AMMM Final Project

Wilmer Uruchi Ticona

January 9, 2019

Contents

1 Problem Statement

A bus company has a set S of bus services to operate. For that purpose it can use a set D of drivers and a set B of buses. The goal is to assign one bus and one driver to each service, while satisfying some constraints. As expected, a driver (and also a bus) can operate multiple services.

For each bus service we know its starting time, its duration in minutes and kilometers, and the number of passengers to be transported in the service. Each bus service starts in the headquarters of the company and finishes there as well. For each bus in $b \in B$ we know its capacity cap_b (i.e., number of passengers it can transport), and the cost in euros per minute ($euros\_min_b$) and per kilometer ($euros\_km_b$) for using that bus. Finally, for each driver $d \in D$ we know the maximum number of minutes max_d she can work.

We need to help the bus company to decide which bus and driver is assigned to each service. However, not any assignment is valid, since services should be operated by buses with enough capacity; the same bus or driver cannot serve two services that overlap in time; and we should respect the maximum number of working minutes for each driver (e.g. if a driver can work at most 6 minutes, it cannot operate 3 services with durations 4, 2 and 1 minutes). Additionally, we can use at most $maxBuses$ buses.

Among all possible solutions, we want the one with minimum cost. Apart from the cost of using the buses, the company pays each driver CBM euros for the first BM minutes she works, and pays CEM euros for the remaining minutes (if any). For example, if $BM = 200$, $CBM = 0.5$ and $CEM = 0.8$ and a driver works 300 minutes, she will be paid $200 \cdot 0.5 + 100 \cdot 0.8$ euros. If she works 150 minutes, she will be paid $150 \cdot 0.5$ euros. We can assume that always $CEM > CBM$.

Input data:

S = set of bus services s
D = set of drivers d
B = set of buses b
$start[s]$ = starting time of bus service s
$dmin[s]$ = duration in minutes of bus service s
$dkm[s]$ = duration in km of bus service s
$passengers[s]$ = number of passengers to be transported by bus service s
$capacity[b]$ = capacity of bus b
$e\_min[b]$ = cost per minute for using bus b
$e\_km[b]$ = cost per kilometer for using bus b
$max[d]$ = maximum number of minutes driver d can work
$maxBuses$ = maximum number of buses B we can use
CBM = amount paid for the first BM minutes a driver works
CEM = amount paid for the remaining minutes worked by a driver

BM = Base minutes for every driver

$overlap[s_1][s_2] \in \mathbb{B}$ = true iff service s_1 overlaps service s_2, this is a result of pre processing based on $start_s$ and $dmin_s$ over all services.

2 Integer Linear Model

Decision variables:

$service_s \in \mathbb{B}$ = true iff bus service s is covered.
$bs_{b,s} \in \mathbb{B}$ = true iff bus b is assigned to bus service s.
$ds_{d,s} \in \mathbb{B}$ = true iff driver d is assigned to bus service s.
$dbase_d \in \mathbb{Z}^+$ = duration of driver's work for the first BM minutes or less.
$dextra_d \in \mathbb{Z}^+$ = duration of extra driver's work if $dbase_d$ is greater than BM.
$bus_b \in \mathbb{B}$ = true iff bus b takes part in a service s

Objective function:

Minimize

$$\sum_{b\in B}\sum_{s\in S} bs_{b,s}\cdot dmin[s]\cdot e\_min[b] + \sum_{b\in B}\sum_{s\in S} bs_{b,s}\cdot dkm[s]\cdot e\_km[b] + \sum_{d\in D} dbase_d\cdot CBM + \sum_{d\in D} dextra_d\cdot CEM$$

The objective function aims to minimize the total cost of operating the system. For this purpose the cost is divided in 4 parts, from left to right: the total cost of the buses assigned to the services based on time, the total cost of the buses assigned to the services based on distance, the total cost of the drivers assigned to the services for the first BM, and the total cost of the drivers assigned to the services for the minutes past BM.

