Extending Yuck for Vehicle Routing

Michael Marte

May 1, 2021

1 Introduction

Yuck is a FlatZinc solver that combines local search with restarting, global constraints, and lexicographic cost functions. It is open source and provided under the terms of the Mozilla Public License 2.0. Source code, binary packages, and documentation are available from GitHub: https://github.com/informarte/yuck.

Recent research [BFP19] has identified challenges posed to local-search solvers by typical MiniZinc models for vehicle routing. To address these issues, Yuck has been extended with support for single-depot vehicle routing in the form of two global constraints: `giant_tour` and `delivery`.

This system description introduces `giant_tour` and `delivery`, presents a comparative, computational study with very promising results, and closes with perspectives for future work.

2 The `giant_tour` Constraint

The `delivery` contraint requires the model to be based on the giant-tour representation (see [RBW06], section 23.3.1) where, in addition to the nodes to be visited, there is a pair of start and end nodes for each vehicle:

```
predicate giant_tour(
    set of int: StartNodes,
    set of int: EndNodes,
    array[int] of var int: succ)
=
    let {
        set of int: Nodes = index_set(succ),
        int: K = card(StartNodes)
    }
    in circuit(succ)
    /\ forall(i in 1..(K - 1))(succ[min(EndNodes) + i - 1] = min(StartNodes) + i)
    /\ succ[max(EndNodes)] = min(StartNodes);
```

Yuck's implementation of `giant_tour` extends the above decomposition with sanity checks. Its use is recommended because there are plans to provide a native implementation, see section 5.

3 The `delivery` Constraint

Given a giant-tour representation, `delivery` can be used to constrain the arrival times to consider the given service and travel times:

```
predicate delivery(
    set of int: StartNodes,
    set of int: EndNodes,
    array[int] of var int: succ,
    array[int] of var int: arrivalTimes,
    array[int] of int: ServiceTimes,
    array[int, int] of int: TravelTimes,
    bool: WithWaiting,
    var int: totalTravelTime)
=
    let {
        set of int: Nodes = index_set(succ)
    }
    in forall(i in Nodes diff EndNodes)(
        let {
            var int: j = succ[i],
            var int: t = arrivalTimes[i] + ServiceTimes[i] + TravelTimes[i, j],
            var int: u = arrivalTimes[j]
        }
        in if WithWaiting then u >= t else u = t endif
    )
    /\ totalTravelTime = sum(i in Nodes diff EndNodes)(TravelTimes[i, succ[i]]);
```

Yuck provides a native implementation of `delivery` which avoids the performance issues identified by [BFP19]. Departing from the given arrival-times at start nodes, this implementation maintains the arrival times for all other nodes, the total travel time, and the constraint violation

$$\sum_{x \in \texttt{arrivalTimes}} \delta(x) + \delta(\texttt{totalTravelTime})$$

where

$$\delta(x) = \begin{cases} 0 & \text{if the value } a(x) \text{ of } x \text{ is contained in the domain } d(x) \text{ of } x \\ \min(d(x)) - a(x) & \text{if } a(x) < \min(d(x)) \\ a(x) - \max(d(x)) & \text{otherwise.} \end{cases}$$

When a vehicle arrives too early at a node x (earlier than $\min(d(x))$) and waiting is enabled, then `delivery` will postpone the arrival until $\min(d(x))$ but not until later.

With the above cost model, `delivery` can also be used in reified contexts, allowing for soft `delivery` constraints when used in combination with Yuck's `goal_hierarchy` annotation. For example,

```
solve :: goal_hierarchy(sat_goal(delivery(...)) satisfy;
```

would search a solution which minimizes the violation of the given `delivery` constraint.

Regarding the `delivery` constraint and its implementation in Yuck, there are several noteworthy aspects:

- The above definition of `delivery` uses times only for the purpose of presentation. In practice, `delivery` can be used to track all kinds of other things like distances, vehicle loads, or costs.
- Conceptionally, `delivery` corresponds to the routing dimensions[1] of Google OR-Tools.
- Yuck's `delivery` implementation extends the above definition with sanity checks and allows `ServiceTimes` and `TravelTimes` to be empty.
- Yuck's `delivery` implementation requires the assignment to the `succ` variables to form a Hamiltonian circuit at any time during search. The practical implication is that `delivery` can only be used in conjunction with `circuit`.
- Support for time windows is implicit: To impose time windows, just constrain the arrival-time variables accordingly.
- The above signature of `delivery` is not carved into stone as there are ideas for future extensions, see section 5.
- Regarding vehicle routing without waiting, [BFP19] pointed out that `arrivalTimes[j]` cannot be considered as functionally determined by `arrivalTimes[i]` unless the circuit constraint is taken into account. The resulting circular dependency requires to treat all arrival-time variables as search variables, rendering vehicle routing very hard for local search. With its native `delivery` implementation, Yuck avoids this issue by collapsing the entire constraint subgraph for defining arrival times into a single node. To achieve the same effect for vehicle routing with waiting, Yuck waits only as long as necessary, as explained above. This behaviour reduces the solution space in comparison to the above definition of `delivery`.

