ATHANOR: High-Level Local Search Over Abstract Constraint Specifications in ESSENCE

Saad Attieh[1*], **Nguyen Dang**[1], **Christopher Jefferson**[1], **Ian Miguel**[1] and **Peter Nightingale**[2]
[1]School of Computer Science, University of St Andrews, UK
[2]Department of Computer Science, University of York, UK
{sa74,nttd,caj21,ijm}@st-andrews.ac.uk, peter.nightingale@york.ac.uk

Abstract

This paper presents ATHANOR, a novel local search solver that operates on abstract constraint specifications of combinatorial problems in the ESSENCE language. It is unique in that it operates directly on the high level, nested types in ESSENCE, such as set of partitions or multiset of sequences, without refining such types into low level representations. This approach has two main advantages. First, the structure present in the high level types allows high quality neighbourhoods for local search to be automatically derived. Second, it allows ATHANOR to scale much better than solvers that operate on the equivalent, but much larger, low-level representations. The paper details how ATHANOR operates, covering incremental evaluation, dynamic unrolling of quantified expressions and neighbourhood construction. A series of case studies show the performance of ATHANOR, benchmarked against several local search solvers on a range of problem classes.

1 Introduction

Local search [Hoos and Stützle, 2004] is a common method for solving combinatorial optimisation problems. Typically, it operates by generating an initial assignment to the variables in a problem and then tries to iteratively change this assignment, through a sequence of *moves* or *neighbourhoods*, in order to improve an objective. This approach trades completeness for the ability to make rapid improvements to the objective, and often finds good solutions more quickly than systematic search procedures. Given a set of neighbourhood moves, a *metaheuristic* is used to select the move to apply at each step of the search. Common metaheuristics include Hill Climbing [Russell and Norvig, 2016, Chapter 4], Simulated Annealing [Kirkpatrick *et al.*, 1983] and Tabu Search [Glover and Laguna, 1998].

We focus herein on general-purpose local search solvers that accept as input a constraint model — a declarative description of a problem consisting of a set of decision variables under a set of constraints. This is a general, flexible approach in contrast to local search procedures written for, and restricted to, individual problems, such as [Merz and Freisleben, 1997; Jaszkiewicz, 2002].

Existing approaches typically accept as input models written in solver-independent modelling languages like MiniZinc [Nethercote *et al.*, 2007]. The recently-proposed Structured Neighbourhood Search (SNS) [Akgün *et al.*, 2018] differs in that it begins from a specification of a problem in the abstract constraint specification language ESSENCE [Frisch *et al.*, 2005; Frisch *et al.*, 2007; Frisch *et al.*, 2008]. ESSENCE allows problems to be described without commitment to low-level modelling decisions through its support for a rich set of abstract type constructors, such as sets, multisets, sequences and relations, each of which can be nested arbitrarily. Fig. 1 presents an example ESSENCE specification of the Capacitated Vehicle Routing Problem (CVRP) [Fisher, 1995], in which orders must be delivered from one depot to a set of locations via a set of vehicles, while respecting the capacity limit of each vehicle. The parameters to the problem are denoted by the `given` statements. The `find` statement introduces a decision variable. Here a *single* highly structured variable (a set of sequences) suffices to capture the problem. The `minimising` and `such that` statements introduce the objective function and problem constraints respectively.

A neighbourhood describes a set of assignments that can be reached from a given assignment. SNS was motivated by the belief that the structure apparent in an abstract specification could be exploited to generate powerful neighbourhoods for local search. SNS does, however, require the refinement of the neighbourhoods it generates into a lower-level representation prior to search.

In this paper we present a local search solver, ATHANOR, which addresses this limitation by operating directly on ESSENCE itself. We propose that directly operating on ESSENCE specifications can provide an increase in both performance and scalability over existing approaches. We test this hypothesis by comparing ATHANOR with state of the art local search solvers. Source code and all data used in this work are publicly available as a github repository [1].

2 Related Work

Constraint-based local search solvers [Hentenryck and Michel, 2009] such as Oscar-CBLS [Björdal *et al.*, 2015] and

[*]Contact Author

[1]https://github.com/athanor

```
given N : int $number of locations
letting L0 be domain int(0..N) $ 0 is the depot
letting L1 be domain int(1..N)
given weights : function (total) L1 --> int(1..)
given costs : function (total) tuple (L0,L0) --> int(0..)
given vehicleCap : int $ Uniform vehicle capacity
letting totalW be sum([weight | (_,weight) <- weights])
letting mV be totalW/vehicleCap + toInt(totalW % vehicleCap != 0) $ Min number of vehicles
find plan : set (minSize mV, maxSize N) of sequence (maxSize N, injective, minSize 1) of L1
minimising sum r in plan . (sum([costs(tuple(r(i-1), r(i))) | i : int(2..N), i<=|r|])
+ costs((0, r(1))) + costs((r(|r|), 0))) $ from depot to first location, and back from last
$ Capacity restriction
such that forAll route in plan . vehicleCapacity >= sum (_,order) in route . weights(order),
$ Every order delivered exactly once:
allDiff([l | r <- plan, (_,l) <- r]), N = sum p in plan . |p|
```

Figure 1: Capacitated vehicle routing in ESSENCE

Yuck [2] analyse the constraints in a problem to derive a set of invariants – for example, the values assigned to a subset of the variables must be all different. They generate neighbourhoods that maintain these invariants. For example, given a satisfied all different constraint, swapping the values of the variables under the all different will not violate the constraint.

