Project artificial intelligence 2018-2019

Olivier Van den Eede[1], Dries Kennes[1]

[1]KU Leuven - De Nayer

1 Problem Description

The Cambio car sharing project lets you make a reservation for a car at a specific time in a specific zone. The goal of this optimization problem is to find the most optimal location for each car, in order to assign as many reservations as possible in one city.

The city has been divided into zones, every car has a vehicle type and each reservation consists of:

- A required zone;
- A start and end date and time;
- A type of vehicle.

This means a request can only be assigned to the required zone, or a neighboring zone. Only cars of the right type can be used for a reservation. And of-course a reservation can only be assigned if a car is free for the entire duration of the reservation.

The goal is to assign each car to a zone, and assign each reservation to a car. No limit has been specified for the amount of cars assigned to one zone.

The cost of this problem has to be as low as possible, and can be calculated by taking the sum of 1) the P factor if the request has not been assigned; and 2) the Q factor if the request has been assigned to a neighbour as shown in formula 1. Every request has a different P and Q factor.

$$cost = \sum_{r}^{\#Requests} (P_r \cdot r_1 + Q_r \cdot r_2) \tag{1}$$

- $r_1 = 1$ if r is not assigned else 0;
- $r_2 = 1$ if r is assigned to a neighbour else 0.

2 Our Solver

We wrote our solver in Python 3, and will be using it's terminology in this paper.

2.1 Data Structures and Representation

This is an overview of the data structures we use.

2.1.1 Request, Zone and Car

Requests and zones are parsed into objects of their respective class containing all fields necessary to store the input data in basic types. Cars do not have a custom object, they are represented with a simple string.

After parsing the input data, all request and zone objects are stored in 2 lists, and in a dictionary each for fast lookup by id. These objects are all considered immutable so they never have to be copied.

2.1.2 Solution

A particular solution is represented by a Solution object. This object contains a reference to the parsed input data, and 2 dictionaries. These dictionaries represent the relationship between a car and zone, and between a request and a car. The cost function can be calculated based on the `car_zone` and `req_car` dictionaries.

This is the only object that must be copied for every local move. This saves on memory churn and thus CPU time.

2.1.3 Overlap Matrix

To speed up some local moves, we pre-calculate a matrix of overlapping requests. This 2D boolean matrix allows for a fast feasibility check because the slow overlap checking loop is eliminated.

2.2 Algorithm

Our local search algorithm is composed of 3 major parts.

2.2.1 Greedy Assign

The greedy assign function loops over the currently unassigned requests, and tries to find a car already assigned to the right zone or a neighboring zone. If a suitable car is found, the request will be assigned. If no suitable car is available, the function will take a new car and assign it to the required zone and request.

This function is used to create our initial solution, and guaranties a feasible solution. After each of our local moves, this function in called to try and fill all gaps and assign as many requests as possible.

2.2.2 Local Moves

Our algorithm is based around 5 small local changes:

- Move a request from the 'optimal' zone to one of its neighbors
- Move a request assigned to a neighbor zone to the 'optimal' zone.
- Swap the car assigned to a request with another car in the same zone.
- Unassign 1 car from all its requests.†
- Unassign 1 request.†

The local moves with † are twice as likely to be chosen. This multiplier has been imperially determined.

None of these moves make the solution infeasible. This potentially limits the search space that can be reached, but it also make the computation less complex and thus much faster.

2.2.3 Metaheuristic

To control all small local changes, we implemented simulated annealing. This allows our algorithm to find a different solution in the neighborhood of the working solution. This new solution will be accepted if the cost is lower, and accepted or rejected with a random probability based on the simulated annealing factors. When the stop-condition for our algorithm is reached, a new random initial solution will be generated and start again.

Our parameters:

- $T_{\max} = 1000$;
- $T_{\min} = 10$;
- $\alpha = 0.65$;
- $N = 5000$.

This results in $5000\lceil \log_{0.65}(1000/10) \rceil = 55000$ per completed run.

The data we used to justify our simulated annealing parameters can be found in the attached archive, or online on cambio.dries007.net/sa_parameter_tests.

2.3 Determinism

Despite our best efforts our solver is not deterministic, i.e., the same seed input does not result in the same output. We are not sure why that is the case, but decided it was not worth spending more time on.

3 Results

We consider our implementation a success. We reach results at least 10% better than the published results on Toledo with the 300 seconds on 2 threads. The results in table 1 results are the median of a number of runs. Figure 1 is a separate run because the data to generate the graphs is not saved on normal runs. Every line on the graph is one iteration of the simulated annealing part.

Bestand	Toledo	Score	Improvement
`100_5_14_25.csv`	10015	8795	12.18%
`100_5_19_25.csv`	6700	6045	9.78%
`210_5_33_25.csv`	14325	11180	21.95%
`210_5_44_25.csv`	5890	4045	31.32%
`360_5_71_25.csv`	12955	7490	42.18%

Table 1: Our results compared to the scores on Toledo.