4 Computational Study

To verify the efficacy of Yuck's native `delivery` implementation, a computational study was performed. In this study, various FlatZinc solvers were applied to various VRP benchmarks, 968 instances in total and with the common goal to minimize total travel time. The following FlatZinc solvers were tested and compared:

- Chuffed[2] 0.10.4, as packaged with MiniZinc 2.5.5
- Gecode[3] 6.3.0, as packaged with MiniZinc 2.5.5
- Google OR-Tools[4] 8.1.8487 CP-SAT solver
  (Notice that Google OR-Tools has a routing library based on local search, but it cannot be used via MiniZinc.)
- Oscar/CBLS[5] 20201007
- Yuck

The following benchmarks were used:

[1] https://developers.google.com/optimization/routing/dimensions
[2] https://github.com/chuffed/chuffed
[3] https://www.gecode.org/
[4] https://developers.google.com/optimization
[5] https://www.it.uu.se/research/group/optimisation/software#fznoscarcbls

- CVRP (VRP with vehicle capacities):
  - Augerat (A, B, and P) [Aug+98] (30 - 100 nodes to visit)
  - Uchoa [Uch+17] (100 - 1000 nodes to visit)
- TSPTW (TSP with time windows):
  - Ascheuer [AFG01] (10 - 233 nodes to visit)
  - Dumas [Dum+95] (20 - 200 nodes to visit)
  - Gendreau [Gen+98] (20 - 100 nodes to visit)
- CVRPTW (VRP with vehicle capacities and time windows):
  - Homberger [GH99] (200 - 1000 nodes to visit)
  - Solomon [Sol87] (25 - 100 nodes to visit)

The MiniZinc models and the instances are available from Github:
https://github.com/informarte/minizinc-benchmarks/tree/yuck-testing

The MiniZinc models were derived from the CVRP model which was used in the MiniZinc challenge 2015. For each problem, there are two top-level models, e.g. `cvrp_cp.mzn` and `cvrp_yuck.mzn` where the CP model uses the `delivery` decomposition while the Yuck model uses the native `delivery` implementation. To avoid code duplication, both top-level models are based on a common ancestor, e.g. `cvrp.mzn`, which in turn is based on `vrp.mzn`; it's like a class hierarchy with super classes which forward-declare and use predicates to be defined by sub classes.

The DataZinc files were generated from Keld Helsgaun's collection of VRP benchmarks[6]. Two things about the instance files are worth noting:

- The CVRP and CVRPTW instances come with a number k of vehicles or tours, either as part of their filename, or defined by the VEHICLES property.
- The instance files neither specify the optimization goal nor the semantics ok k. (This makes sense because these benchmarks can be used with various primary optimization goals, like minimizing the total travel time, the makespan, or the number of vehicles.)

Now, to avoid comparing apples to oranges, it is important to know the circumstances under which the best-known total travel times were computed. To answer this question, it was necessary to consult the original papers:

- [Aug+98], introduction: "The Capacitated Vehicle Routing Problem (CVRP) we consider in this paper consists in an optimization that deals with the distribution of a commodity from a single depot to a given set of n customers with known demand, using a given number k of vehicles having all the same capacity C."
- [Uch+17], section 3.3.2: "The number $K_{\min}$ indicated in each instance should be taken only as a lower bound on the number of routes in a solution."
- [GH99], introduction: "The objective function considered here combines the minimization of the number of vehicles (primary criterion) and the total travel distance (secondary criterion)."

[6] http://akira.ruc.dk/~keld/research/LKH-3/

- [Sol87], introduction: "We also assume that the number of vehicles used is free, i.e., the fleet size is determined simultaneously, using the best set of routes and schedules."

Concluding, the k values of the Augerat instances are hard constraints, in all other cases they are lower bounds.

To support the different requirements, the MiniZinc models introduce the parameters `MinK` (the lower bound on the number of vehicles) and `MaxKToMinKRatio` from which they compute `MaxK` (the maximum number of vehicles) as the product of `MinK` and `MaxKToMinKRatio`.