Explanation-based [Prud'homme *et al.*, 2014] and propagation-guided [Perron *et al.*, 2004] large neighbourhood search (LNS) are both built upon a standard systematic constraint solver. Given an assignment that forms a feasible solution, the solver creates a neighbourhood by selecting a subset of the variables to be unassigned. The solver then performs a systematic search on these unassigned variables.

A unifying feature of these solvers is that they derive neighbourhoods from a low level constraint model where the only variable types are **int** or **bool**. As ATHANOR constructs neighbourhoods directly from an ESSENCE specification, the derived neighbourhoods are more semantically meaningful. The rich variety of high level types in ESSENCE mean that most problems are described in only one or two ESSENCE variables with much of the problem constraints already enforced by the variables' type invariants. As demonstrated in Section 8, more complex neighbourhoods can be generated from highly structured/nested variable types.

Structured Neighbourhood Search [Akgün *et al.*, 2018] is also based on a standard systematic constraint solver. It generates neighbourhoods from ESSENCE specifications (as we do in this paper) then applies CONJURE and SAVILE ROW to refine them (alongside the model) into the input language of a backtracking constraint solver. SNS uses an adapted version of the MINION solver that applies the neighbourhoods in a similar way to LNS. SNS contrasts with the work described in this paper as ATHANOR directly operates on unrefined abstract ESSENCE variables. This leads to a more scalable approach, especially due to the dynamic creation and deletion of values and constraints (discussed in Section 5).

[2]https://github.com/informarte/yuck

3 Incremental Evaluation

During search, in order to decide whether or not to accept a move, ATHANOR must evaluate if it results in an improved assignment. To evaluate a move efficiently, ATHANOR must update the state of the solver incrementally, avoiding the need to recompute the entire state of the solver.

ATHANOR represents an ESSENCE specification as a pair of abstract syntax trees (ASTs), one representing the constraints in the specification, the other representing the objective function. The leaves of these trees represent the abstract variables in the specification, which are assigned a value from their domain. Hence the constraint AST can be evaluated to a single Boolean value and the objective AST can be evaluated to a single integer. When a variable is reassigned to a new value, all its ancestors (the nodes on the path from the associated leaf back to the root) are reevaluated.

At the start of the search, the solver begins by assigning a random value to each variable and performing a full evaluation of the AST. Afterwards, in order to facilitate incremental evaluation, every node in the AST attaches a trigger to each of its children, which is used to notify the parent of changes to the child nodes that might affect its value. Every type of node (**int**, **set**, etc.) generates a different set of trigger events. All nodes can generate `valueChanged()`, notifying the parent that the value assigned to the child has changed. However, for higher level types such as **set** or **sequence**, giving more exact changes allows for much better incremental evaluation. Therefore, high level types provide more descriptive trigger events. For example, **set** also supports the triggers `valueAdded()`, `valueRemoved()` and `memberValueChange()`. Incremental evaluation is further supported by allowing constraints to only observe trigger events generated by a single element of a container such as a **sequence** or **set**.

Figure 2 gives an example AST state during incremental evaluation, using a snippet of one of the constraints found in the CVRP problem (Figure 1). While **set** variables are unordered in ESSENCE, they are given an arbitrary order in ATHANOR, so we can refer to their elements.

When adding 3 to the 2nd member of `plan`, incremental

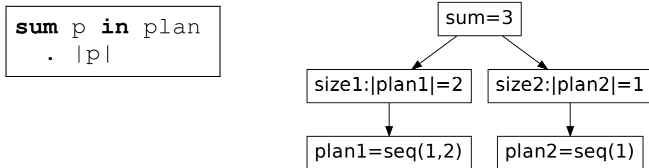

Figure 2: AST evaluation, sum of the sizes of sequences, given plan $= \{seq(1,2), seq(1)\}$

evaluation proceeds as follows:

- 3 is added to `plan2`. Its value is now `seq(1,3)`.
- `valueAdded()` is sent to the parent: `size2`.
- `size2` updates its value to `2`, the event `valueChanged()` is sent to its parent: `sum`.
- `sum` updates its value by subtracting the old value `1` and adding the new value `2` [3].
- The `sum` node now has the value `4`, the event `valueChanged()` is forwarded to its parent.

