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 l including a distinguished location l_s, every route starts at the same time from l_s and also end at l_s, maximum 12 hours later. All other locations should be visited once among all routes.
There is a known distance $dist_{l_1,l_2}$, in minutes, between each pair of locations l_1 and l_2. Also, except for l_s at each location the vehicle must perform a task during t_t minutes, that must start in the range of a $[min_l, max_l]$ 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.

Given:
Point set L
Starting point $s \in L$
Max time T
Interval set W

$$\forall\, l_i, l_j \in L : dist_{ij} = dist_{ji} = dist(l_i, l_j)$$

Produces:
Route set R
Assignment set A

$$\forall\, (r, l, t) \in A\colon r \in R,\ l \in L,\ t \in W_l$$

Minimize |R| and $\max(t)\ \forall\, (r, l, t) \in A \mid l = s$

2. OPL model[1]

2.1 Data structures

```
int nP = ...;
int S = nP+1;
int nT = nP;
int workTime = ...;
int bigM = workTime * 2;

range T = 1..nT;
```

[1] Full model available in Project/Project.mod

```
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 Instance example

```
nP = 4;         // number of destination points
workTime = 720; // maximum time to finish (12h in minutes)

// time (minutes) spent in a task in each point
task = [ 3 1 6 7 0 ];

              //  min  max
task_window = [[   84  132 ] // 1
               [   74  115 ] // 2
               [  238  284 ] // 3
               [  192  236 ] // 4
];

       //     1     2     3     4     S
dist = [[     0    77   116    22    73  ] // 1
        [    77     0     4    75    83  ] // 2
        [   116     4     0    57    48  ] // 3
        [    22    75    57     0    67  ] // 4
        [    73    83    48    67     0  ] // S
];
```

2.3 Objective 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]);
```

2.4 Constraint definitions

2.4.1 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]));
```

2.4.2 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]));
}
```

2.4.3 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];
   }
```

2.4.4 Constraint set 4 (worktime behind maximum)

```
forall (t in T)
       end_time[t] <= workTime;
```

2.4.5 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;
```

2.4.6 Constraint set 6 (defines the order of the visits)

If truck t visits any point it also visits the Source/Destination point, avoids travelling 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].

3.1 Greedy Randomized Adaptive Search Procedure (GRASP)

3.1.1 Constructive phase pseudo-code

```
procedure construct(quality(·), alpha)
	initialize empty solution x
	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.

[2] All the source code of the solvers is available in the GRASP directory.

3.2 Biased Random-Key Genetic Algorithm (BRKGA)

3.2.1 Chromosome structure

For each individual there is a set of genes.
Each of this genes has a value between 0 and 1 and it is associated to an assignment between a point and a vehicle.

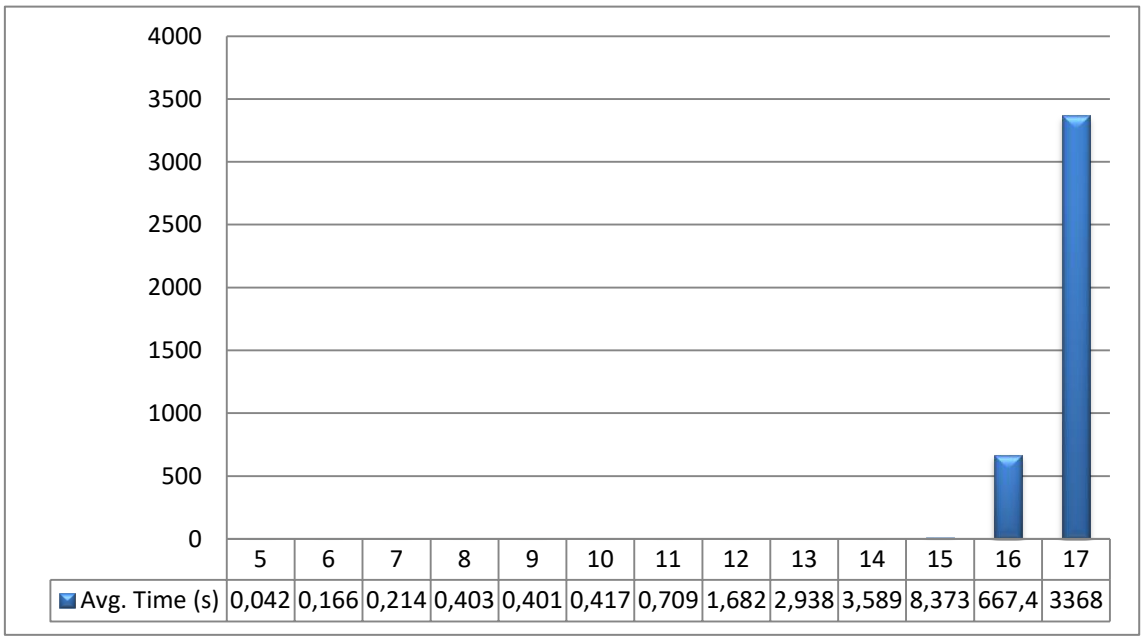
3.2.2 Decoder algorithm pseudo-code