We are left with the task of choosing the ratio. On the one hand, a higher ratio entails more start and end nodes and hence more search variables, on the other hand it gives solvers more wiggle room. A pre-study revealed that the ratio is crucial for solver performance and is best defined for each pair of solver and problem. For the main study, the following ratios were used:

	Uchoa	Homberger	Solomon
Chuffed	2	4	2
Gecode	2	3	2
OR-Tools	2	4	2
Oscar/CBLS	2	4	4
Yuck	2	4	4

The studies were performed on a first-gen i7 quad-core with 20 GB RAM. The runtime was limited to 60 seconds per instance, four instances were solved in parallel, and solver heap space was limited to 4 GB, with exception of the Homberger benchmark: High memory demands by Chuffed, Gecode, and OR-Tools required to double memory and to halve the number of solvers running in parallel.

Oscar/CBLS was run with the `-XuseCMG` option. ("CMG" is short for "Compound Move Generation", a generic technique developed by [BFP19] to improve the performance of local search.)

Appendix A presents the results of the main study in terms of gaps, i.e. the ratios of the objective values to the best known objective values. Table 1 summarizes the key performance indicators while the remaining figures (one per benchmark) compare the results graphically. Regarding the new version of Yuck, the key findings are:

- Yuck solved all of the CVRP instances, all TSPTW but 7 (out of 135) Dumas instances, and all CVRPTW but 2 (out of 300) Homberger instances.
- Regarding the quality of solutions, the mean gap was 1 or very close to 1 for Augerat, Ascheuer, Dumas, Gendreau, and Solomon, 1.29 for Uchoa, and 2 for Homberger.
- Yuck outperformed the other solvers.

5 Future Work

There are several ideas for future work:

- Provide a native `giant_tour` implementation: The `circuit` constraint does not know about the giant-tour representation with its start and end nodes. A specialized implementation could exploit this knowledge to provide better moves more efficiently.
- Improve the versatility of the `delivery` constraint: Replace the `WithWaiting` flag with waiting-time variables to allow for computing and minimizing the total waiting time as in [Sol87]. (To forbid waiting, the waiting-time variables could be set to 0.)

- Consider a renaming of `delivery` and its arguments: The application of `delivery` is neither limited to delivery problems nor to travel times and hence it might make sense to generalize the current definition.
- Put the Yuck results into perspective: Compare Yuck to a specialized TSP/VRP solver, like LKH-3 [Hel17].

References

[AFG01] N. Ascheuer, M. Fischetti, and M. Grötschel. "Solving the Asymmetric Travelling Salesman Problem with time windows by branch-and-cut". In: *Mathematical Programming* 90.3 (2001), pp. 475–506.

[Aug+98] P. Augerat et al. *Computational results with a branch and cut code for the capacitated vehicle routing problem*. Tech. rep. R.495. IASI-CNR, 1998.

[BFP19] G. Björdal, P. Flener, and J. Pearson. "Generating Compound Moves in Local Search by Hybridisation with Complete Search". In: *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 16th International Conference, CPAIOR 2019, Proceedings*. Ed. by L.-M. Rousseau and K. Stergiou. Vol. 11494. Lecture Notes in Computer Science. Springer, 2019, pp. 95–111.

[Dum+95] Y. Dumas et al. "An Optimal Algorithm for the Traveling Salesman Problem with Time Windows". In: *Operations Research* 43.2 (1995), pp. 367–371.

[Gen+98] M. Gendreau et al. "A Generalized Insertion Heuristic for the Traveling Salesman Problem with Time Windows". In: *Operations Research* 46.3 (1998), pp. 330–335.

[GH99] H. Gehring and J. Homberger. "A Parallel Hybrid Evolutionary Metaheuristic for the Vehicle Routing Problem with Time Windows". In: *Evolutionary Algorithms in Engineering and Computer Science: Recent Advances in Genetic Algorithms, Evolution Strategies, Evolutionary Programming, Genetic Programming and Industrial Applications*. Ed. by K. Miettinen et al. Wiley, 1999, pp. 57–64. ISBN: 978-0-471-99902-7.

[Hel17] K. Helsgaun. *An Extension of the Lin-Kernighan-Helsgaun TSP Solver for Constrained Traveling Salesman and Vehicle Routing Problems*. Tech. rep. Roskilde Universitet, 2017.

[RBW06] F. Rossi, P. van Beek, and T. Walsh, eds. *Handbook of Constraint Programming*. Elsevier, 2006. ISBN: 978-0-444-52726-4.

[Sol87] M. M. Solomon. "Algorithms for the Vehicle Routing and Scheduling Problems with Time Window Constraints". In: *Operations Research* 35.2 (1987), pp. 254–265.

[Uch+17] E. Uchoa et al. "New benchmark instances for the Capacitated Vehicle Routing Problem". In: *European Journal of Operational Research* 257.3 (2017), pp. 845–858.