4 Violation Counts

In Section 3 we described how non-Boolean types are incrementally updated. A similar procedure is used for Boolean expressions, except that Booleans also store a violation count. A violation count is an integer which indicates the magnitude of the change required to satisfy a Boolean expression. Our method for calculating violations is inspired by [Hentenryck and Michel, 2009]. For example, given two integers x and y and the constraint $c(x = y)$, the violation of c is $|x - y|$. A violation count is also attached to each ESSENCE variable in the scope of a constraint. These violation counts are used to indicate to what extent each variable is responsible for constraint violations. This helps to guide the solver when selecting which ESSENCE variable to modify while searching for a feasible solution. In ATHANOR we extend violation counts to support variables with highly nested types, by allowing violation counts to be attached to elements contained in structured variables. Consider the example shown in Figure 3. ATHANOR must consider how violations are attributed when elements in s violate the constraint c. These violation counts guide ATHANOR to the cause of violated constraints.

```
find s : set of τ
such that forAll i in s . c(i)
```

Figure 3: Constraint on a nested type, $c(i)$ is a constraint on i

When a violation is attributed to an element i of a containing structure s such as a **set** or **sequence**, the same violation is added to s and successively the structure that contains s, and so on to the outermost structure. Therefore the violation on any containing structure s is the sum of the violations directly attributed to s and the violation count of the elements of s.

Considering the constraint $|s| = 1$, if this constraint is violated, all the elements in s are equally to blame. Therefore, we do not attribute the violation to the elements in s, but to s as a whole. However, for a constraint like that shown in Figure 3, ATHANOR assigns a violation to only those elements in s that are causing the violation. The set s itself inherits the violation of its elements so that it may be distinguished from other variables; it is natural to consider a set with two violating elements to have a larger violation than a set with one violating element.

5 Dynamic Unrolling of Quantifiers

Although ESSENCE specifications have a fixed number of abstract variables, these variables are usually of container types (e.g. **set**, **sequence** and **partition**). The values of such variables can vary in size as elements are introduced or deleted during search. ESSENCE also allows quantification over such containers, applying a constraint to each of the elements of such types. Pre-generating all possible values and constraints, as is done in low-level local search solvers, is often infeasible. For example in the optimisation variant of the Social Golfers (Figure 4) where the size of `sched` must be maximised, a small instance ($g = 8, s = 8$) permits approximately 4.5×10^{47} possible partitions. Low level solvers can neither support variables with such large domains, nor allow quantification over sets which can grow to such a large size. In contrast, ATHANOR performs no such pre-generation of values or constraints. Instead, the solver dynamically adds and deletes elements of structured variables along with the constraints on such elements during search, allowing ATHANOR to scale to much larger domains.

We illustrate ATHANOR's dynamic unrolling through an example shown in Figure 5. In the AST representation, we have a `forAll` node, representing the quantification. The `forAll` node has one child for each item in s. It also stores the expression (`i%2=0`) applied to each element of s. This expression is a template, meaning that it is represented by an incomplete AST. The AST is incomplete as i in the expression is a placeholder. We call this an iterator. When a new element is added to s, a child is added to the `forAll` node, the expression template is copied into this new child and the placeholder is replaced with the newly added element. The AST subtree representing the copied expression is then evaluated in a similar fashion to the evaluation of the entire AST at the start of the search. The nodes in the subtree then begin triggering on their children as per Section 3.

6 Neighbourhood Construction

The method by which ATHANOR derives neighbourhoods from an ESSENCE specification is inspired by the neighbourhood generation rules used by Structured Neighbourhood Search [Akgün *et al.*, 2018]. ATHANOR makes use of a set of neighbourhood templates. Each template contains a set of criteria which is matched against every variable type present in an ESSENCE specification. If the type and domain

[3] The sum node caches the values of its children.

```
given w, g, s : int(1..)
letting Golfers be new type of size g * s
find sched : set (size w) of partition (regular, numParts g, partSize s) from Golfers
such that forAll g1, g2 : Golfers, g1 != g2 .
            (sum week in sched . toInt(together({g1, g2}, week))) <= 1
```

Figure 4: Social Golfers in ESSENCE

```
find s : set of int(1..5)
such that forAll i in s. i % 2 = 0
```

Figure 5: Quantifying over a set

of a variable satisfy the criteria for a neighbourhood template, the template is instantiated into a neighbourhood that is used to alter the value of the variable. Since each neighbourhood is directly linked to a variable type and domain, ATHANOR's neighbourhoods are able to preserve type invariants – for example, the uniqueness of elements in a set. We argue that this allows for neighbourhoods that are more semantically meaningful, reducing the time spent on finding assignments to highly structured variables (`set of partition (regular, numParts 3)`) and instead focusing on satisfying problem constraints or improving the objective.

ATHANOR's neighbourhoods are divided into three categories, *direct*, *synchronised* and *higher-order*. Direct neighbourhoods only examine the outer structure of a type. Every supported domain has at least one direct neighbourhood which simply generates a random value from that domain. While assigning a random value is sufficient for integer and boolean variables, we also want neighbourhoods that can make more incremental changes to higher level types. For example, given a **set** type, neighbourhoods such as `setAdd` (add a value), `setRemove` (remove a value) and `setSwap` (swap one value for another) are generated. Similar neighbourhoods are also produced for the ESSENCE type **sequence**. However, since the sequence type encodes an order of elements, neighbourhoods that focus on changing the element order are also generated such as `sequenceReverseSub` (reverse a sub-sequence) or `sequencePositionsSwap` (swap positions of elements).

