Warehouse Allocation Problem

Czech Technical University in Prague

Evoluční optimalizační algoritmy (A0M33EOA)

Marek Jagoš

6 January 2024

Contents

Contents

1 Warehouse Allocation Problem

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 is precomputed. This vector contains warehouse indices sorted by formula $T_C W[c][w] + S_W[w]$, where $T_C W[c][w]$ represents the cost of associating a specific customer with a warehouse, and $S_W[w]$ the warehouse setup cost. This yields a vector which has the cheapest warehouses for a customer at the start of the vector. Including raw setup cost of a warehouse is slightly inaccurate, but it served very well.
When looking for warehouse for a customer, this is utilized such that each customer receives first cheapest warehouse that can accommodate it. This is used in a perturbation function. It is very cheap and effective, and some degree of randomization (such as randomly select from 5 closest available warehouses) is possible.

2.2 Generating solutions

Although the approach above could be used to initialize the solution and immediately get very solid results, this approach has a major flaw and that is that the precomputation happens on problem level and therefore each initialized solution would 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, utilizing the approach described above.

2.4 Repairing infeasible solutions

Random initialization and kicking a customer 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

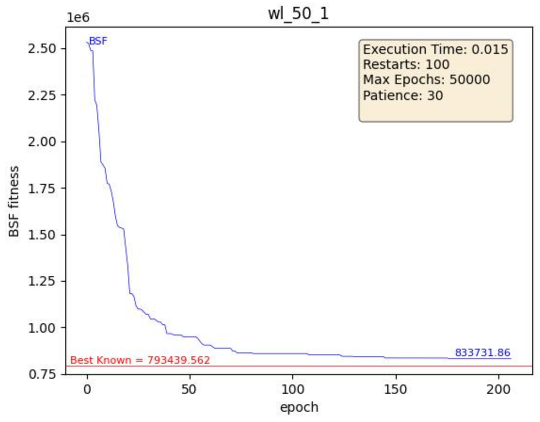
To guide the selection, a simple rank-based heuristic was implemented. The selection is random, but customers with higher fitness function contribution are prefered. The weight of selecting a customer is proportional to r^2, where r represents the rank of the customer's contribution to the fitness function.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

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's not the case, the solution stays the same and patience of the algorithm decreases. This approach does not enable escaping from local optima, instead the solver is restarted when solution stagnates.

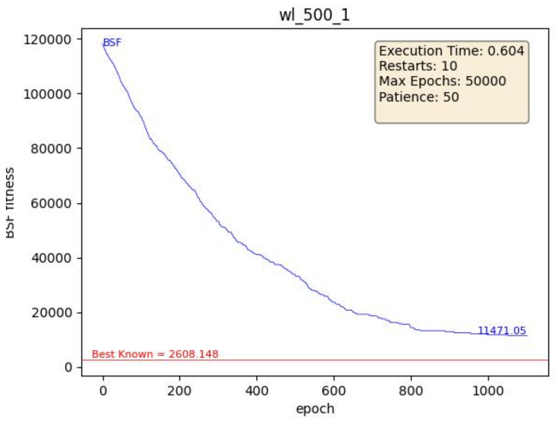
2.7 Results

All of the following charts represent a single solution. The number of evaluations is equal to the number of epochs.



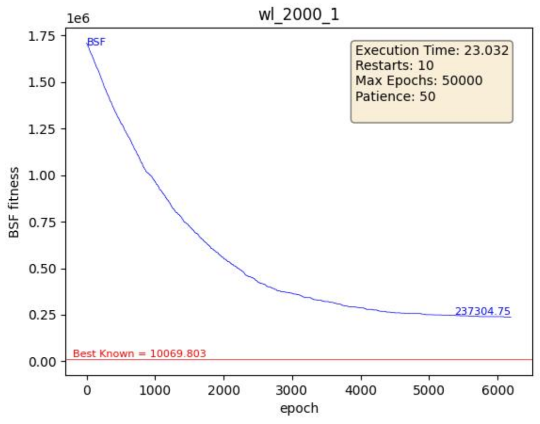
Execution Time: 0.015
Restarts: 100
Max Epochs: 50000
Patience: 30

Figure 1: Local Search applied on wl_50_1



Execution Time: 0.604
Restarts: 10
Max Epochs: 50000
Patience: 50

Figure 2: Local Search applied on wl_500_1



Execution Time: 23.032
Restarts: 10
Max Epochs: 50000
Patience: 50

Figure 3: Local Search applied on wl_2000_1

3 Evolutionary algorithm

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

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 picked randomly, the best solution is declared winner and becomes a parent.