A Results

Benchmark	Solver	Success rate	Min gap	Mean gap	Median gap	Max gap
Augerat	chuffed	95.95%	1.12	2.56	2.57	3.93
	gecode	95.95%	1.11	2.51	2.48	3.75
	or-tools	95.95%	1.11	2.46	2.45	3.75
	oscar-cbls	5.41%	1.00	1.04	1.03	1.08
	yuck	100.00%	1.00	1.02	1.01	1.09
Uchoa	chuffed	79.00%	1.64	4.11	4.01	9.00
	gecode	74.00%	1.64	4.19	4.06	9.00
	or-tools	9.00%	2.16	3.89	4.01	5.25
	oscar-cbls	0.00%				
	yuck	100.00%	1.04	1.29	1.23	2.31
Ascheuer	chuffed	86.00%	1.00	1.04	1.04	1.14
	gecode	78.00%	1.00	1.04	1.03	1.15
	or-tools	96.00%	1.00	1.04	1.00	1.14
	oscar-cbls	50.00%	1.00	1.01	1.00	1.04
	yuck	100.00%	1.00	1.00	1.00	1.01
Dumas	chuffed	53.33%	1.00	1.03	1.00	1.21
	gecode	84.44%	1.00	1.08	1.04	1.45
	or-tools	51.85%	1.00	1.03	1.00	1.23
	oscar-cbls	48.15%	1.00	1.01	1.00	1.08
	yuck	94.81%	1.00	1.00	1.00	1.08
Gendreau	chuffed	15.71%	1.00	1.06	1.00	1.27
	gecode	20.00%	1.00	1.04	1.00	1.51
	or-tools	11.43%	1.00	1.04	1.00	1.27
	oscar-cbls	18.57%	1.00	1.01	1.00	1.11
	yuck	100.00%	1.00	1.01	1.00	1.12
Homberger	chuffed	52.67%	1.67	5.54	4.92	12.11
	gecode	33.33%	1.71	4.38	4.13	8.08
	or-tools	0.67%	6.47	6.66	6.66	6.85
	oscar-cbls	0.00%				
	yuck	99.33%	1.00	2.00	1.67	28.43
Solomon	chuffed	97.04%	1.00	2.36	2.16	6.58
	gecode	83.43%	1.00	2.51	2.17	7.30
	or-tools	92.31%	1.00	2.19	2.03	6.32
	oscar-cbls	71.60%	1.00	1.45	1.27	3.36
	yuck	100.00%	1.00	1.03	1.00	1.20