Synchronised neighbourhoods operate on multiple variables simultaneously. For example, `setMove` (moving an element from one set to another) or `sequenceCrossOver` exchanging elements between two sequences. *Higher-order* neighbourhoods select one or more elements of type τ from a **set**, a **sequence** or any container type c and apply any of the neighbourhoods for τ on the selected elements of c. For example, given the type `set of sequence of ...`, a *higher-order* neighbourhood can select one sequence from the set and apply the `sequenceReverseSub` neighbourhood on that sequence. *Higher-order* neighbourhoods are particularly useful when combined with *synchronised* neighbourhoods, where a *higher-order* neighbourhood selects two elements from a container and the *synchronised* neighbourhood causes the elements to interact. For example, given a type `multiset of set of ...`, a *higher-order* neighbourhood may select two sets from the multi-set and then use the *synchronised* neighbourhood `setMove` to move an element from one set to the other. This means that given a type τ_1 `of` τ_2 `of ... of` τ_n, the combination of *higher-order*, *direct* and *synchronised* neighbourhoods allow every level of a variable's values to be manipulated.

Finally, ATHANOR's neighbourhood templates also detect attributes used in the type constructors to further tailor the neighbourhoods. For example, ATHANOR will not generate neighbourhoods which change the cardinality of a **multiset** with a fixed size attribute. Similarly, given a **sequence** with the **injective** attribute, ATHANOR will ensure that the generated neighbourhoods always maintain that the elements of the sequence are distinct.

Examples of the high performing neighbourhoods generated by ATHANOR in our experiments are given in Section 8.1 and Section 8.2.

7 Search

Once a set of neighbourhoods has been automatically constructed, the search algorithm of ATHANOR uses standard local search techniques. Neighbourhoods are treated as black boxes, each of which has the potential to improve on either the violation or the objective. A regret minimisation multi-armed bandit [Auer *et al.*, 2002] is used to track which neighbourhoods are successful, biasing the search towards neighbourhoods which have previously improved the assignment.

There is a single global variable assignment which is accessible by all parts of the ATHANOR solver, and five functions which access this global assignment:

- OBJECTIVEG(): The objective value of the global assignment (smaller is better).
- VIOLATIONG(): The violation of the global assignment.
- SETRANDOMASSIGNMENT() : Set the global assignment to a random value.
- MABAPPLYNEIGHBOURHOOD(Q) : Using a MAB (Multi-armed Bandit), choose a neighbourhood to apply. Then, use the violation counts to choose where to apply that neighbourhood. If the violation counts are all zero, choose randomly with even probability. Q switches between two different MABs, one for finding a non-violating assignment, and one for finding an optimal solution.
- UNDOLASTNEIGHBOURHOOD() : Undo the last call to MABAPPLYNEIGHBOURHOOD. This also marks this choice of neighbourhood as “failed” for the MAB.

Algorithm 1 presents the entire search procedure used in the current version of ATHANOR. Search begins with the procedure ATHANOR, which first tries to find a solution that violates no hard constraints, and then from this solution searches for other solutions which improve the optimisation function.

The RUN procedure contains the main search loop of ATHANOR. RUN alternates between hill climbing (CLIMB) and exploration by random walk (RANDOMWALK). If climbing fails to improve the solution we increase the length of the random walk (n_r) by Z (1.3 in ATHANOR), and if the length of the random walk gets longer than a constant L (500 in ATHANOR), we reset the length of the random walk and the current target objective β. Once a solution with no violations is found, no violations are permitted during either hill climbing or the random walk phase. We leave for future works the investigation of using more advanced search strategies, and the tuning of the constants used in ATHANOR's algorithm.

Algorithm 1 Search algorithm of ATHANOR

```
procedure ATHANOR
    SETRANDOMASSIGNMENT( )
    RUN(VIOLATIONG)            ▷ Move to feasible solution
    RUN(OBJECTIVEG)            ▷ optimise objective
procedure RUN(Q: function)
    I ← 10, Z ← 1.3, L ← 500   ▷ algorithm constants
    n_r ← I                    ▷ number of random moves to take
    β ← Q()                    ▷ Quality to beat
    while time limit not reached do
        CLIMB(Q)
        if Q() = 0 and Q = VIOLATIONG then
            return             ▷ found feasible solution
        if Q() < β then        ▷ found a better assignment
            β ← Q(), n_r ← I
        else
            n_r ← n_r × Z
        RANDOMWALK(Q,n_r)
        if n_r > L then        ▷ Reset random walk length and β
            β ← Q(), n_r ← I
procedure CLIMB(Q)
    o ← OBJECTIVEG( ), v ← VIOLATIONG( )
    i ← 0                      ▷ iterations spent without improving
    while i ≤ limit do         ▷ limit is a tunable parameter
        MABAPPLYNEIGHBOURHOOD(Q)
        o_2 ← OBJECTIVEG( ), v_2 ← VIOLATIONG( )
        if (v ≠ 0 ∧ v_2 ≤ v) ∨ (v = 0 ∧ o_2 ≤ o) then
            (o, v) ← (o_2, v_2)    ▷ New solution accepted
            i ← 0
        else
            i ← i + 1
            UNDOLASTNEIGHBOURHOOD( )
procedure RANDOMWALK(Q,r)
    i ← 0
    while i ≤ r do
        MABAPPLYNEIGHBOURHOOD(Q)
        if Q = OBJECTIVEG ∧ violation > 0 then
                               ▷ No violations if improving objective
            UNDOLASTNEIGHBOURHOOD( )
        else
            i ← i + 1
```

