Algorithmic Methods for Mathematical Models

**COURSE PROJECT**

Eloy Gil Guerrero

eloy.gil@est.fib.upc.edu

**1. Problem statement**

Route scheduling is one of the most important tasks of logistic companies.

Given a set of locations including a distinguished location , every route starts at the same time from and also end at , maximum 12 hours later. All other locations should be visited once among all routes.

There is a known distance , in minutes, between each pair of locations and . Also, except for at each location the vehicle must perform a task during minutes, that must start in the range of a time window, therefore vehicles may arrive too early and wait until the task can start. Having an unlimited number of vehicles, the goal is to find the minimum number needed to visit all locations and their routes. If there are two solutions with the same number the best will be the one in which the latest vehicle arrives at its final destination sooner.

**2. OPL model[[1]](#footnote-1)**

**2.1 Data structures**

int nP = ...;

int S = nP+1;

int nT = nP;

int workTime = ...;

int bigM = workTime \* 2;

range T = 1..nT;

range P\_s = 1..nP;

range P = 1..S;

range window = 1..2;

int dist[p in P][q in P] = ...;

int task[p in P] = ...;

int task\_window[p in P\_s][w in window] = ...;

dvar int+ arrive\_pt[p in P\_s][t in T];

dvar int+ end\_time[t in T];

dvar boolean pt[p in P][t in T];

dvar boolean from\_p\_to\_q[t in T][p in P][q in P];

**2.2 Objetive function**

The S row of *pt* tells us the trucks which are in use. Multiplying it by *bigM* gives importance and if two solutions draw, it will break the tie adding the maximum arrival time.

minimize (sum(t in T) pt[S][t] \* bigM) + (max(t in T) end\_time[t]);

**2.3 Constraint definitions**

**2.3.1 Constraint set 1 (no time travelling allowed between two points)**

The time difference between the arriving time at the 2nd point and the arriving time at the 1st is equal or bigger than the distance between the 1st and the 2nd point plus the time spent in the task, if such a track actually exists. Otherwise this comparison is avoided using the *bigM* value that is a way bigger negative value.

forall (p in P\_s, q in P\_s)

forall (t in T)

arrive\_pt[q][t] - arrive\_pt[p][t] >= (from\_p\_to\_q[t][p][q] \* (dist[p][q] + task[p])) - (bigM \* (1-from\_p\_to\_q[t][p][q]));

**2.3.2 Constraint set 2 (same as Constraint 1 but for the special cases of Source/Destination point)**

forall (p in P\_s, t in T) {

arrive\_pt[p][t] >= from\_p\_to\_q[t,S,p] \* dist[S][p];

end\_time[t] >= arrive\_pt[p][t] + (from\_p\_to\_q[t][p][S] \* (dist[p][S] + task[p]));

}

**2.3.3 Constraint set 3 (task started during the specified time window)**

forall (p in P\_s, t in T) {

arrive\_pt[p][t] >= task\_window[p][1] \* pt[p][t];

arrive\_pt[p][t] <= task\_window[p][2] \* pt[p][t];

}

**2.3.4 Constraint set 4 (worktime behind maximum)**

forall (t in T)

end\_time[t] <= workTime;

**2.3.5 Constraint set 5 (each point is visited exactly once, except the Source/Destination point)**

forall (p in P\_s)

sum(t in T) pt[p,t] == 1;

**2.3.6 Constraint set 6 (defines the order of the visits)**

If truck *t* visits any point it also visits the Source/Destination point, avoids to travel from a point to itself and If truck *t* visits a point *p* it must have a previous and next point.

forall (t in T, p in P) {

pt[S][t] >= pt[p][t];

from\_p\_to\_q[t,p,p] == 0;

sum(q in P) from\_p\_to\_q[t,p,q] == pt[p][t];

sum(q in P) from\_p\_to\_q[t,q,p] == pt[p][t];

}

**3. Metaheuristics**

Due to the complexity of the optimization problem, heuristic algorithms are needed. I’ve implemented in Python a custom GRASP solver (with both Best and First Improvement) and also a BRKGA solver[[2]](#footnote-2).

