AMMM Project Report

Martin Galajda, Magali Bonzom

June 13, 2018

1 Introduction

1.1 Problem Statement

A logistic company needs to place a set of packages inside a set of trucks. All trucks have the same size width & length, with a maximum capacity. There is a limited number of packages, which have varied dimensions, that can be placed into a truck. The main objective is to assign all packages to some truck and to minimize the load of the truck with the highest load and number of trucks used, in order to achieve the smallest possible amount of transportation cost. This problem is an example of optimization problem which can be solved by integer linear programming techniques.

2 Integer linear program

In order to explain the model and formal solution, we first define the set of parameters and decision variables that have been used in our Integer Linear Model.

2.1 Input parameters, decision variables

Parameter	**Explanation**
int pLength	number of packages
int tLength	number of trucks
int xTruck	width of truck
int yTruck	length of truck
int capacityTruck	capacity of truck, which is the same for all trucks. The data are received from the data file.

Parameter	**Explanation**
range T = 1..tLength;	Is the set of all truck identifiers ranging from 1 to tLength
range P = 1..pLength;	Is the set of all package identifiers ranging from 1 to pLength
range X = 1..xTruck;	Is the set of all values for x coordinates in truck, going from 1 to xTruck
range Y=1..yTruck;	Is the set of all values for y coordinates in truck, going from 1 to yTruck

Parameter	Explanation
int package_x[p in P]	Set of the lengths of packages
int package_y[p in P]	Set of the widths of packages
int packageWeight[p in P]	Set of the weights of a packages
int incomp[1..pLength][1..pLength]	Matrix with two dimensions, representing relation between two packages. If value in matrix equals to 1, then two packages cannot be placed in the same truck
int truckPkgPositionMap [t in T][x in X][y in Y]	Used for drawing solution to console

Decision Variables	Explanation
dvar boolean pt [p in P][t in T];	Package p is placed in truck t, consisting of two dimensions: the identifier of the package and of the truck
dvar boolean pxy [p in P][1..xTruck][1..yTruck];	Package p placed in the cell with coordinates (x,y)
dvar boolean pbl [p in P][1..xTruck][1..yTruck];	the bottom left cell of package p has coordinates (x,y)
rdvar boolean usedTruck [t in T];	Truck T is used
dvar float+ z;	Variable representing objective function value

2.2 Objective Function

The goal of this assignment is to minimize the transportation cost. Using the defined parameters and decision variables it is possible to define the multi-objective function, by multiplying the capacity of the truck with the number of the used trucks, and adding the load of the truck with the highest load:

$$z = numberOfUsedTrucks * capacityTruck + highestLoadedTruck$$

We obtain an objective function by setting a constraint, that makes sure that z is greater or equal than the used capacity of all trucks plus the load of the highest loaded truck:

$$capacityTruck * \sum_{t \in T} usedTruck_t + highestLoadedTruck \leq z$$

By trying to minimize z, the integer linear program converges to an optimal solution.

2.3 Constraints

To obtain a feasible solution to our problem, we need to respect the following 10 constraints:

Constraint 1

$$\forall t \in T : \sum_{p \in P} pt_{p,t} * packageWeight_p \leq capacityTruck$$

- The constraint defines that every truck carries at most *capacityTruck* kg of package weight.

Constraint 2

$$\forall p \in P : \sum_{t \in T} pt_{p,t} = 1$$

- Every package is placed into exactly 1 truck.
- We sum over all the trucks and make sure it is set to 1.

Constraint 3

$$\forall p_1 \in P, \forall p_2 \in P : p_1 \neq p_2, \forall t \in T, \forall x \in X, \forall y \in Y : pt_{p_1,t} + pt_{p_2,t} + pxy_{p_1,x,y} + pxy_{p_2,x,y} \leq 3$$

- Two packages that are in the same truck cannot overlap.
- The equation expresses the addition of placing the first package in a truck, a second package in the same truck, the specific x,y location of package 1 in truck t, and the specific x,y location of package 2 in truck t.
- To be sure that the two packages do not overlap, at least one of the boolean need to be false, so it is less or equal to 3 (they are either in different trucks or do not overlap in the same truck).