8 Case Studies

Our hypothesis is that ATHANOR derives its performance from the presence of high level (nested structured) variables in a specification, as they lead to more complex neighbourhoods being generated. We have therefore split problem classes into two categories. *Structured* problems are those that contain variables with a nested structure, for example **set** of **sequence** of **int**. *Unstructured* problems are those which have little nesting of types – for example, **function int** --> **int**.

We compare the performance of ATHANOR against six other solvers. Three of them are constraint-based local searches, including Oscar-CBLS [Björdal *et al.*, 2015], Yuck, and Structured Neighbourhood Search (SNS) [Akgün *et al.*, 2018]. We also compare against both explanation-based (LNSEB) [Prud'homme *et al.*, 2014] and propagation-guided (LNSPG) [Perron *et al.*, 2004] large neighbourhood search in Choco 4.0.9. Finally, Chuffed [4] (a systematic constraint solver) was used as a performance baseline for the local search solvers.

The refined models were generated by CONJURE [Akgün *et al.*, 2011; Akgün *et al.*, 2013] – an automated modelling tool for ESSENCE– and were further tailored for the target solvers by SAVILE ROW [Nightingale *et al.*, 2014; Nightingale *et al.*, 2017; Nightingale *et al.*, 2015]. The MiniZinc backend of SAVILE ROW along with MiniZinc 2.1.7 [Nethercote *et al.*, 2007] were used to produce FlatZinc inputs for Chuffed, Oscar-CBLS and Yuck. Models were hand crafted for LNS. We ensure that they were as close as possible to the models given to the other solvers. Unlike ATHANOR, modelling for local search or systematic solvers involves a choice of whether or not to include symmetry-breaking constraints. Both symmetry-broken and non-symmetry-broken models were given to all solvers[5] and the best performing model for each problem is reported.

The instances used in this work come from popular benchmarking datasets whenever possible, and are randomly generated otherwise. The instances for TSP, CVRP, Knapsack and Social Golfers are taken from TSPLIB [Reinelt, 1995], VRP-REP [Mendoza *et al.*, 2014], Pisinger's hard knapsack [Pisinger, 2005], and CSPLib [Miguel *et al.*, 2000], respectively. For Sonet and MEB, a parameterised instance generator is manually created for each problem, and non-trivial instances are generated using the automatic algorithm configuration tool irace [López-Ibáñez *et al.*, 2016] to search in the parameter space of the generator. The number of instances ranges from 30 to 60 per problem class. All instances or reference links to them are available in the github repository of ATHANOR[6].

For each problem class, all solvers were run ten times (ex-

[4] https://github.com/chuffed/chuffed

[5] Oscar-CBLS had issues running some of the symmetry broken models. On these problems we used the no symmetry model.