Table 1: Key performance indicators

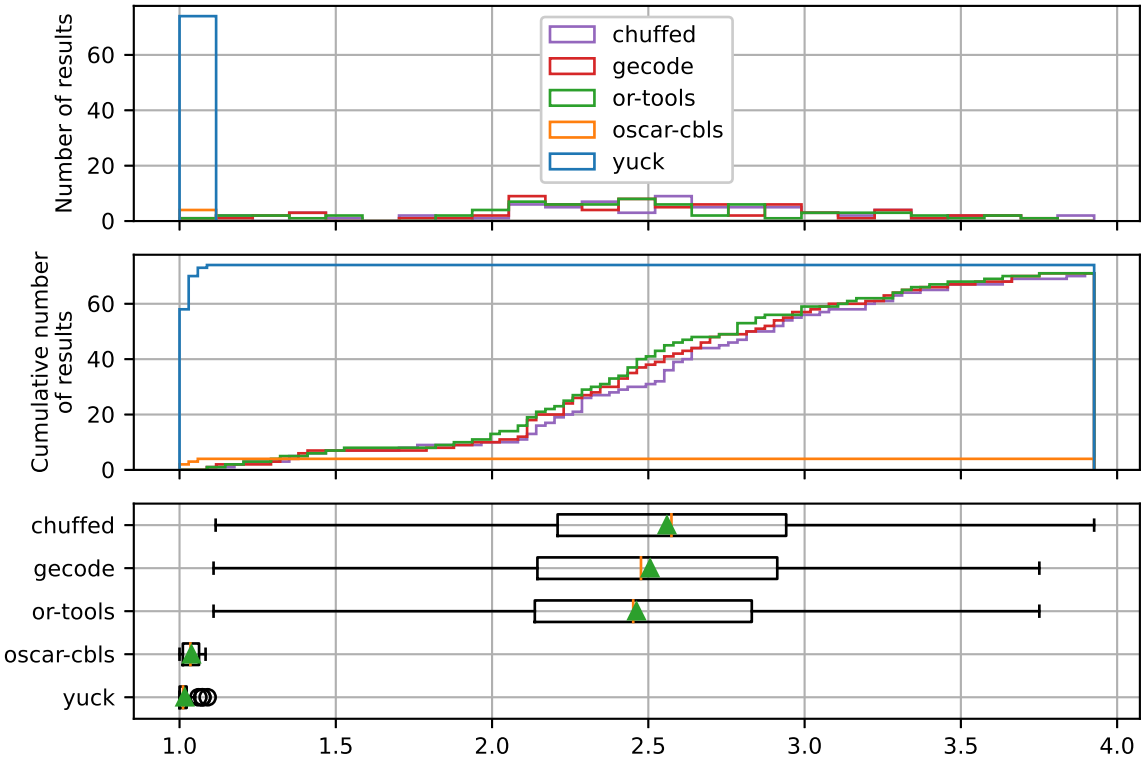

Figure 1: Augerat (CVRP)

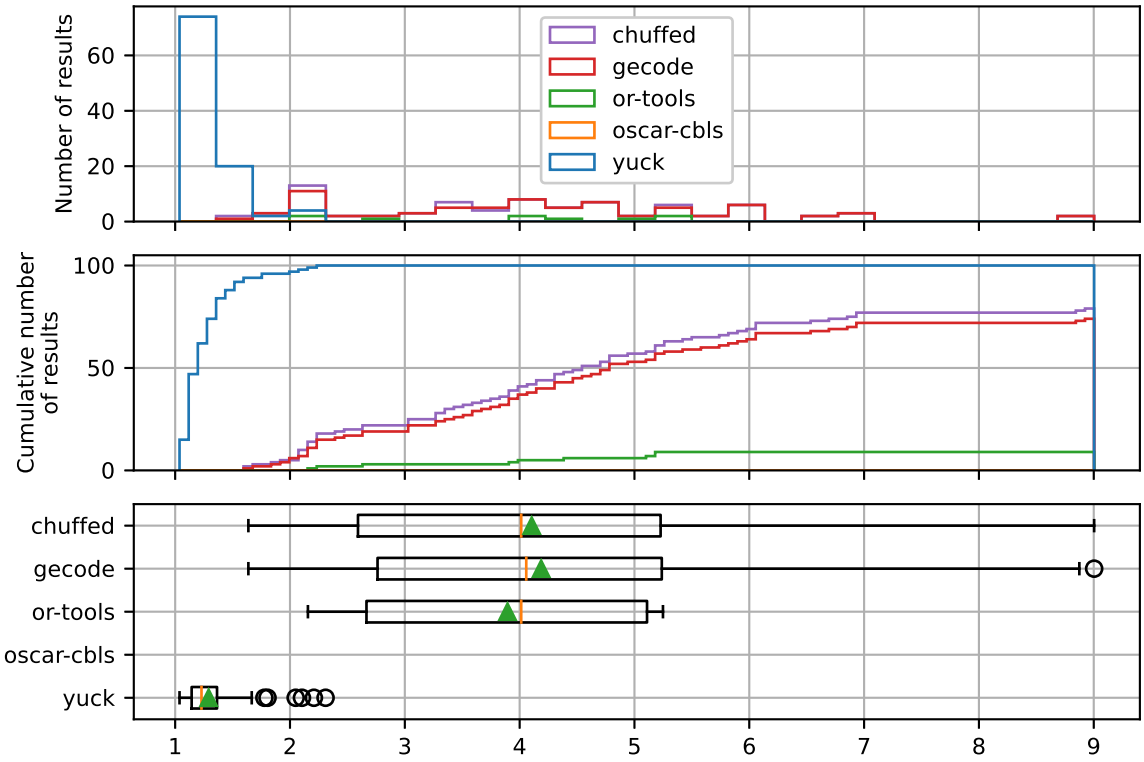

Figure 2: Uchoa (CVRP)