Constraints:

$$bs_{b,s} = 0 \quad \forall\ b \in B, \forall s \in S : passengers[s] > capacity[b]$$

A bus b cannot take part in service s if the quantity of passengers required by s is greater than the capacity of bus b.

$$\sum_{b\in B} bs_{b,s} = 1 \quad \forall s \in S$$

All services s must have exactly 1 bus.

$$bus_b \cdot nServices \geq \sum_{s\in S} bs_{b,s} \quad \forall\ b \in B$$

$$\sum_{b \in B} bus_b \leq maxBuses$$

For all buses b, bus_b will be 1 only if this bus is included in one of the services. Also, the sum of bus_b for all the solution cannot exceed $maxBuses$, meaning that no more than $maxBuses$ can work in the system.

$$\sum_{d \in D} ds_{d,s} = 1 \quad \forall\ s \in S$$

All services s must have exactly 1 driver.

$$\sum_{s \in S} ds_{d,s} \cdot dmin[s] \leq max[d] \quad \forall\ d \in D$$

A driver d can work up to $max[d]$ minutes.

$$ds_{d,s_1} + ds_{d,s_2} \leq 1 \quad \forall\ d \in D, \forall\ s_1, s_2 \in S : overlap[s_1][s_2] = 1, \forall\ s_1 < s_2$$

A driver cannot work in two services that overlap.

$$bs_{b,s_1} + bs_{b,s_2} \leq 1 \quad \forall\ b \in B, \forall\ s_1, s_2 \in S : overlap[s_1][s_2] = 1, \forall\ s_1 < s_2$$

A bus cannot work in two services that overlap.

$$dbase_d + dextra_d = \sum_{s \in S} ds_{d,s} \cdot dmin[s] \quad \forall\ d \in D$$

$$dbase_d \leq BM \quad \forall\ d \in D$$

The total cost is divided in two decision variables $dbase_d$ and $dextra_d$. Since $CEM > CBM$, the objective function will try to minimize the time in $dextra_d$, because it implies a higher cost for the objective function.

3 Heuristics: Greedy + Local Search

3.1 Greedy

Algorithm 1 greedy(problem)

Ensure: A low cost distribution of drivers and buses is chosen

$solution \leftarrow$ create empty solution
$sortedBuses \leftarrow$ sorted buses in ascending order according to their cost in minutes and kilometers
$sortedDrivers \leftarrow$ sorted drivers in ascending order according to their current working minutes
$sortedServices \leftarrow$ sorted services in descending order according to their duration in minutes and kilometers
while length of $sortedServices > 0$ **do**
  $service \leftarrow sortedServices[0]$
  for $driver$ **in** $sortedDrivers$ **do**
    $currentAssignments \leftarrow$ get current Assignments for $driver$
    if $serivce$ does not overlap with $currentAssignments$ **and** maxMinutes of $driver$ are not exceeded **then**
      $solution \leftarrow$ save new assignment for $service$ with $driver$
      $sortedServices.pop(0)$
      $sortedDrivers \leftarrow$ sort again including new changes
    end if
  end for
end while
$maxBuses \leftarrow$ gets the maximum quantity of buses allowed from sortedBuses in order
while length of $sortedServices > 0$ **do**
  $service \leftarrow sortedServices[0]$
  for $bus \in maxBuses$ **do**
    $currentAssignments \leftarrow$ get current Assignments for bus
    if $serivce$ does not overlap with $currentAssignments$ **and** capacity of bus is not exceeded **then**
      $solution \leftarrow$ save new assignment for $service$ with bus
      $sortedServices.pop(0)$
      $maxBuses \leftarrow$ sort again including new changes
    end if
  end for
end while
return $solution$

3.2 Local Search

Algorithm 2 localSearch(solution, alpha)

