Warehouse Allocation Problem

Czech Technical University in Prague

Faculty of Electrical Engineering

Marek Jagoš

January 2025

Contents

List of Figures

1 Warehouse Allocation Problem

The Warehouse Allocation Problem is an optimization problem where the goal is to assign geographically distributed customers to warehouses in such a way that the total cost, consisting of transportation costs and warehouse setup costs, is minimized. Each warehouse has a limited capacity, and each customer has a specific demand. There are

- N warehouses, each warehouse w has capacity cap_w and set-up cost s_w
- M customers, each customer c demands certain amount of goods d_c
- For each pair $< c, w >$ a cost t_{cw} for delivering the goods from w to c is defined.

1.1 Solution representation

The solution is represented as array C_w of size M, where $i-th$ value represents a number of the warehouse assigned to $i-th$ customer. This representation was chosen because it is simple and does not yield data redundancy. A solution is only feasible if the sum of customer demands assigned to a given warehouse is less than the capacity of that warehouse.

$$\sum_{c \in C_w} d_c \leq cap_w, \quad \forall w \in \{1, \ldots, N\}$$

1.2 Fitness function

The fitness function is the accumulated cost of

- Warehouse setup costs s_w of warehouses that have at least 1 customer assigned to it in C_w,
- Delivery costs from each customer to given warehouse t_{cw} according to C_w.

$$f(x) = \sum_{w \in N} (I(|C_w| > 0) \cdot s_w + \sum_{c \in C_w} t_{cw})$$

2 Local Search

The Local Search was the first implemented approach and it quickly became apparent that the solution needs to be very computationally efficient to solve some of the larger input files. The initial design included, at first glance, basic operations, such as fetching all warehouses that can accommodate a customer, but these proved to be computationally infeasible. Finally, the approach was reduced and settled on something that is computationally cheap but very effective.

2.1 Finding warehouses for customers

For each customer, a vector $cheapest_w$ is pre-computed, which contains warehouses sorted by $t_{cw} + s_w$. This yields a vector which has the customer's cheapest warehouses at the beginning. Including raw setup cost of a warehouse is slightly inaccurate, but it served well.
When looking for a warehouse to accommodate a customer, $cheapest_w$ is iterated and the first warehouse that can accommodate it is used. This approach can also achieve some degree of randomization, such as randomly selecting 5 closest available warehouses.

2.2 Initializing solutions

Although the above approach could be used to initialize the solution and immediately get very solid results, it would cause every solution to start with the same solution vector. This makes restarts almost useless as the diversity between runs would be extremely low. For that reason, the solution is initialized randomly.

2.3 Kicking a customer

The perturbation function is simply forcing a customer to leave a warehouse and find another one.

2.4 Repairing infeasible solutions

Random initialization and selected perturbation function can create situations where a customer has nowhere else to go. In this case, a random warehouse is closed down (all customers of a warehouse are forced to leave) and the dangling customer is moved in. This is recursive, as the newly kicked out customers might find themselves without a warehouse as well. This could cause issues in datasets where capacity and demands are extremely tight, but this is not the case in the provided datasets.

2.5 Selecting a customer to kick

To guide the selection, a simple rank-based heuristic was implemented. The selection is random, but customers with higher fitness function contribution are preferred. The weight of selecting a customer is proportional to r^2, where r represents the rank of the customer's contribution to the fitness function.
This heuristic function improved the results and considerably reduced the number of epochs.

2.6 Local search algorithm

The Local Search algorithm follows a first-improvement strategy with a predefined maximum number of epochs and a patience threshold. Neighboring solutions are generated using the previously described perturbation function, and if a neighboring solution improves upon the current one, it is accepted as the new solution. If that is not the case, the solution stays the same and the patience of the algorithm decreases. This approach does not enable escaping from local optima, instead the solver is restarted when solution stagnates.

2.7 Results

The following parameters were passed to the local search solver:

```
HEURISTIC = true;
RESTARTS = 100;
MAXEPOCH = 50000;
PATIENCE = 25;
```

Figures 1, 2, 3 contain the best fitness per evaluation averaged over 100 runs of the Local Search solver with the above configuration.