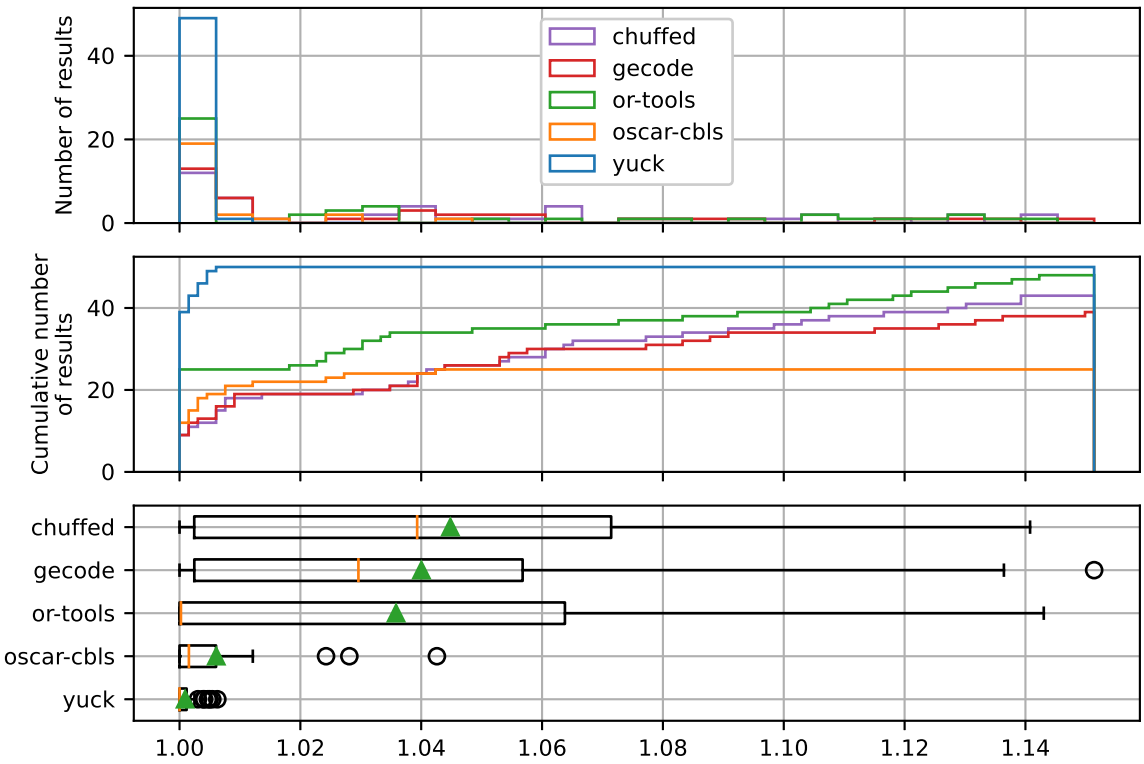

Figure 3: Ascheuer (TSPTW)

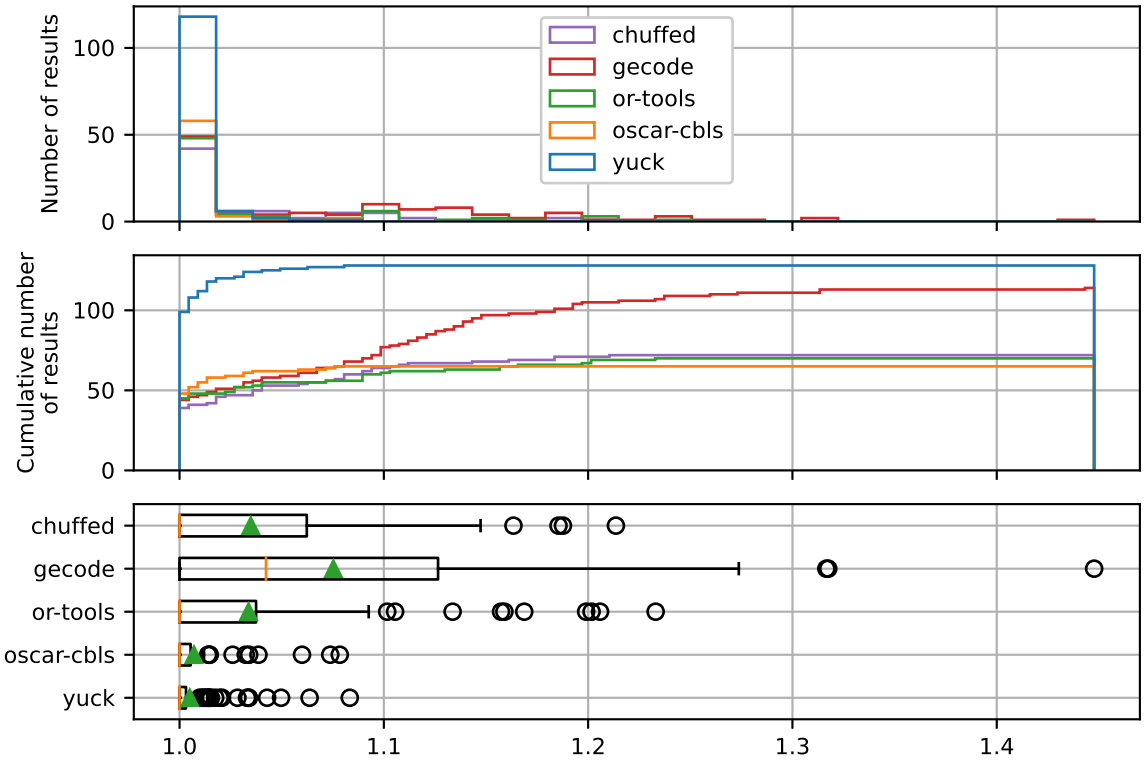

Figure 4: Dumas (TSPTW)