Ensure: A low cost distribution of drivers and buses is chosen after iterations
  $bestNeighbor \leftarrow solution$
  $currentBestValue \leftarrow solution$ objective value
  $nRCLdrivers \leftarrow$ **length**(drivers) * alpha
  $nRCLbuses \leftarrow$ **length**(buses) * alpha
  $driversSolution \leftarrow$ sorted $nRCLdrivers$ drivers from $bestNeighbor$ in descending order
  $busesSolution \leftarrow$ sorted $nRCLbuses$ buses from $bestNeighbor$ in descending order
  $driversSolutionASC \leftarrow$ sorted $nRCLdrivers$ drivers from $bestNeighbor$ in ascending order
  $busesSolutionASC \leftarrow$ sorted $nRCLbuses$ buses from $bestNeighbor$ in ascending order
  $num \leftarrow$ number of services
  for $i = 0$ **to** num **do**
    If bestNeighbor.AssignDrivers[i] **not in** $driversSolution$: **continue**
    for $j = num - 1$ **to** 0 **do**
      If bestNeighbor.AssignDrivers[j] **not in** $driversSolutionASC$: **continue**
      $solutionNeighbor \leftarrow$ new solution based on $solution$
      $service1 \leftarrow$ service from Assigned Services of $solutionNeighbor$ with index i
      $service2 \leftarrow$ service from Assigned Services of $solutionNeighbor$ with index j
      if $service1$ and $service2$ can exchange $drivers$ **then**
        Exchange $drivers$
        $newValue \leftarrow solutionNeighbor$ objective value
      end if
      if $currentBestValue > newValue$ **then**
        $currentBestValue \leftarrow newValue$
        $bestNeighbor \leftarrow solutionNeighbor$
      end if
    end for
  end for
  for $i = 0$ **to** num **do**
    If bestNeighbor.AssignBuses[i] **not in** $busesSolution$: **continue**
    for $j = num - 1$ **to** 0 **do**
      If bestNeighbor.AssignBuses[j] **not in** $busesSolutionASC$: **continue**
      $solutionNeighbor \leftarrow$ new solution based on $solution$
      $service1 \leftarrow$ service from Assigned Services of $solutionNeighbor$ with index i
      $service2 \leftarrow$ service from Assigned Services of $solutionNeighbor$ with index j
      if $service1$ and $service2$ can exchange $buses$ **then**
        Exchange $buses$
        $newValue \leftarrow solutionNeighbor$ objective value
      end if
      if $currentBestValue > newValue$ **then**
        $currentBestValue \leftarrow newValue$
        $bestNeighbor \leftarrow solutionNeighbor$
      end if
    end for
  end for
  return $bestNeighbor$

3.3 Restricted Candidate List in Local Search

For problems with a high quantity of services (400+), we decided to restrict the quantity of **Exchanges** in local search. For this purpose we use the value *alpha* to calculate a number of possible exchanges, for example, a value of 1 means that the whole neighborhood is going to be explored, and a value of 0.1 means that only 10% of the neighborhood is going to be explored.

We can find this restriction in the algorithm for Local Search where the variables $driversSolution$, $busesSolution$, $driversSolutionASC$, $busesSolutionASC$ are restricted to a number of items equal to the product of *alpha* and the total number of drivers and of buses, and we obtain $nRCLdrivers$, and $nRCLbuses$ respectively. Then, we use these numbers to obtain the drivers and buses sorted accordingly:

- $busesSolution \leftarrow$ first $nRCLbuses$ number of buses from the sorted list of buses of the problem in descending order according to the cost of operation per minute and kilometer.
- $busesSolutionASC \leftarrow$ first $nRCLbuses$ number of buses from the sorted list of buses of the problem in ascending order according to the cost of operation per minute and kilometer.
- $driversSolution \leftarrow$ first $nRCLdrivers$ number of drivers from the sorted list of drivers of the problem in descending order according to the current number of minutes assigned to the driver.
- $driversSolutionASC \leftarrow$ first $nRCLdrivers$ number of drivers from the sorted list of drivers of the problem in ascending order according to the current number of minutes assigned to the driver.

Then, these lists act as a filter in the exchange iterations. The objective is to try to exchange between $drivers$ with a high number of minutes assigned, and $drivers$ with a low number of minutes assigned, so the local search balances the number of assigned minutes per $drivers$ **closer to the mean** while trying to avoid exploring the whole local neighborhood, which in big scenarios takes too much time and computational power.