[6] https://github.com/athanor

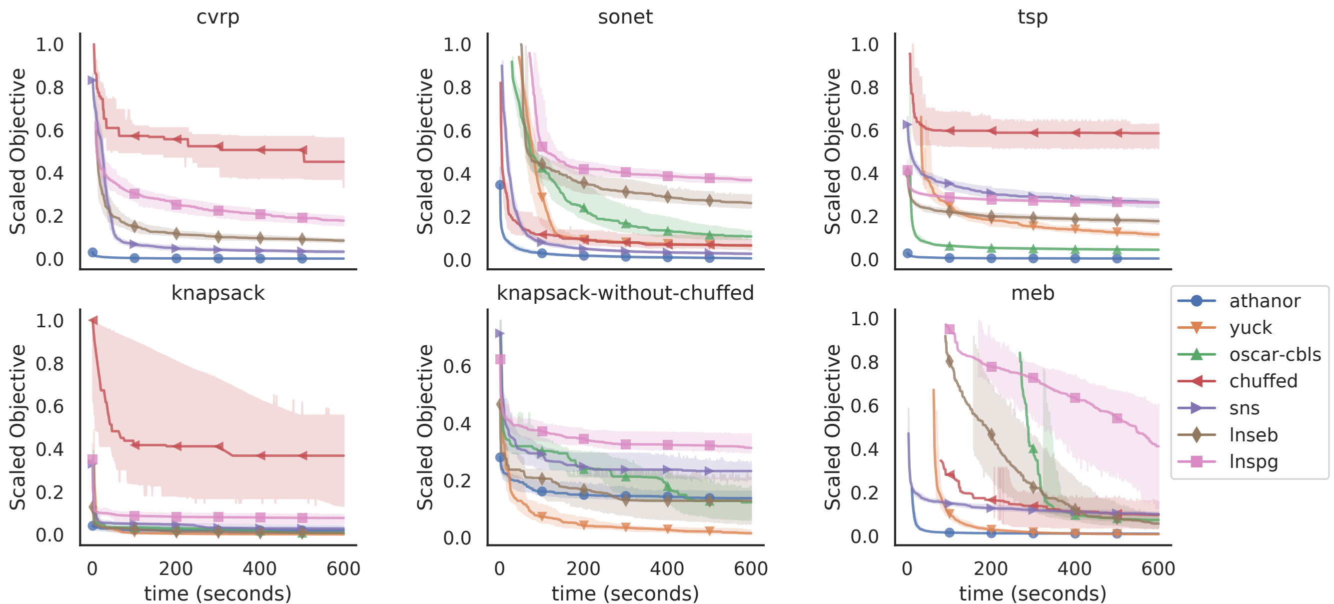

Figure 6: Normalised objective value of the best version (with/without symmetry breaking) of each solver on five optimisation problems: CVRP with `set (maxSize ...) of sequence (injective, ...) of int`, SONET with `set (maxSize ...) of set of int`, TSP with `sequence of int`, Knapsack with `set of int` and MEB with `function int --> int`.

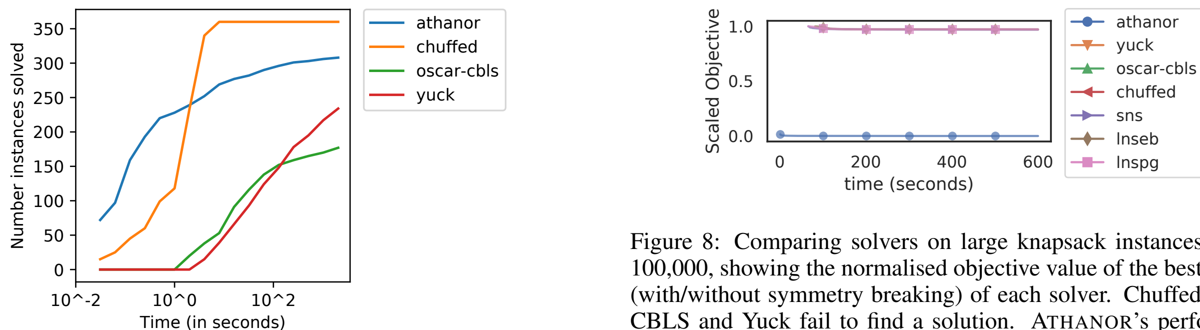

Figure 7: The Social Golfers Problem. The cumulative number of instances solved by each solver in less than the corresponding time point, showing the best of with/without symmetry breaking for solvers other than ATHANOR. LNS and SNS are omitted as they do not support satisfaction problems.

Figure 8: Comparing solvers on large knapsack instances of size 100,000, showing the normalised objective value of the best version (with/without symmetry breaking) of each solver. Chuffed, Oscar-CBLS and Yuck fail to find a solution. ATHANOR's performance stands out from all the rest, making the lines for other solvers (SNS and LNS) almost coincide on this plot.

cept Chuffed, as it is deterministic). For optimisation problems (CVRP, Sonet, MEB, TSP, and CVRP) the best objective found by each solver is logged every second. For satisfaction problems (Social Golfers), the time to reach the first solution is recorded. For each problem class p, results are aggregated and shown in Figures 6 and 7 as follows:

Optimisation problems: for each instance of the problem class, the objective found at each second is rescaled to a range $[0, 1]$, where 0 and 1 are the minimum and maximum objectives (respectively) found across all solvers throughout all runs on the instance. This allows results to be aggregated across multiple instances of the same problem class. The rescaling is necessary as the sizes of instances tested, and hence the objectives found, cover a significant range. For each class, a graph shows the median (rescaled) objective found by every solver at every second, with ribbons marking the 95% confidence interval.

Satisfaction problems: we show the number of runs where each solver found a solution in the given time.

8.1 Structured Problems

CVRP [Fisher, 1995] and Sonet (CSPLib 56) are highly structured optimisation problems. ATHANOR derives several interesting neighbourhoods from the type constructors in these problems. For CVRP, ATHANOR generates ten neighbourhoods from the `set of sequence` type. The best-performing neighbourhoods, which are listed below, are popularly used by CVRP-specific local search solvers:

- select one sequence and reverse a contiguous portion, which reflects the famous two-opt move [Croes, 1958];
- select a sequence and swap two locations, which gives

the solver the flexibility to optimise the route of a particular vehicle;

- select two sequences and move a location from one to the other, which helps to move costly locations from one vehicle's route to another;
- select two sequences and exchange locations between the two sequences.