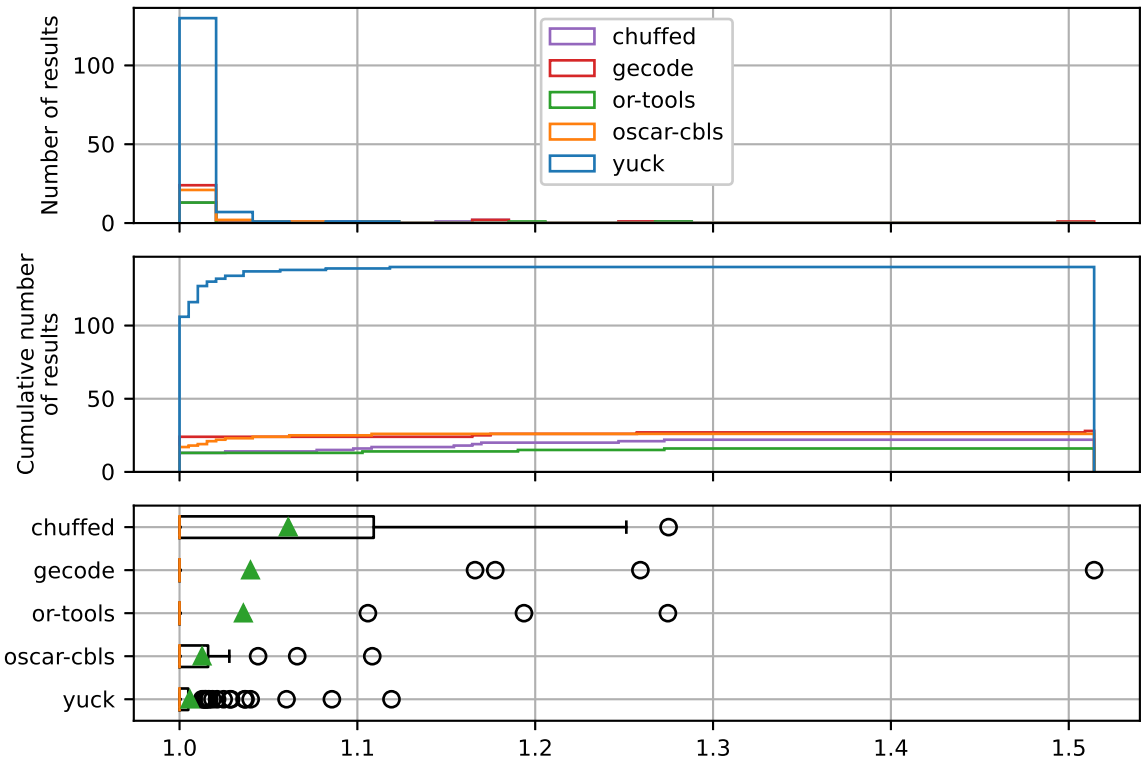

Figure 5: Gendreau (TSPTW)

