improvement algorithm implements only a 1-1-exchange:

	CPUt (seconds)		$z(x)$	
Instance	construction	improvement	construction	improvement
pb_100rnd0100.dat	0.201024	0.074673	342	343
pb_200rnd0100.dat	0.445235	0.096216	351	357
pb_500rnd0100.dat	2.679592	0.098294	285	285
pb_1000rnd0100.dat	10.015316	0.091216	49	49
pb_2000rnd0100.dat	78.835489	0.090177	37	37

Table 2: CPUt with a naive implementation

This implementation meets the expected aims of the algorithm. But in looking the CPUt measured on the construction heuristic, the measures are growing very fast. This implementation appears clearly not efficient.

[1]The computer used for the experiments is a MacBook Pro produced in 2015 with a 3,1 GHz Intel Core i7 processor and a memory of 16 Go 1867 MHz DDR3. The OS is macOS High Sierra (v10.13.6)

With a revisited implementation, Table 3 reports CPUt measured at the end of the construction and the improvement phases. Moreover, the improvement algorithm implements now four moves: 2-1-exchange, 1-1-exchange, and 0-1-exchange:

	CPUt (seconds)		$z(x)$	
Instance	construction	improvement	construction	improvement
pb_100rnd0100.dat	0.233099	0.191633	342	351
pb_200rnd0100.dat	0.259617	0.270784	351	360
pb_500rnd0100.dat	0.326458	0.232689	285	285
pb_1000rnd0100.dat	0.570732	0.317668	49	49
pb_2000rnd0100.dat	2.995974	0.328321	37	37

Table 3: CPUt with a revisited implementation

The revisited implementation shows a much better performance of the algorithm in CPUt but also in quality of the solution. In looking carefully the activity of the improvement algorithm (see Table 4), we observe that the algorithm has triggered one 2-1-exchange, one 1-1-exchange and two 0-1-exchange.

```
> 2-1 : x
> 1-1 : x
> 0-1 : xx
```

Table 4: Activity of the improvement algorithm recorded on the instance pb_100rnd0100.dat