Constraint 4

$$\forall p \in P, \forall x \in X, \forall y \in Y : pbl_{p,x,y} * (x + packageWidth_p - 1) \leq truckWidth$$

- The constraint ascertains that the packages do not go out of the truck.
- The constraint makes sure that the package does not move outside the x-direction, the width of the truck.
- This is done by multiplying the starting point/position of the package by the width of the truck.

Constraint 5

$$\forall p \in P, \forall x \in X, \forall y \in Y : pbl_{p,x,y} * (y - packageHeight_p) \geq 0$$

- The constraint ascertains that the packages do not go out of the truck.
- The constraint makes sure that the package does not move outside the y-direction, the height of the truck.

Constraint 6

$$\forall t \in T : \sum_{p \in P} pt_{p,t} * packageWeight_p \leq highestLoadedTruck$$

- Make the *highestLoadedTruck* define the highest loaded truck.
- The package in the truck is multiplied by the weight of the package.
- The load of each truck is set to 1 if a pack is in the truck.

- In the objective function, the *highestLoadedTruck* is set to minimize the value.

Constraint 7

$$\forall p_1, p_2 \in P : p_1 \neq p_2 : \forall t \in T : incomp_{p_1,p_2} * (pt_{p_1,t} + pt_{p_2,t}) \leq incomp_{p_1,p_2}$$

- Packages cannot go into the same truck if *incomp[pOne,pTwo]* is true.
- If a package is not compatible in a truck then it equals to 1, hence the code will be as followed: 1 * 1 + 1, which will be larger than 1. The packages cannot be placed in the same truck.
- If the packages are compatible, it will equal to 0. Hence, we can place the packages as we want.

Constraint 8

$$\forall p \in P : \sum_{x \in X} \sum_{y \in Y} pbl_{p,x,y} = 1$$

- Every package has exactly one bottom left point inside the truck.

Constraint 9

$$\forall p \in P, \forall x_1 \in X, \forall y_1 \in Y :$$

$$(1 - pxy_{p,x_1,y_1}) + \sum_{x_2 = x_1 - packageWidth_p - 1}^{x_1} \sum_{y_2 = y_1}^{y_1 + (packageHeight_p - 1)} pbl_{p,x_2,y_2} = 1$$

- This constraint makes sure that the location of a package is inside the truck, and not outside the width and height range of the package, when *pbl* is set accordingly.
- Makes the connection between *pbl* and *pxy*.

Constraint 10

$$\forall t \in T : \sum_{p \in P} pt_{p,t} \leq usedTruck_t * capacityTruck$$

- The final constraint makes sure that *usedTruck* is set to 1 when a truck is used.

3 Heuristics

To find an optimal solution, the use of Integer Linear Programming guarantees that we will find an optimum (if it exists). However, it is time consuming for medium to big size problems. Therefore, to receive a faster solution we are using heuristics to solve this combinatorial optimization problem. The solution achieved by heuristics is not guaranteed to be optimal, but the solving time is usually much better for bigger problems. For this problem, we are using the GRASP metaheuristic.

3.1 Greedy cost function

The Greedy cost function is defined for every element in the candidate set which represents a pair consisting of package and truck (assigning a package to a given truck).

$$gcf = -400ast - 30poss_{pbl} + 40poss_t - 15s - 3w + 20c$$

where:

ast	= represents a binary variable signaling whether a truck is already assigned
$poss_{pbl}$	= represents number of possibilities to assign pbl to a given truck
$poss_t$	= represent the number of trucks to which a package can still be assigned to
s	= represents the size of the package (width * height)
w	= represents the weight of the package
c	= represents the capacity left in the truck in the current solution