For Sonet, ATHANOR understands from the type information available (**set** of **set** of **int**) that unlike CVRP, the order of elements is not relevant to the problem. Therefore, the neighbourhoods produced focus on adding or removing elements from the inner sets, exchanging elements between the inner sets or adding or removing sets from the outer set. Since the objective of the problem is to minimise the sum of the sizes of each inner set, it is not surprising that the best performing neighbourhood is about selecting a set and removing an integer from that set.

ATHANOR outperforms all other solvers throughout the 600 seconds (see Figure 6) in both CVRP and Sonet. Note that in both CVRP and Sonet, a limit to the size of the outer set is necessary to model the problem for input to the low level solvers. Without this size limit, the outer sets would be impractically large for the low level solvers. Such solvers must use enough variables to represent all possible assignments to the outer set at the start of the search. ATHANOR does not have such limitations.

Figure 7 shows the relative performance of ATHANOR in solving the Social Golfers problem, a structured satisfaction problem. ATHANOR uses the `regular` and `numParts` attributes to deduce that neighbourhoods that add a cell, remove a cell or change the sizes of cells should not be generated. A partition and its cells are unordered, hence it makes no sense to reorder cells or elements within a single cell. Therefore, this leaves one neighbourhood that maintains the invariants of the variable type: swap elements between two cells of a partition. Once again, the type constructor is able to convey the structure inherent in the Social Golfers problem – using information that is not readily available to the low level solvers. While ATHANOR significantly outperforms the other local search solvers, Chuffed outperforms all local search solvers.

8.2 Unstructured Problems

We also experimented on TSP [Reinelt, 1995], Knapsack [Pisinger, 2005] and MEB (CSPLib 48) problems, such problems have less structure than the CVRP and Sonet problems. The Knapsack problem contains a single **set** of **int** variable. Given the relatively simple structure, it is not surprising that ATHANOR does not clearly outperform the other solvers. However, the set does have a variable cardinality [0 . . . number of knapsack items]. This is important as the instances used in the graphs shown were of size 5,000. Another set of Knapsack instances of size 100,000 were generated using the generator provided by [Pisinger, 2005], and all solvers are tested on them. As can be seen from fig. 8, the other solvers struggle to even find an initial solution to the larger instances. After 30 seconds SNS and LNS find feasible solutions for less than a quarter of the instances while Yuck, Chuffed and Oscar fail to find any solutions. Once the solvers have found feasible solutions, they make very little progress throughout the remaining time. Meanwhile, in 0.01 seconds, ATHANOR finds a feasible solution on all instances and makes steady progress throughout the entire 600 seconds. This clearly shows the benefit of ATHANOR's dynamic value and constraint construction.

The TSP is modelled using a **sequence** (**injective**) of locations. The refinement of this specification produces a matrix of integers with an `allDifferent` constraint. The `allDifferent` constraint allows low level local search solvers to also derive that the locations must be distinct. Hence, both high level and low level solvers generate neighbourhoods that maintain the distinctness of the elements. The presence of a **sequence** variable type is still advantageous as it strongly conveys the importance of the order of elements. The experiments show ATHANOR outperforming the other solvers.

ATHANOR performs competitively on the MEB problem, producing the highest quality solution until 250 seconds when Yuck overtakes. ATHANOR cannot derive any additional information from the type constructor **function** (**total**) **int** --> **int**. This reinforces our hypothesis that ATHANOR derives its performance from the highly structured types and its scalability from the variable-sized types.

9 Conclusion

We have presented the benefits of ATHANOR, a local search solver which operates directly on ESSENCE specifications. ATHANOR uses the types of ESSENCE to construct semantically meaningful and type preserving neighbourhoods. Furthermore, ATHANOR instantiates abstract types directly without having to resort to refinements into primitive types. We have presented a framework for incremental evaluation of ESSENCE ASTs whose values change during search. This includes the dynamic construction and deletion of values and constraints during search. Our experiments show both that the high quality neighbourhoods generated by ATHANOR perform well on a range of structured problems and that ATHANOR scales to very large instances.

Acknowledgements

This works is funded by the EPSRC grants EP/P015638/1 and EP/P026842/1. Christopher Jefferson is supported by a Royal Society University Research Fellowship. This work used the Cirrus UK National Tier-2 HPC Service at EPCC (http://www.cirrus.ac.uk) funded by the University of Edinburgh and EPSRC (EP/P020267/1).

References

[Akgün *et al.*, 2011] Özgür Akgün, Ian Miguel, Chris Jefferson, Alan M. Frisch, and Brahim Hnich. Extensible automated constraint modelling. In *AAAI*, pages 4–11, 2011.

[Akgün *et al.*, 2013] Özgür Akgün, Alan M. Frisch, Ian P. Gent, Bilal S. Hussain, Chris Jefferson, Lars Kotthoff, Ian Miguel, and Peter Nightingale. Automated symmetry breaking and model selection in Conjure. In *CP*, pages 107–116, 2013.

[Akgün *et al.*, 2018] Özgür Akgün, Ian P Gent, Chris Jefferson, Ian Miguel, Peter Nightingale, Andras Salamon, and Patrick Spracklen. A framework for constraint based local search using Essence. In *IJCAI*, pages 1242–1248, 2018.