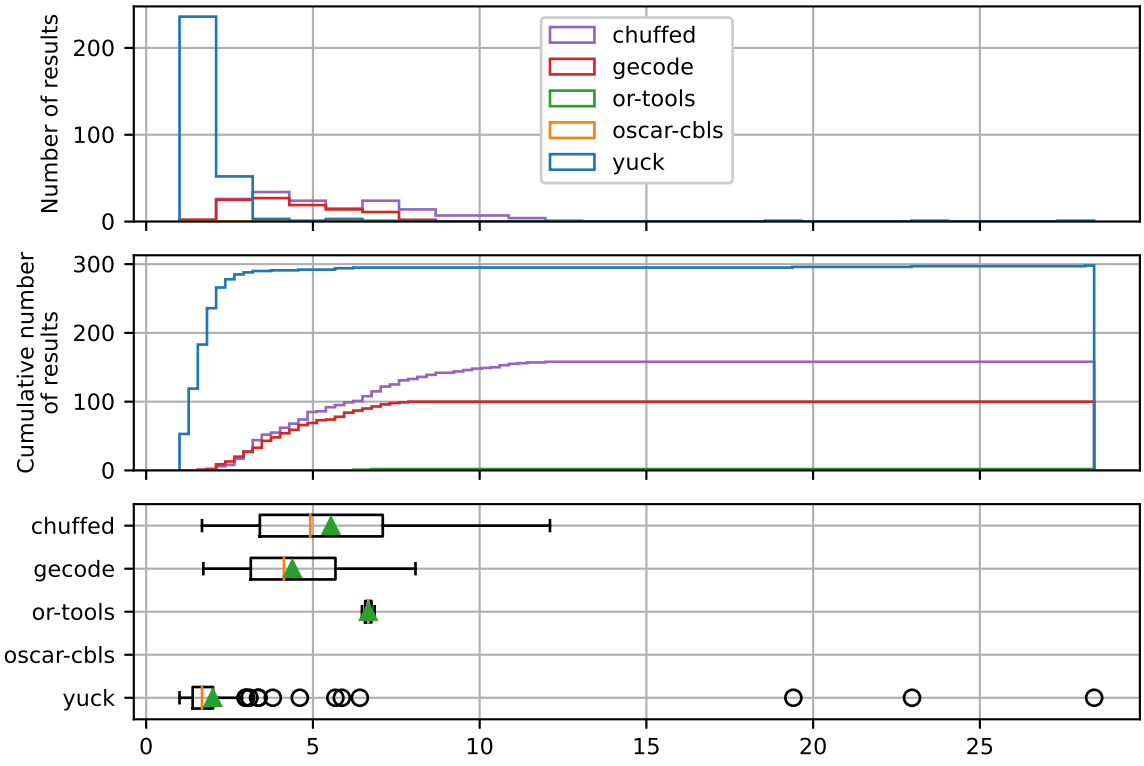

Figure 6: Homberger (CVRPTW)

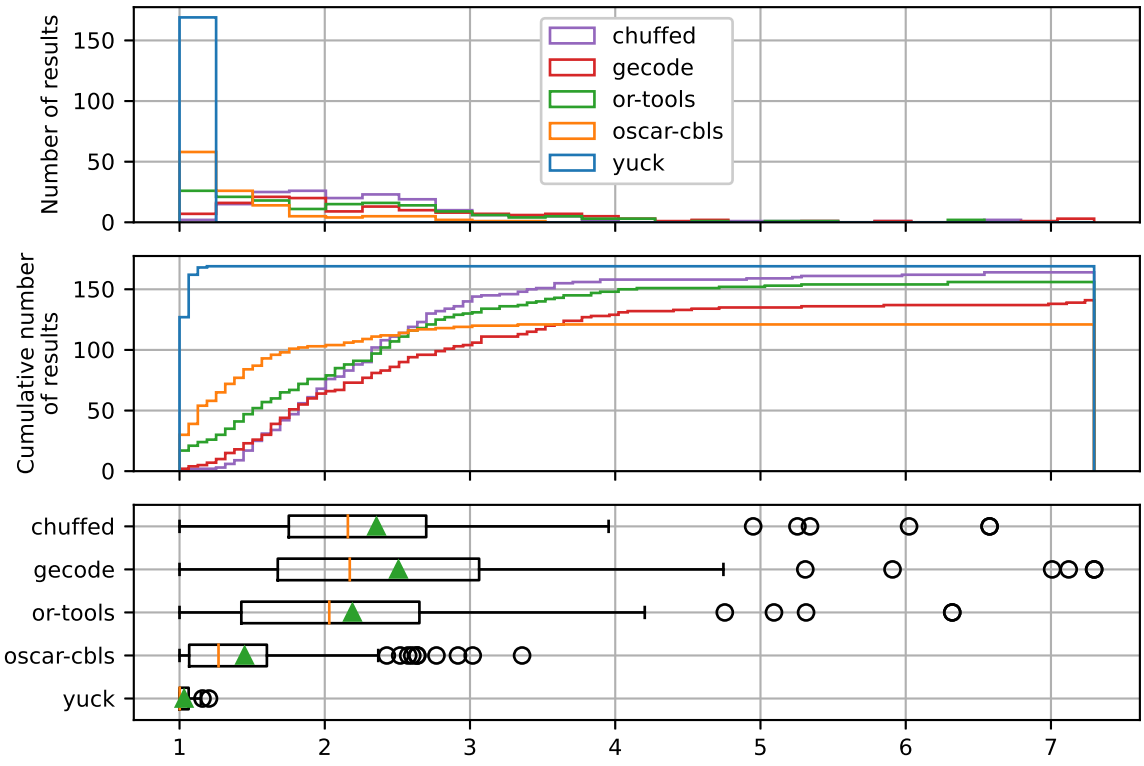

Figure 7: Solomon (CVRPTW)