3.2 Restricted candidate set

For randomly selecting new candidate elements in our constructing algorithm we have used the following restricted candidate set:

$$RCL = \{(p,t) \in C | gcf((p,t)) <= gcf_{min} + alpha * (gcf_{max} - gcf_{min})\}$$

where

C	= candidate set
$alpha$	= parameter set by our heuristics
gcf_{max}	= biggest greedy cost of elements in candidate set
gcf_{min}	= smallest greedy cost of elements in candidate set
(p,t)	= element from candidate set, represents assignment of package p to truck t

3.3 Heuristics pseudocode

Next, we provide a pseudocode for heuristics that has been implemented. We have the pseudocode just for the classic GRASP but it is worth noting that we actually also implemented a modified GRASP that is constrained by a maximum computational time (and not maximum iterations). This modification was used mainly for tuning paremeters for heuristics (we constrained GRASP to search for a solution for some maximum time, and then compared results for different parameters of greedy cost function and alpha).

Additionally, we provided a pseudocode for the Greedy cost function which is parameterized. We have implemented a mechanism which allowed us to tune those parameters by trying different experiments, as well different values for the parameters.

In Function 12 we can observe that we have implemented a mechanism which allows us to change the size of neighbourhood quite easily (just by changing the function parameter). However, the default heuristics only used default parameter with the value 1 for the neighbourhood size.

Also, in Function 11 we can observe that by simply using different parameters for the function we can change our heuristics - using only the best solution from the neighbourhood, or all that have a better cost than the feasible solution provided to the function.

Function 1 GRASP

```
Input: costFunction, alpha, gcf, maxIterations, input, gcfParams
Output: bestSolution, bestCostSolution
 1: $bestCostSolution \leftarrow -\infty$
 2: $bestSolution \leftarrow NIL$
 3: $k \leftarrow 0$
 4: while $k \leq maxIterations$ do
 5:     $k \leftarrow k + 1$
 6:     $feasibleSolution \leftarrow construct(gcf, alpha, input)$
 7:     $costOfNewSolution, bestSolutionInNeighbourhood \leftarrow$
            $localSearch(costFunction, input, feasibleSolution)$
 8:     if $costOfNewSolution < bestCostSolution$ then
 9:         $bestSolution \leftarrow bestSolutionInNeighbourhood$
10:         $bestCostSolution \leftarrow costOfNewSolution$
11: return bestSolution, bestCostSolution
```

Function 2 construct

```
Input: gcf, alpha, input, glfParams
Output: feasibleSolution
 1: $feasibleSolutionObtained \leftarrow false$
 2: while feasibleSolutionObtained == false do
 3:     $solution \leftarrow tryToConstructSol(gcf, alpha, input)$
 4:     $feasibleSolutionObtained \leftarrow$ every package in solution is assigned to some truck
 5: return solution
```

Function 3 tryToConstructSol

```
Input: gcf, alpha, input, gcfParams
Output: feasibleSolution
1: $solution \leftarrow initializeSolution(input)$
2: $candidateSet \leftarrow initializeCandidateSet(input)$
3: while candidateSet is not empty do
4:     $min, max, precomputedCost \leftarrow$
       $getElementCostRange(gcf, input, solution, candidateSet, gcfParams)$
5:     $candidateSolutions \leftarrow getCandidateSolutions(min, max, precomputedCost, alpha)$
6:     $newElementSolution \leftarrow$ *randomly select element from candidateSolutions*
7:     $addNewSolution(input, solution, newElementSolution)$
8:     $updateCandidateSet(input, solution, candidateSet, newElementSolution)$
9: return solution
```