3.2 Crossover

Crossover operator is a simple **1-point crossover with repair and random cutoff**. This approach is (fairly) computationally simple, which is necessary to solve the large input files, while it allows preservation of continuous segments of good solutions, yet can change the solution drastically if the population is diverse enough. 1-point crossover can generate solutions with overflowing warehouses. Such warehouses have their customers moved until they do not overflow anymore. This applies the local search infeasible solution repairing mechanism.

3.3 Generational replacement strategy

The number of offspring generated is equal to the size of the population in order to perfrom **generational replacement**. It would be worth exploring other methods as well.

3.4 Results

All of the following charts contain the BSF of a single distinct evolutionary algorithm execution. Since the population equals to 100, one epoch equals to 100 evaluations.

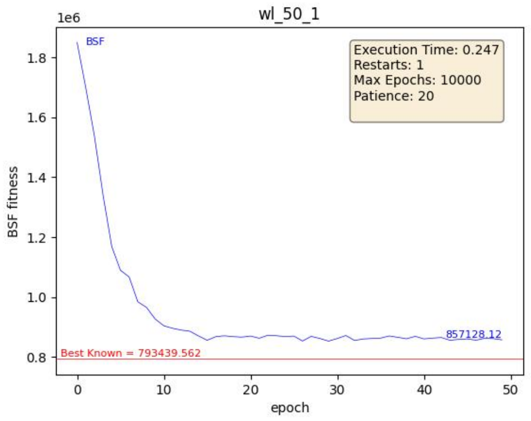


Figure 4: EA applied on wl_50_1

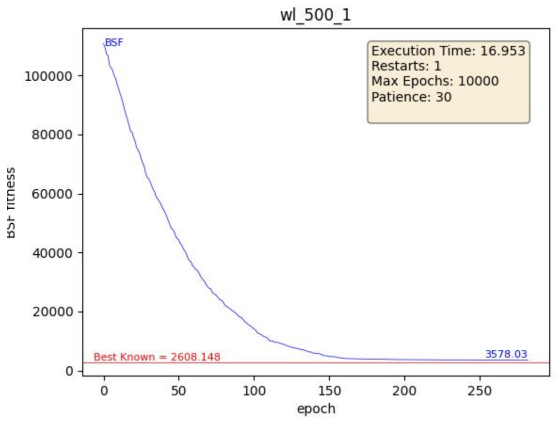


Figure 5: EA applied on wl_500_1

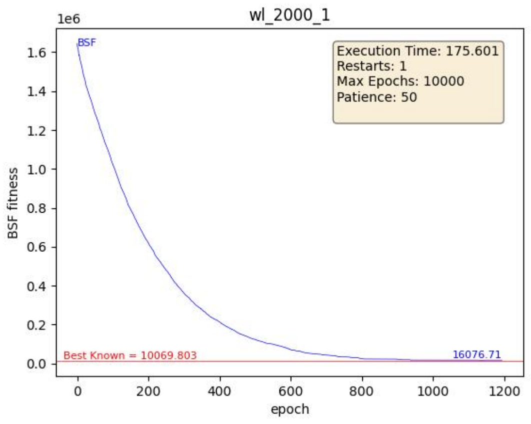


Figure 6: EA applied on wl_2000_1

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.
For each new offspring, there is a parametrized chance that a local search will be applied to it.

4.1 Results

All of the following charts contain the BSF of a single distinct memetic algorithm execution. Since the population equals to 100, one epoch equals to 100 evaluations potentially multiplied by depth of a local search on an offspring.