```
procedure decode(quality(·), chromosome)
        initialize empty solution x
        initialize candidate set C
        while C is not empty do:
                CS = { quality(s) * chromosome(s) ∀ s ∈ C }
                s = min(CS)
                add s to x
                update candidate set C
        end while
        return x
end decode
```

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], with 17 destination points.



	5	6	7	8	9	10	11	12	13	14	15	16	17
Avg. Time (s)	0,042	0,166	0,214	0,403	0,401	0,417	0,709	1,682	2,938	3,589	8,373	667,4	3368

Figure 1: Average execution time in CPLEX

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.

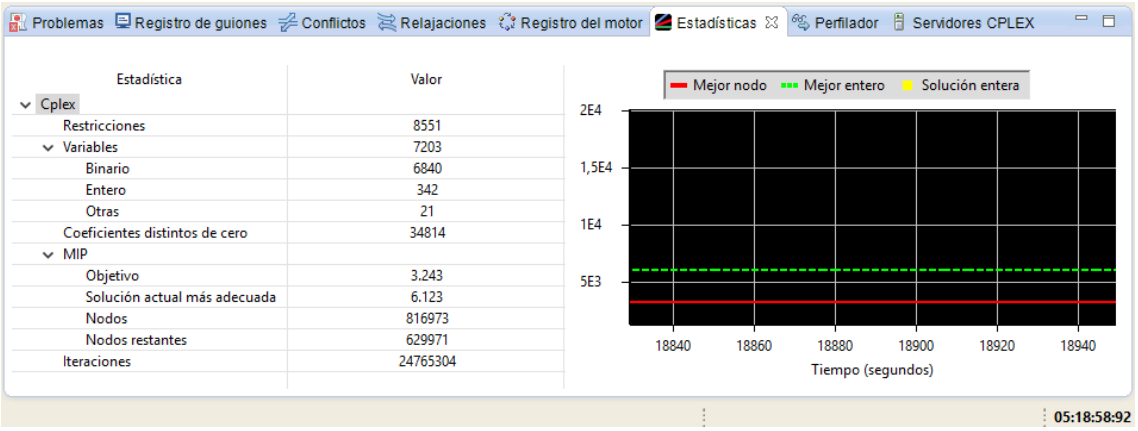


Figure 2: CPLEX screenshot for instance_18.dat after 5h+ running

[3] The path of instance 17 is InstanceGen/instance_17.dat, like many other instances created automatically with my Python instance generator (InstanceGen/generator.py)

4.2 Objective function value comparison

4.2.1 CPLEX optimal vs GRASP-BI, GRASP- FI & BRKGA in executions of 30 seconds each

With CPLEX we can get an optimal solution for the problem, but when the size increases it turns unfeasible in an acceptable period of time.
In the following figure it is clear than heuristics are critical to obtain solutions for complex problems when the size is too big to find the optimal one.

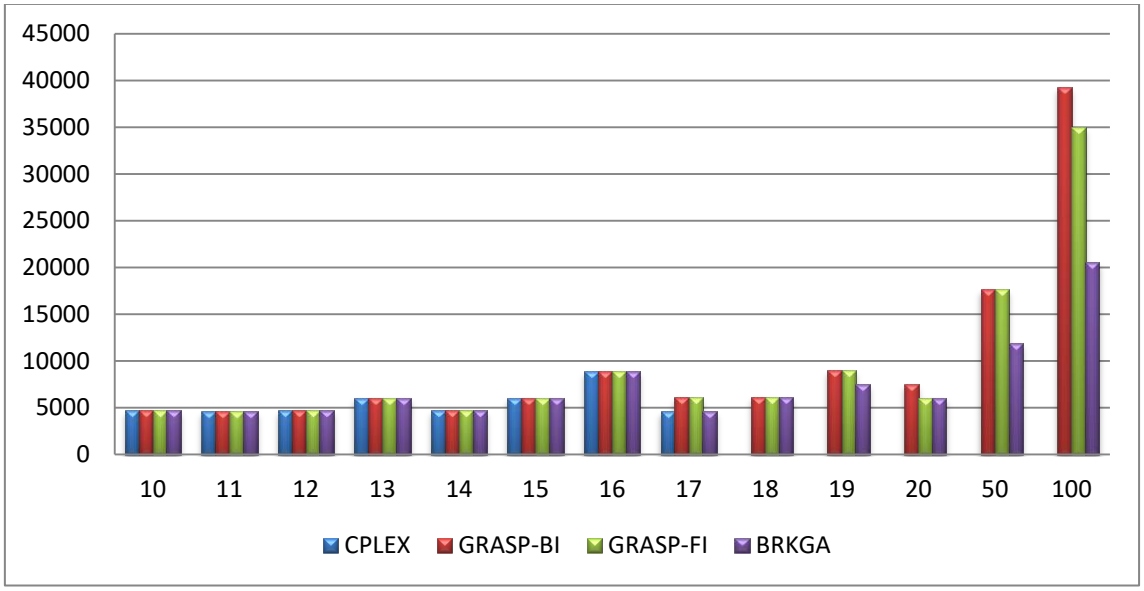


Figure 3: Objective function value evolution in executions of 30 seconds each

The GRASP and BRKGA implementations find quickly good solutions, for small instances such as smaller than instance 16 (that are solved in CPLEX in less than 10 seconds) they arrive to optimal solutions.

Instance 17 (with 18 total points) is where the first differences arise: Both Best Improvement and First Improvement variations of GRASP get similar values while BRKGA arrives until a solution with the same objective value of the CPLEX optimal.