Function 4 getElementCostRange

```
Input: gcf, input, solution, candidateSet, gcfParams
Output: min, max, greedyCostPackageTruck
1: Initialize greedyCostPkgTruck to 0 for every package and truck
2: $min \leftarrow \infty$
3: $max \leftarrow -\infty$
4: for possibleAssignment in candidateSet do
5:     $p, t \leftarrow possibleAssignment$
6:     $greedyCostPkgTruck_{p,t} \leftarrow gcf(input, solution, possibleAssignment, gcfParams)$
7:     if $greedyCostPkgTruck_{p,t} < min$ then
8:         $min \leftarrow greedyCostPkgTruck_{p,t}$
9:     if $greedyCostPkgTruck_{p,t} > max$ then
10:        $max \leftarrow greedyCostPkgTruck_{p,t}$
11: return min, max, greedyCostPkgTruck
```

Function 5 getCandidateSolutions

```
Input: min, max, greedyCostPackageTruck, alpha
Output: possibleAssignments
1: *margin* $\leftarrow$ *(max - min) * alpha*
2: *possibleAssignments* $\leftarrow$ Create empty list
3: for p, t in keys(greedyCostPackageTruck) do
4:     if $greedyCostPackageTruck_{p,t} \leq min + margin$ then
5:         $possibleAssignments \leftarrow possibleAssignments + (p, t)$
```

Function 6 addNewSolution

```
Input: input, solution, newElementSolution
Output: solution
 1: $p,\ t \leftarrow newElementSolution$
 2: $pbl_x, pbl_y \leftarrow solution["possiblePbl"]_{p,t}$
 3: $solution \leftarrow updatePxy(input, solution, p, (pbl_x, pbl_y))$
 4: $solution["pbl"]_{p,pbl_x,pbl_y} \leftarrow solution["possiblePbl"]_{p,t}$
 5: $solution["pt"]_{p,t} \leftarrow 1$
 6: $solution["usedTruck"]_t \leftarrow 1$
 7: $loadOfTruck \leftarrow computeLoadOfTruck(solution, t, input)$
 8: if $loadOfTruck > solution["highestLoadedTruck]$ then
 9:     $solution["highestLoadedTruck"] \leftarrow loadOfTruck$
10: return solution
```

Function 7 updatePxy

```
Input: input, solution, p, pbl
Output: solution
 1: $packageWidth \leftarrow input["packageWidth"]_p$
 2: $packageHeight \leftarrow input["packageHeight"]_p$
 3: $startOffsetX \leftarrow pbl_0$
 4: $startOffsetY \leftarrow pbl_1$
 5: for x in 0..packageWidth do
 6:     $currentX \leftarrow startOffsetX + x$
 7:     for y in 0..packageHeight do
 8:         $currentY \leftarrow startOffsetY - y$
 9:         $solution["pxy"]_{p,currentX,currentY} \leftarrow 1$
10: return solution
```

Function 8 updateCandidateSet

```
Input: input, solution, candidateSet, newElementSolution
Output: possiblePt
 1: $possiblePt \leftarrow candidateSet$
 2: $p, t \leftarrow newElementSolution$
 3: Assign 0 to $possiblePt$ for every truck == t
 4: $packagesToBeChecked \leftarrow \{pkg \in P | possiblePt_{pkg,t} == 1\}$
 5: for package in packagesToBeChecked do
 6:     $possiblePt_{package,t}, solution["possiblePbl"]_{package,t} \leftarrow$
        $canBeAssignedToTruck(input, solution, package, t)$
 7: return possiblePt
```

Function 9 canBeAssignedToTruck

```
Input: input, solution, package, t
Output: canBeAssigned, newPblForPackage
1: Check from the solution if the truck has enough capacity for the package
2: Check from the input if a truck and a package are incompatible
3: Check if there exists a point in the truck which can become the new bottom-left position for a package
4: if all checks are successfull then
5:     possiblePbl ← First point that can become bottom-left position for
       package, starting from bottom-left most corner and continuing upwards
       and to the right
6:     return true, possiblePbl
7: else
8:     return false, (-1, -1)
```