The approach for buses is similar. In this case, the random construction algorithm has made sure that the buses with lower cost of operation are selected for the initial solution, then, the local search tries to exchange these buses to find a better objective value. Although, an algorithm that tries to assign the lower cost buses to as many services as possible would have been better, but that idea did not translate well to the python code. However, we think it is worth exploring in other similar problems.

This same concept is applied for GRASP, as we use the same algorithm for Local Search for the Meta-heuristics experiment.

3.4 Greedy + Local Search

Algorithm 3 runLocalSearch(solution, alpha)

Ensure: A low objective value solution is found

If *solution* **is** Infeasible : **return Empty**
$bestSolution \leftarrow solution$
$bestValue \leftarrow$ objective value of *solution*
$services \leftarrow$ list of services from problem
$keepIterating \leftarrow True$
while $keepIterating$ **is** $True$ **do**
  $keepIterating \leftarrow False$
  $neighbor \leftarrow localSearch(solution, alpha)$
  $currentValue \leftarrow$ objective value of *neighbor*
  if $bestValue > currentValue$ **then**
    $bestSolution \leftarrow neighbor$
    $bestValue \leftarrow currentValue$
    $keepIterating \leftarrow True$
  end if
end while
return $bestSolution$

4 Meta-heuristics: GRASP

4.1 Random Construction

Algorithm 4 constructionRandom(problem)

Ensure: A random cost distribution of drivers and buses is chosen
  $solution \leftarrow$ create empty solution with data from problem
  $buses \leftarrow$ list of buses from problem
  $drivers \leftarrow$ list of drivers from problem
  $services \leftarrow$ list of services from problem
  while length of $services > 0$ **do**
    $service \leftarrow$ random $service$ from $services$
    for $driver$ **in** $drivers$ **do**
      $currentAssignments \leftarrow$ get current Assignments for $driver$
      if $serivce$ does not overlap with $currentAssignments$ **and** maxMinutes of $driver$ are not exceeded **then**
        $solution \leftarrow$ save new assignment for $service$ with $driver$
        remove $service$ from $services$
        $drivers \leftarrow$ sort in ascending order by current assigned minutes
      end if
    end for
  end while
  $services \leftarrow$ list of services from problem
  while length of $services > 0$ **do**
    $service \leftarrow$ random from $services$
    for bus **in** $buses$ (up to a set max quantity of buses) **do**
      $currentAssignments \leftarrow$ get current Assignments for bus
      if $serivce$ does not overlap with $currentAssignments$ **and** capacity of bus is not exceeded **then**
        $solution \leftarrow$ save new assignment for $service$ with bus
        remove $service$ from $services$
      end if
    end for
  end while
  return $solution$

4.2 GRASP

Algorithm 5 GraspSolver(maxTime, alpha)

Ensure: A low objective value solution is found
  If *solution* **is** Infeasible : **return Empty**
  $bestSolution \leftarrow$ Empty
  $bestValue \leftarrow$ Infinite
  $maxExecutionTime \leftarrow maxTime$ (maximum time allowed to iterate)
  while $maxExecutionTime$ **is not exceeded do**
    $solutionGrasp \leftarrow constructionRandom()$
    If *solutionGrasp* **is** Infeasible : **continue**
    $solutionGrasp \leftarrow runLocalSearch(solutionGrasp, alpha)$
    $solutionBestValue \leftarrow$ objective value of $solutionGrasp$
    if $bestValue > solutionBestValue$ **then**
      $bestSolution \leftarrow solutionGrasp$
      $bestValue \leftarrow solutionBestValue$
    end if
  end while
  return $bestSolution$