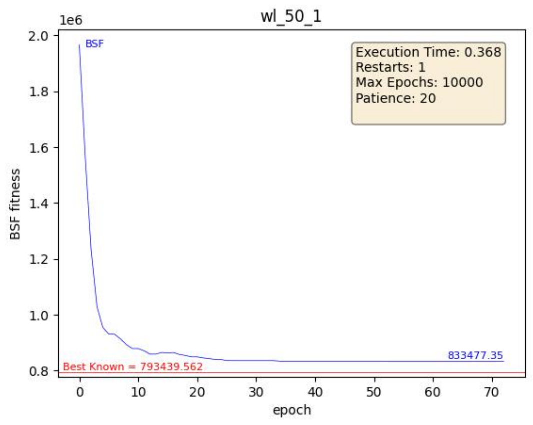


Figure 7: Memetic Algorithm applied on wl_50_1

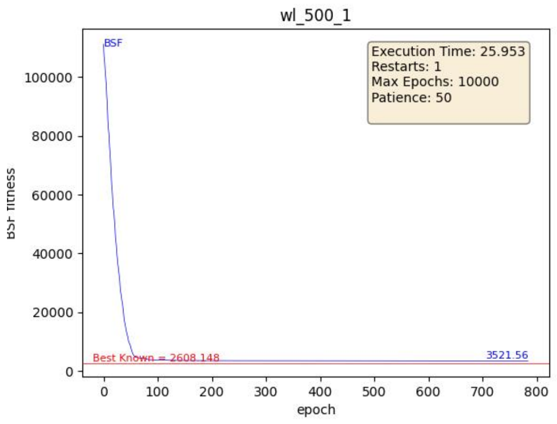
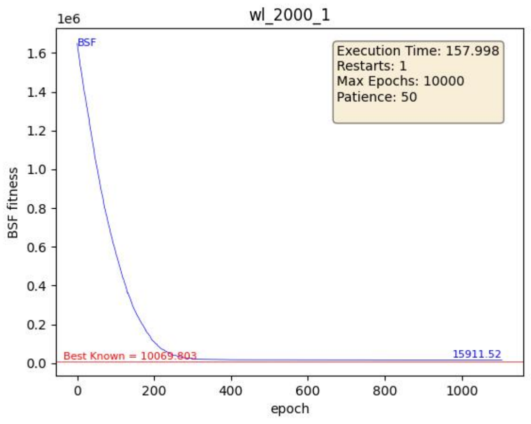


Figure 8: Memetic Algorithm applied on wl_500_1



Execution Time: 157.998
Restarts: 1
Max Epochs: 10000
Patience: 50

Figure 9: Memetic Algorithm applied on wl_2000_1

5 Observations

- Memetic algorithm achieved best results and fastest convergence in terms of epochs
- The local search in itself is competitive on small problem instances, but since it lacks local optima escape mechanism, it fails to approach optimum on large problem instances.
- The evolutionary algorithm is capable of finding near-optimal solutions, even for the largest problem instances. It however does not perform significantly better on smaller problem instances.
- The memetic algorithm converges to near-optimal solutions very quickly, and is able to slightly surpass evolutionary algorithm in terms of finding the best solution. It would be worth researching whether increasing patience and max epochs, and perhaps aggression of the local search in the later stages of the search would enable us to come even closer to best known solution.
- The local search execution time is extremely fast, averaging to just 3ms per epoch on the largest instance. The EA is still very efficient, but suffers from burden of evaluation many solutions at once.

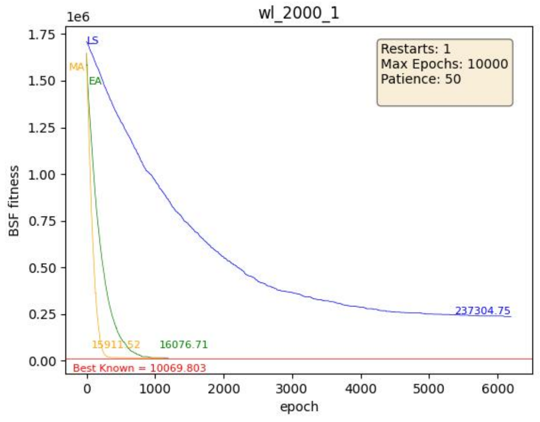


Figure 10: Comparison of all methods on wl_2000_1