Function 10 localSearch

```
Input: costFunction, input, feasibleSolution
Output: alternativeSolutionSet, bestCostSeen, bestInNeighbourhood
1: $solutionSet, bestCostSeen, bestSolution$
       $\leftarrow constructAlternativeSolutionSet(costFunction, feasibleSolution)$
2: $bestCostSeen \leftarrow costFunction(input, feasibleSolution)$
3: $bestSolution \leftarrow feasibleSolution$
4: while solutionSet is not empty do
5:     $anotherFeasibleSol$ ← Select new solution from $solutionSet$
6:     $altSolutionSet, altBestCostSeen, altBestSolution \leftarrow$
           $constructAlternativeSolutionSet(costFunction, feasibleSolution)$
7:     $solutionSet \leftarrow solutionSet + altSolutionSet$
8:     if $altBestCostSeen < bestCostSeen$ then
9:         $bestCostSeen \leftarrow altBestCostSeen$
10:        $bestSolution \leftarrow altBestSolution$
11: return bestCostSeen, bestSolution
```

Function 11 constructAlternativeSolutionSet

Input: costFunction, input, feasibleSolution, useOnlyBest=True
Output: alternativeSolutionSet, bestCostSeen, bestInNeighbourhood

```
 1: $costFeasibleSolution \leftarrow costFunction(input, feasibleSolution)$
 2: $bestCostSeen \leftarrow costFeasibleSolution$
 3: $neighbourhood \leftarrow constructNeighbourhood(input, feasibleSolution)$
 4: for combinations in neighbourhood do
 5:     $alternativeSolutions \leftarrow$ Create empty list
 6:     $alternativeSolutions \leftarrow alternativeSolutions + feasibleSolution$
 7:     $foundFeasibleSolutions \leftarrow$ Create empty list
 8:     Iteratively construct alternative solutions from combinations of possible
        reassignments by removing given package-truck assignments and
        trying alternatives
 9:     Add feasible solutions with lower cost than feasibleSolution to
        alternativeSolutionSet
10:     Update bestCostSeen and bestInNeighbourhood if necessary
11: if useOnlyBest == true then
12:     Update alternativeSolutionSet to contain only bestInNeighbourhood
13: return alternativeSolutionSet, bestCostSeen, bestInNeighbourhood
```

Function 12 constructNeighbourhood

Input: input, feasibleSolution, distance
Output: allPossibleAlternatives

```
 1: $k \leftarrow distance$
 2: $setOfAllCombinations \leftarrow$ create all k-combinations from 1..number of packages
 3: $allPossibleAlternatives \leftarrow$ Initialize empty list
 4: for combinationToReassign in setOfAllCombinations do
 5:     $alternatives \leftarrow$ Initialize empty list
 6:     for elementIndex in 1..distance do
 7:         $p \leftarrow combinationToReassign_{elementIndex}$
 8:         $t \leftarrow$ truck to which is package is assigned in feasibleSolution
 9:         $newAlternative_{packageToRemove} \leftarrow p$
10:         $newAlternative_{truckToRemove} \leftarrow t$
11:         $newAlternative_{assignments} \leftarrow \{(p, truck) | truck \in T : truck \neq t\}$
12:         $alternatives \leftarrow alternatives + newAlternative$
13:     $allPossibleAlternatives \leftarrow allPossibleAlternatives + alternatives$
14: return solution
```

Function 13 gcf - greedy cost function

Input: input, solution, potentialElemSolution, params
Output: gcfVal

```
1: $k \leftarrow distance$
2: $p, t \leftarrow potentialElemSolution$
3: $alreadyAssignedTruck \leftarrow$ 1 if truck is already used, otherwise 0
4: $capacityLeft \leftarrow$ capacity left in the truck
5: $pkgWeight \leftarrow input["packageWeight]_p$
6: $pkgSize \leftarrow input["packageX]_p * input["packageY"]_p$
7: $possibilitiesForPkg \leftarrow$ number of trucks to which can be package assigned
8: $gcfVal \leftarrow$ params["costAlreadyAssignedTruck"] * alreadyAssignedTruck
9: $gcfVal$ += possibilitiesForPkg * params["costForPossibleTruck"]
10: $gcfVal$ += pkgSize * params["costForPackageSize"]
11: $gcfVal$ += pkgWeight * params["costForPackageWeight"]
12: $gcfVal$ += capacityLeft * params["costForCapacityLeft"]
13: return gcfVal
```