After that, in every single instance BRKGA performs equal or better than GRASP, an effect that looks like is more and more evident with the increase of the number of destination points. In the biggest instance shown here, GRASP with Best Improvement found a solution that needs of 27 vehicles while GRASP with First Improvement arrived to a different one that needs 24. Meanwhile BRKGA only needs 14 vehicles to visit the 100 destination points of this instance, which makes this algorithm the best option to solve the problem.

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

The behaviour of the different heuristics is pretty similar than in the previous experiment. This time it looks like GRASP-BI found its way into a better solution for Instance 100 while GRASP-FI found a similar one. Also, with this increment in the execution time BRKGA improved from a solution for instance 50 which had an objective function value of 11890 to another with 10482. This reduces the number of vehicles needed from 8 to 7.

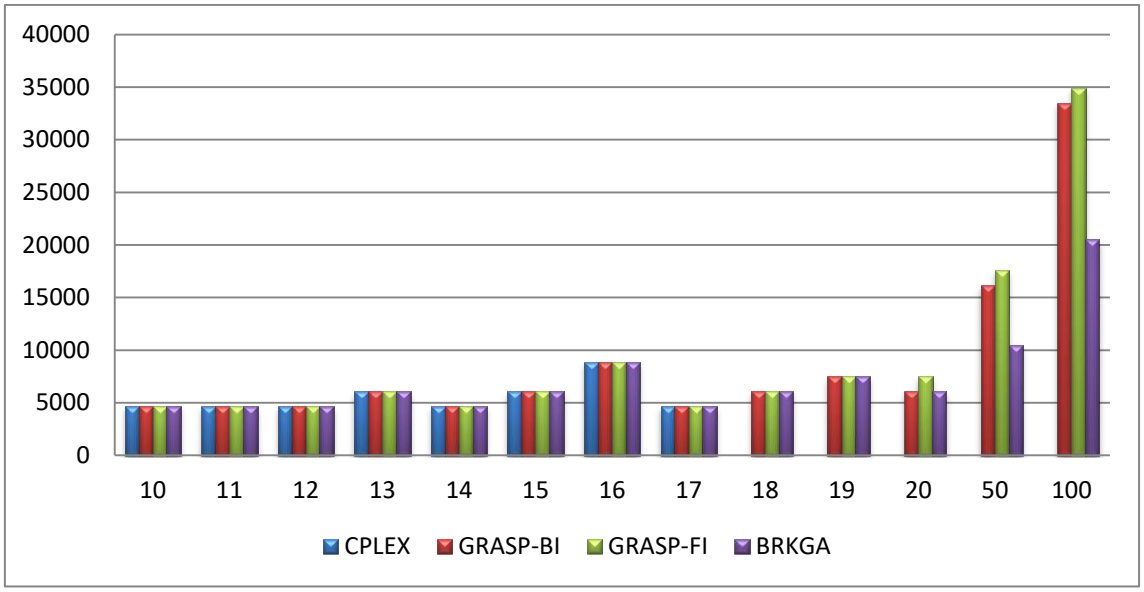


Figure 3: Objective function value evolution in executions of 30 seconds each

There is an obvious limitation that is the impossibility of creating a serialized route, because the instance generator distributes the time windows pseudo-randomly among the central part of the worktime period, trying to prevent the generation of unfeasible instances. The impact of this overlap is obvious here, because even with much more time there are cases where there is not a big improvement, and obviously improving turns harder after a while.

The conclusion of these executions could be that there is enough evidence to say that BRKGA is the best option for this problem.

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. Heuristics are a great tool to approach complex

Annex I. Instructions

1. Instance generation

There is a Python generator in InstanceGen/generator.py, it creates instances that are compatible with CPLEX and GRASP/BRKGA solvers.

Script usage:

```
generator.py N
```

N: Positive integer, represents the number of destination points to create

Output: instance_N.dat file in the same directory.

2. OPL model execution

The model is located at Project/Project.mod, it can be manually executed in CPLEX adding any of the instances previously generated or new ones. Also, there is a script in Project/autorun.mod that can be used to execute different instances many times to extract an average.

3. GRASP/BRKGA execution

The manual way consists in executing GRASP/main.py with the following arguments:

1st: Path to instance (ex. '../InstanceGen/instance_6.dat')

2nd: Solver selected (GRASP-BI | GRASP-FI | BRKGA)

3rd: Target execution time (in seconds)

Example: `main.py ../InstanceGen/instance_6.dat BRKGA 30`

4. Batch execution script

Alternative to manual execution for GRASP and BRKGA.

It is in: SolutionGen/solutionGen.py

Script usage:

```
solutionGen.py batchinputfile > outputfile
```

The input files must be the relative path to the main.py script and its arguments. Check SolutionGen/input/*.txt files to see some examples of the expected content of the input files for this script.

5. Result extraction script

Makes easier to copy & paste results to the Excel spreadsheet.

It is in: SolutionGen/solutionExt.py

Script usage:

```
solutionExt.py outputfile
```

It shows the number of vehicles (NT) needed in every solution and also the objective function value (Q).

6. Graph generation

The Excel file Project/graph.xlsx is a spreadsheet used to generate the different figures and also store and compare the results obtained.