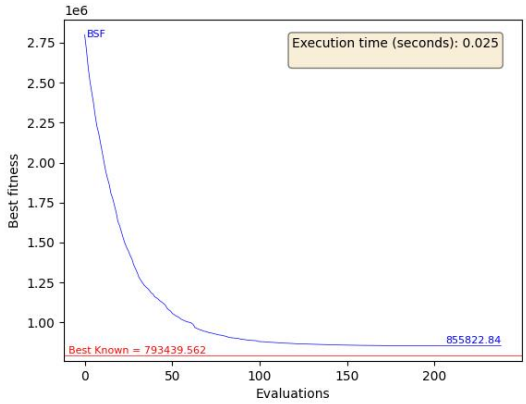

Figure 1: Local Search wl_50_1

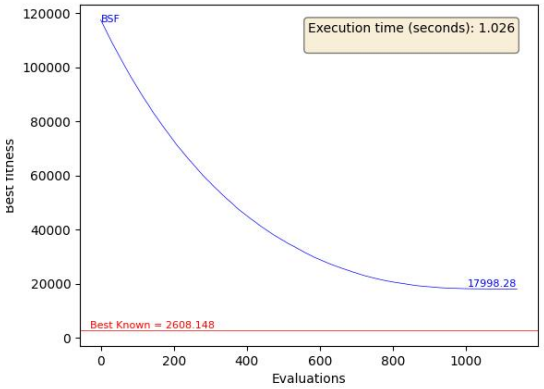

Figure 2: Local Search wl_500_1

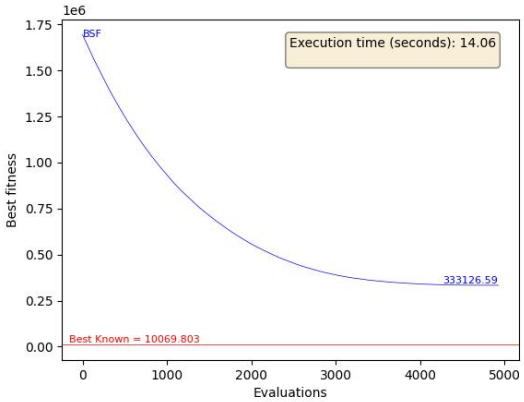

Figure 3: Local Search wl_2000_1

3 Evolutionary algorithm

The evolutionary algorithm extends the Local Search approach. Solutions in a population are generated randomly to preserve diversity. The perturbation function is reused as a mutation operator.

3.1 Selecting parents

For each offspring, 2 parents are selected through **tournament selection** - 2 tournaments are held, in each of them N solutions from the population are randomly selected. The solution with smallest fitness value is declared winner and becomes a parent.

3.2 Crossover

The crossover operator is a **1-point crossover with repair and random cutoff**. This approach is computationally simple, while it allows preservation of continuous segments of good solutions, yet it can change the solution considerably if the population is diverse enough.
1-point crossover can generate solutions with overflowing warehouses. Such warehouses have their customers moved until they no longer overflow.

3.3 Generational replacement strategy

The number of offspring generated is equal to the size of the population and generational replacement occurs.

3.4 Results

The following parameters were passed to the evolutionary algorithm solver:

```
HEURISTIC = true;
RESTARTS = 100;
MAXEPOCH = 50000;
PATIENCE = 25;
POPULATION_SIZE = 100;
TOURNAMENT_SIZE = 5;
```

Figures 4, 5, 6 contain the best fitness per evaluation averaged over 100 runs of the Evolutionary Algorithm solver with the above configuration.

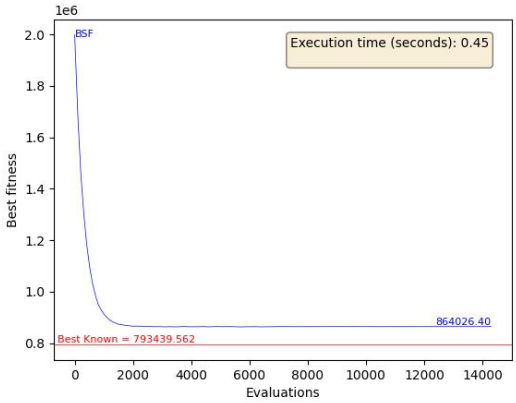

Figure 4: Evolutionary Algorithm wl_50_1

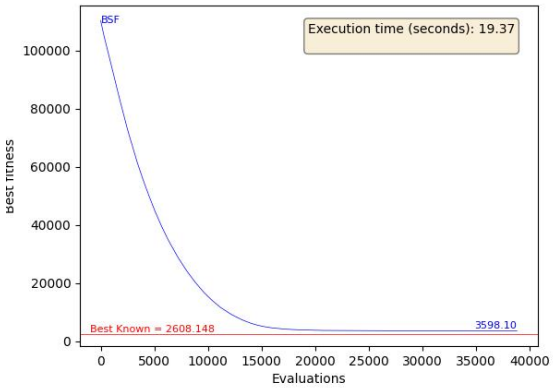

Figure 5: Evolutionary Algorithm wl_500_1

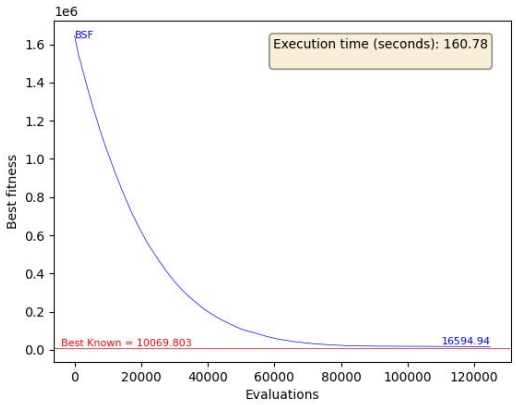

Figure 6: Evolutionary Algorithm wl_2000_1

4 Memetic algorithm

The memetic algorithm combines the methods above and it proved to be very effective, as the Local Search approach in itself is very efficient in finding local solutions, but loses diversity very quickly. Combining it with Evolutionary algorithm allows us to preserve diversity for longer and find solutions that are closer to the best known solution. For each new offspring, there is a parametrized chance that a local search will be applied to it.