4 Experiments

4.1 Experiment set-up

For running the experiments, we have created two instances generators. One is based on generating trucks and then filling them with packages, and the second one is based on generating packages and creating trucks for them on the way. We have solved all instances using IPL and heuristics, and compared the results.

4.2 Evaluation

For comparing the results obtained from the heuristics and IPL program, we had to come up with some formula which quantifies the size of the problem. The solution that we came up is:

$$instanceSize = constraints + variables$$

We compute the number of *constraints* and *variables* from the input data in the following way:

$$constraints = 1 + 2 * pl + 3 * tl + 2 * pl * xt * yt + \frac{pl! * tl}{(pl-2)!} + \frac{pl! * tl * xt * yt}{(pl-2)!}$$

$$variables = pl * tl + 2 * pl * xt * yt + 1 + tl$$

where:

pl = number of packages

tl = number of trucks

xt = width of truck

yt = height of truck

4.3 Comparing heuristics and ILP

For the heuristics which we have compared, we have used the one that has the neighbourhood size which equals to 1(meaning it always tries to just reassign one package-truck assignment) and always keeps the best solution found in the neighbourhood. Also, for the *alpha* parameter that is applied by GRASP, we used the value 0.05. We tuned the parameters for the greedy cost function by implementing a convenient mechanism which allowed us to run the same experiments with different parameters for the greedy cost function. We have constrained our GRASP heuristics with a maximum of 60 iterations. Furthermore, for the heuristics, we have run the experiments 5 times and averaged the results.

The results obtained can be seen in Figure 1. We can clearly observe that even though the heuristics sometimes do not achieve an optimal solution, the solving time is much better. Heuristics seem to be a better choice when we do not care strictly about the optimal solution, but are satisfied with a near-optimal solution. We can also observe that sometimes ILP program solves problems with bigger size quicker than those with smaller size. We assume that it is because ILP program sometimes achieves to make a good cut (used in Branch and Cut algorithm) quickly and that significantly improves solving time.