5 Greedy Cost Function

$d = drivers$
$b = buses$
$s = services$
$costB$ = cost of operation for the buses assigned to the services
$costD$ = cost of operation for the drivers assigned to the services
$overlaps[s1[s2]$ = 1 iff service $s1$ and $s2$ overlap
$bs_{b,s}$ = true iff bus b is assigned to bus service s.
$ds_{d,s}$ = true iff driver d is assigned to bus service s.
$dmin[s]$ = duration in minutes of bus service s
$max[d]$ = maximum number of minutes driver d can work
$Assign_b$ = set of services to which bus b has been assigned
$Assign_d$ = set of drivers to which driver d has been assigned
$maxBuses$ = maximum number of buses allowed in the system
bus_b = 1 iff bus b is included in the system
$q(s, d, b) = min\{q(s, d, b)\}$

$val()$ = gives the objective value of using a driver d or bus b

$$q(s,d,b) = \begin{cases} \emptyset & \text{if max capacity of } b < \text{max passengers of } s \\ \emptyset & \sum_{b\in B} bs_{b,s} < 1 \text{(a service does not have a bus)} \\ \emptyset & \sum_{d\in D} ds_{d,s} < 1 \text{(a service does not have a driver)} \\ \emptyset & \sum_{s\in S} ds_{d,s} \cdot dmin[s] \geq max[d] \text{ (a driver exceeds his max time)} \\ \emptyset & \sum_{b\in B} bus_b > maxBuses \text{ (maximum number of buses exceeded)} \\ \infty & val(b1) = val(b2)\ \forall b1 < b2 \in B \text{ and } val(d1) = val(d2)\ \forall d1 < d2 \in D \\ costB + costD & s1 \neq s2\ \forall s1, s2 \in Assign_d\ ,\ s1 \neq s2\ \forall s1, s2 \in Assign_b \end{cases}$$

6 Comparative Results

		ILOG CPLEX		Greedy + Local		GRASP	
File	Services	Time (s)	Result	Time (s)	Result	Time (s)	Result
1	10	0.34	1035	2.73 ($\alpha = 1$)	1936.75	1802.82 ($\alpha = 1$)	1707.5
2	50	0.54	410.66	42.67 ($\alpha = 1$)	1148.35	1858.43 ($\alpha = 1$)	1019.25
3	50	0.51	519.43	45.48 ($\alpha = 1$)	1336.41	1820.14 ($\alpha = 1$)	1091.44
4	100	4.16	2741.3	234 ($\alpha = 1$)	6886.25	2931.7 ($\alpha = 1$)	5692.75
5	100	7.97	3036.4	133.73 ($\alpha = 1$)	6578.48	2896.55 ($\alpha = 1$)	5615.36
6	150	15.75	10793	66.94 ($\alpha = 0.1$)	24631.29	1914.48 ($\alpha = 0.1$)	20968.61
7	395	1278.67	15363	221 ($\alpha = 0.02$)	42175.81	1997 ($\alpha = 0.02$)	39415.55
8	400	4186.71 (*)	30327.21	119.28 ($\alpha = 0.04$)	68402.28	2190.9 ($\alpha = 0.04$)	61016.01
9	425	3679.46 (**)	53163.39	85.98 ($\alpha = 0.04$)	106368.91	2206.41 ($\alpha = 0.04$)	92182.68

(*) Stopped at Gap 0.03%

(**) Stopped at Gap 0.02%

File	Name
1	examen.dat
2	50serv_instances_1.dat
3	50serv_instances_0.dat
4	100serv_instances_0.dat
5	100serv_instances_2.dat
6	150s_instances_0.dat
7	example_0.dat
8	400s_instances_0.dat
9	425s_instances_0.dat