[Auer *et al.*, 2002] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2):235–256, May 2002.

[Björdal *et al.*, 2015] Gustav Björdal, Jean-Noël Monette, Pierre Flener, and Justin Pearson. A constraint-based local search backend for MiniZinc. *Constraints*, 20(3):325–345, 2015.

[Croes, 1958] Georges A Croes. A method for solving traveling-salesman problems. *Operations research*, 6(6):791–812, 1958.

[Fisher, 1995] Marshall Fisher. Vehicle routing. *Handbooks in operations research and management science*, 8:1–33, 1995.

[Frisch *et al.*, 2005] Alan M. Frisch, Matthew Grum, Chris Jefferson, Bernadette M. Hernández, and Ian Miguel. The Essence of Essence. *Modelling and Reformulating Constraint Satisfaction Problems*, pages 73–88, 2005.

[Frisch *et al.*, 2007] Alan M. Frisch, Matthew Grum, Chris Jefferson, Bernadette M. Hernández, and Ian Miguel. The design of Essence: A constraint language for specifying combinatorial problems. In *IJCAI*, pages 80–87, 2007.

[Frisch *et al.*, 2008] Alan M Frisch, Warwick Harvey, Chris Jefferson, Bernadette Martínez-Hernández, and Ian Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints*, 13(3):268–306, 2008.

[Glover and Laguna, 1998] Fred Glover and Manuel Laguna. Tabu search. In *Handbook of combinatorial optimization*, pages 2093–2229. Springer, 1998.

[Hentenryck and Michel, 2009] Pascal Van Hentenryck and Laurent Michel. *Constraint-based local search*. The MIT press, 2009.

[Hoos and Stützle, 2004] Holger H. Hoos and Thomas Stützle. *Stochastic local search: Foundations & applications*. Elsevier, 2004.

[Jaszkiewicz, 2002] Andrzej Jaszkiewicz. On the performance of multiple-objective genetic local search on the 0/1 knapsack problem-a comparative experiment. *IEEE Transactions on Evolutionary Computation*, 6(4):402–412, 2002.

[Kirkpatrick *et al.*, 1983] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[López-Ibáñez *et al.*, 2016] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58, 2016.

[Mendoza *et al.*, 2014] Jorge E Mendoza, C Guéret, M Hoskins, H Lobit, V Pillac, T Vidal, and D Vigo. Vrp-rep: the vehicle routing community repository. In *Third Meeting of the EURO Working Group on Vehicle Routing and Logistics Optimization (VeRoLog). Oslo, Norway*, 2014.

[Merz and Freisleben, 1997] Peter Merz and Bernd Freisleben. Genetic local search for the tsp: New results. In *Proceedings of 1997 Ieee International Conference on Evolutionary Computation (Icec'97)*, pages 159–164. IEEE, 1997.

[Miguel *et al.*, 2000] Ian Miguel, Bram Hnich, Ian Gent, Ian Walsh, Christopher Jefferson, and Özgür Akgün. CSPLib: A problem library for constraints, 2000. Available from `http://http://www.csplib.org/`.

[Nethercote *et al.*, 2007] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling language. In *CP*, LNCS 4741, pages 529–543. Springer, 2007.

[Nightingale *et al.*, 2014] Peter Nightingale, Özgür Akgün, Ian P. Gent, Chris Jefferson, and Ian Miguel. Automatically improving constraint models in Savile Row through associative-commutative common subexpression elimination. In *CP*, LNCS 8656, pages 590–605. Springer, 2014.

[Nightingale *et al.*, 2015] Peter Nightingale, Patrick Spracklen, and Ian Miguel. Automatically improving SAT encoding of constraint problems through common subexpression elimination in Savile Row. In *CP*, LNCS 9255, pages 330–340. Springer, 2015.

[Nightingale *et al.*, 2017] Peter Nightingale, Özgür Akgün, Ian P. Gent, Christopher Jefferson, Ian Miguel, and Patrick Spracklen. Automatically improving constraint models in Savile Row. *Artificial Intelligence*, 251:35–61, 2017.

[Perron *et al.*, 2004] Laurent Perron, Paul Shaw, and Vincent Furnon. Propagation guided large neighborhood search. In *CP*, LNCS 3258, pages 468–481. Springer, 2004.

[Pisinger, 2005] David Pisinger. Where are the hard knapsack problems? *Computers & Operations Research*, 32(9):2271–2284, 2005.

[Prud'homme *et al.*, 2014] Charles Prud'homme, Xavier Lorca, and Narendra Jussien. Explanation-based large neighborhood search. *Constraints*, 19(4):339–379, 2014.

[Reinelt, 1995] Gerhard Reinelt. Tsplib95. *Interdisziplinäres Zentrum für Wissenschaftliches Rechnen (IWR), Heidelberg*, 338, 1995.

[Russell and Norvig, 2016] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach, Global Edition, Third Edition*. Pearson, 2016.