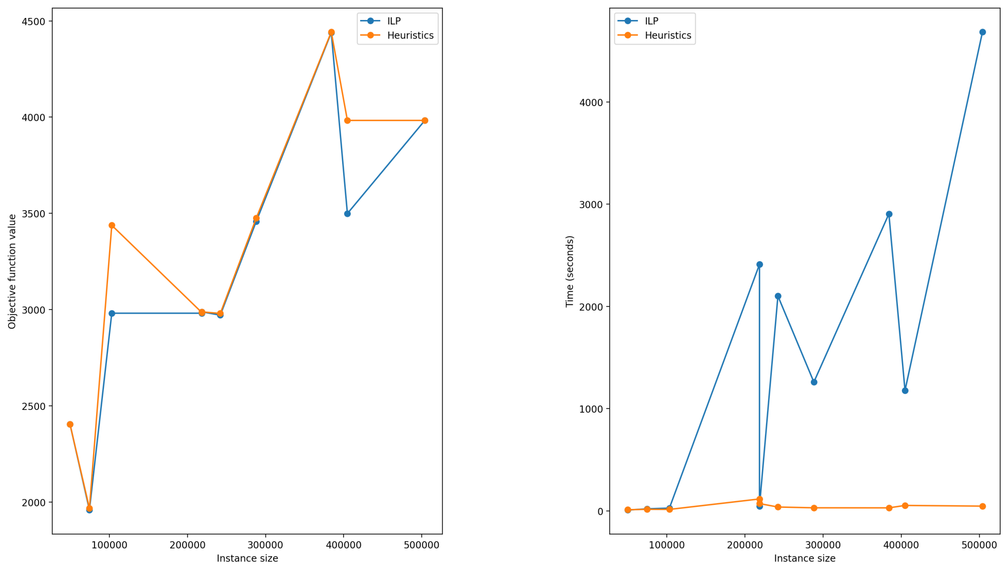

Figure 1: ILP vs Heuristics 1

4.4 Comparing heuristics

For comparing the heuristics, we have chosen heuristics which has been compared with IPL, and we also have created one new heuristic. The Heuristics 2 differs from Heuristics 1 in a sense that it uses neighbourhood of size 2. The next difference is that it takes the first solution from the neighbourhood that is better than the feasible solution, and doesn't consider another. The last difference is that it uses a value of *alpha* equal to 0.5. The idea behind Heuristics

2 is that it is supposed to construct more diverse solutions by setting a higher alpha, and also explores more in the local search as it uses a bigger distance for exploring the neighbourhood. The hypothesis stated that it could lead to better solutions for problems with bigger sizes.

The results obtained can be seen in Figure 2. It is noticeable that our hypothesis, by constructing a feasible solution with more randomness, and by exploring a bigger size of the neighbourhood, we should be able to obtain better solutions, was not true. We can observe that it did not improve the objective function value obtained (in fact, the value was sometimes worse). Moreover, we can see that for instances with a bigger size, the solving time of Heuristics 2 grows exponentially.

It is worth pointing out that we have even tried to use heuristics which explored neighbourhood of size 3 and kept all solutions from the neigbourhood that were better than the feasible solution obtained. The solving time for this heuristics for the biggest problem was extremely big (we could not get a result in reasonable time).

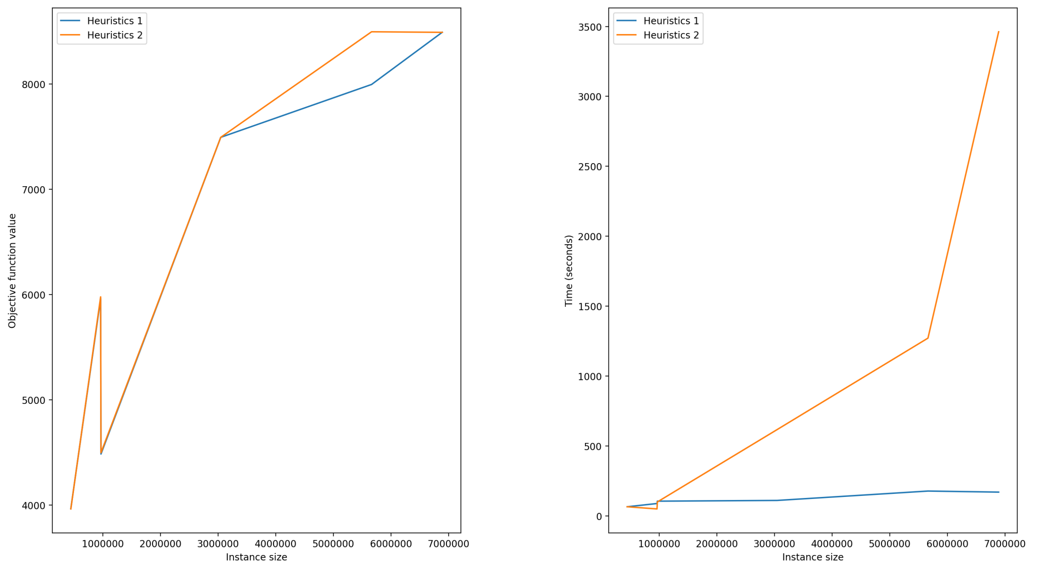

Figure 2: Heuristics 1 vs Heuristics 2