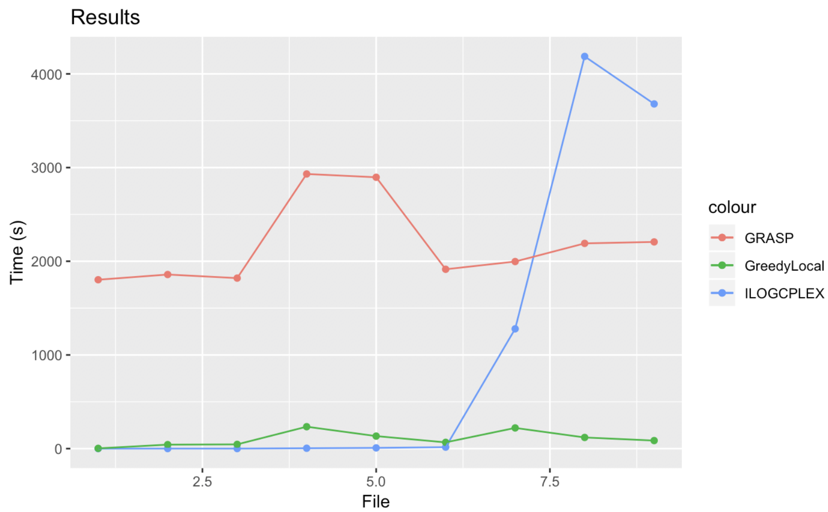

Figure 1: Plot of Experiments (File) x Time

As the problems get more complex, the ILOG CPLEX required time becomes the highest. GRASP time is constant because we are setting the maximum running time to 1800 seconds for each experiment. Greedy time stays low thanks to RCL manipulation.

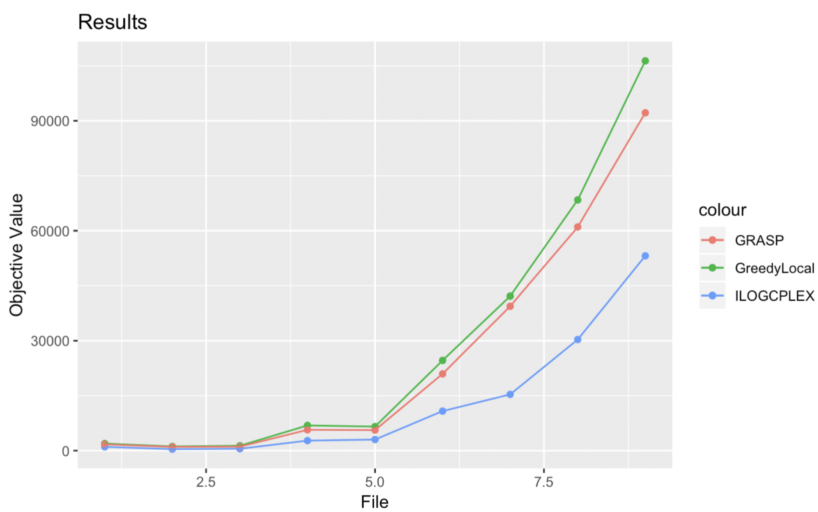

Figure 2: Plot of Experiments (File) x Objective Value

The Greedy solution objective value is the highest, which is expected. GRASP result is lower than Greedy and as the problems get more complex, the improvement ratio seems to increase. ILOG CPLEX results stay the lowest; however, in the case of the last 2 experiments, we stopped the program some minutes past 1 hour, so the final result might have been lower but the required time would have increased by a large amount.

7 Experiment Instructions

We have included a file called Main.py in the archive *PythonHeuristics.zip* that contains all the necessary code perform the experiments. To execute a Greedy + Local Search, use the following code:

```
experiment = Experiment('name_of_instance.dat')
experiment.GreedyPlusLocal(alpha)
```

To execute GRASP, use the following code:

```
experiment = Experiment('name_of_instance.dat')
experiment.GRASP(maxTime, alpha)
```

Where *alpha* is a floating point number between 0 and 1, that restricts the list of candidates. *maxTime* is the maximum running time for GRASP in seconds.

8 Included Documents

You will find included the following documents:

- examen.dat
- 50serv_instances_1.dat
- 50serv_instances_0.dat
- 100serv_instances_0.dat
- 100serv_instances_2.dat
- 150s_instances_0.dat
- example_0.dat
- 400s_instances_0.dat
- 425s_instances_0.dat
- instanceGenerator.rar
- AMMM_Project.mod

- AMMM_Project.dat
- PythonHeuristics.zip