3.1 Performance

Our solver manages to perform in the order of 12000 local moves per second (or 12kHz) with an input of 100 requests on an Intel i5-4670 CPU running at 4GHz. For the 210 requests example file we reach 10kHz. With 360 requests the performance is 5KHz.

This performance is reached thanks careful analysis of the code and runtime profiling. Removal of unused sections of the algorithm such as the feasibility check (since we never make a solution non feasible), made a very considerable (up to 50%) difference. List creations have been replaced with generators wherever possible. Short circuiting also helps reduce the average time spent checking overlap for example. The solution object is only cloned if it has to be, meaning if a local move failed, it can be used again for the next attempt.

With our simulated annealing parameters as mentioned in 2.2.3, the time for a single iteration of our algorithm remains low (order of ten seconds). This means that we can run many iterations in the given time limit (5 minutes). This creates many opportunities to find a good solution, even if our algorithm cannot reach all of the solution space.

3.2 Search Space

Because our algorithm uses *Greedy Assign* after every successful local move, it cannot reach all of the search space. For example: We cannot unassign two cars without attempting to assigning requests to them. We consider this a downside of our implementation.

3.3 Possible Improvements

We consider the following items possible improvements, or at least worthy of future research:

- Attempting to reach more of the search space, as mentioned in 3.2.
- Adding more local moves.
- Researching the chance multipliers the local moves use.
- Allowing the solution to become infeasible, to reach disjointed regions on the

4 Appendix

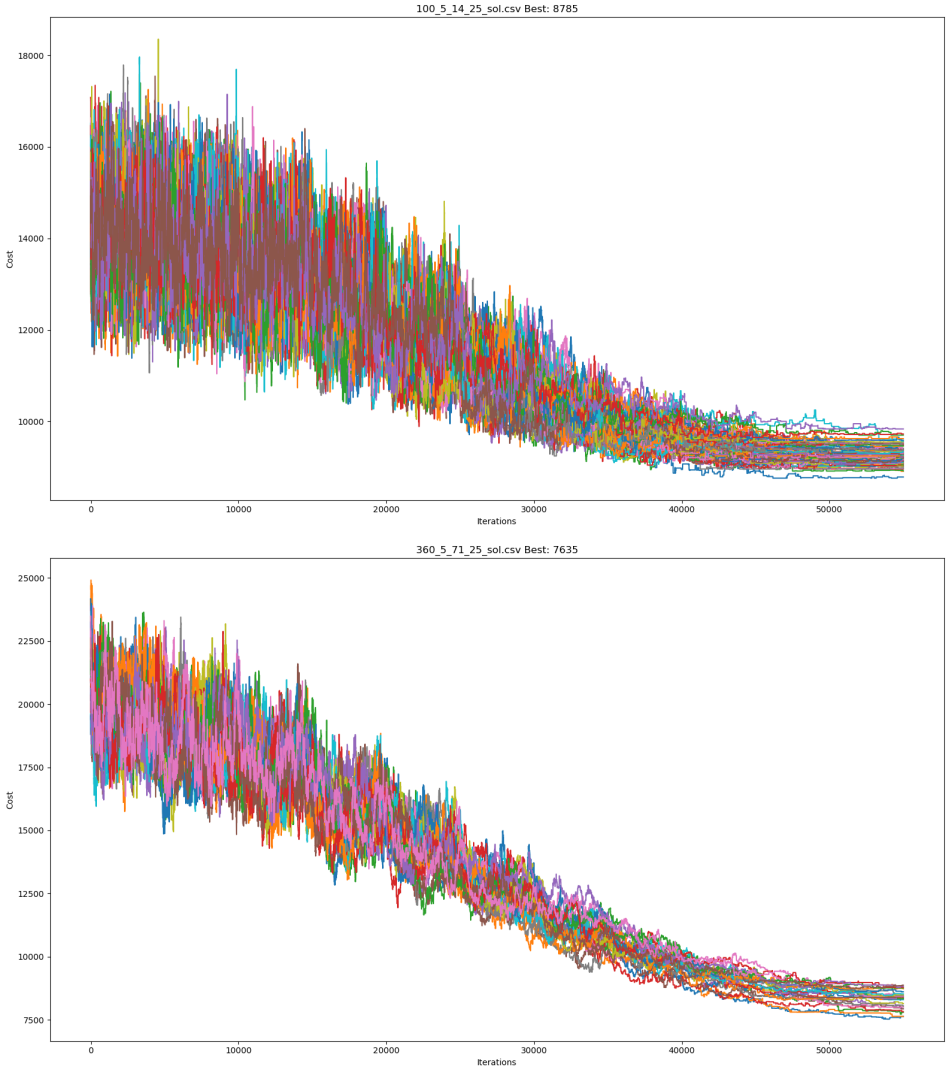

Figure 1: Debug run of `100_5_14_25.csv` and `360_5_71_25.csv`. These images are also hosted on `cambio.dries007.net/img`.