**3.1 GRASP**

**3.1.1 Constructive phase pseudo-code**

**procedure** construct(quality(·), alpha, problem)

x = initialize empty solution with problem

initialize candidate set (C)

**while** C **is** not empty **do:**

C = sort(C) by quality, asc (lower value is better)

minC = worst quality value of candidates in C (last candidate)

maxC = best quality value of candidates in C (first candidate)

RCL = { s ∈ C | quality(s) <= minC + alpha \* (maxC – minC) }

select s randomly from the RCL

add s to x

update candidate set C

**end** while

**return** x

**end** construct

**3.1.2 Local search pseudo-code**

**procedure** localSearch(quality(·), N(·), x)

initialize bestNeighbour with solution x

H = { n ∈ N(x) | quality(n) < quality(bestNeighbour) } (lower value is better)

**while** H **is** not empty **do:**

q = +infinity

**foreach** neighbour **in** H **do:**

**if** quality(neighbour) < q **do:**

q = quality(neighbour)

bestNeighbour = neighbour

**if** algorithm is GRASP-FI**:** break

H = { n ∈ N(bestNeighbour) | quality(n) < quality(x) }

**end** while

**return** bestNeighbour

**end** localSearch

**3.1.3 Greedy function description**

The greedy function in GRASP is the same as the objective function in the OPL model, that is: *N \* K + T*

N is the number of routes of the solution, K is a big constant and T is the time spent by the last vehicle to arrive back to the starting point.

**3.2 BRKGA**

**3.2.1 Chromosome structure**

[ content goes here ]

**3.2.2 Decoder algorithm pseudo-code**

[ content goes here ]

**4. Performance comparison**

**4.1 CPLEX Execution Time evolution**

Using CPLEX we can obtain optimal solutions until the size of the instance makes the calculation impossible in an acceptable period of time. During the experimental phase, the biggest instance that is solved in less than 2 hours, in average, is instance 17[[3]](#footnote-3), with 17 destination points.

Figure 1

As it is shown in figure 1, it is clear that the average execution time evolution follows an exponential distribution, which makes the execution of the instance 18 (it already has 19 points, including the starting one) unfeasible. It could happen that after more than 5 hours of execution it is not even close, as it is shown in figure 2.

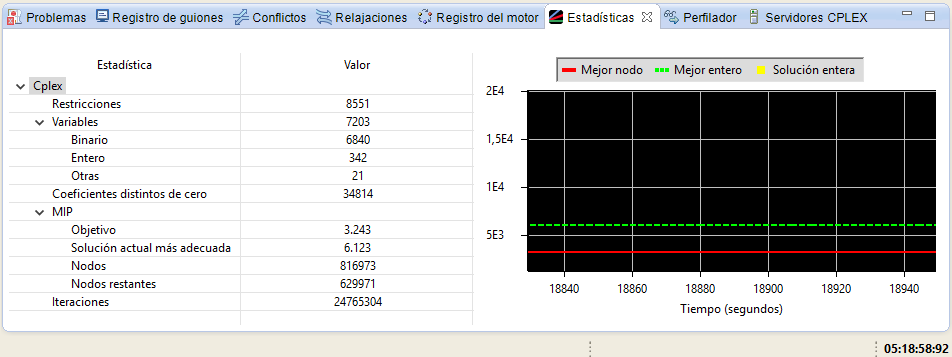
****

Figure 2

**4.2 Objective function value comparison**

**4.2.1 CPLEX optimal vs GRASP-BI, FI & BRKGA in 30s executions**

[ content goes here ]

**4.2.2 CPLEX optimal vs GRASP-BI, FI & BRKGA in 300s executions**

[ content goes here ]

**5. Conclusions**

This is an interesting project that makes possible to experiment with the different ways to get solutions for hard problems and also ensure that OPL is great to obtain optimal values when the size of the problem is not really big.

The use of heuristics is fundamental when the number of possibilities is huge and really useful to obtain a valid solution, assuming that it could be pretty far from the optimal one, but still good enough, even more thinking in the fact that it is being calculated in just a small period of time, in comparison.

1. Full model available in Project/Project.mod [↑](#footnote-ref-1)
2. All the source code of the solvers is available in the GRASP directory. [↑](#footnote-ref-2)
3. Instance 17 is located in InstanceGen/instance\_17.dat, like many other instances created automatically with my Python instance generator, that is InstanceGen/generator.py [↑](#footnote-ref-3)