4.1 Results

The following parameters were passed to the memetic algorithm solver:

```
HEURISTIC = true;
RESTARTS = 100;
MAXEPOCH = 50000;
PATIENCE = 25;
POPULATION_SIZE = 100;
TOURNAMENT_SIZE = 5;
LS_PROBABILITY = 0.5;
LS_MAX_DEPTH = 20;
LS_PATIENCE = 5;
```

Figures 7, 8, 9 contain the best fitness per evaluation averaged over 100 runs of the memetic algorithm solver with the above configuration.

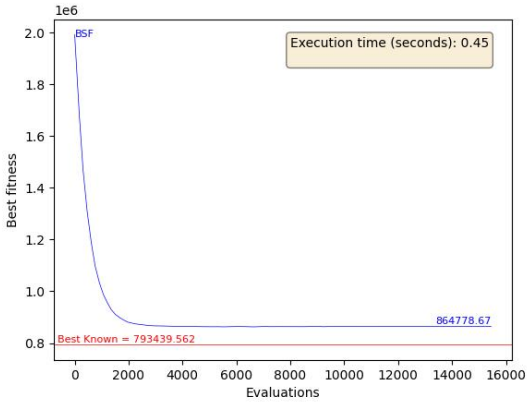

Figure 7: Memetic Algorithm wl_50_1

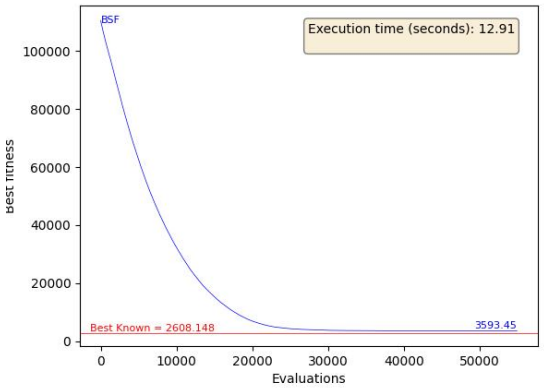

Figure 8: Memetic Algorithm wl_500_1

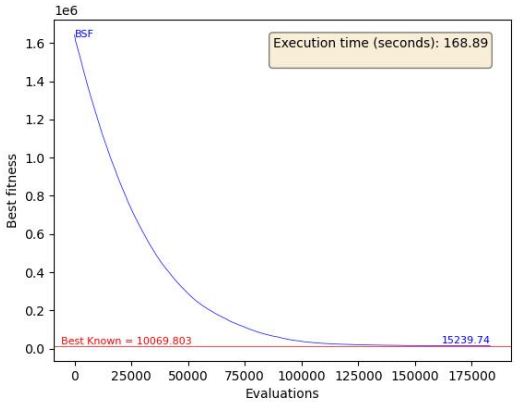

Figure 9: Memetic Algorithm wl_2000_1

5 Results

The comparison of the mentioned results can be seen in the figure 10. While the Local Search algorithm is able to achieve fastest convergence towards the best known solution, it gets stuck repeatedly in local optimum on large problem instances. The evolutionary algorithm and memetic algorithm converge slower in terms of evaluations, but but are able to achieve decent fitness values even for large problem instances. The memetic algorithm is able to reach slightly better fitness values than evolutionary algorithm.

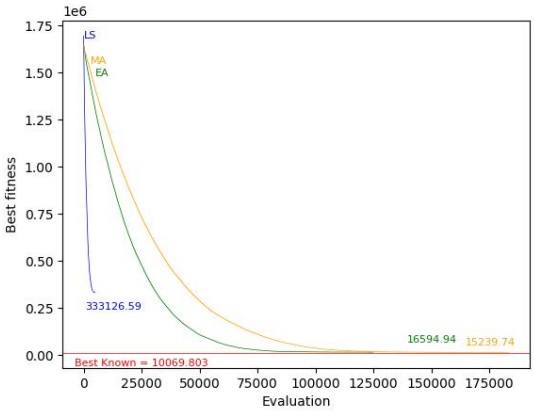

Figure 10: Comparison wl_2000_1

5.1 Other observations

- All of the approaches are capable of producing reasonable results on small problem instances, achieving values within 10% of the best known solution as seen at 1 4 7. Evolutionary and memetic algorithms are able to find decent solutions on the largest problem instance and that is within 65% of the best known solution. 6 9.
- Memetic algorithm achieved best results and fastest convergence in terms of epochs, but slowest convergence in terms of evaluations. 7 8 9
- The local search execution time is extremely fast, averaging to just 3ms per epoch on the largest instance. 3